

# PATCHING – A FRAMEWORK FOR ADAPTING IMMUTABLE CLASSIFIERS TO EVOLVING DOMAINS

SEBASTIAN KAUSCHKE

Dissertation zur Erlangung des  
akademischen Grades *Doktor-Ingenieur* (Dr.-Ing.)

Dissertationsschrift in englischer Sprache von  
Dipl. Inform. Sebastian Kauschke  
aus Darmstadt, Deutschland  
geb. am 30.05.1983 in Erbach

Erstreferent: Prof. Dr. Johannes Fürnkranz  
Korreferent: Prof. Dr. Max Mühlhäuser  
Korreferentin: Prof. Dr. Barbara Hammer

Tag der Einreichung: 12.07.2019  
Tag der Prüfung: 06.09.2019



Knowledge Engineering Group  
Technische Universität Darmstadt  
Darmstadt, 2019

16.09.2019 – Version 1.1

Sebastian Kauschke: *Patching – A Framework for Adapting Immutable Classifiers to Evolving Domains.*

Darmstadt, Technische Universität Darmstadt  
Jahr der Veröffentlichung auf TUPrints: 2019  
URN: urn:nbn:de:tuda-tuprints-90890  
Tag der mündlichen Prüfung: 06.09.2019

Veröffentlicht unter CC BY-SA 4.0 International  
<https://creativecommons.org/licenses>

© 16.09.2019

# Abstract

---

Machine learning models are subject to changing circumstances, and will degrade over time. Nowadays, data are collected in vast amounts: Personal data is retrieved by our phones, by our internet browser, via our shopping behavior, and especially through all the content that we upload to social media platforms. Machines in factories, cars, essentially every device that is not purely mechanical anymore, may be also collecting data. This data is often used to build predictive models, e. g., for recommender systems or remaining lifetime estimation. As all things in life, the data and the knowledge extracted from a person or machine is subject to change, which is called concept drift. This concept drift may be caused by varying circumstances, changes in the expected outcome, or completely new requirements for the task. In any case, to keep a model operative, adaptive learning mechanisms are required to deal with the drift.

Related works in this area cover a plethora of adaptive learning mechanisms. Usually, these algorithms are made to learn on streams of data from scratch. However, we argue that in many real-world scenarios this type of learning does not fit the actual application. It is rather, that stationary models are trained in a sandbox environment on large datasets, which are then put into practical use. If these models are not specifically constructed to be adaptive, any concept drift will lower the performance. Since training such a model, e. g., a deep neural network, can be expensive in regards of cost and time required, it is desirable to use it as long as possible.

We introduce a new paradigm of adapting existing models. Our goal is to keep the existing models as long as possible, and only adapt it to the concept drift where it is necessary. We solve this by computing partial adaptations, so called patches. Via this mechanism, we can assure the existing model to live longer, and keep the learning required for adaptation to a minimum. The *Patching* mechanism elongates the lifetime of a machine learned model, helps to adapt with fewer observed instances, aids in individualizing an existing model, and generally increases the models' cost efficiency.

In this dissertation we first introduce a general framework for learning patches as adaptation mechanisms. We evaluate the concept, and compare it against state of the art stream learning mechanisms. When dealing with normal stream scenarios, it is reasonable to apply *Patching*. However, when dealing with scenarios which it is intended for, *Patching* excels in adaptation speed and overall performance.

In a second contribution we specialize the patching idea on neural networks. Since neural networks are expensive and time consuming in training, we require a way of adapting them quickly. Although neural networks can be adapted via the normal training process, training

them with newer data can lead to side effects such as catastrophic forgetting. Depending on the size and complexity of the network, adapting them can also be either expensive or—when given only few examples—unsuccessful. We propose neural network patching (*NN-Patching*) as a solution to this issue. In *NN-Patching*, the underlying network remains unchanged. However, a neural patch is trained by using the inner activations of the base network. These represent latent features that can be useful towards the given task. An error estimator network determines, whether the patch network or the base network is better suited to classify an instance. *NN-Patching* shows even more significant improvements than *Patching*, with quick adaptation and overall adaptive capabilities that rival those of the theoretically more capable competition.

The final contribution is geared towards the use in scenarios that require model individualization or deal with re-occurring concepts. For this task we propose *Ensemble Patching*, a variant of *Patching* that builds an ensemble of patches. These patches are learned in such a way, that they each cover a distinctive type of concept drift. When a new concept emerges, a certain error pattern will occur for the base classifier. A specific patch is then learned. All ensemble members are managed via a recurrent network called the ensemble conductor. This separately trained model will conduct the ensemble decision, and is the key player for the adaptation. When concepts become outdated, the conductor will put less weight on the decisions of the respective patches, but by its structure it can quickly reactivate them, should older concepts become relevant again. Our evaluation demonstrates that this ensemble technique handles recurring concepts very well. *Ensemble Patching* can also be employed in a stream classification scenario, where computational efficiency is important.

## Zusammenfassung

---

Durch maschinelles Lernen erstellte Modelle sind sich ändernden Bedingungen unterworfen, was auf Dauer zu einer Leistungsverschlechterung führen kann. Heutzutage werden in allen Lebenslagen Daten über uns gesammelt: Sei es über unsere Smartphones, von unserem Internet Browser, durch unser Einkaufsverhalten, und natürlich durch all den Inhalt, den wir in sozialen Netzwerken teilen und hochladen. Maschinen in Fabriken, Autos, grundsätzlich jedes Gerät das nicht mehr rein mechanisch funktioniert, kann Daten über seine Nutzungsweise sammeln. Diese Daten werden von den Herstellern gerne dazu genutzt, um Vorhersagemodelle aufzubauen, z.B. für Kaufempfehlungen oder zur Verschleiss-Vorhersage.

Wie alle Dinge im Leben unterliegen auch die Information und das Wissen, dass aus solchen Daten extrahiert wurde, konstanten Veränderungseinflüssen. Wir nennen das Concept Drift. Dieser Concept Drift kann verschiedenartig sein: Veränderte Rahmenbedingungen,

veränderte Erwartungshaltung an das Resultat des Modells, oder auch komplett neue Anforderungen können einen Grund darstellen. In jedem Fall muss ein vorhandenes Modell angepasst, oder sogar neu gelernt werden, um diesen Veränderungen standzuhalten.

Verwandte Arbeiten in diesem Gebiet decken eine große Menge an adaptiven Lernverfahren ab. Oftmals sind diese Algorithmen so ausgelegt, dass der komplette Lernvorgang auf einem Datenstrom stattfindet. Wir argumentieren allerdings, dass dieses Vorgehen in echten Anwendungen selten so stattfindet. Stattdessen werden stationäre Modelle auf großen Datensätzen in einer Sandbox gelernt, und danach in ein Produktivsystem eingesetzt. Sofern diese Modelle nicht explizit darauf ausgelegt wurden, adaptiv zu sein, kann die Performance durch Concept Drift negativ beeinflusst werden. Das Training eines solchen Modells, z.B. eines tiefen neuronalen Netzwerks, ist in der Regel ein langwieriger und kostspieliger Prozeß, weswegen eine lange Nutzbarkeit des Modells wünschenswert ist.

Wir stellen ein neues Konzept zur Adaption bestehender Modelle vor: Unser Ziel ist es, die Lebenszeit vorhandener Modelle signifikant zu verlängern, und das Modell nur an den Stellen anzupassen, wo es Fehler macht. Dazu berechnen wir partielle Modell-Adaptationen, sogenannte Patches. Ob ein Patch benötigt wird oder nicht, wird mittels eines Error-Estimators bestimmt. Dieser Estimator und der Patch sind adaptiv, und werden mit den neusten Daten aktuell gehalten. Sie können gegebenenfalls problematische Instanzen abfangen und korrekt klassifizieren. Unproblematische Instanzen werden weiterhin mit dem Basismodell klassifiziert. Durch diesen Mechanismus wird der Lernaufwand zur Adaption minimiert, und das Modell bleibt dennoch einsatzfähig.

In dieser Dissertation führen wir zunächst ein generelles Framework für das Lernen von Patch-Adaptionen ein. Wir evaluieren das Konzept und treffen Vergleiche zu bestehenden Mechanismen. Beim Einsatz in generischen Datenströmen kann die *Patching*-Methode mit dem state of the art mithalten. In Szenarien, die dem primären Einsatzzweck von Patching entsprechen, können wir allerdings signifikante Verbesserung in der Adaptionsgeschwindigkeit und Gesamtperformance erzielen.

In einem zweiten Beitrag spezialisieren wir die *Patching*-Idee auf den Einsatz mit neuronalen Netzwerken. Die Erzeugung neuronaler Netze ist mit hohem Zeit- und Rechenaufwand verbunden. Sie eignen sich zwar prinzipiell für stetige Adaption durch weitere Trainingsschritte, jedoch ist dies ebenfalls aufwendig und kann zu unerwünschten Nebeneffekten wie Catastrophic Forgetting führen. Wir stellen Neural Network Patching (*NN-Patching*) als Lösung für dieses Problem vor: Bei *NN-Patching* bleibt das Basisnetzwerk unangetastet, und die Adaption findet über einen neuronalen Patch statt. Wie bei *Patching* ist auch hier ein Error Estimator im Einsatz, der die Notwendigkeit, eine Instanz mittels des Patches zu klassifizieren, abschätzt. Der Vorteil der neuronalen Patches liegt in der Möglichkeit, die inneren Aktivierungen des Basisnetzwerkes abzugreifen. Dadurch können latente Feature-Repräsentationen aus dem Basisnetzwerk weitergenutzt werden, was

hilfreich für die Adaption ist. *NN-Patching* kann in diesem Bereich auch signifikante Verbesserungen gegenüber üblichen Adaptionsmethoden erzielen.

Der finale Beitrag dieser Dissertation beschäftigt sich mit Patching für Szenarien mit wiederkehrenden Konzepten oder Individualisierungscharakter. Hierfür stellen wir *Ensemble Patching* vor, eine Variante von Patching die ein Ensemble aus Patches generiert. Jeder dieser Patches wird so trainiert, dass er eine gewisse Art von Concept Drift abdeckt. Wenn ein neues Konzept auftritt, verursacht dies ein bestimmtes Fehlermuster im Basisklassifizierer. Für jedes spezifische Muster wird ein Patch gelernt. Ein rekurrentes neuronales Netzwerk lernt darauffolgend, eine Gesamtentscheidung anhand der aktuellen Situation und mit den Resultaten aller Patches zu generieren. Dieser sogenannte Ensemble Conductor kann schnell zwischen verschiedenen Konzepten hin- und herwechseln, und somit auf wiederkehrende Konzepte reagieren. Zudem ist *Ensemble Patching* auch in rechenzeitkritischen Datenstrom-Szenarien einsetzbar, da der Rechenaufwand zur Adaption reduziert werden konnte.

# Publications

---

Parts of this thesis have been previously published in the following publications:

- Fleckenstein, Lukas, Sebastian Kauschke, and Johannes Fürnkranz (Apr. 2019). “Beta Distribution Drift Detection for Adaptive Classifiers.” In: *Proceedings of the 2019 European Symposium on Artificial Neural Networks – ESANN’19*.
- Kauschke, Sebastian, Lukas Fleckenstein, and Johannes Fürnkranz (July 2019). “Mending is Better than Ending: Adapting Immutable Classifiers to Nonstationary Environments using Ensembles of Patches.” In: *Proceedings of the 2019 International Joint Conference on Neural Networks – IJCNN’19*.
- Kauschke, Sebastian and Johannes Fürnkranz (Feb. 2018). “Batchwise Patching of Classifiers.” In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence – AAAI’18*.
- Kauschke, Sebastian and David Hermann Lehmann (Dec. 2018). “Towards Neural Network Patching: Evaluating Engagement-Layers and Patch-Architectures.” In: *arXiv preprint arXiv:1812.03468*.
- Kauschke, Sebastian, David Hermann Lehmann, and Johannes Fürnkranz (July 2019). “Patching Deep Neural Networks for Nonstationary Environments.” In: *Proceedings of the 2019 International Joint Conference on Neural Networks – IJCNN’19*.
- Esel, Ousama, Sebastian Kauschke, and Stephan Rinderknecht (May 2019). “Predicting and Forecasting the Lifetime of Automotive Vehicle Components.” In: *29. VDI-Fachtagung Technische Zuverlässigkeit 2019*. VDI-Berichte 2345. Düsseldorf: VDI Wissensforum GmbH, pp. 321–336.
- Heuschkel, Jens and Sebastian Kauschke (Oct. 2018). “More Data Matters: Improving CGM Prediction via Ubiquitous Data and Deep Learning.” In: *Proceedings of the 3rd International Workshop on Ubiquitous Personal Assistance (co-located with Ubicomp)*, pp. 1–8.
- Kauschke, Sebastian (Oct. 2016). “Improving Cargo Train Availability with Predictive Maintenance: An Overview and Prototype Implementation.” In: *AET Papers Repository*. Association for European Transport.
- Kauschke, Sebastian, Johannes Fürnkranz, and Frederik Janssen (Oct. 2016). “Predicting Cargo Train Failures: A Machine Learning Approach for a Lightweight Prototype.” In: *Proceedings of the 19th International Conference on Discovery Science – DS’16*, pp. 151–166.
- Kauschke, Sebastian, Max Mühlhäuser, and Johannes Fürnkranz (Oct. 2018). “Leveraging Reproduction-Error Representations for Multi-Instance Classification.” In: *Proceedings of the 21st International Conference on Discovery Science – DS’18*.

- Kauschke, Sebastian, Immanuel Schweizer, Michael Fiebrig, and Frederik Janssen (2014). "Learning to Predict Component Failures in Trains." In: *Proceedings of the 16th LWA Workshops: KDML, IR and FGWM*. Ed. by Thomas Seidl, Marwan Hassani, and Christian Beecks. Aachen, Germany: CEUR Workshop Proceedings, pp. 71–82.
- Kauschke, Sebastian, Immanuel Schweizer, and Frederik Janssen (Oct. 2015). "On the Challenges of Real World Data in Predictive Maintenance Scenarios: A Railway Application." In: *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB*. Ed. by Sebastian Görg, Gilbert Müller, and Ralph Bergmann. Trier, Germany: CEUR Workshop Proceedings, pp. 121–132.
- Meurisch, Christian, Philipp M. Scholl, Usman Naeem, Veljko Pejovic, Florian Müller, Elena Di Lascio, Pei-Yi (Patricia) Kuo, Sebastian Kauschke, Muhammad Awais Azam, and Max Mühlhäuser (Oct. 2018). "UPA18: 3rd International Workshop on Ubiquitous Personal Assistance." In: *Proceedings of the 2018 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp): Adjunct Publication*. ACM.
- Kauschke, Sebastian, Frederik Janssen, and Immanuel Schweizer (Oct. 2015). *Advances in Predictive Maintenance for a Railway Scenario - Project Techlok*. Tech. rep. Knowledge Engineering Group.

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Research Challenges . . . . .	3
1.2	Contributions . . . . .	4
1.3	Outline . . . . .	5
<b>2</b>	<b>Foundations</b>	<b>7</b>
2.1	Machine Learning . . . . .	7
2.1.1	Notation . . . . .	8
2.1.2	Classification . . . . .	9
2.1.3	Regression . . . . .	10
2.2	Evaluating ML Algorithms . . . . .	10
2.2.1	Performance Measures . . . . .	11
2.2.2	Evaluation Procedures . . . . .	11
2.2.3	Over- and Underfitting . . . . .	12
2.2.4	Statistical Evaluation . . . . .	12
2.3	Rule Learning . . . . .	14
2.4	Decision Tree Learning . . . . .	14
2.4.1	Random Forest . . . . .	16
2.5	Artificial Neural Networks . . . . .	16
2.5.1	Convolutional Neural Networks . . . . .	20
2.5.2	Recurrent Neural Networks . . . . .	22
2.5.3	Feature Hierarchy in ANNs . . . . .	24
<b>3</b>	<b>Adaptive Machine Learning</b>	<b>25</b>
3.1	Concept Drift . . . . .	25
3.2	Online Adaptive Learning Procedure . . . . .	27
3.3	Change Detection . . . . .	27
3.3.1	Control Charts . . . . .	28
3.3.2	Distribution Monitoring . . . . .	29
3.4	Concept Drift Learner Evaluation . . . . .	29
3.4.1	Interleaved Test-Then-Train Procedure . . . . .	30
3.4.2	Evaluation Metrics . . . . .	30
3.5	Adaptive Learning Methods . . . . .	32
3.5.1	Concept Drift Learners . . . . .	32
3.5.2	Ensembles in Concept Drift . . . . .	33
3.5.3	Recurring Concept Learning . . . . .	34
3.6	Transfer Learning . . . . .	35
3.7	Datasets . . . . .	36
3.7.1	SEA Concepts . . . . .	36
3.7.2	Rotating Hyperplane . . . . .	36
3.7.3	20 Newsgroups . . . . .	37
3.7.4	MNIST and NIST . . . . .	37
<b>4</b>	<b>Classifier Patching</b>	<b>39</b>
4.1	Introduction . . . . .	39

4.2	Data Stream Learning . . . . .	40
4.2.1	Instance-wise Stream Learning . . . . .	41
4.2.2	Batch-wise Stream Learning . . . . .	41
4.2.3	Our Goal . . . . .	42
4.3	Patching Classifiers . . . . .	42
4.3.1	Problem Description . . . . .	42
4.3.2	The Patching Approach . . . . .	42
4.3.3	Error Region and Patch Learning . . . . .	44
4.4	Related Work . . . . .	44
4.5	Experimental Setup . . . . .	45
4.5.1	The Datasets . . . . .	48
4.5.2	The Patching Environment . . . . .	49
4.5.3	Benchmark Algorithms . . . . .	50
4.6	Results . . . . .	51
4.6.1	Evaluation Measures . . . . .	51
4.6.2	Patching Scenario . . . . .	52
4.6.3	Transfer Learning Datasets . . . . .	54
4.6.4	Concept Drift Datasets . . . . .	56
4.6.5	Significance Testing . . . . .	58
4.7	Discussion of the Approach . . . . .	59
4.8	Conclusion . . . . .	60
5	<b>Neural Network Patching</b> . . . . .	61
5.1	Introduction . . . . .	61
5.2	Related Work . . . . .	63
5.2.1	Adaptive Learning with Neural Networks . . . . .	63
5.2.2	Transfer Learning with Neural Networks . . . . .	63
5.3	Deep Neural Network Adaptation . . . . .	64
5.3.1	Neural Network Patching . . . . .	64
5.4	Preliminary Experimental Setup . . . . .	66
5.4.1	Evaluation Measures . . . . .	66
5.4.2	Evaluation Datasets . . . . .	67
5.5	Preliminary Evaluation . . . . .	68
5.5.1	Base Classifier Architectures . . . . .	68
5.5.2	Patch Architecture Notation . . . . .	70
5.5.3	Engagement Layer Selection . . . . .	70
5.5.4	Patch Architecture . . . . .	72
5.5.5	Training and Error Estimation . . . . .	74
5.6	Experimental Setup . . . . .	78
5.6.1	Benchmark Algorithms . . . . .	79
5.7	Results . . . . .	81
5.7.1	Deep Networks – The <i>MNIST</i> dataset . . . . .	81
5.7.2	Convolutional Networks – The <i>NIST</i> dataset . . . . .	82
5.7.3	Significance Testing . . . . .	83
5.8	Conclusion . . . . .	83
6	<b>Ensembles of Patches</b> . . . . .	87
6.1	Introduction . . . . .	88
6.2	Problem Description . . . . .	89
6.3	Related Work . . . . .	89

6.4	Beta Distribution Drift Detection . . . . .	90
6.4.1	Problem Definition . . . . .	91
6.4.2	Beta Distribution Drift Detection Method . . . . .	91
6.4.3	Experiment Setup . . . . .	93
6.4.4	Results . . . . .	95
6.4.5	Conclusion . . . . .	96
6.5	Ensembles of Patches . . . . .	96
6.5.1	Components of Ensemble Patching . . . . .	97
6.5.2	Classification . . . . .	97
6.5.3	Adaptation . . . . .	98
6.5.4	Dimensionality Reduction . . . . .	99
6.5.5	New Patch Creation . . . . .	99
6.5.6	Existing Patch Update . . . . .	100
6.5.7	Ensemble Conductor . . . . .	100
6.6	Experiment Setup . . . . .	100
6.6.1	The MNIST Dataset . . . . .	100
6.6.2	Rotating Hyperplane . . . . .	101
6.6.3	The SEA concepts . . . . .	101
6.6.4	Evaluation Measures . . . . .	101
6.6.5	Benchmark Algorithms . . . . .	102
6.6.6	Evaluation Procedure . . . . .	102
6.7	Results . . . . .	103
6.7.1	Significance Testing . . . . .	105
6.7.2	Comparison to Patching . . . . .	107
6.8	Conclusion . . . . .	109
7	<b>Conclusion</b> . . . . .	111
7.1	Challenges Revisited . . . . .	111
7.2	Outlook . . . . .	113
7.3	Final Remarks . . . . .	115
A	<b>Appendix</b> . . . . .	117
A.1	Results for NN-Patching Variants . . . . .	117
A.2	Adaptation Process in Ensemble Patching . . . . .	121
A.3	Decay Convergence Proof . . . . .	122
	<b>Bibliography</b> . . . . .	123

# List of Figures

---

Figure 2.1	An Example for a linear regression (orange) fitted to some datapoints. . . . .	10
Figure 2.2	Friedman/Nemenyi statistical test on the average ranks of four classifiers. Methods within the CD are visually connected. . . . .	13
Figure 2.3	Friedman statistical test on the average ranks of four classifiers with Wilcoxon pairwise comparisons. Methods that are <b>not</b> significantly different are connected. . . . .	13
Figure 2.4	Linear regression compared to the approximation of a rule-learner . . . . .	14
Figure 2.5	Example decision tree . . . . .	15
Figure 2.6	Feed-forward neural network for binary classification. . . . .	17
Figure 2.7	Activation Functions for Artificial Neurons. . .	18
Figure 2.8	Gradient descent with different learning rates (schematic depiction). . . . .	20
Figure 2.9	Recurrent neural network with hidden state. .	22
Figure 2.10	Recurrent neural network unfolding. . . . .	23
Figure 2.11	Feature hierarchy and evolution of convolutional layer features over three steps Zeiler and Fergus (2013). . . . .	24
Figure 3.1	Illustration of real and virtual concept drift . .	26
Figure 3.2	Illustration of different types of concept drift .	27
Figure 3.3	Phases during the course of the stream. CPs are shown as grey vertical lines. . . . .	31
Figure 3.4	The <i>MNIST</i> dataset of handwritten digits. . . .	37
Figure 4.1	Learning error regions on a 2-dimensional dataset with a base classifier $M$ and instances from a different distribution $D_i$ . . . . .	43
Figure 4.2	Phases during the course of the stream. CPs are shown as gray vertical lines (at point 3.5), the initialization phase ends at the dashed vertical line. . . . .	46
Figure 4.3	Results of stream classification accuracy progression for problems that resemble the intended use of <i>Patching</i> . . . . .	52
Figure 4.4	Results of stream classification accuracy progression for <i>transfer</i> problems . . . . .	55
Figure 4.5	Results of stream classification accuracy progression for <i>concept drift</i> problems . . . . .	56
Figure 4.6	Statistical comparison of the final ranks. . . . .	58
Figure 4.7	Statistical comparison of the adaptation ranks. .	59

Figure 5.1	Patching of feed-forward networks – blue: the original network, green: the patch, red: the error region network . . . . .	65
Figure 5.2	Accuracy Progression Diagram on the $NIST_{flip}$ dataset. . . . .	72
Figure 5.3	Patch training paradigms: <i>Inclusive</i> , <i>Exclusive</i> and <i>Semi-Exclusive</i> patch training. . . . .	77
Figure 5.4	Transfer learning with neural networks: <i>Freezing</i> .	80
Figure 5.5	Results on the $MNIST_{flip}$ dataset as a data stream.	80
Figure 5.6	Results on the $MNIST_{remap}$ dataset as a data stream. . . . .	81
Figure 5.7	Results on the $NIST_{remap}$ dataset as a data stream.	81
Figure 5.8	Results on the $NIST_{appear}$ dataset as a data stream.	82
Figure 5.9	Results on the $NIST_{rotate}$ dataset as a data stream.	82
Figure 5.10	Friedman test with Wilcoxon signed-rank test for pairwise comparison. . . . .	83
Figure 6.1	Two example <i>Beta</i> probability density functions with their boundaries that contain 99.7% of the distribution. . . . .	93
Figure 6.2	Accuracy comparison on the Rotating Hyperplane dataset ( $20^\circ$ ). . . . .	95
Figure 6.3	Sequential prediction of the ensemble conductor $\mathbb{C}$ . . . . .	98
Figure 6.4	Influence of the data features given to the conductor. . . . .	99
Figure 6.5	Accuracy on abrupt drift with reoccurring concepts. . . . .	103
Figure 6.6	Accuracy on gradual drift with different speed.	106
Figure 6.7	Accuracy on <i>SEA</i> . . . . .	106
Figure 6.8	Wilcoxon signed-rank test on the adaptation rank.	106
Figure 6.9	Comparison between <i>Ensemble Patching</i> and the original <i>Patching</i> on the $MNIST$ -based data sets.	108
Figure 6.10	Comparison between <i>Ensemble Patching</i> and the original <i>Patching</i> on <i>SEA</i> . . . . .	108
Figure 6.11	Comparison between <i>Ensemble Patching</i> and the original <i>Patching</i> on <i>RotHyp</i> . . . . .	109
Figure A.1	Adaptation process in the <i>Ensemble Patching</i> algorithm. . . . .	121

# List of Tables

---

Table 2.1	The golf example – a well known teaching dataset.	9
Table 4.1	Summary of the datasets used in the experiments	47
Table 4.2	Experiment results for the <i>Patching</i> scenario .	53
Table 4.3	Experiment results in the <i>transfer</i> domain . . .	54
Table 4.4	Experiment results in the <i>concept drift</i> domain	57
Table 5.1	Summary of the datasets used in the experiments	67
Table 5.2	Base classifier accuracy on unaltered datasets.	69
Table 5.3	Fully-Connected Architectures. . . . .	69
Table 5.4	Convolutional Architectures. . . . .	70
Table 5.5	Best engagement layer per dataset. The patch user here had a single hidden layer with 128 units. . . . .	73
Table 5.6	Top Ranked Patch Architectures – Grouped by Evaluation Measure. . . . .	74
Table 5.7	Number one ranks of <i>NN-Patching</i> variants in preliminary evaluation (on 20 datasets). . . . .	78
Table 5.8	Network architectures for base neural networks in the experiments. . . . .	79
Table 5.9	Results for <i>MNIST</i> (Base classifier is a fully connected deep network) . . . . .	85
Table 5.10	Results for <i>NIST</i> (Base classifier is a convolutional network) . . . . .	86
Table 6.1	Results on the Bit-Stream dataset (Standard deviation in brackets). . . . .	95
Table 6.2	Results on synthetic datasets (Accuracy). . . . .	96
Table 6.3	Results on the Elec2 dataset. . . . .	96
Table 6.4	Experimental results for <i>MNIST</i> – Overview .	104
Table 6.5	Experimental results for <i>RotHyp</i> – Overview . .	105
Table 6.6	Experimental results <i>SEA</i> – Overview . . . . .	107
Table A.1	Comparison of neural network patching and transfer learning techniques on <i>MNIST</i> with FC-NN base classifiers. . . . .	117
Table A.2	Comparison of neural network patching and transfer learning techniques on <i>NIST</i> with FC-NN base classifiers. . . . .	118
Table A.3	Comparison of neural network patching and transfer learning techniques on <i>MNIST</i> with CNN base classifiers. . . . .	119
Table A.4	Comparison of neural network patching and transfer learning techniques on <i>NIST</i> with CNN base classifiers. . . . .	120

# Acronyms

---

ACE	<i>Adaptive Classifier Ensemble</i>
ADWIN	<i>Adaptive Sliding Window</i>
AI	<i>Artificial Intelligence</i>
ANN	<i>Artificial Neural Network</i>
AUE	<i>Accuracy Updated Ensemble</i>
AWE	<i>Accuracy Weighted Ensemble</i>
BD <sup>3</sup>	<i>Beta Distribution Drift Detection</i>
CCP	<i>Conceptual Clustering and Predictions</i>
CD	<i>Critical Distance</i>
CNN	<i>Convolutional Neural Network</i>
CP	<i>Change Point</i>
CV	<i>Cross Validation</i>
DACC	<i>Dynamic Adaptation to Concept Changes</i>
DDM	<i>Drift Detection Method</i>
DM	<i>Data Mining</i>
DNN	<i>Deep Neural Network</i>
DWM	<i>Dynamic Weighted Majority</i>
EDDM	<i>Early Drift Detection Method</i>
EWMA	<i>Exponential Weighted Moving Average</i>
FAE	<i>Fast Adapting Ensemble</i>
FASE	<i>Fast Adaptive Stacking of Ensembles</i>
FC-NN	<i>Fully Connected Neural Network</i>
FNR	<i>False Negative Rate</i>
FPR	<i>False Positive Rate</i>
GRU	<i>Gated Recurrent Unit</i>
HDDM	<i>Hoeffding Drift Detection Method</i>

KL *Kullback-Leibler*  
LOO *Leave one out cross validation*  
LSTM *Long Short-term Memory*  
MAE *Mean Absolute Error*  
ME *Model Error Estimation*  
ML *Machine Learning*  
MLP *Multilayer-Perceptron*  
MOA *Massive Online Analysis*  
MSE *Mean Squared Error*  
NB *Naïve Bayes*  
PAC *Probably Approximately Correct*  
PE *Patch Error Estimation*  
RELU *Rectified Linear Unit*  
RIPPER *Repeated Incremental Pruning to Produce Error Reduction*  
RF *Random Forest*  
RMSE *Root Mean Squared Error*  
RNN *Recurrent Neural Network*  
SGD *Stochastic Gradient Descent*  
SPC *Statistical Process Control*  
VFDT *Very Fast Decision Tree*  
WEKA *Waikato Environment for Knowledge Analysis*

# Symbols

---

$A(\mathbf{x})$	Autoencoder neural network (reproduces given input $\mathbf{x}$ through a lower dimensional feature representation).
$\alpha$	Shape parameter for the Beta distribution function.
$Beta$	Beta distribution function $Beta(\pi \alpha, \beta)$ .
$\beta$	Shape parameter for the Beta distribution function.
$Bin$	Binomial distribution function $Bin(k_i n_i, \pi_i)$ .
$C$	The set of possible classes $C = \{c_1, \dots, c_n\}$ .
$c$	A label/class $\in C$ .
$CE(X)$	Cross-entropy loss function.
$\mathbb{C}$	Ensemble conductor. Decides an ensemble based on the decisions of the members.
$D$	A set of examples used for training or testing $D = \{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_{ D }, \mathbf{y}_{ D })\}$ .
$\mathcal{D}$	A domain in transfer learning.
$\mathbb{D}$	Drift detector (Detects concept drift on a stream of Bernoulli trials).
$\delta()$	Error/loss function $\delta = \mathcal{Y} \times \mathcal{Y} \rightarrow [0; \infty) \subset \mathbb{R}$ .
$\eta$	Learning rate in neural network training.
$l()$	Label definition function $l(\mathbf{x}) = \mathbf{y}$ , $\mathbf{y}$ in $C$ .
$M(\mathbf{x})$	A predictive model (a function that is learned to predict $\mathbf{y}$ for a given $\mathbf{x}$ ).
$n$	A scalar value $\in \mathbb{Z}$ .
$P(\mathbf{x})$	A patch model (trained to classify a subset of the instance space $\mathcal{X}$ ).
$\pi$	Error rate.
$\Pr(a)$	Probability of $a$ .
$\Pr(a b)$	Probability of $a$ , given $b$ .
$S$	A sequence of elements $S = (s_1; \dots; s_n)$ .
$\sigma$	Standard deviation.

<i>Softmax</i>		The Softmax function.
$\mathcal{T}$		A task in transfer learning.
$w$		A weight scalar ( $w$ ) or vector ( $\mathbf{w}$ ).
$\mathcal{X}$		The input space $\mathcal{X} \subseteq \mathbb{R}^n$ .
$X$		A set of vectors $X = (\mathbf{x}_1; \dots; \mathbf{x}_n)$ .
$\mathbf{x}$		A vector $\mathbf{x} \in \mathcal{X}$ .
$(\mathbf{x}, \mathbf{y})$		An example.
$\mathcal{Y}$		The output space $\mathcal{Y} \subseteq \mathbb{R}^n$ .
$\mathbf{y}$		The prediction target $\mathbf{y} = f(\mathbf{x})$ , $\mathbf{y} \in \mathcal{Y}$ .
$\hat{\mathbf{y}}$		The prediction $\hat{\mathbf{y}} = M(\mathbf{x})$ , $\hat{\mathbf{y}} \in \mathcal{Y}$ .

# Introduction

---

*Artificial Intelligence* (AI) is a subdomain of computer science, concerned with the creation of computer systems that show some kind of intelligent behavior. While the term itself and the definition of being *intelligent* are debatable<sup>1</sup>, AI has shown a significant rise in popularity over the last decade. Especially the term *Machine Learning* (ML), which is mostly concerned with learning from large amounts of historical data in order to gain knowledge that can be applied on future data, is currently very popular<sup>2</sup>.

Nowadays, data are collected in vast amounts. Personal data is retrieved by our phones, by our internet browser, via our shopping behavior, and especially through all the content that we upload to social media platforms. But also machines in factories, cars, essentially every device that is not purely mechanical anymore, may be collecting data.

Ideally, these data are then transferred and stored in large databases, where they can be evaluated and prepared for *Data Mining* (DM) and ML, which might gain interesting insights. While DM is the general process of extracting knowledge from large amounts of data, ML aims at generating prediction models. In the case of personal information, the extracted knowledge and generated models can be used to have better assistance systems: automatically sort your mailbox based on your preferences, keep track of health-related aspects, motivate you to exercise more, and—of course—advertise products to you based on what you bought in the past.

Prediction models learned on machine data can, on the other hand, be used to give estimations of the machines' state, or to predict upcoming mechanical issues (Sipos et al. 2014; Hashemian 2011; Peng, Dong, and Zuo 2010). This can help moving from scheduled maintenance processes to predictive maintenance, reducing maintenance cost of a machine and avoiding failures (Esbel, Kauschke, and Rinderknecht 2019).

Another current hot topic in AI is autonomous driving, where the car is steered by the AI with the help of sensors such as cameras, radar, and lidar. This is one of the most complex endeavours today (Hodges et al. 2019; Bojarski et al. 2016).

As all things in life, the data and the knowledge extracted from a person or machine are subject to change. This change may be caused by varying circumstances or a change in the desired result. For example, when you buy a washing machine, it does not make sense for a

---

1 <http://jmc.stanford.edu/artificial-intelligence/what-is-ai>

2 According to google trends on the terms *artificial intelligence* and *machine learning*.

recommender system to suggest additional washing machines. However, if you bought coffee beans, the same logic does not necessarily apply. A second example would be the availability of more data. For instance, imagine the newest revision of a machine has more sensors than the previous one. These additional sensors could be leveraged for better predictions, but only if the predictive model is able to use them properly.

In ML, such a change in the underlying structure is called concept drift (Gama, Medas, et al. 2004). In order to deal with concept drift, a prediction model must be adapted to the drift. In the worst case, this means retraining the model on new data, which requires a lot of data and can be computationally complex. In the best case, the model was designed to be adapted and can handle the adaptation just fine. However, most machine-learned models are constructed as stationary models with little to no adaptive capabilities, and adaptation itself can be a complex process with many potential issues.

In this thesis, we investigate means to handle concept drift without having to re-train or alter the underlying predictive model. In general, there can be numerous reasons why we would not want to alter the existing model itself. For example when the model is a black box. This can be the case if we do not have access to the model for some reason, or it may be that the model is cast in hardware. Another reason might be complexity. Training a classification model can be a difficult and cumbersome task, which currently still requires engineers with domain knowledge.

However, the number of ML practitioners that are mostly concerned with applying ML is constantly growing. In order to achieve their main goals, it can be unnecessary to go through all the model training steps again and again, if all you need is to adapt an existing model that is very close to what you want to achieve already. Cost of training and preparation is also an argument for corporate use of ML models. Imagine the manufacturer of production machines that incorporate a predictive maintenance model. The release of a minor hardware upgrade would lead to the necessity of all models being re-trained, when the actual effect of the upgraded parts might be limited. As a final example we would like to motivate the necessity of individualizing general models. A handwriting recognition system may be well suited for general use, but of course adaptation towards the user's own handwriting may just add that layer of refinement that takes a product to the next level.

This thesis is aimed at dealing with the scenario, where adapting the existing classifier is infeasible or even impossible. We will elaborate on the question, how to make an existing model adaptable, and, based on the resulting framework, explore further possibilities of adapting complex classifiers such as deep neural networks. We also investigate measure to provide constant adaptation for such an immutable model. In the following sections we will state the underlying challenges and research questions, followed by the contributions presented in this thesis to solve these challenges.

## 1.1 Research Challenges

In this thesis, we tackle a previously neglected problem: Adapting a fixed classifier for new concepts. This task requires a new methodology, and with that methodology some challenges arise. Our overall goal is to increase the lifespan of existing models and make them reusable for similar scenarios without having to re-engineer them. Our target audience is the people who apply ML to everyday problems, and may not have a background in ML research. In order to do this, we want to provide a high-level framework that is easy to use on practically any classifier that makes a prediction. In the following we give an overview on the challenges.

1. **Make existing systems adaptable.** First, we investigate a framework that allows us to adapt an existing black-box classifier. We assume that the original model still classifies instances, but the resulting classification may be erroneous due to concept drift. Our goal is, to establish a mechanism that can adapt this classifier, based on the data (instances) and the result of the classifier.
2. **Fix only aspects of model that are broken.** When concept drift occurs, it is possible that it does not affect all of the instances or the whole instance space. It may be a partial concept drift, such that the original classifier still classifies part of the instances well. We want to find out exactly which instances are not classified well, and fix the classification for them, explicitly.
3. **Leverage the existing model.** Depending on how much information we can retrieve from the original classifier, it is feasible to assume that this information can help in handling the concept drift. Especially when we consider neural networks and their hidden layers, information can be gained from inside the network at classification time. We want to find a method that can leverage these latent features for better adaptation.
4. **Increase model re-usability.** The goal of our efforts is the prolonged use of otherwise outdated models, in order to cut down the cost of training/enhancing models permanently. In the most extreme case, a model can be used for a related task in another domain. We investigate the possibilities of transfer learning with our proposed method.
5. **Learn online.** In data stream classification scenarios, a continuous stream of data exists. This scenario has certain implications w. r. t. computational efficiency and storage capacity, which we will address with our method.
6. **Deal with recurring concepts.** When concept change occurs in a data stream, it is reasonable to assume that concepts re-occur

from time to time. This may be caused by seasonal effects or other recurring patterns. We want to be able to deal with recurring concepts without having to re-adapt to concepts that have been there before, as this will prolong the adaptation process.

## 1.2 Contributions

This thesis contributes the following methods to the corpus of adaptive machine learning methodology.

### **The Patching Framework**

The patching framework enables any blackbox-classifier to be made adaptable. The main concept of patching is to separately learn, where in the instance space the underlying classifier errs, and specifically add smaller classifiers, so-called patches, to fix the classification in those regions. The framework allows free selection of the error-region learner as well as for the patches themselves, and can be applied to any classifier. The error-region learner and the patches are trained on the instance attributes, and additionally receive the output of the base classifier as input. The system works in a batch-setting, where we assume that the true labels of the instances will arrive after some time, so they can be used for the adaptation process.

### **Neural Network Patching**

Neural network patching is an extension to the patching framework, which is geared towards (deep) neural networks. The idea of neural network patching is to leverage the latent feature representations inside the layers of the network. The patch and error estimator attach to one of the network layers, and do their classification based on these representations. In this approach, the patch is ideally provided with the most useful data for the given task. This leads to a simpler patch architecture and hence faster adaptation. This chapter discusses where that ideal attachment is, given a certain network architecture and task. A set of heuristic rules is derived based on an empirical study.

### **Ensemble Patching**

The second extension to the patching framework is ensemble patching. It is especially geared towards long-term adaptation of an immutable classifier. One of the main features of ensemble patching is its ability to quickly adapt to recurring concepts. This is achieved by not removing any members from the ensemble of patches, combined with a novel ensemble decision method that can quickly adapt to the current concept. Together with a novel active drift detection method, this ensemble of patches adapts faster and more consistently than state-of-the-art ensemble learners.

### **Beta-Distribution Drift Detection**

*Beta Distribution Drift Detection (BD<sup>3</sup>)* is a novel active drift detection method that uses beta distributions to detect significant changes in the results of bernoulli trials, such as can be encountered when dealing

with misclassifications of a classifier. We use Bayesian inference to iteratively update the assumptions about the error-rate of a classifier, and base its drift detection on confidence bounds.  $BD^3$  shows advantages compared to similar methods regarding false-positive drift detections. It is used in ensemble patching as an active drift detector.

## 1.3 Outline

The remainder of this thesis consists of the following chapters. Chapter 2 gives a basic understanding of ML in general, and Chapter 3 introduces concept drift and adaptive learning methods in particular. Chapter 4 introduces the patching framework and its basic idea and concepts. Chapter 5 extends the patching framework with neural network patching, followed by ensemble patching and the beta-distribution drift detection in Chapter 6. Finally, we conclude this thesis in Chapter 7 and motivate future work in this research area.



# Foundations

---

This chapter covers the foundations required to understand the contributions of this thesis. Besides basic principles, ideas and definitions, seminal work from related domains are introduced. The purpose is to provide a basic insight into how these methods work, which will help to understand the more technical parts in the later sections. We cover rule and decision tree learning methods, as well as give an introduction to artificial neural networks in general, and on convolutional and recurrent neural networks specifically. We omit some well-known learning methods, because they are not relevant for understanding this thesis.

First we will give an introductory overview on stationary machine learning in Section 2.1, followed by evaluation techniques in Section 2.2. Afterwards, we introduce popular learning paradigms in Sections 2.3–2.5.

## 2.1 Machine Learning

*Machine Learning* is a subfield of computer science that is concerned with algorithms that *learn* to program computers without the necessity of human involvement. Mitchell (1997) defines ML as follows:

“A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ .”

ML is usually divided in two subcategories by the class of the task  $T$ , which are *supervised* and *unsupervised* learning. Additionally, there is *reinforcement* learning, which is a category between *supervised* and *unsupervised* learning.

*Supervised learning* is the class of tasks, that incorporate learning from examples. This means, that experience  $E$  is usually training data which consists of input values that are annotated with an expected target value, the so-called *class* or *label*. The task  $T$  is to predict the target value for unseen data. The performance  $P$  usually measures the quality of that prediction, and the goal is to maximize the quality.

*Unsupervised learning* deals with the problem of finding structure and group membership in non-annotated data. This process is called clustering: in this case  $E$  only consists of the data as input values, and the target is to find subsets of the data instances that are ideal with respect to some cluster quality measure  $P$ . Usually, some measure of distance or density is used to find clusters.

A category between *supervised* and *unsupervised* is *reinforcement learning*. It relies on the availability of direct feedback, and  $E$  is the experience of previously performed actions, that are judged by a reward function  $P$  given the goal task  $T$ . The primary goal is to find the ideal action for any possible situation. Opposed to *supervised learning*, the feedback does not contain the actual information of what you want to learn, but only how well you did it already.

Besides these main groups there are subcategories such as semi-supervised learning, which is supervised learning but with the disadvantage that not all of the true target values are available.

### 2.1.1 Notation

This thesis is mainly concerned with supervised learning, hence we introduce a notation for all its components. Formally, supervised learning is concerned with the relation  $f$  between the input space  $\mathcal{X}$  and the output space  $\mathcal{Y}$ :

$$f : \mathcal{X} \times \mathcal{Y} \rightarrow \{true, false\}$$

In supervised learning, one of the usual assumptions is that the task  $T$  remains stationary over time, which means we can define the relation as a function  $f : \mathcal{X} \rightarrow \mathcal{Y}$  of  $\mathbf{x}$ :

$$\mathbf{y} = f(\mathbf{x}), \quad \mathbf{y} \in \mathcal{Y}.$$

The vector  $\mathbf{x} = (a_1, \dots, a_n)$ ,  $\mathbf{x} \in \mathcal{X}$  is called an instance, and consists of real-valued attributes  $a_i, i \in 1..n$ . An instance represents an arbitrary object via its attributes, e. g., an image via a list of pixels or a person via height, gender, and age.

$\mathcal{X} \subseteq \mathbb{R}^n$  is called the feature or input space. Attributes can be numerical, ordinal, or categorical, and may require a prior transformation into the feature space.

We call a mapping between  $\mathcal{X}$  and  $\mathcal{Y}$  a *hypothesis* or *model*  $M(\mathbf{x})$ . The model should resemble the true function  $f$  as closely as possible. This is achieved via a process called function approximation, and the main challenges in supervised learning is to find an adequate function  $M$ . The result of the model function given an instance  $\mathbf{x}$  is called a prediction:

$$\hat{\mathbf{y}} = M(\mathbf{x}), \quad \hat{\mathbf{y}} \in \mathcal{Y}.$$

In order to determine  $M$ , a training set  $D_{train}$  is used. It contains *examples*, various observations of  $\mathbf{x} \in \mathcal{X}$  and their true  $\mathbf{y}$ -values, such that an example constitutes a tuple  $(\mathbf{x}, \mathbf{y})$  and the training data is a set of examples:

$$D_{train} = \{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_{|D_{train}|}, \mathbf{y}_{|D_{train}|})\}$$

We say that  $M$  is *learned from* or *trained* on  $D_{train}$ .

The quality of a learned model  $M$  is determined via a test set  $D_{test}$ . The test set contains instances from the same distribution as the training

set, but should not contain instances that are identical to instances in  $D_{train}$ . Otherwise, the evaluation will not give a valid performance estimate of the model. The performance itself is calculated via a *loss* or *error* function:

$$\delta = \mathcal{Y} \times \mathcal{Y} \rightarrow [0; \infty) \subset \mathbb{R} \quad \forall \mathbf{y} \text{ in } \mathcal{Y} : \delta(\mathbf{y}, \mathbf{y}) = 0$$

Usually, the loss is computed over all predictions on  $D_{test}$ . Via such a loss function, a classifier that minimizes the expected loss can be found.

The prediction target  $\mathbf{y}$  can be either numerical or categorical. Categorical prediction is called classification, whereas numerical prediction is called regression.

### 2.1.2 Classification

In classification, the task is to determine a categorical prediction for a given data instance  $\mathbf{x}$ . In this categorical setting, the target  $\mathbf{y}$  is a *label* or *class*, selected from a finite set of labels:

$$\mathbf{y} = \{l_1, l_2, \dots, l_n\} \in C$$

For an instance  $\mathbf{x}$ , one or multiple labels can be assigned. The first is called single-label, the latter multi-label classification. In this thesis we are only concerned with the case of single-label classification. We will also refer to the label of an instance as  $l(\mathbf{x})$ ,  $l()$  being a function that defines the label.

A well-known teaching example is the so-called golf problem. The goal is to determine, based on some information about the weather, if it is a good day to go outside and play golf.

Table 2.1: The golf example – a well known teaching dataset.

Temperature	Outlook	Humidity	Windy	Play Golf?
hot	sunny	high	false	no
hot	sunny	high	true	no
hot	overcast	high	false	yes
cool	rain	normal	false	yes
cool	overcast	normal	false	yes
cool	sunny	normal	false	?

In order to make this decision, a few observations (Table 2.1) have been recorded. Based on these observations, a classification model can be trained that will assert new instances. The training data consists of observations about the temperature, wind, and humidity at a given date. A solution for this problem could be a set of consecutively executed rules. Each of the rules can decide the class, if all of the attribute tests are found to be true. If the rule is not triggered, the

next rule in the set is executed. Should no rule be triggered, a *default rule* classifies the remaining instances. Such a set of rules for the golf question could be as follows:

```
IF outlook=overcast AND windy=false THEN class=yes
IF humidity=normal THEN class=yes
DEFAULT: class=no
```

In the following sections we will elaborate on the training and evaluation of ML-models for different tasks. This includes learning rule sets, but also other methods that can be used to learn a classifier or regressor.

### 2.1.3 Regression

In regression learning, the goal is to predict one or multiple numerical values from an input value  $\mathbf{x} \in \mathbb{R}^n$ . In this case,  $\mathbf{y}$  is a vector:

$$\mathbf{y} = (y_1, \dots, y_n) \in \mathbb{R}^n$$

For example, let us assume we want to estimate the weight of a person based on body height. A number of example measurements is shown in Figure 2.1.

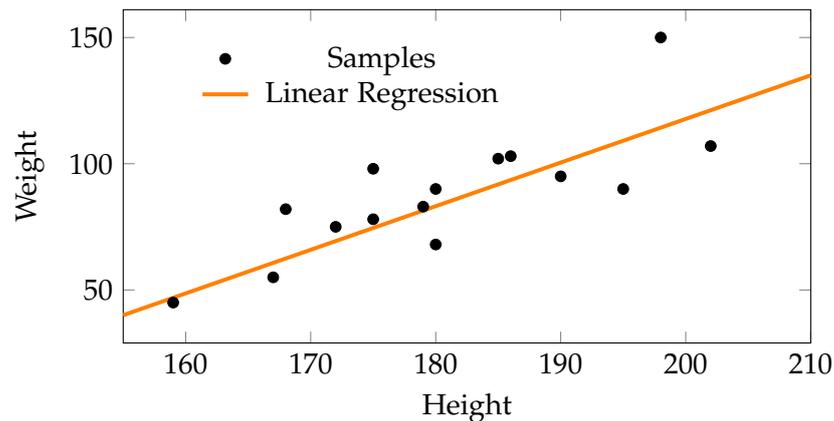


Figure 2.1: An Example for a linear regression (orange) fitted to some data-points.

A simple way of doing such a regression is *Linear Regression*, which predicts the value of a scalar  $y \in \mathbb{R}$  based on  $\mathbf{x}$ . The model's prediction  $\hat{y} = \mathbf{w}^T \mathbf{x}$  is a linear function of the input vector multiplied by a weight vector  $\mathbf{w} \in \mathbb{R}$ . The weight vector determines the influence of each component of  $\mathbf{x}$  on the result. In Figure 2.1, the line is the result of such a regression.

## 2.2 Evaluating ML Algorithms

Experimental evaluation of any ML technique requires solid performance metrics and evaluation procedures. Usually, for training a

classifier or regressor, only a limited amount of data is available. These data should be used in an ideal way to train the classifier and—at the same time—get a realistic estimate of the resulting performance.

### 2.2.1 Performance Measures

The most common measures of performance in classification are accuracy, recall, and precision.

$$\text{Accuracy} = \frac{\text{Number of correctly classified instances}}{\text{Total number of instances}} \quad (2.1)$$

Accuracy (Equation 2.1) is the number of correctly classified instances from a set of instances. Classifying 80 instances out of 100 correctly results in an accuracy of 0.8, or 80%.

$$\text{Precision} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}} \quad (2.2)$$

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}} \quad (2.3)$$

Precision on the other hand (Equation 2.2) provides information on the ability of a classifier, to correctly identify members of a class (also called specificity), while recall (Equation 2.3) is the ability to find instances of a class at all (sensitivity). Both are often combined in the so-called *F-Measure* or *F-Score*, which is the arithmetic mean of precision and recall. For regression tasks, the error of the regressor is measured instead. Metrics such as the *Root Mean Squared Error* (RMSE) (Equation 2.4), or the *Mean Absolute Error* (MAE) (Equation 2.5) are used for this task. They are calculated over all instances of a set.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2} \quad (2.4)$$

$$\text{MAE} = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j| \quad (2.5)$$

### 2.2.2 Evaluation Procedures

In order to provide a realistic estimate of the performance of a classifier, it is mandatory to execute the evaluation on data that is disjunct from the training data. Given a finite set of training data, it is therefore required to split this set into multiple parts.

**TRAIN-TEST SPLIT.** In the train-test approach, the dataset  $D$  is split into two subsets,  $D_{\text{train}}$  and  $D_{\text{test}}$ . The model is learned on  $D_{\text{train}}$ , and the performance metrics are calculated on  $D_{\text{test}}$ . Although this can give a realistic estimate, the distribution of both sets may be different, caused by a sub-optimal selection (e. g., random selection). In that case, the estimation will be less precise.

**CROSS VALIDATION.** To avoid the selection issue and get a more precise approximation, *Cross Validation* (CV) is often used. In CV,  $D$  is divided into  $k$  disjunct subsets. The training is then conducted on  $k - 1$  of those subsets, while the performance is calculated on the remaining subset. The process is repeated for all  $k$  iterations, and the results averaged. This *k-fold CV* gives a better approximation of the performance, albeit at the cost of having to learn  $k$  models instead of one. Typical values of  $k$  are 5 and 10, with multiple repetitions to assure repeatable results, e. g., 10x10 CV. One special case of CV is *Leave one out cross validation* (LOO), where the test set consists of only a single instance. This is the most expensive validation method, but gives the most accurate approximation.

Both of these splitting approaches require the instances to be completely independent of each other, since instances are randomly assigned to the sets. There must be no sequential relations between instances (i. e., one instance is the consequence of another). Otherwise, the evaluation procedure is not valid.

### 2.2.3 Over- and Underfitting

When learning a model, one of the main problems is to assure that it is neither too specific nor too general w. r. t. the training data. These effects are called *overfitting* and *underfitting*. Overfitting refers to situations where the learned model fits the training data (including potential noisy samples) too closely and is therefore not able to generalize to yet unseen instances. Underfitting describes the opposite: The model is not sufficiently well adapted for the dataset, and gives unprecise predictions.

### 2.2.4 Statistical Evaluation

In general, statistical hypothesis testing is used to quantify the likelihood of two data samples being from the same distribution.

In the ML domain, it is usually used to determine significant differences in the performance of two or more models. In order to do so, an appropriate test must be selected and applied on the observed evidence. Often, we are interested to see if the *null hypothesis*—the assumption that the distributions are **not** significantly different—can be invalidated given the observations. If the null hypothesis is rejected, it is *likely*, that the observed models differ in performance. If the null hypothesis is not rejected, any observed difference can likely be attributed to chance.

For comparing two classifiers, the well-known t-test might come to mind. However, Demšar (2006) deems this test conceptually inappropriate, and recommends to use the signed-rank test by Wilcoxon (1945) instead.

For the comparison of more than two classifiers, the Friedman test with the post-hoc Nemenyi test (Demšar 2006) is widely used throughout literature. It compares the performance of a set of classifiers over multiple datasets by their average ranks. After the null hypothesis is rejected via the Friedman test, the Nemenyi test is executed. The Nemenyi test calculates the *Critical Distance* (CD) for the ranks of the classifiers. When two classifiers exceed this distance with their average rank, they are considered to perform different with statistical significance. The result of Friedman and Nemenyi tests can be depicted as shown in Figure 2.2. Two or more classifiers are connected, when they are in the same significance group. In the figure, *Method 1* is significantly distinct from *Methods 3* and *4*. However, it is not different from *Method 2*. *Methods 2, 3, and 4* are in the same significance group.

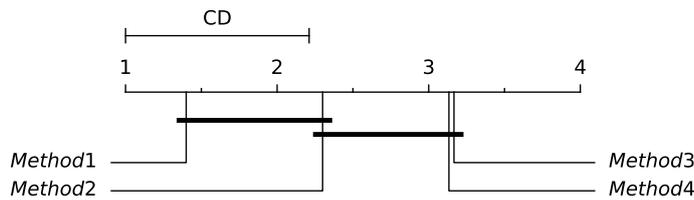


Figure 2.2: Friedman/Nemenyi statistical test on the average ranks of four classifiers. Methods within the CD are visually connected.

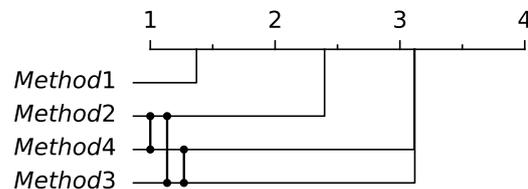


Figure 2.3: Friedman statistical test on the average ranks of four classifiers with Wilcoxon pairwise comparisons. Methods that are **not** significantly different are connected.

Besides the Friedman/Nemenyi combination, we like to use a pairwise comparison in addition. One of the main problems of the Nemenyi post-hoc test is its sensitivity towards the compared methods. This means, when comparing a set of methods, the results can be somewhat manipulated by a “clever” selection, e. g., removing or adding methods to the compared set.

To display the results of the combined Friedman/Wilcoxon test, we propose the visualization in Figure 2.3. In this visualization, two methods are connected, if they are **not** significantly different, i. e., the null hypothesis could not be rejected.

When comparing Figures 2.2 and 2.3, which are based on the same observations, we can already spot a difference: *Method 1* and *2* are significantly different for the Wilcoxon pairwise comparison, while they are in the same group when using the Nemenyi test.

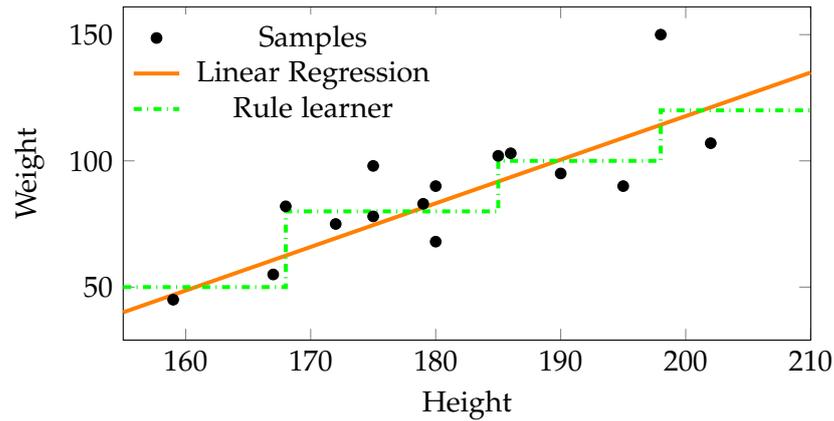


Figure 2.4: Linear regression compared to the approximation of a rule-learner

## 2.3 Rule Learning

Rule learning is the task of learning a set of rules that describe a dataset with respect to a certain classification or regression task. Usually, a separate-and-conquer algorithm is used to induce rules from a set of examples. An example consists of a set of attributes which can be numerical or categorical, and the respective categorical or numerical target value.

In the case of a two-class classification problem, separate-and-conquer rule learning tries to induce a rule that covers a set of positive examples that is as large as possible. These examples are then removed from the training set, and further induction steps are executed, until all positive examples are covered. The result is a set of rules, each consisting of conjunctions of attribute tests: simple *if-then* rules. Attribute tests can test for equality or any relation such as  $\geq$ ,  $\leq$ ,  $>$ , and  $<$  between the attribute value and a constant value determined by the rule learning algorithm.

The resulting rules are generally considered to have a higher interpretability than other learners such as support-vector machines or neural networks.

A substantial overview to separate-and-conquer rule learning is given in Fürnkranz 1999. In this thesis we will mostly refer to the *Repeated Incremental Pruning to Produce Error Reduction (RIPPER)* algorithm by Cohen 1995, as an example of a powerful rule induction algorithm that is widely implemented.

## 2.4 Decision Tree Learning

Another popular paradigm for learning classification and regression models are decision trees. In a decision tree, at each node, attribute values are compared against constants, much like in rule learning. The tree is constructed in a hierarchical manner, and traversed from the

root until a leaf is reached. Leafs represent the resulting classification or regression value, i. e., the class in case of the golf problem. A decision tree for the examples in Table 2.1 is depicted in Figure 2.5.

However, many other decision trees for this problem are possible. The goal of a decision tree learning algorithm is therefore to find a useful decision tree that is not unnecessarily complex, given a limited set of training data.

The most widely known decision tree learning algorithm is *C4.5* by Quinlan (1993), which is an extension of the *ID3* algorithm (Quinlan 1986).

Decision trees are usually induced in a top-down fashion. An appropriate root node and split criterion are selected, and the instances are split for each branch that emerges from the tests at that node. A useful split criterion can be chosen based on the entropy it provides w. r. t. the class of the instances. In order to find the best split attribute for a node, the information gain (Equation 2.6) between the split subsets  $S_i$  of a set of examples  $S$  is calculated, and the attribute with the highest gain chosen as a split criterion.

$$Gain(S, A) = E(S) - \sum_i \frac{|S_i|}{|S|} \cdot E(S_i) \quad (2.6)$$

where

$$E(S) = - \sum_{j=1}^C p_j \cdot \log_2 p_j \quad (2.7)$$

is the entropy of a data set  $S$ . Here,  $p_j$  is the relative frequency for some class  $j$ . Afterwards, the attribute that maximizes the information gain between subsets is used as split-criterion.

Alternative implementations, like the *CART* algorithm (Breiman et al. 1984), are splitting based on the Gini-index, which measures the impurity of a set of samples:

$$Gini(S) = 1 - \sum_{j=1}^C p_j^2 \quad (2.8)$$

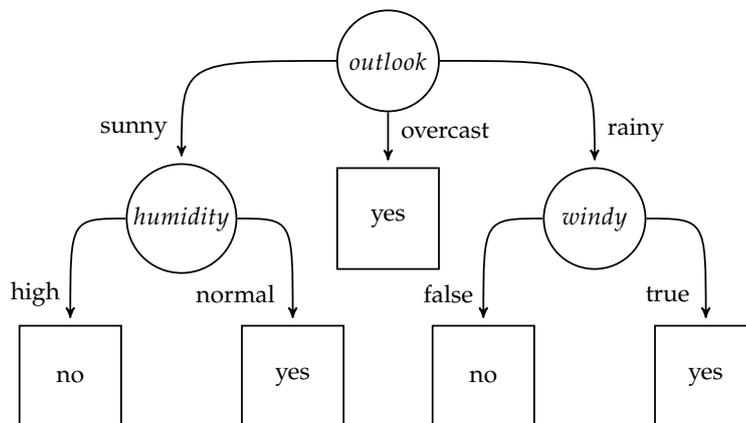


Figure 2.5: Example decision tree for the playing golf problem.

where  $p_j$  is again the relative frequency for some class  $j$ . The Gini-index for a split based on a certain attribute  $A$  is given by:

$$Gini(S, A) = \sum_i \frac{|S_i|}{|S|} \cdot Gini(S_i) . \quad (2.9)$$

Finally, the attribute with the minimum average *Gini* is selected for splitting at a node of the tree.

However, decision trees are prone to overfitting. In order to face that problem, a common strategy is to keep the decision trees small, either by limiting the growth during training, or by pruning afterwards. Pruning refers to the strategy of removing the subtree rooted at a node, and replacing the node by the leaf with the most common class affiliated with that node (Mitchell 1997).

### 2.4.1 Random Forest

A special case of decision trees are random forests. The *Random Forest* (RF) algorithm (Breiman 2001) builds an ensemble of decision trees. It uses the bootstrapping approach (Breiman 1996a) to create a subset of the training set, on which each tree is built upon. In bootstrapping, the subset is drawn with replacement, i. e., instances can be drawn multiple times. The trees are grown on this subset via a random feature selection, and not pruned afterwards. This makes the construction of a random tree much faster than a regular decision tree. Due to lack of pruning, single trees do not generalize well. However, the ensemble is usually large (100 or more trees), and the ensemble decision is conducted via majority voting, which leads to better overall generalization. Random forests give very good predictions with little hyperparameter optimization, hence making them a good baseline approach for many applications. They often outperform regular decision trees, while simultaneously being faster to train. They can be used for both classification and regression tasks.

## 2.5 Artificial Neural Networks

The final paradigm in handling classification and regression tasks that we want to elaborate on are *Artificial Neural Networks* (ANNs). Neural networks, especially *Deep Neural Networks* (DNNs) (Goodfellow, Bengio, and Courville 2016; Geoffrey E Hinton and Salakhutdinov 2006) with multiple hidden layers have become very popular in recent years because of their ability to handle complex tasks like image classification and segmentation. However, as opposed to rule learners, they cannot be interpreted by humans, which makes it difficult to apply them in certain scenarios, such as medical or security-relevant applications.

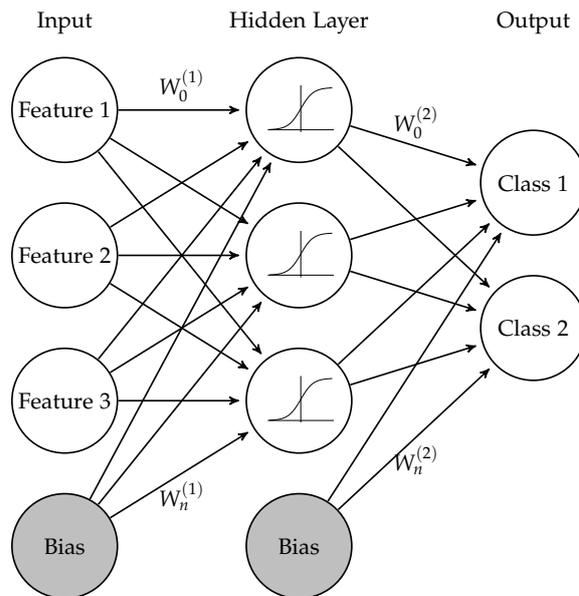


Figure 2.6: Feed-forward neural network for binary classification.

ANNs are loosely inspired by nature, in the way that they consist of consecutive layers of artificial neurons. An example of a neural network is shown in Figure 2.6.

These layers are ordered in a feed-forward fashion, so that information flow in the network is directed only from the input towards the output at prediction time. In these so-called feed-forward networks or *Multilayer-Perceptrons* (MLPs) (Rosenblatt 1958), all nodes from a layer are connected with all nodes of the following layer (also called fully-connected) via a weighted connection. There are no connections between nodes of the same layer. These properties make ANNs computational graphs.

The input layer of such a network consists of the attributes of the data instances, and the nodes in the output layer correspond to the prediction target. In case of classification, this layer represents the classes, in case of regression it represents a single or multiple target values.

ANNs only accept numerical input, so all categorical attributes need to be encoded as numbers. Usually, categorical values are encoded via "one-hot-encoding", which encodes a categorical attribute with  $k$  values into  $k$  binary attributes, where only one attribute is set to 1 and the rest to 0.

The layers between the input and the output are referred to as *hidden layers*, and it can be shown that a single layer is sufficient to approximate arbitrary continuous functions (Hornik 1991). Each node in the network receives the weighted values of all its predecessors plus an additional bias term as inputs. It then applies an activation function on the sum of these inputs. This is the computational equivalent to the *firing* process of a natural neuron. During the training phase of the

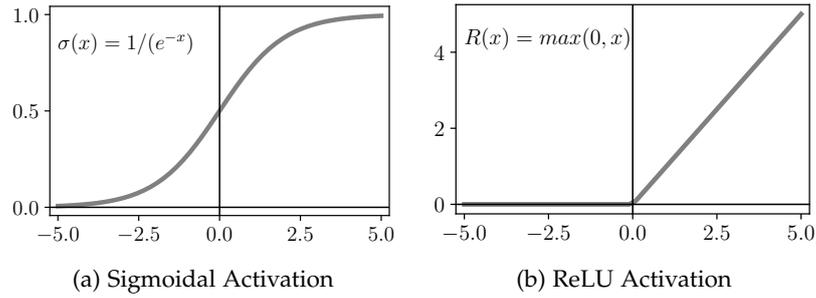


Figure 2.7: Activation Functions for Artificial Neurons.

network, the weights and the biases for each connection and neuron are learned by the training algorithm.

A layer where each node is connected to all nodes of the previous layer is called *fully-connected*. Besides from fully-connected layers, there are special layers that execute different types of operations, such as spatial convolutions. We will elaborate on convolutional layers in the next section. Networks that consist only of fully-connected layers are called *Fully Connected Neural Networks* (FC-NNs)

In order for the network to be able to learn non-linear functions, the activation function must be non-linear. Historically, functions such as the sigmoid (Figure 2.7a) or tangens hyperbolicus (tanh) were used as activation functions. More recently, simpler functions like the *Rectified Linear Unit* (ReLU) have proven to be more successful, especially in DNNs (Figure 2.7b).

Classification in a neural network is usually conducted by assigning the network  $|C|$  output neurons, where  $C$  is a set of target classes. If the target values are encoded with the one-hot-encoding scheme, the network's numerical output can be interpreted as probabilities for each class. This means that the true class should receive a probability of 1 and the other classes a probability of 0. Therefore, we need to ensure that the activations of the output nodes add up to 1, which is commonly done using the Softmax function:

$$\text{Softmax}(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}} \text{ for each } j \text{ in } 1, \dots, K. \quad (2.10)$$

The Softmax function converts a  $K$ -dimensional vector  $\mathbf{x}$  into a  $K$ -dimensional vector  $\text{Softmax}(\mathbf{x})$  which ranges from  $0 \dots 1$ , such that all  $\text{Softmax}(\mathbf{x})_j$  add up to 1.

In order to obtain the correct weights between neurons in the network, the backpropagation algorithm (Rumelhart, Geoffrey E Hinton, and Williams 1986) is used. It leverages the model error (or loss) that occurs after a forward pass of an instance through the network. The model loss is the difference between the empirical distribution defined by the training set and the distribution defined by the model, and the goal is to minimize the loss.

In the case of classification, the cross-entropy loss function  $CE$  is popular. For binary classification problems ( $c \in \{0, 1\}$ ), the cross-entropy loss for a set of examples  $X = (\mathbf{x}_1, \mathbf{y}_1) \dots (\mathbf{x}_n, \mathbf{y}_n)$  is given as:

$$CE(X) = - \sum_{i=1}^n y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i) \quad (2.11)$$

where  $\hat{y}_i$  is the predicted probability of  $\mathbf{x}_i$  being of class 1. The cross-entropy can also be used in a multinomial setting for multi-class classification.

Regression problems require different loss functions, such as RMSE (Equation 2.4) or MAE (Equation 2.5).

During the training process, the goal is to find a configuration of network weights and biases that minimize the overall loss on the training set. This is achieved via gradient descent. The derivatives of the loss function provide the information on how to adjust weights and biases. They are updated by propagating the calculated loss backwards through the network, adjusting the weights at each step as shown in Equation 2.12.

$$w_{t+1} = w_t - \eta \frac{dLoss}{dw} \quad \text{and} \quad b_{t+1} = b_t - \eta \frac{dLoss}{db}. \quad (2.12)$$

The hyperparameter  $\eta$  is the learning rate and determines the size of the gradient update. The chain rule is utilized to compute the derivatives of the loss function effectively. The ideal solution is a global minimum in the non-convex error-plane of the loss function. The learning rate helps to avoid overshooting such a minimum, as shown in Figure 2.8. Adjusting the learning rate iteratively is a common practice to allow for faster approximation and the avoidance of overshooting.

With the chain rule for derivatives, commonly computed parts of the total derivative can be separated and stored. This results in a significant reduction of computational complexity, since partial derivatives can be reused and are not computed redundantly (Rumelhart, Geoffrey E Hinton, and Williams 1986).

The gradient descent approach only guarantees convergence to a local optimum. Convergence to the global minimum of the loss function is uncommon for models with a large amount of parameters.

However, gradient descent steps are calculated on the whole dataset at once, which can be impractical in actual applications. Lecun et al. (1998) have therefore derived an online version of gradient descent, called *Stochastic Gradient Descent* (SGD). In SGD, the weights are computed iteratively on smaller sets of randomly drawn samples of the training set, so-called mini-batches. Mini-batch SGD is one of the most used algorithms for training neural networks (Goodfellow, Bengio, and Courville 2016).

More advanced optimizers include *Adam* (Kingma and Ba 2014) or *AdaGrad* (Duchi, Hazan, and Singer 2011), but most of them are extensions of SGD.

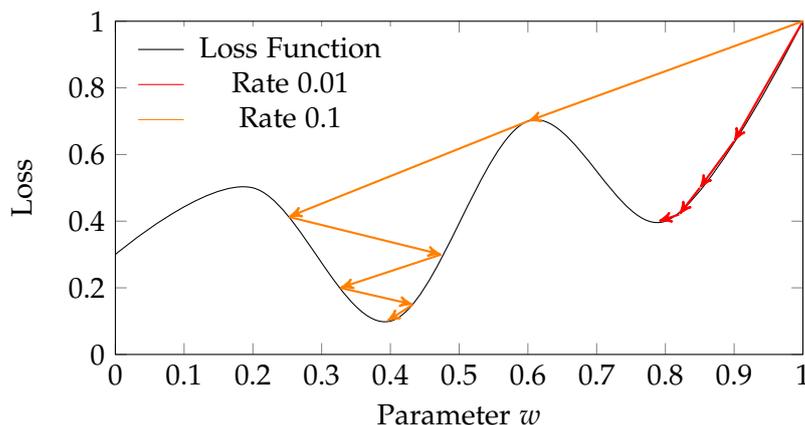


Figure 2.8: Gradient descent with different learning rates (schematic depiction).

A common issue in training neural networks is the so-called vanishing gradient problem. In the backpropagation process, when error gradients are propagated backwards through the layers of the network to update the network weights, they sometimes become very small. Through the small gradient, the step size in SGD will become near zero, which can stop the network from converging towards the optimum. Activation function such as ReLU mitigate this issue: The gradient is constant (Figure 2.7b).

## 2.5.1 Convolutional Neural Networks

In the 1960s, neuro-biologists Hubel and Wiesel (1962) discovered simple and complex cells in the visual cortex of cats. The cells are organized in so-called receptive fields, which are locally connected in such a way that they are capable of detecting complex features like edges and corners. These findings inspired computer scientists to apply similar principles on ANNs, the outcome being convolutional layers. ANNs with convolutional layers are also referred to as *Convolutional Neural Networks* (CNNs).

### Convolutional Layers

Convolutional layers were first introduced by Fukushima (1980), picking up the idea of the receptive fields. However, their popularity increased dramatically, when LeCun, Boser, et al. (1989) applied the idea on hand-written digit classification, and could show substantial improvements in the domain. In the following we elaborate on the principles of training and applying convolutional networks to provide a basic understanding for the contributions of this thesis.

Convolutional layers in ANNs resemble the locally connected receptive fields found in living beings. The idea is, that neighbouring receptors can influence each other in a way, that typical optical struc-

tures such as edges can be detected. In computer science, especially for image recognition, this is usually achieved via applying filter kernels on an image. A popular example is the Canny edge detector (Canny 1986), which was widely used in the fields of image classification and segmentation. The purpose of convolutional layers is to enable the network to learn these filters by itself via backpropagation, without having to rely on pre-calculated filter kernels.

Therefore, convolutional layers in ANNs consist of trainable filter kernels and biases. The filter slides over the input data, which can be a two-dimensional image or a time series, and generates a single output for each step: a so-called feature map. The feature map consists of scalar matrix products of the filter matrix and sliding window fractions of the input data.

A convolutional layer is characterized by following parameters:

**Number of filters:** This specifies the amount different filters to be trained. Each filter generates its own feature map.

**Filter size:** In a two-dimensional convolutional layer, the filter is usually a square. Common filter sizes are 3x3, 5x5, or 7x7. A suitable filter size depends on the size of the input data and the specifics of the given task. The filter size is the size of the window (kernel) that slides over the data.

**Stride:** Stride indicates the step size of the sliding movement. If the stride is chosen to be one, the convolution will be calculated for each pixel of an image. Higher stride can be used to reduce the resulting feature map.

**Padding:** Applying the filter kernel to the edge of an image causes the filter to reach outside the image, where no pixel values can be retrieved. The padding setting decides how this is handled, for example *zero-padding* will fill the outside values with zeros.

In comparison to fully-connected layers, convolutional layers learn the spatial relation between pixels or adjacent values. Caused by the filters they have fewer trainable parameters, which also aids in the training process of the network.

However, the output (feature map) of a convolutional layer can be large, depending on the amount of filters, filter size and especially the stride size. Usually, convolutional layers are stacked on top of each other, which leads to a hierarchical construction of increasingly complex feature detectors. In order to make a selection on which of the resulting features are the most important, pooling layers are used.

## Pooling Layers

Another discovery by Hubel and Wiesel (1962) in the visual cortex led to the second key concept of CNNs. They discovered that complex cells exhibit spatial invariance (i. e., complex receptive fields are not dependent on the exact location of an object). CNNs simulate this

behavior through pooling layers: The pooling filter combines the filter with the image and outputs the highest activation. This is called max-pooling, and it is the most common pooling technique for image applications. Max-pooling was first proposed by Zhou and Chellappa (1988), but other pooling operations, such as average-pooling, exist.

Where a fully-connected network would need to learn the exact interaction between each single pixel and its neighbors, a CNN can learn that interaction as a filter to create the spatial invariance. This makes convolutional layers much easier and faster to compute, compared to FC-NNs. Pooling layers are characterized by the following three parameters:

**Pool size:** Size of the window which shifts over the data to generate the pooling output.

**Stride:** Step size with which the pooling window moves.

**Padding:** Padding schema, equal to padding for convolutional layers.

During the max-pooling operation, the kernel shifts over the data and returns the maximum value from the fraction of the data covered by the pooling window. Max-pooling often increases the classification performance of a model due to the gain in spatial invariance.

## 2.5.2 Recurrent Neural Networks

*Recurrent Neural Networks* (RNNs) (Rumelhart, Geoffrey E Hinton, and Williams 1986) are essentially neural networks for processing sequential data. The basic idea is that they have hidden internal states that enable them to make decisions on a temporal sequence of values, as depicted in Figure 2.9. Here, a sequence  $X$  consisting of elements  $x_1 \dots x_n$  is passed through a function  $f$  that generates the content of the hidden state  $h$ . Hence,  $h$  represents a summary of previous parts of the sequence passed through  $f$ .

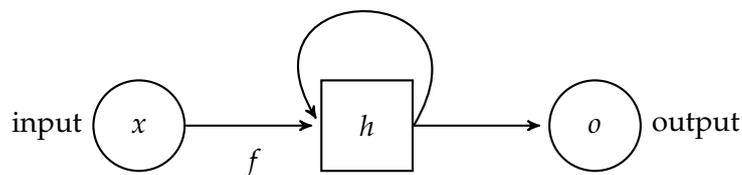


Figure 2.9: Recurrent neural network with hidden state.

The hidden state enables RNNs to be used for classification on longer sequences than would be practical for feed-forward networks with a fixed input size. Most recurrent network architectures can also process sequences of variable length.

Recurrent neural networks are usually designed for one of the following purposes:

- Produce an output at each time step for a given sequence.
- Produce a single output on the entire sequence.
- Produce a sequential output on the entire sequence.

In order to learn such a network via backpropagation, the recurrent part of the network is usually unfolded into a directed acyclic computational graph (Goodfellow, Bengio, and Courville 2016). This is conducted in such a way that the learned model always has the same input size, regardless of sequence length. Furthermore, the same transition function  $f$  can be used in every time step, avoiding the necessity to compute multiple functions for each time step. In Figure 2.10, this is shown schematically.

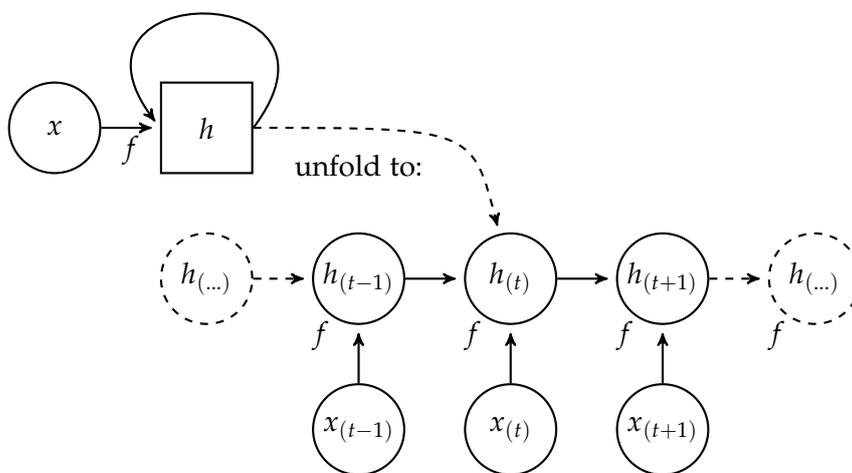


Figure 2.10: Recurrent neural network unfolding.

The vanishing gradient problem causing the network to stop learning is even more pronounced in RNNs (Hochreiter, Bengio, et al. 2001) because of the unfolding characteristic.

### Long Short-Term Memory and Gated Units

A special type of network cell that tackles the vanishing gradient problem are *Long Short-term Memory* (LSTM) units, introduced by Hochreiter and Jürgen Schmidhuber (1997). LSTM units extend a neural network unit by self-looping hidden states. The value of the hidden state is controlled by three sigmoidal gates: Two gates trigger the hidden state to be overwritten with a new value or reset. A third gate triggers the state to be propagated to the next connected neurons. This flipflop-like mechanism creates paths through the network that have derivatives that cannot vanish. A simplification of the LSTM unit is the *Gated Recurrent Unit* (GRU) by (Cho, Van Merriënboer, Gulcehre, et al. 2014; Cho, Van Merriënboer, Bahdanau, et al. 2014). It has only one gate that controls setting and forgetting the hidden state, but has otherwise very similar capabilities.

### 2.5.3 Feature Hierarchy in ANNs

One interesting feature of ANNs is the hierarchical feature representations that evolve in the hidden layers of the network during the training process.

Zeiler and Fergus (2013) have researched and visualized this process for CNNs. They trained a CNN with consecutive convolutional layers for a face recognition task and found a way to visualize the resulting kernels. A part of the results is shown in Figure 2.11.

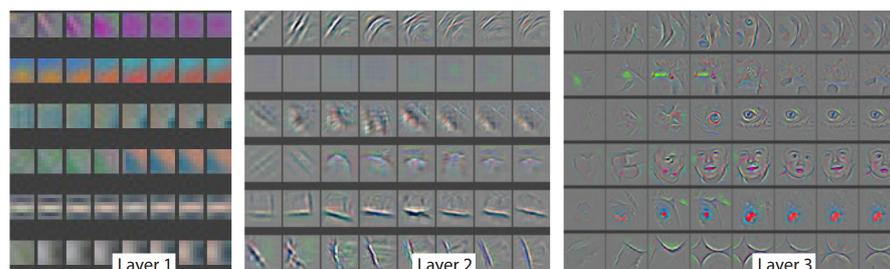


Figure 2.11: Feature hierarchy and evolution of convolutional layer features over three steps Zeiler and Fergus (2013).

The first layer of the CNN learns a variety of frequency-selective filters, orientation-selective filters and coloured blobs (Krizhevsky, Sutskever, and Geoffrey E. Hinton 2012). The next convolutional layer uses a mixture of these features to generate higher level features, producing a feature hierarchy. These higher level features are combined again in the following layer to produce even more complex features. After even more layers (the authors used 5 layers in total), the re-combined features can detect arbitrarily complex shapes, e. g., human faces or other complex shapes, depending on the classification task that is given.

For fully-connected ANNs, Yosinski et al. (2014) have made similar findings regarding the hierarchy: Apparently, earlier layers in the network contain more general features, whereas later layers contain features more specific to the classification task.

# Adaptive Machine Learning

---

Most work in *Machine Learning* assumes a stationary distribution of data. However, in real-world applications there is a significant chance that this assumption might not hold true. Hence, classification and regression models have to be derived that can incorporate such change, which we refer to as concept drift. Adaptive learning is concerned with updating predictive models to handle such a drift. This chapter elaborates on the notion of concept drift and introduces relevant adaptive algorithms that deal with it.

We start by defining concept drift in Section 3.1, followed by the online adaptive learning procedure in Section 3.2. We then go into the details about change detection (Section 3.3) and the evaluation of adaptive learners (Section 3.4).

The foundations for adaptive and transfer learning methods are introduced in Sections 3.5 and 3.6, with a focus on ensemble methods and recurring concept learning. The chapter concludes with Section 3.7, which introduces widely used datasets from this domain that we will refer to in later chapters.

## 3.1 Concept Drift

The term concept in a ML-setting usually describes the relation between the input data and the target variable. It is that concept that needs to be captured by a classification model in order to make correct predictions.

Concept drift refers to change in a concept, which may lead to misclassification by a given predictor. Its cause is a change of the distribution from which the instances are drawn. This is also referred to as *non-stationarity*.

Classification can be described by the prior probabilities of the classes  $\Pr(c)$  and the class conditional probability density function  $\Pr(\mathbf{x}|c) \forall c \text{ in } C$  according to Bayesian decision theory (Duda, Hart, and Stork 2012). A decision, if a given sample  $\mathbf{x}$  belongs to class  $c$ , can be made according to the posterior probability:

$$\Pr(c|\mathbf{x}) = \frac{\Pr(\mathbf{x}|c) \cdot \Pr(c)}{\Pr(\mathbf{x})} .$$

Hence, concept drift can be defined as any scenario where the posterior probability changes over time  $t$  (Elwell and Polikar 2011), i. e.:

$$\Pr_{t+1}(c|\mathbf{x}) \neq \Pr_t(c|\mathbf{x}) .$$

However, observing a change in the evidence  $\Pr(\mathbf{x})$  is not a sufficient indicator for a concept drift, since the class prior probabilities  $\Pr(c)$  are independent of the features. More indicative are the classes' decision boundaries. We refer to the cases where the boundaries remain the same, but the likelihood  $\Pr(\mathbf{x}|c)$  changes, as *virtual concept drift*. Scenarios with changing boundaries are usually called *real concept drift* (Gama, Žliobaitė, et al. 2014).

A comparison of both is shown in Figure 3.1. Here, virtual drift is indicated by a stable environment, while the instances are sampled from different parts of the instance space. Real drift however involves a change in the boundaries. In this case, previous knowledge about the boundaries becomes obsolete. In practice, both types of drift can often occur simultaneously.

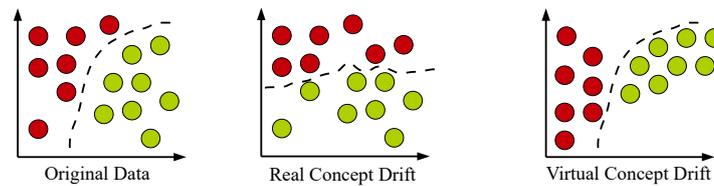


Figure 3.1: Real and virtual concept drift depicted in a two-dimensional space (Gama, Žliobaitė, et al. 2014).

A transition between concepts can occur in multiple ways. Gama, Žliobaitė, et al. (2014) have categorized four principal types of concept transitions, as shown in Figure 3.2.

**Sudden** or **abrupt** drift, is when the concept switches immediately without any intermediate step. This can be the case, for example, when a customer who has been interested in a product for a while, purchased that product.

**Incremental** drift, on the other hand, involves some sort of smooth transition between concepts, possibly containing an infinite amount of intermediate concepts. This can happen when a sensor slowly degrades and gives increasingly inaccurate readings.

**Gradual** drift describes a back-and-forth change with increasing frequency, until the target concept is finally reached. This can happen, when a persons interest in a news topic slowly fades, but they come back to the previous interest every once in a while.

**Reoccurring** drift describes concepts that re-occur, for example based on seasonal aspects.

**Outliers** are not concept drift at all, but instead noise that may be misleading. One crucial aspect in adaptive learning is not to adapt to outliers, because they do not resemble any real concept.

In order to adapt to drifting concepts, predictive models have to employ mechanisms that detect the change and trigger adaptation mechanisms. Detecting concept drift can be done via *active* or *passive drift detection* methods. *Active* drift detection employs a drift detection mechanisms that monitors either the environment or the model, whereas *passive* drift detection handles drift implicitly, by always assuming that change is happening.

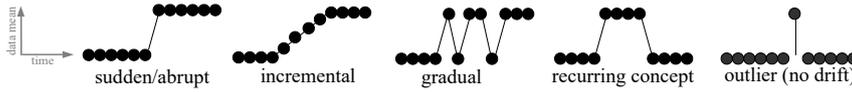


Figure 3.2: Types of concept drift (Gama, Žliobaitė, et al. 2014).

## 3.2 Online Adaptive Learning Procedure

Gama, Žliobaitė, et al. (2014) define adaptive learning as three-step process: Given a decision model  $M$  that maps the input variables to the target:  $\mathbf{y} = M(\mathbf{x})$ , a *learning algorithm* specifies how to build a model from a set of instances by the following steps:

1. **Predict.** When new examples  $\mathbf{x}_t$  arrive, a prediction  $\hat{\mathbf{y}}_t$  is made using the current model  $M_t$ .
2. **Diagnose.** After some time, we receive the true label  $\mathbf{y}_t$  and can estimate the loss as  $f(\hat{\mathbf{y}}_t, \mathbf{y}_t)$ .
3. **Update.** We can use the example  $\mathbf{x}_t, \mathbf{y}_t$  for the model update to obtain  $M_{t+1}$ .

Depending on the settings and the requirements of the learner, it can be feasible to process the update step on more than only the most recent examples. For example, we can store multiple examples, and use them for a batch-updates. One of the most common approaches is to maintain a sliding window of examples, as implemented in the *FLORA* algorithms by Widmer and Kubat (1996). Sliding window approaches with fixed and variable size are common. The latter requires active drift detection to appropriately adjust the window.

## 3.3 Change Detection

A change detection method is helpful for any algorithm that mitigates concept drift and requires a trigger mechanism. Not all adaptation mechanisms require explicit change detectors, but for those who do, the change detector plays a crucial role. Change detection methods characterize and quantify concept drift, and identify *Change Points* (CPs) or intervals (Basseville and Nikiforov 1993). In this thesis, we are dealing with explicit drift detection, and most relevant to our case are drift detection methods based on (i) control charts, and (ii) differences between distributions. Typically, in order to detect concept drift, the change detectors monitor the performance of the given system (Widmer and Kubat 1996), and suggest a drift when the performance degrades.

### 3.3.1 Control Charts

A set of well-established drift detection methods are based on *Statistical Process Control* (SPC) (Klinkenberg and Renz 1998; Lanquillon 2001; Gama, Medas, et al. 2004; J. B. Gomes, Menasalvas, and Sousa 2011), where a sequence of examples  $S = (\mathbf{x}_i, \mathbf{y}_i), i \in 1 \dots N$  is classified. The model's prediction  $\hat{\mathbf{y}}_i$  is either *correct* ( $\hat{\mathbf{y}}_i = \mathbf{y}_i$ ) or *incorrect* ( $\hat{\mathbf{y}}_i \neq \mathbf{y}_i$ ) for each example, which constitutes a set of Bernoulli trials. From these sequence of trials we can estimate the probability of  $n$  errors (in this set of  $N$  examples) via the binomial distribution. When observing the sequence  $S$ , we can calculate the probability  $p_i$  of observing *incorrect* for each element  $S_i \in S: p_i = (\text{error}_i / i)$ , and the standard deviation is given by

$$\sigma_i = \sqrt{\frac{p_i(1 - p_i)}{i}}$$

The SPC algorithm tracks the minimum probability  $p_{min}$  and minimum standard deviation  $\sigma_{min}$ , and compares it against the current values  $p_i$  and  $\sigma_i$ . If  $p_i + \sigma_i < p_{min} + \sigma_{min}$ , the values are updated:

$$p_{min} = p_i, \quad \sigma_{min} = \sigma_i$$

After a sufficiently large number of observations the binomial distribution is approximated by the normal distribution with the same mean and variance. If we assume that, in a non-drifting scenario, the probability should not change, we can use the  $2 \cdot \sigma$  and  $3 \cdot \sigma$  confidence intervals to establish *warning* and *out-of-control* states, where *out-of-control* resembles a significantly increased error with a probability of 99%, whereas *warning* sets the threshold to 95%. Gama, Medas, et al. (2004) call this procedure the *Drift Detection Method* (DDM).

Even if DDM has sufficient capabilities detecting abrupt or fast gradual changes, it has difficulties when the change is slow and gradual. That problem is addressed by the *Early Drift Detection Method* (EDDM) proposed in Baena-Garcia et al. (2006).

Here, the distance  $p'_i$  between two errors and its standard deviation  $\sigma'_i$  is considered instead of the number of errors. The warn level is reached if

$$(p'_i + 2 \cdot \sigma'_i) / (p'_{max} + 2 \cdot \sigma'_{max}) < \alpha$$

and the drift level at

$$(p'_i + 2 \cdot \sigma'_i) / (p'_{max} + 2 \cdot \sigma'_{max}) < \beta,$$

where  $\alpha$  and  $\beta$  are application-dependent hyperparameters. Similar approaches, such as the *Exponential Weighted Moving Average* (EWMA) by Ross et al. (2012), advance on the idea of monitoring distributions. The EWMA downweights the influence of older examples, and calculates an estimate of the error rate on the weighted examples.

### 3.3.2 Distribution Monitoring

Another way of approaching the change detection problem is via distribution monitoring. The idea of these methods is to monitor and compare the error distribution in two separate time windows, e. g., one long-term window and one short-term window. A significant change in the short-term window opposed to the long-term window indicates a recent concept drift.

A popular example for such a method is the *Adaptive Sliding Window* (ADWIN) approach by Bifet and Gavaldà (2007). It compares two windows of data based on their expected values. Whenever two large enough subwindows  $W_0$  and  $W_1$  of a sliding window  $W$  exhibit sufficiently distinct averages, i. e.,  $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \geq \epsilon_{cut}$ , a change in the environment can be concluded. The older portion of the window is then dropped. The cutting threshold  $\epsilon_{cut}$  is computed as

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot \sigma_W^2 \cdot \ln \frac{2}{\delta'}} + \frac{2}{3m} \ln \frac{2}{\delta'}$$

which is an adapted version of the *Hoeffding bound*, specialized for the normal distribution that is assumed for large window sizes. Here,  $\sigma_W^2$  is the observed variance of the elements in the window  $W$  and  $m$  the harmonic mean between the subwindow lengths  $n_0$  and  $n_1$ .

In this case,  $\delta$  represents the confidence value. However, since  $n$  different combinations of subwindows need to be evaluated, where  $n$  is the length of the window  $W$ , the confidence value gets adapted to  $\delta' = \frac{\delta}{\ln n}$  in order to avoid multiple hypothesis testing problems.

The *Hoeffding Drift Detection Method* (HDDM) (Frias-Blanco et al. 2015) applies statistical tests based on the moving average of two windows. For constructing those windows from a stream of samples, Hoeffdings inequality for two-sample statistical testing (Hoeffding 1963) is used, which considers the difference between averages. In contrast to other drift detection methods, HDDM does not rely on any assumptions related with the probability density function that generates the measured values.

Another example for distribution monitoring is the method by Dasu et al. (2006), which uses the *Kullback-Leibler* (KL) divergence to determine the difference between distributions of different windows.

## 3.4 Concept Drift Learner Evaluation

Like stationary ML, adaptive ML also requires performance metrics and sound evaluation procedures. In the domain of concept drift learning, we are mainly concerned with streams of data. These streams incorporate the change in a sequential manner, which prohibits us to apply stationary methods such as *k-fold* cross validation, since no randomization or splitting of the data is possible without disturbing the natural order.

The main goal in proper evaluation is to give an estimate of the predictive performance of a model. In this case, the performance over time, and especially the performance when it is dealing with concept drift. A classifier may perform well at the beginning of a stream, but then deteriorate after one or multiple concept changes have happened. Usual metrics such as accuracy might be misleading, since they do not take stream properties into account.

### 3.4.1 Interleaved Test-Then-Train Procedure

One of the most common testing procedures in stream settings is the *Interleaved Test-Then-Train* or *Prequential* method. It can be executed on single instances as well as on batches of instances. The general idea is, that each instance is used to test the model before it can be used to enhance the model in the update step (Section 3.2). Therefore, the model will always be tested on previously unseen instances, which has the advantage that no separate hold-out set is needed. Furthermore, we can obtain accuracy readings over time, which will lead us to an accuracy plot as shown in Figure 3.3. The evaluation can be conducted in three styles: (i) landmark window style, where consecutive batches of data are evaluated separately, (ii) sliding window, where the most recent  $w$  instances are evaluated and (iii) via a forgetting mechanism that emulates a hold-out set (Gama, Sebastião, and Rodrigues 2013). In this thesis, we work in a batch setting, so we are always applying the landmark style prequential evaluation.

### 3.4.2 Evaluation Metrics

When encountering a data stream with concept drift, the model is going through different phases as shown in Figure 3.3. In the first phase (the *Base*-phase), the model operates at its optimal capability and no drift has occurred yet. Then, a concept drift occurs and the model enters the *Adaptation*-phase. The model's performance drops and it starts adapting to the new concept. Finally, there is the *Finish*-phase, where the model has re-gained some level of performance that resembles its optimal capability. In cases where the model is trained from scratch directly on the data stream, these phases are preceded by an *Initialization*-phase, which is similar to the *Adaptation*-phase.

There are three potential aspects that we want to measure when comparing different methods:

1. Adaptation speed – The time required to recover from an abrupt concept drift.
2. Overall adaptation capabilities – Algorithms overall capability to adapt to change.
3. Resources required to compute the adaptation – Computation and memory requirements of the algorithm.

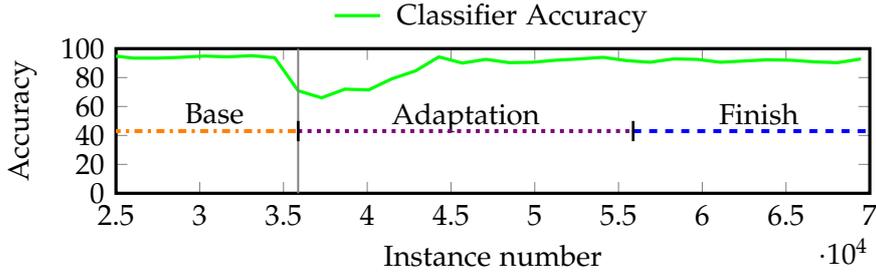


Figure 3.3: Phases during the course of the stream. CPs are shown as grey vertical lines.

First, the speed at which a model is able to adapt to a new concept is often the most important aspect. The fewer training examples are required to recover to previous levels of accuracy, the better. Secondly, the model’s overall capabilities w. r. t. adapting to the new concept are important. If the model cannot properly approximate the desired function, it will not be satisfactory in the long run. In some circumstances resources are constrained, so the computational power or memory may be limited. However, we recognize the fact that even the smallest systems today become more and more powerful, so this is not one of our main concerns in this thesis. This leaves us with the following metrics to measure adaptation performance:

**Final Accuracy (F.Acc.):** Classification accuracy, measured in the *Finish*-phase of the data stream, when no more concept drift occurs. This measurement represents the overall capabilities of the model to encompass the underlying concept.

**Recovery Speed (R.Spd):** The number of instances or batches that a classifier requires during the *Adaptation* phase to achieve 95% of its final accuracy. The fewer examples a model requires to adapt, the faster the adaptation.

**Adaptation Rank (Ad.Rk):** Average rank of the classifier during the *Adaptation* phase. This is used to compare different methods against each other. We rank the methods for all the steps in the *Adaptation*-phase, and compute the average rank to compare them.

**Final Rank (F.Rk):** Average Rank of the classifier during the *Finish* phase, also used for comparison. We rank the methods for all the steps in the *Finish*-phase, and compute the average rank.

### Statistical Evaluation

For testing statistical significance we rely on the methods described in Section 2.2.4. We will compare different methods based on their adaptation rank or final rank, in order to show performance differences.

## 3.5 Adaptive Learning Methods

In this section we give an overview to work regarding concept drift and transfer learning, since the task of adapting a classifier can also be used in a setting where knowledge transfer is required. At first we discuss seminal and related work in the domain of concept drift learning, followed by ensemble methods for concept drift.

### 3.5.1 Concept Drift Learners

Schlimmer and Granger (1986) have introduced the *STAGGER* method for incremental learning in a binary classification setting. *STAGGER* maintains an ensemble of weighted concept descriptions. These descriptions capture combinations of feature representations with respect to the classes, similar to classification rules. The descriptions are represented in the form of conjunctions, disjunctions, and negation. The concept description weights are continuously updated, so that in case of a stable concept, the classification is improved. When concept drift occurs, new descriptions are added to the ensemble to incorporate the drift. In their work, Schlimmer and Granger introduce a data set which consists of geometrical shapes represented by categorical features. It has since become one of the most popular datasets in concept drift learning (Tsymbal 2004), although it can be considered very simple, nowadays.

Widmer and Kubat (1992) proposed the *FLORA*-Framework for incremental rule learning. It is based on the theoretical work by Kubat (1989). The main idea is centered around keeping a window of current examples and hypotheses, simultaneously storing concept descriptions in a re-usable way, and controlling both via heuristics that monitor the system behavior and are responsible for the adaptation.

The *FLORA*-Algorithm processes a stream of data by applying a sliding window, from which the concept descriptions are derived. They are updated with each new instance by removing the oldest example, when a new example arrives. *FLORA2* extends this by automatically choosing an appropriate window size.

Besides these rule and context-description based approaches, decision trees are also suited to incorporate non-stationary learning. The *Very Fast Decision Tree* (VFDT) algorithm proposed by Domingos and Hulten (2000) builds an incremental decision tree (a *Hoeffding Tree*), which uses constant memory and constant time per sample. *Hoeffding Trees* are based on the idea of the Hoeffding bound (Hoeffding 1963). They were constructed for incremental learning, which makes them popular in the domain of concept drift. Additional examples of their use can be found in (Hulten, Spencer, and Domingos 2001; Bifet and Gavaldà 2009).

The VFDT implementation extends the general *Hoeffding Tree* approach by modifications for speed-up, which is one of the main requirements in online learning scenarios with high frequency of incoming instances.

### 3.5.2 Ensembles in Concept Drift

While the algorithms in the previous section are geared towards solving concept drift learning, in more recent years, many ensemble methods with the same goal have emerged.

A seminal approach that maintains an ensemble of classifiers for dealing with concept drift is the *Accuracy Weighted Ensemble* (AWE) by Wang et al. (2003). The method uses the data stream in a batch-wise manner. A single classifier is trained for each batch of data. To achieve an ensemble decision, these classifiers are combined via weighted voting, and the weights are determined on the most recent batch based on the error rate. This algorithm keeps a maximum of  $K$  classifiers, more specifically, the top  $K$  classifiers based on accuracy.

The *Accuracy Updated Ensemble* (AUE) (Brzezinski and Stefanowski 2011) builds on AWE and also adds one classifier for each new batch of data. However, instead of using only the existing members' weights for a decision, classifiers can be updated or deleted based on the most recent data.

Similar to AWE, the *Dynamic Weighted Majority* (DWM) approach by Kolter and Maloof (2007) maintains a weighted ensemble of classifiers, but with a flexible ensemble size. New classifiers are added if the ensemble errs on the most recent data. In addition, members are removed if their weight falls below a certain threshold.

Oza (2005) proposes online versions of the well-known ensemble learning algorithms *Bagging* (Breiman 1996b) and *Boosting* (Freund and Schapire 1996). Online Bagging trains each of the ensemble members  $k$  times on a new sample from the incoming data stream, where  $k$  is drawn from a Poisson distribution with  $\lambda = 1$ . The first classifier in Online Boosting (*OzaBoost*) is trained in the same way. For each following data batch, the existing ensemble members are evaluated on the most recent data and  $\lambda$  is updated according to the correctness of the prediction.

The *Learn<sup>++</sup>* algorithm by Elwell and Polikar (2011) trains a neural network on each batch and assigns a performance based weight to all ensemble members. In contrast to other accuracy weighting approaches, each member stores its error on all previously evaluated batches. A sigmoidal weight function is applied to decrease the influence of older batches, which can be adjusted via hyper-parameters. Ensemble members are not deleted and each of them can receive a high weight independent of age, which enables the method to adapt to recurring concept drift.

The final ensemble algorithm that we would like to mention in this section is the *Fast Adaptive Stacking of Ensembles* (FASE) by Frías-Blanco et al. (2016). It maintains a fixed-size ensemble of adaptive classifiers trained by *Online Bagging* (Oza 2005). Each adaptive classifier is paired with an active drift detector that signals a warning before a drift. If a warning is signaled, a second instance of the classifier starts training in parallel and the predictions of both are combined by weighted voting based on the error rate. Once a drift follows the warning, the old classifier is deleted. In order to combine the predictions of all adaptive classifiers, an additional classifier is trained that receives the predictions as input and returns the final ensemble decision. Every time the system detects a drift, the worst performing ensemble member is deleted and a new one added.

### 3.5.3 Recurring Concept Learning

Finally, we would like to point out some related work on recurring concept drift, i. e., algorithms that were specifically engineered towards handling recurring concepts, for example, to deal with seasonal effects.

The seminal algorithm designed for recurring concepts is *FLORA3* by Widmer and Kubat (1993). Contrary to *FLORA2* (Section 3.5.1), it can restore old hypotheses that fit the current concept.

Most of the recent approaches on handling recurrent concept drift are based on ensembles of classifiers.

The *Adaptive Classifier Ensemble* (ACE) system proposed in (Nishida, Yamauchi, and Omori 2005) consists of an online learner, multiple batch learners, and active drift detection. Each arriving example updates the online learner and is stored in a buffer. If the buffer reaches a maximum capacity or a drift is detected, a new batch learner is trained on the stored data. The final hypothesis is composed of a weighted majority vote between all learners. Hence, previously learned concepts can be weighted higher, to adjust for recurring concepts.

The *Conceptual Clustering and Predictions* (CCP) framework by Katakis, Tsoumakas, and Vlahavas (2010) dynamically creates an ensemble of classifiers, where each classifier handles a single concept. Each batch of data is mapped into a vector of predefined features such as the mean and standard deviation of a numerical attribute from all samples that belong to the same class. Concepts are then defined by clustering the vector space, and training one classifier per cluster. When a new batch of instances arrives, the cluster is identified by the concept classifier, and the corresponding classifier is applied.

The ensemble method proposed by Gama and Kosina (2014) learns one classifier per concept. An additional meta learner predicts the classifier's performance on a given batch of data. The meta classifiers are trained on the original features, but with binary labels that indicate whether their corresponding learner is able to correctly classify a

sample. If an active drift detection method signals a concept change, the current classifier and its corresponding meta-learner are stored. Afterwards, the meta-learners from previously learned concepts are asked if their classifier can perform well on the new data. If none of them are suitable, a new pair of classifiers is created.

## 3.6 Transfer Learning

Transfer learning is concerned with the problem of applying models from a known source domain to a target domain. The target domain can be similar, but it is not necessarily from the same distribution. For example, when the source task was to identify apples, a part of that knowledge may come in handy when the target is to classify oranges. Multi-task learning (Caruana 1997) aims for similar goals, but transfer learning differs from multi-task learning in such a way, that its primary optimization goal is the target task, and not all of the tasks simultaneously. Pan and Yang (2010) give an excellent overview on transfer learning, and formally define it as follows:

Given a source domain  $\mathcal{D}_S$  and a learning task  $\mathcal{T}_S$ , a target domain  $\mathcal{D}_T$  and a learning task  $\mathcal{T}_T$ , transfer learning aims to help improve the learning of the target predictive function  $f_T(\cdot)$  in  $\mathcal{D}_T$  using the knowledge in  $\mathcal{D}_S$  and  $\mathcal{T}_S$ , where  $\mathcal{D}_S \neq \mathcal{D}_T$ , and  $\mathcal{T}_S \neq \mathcal{T}_T$ .

The definition implies, that when  $\mathcal{D}_S = \mathcal{D}_T$  and  $\mathcal{T}_S = \mathcal{T}_T$ , the problem is a not a transfer learning problem, but a traditional machine learning problem. Hence, the following combinations are regarded as transfer learning problems:

1. **Domain difference.** The feature spaces between the domains, or the probability distribution between the domains can differ.
2. **Target difference.** Similar to the domain, for the target the label spaces, or the probability distributions between the domains can be different.

So the main questions in transfer learning are: (i) *what* to transfer, (ii) *how* to transfer, and (iii) *when* to transfer. The first is concerned with the determination, which aspects from the source domain can help to advance the solution in the target domain. This directly leads to the second aspect: finding appropriate algorithms and methods which can do exactly this. Finally, it should be clearly determined that a transfer is actually useful, and aids in solving the problem. In some situations, this may not be the case. Most current research, however, is concerned with (i) and (ii), although situations exist where transfer has a negative impact, the so-called *negative transfer*.

Pan and Yang (2010) provide a categorization of transfer learning settings:

- **Inductive transfer.** Here, the target task differs from the source, regardless of source and target domain. This scenario requires

some labeled data from the target domain in order to *induce* a predictive model.

- **Transductive transfer.** In this setting, source and target task are equal, but the domains differ. Labeled data for the target domain exist in large amounts.
- **Unsupervised transfer.** In the third setting, the target task is different to the source task, but they are related. However, no labeled data exist, making this type of learning unsupervised.

In this work, we are mostly concerned with the inductive transfer learning scenario with equal source and target domains. This type of transfer is similar to concept drift learning, when the target task is completely different from the source task, but the features remain the same.

## 3.7 Datasets

Throughout this thesis we use different datasets for empirical evaluation. Most of them are also covered in related work and can be seen as "standard" for the domain. However, in order to make more precise observations how a classifier handles certain types of drift, we used existing datasets and introduced concept drift to them. In this section we will describe the datasets roughly, more precise definitions of how they were used and altered are given in the respective experimental sections.

### 3.7.1 SEA Concepts

The SEA concepts dataset introduced by Street and Kim (2001) is a dataset with abrupt concept drift consisting of three attributes, where the attributes are used to generate the resulting binary label. All three attributes are real-valued and have random values between 0 and 10. The class is set via a function that relates the attributes to the class: A threshold value  $\theta$  decides the class of an instance, combined with a certain amount of class noise. One of the used functions is for example a linear combination of the attributes.

This allows us to create drifting streams of data with different relations between the attributes and the classes, where the threshold changes in the course of the stream.

### 3.7.2 Rotating Hyperplane

The second dataset we use is based on a rotating hyperplane in a  $d$ -dimensional space as proposed by Hulten, Spencer, and Domingos (2001), with which a binary stream classification problem can be constructed. The hyperplane is defined by the set of points  $x$  that

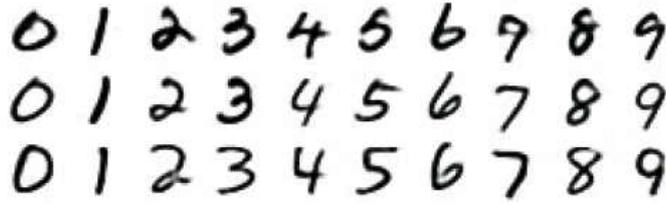


Figure 3.4: The *MNIST* dataset of handwritten digits.

satisfy  $\sum_{i=1}^d w_i x_i = w_0$  where  $x_i$  is the  $i$ -th attribute of  $x$ . The class for each instance is determined as follows: If  $\sum_{i=1}^d w_i x_i \geq w_0$ , the class is *positive*, otherwise it is *negative*. By changing the weights  $w_i$ , the orientation and position of the hyperplane can be changed.

### 3.7.3 20 Newsgroups

Another dataset that we use in our evaluation is *20 Newsgroups*<sup>1</sup>. This dataset consists of newsgroup entries and contains two hierarchical category attributes, one being the top-category and another being the subcategory. We derive a transfer learning task from this dataset, such that the goal is to classify the top-category. Therefore, we vectorize the text part of the dataset and split it based on the subcategories. This means, that if  $A$  and  $B$  are top-categories, and each has two subcategories  $A_1, A_2$  and  $B_1, B_2$ , we split it so that  $A_1$  and  $B_1$  are in the training set, whereas  $A_2$  and  $B_2$  are put in the test set. Therefore, the training and test sets are made up from different subcategories and will show a varying distribution.

### 3.7.4 MNIST and NIST

The final and most important dataset is the *MNIST*<sup>2</sup> dataset of handwritten digits (Figure 3.4). *MNIST* consists of a total of 70,000 digits. We randomize the order, use it as a stream and introduce different types of change. For example, we alternate the pixel values themselves, e. g., by rotating and flipping the images, or we modify the classes of the numbers, so that classes appear/disappear during the stream, or certain numbers are labeled with a different class.

The *NIST*<sup>3</sup> dataset contains 810,000 digits and characters, to which we apply similar transformations as to the *MNIST* data. Contrary to *MNIST*, *NIST* items are not pre-aligned, and the image size is 128x128 pixels. We use all digits 0-9 and upper-case characters A-Z for a total of 36 classes as data stream and draw a random sample for each scenario.

1 <http://www.iesl.cs.umass.edu/data>

2 <http://yann.lecun.com/exdb/mnist/>

3 <https://www.nist.gov/srd/nist-special-database-19>



# Classifier Patching

---

In this chapter we present classifier patching—or short, *Patching*—an approach for adapting an existing black-box classification model to new data.<sup>1</sup>

Instead of creating a new model, patching infers regions in the instance space where the existing model is error-prone, by training a classifier on the previously misclassified data. It then learns a specific model to determine the error regions. This allows to patch the old model’s predictions for them. *Patching* relies on a strong, albeit immutable, existing base classifier, and the idea that the true labels of seen instances will be available in batches at some point in time after the original classification. We experimentally evaluate our approach, and show that it meets the original design goals.

Furthermore, we compare our approach to existing methods from the domain of ensemble stream classification in both concept drift and transfer learning situations. *Patching* adapts quickly and achieves high classification accuracy, outperforming state-of-the-art competitors in either adaptation speed or accuracy in many scenarios.

The structure of this chapter is as follows. Section 4.1 gives a motivation on a the research problem, followed by a short explanation of the learning scenario in Section 4.2. In Section 4.3, we formalize the problem, explain the patching approach, and relate it to previous work in concept drift and transfer learning in Section 4.4. We then describe our evaluation scenario and briefly explain the datasets and algorithms we compare our approach to (Section 4.5). The experimental results are summarized in Section 4.6. We discuss our approach in Section 4.7, and give a conclusion in Section 4.8.

## 4.1 Introduction

In practical applications, one occasionally faces a scenario where a given classification model needs to be adapted to a changing environment, but neither can the model itself be modified nor can it be re-trained because the necessary expert knowledge or training data are not available.

For example, consider the need for *adaptation of legacy or expert models*, which have often been developed and successfully deployed over decades, so that the required expertise for re-programming them is

---

<sup>1</sup> This chapter is based on Kauschke and Fürnkranz (2018), where this work was published at the AAAI conference in 2018.

no longer available. Moreover, they may be implemented in hardware, making it infeasible and impractical to re-program the entire system.

Another application scenario is the *specialization of universal models*. Often, classification models have been developed for a general setting, but need to be refined to a particular context. A typical case is the personalization of a general user model to an individual user. The general model is trained on historical data gathered from many users, and therefore gives a good approximation of the average person's behavior. However, when taking into account the specific preferences, needs or abilities of a single person, it is possible to improve the model's predictions. Of course, the adaptation should only occur where necessary and helpful, and should in general not hamper the performance of the original model.

A last example scenario is *efficient model re-use*. In deep learning, large amounts of data and computational power are needed to train a model. For pattern recognition, such models are then often re-used in slightly different contexts by taking the first layers of a generally trained image classification network and re-training only the final layer for a new classification task. However, this technique is specific to a neural network architecture, whereas we aim for a generally applicable method.

In order to solve these or similar issues, we propose classifier patching, a technique which allows to adapt general black-box classification models to a new context. The key idea behind this approach is to train classifiers that identify the regions of the instance space in which adaptations are needed, and then training local classifiers for these regions. We assume a setting where batchwise labels of incoming examples are available.

The adaptation is triggered when a decaying performance of the base model is detected, with the goal of finding local patches to the global classification model that act in a flexible and efficient way without having to re-train the model from scratch.

## 4.2 Data Stream Learning

In most machine learning settings, we are dealing with data from a stationary, but unknown distribution. In these settings, the learning is performed on the existing data, with no need to update the model after the training process. A generalization of these settings allow for the existence of *concept drift*, a change or variation of the underlying concept that determines the target of the learning task.

Classification algorithms that deal with data streams are usually engineered to handle such non-stationarity, either by implicitly or explicitly detecting the concept drift, and handling it via a coping mechanism. In Section 3.1, we introduced different types of concept drift: *gradual*, *abrupt*, *incremental* and *recurring*, all of which can be handled by explicit drift detection. However, depending on the speed of the drift, it may be difficult to detect a slow incremental drift, if the

drift detection algorithm is geared towards detecting abrupt drift, and vice versa.

### 4.2.1 Instance-wise Stream Learning

An important concern for learning with data streams is the availability of the ground truth, i. e., the true labels. The minimum requirement for supervised learning in a stream is, that the labels become available at *some point* in the future. This could be, for example, after an expert has had the time to manually inspect the instances and label them, or via a mechanical turk mechanism or similar. Some scenarios have the advantage that labels are available almost immediately. A common assumption in data stream classification is, that the label of an instance  $x^t$  at a given time  $t$  is available before the arrival of instance  $x^{t+1}$  (cf. Oza 2005; Bifet and Gavaldà 2009; Bifet, Holmes, et al. 2010; Kosina and Gama 2015; Gama, Žliobaitė, et al. 2014). This is often paired with the implication that instances arrive at a very high frequency, and the learner is required to handle them immediately, meaning that he can only learn from the instance once, and not store it for later processing. An example of such a high-speed algorithm is the *Very Fast Decision Tree* by Domingos and Hulten (2000).

### 4.2.2 Batch-wise Stream Learning

However, in many scenarios instance labels do not arrive one by one, but in batches. For example, when trying to learn to predict a machine status, data annotation usually takes place afterwards, when a (new) problem with the machine was observed. However, the predictor works in real-time for the existing machine. Another example would be, that—after adapting the classifier—some sort of verification or sanity check has to be executed, in order to assure flawless operation of the predictor. This could be the case in any scenario which has implications on the safety of people. These examples show that high frequency classifier updates are not relevant to some scenarios: Instead, the model is updated offline and exchanged when ready. This allows the use of more traditional machine learning methods, but also creates new challenges. For example, when we assume there is an abrupt concept drift in a batch of data, it becomes important to determine exactly where that drift happened, in order to only update the classifier with data that was observed after the drift. Otherwise, the update-step will result in an update towards the old concept and may decrease the performance. Furthermore, there could be multiple drifts in a large batch, which may make the update even more complicated. Splitting a large batch into multiple adequately sized ones can be a solution in this case. On the other side of the spectrum, small batch sizes may have not enough examples to account for an ideal learning process, so gathering more examples before an update is a possible solution. The

required batch size is therefore always dependent on the classification problem and its complexity.

### 4.2.3 Our Goal

The approach that we are presenting in this chapter is mainly geared towards the domain of concept drift and stream learning. However, it is also applicable to transfer learning situations as described in Section 3.6, especially inductive transfer with equal source and target domains. In this work, we therefore relate to specific inductive transfer learning experiments that were conducted in (Dai et al. 2007; Gao et al. 2008; Pan and Yang 2010) for comparison.

In the following, we describe our batchwise approach to the data stream learning with non-stationarity and transfer learning problems.

## 4.3 Patching Classifiers

The *Patching* framework is especially geared towards leveraging an existing black-box classifier, the idea being that in scenarios with concept drift, it can often be the case that only parts of the existing concept changes, and others may be completely unaffected. In these cases it would be counterproductive to train a new classifier. Instead, what we would like to do is determine where in the instance space the classifier errs, and fix only the classification in these so-called error regions with smaller models, called patches. In the following, we introduce the *Patching* framework, starting with a formal problem description.

### 4.3.1 Problem Description

We assume a general instance space  $D$  of instances  $\mathbf{x}$  with labels  $l(\mathbf{x})$ , and a black-box classifier  $M$ .  $M$  is immutable and inscrutable and is able to classify the examples of  $D$  well, i. e., with a high probability  $\Pr(M(\mathbf{x}) = l(\mathbf{x}))$ . We now receive new batches of examples  $D_i$ ,  $i > 0$ , for which  $M$  makes imperfect predictions, presumably because the labeling function  $l_i$  underlying  $D_i$  is slightly different from the function  $l$ , which  $M$  is approximating. Formally, the assumption is that for  $\mathbf{x} \in D_i$ ,

$$\Pr(M(\mathbf{x}) = l_i(\mathbf{x})) \leq \Pr(M(\mathbf{x}) = l(\mathbf{x})). \quad (4.1)$$

Our goal is to learn classifiers  $C_i$  that approximate  $l_i$  as close as possible.

### 4.3.2 The Patching Approach

A straight-forward way of addressing this problem is to directly train a classifier  $P_i = f(D_i)$  from the training set  $D_i$ . However, this approach

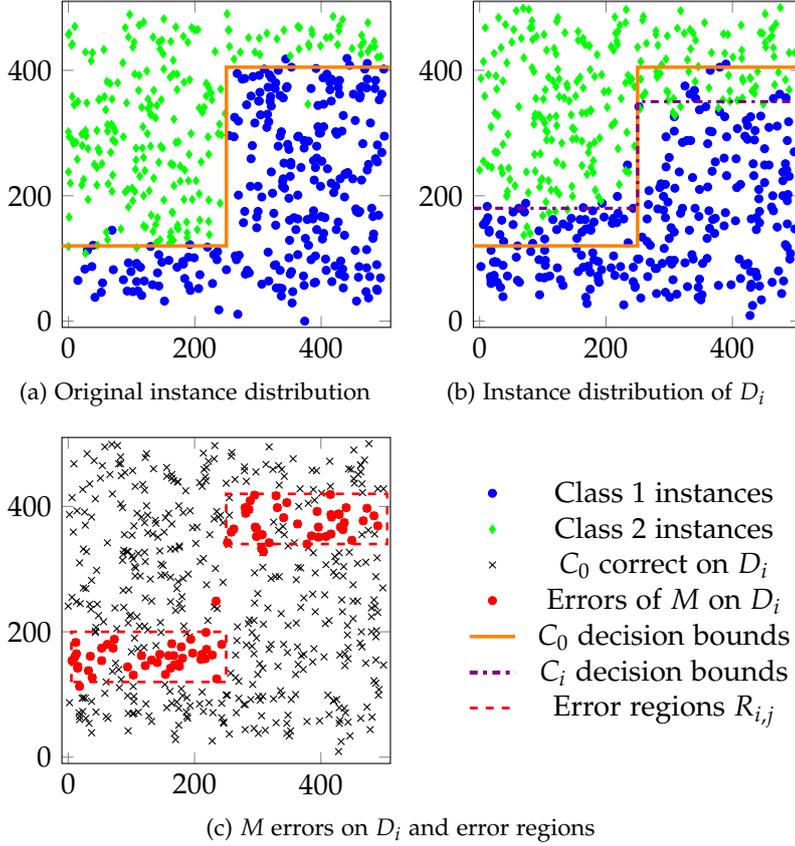


Figure 4.1: Learning error regions on a 2-dimensional dataset with a base classifier  $M$  and instances from a different distribution  $D_i$

is quite wasteful in terms of training examples because it requires to re-train a complete classifier from a sufficient number of samples. If  $|D_i|$  is rather small, we can expect that the classification performance of  $P_i$  is inferior to the performance of a suitable combination  $C_i$  of the classifiers  $M$  and  $P_i$ , because  $M$  is trained on a much larger set of instances. Formally, we expect that there is

$$\Pr(C_i(\mathbf{x}) = l_i(\mathbf{x})) \geq \Pr(P_i(\mathbf{x}) = l_i(\mathbf{x})), \mathbf{x} \in D_i. \quad (4.2)$$

For constructing such a classifier  $C_i$ , our approach aims at combining  $M$  and  $P_i$  in such a way, that the resulting ensemble improves the performance over  $M$  and  $P_i$  alone. This is achieved in two steps:

- (i) We train a classifier  $E_i$  that is able to identify one or more *error regions*  $R_{i,j}$ , in which  $M$  misclassifies data of  $D_i$  (illustrative example shown in Figure 4.1).
- (ii) We train a new classifier  $P_{i,j} = f(R_{i,j} | M)$ , a so-called *patch*, for each such region.

These two steps are further explained below. Optionally, the original prediction of  $M$  can be added as an additional attribute to both the error region and the patch learning steps.

At classification time for batch  $D_i$ , the patching classifier  $C_i$  first uses  $E_i$  to determine whether  $\mathbf{x}$  lies in one of the error regions  $R_{i,j}$ , and then uses the corresponding classifier  $P_{i,j}$  for classification. If  $\mathbf{x}$  does not lie in any error region (i. e., if  $E_i(\mathbf{x}) = 0$ ), the classifier uses  $M$  for classifying the example. More formally,

$$C_i(\mathbf{x}) = \begin{cases} P_{i,j}(\mathbf{x}) & \text{if } E_i(\mathbf{x}) = 1 \wedge \mathbf{x} \in R_{i,j}(\mathbf{x}), \\ M(\mathbf{x}) & \text{if } E_i(\mathbf{x}) = 0. \end{cases} \quad (4.3)$$

### 4.3.3 Error Region and Patch Learning

After receiving a new batch of examples  $D_i$  and the corresponding true labels, we train a classifier that learns in which part of the instance space the base classifier is likely to err. Therefore, we define a new training set consisting of all examples  $\mathbf{x} \in D_i$ , which are labeled with  $l_i(\mathbf{x})$  in  $D_i$ , and which are now re-labeled as

$$e_i(\mathbf{x}) = \mathbb{1}(M(\mathbf{x}) \neq l_i(\mathbf{x})). \quad (4.4)$$

In principle, any classifier  $E_i$  can be trained on this dataset to predict the errors of  $M$  on  $D_i$ . However, we assume a tree-based or rule-based classifier, which divides the instance space into smaller regions  $R_{i,j}$ .

Each *error region*  $R_{i,j}$  corresponds to a single rule (or the leaf of a decision tree), which predicts 1, i. e., predicts that all examples of  $D_i$  covered by this rule will be misclassified by  $M$ .

Rules that predict 0 are ignored, as they identify regions where  $M$  still operates well.

In order to learn *patches* for the error regions  $R_{i,j}$ , we train one new classifier  $P_{i,j}$  for each region using all training examples  $(\mathbf{x}, l_i(\mathbf{x})) \in D_i$  where  $\mathbf{x} \in R_{i,j}$ . This classifier now serves as the predominant classifier for the decision space region determined by  $R_{i,j}$ .

## 4.4 Related Work

The idea of patching is related to several well-known concepts in machine learning, especially concept drift and transfer learning. Unlike these works, we assume a fixed, immutable base classifier, and our goal is not to re-learn or modify this classifier, but to track and model changes in the data *relative* to it. In Section 3.5 we introduced related work in the domain of concept drift learning, which is the primary category of learning problem that we are trying to solve with our framework. Now we elaborate on related work with similar ideas.

The concept of *Patching* is quite similar to ensemble methods such as *Stacking* (Wolpert 1992; Ting and Witten 1999), where the idea is to collectively correct the predictions of multiple classifiers by training a meta classifier that combines their predictions.

Most related to our technique are *Arbitrating* (Ortega, Koppel, and Argamon 2001) and *Grading* (Seewald and Fürnkranz 2001), which both already feature the idea of training separate classifiers that indicate where the base classifiers in an ensemble err. However, these classifiers are then employed for filtering the predictions in an ensemble, whereas we train a separate classifier on the error regions.

In this respect, *Patching* is also related to *Boosting* (Freund and Schapire 1996) or *Additive Logistic Regression* (Jerome H. Friedman, Hastie, and Tibshirani 2000), but the setting differs in that there, multiple iterations are performed on the same data, whereas the goal here is the adaptation to new data.

The *Accuracy Weighted Ensemble* (AWE) by (Wang et al. 2003) constructs an ensemble of weighted classifiers based on their accuracy w. r. t. a time-evolving environment. It differs from *Patching*, such that the ensemble is constantly extended, whereas our ensemble of patches is re-constituted for every learning step.

The *Accuracy Updated Ensemble* (AUE) by (Brzezinski and Stefanowski 2011) extends this idea such that updates based on the current distribution are possible.

A similar ensemble method is *Dynamic Adaptation to Concept Changes* (DACC) by Jaber, Cornuéjols, and Tarroux (2013), which also relies on an ensemble of classifiers with a weighted vote. DACC, however, provides the feature that members from the lower performing half of the ensemble are removed by a random deletion strategy. The ensemble decision only takes the better half of members into account. This way, immature learners disappear over time and the ensemble is kept small.

Oza (2005) provides the online equivalents to the well-known Bagging (Breiman 1996a) and Boosting (Freund and Schapire 1996) algorithms, namely *OzaBoost* and *OzaBag*.

Finally, *Patching* can be compared to *Gradient Boosting* (Jerome H. Friedman 2002), where an ensemble of error-correcting models is established. Although similar, we do not learn an error correction in *Patching*, but instead divide this in two steps: learning the error regions and the patch for it.

## 4.5 Experimental Setup

In this section, we elaborate on the experimental setup. We conduct experiments in three scenarios: We use (i) *Patching* as intended, in a scenario with a given classifier that is patched based on the knowledge gained from new instances. Additionally, we apply it on (ii) *concept drift* and (iii) *transfer learning* tasks.

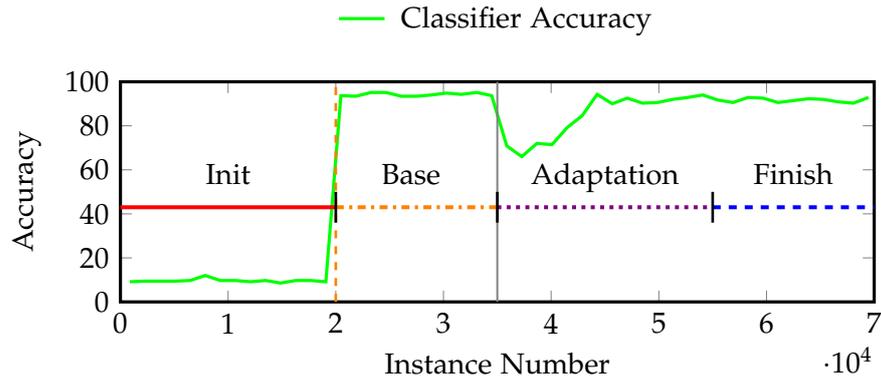


Figure 4.2: Phases during the course of the stream. CPs are shown as gray vertical lines (at point 3.5), the initialization phase ends at the dashed vertical line.

We use the *Massive Online Analysis* (MOA) framework<sup>2</sup> (Bifet, Holmes, et al. 2010) and the *Waikato Environment for Knowledge Analysis* (WEKA) toolkit<sup>3</sup> (Witten and Frank 2005) for machine learning, thereby simulating a real-world scenario where instances arrive one by one and labels are available in batches some time after. From a data stream, we preserve multiple batches of examples  $D_i$ , together with their respective labels  $l_i(\mathbf{x})$  for the learning steps. We allow the *Patching* algorithm to store the most recent  $n$  batches, all of which will be used to learn the error regions and respective patches. Different sizes of  $n$  yield different purposes: small  $n$  are required for quicker adaptation, but larger  $n$  usually result in a better accuracy in the long run. In our experiments we use a compromise value of  $n = 8$ , which yielded overall adequate results in preliminary runs.

We use a chunk/batch-based evaluation method to retrieve the performance of the classifier every  $m$  instances, for which we split each dataset into a certain amount of batches (Table 4.1). In this way, we can use the whole dataset for training and evaluation without having to rely on a separate hold-out set. The classifiers can use previously evaluated instances to incrementally expand their learning base.

A typical run of patching consists of four phases (illustrated in Fig. 4.2):

1. **Initialization (Init):** In the first phase, we create the base classifier. In the evaluation section we will not show this phase, because it is irrelevant to our findings. A real world application would start after this phase.
2. **Base:** In this phase, the underlying concept remains the same as in the *Init* phase. The patching algorithm will start to collect batches of instances, learn errors and update regularly. We can

<sup>2</sup> <http://moa.cms.waikato.ac.nz>

<sup>3</sup> <http://www.cs.waikato.ac.nz/ml/weka/>

get a close estimation of the performance before any concept drift.

3. **Adaptation:** This phase starts with the first CP, where one or multiple changes in the data occur. It ends when the performance has stabilized after the changes.
4. **Finish:** In the final phase, the concept remains stable. This phase is later used to calculate the final performance estimates the classifiers can achieve.

This setup represents a real-world usage scenario of classification on batches of instances. For the *transfer learning* experiments we aim at getting insight into how many instances are required to reach a certain performance level compared to the original data distribution. The *Patching* scenario skips the initialization altogether, and starts at the change point. In this scenario quick adaptation is the key performance indicator. Here, the benchmark algorithms we compare against start from scratch, whereas *Patching* has the benefit of a given classifier (constructed from the data in the Initialization phase).

Table 4.1: Summary of the datasets used in the experiments

Dataset	Drift Type	Init	Total	Chunks	CPs
<i>Patching Domain</i>					
$MNIST_{split}$	—	—	1000	100	—
$MNIST_{switch}$	—	—	1000	100	—
$MNIST_{appear}$	—	—	1000	100	—
$20NG_{R/T}$	—	—	1000	100	—
$20NG_{R/S}$	—	—	1000	100	—
<i>Transfer Domain</i>					
$MNIST_{flip}$	abrupt	20k	140k	200	1
$20NG_{R/T}$	abrupt	1600	4340	22	1
$20NG_{S/T}$	abrupt	1600	4324	22	1
$20NG_{R/S}$	abrupt	1600	4762	24	1
<i>Concept Drift Domain</i>					
$SEA_{lin}$	abrupt	100k	500k	200	2
$SEA_{multi}$	abrupt	100k	500k	200	2
$RotHyp$	gradual	100k	500k	100	—
$MNIST_{merge}$	abrupt	20k	70k	100	1
$MNIST_{appear}$	abrupt	20k	70k	100	1
$MNIST_{switch}$	abrupt	20k	70k	100	1

## 4.5.1 The Datasets

We evaluate our approach in a variety of different scenarios, encompassing (i) its intended use (*Patching* scenario) as well as (ii) *concept drift* and (iii) *transfer learning*. We relate to specific experiments that were conducted in (Dai et al. 2007; Gao et al. 2008; Pan and Yang 2010) for comparison. Those experiments are located in the domain of inductive transfer learning with source and target tasks being different, but related.

For the *Patching* scenario, we create experiments based on *MNIST*<sup>4</sup>, to which we introduce various changes. We also use the *20 Newsgroups*<sup>5</sup> dataset, which originates from the domain of *transfer learning* (Dai et al. 2007; Gao et al. 2008). In the *transfer learning* and *concept drift* experiments, we also rely on modified *MNIST* variants. Additionally, the *SEA concepts* (Street and Kim 2001) and a dataset created from a *rotating hyperplane* as described in (Hulten, Spencer, and Domingos 2001) are used for the *concept drift* domain.

We introduced the datasets in Section 3.7, and will now give a short summary of the changes we made in order to set them up as data streams with concept drift or for transfer learning. All datasets are explained below and described in Table 4.1 in detail. It shows a summary of all studied datasets, the type of drift that occurs (Webb et al. 2016), and the number of CPs. The *Init* value specifies the number of instances used for the *Initialization* phase, *Total* specifies the total number of instances, and *Chunks* is the number of batches the dataset is divided into.

### The MNIST Dataset

The first dataset is the *MNIST* dataset of handwritten digits. It consists of a total of 70,000 digits. We are randomizing the order, use it as a stream and introduce four different variants of change:

*MNIST<sub>merge</sub>*: The labels of the stream are changed such that classes 4, 5, 7 and 9 are labeled as 9 after the CP.

*MNIST<sub>appear</sub>*: The labels 3, 5, 7, 9 do not exist during the initialization, and appear at the CP.

*MNIST<sub>flip</sub>*: The pixel values are manipulated, so the written digits are flipped both horizontally and vertically.

*MNIST<sub>switch</sub>*: The classes of digits are switched such that 2 becomes 4, 3 becomes 5, and vice versa.

### 20 Newsgroups

The second dataset that we use in our evaluation is *20 Newsgroups*. This dataset consists of newsgroup entries and contains two hierarchical

<sup>4</sup> <http://yann.lecun.com/exdb/mnist/>

<sup>5</sup> <http://www.iesl.cs.umass.edu/data>

category attributes, one being the top-category and the other the subcategory. We derive a *transfer learning* task from this dataset, such that the goal is to classify the top-category. We vectorize the text part of the dataset and split it based on the subcategories. This means, that if  $A$  and  $B$  are top-categories, and each has two subcategories  $A_1, A_2$  and  $B_1, B_2$ , we split it such that  $A_1$  and  $B_1$  will be in the training set, whereas  $A_2$  and  $B_2$  are put in the test set. Therefore, the training and test sets are made up from different subcategories and will show a varying distribution.

$20NG_{R/S}$ : Top category changes from REC to SCI.

$20NG_{R/T}$ : Top category changes from REC to TALK

$20NG_{S/T}$ : Top category changes from SCI to TALK

### The SEA Dataset

The SEA concepts dataset is a dataset with abrupt concept drift (Street and Kim 2001) consisting of three attributes, where the attributes are used to generate the resulting binary label. All three attributes are real-valued and have random values between 0 and 10.

$SEA_{lin}$ : A linear relation between attributes and class label exists.

$SEA_{multi}$ : A multiplicative relation between attributes decides the class label.

A threshold value  $\theta$  will be changed during the course of the instance stream to introduce concept change. We generate a stream of 500,000 instances with 2 CPs per dataset. Furthermore, 10% class noise is added.

### Rotating Hyperplane

The last dataset we use is based on a rotating hyperplane in a  $d$ -dimensional space as proposed in (Hulten, Spencer, and Domingos 2001), with which a binary stream classification problem can be constructed. We generate a stream  $RotHyp$  of 500,000 instances with 10 numeric attributes and introduce a slow movement of the hyperplane. Thereby, we simulate a continuous gradual shift in the problem space.

## 4.5.2 The Patching Environment

*Patching* builds an ensemble of classifiers, consisting of three classification steps: (i) the base classifier  $M$ , (ii) the error region classifiers  $E_i$  and (iii) the patches  $P_{i,j}$ . Our implementation allows any WEKA classifier to be used for each of the steps. In our experiments, we primarily use random forests (Breiman 2001) with 100 random trees, mostly because this is a fast algorithm and gives good results without requiring extensive parameter optimization.

As a base classifier, we train a random forest on all instances from the *Init* phase.

In order to learn the error region classifiers, random forests only return binary information, which does not allow us to identify multiple error regions. For this reason, we use a version of the RIPPER rule learning algorithm (Cohen 1995), specifically modified to determine by which of its rules an instance has been classified, allowing us to obtain precise error regions by using each triggered rule as a region. For some problems, this works remarkably well, outperforming the binary error regions. However, in general scenarios the binary variant performs equally good, most likely a result of the random forests behavior.

The entire patching framework is implemented in MOA and publicly available on GitHub<sup>6</sup>.

### 4.5.3 Benchmark Algorithms

Due to the fact that our method uses an ensemble of classifiers for data that occurs in batches, we rely on the extensive survey by H. M. Gomes et al. (2017) to choose appropriate algorithms to compare against.

In order to ensure comparable results between the algorithms, we rely on random decision trees as the general underlying classifier. *Patching* can use any given classifier as base classifier, error region detector, and for the patches. Our goal of parameterizing the algorithms is that they have a maximum of 500 (random) trees in their ensemble to achieve comparable results. When we use RIPPER as error region detector, we cannot guarantee a maximum number of error regions and hence patches. Therefore we decided to use random forests (with 100 trees each) for error regions and the patch. We only construct one (complex) error region from the random forest. *Patching* can be configured to keep multiple batches in a FIFO queue. We keep the most recent 8 batches for the concept drift and transfer problems ( $Patching_{RF-RF-8}$ ), and all batches for the *Patching* scenario ( $Patching_{RF-RF-all}$ ), where the batch size is very small and all gathered information is crucial. The base classifier for *Patching* is trained as a *Random Forest* with 100 random trees.

We evaluate against algorithms with similar capabilities: From the related work in Section 4.4, the algorithms *AWE*, *AUE*, and *OzaBoost* are suitable and properly implemented in MOA, so they can be used for evaluation purposes.

For *AWE*, we set the ensemble size to 5 random forests, each consisting of 100 random trees. *AUE* is configured similarly, with a maximum of 500 random trees. *OzaBoost* is allowed to use 500 random *Hoeffding Trees* (Domingos and Hulten 2000), since it does not support generic random trees.

<sup>6</sup> <https://github.com/Shademan/Patching/>

Since there is—to our knowledge—no transfer learning algorithm that deals with the transfer learning problem as a streaming situation, we will also apply said algorithms for the transfer learning experiments. In our experiments, we choose the baseline performance to be a classifier that is trained on the data of the *Init* phase and has no capabilities to adapt during the course of the stream. It is also a *Random Forest* consisting of 100 random trees.

## 4.6 Results

In this section, we give an overview of our experimental results for the datasets described in the Section 4.5.1. We treat all problems as stream problems, which allows us to measure how quickly a classifier reacts to concept change, or how much additional information it needs to adapt to a transfer situation. Furthermore, we show some example graphs of the adaptation behavior for selected datasets.

### 4.6.1 Evaluation Measures

For the comparison of the algorithms we use the following metrics:

- **Final Accuracy (F.Acc.):** Classification accuracy, measured in the second half of the *Finish* phase.
- **Recovery Speed (R.SpD):** Number of instances that a classifier requires during the *Adaptation* phase to achieve 95% of its final accuracy.
- **Adaptation Rank (Ad.Rk):** Average Rank of the classifier during the *Adaptation* phase.
- **Final Rank (F.Rk):** Average Rank of the classifier during the *Finish* phase.

Final accuracy should tell the overall capabilities of the adaptation algorithm, without the effects of the concept drift interfering. Hence, it is measured in the *Finish* phase, when the concept has remained stable for a while. The recovery speed is a measure for the speed of the adaptation. It is measured relative to the final accuracy, therefore an algorithm can have a high recovery speed, but still display a low overall performance. The measures must always be viewed in conjunction.

In order to directly compare the algorithms, we calculated final rank and adaptation rank. The ranks give a more objective view on the performance of an algorithm during a certain phase of the stream.

We ran each algorithm 10 times on every dataset with different random seeds and averaged over the results for the 10 runs. The standard deviation of accuracy in those 10 runs was smaller than 2%.

## 4.6.2 Patching Scenario

In Table 4.2 we show the results of the datasets which put *Patching* to its intended use: Leveraging a given model and adapt to changes relative to it.

As we can see, *Patching* excels in almost all scenarios w.r.t. the adaptation rank (Ad.Rk) and the final rank (F.Rk). The recovery speed (R.Spd) values may be misleading, since the final accuracy of *Patching* is higher for most datasets. The example results in Figure 4.3 demonstrate the behavior over the course of the instances. *Patching* shows both a better start and quicker adaptation, since it can leverage the given classifier. Although in some settings, it does not improve significantly from it. Overall, *Patching* achieves the highest rank for adaptation and final accuracy in five of six datasets.

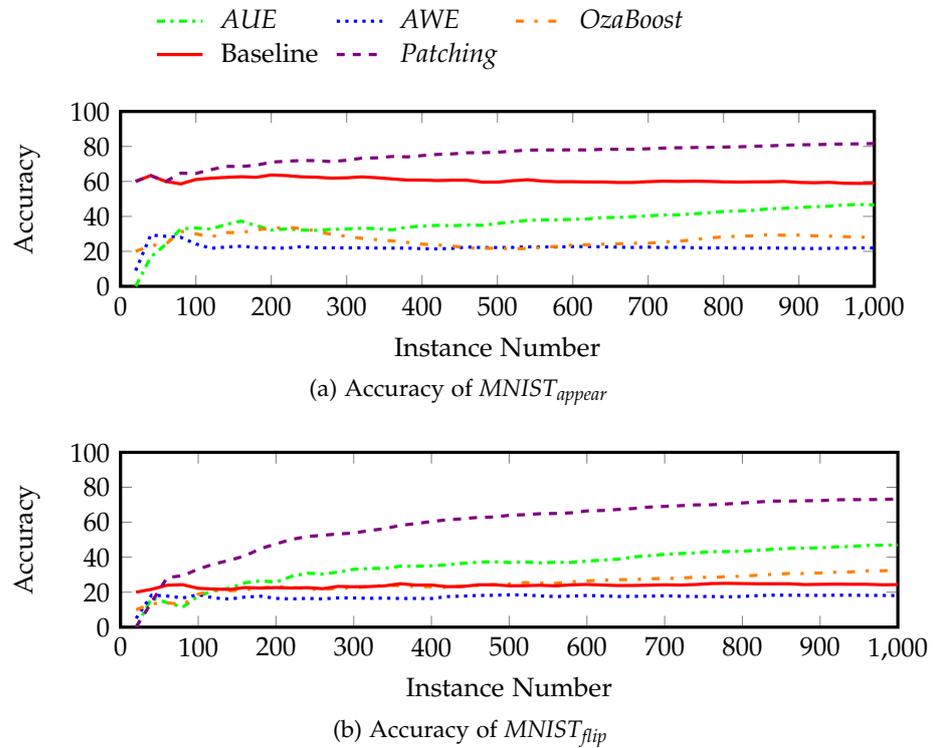


Figure 4.3: Results of stream classification accuracy progression for problems that resemble the intended use of *Patching*.

Table 4.2: Experiment results for the *Patching* scenario

Classifier	F.Acc	R.Spд	Ad.Rk	F.Rk
<i>MNIST<sub>appear</sub></i>				
Baseline	59.41%	—	1.96	2.00
<i>AUE</i>	43.83%	>20k	3.24	3.00
<i>AWE</i>	25.56%	20	4.60	5.00
<i>OzaBoost</i>	37.18%	360	4.16	4.00
<i>Patching<sub>RF-RF-all</sub></i>	<b>81.85%</b>	220	<b>1.04</b>	<b>1.00</b>
<i>MNIST<sub>merge</sub></i>				
Baseline	70.71%	—	1.96	2.00
<i>AUE</i>	48.12%	100	4.24	3.00
<i>AWE</i>	42.51%	20	3.48	5.00
<i>OzaBoost</i>	44.37%	80	4.28	4.00
<i>Patching<sub>RF-RF-all</sub></i>	<b>86.66%</b>	20	<b>1.04</b>	<b>1.00</b>
<i>MNIST<sub>flip</sub></i>				
Baseline	24.55%	—	3.48	4.00
<i>AUE</i>	45.03%	>20k	2.68	2.00
<i>AWE</i>	22.28%	20	4.68	5.00
<i>OzaBoost</i>	28.65%	140	3.00	3.00
<i>Patching<sub>RF-RF-all</sub></i>	<b>74.19%</b>	>20k	<b>1.16</b>	<b>1.00</b>
<i>MNIST<sub>switch</sub></i>				
Baseline	59.73%	—	1.84	2.00
<i>AUE</i>	43.68%	>20k	3.20	3.00
<i>AWE</i>	25.35%	20	4.64	5.00
<i>OzaBoost</i>	37.17%	360	4.16	4.00
<i>Patching<sub>RF-RF-all</sub></i>	<b>83.35%</b>	320	<b>1.16</b>	<b>1.00</b>
<i>20NG<sub>R/S</sub></i>				
Baseline	67.18%	—	2.08	2.00
<i>AUE</i>	61.26%	260	3.88	4.00
<i>AWE</i>	56.04%	0	4.48	5.00
<i>OzaBoost</i>	65.08%	440	3.48	3.00
<i>Patching<sub>RF-RF-all</sub></i>	<b>73.09%</b>	0	<b>1.08</b>	<b>1.00</b>
<i>20NG<sub>R/T</sub></i>				
Baseline	63.65%	—	4.64	5.00
<i>AUE</i>	68.75%	0	3.08	3.45
<i>AWE</i>	68.56%	0	3.28	3.55
<i>OzaBoost</i>	<b>82.01%</b>	160	<b>1.52</b>	<b>1.00</b>
<i>Patching<sub>RF-RF-all</sub></i>	77.83%	240	2.48	2.00

### 4.6.3 Transfer Learning Datasets

The results for transfer learning are shown in Table 4.3. Each of the applied algorithms shows strengths and weaknesses regarding both adaptation speed and final performance.

*Patching* performs well overall, but not significantly better (cf. Figure 4.4), the only exception being  $20NG_{S/T}$ . In the cases of  $MNIST_{flip}$  and  $20NG_{R/S}$ , the base classifier gives no real advantage.

For  $MNIST_{flip}$ , this can be explained. Flipping the numbers yields new pixel patterns, that are substantially different to the original numbers. The numbers ‘1’ and ‘8’ still look similar to before, so none of the methods should have a big disadvantage with them. However, numbers ‘6’ and ‘9’ look almost the same when flipped on both axes. Hence, what the base classifier may classify as a ‘6’ is now a ‘9’. This is a pattern that the patch classifier can pick up very fast, which gives it the adaptation advantage in this case.

Table 4.3: Experiment results in the *transfer* domain

Classifier	F.Acc	R.Spd	Ad.Rk	FRk
<i>MNIST<sub>flip</sub></i>				
Baseline	23.79%	—	4.60	5.00
<i>AUE</i>	<b>93.62%</b>	3500	3.00	<b>1.19</b>
<i>AWE</i>	91.24%	1400	<b>1.60</b>	2.29
<i>OzaBoost</i>	89.28%	5600	3.53	4.00
<i>Patching<sub>RF-RF-8</sub></i>	91.39%	2100	2.27	2.52
$20NG_{R/S}$				
Baseline	64.36%	—	3.33	5.00
<i>AUE</i>	71.70%	360	3.78	3.10
<i>AWE</i>	<b>91.79%</b>	240	<b>1.37</b>	<b>1.00</b>
<i>OzaBoost</i>	67.73%	300	4.44	3.90
<i>Patching<sub>RF-RF-8</sub></i>	85.03%	720	2.07	2.00
$20NG_{S/T}$				
Baseline	48.75%	—	4.52	5.00
<i>AUE</i>	79.22%	300	2.38	3.50
<i>AWE</i>	78.81%	180	2.62	3.50
<i>OzaBoost</i>	86.78%	540	<b>2.19</b>	2.00
<i>Patching<sub>RF-RF-8</sub></i>	<b>91.07%</b>	780	3.29	<b>1.00</b>

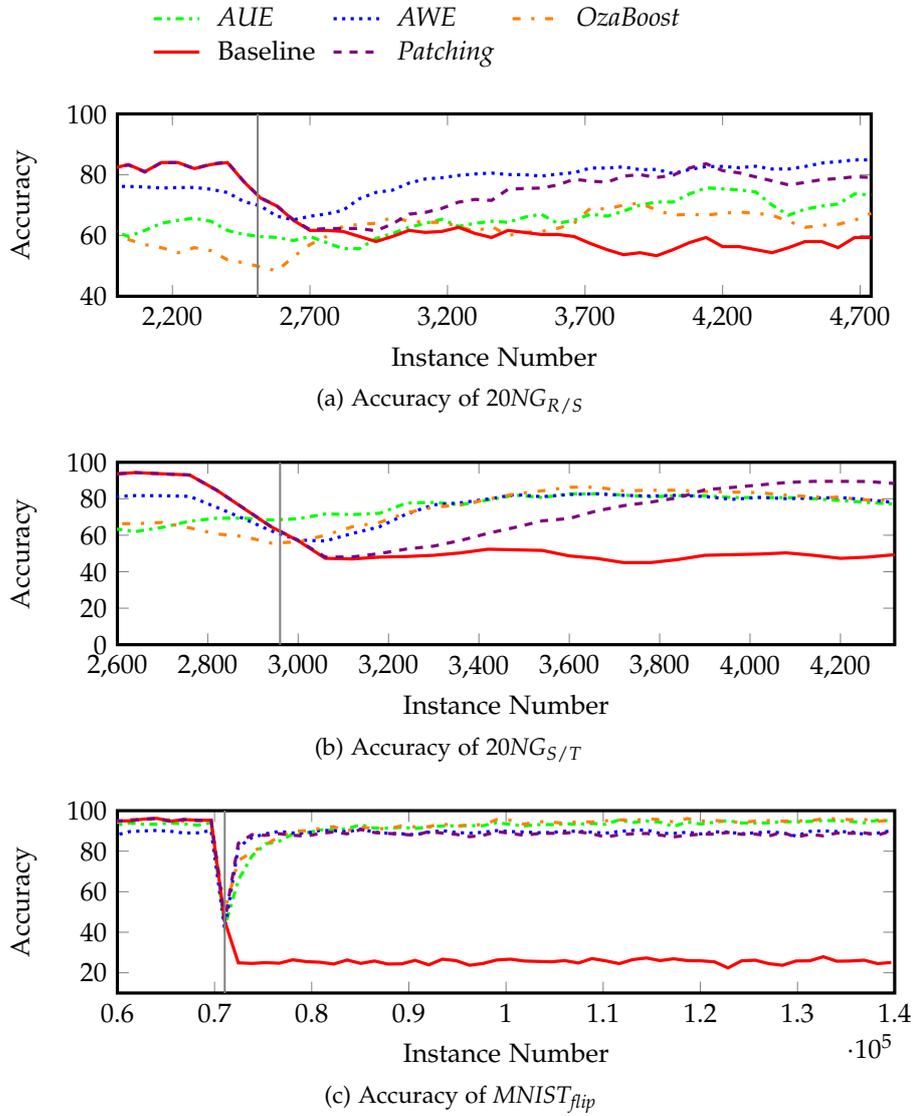


Figure 4.4: Results of stream classification accuracy progression for *transfer* problems

#### 4.6.4 Concept Drift Datasets

Table 4.4 shows the results for all concept drift-related datasets. As we can see, *Patching* performs well in adaptation rank where it achieves the highest rank 4 out of 6 times. In final rank, however, *Patching*, *OzaBoost* and *AUE* perform very similar with no significant differences. It has to be mentioned that the other ensemble methods are continuously improving algorithms and might start to outperform *Patching* at some point, given more instances (Figure 4.5).

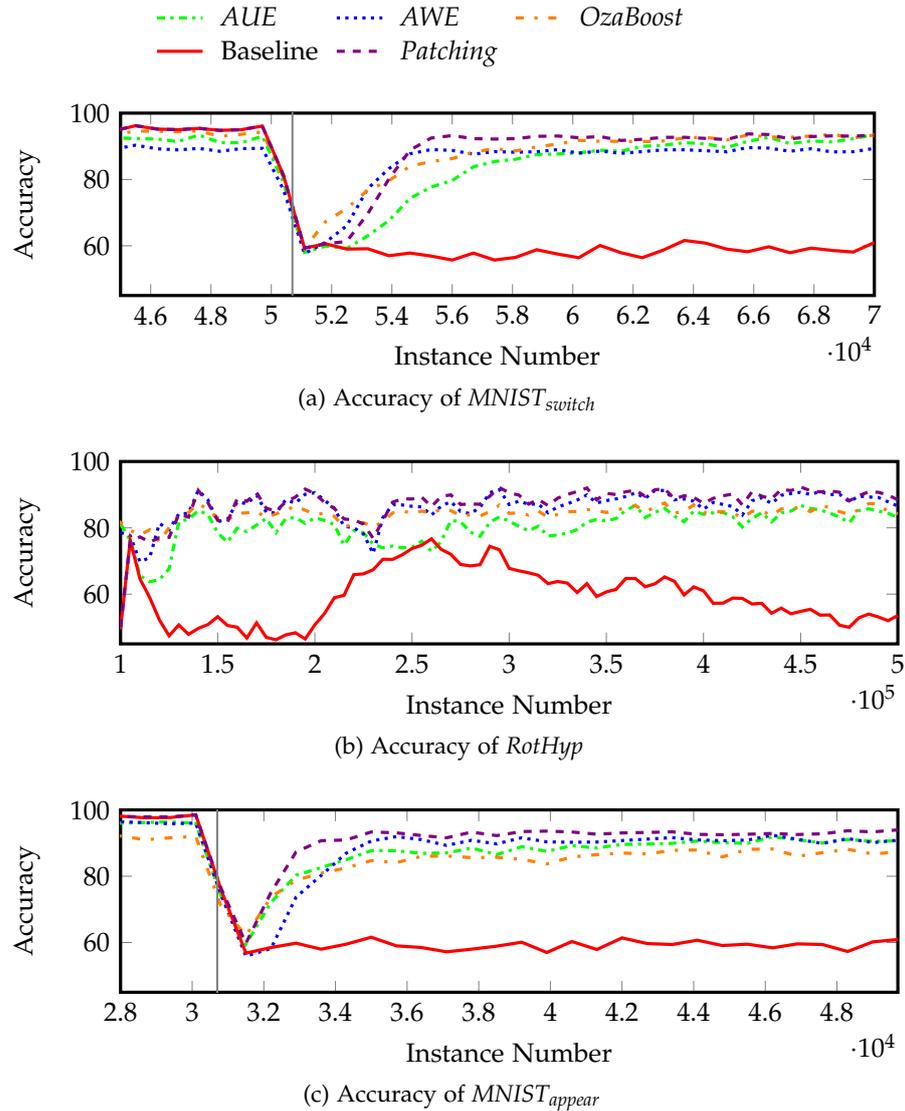


Figure 4.5: Results of stream classification accuracy progression for *concept drift* problems

Table 4.4: Experiment results in the *concept drift* domain

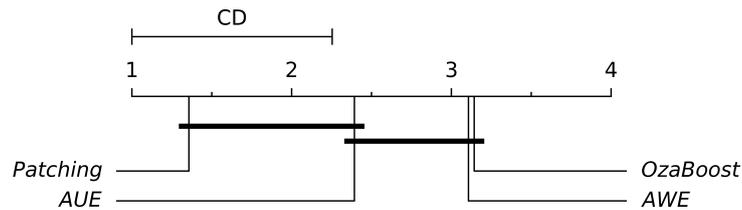
Classifier	F.Acc	R.Spd	Ad.Rk	F.Rk
<b><i>MNIST<sub>appear</sub></i></b>				
Baseline	59.23%	—	4.47	5.00
<i>AUE</i>	90.87%	2100	3.00	2.67
<i>AWE</i>	90.94%	2800	2.80	2.33
<i>OzaBoost</i>	87.23 %	1400	3.67	4.00
<i>Patching<sub>RF-RF-8</sub></i>	<b>93.21%</b>	1400	<b>1.07</b>	<b>1.00</b>
<b><i>MNIST<sub>switch</sub></i></b>				
Baseline	59.54%	—	3.89	5.00
<i>AUE</i>	91.42%	4200	3.44	2.47
<i>AWE</i>	91.23%	2100	2.33	2.53
<i>OzaBoost</i>	88.46%	4200	3.33	4.00
<i>Patching<sub>RF-RF-8</sub></i>	<b>93.77%</b>	3500	<b>2.00</b>	<b>1.00</b>
<b><i>RotHyp</i></b>				
Baseline	53.02%	—	4.90	5.00
<i>AUE</i>	84.78%	10000	3.90	4.00
<i>AWE</i>	89.96%	5000	2.24	1.64
<i>OzaBoost</i>	88.11%	0	<b>1.62</b>	3.00
<i>Patching<sub>RF-RF-8</sub></i>	<b>89.98%</b>	15000	2.33	<b>1.36</b>
<b><i>SEA<sub>multi</sub></i></b>				
Baseline	85.59%	—	4.84	5.00
<i>AUE</i>	<b>99.38%</b>	0	2.35	<b>1.00</b>
<i>AWE</i>	98.20%	0	2.32	3.14
<i>OzaBoost</i>	97.71%	0	3.94	3.81
<i>Patching<sub>RF-RF-8</sub></i>	98.92%	0	<b>1.55</b>	2.05
<b><i>SEA<sub>lin</sub></i></b>				
Baseline	67.35%	—	4.90	5.00
<i>AUE</i>	<b>99.81%</b>	5000	2.20	1.62
<i>AWE</i>	99.54%	0	2.56	2.81
<i>OzaBoost</i>	95.20%	0	3.83	4.00
<i>Patching<sub>RF-RF-8</sub></i>	99.78%	5000	<b>1.51</b>	<b>1.57</b>

### 4.6.5 Significance Testing

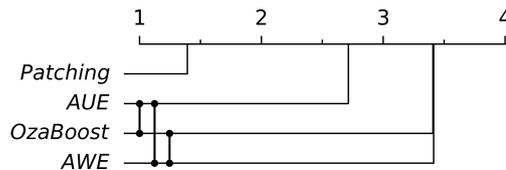
In order to show that *Patching* has significant advantages, we applied the nonparametrical Friedman test (M. Friedman 1937) to our average ranks of both adaptation and final accuracy, followed by the Nemenyi post-hoc test (Nemenyi 1963) as shown in (Demšar 2006) to assure pairwise significance and calculate critical distances. For this calculation we removed the Baseline from the ranking, since it distorts the Friedman test on the null-hypothesis that all algorithms perform equally well. The significance level we chose is 0.95.

**Friedman-Nemenyi:** The Figures 4.6a and 4.7a display the average rank of the algorithms. Connections via horizontal bars indicate, that two algorithms are in the same significance group. If the distance on the average rank exceeds the critical distance (CD=1.22), this indicates a rejection of the null-hypothesis.

**Friedman-Wilcoxon:** The Figures 4.6b and 4.7b display the average rank of the algorithms. Algorithms are compared pairwise, and are connected via vertical lines if the null hypothesis is *not* rejected.



(a) Friedman test with Nemenyi post-hoc test (Critical Distance = 1.22)



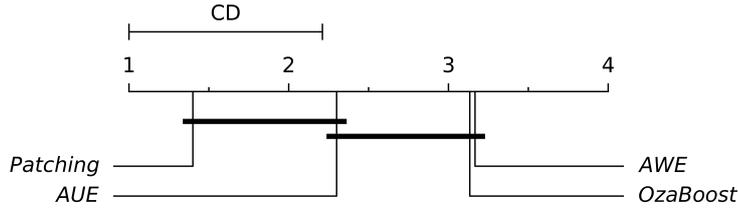
(b) Friedman test with Wilcoxon pairwise comparison

Figure 4.6: Statistical comparison of the final ranks.

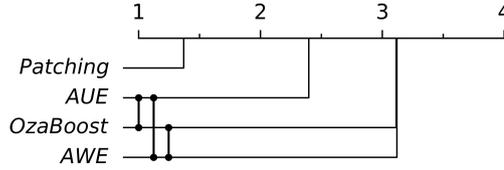
Regarding the final ranks (Figure. 4.6a), *Patching* significantly outperforms *AWE* (distance 1.77) and *OzaBoost* (distance 1.73) and is in the same group as *AUE* (distance 0.9) when compared via the critical distance.

In Figure 4.6b we show the pairwise comparisons. In the pairwise comparison, *Patching* has a significant advantage.

For the adaptation rank (Figure 4.7) the distances are similar, which means that *Patching* performs significantly better compared to *AWE*



(a) Friedman test with Nemenyi post-hoc test (Critical Distance = 1.22)



(b) Friedman test with Wilcoxon pairwise comparison

Figure 4.7: Statistical comparison of the adaptation ranks.

and *OzaBoost*, while *AUE* is just within the range of non-significance. Again, in the pairwise comparison, a significant advantage is indicated.

The statistical evaluation was performed on all ranked results. When we remove the results of the *Patching*-scenario from the calculation, there is no significant advantage anymore, which was to be expected. Our approach relies heavily on the previous knowledge, where it can get the cold-start advantage in the *Adaptation* phase.

## 4.7 Discussion of the Approach

In terms of drift detection, *Patching* employs an implicit mechanism. By learning the error regions, it detects if the base classifiers exhibits erroneous behavior. Through the learning algorithm, outliers or insignificant concept drift will not be recognized as error region. This solves the problem of threshold-setting in active drift-detection methods, and increases practical applicability of the approach. However, it also requires the errors to be of such form, that the error region classifier can properly detect them. This might not be the case for all scenarios, e. g., when a subtle concept drift affects the complete feature space, and only few, widely spread examples are observed. In this case, *Patching* might discard the few instances as local outliers at first, until more examples have been collected. A disadvantage of implicit drift detection on batches of data is the fact, that it is not precisely discovered in which part of the batch the drift occurred. This can be a problem, when the batch size is large, since multiple consecutive concepts can occur in one batch. In order to mitigate this issue, large batches should be artificially broken down into smaller batches.

In general, it may happen that the complete instance space is erroneous. This is not ideal for our approach, since one of the main

prerequisites is the assumption, that the base classifier still works well in some or most parts. In this case, *Patching* will still be able to handle the scenario, but all the predictions will be diverted to the patches.

## 4.8 Conclusion

In this chapter, we have introduced *Patching*, i. e., the idea of learning local corrections to existing classifiers, thereby eliminating the need for re-learning an existing classifier. We have shown experimentally that classifier patching works well for its intended use: scenarios where the classifier can leverage a pre-existing model for new situations, where (partial) adaptation is required and regions of erroneous instances can be determined and patched.

Because of its design, it adapts faster in these scenarios than the benchmark algorithms. *Patching* can also be applied to scenarios in *transfer learning* or situations of *concept drift*, where it manages to rival state-of-the-art competitors in either adaptation speed or accuracy in many scenarios. However, for massive online analysis, where classification is learned from zero, we recommend the state-of-the-art ensemble algorithms that are mentioned in this chapter for their focus on computationally efficient behavior.

In Chapter 5 we extend our approach specifically towards the use with neural networks. We also mitigate the issues caused by the constant re-learning of the error regions in favor of updateable error estimators, and enhance the performance by introducing completely updateable ensemble members in Chapter 6.

# Neural Network Patching

---

In this chapter we present neural network patching, an approach for adapting neural network models to non-stationary environments<sup>1</sup>. It is the enhanced approach of applying the *Patching* idea to neural networks. Instead of creating or updating a network to accommodate concept drift, neural network patching leverages the inner layers of a previously trained network to learn a patch that enhances the classification or adapts it towards concept drift.

Neural network patching is based on the idea that the original network can still classify a majority of instances well, and that the latent feature representations encoded in the network aid the classifier to cope with unseen or changed inputs. It learns (i) a predictor that estimates whether the original network will misclassify an instance, and (ii) a patching network that fixes the misclassification.

We evaluated this technique on several datasets, comparing it to similar methods. Our finding is that neural network patching is adapting quickly to concept changes, while also maintaining long-term learning capabilities similar to more complex methods that update the entire network.

This chapter is structured as follows. Section 5.1 motivates the necessity for neural network patching. In Section 5.2 we discuss related work from the domain of adaptive learning. In Section 5.3 we elaborate on adaptive learning for neural networks in general, and present the preliminary experimental setup in Section 5.4. We explain *NN-Patching* in Section 5.5, present the final experiments in Section 5.6, and give experimental results in Section 5.7. We conclude our findings in Section 5.8.

## 5.1 Introduction

Nowadays, DNNs comprise the state-of-the-art in many domains such as image classification and segmentation, text translation, time-series prediction, and many more (LeCun, Bengio, and G. Hinton 2015; Jürgen Schmidhuber 2015).

The main advantage of deep networks is their layered architecture, which is easier to train compared to networks with a single hidden layer, given enough training data is present, although the single layer network can provide the mathematically identical function (Ba and Caruana 2014). The possibility of training bigger and deeper networks,

---

<sup>1</sup> This chapter is based on Kauschke, Lehmann, and Fürnkranz (2019), where this work was published at the IJCNN conference in 2019.

which has emerged from recent technical advances, has enabled neural networks to deal with increasingly complex problems. In a trained network, each of the layers represents a different stage of abstraction from the input data, similar to how we believe the human brain processes information. Advanced techniques such as convolutional layers or long-short-term-memory units provide functionality that is beneficial to certain problems, for example when dealing with image data or sequential prediction tasks. A typical network for image classification consists of multiple layers of convolutional units (He et al. 2016), each representing feature detectors with different grades of abstraction. Early layers detect simple structures such as edges or corners. Later layers comprise more complex features related to the given task, for example eyes or ears, if the goal is to recognize faces.

Due to the vast amounts of data available today, building highly capable DNNs for certain tasks has become feasible, but it is still a very resource intensive process. Researchers are spending increasing amounts of time developing network architectures for certain problems. Together with the actual training time, a complex DNN can become expensive. Handling concept drift is even more complex with DNNs, caused by the complex adaptation required to update the network for the changed task. However, it is an important task in order to avoid perfectly capable systems from degrading or even becoming unusable over time.

In this chapter, we provide a solution to this problem. We recognize the fact, that building a well-working neural network for a certain task can be cumbersome and require many iterations w. r. t. the choice of architecture and the hyper-parameters. Once such a network is established and properly trained, a prolonged use of it is usually appreciated. However, it is not guaranteed that the underlying problem domain remains stationary, and it is desirable that the network can adapt to such changes.

To mitigate the problem, we build upon our idea of *Patching* from Chapter 4. We try to recognize regions where the model errs, and learn local models—so-called *patches*—that repair the original model in these erroneous parts.

For dealing with neural networks in particular, we present *neural network patching (NN-Patching)*, a variant of *Patching* that is specifically tailored to neural network classifiers.

It allows existing neural networks to be adapted to new scenarios by adding a network layer on top of the existing network. This layer leverages the resulting neuron activations of inner layers of the network to enhance its capabilities. Furthermore, the patching network is only activated, when the underlying base prediction is likely to be erroneous.

## 5.2 Related Work

Our proposed method deals with concept drift or other types of change (such as in transfer learning) via an adaptation mechanism, which is why research in the general areas of concept drift or transfer learning and adaptive neural networks is generally relevant to our approach. In this section we introduce related work in these fields. For other related work, please refer to Section 3.5.

### 5.2.1 Adaptive Learning with Neural Networks

Learning in an incremental, adaptive way with neural networks is difficult because of how neural networks are trained. As described by French (1999), connectionist networks tend to forget previously learned knowledge when learning new patterns. This is called *catastrophic forgetting*, and is a manifestation of the so-called *stability-plasticity-dilemma* (Carpenter and Grossberg 1987). In human brains, forgetting is a natural process that happens gradually. However, in ANNs trained by backpropagation, learning new information requires a reconfiguration of the weights in the network. This can lead to forgetting concepts that have been learned before. While this may be a wanted behavior, it can also lead to forgetting concepts that are still relevant. Although researchers have provided solutions to the problem via specialized network architectures (Kirkpatrick et al. 2017), the problem is not generally solved.

As a consequence of this, as of today, neural networks do not play a prominent role in the domain of adaptive learning under concept drift. Most state-of-the-art techniques are based on classical machine learning methods such as decision trees, Bayesian learners or ensembles thereof. One of the few efforts to employ neural networks in said domain is *Learn<sup>++</sup>* by Polikar (Polikar et al. 2001) and its variations. *Learn<sup>++</sup>* is an incremental learning algorithm, that trains one neural network for each new batch of data, and combines their decisions via weighted voting.

### 5.2.2 Transfer Learning with Neural Networks

In transfer learning, the primary goal consists of leveraging knowledge from a known source task to solve a task from a (similar) target domain (Torrey and Shavlik 2009; Pan and Yang 2010). Transfer learning is highly related to the field of domain adaptation (Ben-David et al. 2010).

In transfer scenarios, e. g., (Long et al. 2015; Ganin and Lempitsky 2015; Oquab et al. 2014; Gong, Grauman, and Sha 2013), the adaptation is usually achieved by mitigating the shift in data distributions via beneficial representations or kernel transformations. In the supervised scenarios we are interested in, the existing knowledge is combined

with additional information from the target task—or even a third, related task—to facilitate the transfer (Hoffman et al. 2014; Saenko et al. 2010; Bengio 2012; Geng et al. 2016). The usual approach with deep CNNs aims at the re-use of latent representations that are built while learning the network (Tzeng et al. 2014). Yosinski et al. (2014) elaborate on how deep CNNs learn layers of representations which tend to develop from being more general to being specific w. r. t. the given task.

Our work is related to both concept drift and transfer learning in that we want to enable an existing neural network classifier to adapt for concept drift on a stream of data, as well as to a completely new scenario as in transfer learning. Ideally we can achieve this with as few labeled examples as possible. In the following sections we will describe the core idea of our method and how it can be used to achieve said goals.

## 5.3 Deep Neural Network Adaptation

Since neural networks are usually trained by backpropagation, adapting a neural network towards a changed scenario can be achieved via training on the latest examples, hence refining the weights in the network towards the current concept. However, this may lead to *catastrophic forgetting* (French 1999) and—depending on the size of the networks—may be costly. To mitigate this issue, a common approach is to train only part of the network and not adapt the more general layers (Yosinski et al. 2014), but only the specific layers relevant to the target function. For example, Cireşan, Meier, and Jürgen Schmidhuber (2012) leverage this behavior to achieve transfer to problems with higher complexity than the original problem the network was intended for.

We observe three principal properties of neural networks w. r. t. adaptive learning: (i) neural networks are useful towards adaptation tasks, caused by their hierarchical structure, (ii) neural networks can be trained such that they adapt to changed environments via new examples, and (iii) this adaptation may lead to *catastrophic forgetting*. In our proposed method, we want to leverage the advantages of (i) and (ii), but avoid the disadvantages of (iii). In the following we explain the patching procedure for neural networks.

### 5.3.1 Neural Network Patching

The goal of *NN-Patching* is to apply the idea of *Patching* on the special case of neural network classifiers. More specifically, we want to advance it, so that it can leverage the inner layers of a deep neural network. In these layers, abstract representations and intermediate features are stored that may be crucial to the classification. We also want to improve the training process such that it is iterative and con-

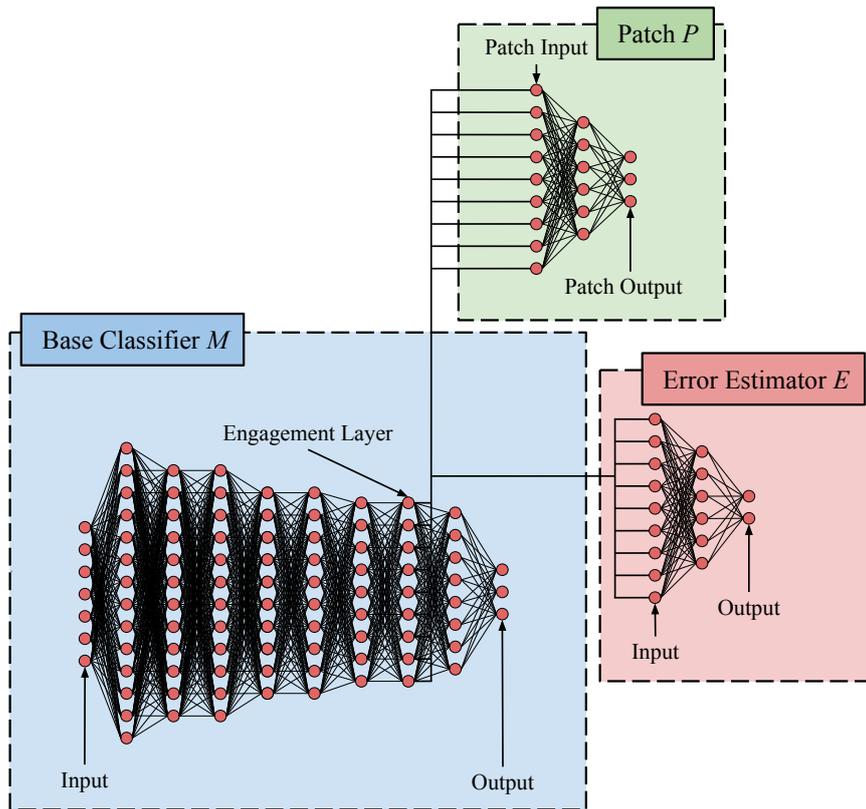


Figure 5.1: Patching of feed-forward networks – blue: the original network, green: the patch, red: the error region network

tinuous instead of disruptive. In the following, we describe how these advances are achieved.

We tailor the *Patching*-procedure to the specific case of neural network classifiers. The idea is depicted in Figure 5.1. *NN-Patching* therefore consists of three steps:

1. **Learn a classifier  $E$  that determines where  $M$  errs.** In this step, when receiving a new batch of labeled data, the data is used to learn a classifier that estimates where  $M$  will misclassify instances.
2. **Learn a patch network  $P$ .** The patch network engages to one inner layer of  $M$  and takes the activations of that layer as its own input (Fig. 5.1).
3. **Divert classification from  $M$  to  $P$ , if  $E$  is confident.** When an instance is to be classified, the error detector  $E$  is executed. If the result is positive, classification is diverted to  $P$ , otherwise to  $M$ .

In contrast to the original procedure, neural networks enable us to iteratively update both  $E$  and  $P$  over time. We will hence not create separate versions for each new batch, but rely on the existing one and update it via backpropagation with the instances from the latest batch.

To learn the patch network, we must engage in one of the inner layers of  $M$ . The selection of this layer is non-trivial. This is also true

for the selection of the patch and error estimator architecture. On the one hand, the patch needs to be sufficiently large to be able to learn the required function, but on the other hand it needs to be as small as possible, so it can be trained with fewer examples. The bigger the size of the patching network, the more weights have to be trained, which then requires more training instances. In an ideal world, the patch architecture would be fit perfectly to the problem at hand. Ideas such as the biologically inspired process of neurogenesis (Aimone et al. 2014) have been applied to ANNs (Draeos et al. 2017) to achieve this behavior. It allows growing a network iteratively for it to be sufficiently complex to solve the given task. Unfortunately, these endeavours are not yet mature enough to be universally applicable.

Another challenge in training the patch network is the selection of training instances. In *Patching*, the patch model  $P$  is always trained on the instances in the error-region alone. This is based on the assumption, that the error-estimator  $E$  can find distinctive error regions.

We conducted an extensive empirical analysis on patch architecture and engagement points in (Kauschke and Lehmann 2018), in order to establish heuristics on ideal engagement layers and patch architectures. Also, we investigate different ways of estimating the error and evaluate the performance of the resulting patches. A summary and discussion of the key findings is given in Section 5.5.

## 5.4 Preliminary Experimental Setup

In this section, we will elaborate on the datasets we used in the preliminary experiments as well as the final experiment setup itself. Our datasets are derived from well known datasets (cf. Section 3.7) and are engineered to give a stream of instances, where each stream contains one or multiple drifts of the underlying concept. We evaluate these streams as sequence of instances, where the true labels are retrieved in regular intervals. These are so-called batches of instances. On the end of each batch we retrospectively evaluate the performance of the classifier, and make adaptations for the next batch.

### 5.4.1 Evaluation Measures

For the comparison of the algorithms we use the same metrics as in the previous chapter (Section 4.6.1): Final Accuracy (F.Acc.), Recovery Speed (R.Spd), Adaptation Rank (Ad.Rk), and Final Rank (F.Rk).

These metrics allow us to judge the compared algorithms regarding their overall capabilities and adaptation speed. The rankings are used for statistical analysis.

Table 5.1: Summary of the datasets used in the experiments

Dataset	Init Size	CPs	Total Size	Chunks
<i>MNIST Dataset</i>				
$MNIST_{flip}$	40k	#70k	140k	100
$MNIST_{rotate}$	20k	#35k	70k	100
$MNIST_{appear}$	15k	#20.4k	50.4k	100
$MNIST_{remap}$	20k	#35.7k	70k	100
$MNIST_{transfer}$	20k	#35.7k	70k	100
<i>NIST Dataset</i>				
$NIST_{flip}$	30k	#40k	100k	100
$NIST_{rotate}$	30k	#40k	100k	100
$NIST_{appear}$	20k	#28.6k	88.6k	100
$NIST_{remap}$	20k	#28k	55.8k	100
$NIST_{transfer}$	20k	#30k	80k	100

## 5.4.2 Evaluation Datasets

We evaluate our findings on 10 scenarios which are based on two datasets. Each scenario represents a different type of concept drift with varying severity up until a complete transfer of knowledge to an unknown problem. The scenarios are summarized in Table 5.1.

The MNIST Dataset.

As in Section 4.5.1, we are using the *MNIST* dataset of handwritten digits. It contains the pixel data of 70,000 digits (28x28 pixel resolution), which we treat as a stream of data and introduce changes to. We created the following drift scenarios:

- $MNIST_{flip}$ : The second half of the dataset consists of vertically and horizontally flipped digits.
- $MNIST_{appear}$ : The digits change during the stream, such that classes 5–9 do not exist in the beginning, but only start to appear at the CP (in addition to 0–4).
- $MNIST_{remap}$ : In the first half, only the digits 0–4 exist. The input images of 0–4 are then replaced by the images of 5–9 for the second half (labels remain 0–4). This scenario only consists of 5 classes.
- $MNIST_{transfer}$ : The first half of the stream only consists of digits 0–4, while the second half only consists of the before unseen digits 5–9.

## The NIST Dataset.

The second dataset is the *NIST*<sup>2</sup> dataset of handprinted forms and characters. It contains 810,000 digits and characters, to which we apply similar transformations as to the *MNIST* data. Contrary to *MNIST*, *NIST* items are not pre-aligned, and the image size is 128x128 pixels. We use all digits 0-9 and upper-case characters A-Z for a total of 36 classes as data stream and draw a random sample for each scenario.

- *NIST<sub>flip</sub>*: The second half of the dataset consists of vertically and horizontally flipped images.
- *NIST<sub>rotate</sub>*: The images in the dataset start to rotate randomly at instance #40k with increasing rotation up to 180 degrees for the last 10k instances.
- *NIST<sub>appear</sub>*: The distribution of the images changes during the stream, so that instances of classes 0-9 do not exist in the beginning, but only start to appear at the CP (mixed in between the characters A-Z).
- *NIST<sub>remap</sub>*: In the first half, only the digits 0-9 exist. The input images are then replaced by the letters A-J for the second half, but the labels remain 0-9. There are 10 classes in this scenario.
- *NIST<sub>transfer</sub>*: The first 30k instances of the stream only consists of digits 0-9, while the following 80k are solely characters A-Z.

## 5.5 Preliminary Evaluation

In this section we give a summary of the work published in (Kauschke and Lehmann 2018). The work contains an empirical study on the prerequisites of *NN-Patching*. We establish heuristics for engagement layer attachment (Section 5.5.4) and patch architectures (Section 5.5.4). In Section 5.5.5 we investigate different ways of estimating the error of the base classifier and discuss three methods of training patches. These findings lead us to two variants of *NN-Patching*, that we then use for further evaluation.

### 5.5.1 Base Classifier Architectures

For *Patching*, we assume that a base classifier exists to learn errors and build patches upon. Since this classifier is not given in our case, we use the first part of the datasets to create it based on popular neural network architectures.

We examine two architectures that are generally suited to solve the scenarios we described: (i) FC-NNs, and (ii) CNNs. Each classifier

<sup>2</sup> <https://www.nist.gov/srd/nist-special-database-19>

architecture is tuned to achieve high accuracy on the unaltered datasets (Table 5.2).

The CNN and the FC-NN are trained on the *Init* fraction (cf. Figure 4.2, *Init*) of the dataset.

For the training in the initialization phase we use a batch size of 64.

Table 5.2: Base classifier accuracy on unaltered datasets.

Dataset	FC-NN	CNN
<i>MNIST</i>	98.87%	99.28%
<i>NIST</i>	94.07%	97.77%

In the following sections we show the architectural details and layer configurations of the chosen architectures.

### Fully-Connected Architectures

The fully-connected architectures for *NIST* and *MNIST* are stated in Table 5.3. The networks both tend to overfit, hence two dropout layers are utilized to counteract this problem. We use fully-connected layers with decreasing number of nodes to build the architectures. ReLU is chosen as activation function.

Table 5.3: Fully-Connected Architectures.

<i>MNIST</i>	<i>NIST</i>
Input(28x28) - Flatten() - Dropout(0.2) - FC(2048) - FC(1024) - FC(1024) - FC(512) - FC(128) - Dropout(0.5) - Softmax(#classes)	Input(128x128) - Flatten() - Dropout(0.2) - FC(1024) - FC(1024) - FC(768) - FC(512) - FC(512) - FC(256) - FC(256) - Dropout(0.5) - Softmax(#classes)
Input(i): Input layer, i = shape of the input Flatten(): Flatten input to one dimension FC(n): Fully Connected, n = number of units Dropout(d): Dropout, d = dropout rate Softmax(n): FC layer with softmax activation, n = number of units	

### Convolutional Architectures

For the CNN architectures, we additionally use convolution and pooling layers (Section 2.5.1). In the architecture for *MNIST*, only two convolutional layers and one pooling layer are required to achieve an accuracy greater than 99.25%. The *NIST* dataset has a total of  $128 \times 128 = 16,384$  attributes. We use one convolutional layer with stride=2 and two pooling layers to reduce the dimensionality of the data, while propagating through the network. In both cases we counteract overfitting with the help of two dropout layers. ReLU is chosen as activation function.

Table 5.4: Convolutional Architectures.

<i>MNIST</i>	<i>NIST</i>
Input(28x28) - Conv2D(32,(3,3),1) - Conv2D(64,(3,3),1) - MaxPooling((2,2),2) - Dropout(0.25) - Flatten() - FC(128) - Dropout(0.5) - Softmax(#classes)	Input(128x128) - Conv2D(32,(7,7),2) - MaxPooling((2,2),2) - Conv2D(64,(5,5),1) - Conv2D(64,(5,5),1) - Conv2D(64,(3,3),1) - Conv2D(64,(3,3),1) - MaxPooling((2,2),2) - Dropout(0.25) - Flatten() - FC(256) - Dropout(0.5) - Softmax(#classes)
Input(i): Input layer, i = shape of the input Flatten(): Flatten input to one dimension FC(n): Fully Connected, n = number of units Conv2D(f,k,s): 2D Convolution, f = number of filters, k = kernel size, s =stride MaxPooling(k,s): Max Pooling, k = kernel size, s =stride Dropout(d): Dropout, d = dropout rate Softmax(n): FC layer with softmax activation, n = number of units	

## 5.5.2 Patch Architecture Notation

In the following sections we will address the architecture of the patching networks via an abbreviated notation. Patches always consist of an input layer, some hidden layers, and a Softmax classification layer. The notation only depicts the hidden layers, excluding the obligatory input and Softmax for brevity.

For example, ‘256x128’ refers to a patch architecture with the following consecutive layers:

Input() - FC(256) - FC(128) - Softmax(num\_classes).

The architecture ‘128’ refers to:

Input() - FC(128) - Softmax(num\_classes).

## 5.5.3 Engagement Layer Selection

In *NN-Patching*, the output of the engagement layer is input to the patch network. Thus, the selection of the engagement layer impacts the adaptation performance. Choosing an engagement layer with no useful features, i. e., features that help in solving the given classification task, results in a low adaptation performance. Our goal is to find regularities and specify heuristics, with which we can simplify the selection task for *NN-Patching*. Therefore, we conduct a series of experiments on the different base network architectures.

The model used in the engagement layer selection experiments is a patch network that is trained on all instances, without any error estimator  $E$ . This way, we can observe the abilities of the patch in isolation, without the complex interrelation with the error estimator. We obtain an estimate of the maximum performance a patch can achieve, when attached to a specific layer.

Both neural network architectures, FC-NNs and CNNs are taken into consideration.

## Accuracy Progression

The goal of the accuracy progression diagram is to show the relation between the attachment layer of a patch and the respective accuracy. Figure 5.2 shows two examples of such a progression. In the diagram, the layers of the network are shown on the  $x$ -axis, and the resulting accuracy of a patch attached to that layer on the  $y$ -axis. For the figure, a ‘128’ patch was used on both a fully-connected and a convolutional architecture on the  $NIST_{flip}$  dataset.

The two network types exhibit a different behavior: While for the FC-NN it seems to be beneficial to attach to earlier layers in the network, the CNN shows more promising results for later layers in the network.

Yosinski et al. (2014) have observed a behavior in FC-NNs that they call *general* vs. *specific*: earlier layers in a FC-NN tend to encompass more general features, whereas later layers are rather specific towards the given classification task. For our example this means, that the patch is better to be engaged to the more general layers, and has to learn the actual classification part by itself.

The CNN (Fig. 5.2b) is different to the FC-NN: The later layers of the network are more suited for attaching a patch.

CNNs have also been found to create a hierarchical feature structure (Zeiler and Fergus 2013): They develop more general features in the earlier convolutional layers, and more task-specific features in the later layers. For our example, this means that the features provided by the later convolutional layers are still very useful for the given task, and more so than those provided by earlier convolution layers. We would argue, that in this case the target task (after the concept drift) is not significantly different from the original task, and so the higher level features work well. However, depending on the severity of the drift/task, this might not hold true for other situations.

Besides from these architecture-specific findings, we can observe a strong correlation between Average and Final Accuracy. The Final Accuracy is usually higher than the Average Accuracy, since the patch network has had more time to adapt to the new concept at the end of the stream. Aside from this difference, the ideal engagement layer is independent of the accuracy measure.

## Accuracy Progression on all Datasets

We applied the accuracy progression approach on all datasets for both architecture types. The top performing engagement layers are listed in Table 5.5.

The table indicates, that the ideal engagement layer for FC-NNs is either the first or second fully-connected layer. For convolutional networks, the ideal layer is either the last pooling layer or one of the two convolutional layers right before the pooling layer. This behaviour could be observed over all datasets.

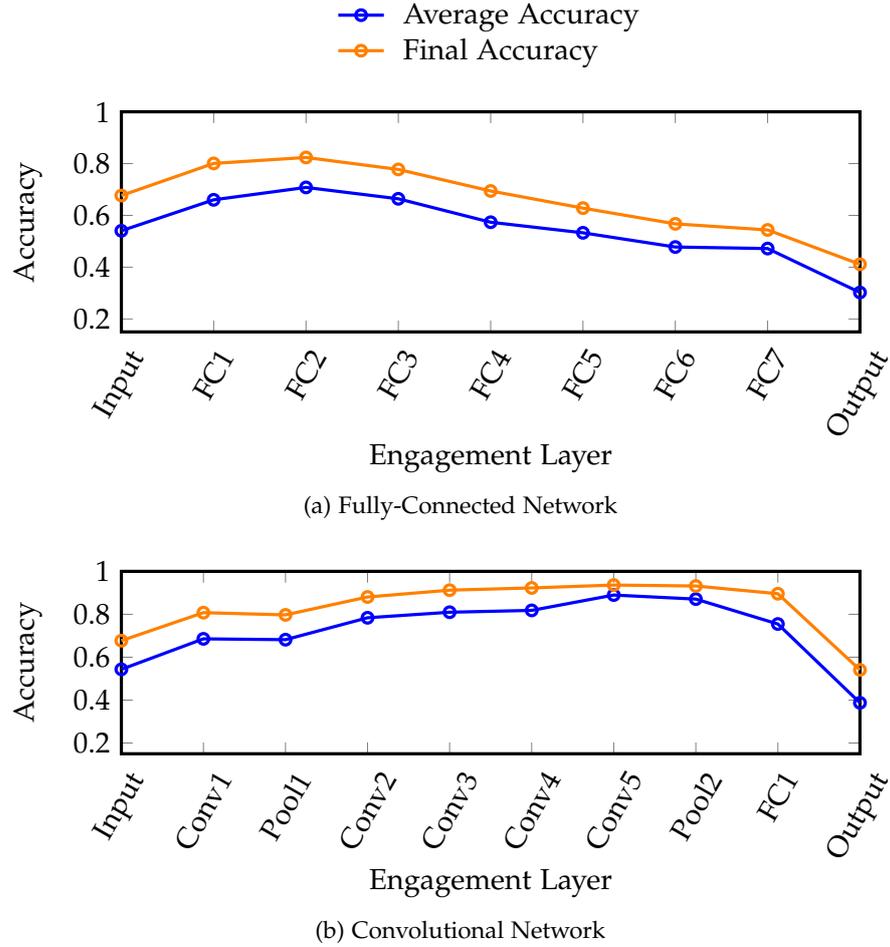


Figure 5.2: Accuracy Progression Diagram on the  $NIST_{flip}$  dataset.

### Heuristics for Engagement Layer Selection

Based on the results from Table 5.5, we infer the following heuristic rules for engagement layer selection:

**Fully-Connected Neural Network:** The best engagement layer is either the first or second fully-connected layer in the network.

**Convolutional Neural Network:** The best engagement layer is either the last convolutional layer or the last pooling layer of the network.

Selecting the best engagement layer is important, since it has a significant impact on the patch. These heuristics narrow down the search space for the optimal engagement layer to two layers. We found the best practice is, to try both candidate layers.

### 5.5.4 Patch Architecture

In order to allow *NN-Patching* to perform well, we investigate the influence of different patch architectures on the patching performance.

Table 5.5: Best engagement layer per dataset. The patch user here had a single hidden layer with 128 units.

	Fully-Connected		Convolutional	
	Avg. Acc.	F. Acc.	Avg. Acc.	F. Acc.
<i>MNIST<sub>appear</sub></i>	FC1	FC1	Pool1	Pool1
<i>MNIST<sub>flip</sub></i>	FC1	FC1	Pool1	Pool1
<i>MNIST<sub>remap</sub></i>	FC1	FC1	Pool1	Conv2
<i>MNIST<sub>rotate</sub></i>	FC1	FC1	Conv2	Pool1
<i>MNIST<sub>transfer</sub></i>	FC1	FC1	Conv2	Conv2
<i>NIST<sub>appear</sub></i>	FC2	FC2	Conv5	Conv5
<i>NIST<sub>flip</sub></i>	FC2	FC2	Conv5	Conv5
<i>NIST<sub>remap</sub></i>	FC1	FC1	Conv5	Conv5
<i>NIST<sub>rotate</sub></i>	FC2	FC2	Conv5	Pool2
<i>NIST<sub>transfer</sub></i>	FC1	FC1	Conv5	Conv5

On top of determining the ideal engagement layer, we have to create the patch  $P$  itself as a neural network.

The architecture of the patch depends on the size of the engagement layer, the amount of data we are dealing with, and the given task. Having more complex patches allows for better capabilities w. r. t. approximating the target function, but also requires more training instances to properly learn the adaptation.

We evaluate 25 patch architectures on all datasets for both FC-NN and CNN architectures. The engagement layer selection is based on the heuristics from Section 5.5.3. We run the experiments on both candidate layers, and average the results over all datasets for each patch architecture. The patch architectures have from one to three fully-connected hidden layers.

The patches are trained without the error estimator, as in the previous experiment.

Table 5.6 shows the ranked results of patch architectures over all datasets. The results of the patches with three hidden layers are not displayed here, because they ranked lower than patches with two hidden layers.

The table shows, that the architectures ‘512’, ‘1024’, ‘1536’, and ‘2048’ are the top 4 architectures for **all** evaluation measures. Therefore, we conclude:

1. Patches do not require more than one hidden layer (for our purposes): Almost all single-hidden-layer patches perform better than the multi-layered patches.
2. Patches do not require high numbers of nodes. All of the top 4 architectures show very similar results. The only significant drop can be observed with hidden layers of less than 512 nodes.

Table 5.6: Top Ranked Patch Architectures – Grouped by Evaluation Measure.

		Average Accuracy		Final Accuracy		Recovery Speed	
Rk.	Patch Arch.	Avg.Rk.	Patch Arch.	Avg.Rk.	Patch Arch.	Avg.Rk.	
1.	1024	4.07	2048	5.57	1024	4.43	
2.	512	4.1	1024	5.92	1536	6.37	
3.	2048	4.25	512	6.28	2048	6.6	
4.	1536	4.28	1536	6.42	512	7.83	
5.	256	5.4	256	7.78	256	7.92	
6.	128	7.83	1024x512	9.87	128	8.6	
7.	512x256	9.6	128	10.08	1024x512	9.68	
8.	1024x512	10.33	1024x256	10.17	1024x256	9.75	
9.	1024x256	10.4	2048x256	10.33	1536x256	10.58	
10.	512x128	10.6	1536x256	10.55	1536x512	10.92	
11.	1536x256	10.95	1536x512	10.65	512x256	12.17	
12.	1536x512	11.0	512x256	10.65	2048x256	12.18	
13.	256x128	11.28	512x128	10.67	2048x512	12.47	
14.	2048x512	11.75	2048x512	11.2	512x128	12.9	
15.	2048x256	11.8	256x128	11.92	256x128	13.7	

Apparently, the task that a patch has to learn is rather simple. The attachment layers provide features with sufficient generality, that the patch itself is not required to learn further abstractions from those features. Otherwise, multi-layered patches would have outperformed the single-layered patches.

For the rest of the chapter, we use patches with a single hidden layer with 512 nodes. This architecture performs well, is easier to train than deeper architectures and has no significant drawbacks w. r. t. the performance metrics.

### 5.5.5 Training and Error Estimation

One of the key aspects of the *Patching*-idea is the error estimation. A patch will only work properly, when the error estimator assigns it instances equal to what the patch has been trained on. In the preliminary evaluation we established architectural aspects of patches without taking this error estimator into account. The error estimator is a neural network which is trained on the stream data to identify instances that the base model will misclassify. It is improbable, that it can solve this task perfectly, e.g., when a new concept has emerged and the estimator has had little time to adapt. At the end of each batch of instances, we receive the true labels. These labels and the current prediction of the estimator can be used to define which instances the patch should be trained upon.

We investigate three ways of training the patch:

1. **Exclusive:** only truly erroneous instances are used for patch training (determined by their true label).

2. **Inclusive:** all available instances are used for patch training.
3. **Semi-Exclusive:** truly erroneous instances are used in union with all instances that  $E$  currently assumes as erroneous.

Especially at the beginning of a new concept, when the classification of the error estimator is not yet refined, it can be useful to train  $P$  with more information about the general population. In the following we elaborate on training the error estimator, the three ways of training the patches, and show experimental results.

### Training the Error Estimator

The error estimator  $E$  is trained to predict if an instance is misclassified by  $M$ . This is a binary classification problem, with the target function being the error region of  $M$ . More specifically,  $E$  is trained with the instances  $\mathbf{x}$  from the most current batch of data ( $\mathbf{x} \in D_i$ ), which are labeled with  $l_i(\mathbf{x})$  in  $D_i$ , and which for this purpose are re-labeled as

$$e_i(\mathbf{x}) = \mathbb{1}(M(\mathbf{x}) \neq l_i(\mathbf{x})). \quad (5.1)$$

After training on  $D_i$ ,  $E$  makes correct predictions with a probability  $\Pr(E)$ .

Additionally, we investigate a different approach to error estimation: Instead of estimating the error of  $M$ , we estimate the error of the patch. The idea is, that the patch is trained on *all* new data, and the error estimator predicts how well it can handle an unseen instance.

Estimating the error for both  $M$  and  $P$  is a difficult task:

Let's assume that, right after a concept drift, a new, stable concept prevails. The estimator is now supposed to model the error of  $M$ . Since the concept is stable, this is stationary learning task. As proposed by the Probably Approximately Correct (PAC) learning model (Valiant 1984), with more data,  $E$  will converge to the true error region.

When learning the error of  $P$ , the problem is different:

Since the patch has to adapt to the new concept, which may require multiple batches of data, its error region is initially changing rapidly. The error estimator will therefore divert many instances to the base classifier. Over time, error on the patch decreases and it can take over the classification. Additionally, the patch has to learn a more complex task, since it is trained to classify all instances.

In general, we assume that it is advantageous to train on a stationary concept, so the error prediction of  $M$  should be the better variant for this scenario. However, when the underlying concept drift is gradual and constantly changing, this assumption might prove wrong.

The resulting two versions of error estimation are:

1. **Model Error Estimation (ME):**  $E$  estimates, if  $M$  is likely to err for an example  $\mathbf{x}$ .
2. **Patch Error Estimation (PE):**  $E$  estimates, if  $P$  is likely to err for an example  $\mathbf{x}$ .

## Training the Patch

In *Patching* (Section 4.3.2), the error estimation is split into separate error regions, which are then treated separately. A patch is trained on all instances inside a region, *exclusively*. Especially when a concept drift has just occurred in a data stream, the regions can be imprecise or affected by residual examples of the previous concept. As we are iteratively refining the error estimator in *NN-Patching*, it is important that we cannot rely on data from previous data batches as *Patching* did. Only the most recent batch  $D_i$  is available for the training phase. The advantage of this is, the error estimator does not need to be trained from scratch for every update. This speeds up the learning process, and iteratively refines the error estimators as well as the resulting patches. We propose three learning schemes for *NN-Patching*:

*Exclusive Patch Training.* Exclusive training is similar to *Patching*, but it differs in that there are no separate error regions, only one general error estimator that identifies all regions.

Exclusive training suffers from incorrect predictions: if  $\Pr(E)$  is low, the estimation differs vastly from the true errors. Figure 5.3a illustrates this problem. It can lead to the patch being trained on partially wrong data. However, the estimation and the true error will converge over time, after more training data from the new concept was learned, or the concept has stopped drifting.

*Inclusive Patch Training.* Inclusive patch training (Figure 5.3b) trains the patch on all available data from the last batch. Via this, initial erroneous predictions of  $E$  become less relevant, since the patch has also learned instances outside the estimated error region. However, the patch has to learn a more complex function, since it solves the complete classification problem instead of only a partial problem. This somewhat contradicts the general idea of patching: splitting up the complex decision task into smaller, patchable subtasks.

*Semi-Exclusive Patch Training.* The *Semi-Exclusive* method (Figure 5.3c) uses the idea of the *Exclusive* training method, but extends the training data with instances from the true error region. Initially, before  $E$  has converged to the true error region, it causes instances from the estimated error region to be classified by  $P$ . However, in order to prepare  $P$  for the case when  $E$  has converged, the additional training on the true errors improves the resulting patch.

## Results on Patch Training Variants

We conducted an experimental analysis on all combinations of error estimation methods with the patch training paradigms. Six variants of *NN-Patching* emerged:

1. *NN-Patching*( $ME$ )<sub>incl</sub>: Model error evaluation paired with *Inclusive* patch training.

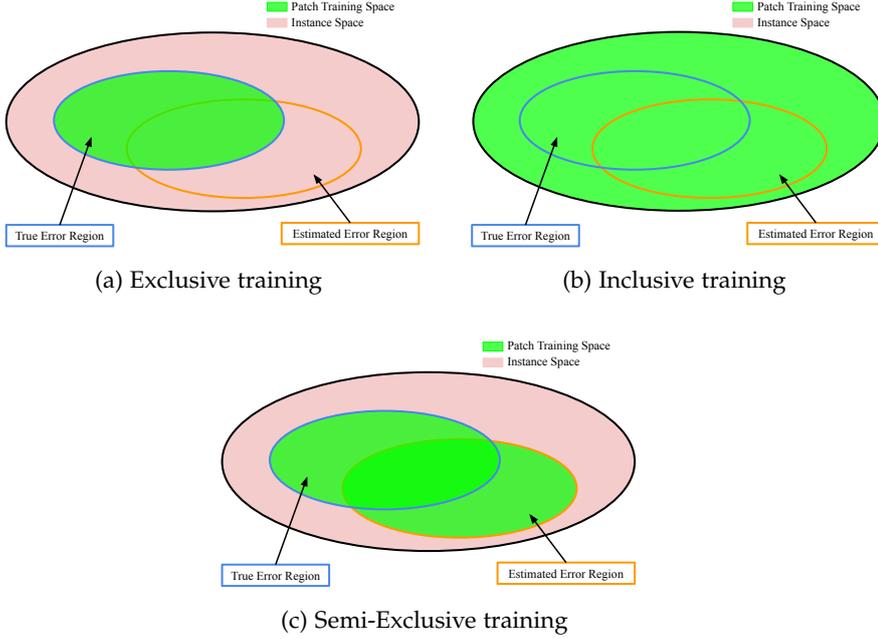


Figure 5.3: Patch training paradigms: *Inclusive*, *Exclusive* and *Semi-Exclusive* patch training.

2.  $NN\text{-Patching}(ME)_{semi}$ : Model error evaluation paired with *Semi-Exclusive* patch training.
3.  $NN\text{-Patching}(ME)_{excl}$ : Model error evaluation paired with *Exclusive* patch training.
4.  $NN\text{-Patching}(PE)_{incl}$ : Patch error evaluation paired with *Inclusive* patch training.
5.  $NN\text{-Patching}(PE)_{semi}$ : Patch error evaluation paired with *Semi-Exclusive* patch training.
6.  $NN\text{-Patching}(PE)_{excl}$ : Patch error evaluation paired with *Exclusive* patch training.

We compared the methods on *MNIST* and *NIST* datasets with both fully-connected and convolutional network architectures, and found the following: When predicting the error for the base classifier, the *Semi-Exclusive* method yields the best results. When predicting the error for the patch, the *Inclusive* method is superior (Table 5.7). The *Exclusive* methods were inferior in all cases. The complete results can be found in Appendix A.1.

The results verify our assumptions about the patch training process: It is helpful for the patch to learn from both the estimated error region and the true error region. However, this only holds true when we are applying the model error estimation. For the patch error estimation, it is the exact opposite. This was to be expected, since we assumed that initially the patch error estimator will give bad predictions. If

Table 5.7: Number one ranks of *NN-Patching* variants in preliminary evaluation (on 20 datasets).

Model	A.Acc	F.Acc	R.Spd
<i>NN-Patching</i> ( <i>ME</i> ) <sub>incl</sub>	1	3	1
<i>NN-Patching</i> ( <i>ME</i> ) <sub>semi</sub>	<b>9</b>	3	4
<i>NN-Patching</i> ( <i>ME</i> ) <sub>excl</sub>	0	0	1
<i>NN-Patching</i> ( <i>PE</i> ) <sub>incl</sub>	8	<b>10</b>	<b>8</b>
<i>NN-Patching</i> ( <i>PE</i> ) <sub>semi</sub>	2	4	0
<i>NN-Patching</i> ( <i>PE</i> ) <sub>excl</sub>	0	0	1

this happens, it is beneficial to have the patch trained on all available instances, because when the classification is diverted to the patch by a wrong prediction, its chances are higher to handle the classification.

Based on these findings, we channel our further observations on the *NN-Patching*(*ME*)<sub>semi</sub> and *NN-Patching*(*PE*)<sub>incl</sub> training schemas.

## 5.6 Experimental Setup

In the preliminary experiments in Section 5.5 we evaluated patch architectures with varying input sizes, layer depths, patch training methods, and error estimators. The experiments showed that a patch architecture with one hidden layer with 512 units and two dropout layers surrounding it performs reasonably well in all experiments. The patches use fully-connected layers with ReLU activations. However, keep in mind that for certain problems, specific architectures such as convolutional layers might be better suited. Moreover, we need to learn the error detector function  $E$ , for which we will provide two approaches as summarized in the following:

- *NN-Patching*<sub>me</sub>: This variant combines the model  $M$ , the *Model Error Estimation* (ME) estimation function  $E$  and the patching network  $P$ , which is trained via the *Semi-Exclusive* patch training paradigm. Both, patch and estimator are trained with new examples after each new batch of instances, and are initialized with random weights.  $P$  is a fully-connected neural network with an input layer the size of the engagement-layer, contains one hidden layer with 512 units, surrounded by two dropout layers (Dropout(0.25)-FC(512)-Dropout(0.5)) and the output layer. We chose the same network architecture for  $E$  as for  $P$ , the only difference being that  $E$  only does binary classification.
- *NN-Patching*<sub>pe</sub>: This variant replaces the error decision for the base network with the *Patch Error Estimation* (PE) function  $E_p$ .  $E_p$  does not estimate the ability of  $M$  to classify an instance, but instead estimates if the patch  $P$  is (already) able to do so. The patch is trained with *Inclusive* patch training. Especially in the

Table 5.8: Network architectures for base neural networks in the experiments.

<b>Architecture: Fully Connected</b>
Dropout(0.2) - FC(2048) - FC(1024) - FC(1024) - FC(512) - FC(128) - Dropout(0.5) - Softmax(#classes)
<b>Architecture: Convolutional</b>
Conv2D(7,7) - MaxPool(2,2) - Conv2D(5,5) - Conv2D(5,5) - Conv2D(3,3) - Conv2D(3,3) - MaxPool(2,2) - Dropout(0.25) - FC(256) - Dropout(0.5) - Softmax(#classes)
FC(n): Fully Connected, n = number of units Conv2D(k): 2D Convolution, k = kernel size Dropout(d): Dropout, d = dropout rate MaxPool(k): MaxPooling, k = kernel size Softmax(n): FC layer with softmax activation (n units)

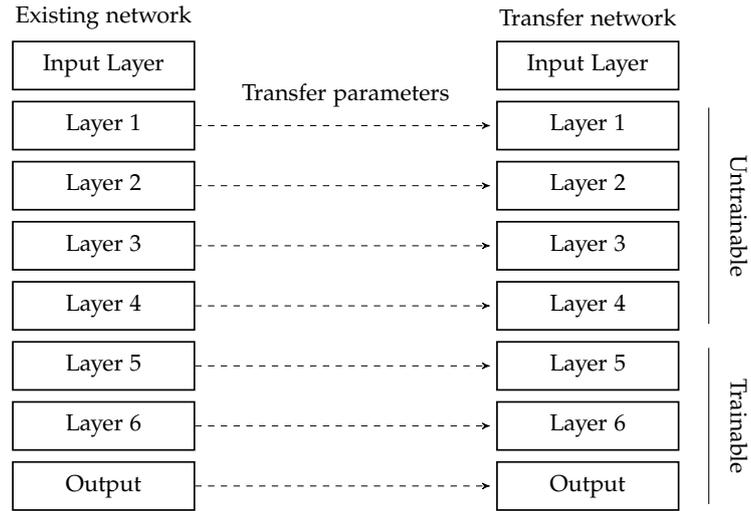
first batches of adaptation, when  $P$  is not yet well trained, it can often be better to use  $M$ , although  $E_p$  indicates that  $M$  is flawed.  $E_p$  is a neural network equal to  $P$  (one hidden layer with 512 units, surrounded by dropout), but contains a sigmoidal output for estimating probabilities on the correctness of  $P$ . The classification is diverted to  $P$ , if  $E_p$  is confident.

The patch  $P$  and the error estimators  $E$  and  $E_p$  are trained on the most recent batch of data. When a new batch arrives, they are updated with that information and hence evolve according to what is the current concept.

### 5.6.1 Benchmark Algorithms

In this section, we elaborate on the benchmark algorithms we compared against. The base classifier is the same for all compared methods. In the case of *MNIST*, it is a fully-connected deep network (see Table 5.8) with ReLU activations. For *NIST*, it is a convolutional neural network with multiple convolutional layers followed by max-pooling and fully-connected layers for classification. Both architectures were trained with the AdaDelta optimizer and the categorical cross-entropy loss function. For the given datasets, these architectures work sufficiently well to demonstrate the abilities of *NN-Patching*. Transfer learning techniques are commonly used in ML to adapt an existing model to a new environment. We will compare our efforts against two well-known approaches from related work, and a non-adaptive baseline.

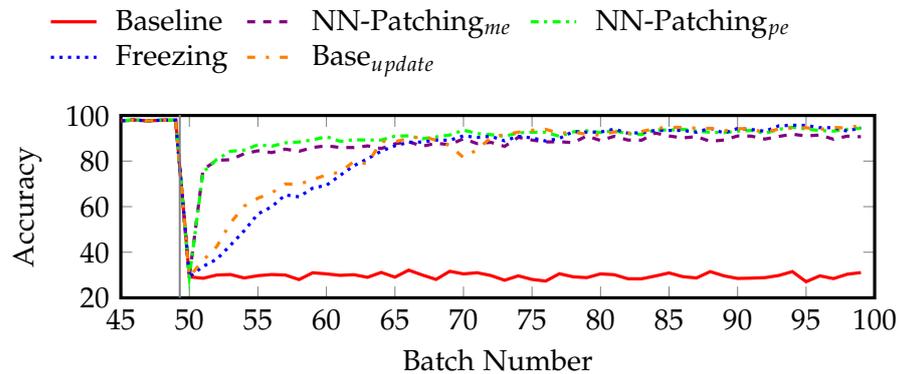
**Freezing:** *Freezing* follows the approach of Oquab et al. (2014) by retraining the last layers of the network. The parameters from the pre-existing network are copied to the transfer network. The transfer network has the same architecture as the pre-existing network. The first layers of the transfer network are non-trainable.

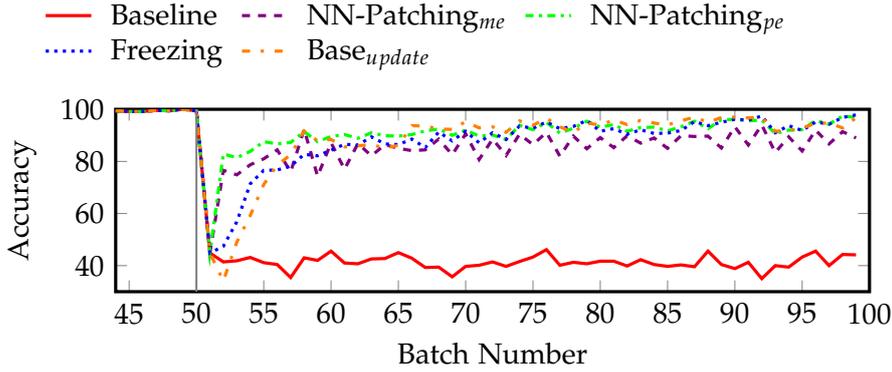
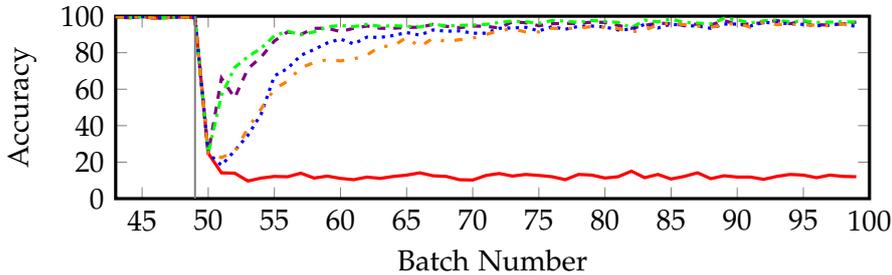
Figure 5.4: Transfer learning with neural networks: *Freezing*.

The latter layers of the transfer network will be trained to adapt (Figure 5.4). This is also sometimes known as pre-training. To compare this method to *NN-Patching*, all layers including the engagement layer are non-trainable. In contrast to *NN-Patching*, the initialization of the trainable layers is not at random, but adopting the weights from the base classifier.

***Base<sub>update</sub>***: The whole base classifier is trained. All weights and parameters are trainable. This approach has the highest number of trainable parameters, hence the model capacity to represent concepts is also high. This approach can also be regarded as a special case of transfer learning, where all layer weights are trainable.

**Baseline**: The original network model  $M$ . This is the non-adaptive baseline for comparison.

Figure 5.5: Results on the  $MNIST_{flip}$  dataset as a data stream.

Figure 5.6: Results on the  $MNIST_{remap}$  dataset as a data stream.Figure 5.7: Results on the  $NIST_{remap}$  dataset as a data stream.

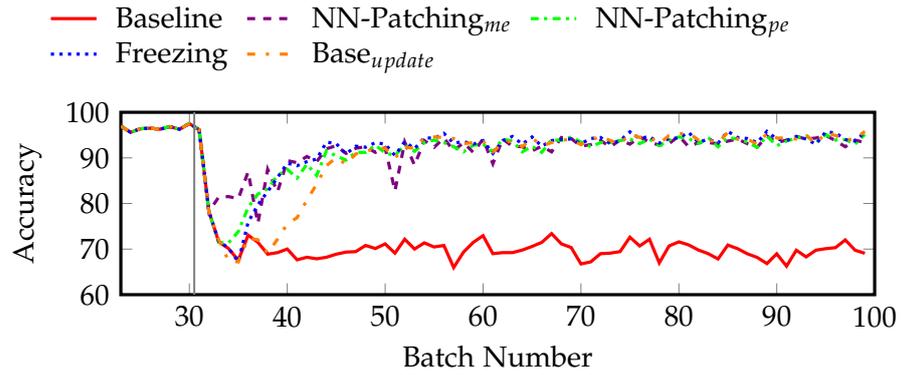
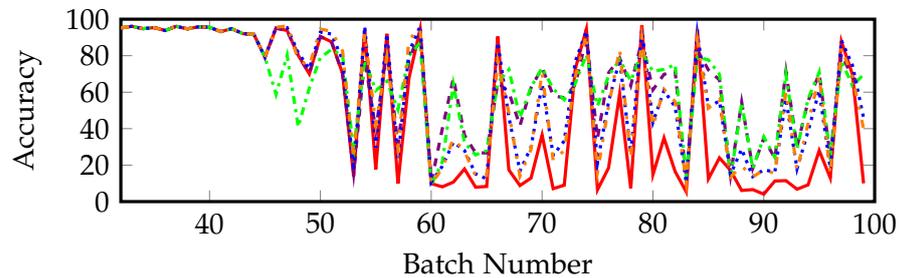
## 5.7 Results

We conducted experiments on nine problems with varying types of concept drift or transfer (see Section 5.4.2). We ran each algorithm ten times on every dataset with different random seeds for the base and patch networks and calculated the average results. The standard deviation of accuracy over these ten runs was always less than 2% in both adaptation and final accuracy for all adapting methods (excluding the non-adaptive Baseline).

### 5.7.1 Deep Networks – The $MNIST$ dataset

On the  $MNIST$  datasets, all adapting algorithms are able to achieve similar results for the final accuracy. The network architecture used here is a deep, fully connected network.

$NN-Patching_{me}$  and  $NN-Patching_{pe}$  achieve the best adaptation ranks in all four datasets, as can be seen in Table 5.9. The  $Base_{update}$ -method profits from the higher capabilities (all weights are being trained) and achieves a high final accuracy in three datasets, albeit paired with a slow adaptation. The example of  $MNIST_{flip}$  in Figure 5.5 shows, that determining the error on the patch side in  $NN-Patching_{pe}$  can be beneficial compared to  $NN-Patching_{me}$ . We suppose that in this case,

Figure 5.8: Results on the  $NIST_{appear}$  dataset as a data stream.Figure 5.9: Results on the  $NIST_{rotate}$  dataset as a data stream.

more classification decisions get diverted to the patch faster, because the patch error estimator has a high confidence.

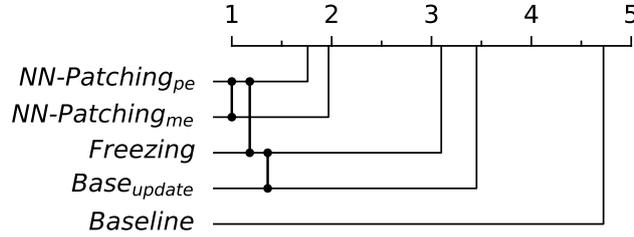
## 5.7.2 Convolutional Networks – The $NIST$ dataset

In the  $NIST$  datasets, the baseline architecture is a convolutional network. As shown in the example plot in Figure 5.7, both  $NN-Patching$ -variants adapt significantly faster than the other methods. In Table 5.10, we can see that in all of the  $NIST$  datasets,  $NN-Patching_{me}$  or  $NN-Patching_{pe}$  achieve the top rank in the adaptation phase four out of five times.  $NN-Patching_{pe}$  also achieves the highest final accuracy in three out of five datasets. While in the  $MNIST$ -datasets, the  $Base_{update}$  algorithm could benefit from having more trainable weights, in  $NIST$  this advantage vanishes. We suppose that if the streams were of arbitrary length, the advantage of being able to have more trainable weights would show, at least if the concept remains stable.

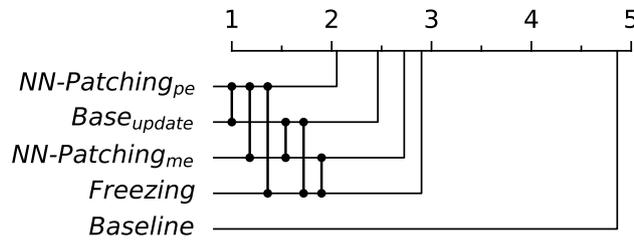
$Freezing$  exhibits similar results. It has a medium amount of trainable weights, and should theoretically be a compromise between adaptation speed and overall capabilities to encompass complex concepts. But in the  $NIST$  streams, it is mostly on par with  $Base_{update}$ .

Regarding the recovery speed, both  $NN-Patching$  variants are significantly faster compared to  $Freezing$  and  $Base_{update}$ . In the  $NIST_{rotate}$

dataset, none of the methods provide a reasonable adaptation beyond the baseline. This is caused by the gradual concept drift in this dataset, and shows in Figure 5.9 in the high variance of accuracy.



(a) Adaptation Rank



(b) Final Rank

Figure 5.10: Friedman test with Wilcoxon signed-rank test for pairwise comparison.

### 5.7.3 Significance Testing

The results of the Friedman test combined with the Wilcoxon signed-rank test to establish pairwise comparisons are shown in Figures 5.10a and 5.10b. The algorithms are ordered by average rank. As the figure indicates,  $NN\text{-Patching}_{me}$  significantly differs in adaptation rank from any non-patching method.  $NN\text{-Patching}_{pe}$ , however, is not significantly different from  $Freezing$ .

Regarding the final rank, all adapting methods do not significantly differ from each other.

## 5.8 Conclusion

In this chapter, we proposed a version of *Patching* that is specifically tailored to neural networks. We investigated various approaches on the construction of a neural patch, and multiple ways of learning an error estimator. We found, that for different scenarios, different patch architectures may be required, but a simple architecture works reasonably fine, if the base classifier provides a sufficient input, i. e., features that are related to the target task. In this case, the patch

can adapt the classification just enough to solve the problem, and does this very efficiently. Another challenge is finding an appropriate layer, where the patch can attach to. Via experimental evaluation, we concluded heuristic rules for different types of networks, in this case fully-connected and convolutional network architectures.

Finally, we evaluated two *NN-Patching*-variants against state-of-the-art methods in adaptive neural network training. Our evaluation is based on data-streams that incorporate concept drift in the form of partial changes, up until completely new scenarios, which fall in the domain of transfer learning.

Network patching can profit from including information from the inner layers of a given network into the patch network. Because the patch size in our experiment is small (one hidden layer with 512 neurons, and dropout surrounding it), it is quick to compute and also attained a very fast concept adaptation. However, the small patch size should lead to an impaired performance in difficult tasks, since a patch is limited in its capability to encompass new concepts. Interestingly, this only occurs rarely in our experiments. It shows that the updating the whole network (*Base<sub>update</sub>*) can leverage the higher network complexity compared to the other methods in some datasets, but the real advantage is limited. Estimating not only the error of the base classifier, but instead the current performance of the patch increases the accuracy for some scenarios, as the results of *NN-Patching<sub>pe</sub>* indicate.

Because of its design, neural network patching is resilient against catastrophic forgetting, while at the same time allowing significantly faster adaptation for non-stationary data. It can be applied on concept drifting streams as well as transfer situations, and is generally easy to use. In contrast to the original patching method, neural network patching iteratively refines both the error decision/model selection and the patch, hence saving computational effort. However, the complexity of learning patches and error estimators is not *online* capable, but that was never the intention. Opposed to many concept drift learning methods, neural network patching does not require explicit drift detection. This is handled implicitly by the error estimator.

However, the method introduces new hyperparameters such as the patch architecture and engagement layer, which need to be tuned for the scenario at hand.

Table 5.9: Results for *MNIST* (Base classifier is a fully connected deep network)

Classifier	A.Acc	F.Acc	R.Spd	Ad.Rk	F.Rk
<i>MNIST<sub>appear</sub></i>					
Baseline	50.55	51.01	—	4.42	5
<i>NN-Patching<sub>me</sub></i>	87.77	94.79	16	1.61	3.06
<i>NN-Patching<sub>pe</sub></i>	<b>87.87</b>	95.12	<b>13.4</b>	<b>1.55</b>	2.18
<i>Freezing</i>	79.88	94.58	32	3.49	3.14
<i>Base<sub>update</sub></i>	79.37	<b>95.61</b>	28.8	3.93	<b>1.62</b>
<i>MNIST<sub>flip</sub></i>					
Baseline	29.64	29.39	—	5	5
<i>NN-Patching<sub>me</sub></i>	87.97	90.89	14.5	2.03	3.98
<i>NN-Patching<sub>pe</sub></i>	<b>90.68</b>	93.68	<b>7.3</b>	<b>1.09</b>	2.7
<i>Freezing</i>	83.91	94.23	15.9	3.81	1.96
<i>Base<sub>update</sub></i>	85.16	<b>94.63</b>	14.3	3.07	<b>1.36</b>
<i>MNIST<sub>remap</sub></i>					
Baseline	39.67	40.91	—	4.99	5
<i>NN-Patching<sub>me</sub></i>	86.65	89.45	10.6	2.88	3.9
<i>NN-Patching<sub>pe</sub></i>	<b>92.04</b>	95.36	<b>4.9</b>	<b>1.23</b>	2.08
<i>Freezing</i>	88.47	94.93	13.7	3.27	2.16
<i>Base<sub>update</sub></i>	89.68	<b>95.43</b>	7	2.63	<b>1.86</b>
<i>MNIST<sub>transfer</sub></i>					
Baseline	0	0	—	5	5
<i>NN-Patching<sub>me</sub></i>	<b>91.73</b>	<b>95.39</b>	<b>6</b>	<b>1.37</b>	<b>1.44</b>
<i>NN-Patching<sub>pe</sub></i>	91.63	95.26	<b>6</b>	1.63	2.02
<i>Freezing</i>	71.22	93.25	28.2	3.34	3.64
<i>Base<sub>update</sub></i>	67.42	93.70	28.2	3.66	2.9

Table 5.10: Results for NIST (Base classifier is a convolutional network)

Classifier	A.Acc	F.Acc	R.Spd	Ad.Rk	F.Rk
<i>NIST<sub>appear</sub></i>					
Baseline	69.82	70.11	—	4.64	5
<i>NN-Patching<sub>me</sub></i>	<b>91.55</b>	93.85	<b>4.7</b>	<b>1.77</b>	3.52
<i>NN-Patching<sub>pe</sub></i>	91.38	94.38	6.5	2.23	3
<i>Freezing</i>	91.40	94.95	7.5	2.45	1.84
<i>Base<sub>update</sub></i>	90.33	<b>95.03</b>	11.2	3.91	<b>1.64</b>
<i>NIST<sub>flip</sub></i>					
Baseline	17.72	18.10	—	4.96	5
<i>NN-Patching<sub>me</sub></i>	89.18	92.29	8.4	1.67	1.96
<i>NN-Patching<sub>pe</sub></i>	<b>90.52</b>	<b>93.75</b>	<b>7.6</b>	<b>1.33</b>	<b>1.04</b>
<i>Freezing</i>	71.3	88.4	36.8	3.08	3.6
<i>Base<sub>update</sub></i>	65.6	88.58	40	3.96	3.4
<i>NIST<sub>remap</sub></i>					
Baseline	11.48	11.27	—	5	5
<i>NN-Patching<sub>me</sub></i>	92.63	95.85	5.5	1.67	2.54
<i>NN-Patching<sub>pe</sub></i>	<b>93.46</b>	<b>96.90</b>	<b>5.2</b>	<b>1.33</b>	<b>1.24</b>
<i>Freezing</i>	85.92	95.21	13.7	3.21	3.42
<i>Base<sub>update</sub></i>	83.74	95.7	20.1	3.79	2.8
<i>NIST<sub>rotate</sub></i>					
Baseline	39.13	43.09	—	3.49	3.74
<i>NN-Patching<sub>me</sub></i>	<b>59.83</b>	61.22	—	3.27	2.64
<i>NN-Patching<sub>pe</sub></i>	58.53	<b>61.93</b>	—	3.9	<b>2.62</b>
<i>Freezing</i>	52.42	58.48	—	<b>2.13</b>	2.72
<i>Base<sub>update</sub></i>	51.28	56.64	—	2.21	3.28
<i>NIST<sub>transfer</sub></i>					
Baseline	0	0	—	5	5
<i>NN-Patching<sub>me</sub></i>	87.88	93.67	18.5	<b>1.45</b>	<b>1.5</b>
<i>NN-Patching<sub>pe</sub></i>	<b>87.90</b>	<b>93.71</b>	<b>17.8</b>	1.55	1.58
<i>Freezing</i>	80.28	91.46	32.4	3.1	3.62
<i>Base<sub>update</sub></i>	71.59	91.79	40.6	3.9	3.3

## Ensembles of Patches

---

In the previous chapters we have shown a general approach to dealing with classification under concept drift: the *Patching* algorithm. We explained the general framework as well as a specific variant of patching, which is especially geared towards neural networks. The general idea of *Patching* is to find out, if the given classifier will misclassify an instance or not, and then fix these potential errors via patches: smaller, local models that are used instead of the original classifier. The error estimation is done via a separate classifier, and the patch itself is a classifier.

In the case of *Patching*, the estimator learns axis-parallel regions in the instance space, where the classifier predominantly errs. Each of those regions is then fixed by a single patch. In the case of *NN-Patching*, the estimator primarily learns more complex error boundaries, and one patch to account for them all. We found, that learning the error estimator can be a task that is as difficult as learning to fix the error itself. In this chapter, we investigate means to simplify the estimation task and create an error estimator that can be trained faster with fewer examples. Also, we would like our method to handle recurring concepts more easily. In long data streams that undergo constant changes, a recurrence of certain concepts is likely. The patching-variants we showed so far do not implicitly deal with this issue. They have to learn the recurring concepts over and over again when they appear. This is costly and implies reduced performance during that re-training period. Another issue that we would like to address are constraints regarding computational efficiency. The *Patching* and *NN-Patching* algorithms were not specifically designed to learn in resource-restricted scenarios. However, in high-speed data stream learning this can be a requirement. *Patching* requires to temporarily store a set of the most recent  $n$  batches, in order to adapt to concept drift. In this chapter, we will see to that problem by only working with the most recent batch of data, which is discarded after the learning step.

In order to fix these issues, we propose *Ensemble Patching*, an ensemble learning approach that fixes errors of a given model via patches. These patches are added to the ensemble when the base learner shows decreased performance for one or multiple classes, which we discover by applying explicit drift detection. Each patch covers the drift of a subset of the classes. Via the drift detection method, the misclassification behavior of the base learner can be matched to an existing patch. This reduces the total amount of patches to the number of seen concept drifts, which is an advantage in the long run. Patches can be added to the ensemble and refined incrementally. The ensemble decision is conducted by a sequence classifier, which allows a high number of

ensemble members that can be added incrementally. Experimental results show that ensemble patching performs well w. r. t. accuracy and adaptation speed<sup>1</sup>.

First, we will give an introduction into our idea in Section 6.1. Section 6.2 gives a formalization of the problem, followed by related work in Section 6.3. In Section 6.4 we introduce a novel drift detection method, and compare it against similar algorithms. In Section 6.5 we explain the *Ensemble Patching* algorithm, elaborate on the experiment setup in Section 6.6, and give experimental results in Section 6.7. Finally, we conclude our findings in Section 6.8.

## 6.1 Introduction

Dealing with data streams that incorporate concept drift is a well researched field. Seminal work (Schlimmer and Granger 1986; Widmer and Kubat 1996) introduces incremental learning and adaptation to concept drift. While learning speed is a major concern in this domain (Domingos and Hulten 2000), recent research focuses on ensemble methods (Brzezinski and Stefanowski 2011; Oza 2005; Wang et al. 2003) to improve the prediction quality. Employing ensembles of classifiers has its challenges: determining the ensemble decision, and adding/removing members from the ensemble when necessary. Each of which is not trivial, when data is nonstationary and every decision taken today might prove wrong tomorrow.

In this chapter we built upon the idea of classifier patching, where the goal is to correct the mispredictions of an existing classifier via smaller classifiers—so-called patches. This method is especially useful when the base classifier is only partially wrong. It avoids the potentially expensive and time-consuming re-training of the classifier. We approach the situation with an ensemble of patches, each of which can be trained incrementally. The ensemble decision is learned via a neural network, which gives a stacked decision (Wolpert and Macready 1996) based on the base classifier and the decisions of the patches. Patches will be added when necessary, and updated when feasible. This allows for a quick adaptation to new and recurring concepts. Through the way the ensemble is constructed, it is especially useful for dealing with recurring concepts, since patches remain in the ensemble and can be reactivated when required.

---

<sup>1</sup> This chapter is based on Kauschke, Fleckenstein, and Fürnkranz (2019), where this work was published at the IJCNN conference in 2019.

## 6.2 Problem Description

The problem description remains the same as in the previous chapters (cf. Section 4.3.1).

For a stream  $D_i$ ,  $i \in 0..I$  with a potentially infinite number of batches  $I$ , we receive new batches of examples  $D_i$ , for which the existing classifier  $M$  makes imperfect predictions. Presumably because the labeling function  $l_i$  underlying  $D_i$  is slightly different from the function  $l$ , which  $M$  is approximating.

Our goal is to learn a classifier ensemble that approximates  $l_i$  as close as possible by combining  $M$  with additional classifiers  $P_j$ ,  $j \in 0..n$  via an ensemble  $\mathbb{C}$ , such that

$$\Pr(\mathbb{C}(\mathbf{x}) = l_i(\mathbf{x})) \geq \Pr(M(\mathbf{x}) = l_i(\mathbf{x})).$$

## 6.3 Related Work

We elaborate on related work in the domain of ensembles in concept drift in Section 3.5.2. In this section, we give a brief overview on the most relevant work for this chapter: adaptive learners with the capability to learn recurrent concepts.

One of the most common strategies to adapt an ensemble of classifiers to non-stationary environments is to weigh each member based on its performance on the current environment. By maintaining a set of members and changing the weights, the ensemble can be adapted for the current concept.

The AWE (Wang et al. 2003) adds one classifier for each new batch of data and keeps them until a fixed size  $n$  is reached. Weights are assigned based on the difference between the *Mean Squared Error* (MSE) of a randomly predicting classifier and the MSE of each ensemble member on the most recent data. The ensemble decision for unlabeled instances is given by weighted majority vote between the  $k$  classifiers with the highest weights. This allows AWE to revert back on former concepts, if they were not already removed from the ensemble. As an extension of AWE, the AUE (Brzezinski and Stefanowski 2011) allows classifiers to be updated or deleted based on the most recent data.

Similar to both AWE and AUE, DWM (Kolter and Maloof 2007) maintains an accuracy weighted ensemble of classifiers, but with a flexible ensemble size. New classifiers are only added if the ensembles' decision on the most recent data is erroneous. In addition, members only get removed if their weight falls below a user defined threshold.

In order to cope with recurrent concept drift, the *Learn<sup>++</sup>*.NSE algorithm (Elwell and Polikar 2011) adds one classifier for each new batch, and stores all of them as well as their performance on all batches after creation. A sigmoidal weight function is applied to the resulting vector of evaluation results to compute a classifiers' weight on the most recent environment.

Highly related to our approach is FASE (Frías-Blanco et al. 2016), a fixed size ensemble of adaptive classifiers trained using online bagging. Each adaptive classifier is paired with an active drift detector. If a warning gets signaled, a second instance of the classifier starts training in parallel and the predictions of both are combined by weighted voting based on the error-rate. Once a drift follows the warning, the old classifier is deleted, otherwise the new one. In order to combine the predictions of all adaptive classifiers, an additional classifier is trained that receives their predictions as input and outputs the final ensemble decision. In Goncalves Jr and Barros (2013), the authors propose a classification method for recurrent drift scenarios by explicitly remembering some instances of the concept, in order to be able to match classifiers to certain concepts.

The *Fast Adapting Ensemble* (FAE) algorithm (Ortíz Díaz et al. 2015) is a multiclass ensemble algorithm which employs batchwise operation to handle recurring concepts. It keeps several classifiers in the ensemble and adds new ones based on an explicit drift detection. FAE conducts a weighted majority vote, and dynamically adjusts these weights.

## 6.4 Beta Distribution Drift Detection

In Section 3.3 we introduced related work in the domain of drift detection. In drift or concept change detection, the goal is to identify a significant change in the underlying concept, such that a classifier can employ a mechanism that adapts to the new concept. Many adaptive learning algorithms require an active drift detection mechanism. Usually, the drift is detected via observing the classification performance of the existing model. When the performance deteriorates significantly, the adaptation process is triggered.

In *Patching*, concept drift is detected implicitly by the error estimator. When the concept changes, the estimator picks up that change in its training process, and thus trigger the patch to learn the new concept. For *Ensemble Patching*, we want to create an ensemble of patches, where each member is responsible for drift on certain classes. In order to estimate drift for every class, we employ an explicit drift detection, allowing us to detect and quantify the amount of drift.

Drift detection methods such as DDM (Gama, Medas, et al. 2004) or EDDM (Baena-Garcia et al. 2006) rely on statistical process control in order to observe a significant change in a stream of bernoulli trials, i. e., the errors of the underlying classifier. Based on the *Probably Approximately Correct* (PAC) learning model (Valiant 1984), we assume that the error rate of the learning algorithm decreases over time, if the concept remains stable and the number of examples increases. A change in the underlying distribution of the process generating that stream implies concept drift. Both of these methods are constructed towards detecting drift in such a sequence.

DDM detects a change in the sequence via a binomial distribution. Therefore, it keeps a memory of recent observations, and decides based on a threshold, whether a concept drift occurred in the memory window. More specifically, it employs two thresholds, one for detecting a *warning*, and one for detecting the actual *drift*. The *warning* is used to trigger coping mechanisms for adaptation. DDM is good in detecting abrupt changes and quick gradual changes. However, it struggles in detecting slow gradual changes: when the example memory is shorter than the drift duration, concept drift may go unnoticed. EDDM mitigates this issue by observing the average distance between errors instead. It also employs two thresholds rule for detecting *drift*, and gives out a *warning* beforehand.

In our target scenario, we are dealing with batches of bernoulli observations. We want to leverage the batch-character of these observations for a better-suited drift detector.

Therefore, we introduce *Beta Distribution Drift Detection* (BD<sup>3</sup>), a method that leverages previously known information about an existing classifier in the form of a beta distribution, and detects drift by assessing if new batches of data operate within the confidence bounds of that distribution. If previous knowledge does not exist, we gather it in the beginning of the drift detection process. The method works on batches of data, as opposed to other methods that work on single instances at a time. The batchwise approach is generally more stable w. r. t. the trade-off between false alerts and false negatives, and, as we think, applies to more real-world scenarios. It has been published at the European Symposium on Neural Networks (ESANN) in 2019 (Fleckenstein, Kauschke, and Fürnkranz 2019).

In Section 6.4.1 we define the problem, followed by the explanation of our algorithm in Section 6.4.2. We describe the experimental setup in Section 6.4.3, discuss our results in Section 6.4.4, and draw some conclusions in Section 6.4.5.

### 6.4.1 Problem Definition

We assume a model  $M$  which classifies a stream of data instances  $D$ . The stream consists of batches  $D_i$ ,  $i \in 0 \dots I$ , where the number of batches  $I$  is large or potentially infinite. For each batch  $D_i$ , the model  $M$  classifies the  $n_i$  instances, producing a corresponding error batch  $E_i$ . Each error batch  $E_i$  consists of binary classifier errors  $e_{i,j}$ , where  $e_{i,j} = 0$ , if  $M$  predicts instance  $j \in 0 \dots n_i$  correctly, and  $e_{i,j} = 1$  otherwise. The goal is to detect whether a batch  $E_i$  shows a significant increase in error rate compared to previous batches. We approach this by fitting beta distributions to the classifier error.

### 6.4.2 Beta Distribution Drift Detection Method

Our drift detection method is based on evaluating the beta distribution of the classification error of a model. For each new batch of data, the

current classifier error is compared against this distribution. If it is outside a certain confidence interval, concept drift is assumed.

### Model Initialization.

The beta distribution has two shape parameters  $\alpha, \beta > 0$ , which we initially derive from previous knowledge about the error-rate  $\pi_0$  of the model. As a rule of thumb we use  $\alpha_0 = \pi_0 \cdot n_0$  and  $\beta_0 = (1 - \pi_0) \cdot n_0$ , where  $n_0$  is the number of instances in the first batch that we receive. If  $\pi_0$  is unknown, we suggest to set it to 0.5 as a starting value.

### Batchwise Model Update.

Given the most recent batch  $D_i$ , the detector receives the corresponding binary error batch  $E_i$ . For a given error rate  $\pi_i$  of the model on this batch, the likelihood for observing  $k_i$  misclassifications on the  $n_i$  samples is given by the binomial distribution  $Bin(k_i|n_i, \pi_i)$ . By putting a prior on the error rate, we are able to compute the posterior probability of the error rate given some data. Since it is a conjugate prior to the binomial, we choose a beta distribution  $Beta(\pi_i|\alpha_i, \beta_i)$ , where  $\alpha_i$  and  $\beta_i$  represent the number of previously misclassified and correctly classified samples, respectively.

Following Bayes Rule, the posterior probability is given as:

$$\Pr(\pi_i|D_i) = \frac{\Pr(D_i|\pi_i) \Pr(\pi_i)}{\Pr(D_i)} \quad (6.1)$$

$\Pr(D_i|\pi_i) = Bin(k|n, \pi_i)$  = probability for observing  $k$  misclassifications out of  $n$  samples of the class.

$\Pr(\pi_i) = Beta(\pi_i|\alpha, \beta)$ , where  $\alpha$  = number of misclassifications,  $\beta$  = number of correct classifications. It can be shown that:

$$\begin{aligned} \Pr(\pi_i|E_i) &= \frac{\Pr(E_i|\pi_i) \cdot \Pr(\pi_i)}{\Pr(E_i)} \\ &= \frac{Bin(k_i|n_i, \pi_i) \cdot Beta(\pi_i|\alpha_i, \beta_i)}{\Pr(E_i)} \\ &= Beta(\alpha_i^*, \beta_i^*) \end{aligned} \quad (6.2)$$

with  $\alpha_i^* = \alpha_i + k_i$  and  $\beta_i^* = \beta_i + (n_i - k_i)$ .

### Drift Detection.

We can now test if the error of a new batch of data is likely to correspond to the classifier's concept, or if a concept change occurred:

1. Compute *warning* and *drift* boundaries that contain 95.0% and 99.7% of the distribution  $Beta(\alpha_{i-1}^*, \beta_{i-1}^*)$ , and the current error rate  $\pi_i = k_i/n_i$ .

If  $\pi_i > upper\_bound_{warning}$  signal warning.

If  $\pi_i > upper\_bound_{drift}$  signal drift and reset shape parameters to  $\alpha_{i-1}^* = \alpha_0$ ,  $\beta_{i-1}^* = \beta_0$ , since the observed error does not

correspond to the previous distribution.  
Reset test counter  $t = 0$ .

2. Set the parameters of the previous posterior distribution  $Beta(\alpha_{i-1}^*, \beta_{i-1}^*)$  as prior for the most recent one,  $\alpha_i = \alpha_{i-1}^*$ ,  $\beta_i = \beta_{i-1}^*$ .
3. Compute the most recent shape parameters
 
$$\alpha_i^* = \frac{\alpha_i}{decay} + k_i,$$

$$\beta_i^* = \frac{\beta_i}{decay} + (n_i - k_i).$$
 Increment test counter  $t = t + 1$ .

The parameter *decay* is used to control the influence of prior knowledge given by the previous batches. As the reliability of that prior increases with the number of tests  $t$ , we decrease *decay* according to  $decay = \frac{1}{\exp(a \cdot (t+b))} + 1.1$ .

We choose  $a = 0.15$ ,  $b = -7$  based on preliminary experiments as a trade-off between abrupt and gradual drift detection. It can be shown that  $\lim_{i \rightarrow \infty} \alpha_i^* + \beta_i^* = n + \frac{n}{decay-1} \quad \forall decay > 1$ . Thus, the decay parameter limits the variance of the beta distribution, preventing it from becoming too narrow, which would increase false alerts. A formal proof of this convergence is given in Appendix A.3. Figure 6.1 shows how different shape parameters  $\alpha$  and  $\beta$  influence the beta distribution, even if their mean  $\pi = \alpha / (\alpha + \beta)$  remains the same.

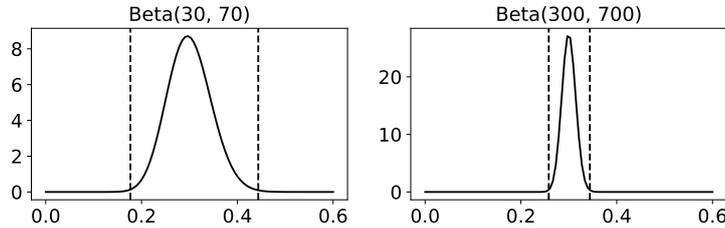


Figure 6.1: Two example *Beta* probability density functions with their boundaries that contain 99.7% of the distribution.

### 6.4.3 Experiment Setup

In this section, we describe the datasets we choose for evaluation, the evaluation procedure, the used metrics, and the algorithms we compare against. We compare our novel approach against two tried-and-tested approaches which follow the idea of detecting drift from classifier error, DDM and EDDM<sup>2</sup> as introduced in Section 3.3.

#### Evaluation Datasets

We evaluate our findings on four datasets. Each scenario represents a different type of concept drift with varying severity and/or gradation:

<sup>2</sup> Implemented in <https://scikit-multiflow.github.io>

- *Bit-Stream*: The first dataset is a stream of bits from a Bernoulli distribution with parameter  $\mu$  as proposed in (Frias-Blanco et al. 2016). Each stream contains 30 CPs, separated by 600 or 2,000 bits for which the concept is stable. In order to simulate different drift magnitudes, the maximum absolute difference between the means of two subsequent concepts is restricted in an interval  $[a, b]$ .
- *SEA Concepts*: SEA is a drifting stream generation scheme (Street and Kim 2001) and used as a standard test for abrupt concept change. The dataset has three features, two determine the class and the third is noise. We generate streams of 40k instances with three CPs at 10k, 15k, and 30k samples, and also vary between 10% and 20% class noise.
- *Rotating Hyperplane*: This dataset has features equal to SEA, but contains a gradual drift behavior. Class labels depend on the placement of the two-dimensional points compared to a hyperplane that rotates during the course of the stream. It starts rotation with a certain angle every 1,000 instances, starting after the first 10k samples, the angles being 20°, 30°, and 40°.
- *Elec2*: This dataset (Harries 1999) is a real-world dataset of electricity prices. It contains 45,312 instances with eight features and binary class labels that indicate price change (up or down), and has an unknown number of drifts.

## Evaluation Measures

For the comparison of the algorithms we use the following metrics:

- **False Positive Rate (FPR)**: The false positive rate, where the drift detection method detects a drift when there is actually none.
- **False Negative Rate (FNR)**: The false negative rate, where the model fails to detect a drift, when there is one.
- **Delay**: The average number of batches between the true drift point and the first true positive detection on the same concept.

We run each algorithm 50 times with a batch size of 200 on every dataset and calculate the average values and standard deviations of FPR, FNR, and Delay for those runs. For the synthetic and real-world datasets we use a *Naïve Bayes* (NB) classifier in the *Interleaved Test-Then-Train* or *Prequential* framework, in which a new batch is first used for evaluating the accuracy and afterwards for updating the classifier (cf. Gama and Kosina (2014) for more details). If a warning gets signaled, a second instance of the classifier starts training in parallel and replaces the primary one once a drift follows the warning, as proposed in (Gama, Medas, et al. 2004).

Table 6.1: Results on the Bit-Stream dataset (Standard deviation in brackets).

Method	600 bits between changes			1,000 bits between changes			
	[0.1, 0.3]	[0.3, 0.5]	[0.5, 0.7]	[0.1, 0.3]	[0.3, 0.5]	[0.5, 0.7]	
DDM	FPR	<b>0.0310</b> (±0.0232)	<b>0.0215</b> (±0.0133)	<b>0.0197</b> (±0.0127)	<b>0.0106</b> (±0.0071)	<b>0.0062</b> (±0.0050)	<b>0.0085</b> (±0.0055)
	FNR	0.5579 (±0.1996)	0.2254 (±0.1975)	0.0115 (±0.0523)	0.5011 (±0.2180)	0.2180 (±0.2783)	0.0114 (±0.0800)
	Delay	0.6122 (±0.3011)	0.3325 (±0.1367)	0.0273 (±0.0468)	2.3346 (±0.9044)	1.4450 (±0.3424)	0.4567 (±0.1183)
EDDM	FPR	0.1719 (±0.0832)	0.1831 (±0.0588)	0.2197 (±0.0461)	0.0861 (±0.0458)	0.1018 (±0.0359)	0.1099 (±0.0299)
	FNR	0.4072 (±0.1942)	0.0923 (±0.1138)	<b>0.0000</b> (±0.0000)	0.4656 (±0.1731)	0.0950 (±0.1438)	<b>0.0000</b> (±0.0000)
	Delay	0.3508 (±0.2182)	0.4337 (±0.1317)	0.0530 (±0.0603)	2.0603 (±0.8537)	2.5416 (±0.4280)	1.3600 (±0.2792)
$BD^3$	FPR	0.0521 (±0.0377)	0.0944 (±0.0402)	0.0618 (±0.0406)	0.0552 (±0.0335)	0.0802 (±0.0422)	0.0383 (±0.0286)
	FNR	<b>0.0312</b> (±0.0414)	<b>0.0000</b> (±0.0000)	<b>0.0000</b> (±0.0000)	<b>0.0270</b> (±0.0421)	<b>0.0000</b> (±0.0000)	<b>0.0000</b> (±0.0000)
	Delay	<b>0.0189</b> (±0.0383)	<b>0.0000</b> (±0.0000)	<b>0.0000</b> (±0.0000)	<b>0.0637</b> (±0.1246)	<b>0.0000</b> (±0.0000)	<b>0.0000</b> (±0.0000)

#### 6.4.4 Results

In Table 6.1, we show the results on the abruptly drifting bit-stream. Compared to DDM and EDDM,  $BD^3$  generally shows low FNR and Delay values. Especially for the smallest change interval of [0.1, 0.3], where DDM and EDDM show high levels of FNR,  $BD^3$  performs better. On the synthetic datasets (Table 6.2),  $BD^3$  shows comparable results to DDM and EDDM. The performance depends mainly on the classifier algorithm, and the chosen drift detector has limited effect. However, the accuracy is marginally above DDM and EDDM, which we attribute to the lower delay achieved by  $BD^3$ . Figure 6.2 shows the accuracy as a stream on the gradually drifting rotating hyperplane dataset, where

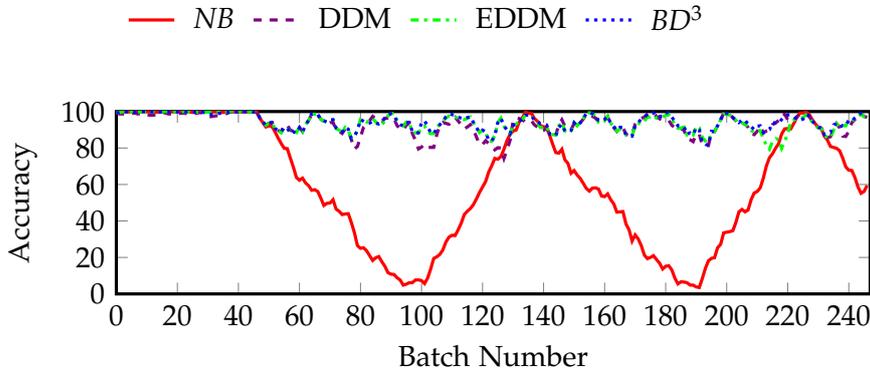


Figure 6.2: Accuracy comparison on the Rotating Hyperplane dataset (20°).

Table 6.2: Results on synthetic datasets (Accuracy).

Method	SEA Concepts (Abrupt)			Rotating Hyperplane (Gradual)		
	0.0	0.1	0.2	20°	30°	40°
No Detector	0.9137 (±0.0014)	0.8487 (±0.0015)	0.7631 (±0.0015)	0.6046 (±0.0079)	0.5933 (±0.0106)	0.5834 (±0.0080)
DDM	0.9417 (±0.0111)	0.8643 (±0.0102)	0.7671 (±0.0100)	0.9221 (±0.0073)	0.9027 (±0.0131)	0.8925 (±0.0110)
EDDM	0.9471 (±0.0023)	0.8621 (±0.0036)	0.7617 (±0.0038)	0.9283 (±0.0064)	0.9109 (±0.0095)	0.9039 (±0.0068)
$BD^3$	<b>0.9496</b> (±0.0024)	<b>0.8717</b> (±0.0039)	<b>0.7750</b> (±0.0057)	<b>0.9315</b> (±0.0061)	<b>0.9133</b> (±0.0085)	<b>0.9053</b> (±0.0058)

Table 6.3: Results on the Elec2 dataset.

	No Detector	DDM	EDDM	$BD^3$
Accuracy	0.7270	0.7232	0.7243	<b>0.7307</b>

$BD^3$ 's advantage in delay shows by faster recovery on dips in the curve. On the Elec2 dataset (Table 6.3),  $BD^3$  is the only drift detector that does not lower the accuracy below the levels of no detector.

## 6.4.5 Conclusion

In this section, we have shown a novel drift detection method that monitors the classifier error via a beta distribution. Change in the classifier's performance is detected as drift, the sensitivity of the detector can be adjusted via a confidence threshold and decay parameters. Existing knowledge about the classifier's performance can be used to set the initial parameters of the distribution, which allows immediate drift detection. Experimental results show that the method is robust against false positives, while also being fast in detecting concept drift.

## 6.5 Ensembles of Patches

The method we propose named *Ensemble Patching* detects concept drift on a class-wise basis, i. e. it monitors the performance of the existing classifier w. r. t. classifying instances from all available classes. If a change is detected in the classifiers' performance, a drift adaptation mechanism is executed. This adaptation mechanism is loosely based on *Patching* described in Chapter 4, which is founded on the idea of adding coping mechanisms (patches) to parts of the instance space, where the base classifier misclassifies instances.

*Ensemble Patching* also adapts to changes by training patches, but in this case, patches cover a certain error pattern on one or multiple classes, instead of regions in the instance space. Patches are neural networks trained on the most recent data (batch), that handle the classification of instances from the drifting concept. A separate en-

semble decision function (the conductor) combines the decisions of the base classifier and the ensemble of patches. This meta-decider is implemented as a recurrent neural network that classifies a sequence of decisions from the base classifier and the patches combined with a representation of the instance itself. The sequential character allows adding unlimited amounts of patches to the ensemble.

### 6.5.1 Components of Ensemble Patching

*Ensemble Patching* consists of five essential components:

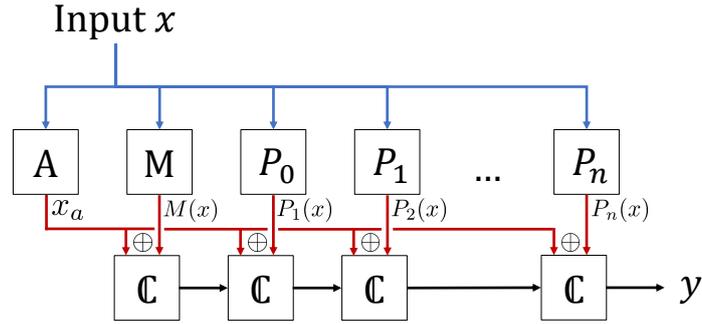
1. The base classifier  $M$ . The existing classifier that classifies the stream, which we want to adapt for concept drift.
2. The instance encoder  $A$ . We use the encoding part of an autoencoder to reduce the dimensionality of the instances as a preprocessing step. The autoencoder is trained on the initial batches of data, when no drift has occurred yet.
3. The drift detector  $\mathbb{D}$ . We realize our class-wise drift detection by applying *Beta Distribution Drift Detection* (Fleckenstein, Kauschke, and Fürnkranz 2019) on a per-class basis. This means the drift is detected for each class separately, based on the prediction of  $M$ . Varying performance of  $M$  w. r. t. instances of a class will be identified as drift.
4. The patches  $P_j, j \in 0..n$ , a potentially infinite array of classifiers that expand the ensemble beyond  $M$ . Each patch monitors the class-wise error rate of itself via *Beta Distribution Drift Detection*.
5. The conductor  $\mathbb{C}$ . A recurrent neural network that takes the encoded instance and the classification decisions of  $M$  as well as all patches  $P_j$  and conducts the ensemble decision.

In the following we will describe these components and elaborate on them.

### 6.5.2 Classification

When classifying a new instance  $\mathbf{x}$ , the following steps (Figure 6.3) are executed:

1.  $\mathbf{x}$  is encoded via  $A$ :  $\mathbf{x}_A = A(\mathbf{x})$ .
2.  $\mathbf{x}$  is classified by  $M$  and all patches  $P_j$ .
3. The encoding and classification results are concatenated and put into a sequence  $S$ .
4. The conductor  $\mathbb{C}$  classifies the resulting sequence  $S$ .

Figure 6.3: Sequential prediction of the ensemble conductor  $\mathbb{C}$ .

### 6.5.3 Adaptation

The adaptation of *Ensemble Patching* works in two phases, *Initialization* and *Continuous Drift Handling*. The whole process is depicted in Appendix A.2. Once the *Initialization* phase is over, *Ensemble Patching* always remains in the *Continuous Drift Handling* phase.

1. **Initialization.** The *Initialization* phase is used to retrieve basic performance estimates of the given classifier (via counting correct- and misclassifications) in the form of a beta-distribution, as well as to build an autoencoder on the instances for dimensionality reduction. We assume that the original concept remains stable for at least two batches, so that we can get an estimate of the classifier error on unaltered data.

The training of the autoencoder is described in Section 6.5.4. After each batch of data and its respective true labels are received, the performance estimate of  $M$  is updated. Then,  $\text{BD}^3$  is applied. After  $\text{BD}^3$  is sufficiently primed, the *Initialization* phase continues until the first concept drift is detected. This starts the *Continuous Drift Handling* phase. While being in the *Initialization* phase, the retrieved instances are used to train an autoencoder  $A$ . This autoencoder is later used to encode instances into a lower-dimensional representation, which is beneficial to the overall ensemble decision.

2. **Continuous Drift Handling.** In this phase, adaptation mechanisms are activated to compensate concept drift. When the drift detector signals concept drift in a subset of classes, it checks the list of patches for a patch that has been trained on one or multiple of these classes. If a suitable patch is found, we determine if the error-rate of instances from that class on the current batch matches the error-distribution of the patch. If so, the patch will be updated with all the instances from that class. This is repeated for all suitable patches. If no patch covers the drifting class, a new patch is created. Finally, the ensemble conductor  $\mathbb{C}$  is updated with the newest batch.

### 6.5.4 Dimensionality Reduction

Our method uses the encoding step of an autoencoder to transform all instances into a lower-dimensional representation as proposed by Geoffrey E Hinton and Salakhutdinov (2006). We discovered, that the ensemble conductor works significantly better when it can access the properties of the instance that it wants to classify. However, putting the—potentially high dimensional—instance directly into the RNN of the conductor bloats the resulting neural network, making it both hard to train and unnecessarily complex. If—as in some datasets we used in the evaluation—the dimensionality is already low, we do not apply this transformation. In preliminary experiments we found out that a dimensionality-reduced version is as or even more helpful. Figure 6.4 shows this: an observable advantage can be obtained by using the encoded features together with the classification sequence.

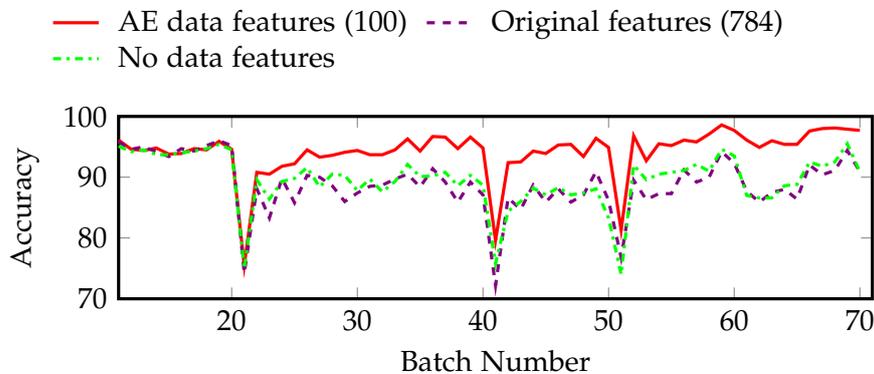


Figure 6.4: Influence of the data features given to the conductor.

The autoencoder we use consists of a single hidden layer with 100 nodes and linear activations. Other activation functions such as rectified linear are not beneficial to our cause, but as this is a highly individual setting, the autoencoder has to be specifically geared towards each dataset.

### 6.5.5 New Patch Creation

If a drifting class or a combination of drifting classes have been detected on the latest batch of data, a new patch is trained. A patch consists of a small neural network with a single ReLU-activated hidden layer with 100 nodes, followed by a dropout layer with dropout rate 0.5 and a Softmax classification layer. The most current batch is split into a training and a validation set. The neural network is trained on the training set. The evaluation set is used to determine the performance and create the beta-distribution for the new patch. This is later used to determine whether a different batch shows similar behaviour, and can be used to further enhance a patch.

## 6.5.6 Existing Patch Update

If drift has been detected, the algorithm first determines whether an existing patch or a combination of several ones can cover the drifting data. Therefore, the patches are evaluated and errors get compared to the patches' beta-distributions, which model their usual error rate for each class they have been trained on. If the recent error is likely under the given distributions, the new data is assumed to belong to the already known concept. The data batch is once again split into a training and evaluation set, and the neural network is then trained with the newest data. The beta-distribution of the patch performance is updated via the results on the evaluation part.

## 6.5.7 Ensemble Conductor

The conductor function  $\mathbb{C}$  operates on the sequence of classifier decisions. The sequence consists of items combining the encoded instance and the classifier decision:

$$S = (x_A \oplus M(x)); (x_A \oplus P_1(x)); \dots; (x_A \oplus P_n(x)). \quad (6.3)$$

The conductor is a GRU network with 100 units and a fully connected Softmax classification layer. It has the advantage that an unlimited number of patches can be added to the ensemble. When new data for learning has been gathered or new patches have been added, the conductor can be incrementally updated to encompass the latest knowledge as well as necessary adaptations towards the current scenario. We chose a GRU network as those have been shown to be computational more efficient and better suited for applications with less data than other current RNN architectures (Chung et al. 2014).

Although the patches make their (informed) decisions, the conductor has to determine the final decision in order to yield the optimal results based on the current situation.

# 6.6 Experiment Setup

We evaluate our methodology on variations of three datasets, each containing multiple drifts. At some point the drift reverts the concept to an already seen scenario, in order to demonstrate recurring concepts.

## 6.6.1 The MNIST Dataset

Once more we use the *MNIST* dataset of handwritten digits. It contains the pixel data of 70,000 digits (28x28 pixel), which we treat as a stream. The base classifier  $M$  is always trained on the first 10,000 instances, the drifts follow at later points. By the third drift, the concept is reverted to a previous concept to simulate concept re-occurrence. We cover the following scenarios:

- $MNIST_{switch}$ : The class labels switch during the course of the stream. At #20k: switch 0 – 1, and 2 – 4. At #40k: switch 6 – 8, and 7 – 9. At #50k: 0 – 1, and 2 – 4 is reverted to original.
- $MNIST_{rotate}$ : The digits 2,5, and 7 are rotated  $45^\circ$  from instance #20k onwards. At #40k: Rotate 2, 5, 7 to  $90^\circ$ . At #50k: rotate 2, 5, 7 back to  $45^\circ$ .
- $MNIST_{appear}$ : In the beginning, classes 3, 5, 7, 9 are missing. At #20k: 3, 5 appear. At #30k: 7, 9 appear; 3, 5 disappear. At #40k: 3, 5 re-appear.

## 6.6.2 Rotating Hyperplane

Rotating hyperplane is a dataset in a  $d$ -dimensional space as proposed in (Hulten, Spencer, and Domingos 2001), with which a binary stream classification problem can be constructed. We generate a stream  $RotHyp$  of 80,000 instances with three numeric attributes and introduce a slow rotation of the hyperplane. We simulate a continuous gradual shift in the problem space in four variants  $RotHyp_n$ , where the concept rotates completely ( $360^\circ$ ) for 1,2,4, and 5 times respectively. This introduces variable drift speeds, which we use to investigate the algorithms' capability of handling these types of drift. We also introduce 5% class noise.

## 6.6.3 The SEA concepts

The SEA concepts dataset is a dataset with abrupt concept drift (Street and Kim 2001) consisting of three attributes, where the attributes are used to generate the resulting binary label. All three attributes are real-valued and have random values between 0 and 10. The class label is decided based on a linear combination of the attributes and a threshold  $\theta$ . We generate a stream, where the threshold value  $\theta$  will be changed during the course of the instance stream to introduce concept (values 10, 7.5, 12, and 8). The stream consists of 70,000 instances with three CPs at #20k, #40k, and #50k. Furthermore, 10% class noise is added.

## 6.6.4 Evaluation Measures

We evaluated scenarios with multiple drift points. For the comparison of the algorithms we use the known metrics Final Accuracy, Recovery Speed, Adaptation Rank and Final Rank.

We ran each algorithm 10 times on every dataset with different random seeds and averaged over the results for the 10 runs. The standard deviation of accuracy in those 10 runs was smaller than 2%. Since the  $RotHyp$  dataset is continuously drifting, the metrics Final Accuracy and Final Rank cannot be calculated.

## 6.6.5 Benchmark Algorithms

We compare our approach against methods that are either technically similar or are intended to handle similar scenarios.

**No Adaptation:** A random forest with 100 trees, trained on the initial 10,000 instances. This is the non-adaptive base classifier  $M$ .

**$MLP_{Baseline}$ :** Multilayer-perceptron with 200 hidden nodes trained and updated on the whole batch-data every time the patching ensemble gets updated. Re-initialized if the ensemble needs a new patch (indicating a yet unseen concept). This MLP is used to show if the decision can be made on the instances alone, without the need for specialized patches.

**Patching:** Random forest with 100 random trees for both error-region classification and the patch, as stated in Section 4.5. Keeps one batch for handling concept drift.

**AUE:** Up to 30 classifiers in the ensemble, selects the best 15 as active. Each classifier is a MLP with 100 hidden nodes.

**$Learn^{++}.NSE$ :** We use a base classifier with 100 random trees, and sigmoid shape parameters  $a=0.5$ ,  $b=10$ , as stated in the original paper (Elwell and Polikar 2011).

## 6.6.6 Evaluation Procedure

We use the prequential evaluation method as implemented in scikit multiflow (Montiel et al. 2018) with a batch size of 1,000 instances per batch. Our base model  $M$  is trained on the first 10,000 instances which contain no concept drift.  $MLP_{Baseline}$  and *Ensemble Patching* require an initialization phase with a stable concept. We assume the first two batches of each stream to be conceptionally stable. The  $BD^3$  drift detectors are parametrized equal to the original paper (Fleckenstein, Kauschke, and Fürnkranz 2019) for drift detection on  $M$ . For drift detection on individual patches,  $a=0$ ,  $b=0$  are selected. This yields a constant decay and is able to allow infinite updates on the beta-distribution.

## 6.7 Results

Table 6.4 shows the experimental results. On the *MNIST*-based datasets, *Ensemble Patching* can especially show its advantage w. r. t. Recovery Speed, and hence the Adaptation Rank. While *Patching* cannot reach a sufficient final accuracy, because it is limited to only the newest batch for learning. The overall accuracy of *Ensemble Patching* is comparable to  $MLP_{Baseline}$  and *AUE*.  $Learn^{++}.NSE$ , although conceptually similar, falls behind in all the rankings. The adaptation advantage shows in the accuracy plots of Figure 6.5: When previous concepts return (the third line of drift), the adaptation is very fast and the return to previous levels is almost instantaneous. A special feature of *Ensemble Patching* is the ability to handle mixtures of previously learned concepts. This can be seen on the third drift in the  $MNIST_{appear}$  results in figure 6.5c,

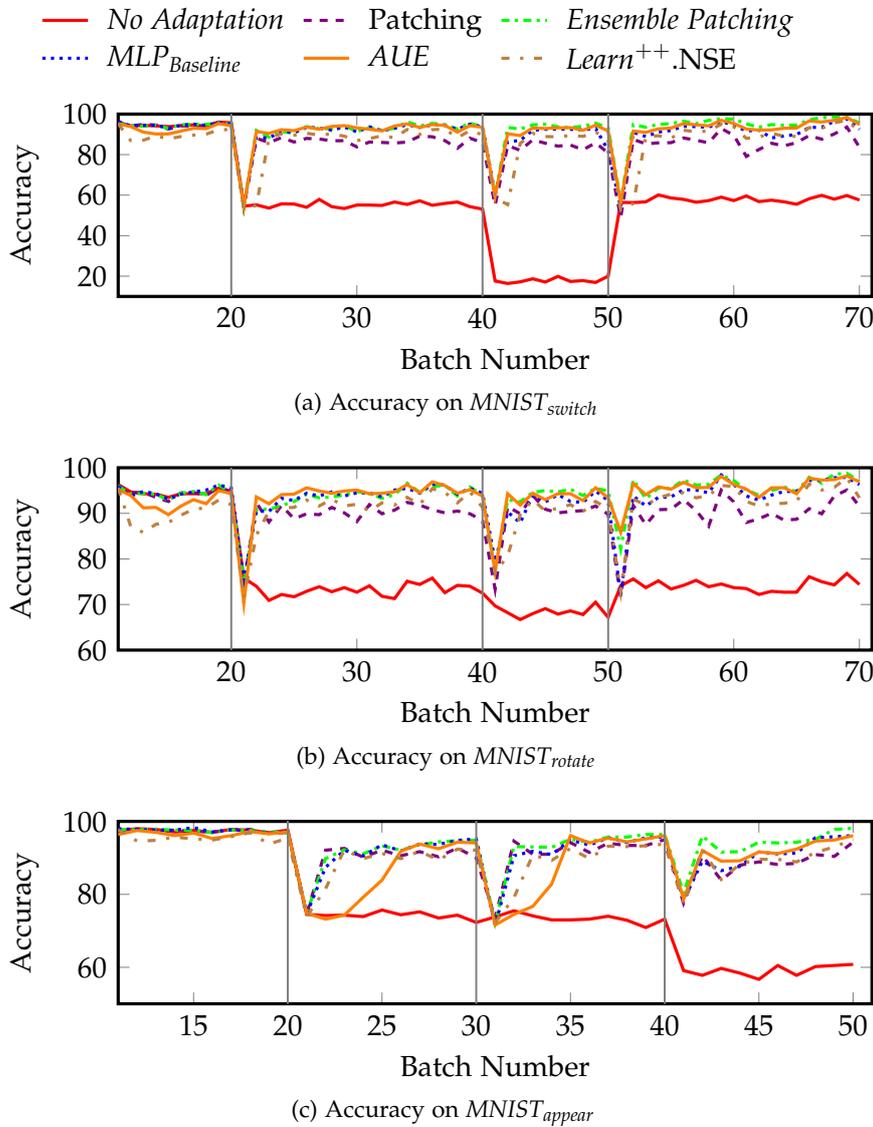


Figure 6.5: Accuracy on abrupt drift with reoccurring concepts.

Table 6.4: Experimental results for *MNIST* – Overview

Classifier	F.Acc	R.Spд	Ad.Rk	F.Rk
<i>MNIST<sub>switch</sub></i>				
No Adaptation	0.4381	—	5.8471	6.0000
<i>Patching</i>	0.8600	<b>1000</b>	4.8818	5.0000
<i>Ensemble Patching</i>	<b>0.9543</b>	<b>1000</b>	<b>1.3448</b>	<b>1.1333</b>
<i>MLP<sub>Baseline</sub></i>	0.9383	2000	2.6151	2.4778
<i>AUE</i>	0.9421	1333	2.0874	2.3333
<i>Learn<sup>++</sup>.NSE</i>	0.9126	2966	4.1339	3.9778
<i>MNIST<sub>rotate</sub></i>				
No Adaptation	0.7248	—	5.8725	6.0000
<i>Patching</i>	0.9106	<b>1000</b>	4.4896	4.9556
<i>Ensemble Patching</i>	<b>0.9586</b>	<b>1000</b>	1.7389	<b>1.3667</b>
<i>MLP<sub>Baseline</sub></i>	0.9494	1066	2.9776	2.5778
<i>AUE</i>	0.9541	<b>1000</b>	<b>1.6916</b>	1.9667
<i>Learn<sup>++</sup>.NSE</i>	0.9331	1700	4.0896	3.9667
<i>MNIST<sub>appear</sub></i>				
No Adaptation	0.6876	—	5.5095	6.0000
<i>Patching</i>	0.9216	<b>1000</b>	3.4905	4.6111
<i>Ensemble Patching</i>	<b>0.9581</b>	1066	<b>1.5333</b>	<b>1.3667</b>
<i>MLP<sub>Baseline</sub></i>	0.9494	2500	2.7095	2.1222
<i>AUE</i>	0.9475	3233	3.3238	2.4778
<i>Learn<sup>++</sup>.NSE</i>	0.9273	1666	4.1810	4.3111

where the overall concept now contains all 10 digits, a previously unseen concept. Here, *Ensemble Patching* can profit from previously trained patches that separately learned the appearing numbers.

For the *SEA*-dataset, we see that most classifiers achieve very good results, given that 10% class noise was inserted into the dataset. *Ensemble Patching* is near the optimum with 89% accuracy, and also adapts the fastest of all compared methods. Figure 6.7 shows this in more detail: *Ensemble Patching* recovers after one batch (1,000 instances), which is not always the case for *AUE* or *Learn<sup>++</sup>.NSE*. The original *Patching* suffers once again from being limited to only the last batch for training.

On the *RotHyp<sub>n</sub>*-datasets all classifiers show similar behavior on the gradually drifting concept. The slower the drift (*RotHyp<sub>1</sub>* in Figure 6.6a), the better the overall performance and adaptation capabilities. However, for the drifting scenario *RotHyp<sub>5</sub>* (Figure 6.6b) with a higher drift frequency, *MLP<sub>Baseline</sub>* and *Learn<sup>++</sup>.NSE* struggle in terms of consistency, while *Patching*, *AUE*, and *Ensemble Patching* remain stable. Regarding the accuracy (Table 6.5), it shows that all ensemble methods are practically equal. With these datasets we wanted to show, that—although *Ensemble Patching* has a mechanism that explic-

itly detects concept drift, which can be problematic for slow gradual drift situations—it detects gradual drift as well as state-of-the-art algorithms.

Table 6.5: Experimental results for *RotHyp*– Overview

Classifier	F.Acc	Ad.Rk
<b><i>RotHyp</i><sub>1</sub></b>		
No Adaptation	0.5739	4.9783
<i>Patching</i>	0.9086	3.5467
<i>Ensemble Patching</i>	<b>0.9170</b>	<b>1.8933</b>
<i>MLP</i> <sub>Baseline</sub>	0.9111	2.9367
<i>AUE</i>	0.9129	3.4067
<i>Learn</i> <sup>++</sup> .NSE	0.9141	2.7650
<b><i>RotHyp</i><sub>2</sub></b>		
No Adaptation	0.5748	4.7683
<i>Patching</i>	0.8822	2.9717
<i>Ensemble Patching</i>	<b>0.8873</b>	<b>2.1017</b>
<i>MLP</i> <sub>Baseline</sub>	0.8760	3.3517
<i>AUE</i>	0.8825	3.4883
<i>Learn</i> <sup>++</sup> .NSE	0.8793	2.8317
<b><i>RotHyp</i><sub>4</sub></b>		
No Adaptation	0.5752	4.3717
<i>Patching</i>	0.8266	2.5800
<i>Ensemble Patching</i>	<b>0.8286</b>	<b>2.0767</b>
<i>MLP</i> <sub>Baseline</sub>	0.7903	3.7717
<i>AUE</i>	0.8194	3.7250
<i>Learn</i> <sup>++</sup> .NSE	0.8103	3.1783
<b><i>RotHyp</i><sub>5</sub></b>		
No Adaptation	0.5749	4.3100
<i>Patching</i>	0.7972	2.6067
<i>Ensemble Patching</i>	<b>0.7994</b>	<b>2.0933</b>
<i>MLP</i> <sub>Baseline</sub>	0.7436	3.9667
<i>AUE</i>	0.7871	3.6533
<i>Learn</i> <sup>++</sup> .NSE	0.7808	3.1267

### 6.7.1 Significance Testing

We conduct the Friedman test combined with the Wilcoxon signed-rank test on the average adaptation ranks, since the *RotHyp*-datasets have no final rank. The null hypothesis is rejected for a significance level of 0.05. Furthermore, we applied the Wilcoxon signed-rank test to establish pairwise comparisons. The results are shown in Figure 6.8. The algorithms are ordered by average rank. A connection between

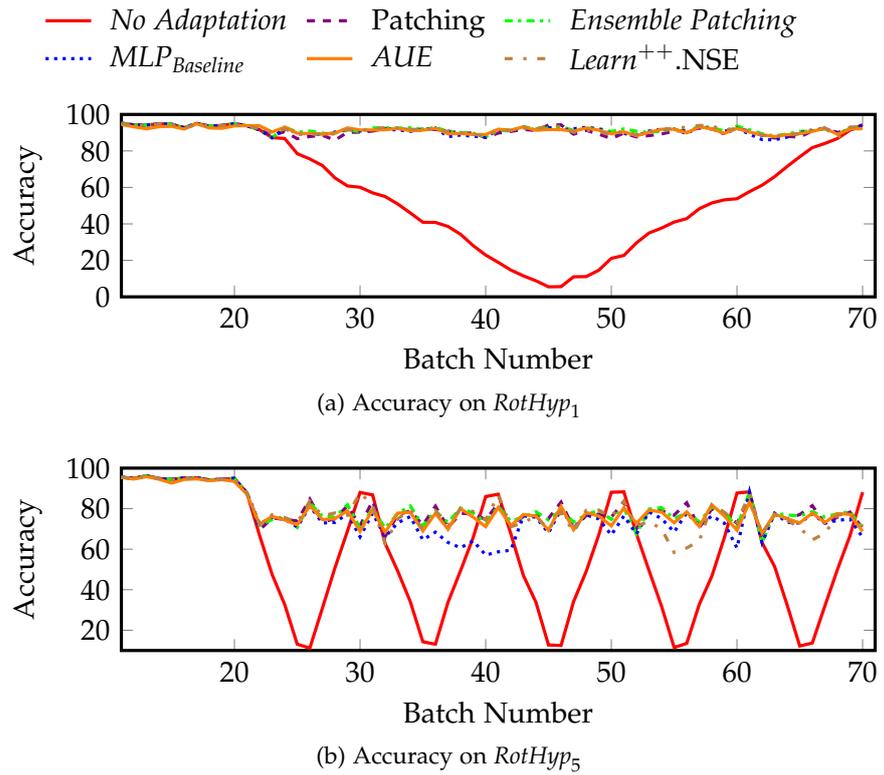


Figure 6.6: Accuracy on gradual drift with different speed.

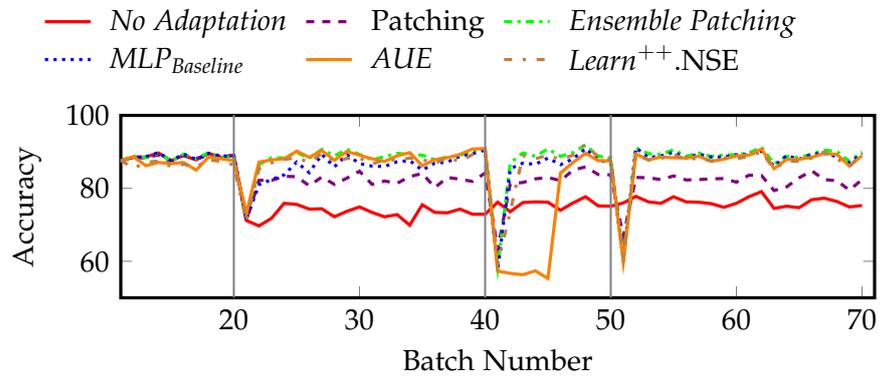


Figure 6.7: Accuracy on SEA

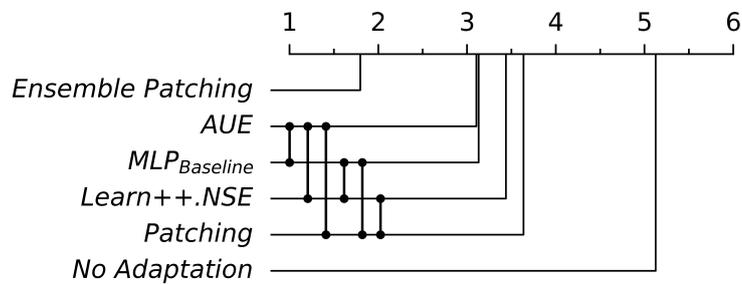


Figure 6.8: Wilcoxon signed-rank test on the adaptation rank.

Table 6.6: Experimental results *SEA*– Overview

Classifier	F.Acc	R.Spd	Ad.Rk	F.Rk
	<i>SEA</i>			
No Adaptation	0.7487	—	5.3480	6.0000
<i>Patching</i>	0.8244	<b>1000</b>	4.5000	5.0000
<i>Ensemble Patching</i>	<b>0.8930</b>	<b>1000</b>	<b>1.5966</b>	<b>1.3611</b>
<i>MLP<sub>Baseline</sub></i>	0.8818	1166	2.8459	3.0000
<i>AUE</i>	0.8846	2533	3.4188	2.5556
<i>Learn<sup>++</sup>.NSE</i>	0.8840	1333	3.0693	2.9444

two algorithms means that these two are *not* significantly different. This type of figure is based on the one proposed by Demšar (2006) for the Nemenyi post-hoc test. As the figure indicates, *Ensemble Patching* significantly differs in adaptation rank from all other methods.

## 6.7.2 Comparison to Patching

In the previous evaluation we compared *Ensemble Patching* to other learners, also including *Patching* from Chapter 4. *Patching* has one crucial parameter that affects its abilities drastically: the amount of data batches that are kept for learning. The more batches are kept, the more it can adapt to complex new concepts. The fewer batches are kept, the faster the adaptation. We use a size one batch limit in the experiments of this chapter, since our goal was not to keep unnecessarily much data.

However, the results show a reduced accuracy compared to the evaluation results of Section 4.6, where a window of eight batches was used.

In order to analyze how *Ensemble Patching* can improve upon *Patching*, we now compare both algorithms directly. *Patching* will be used in two variants: *Patching<sub>8</sub>* with a window size of eight, and *Patching<sub>2</sub>* with a window size of two.

The results on the *MNIST*-based data sets are shown in Figure 6.9. Clearly, *Ensemble Patching* adapts faster, especially on the third concept drift (previous concept). It also shows the highest final accuracy. *Patching<sub>8</sub>* performs similar w. r. t. final accuracy, but is slower in adaptation. *Patching<sub>2</sub>* is quick in adaptation, but its performance hits a certain limit, which also negatively affects the final accuracy scores. On the *SEA*-dataset (Figure 6.10), the exact same behaviour can be observed.

In the case of *RotHyp* (Figure 6.11), the results are quite different. *Patching<sub>8</sub>* cannot cope with rapidly changing concepts (especially Figure 6.11b). By keeping eight batches, it always incorporates information from older concepts into its updating step. *Patching<sub>2</sub>* suffers less from this effect, since it keeps fewer batches. *Ensemble Patching* outperforms both, it is almost unaffected by the constant change.

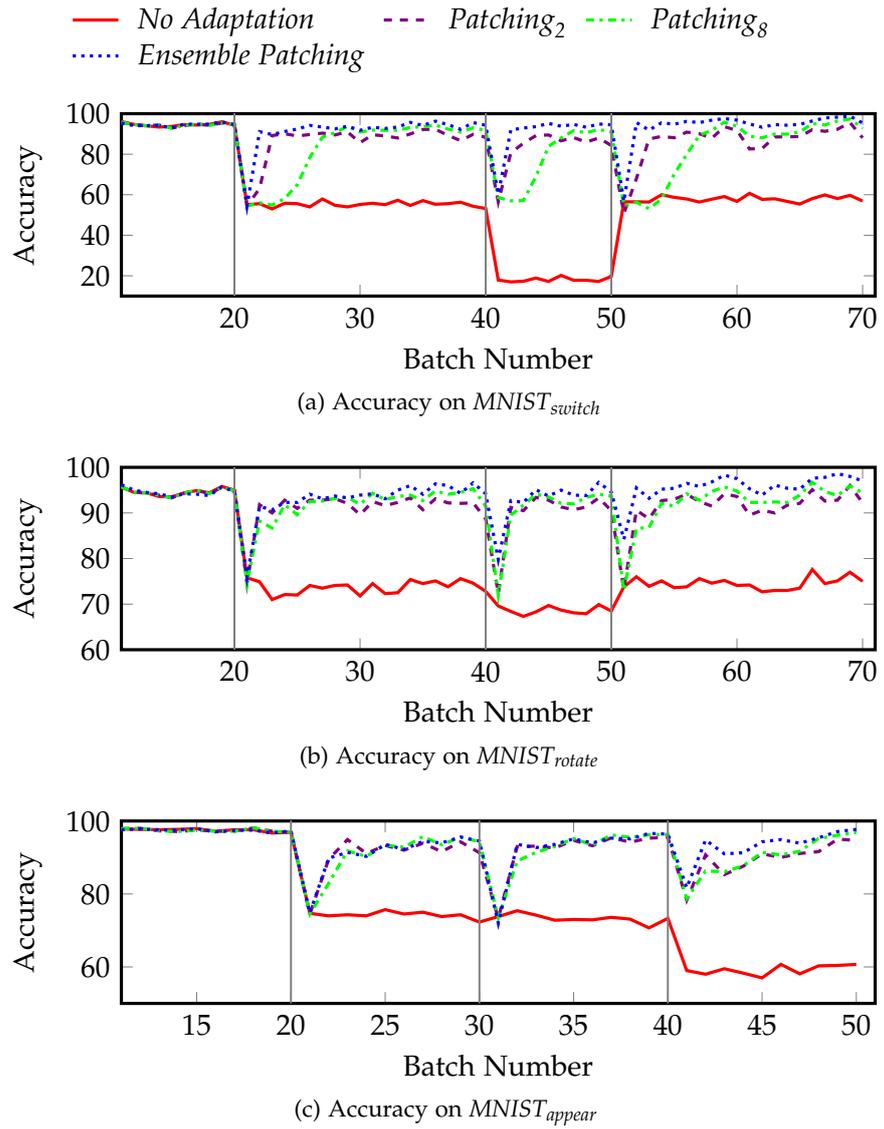


Figure 6.9: Comparison between *Ensemble Patching* and the original *Patching* on the *MNIST*-based data sets.

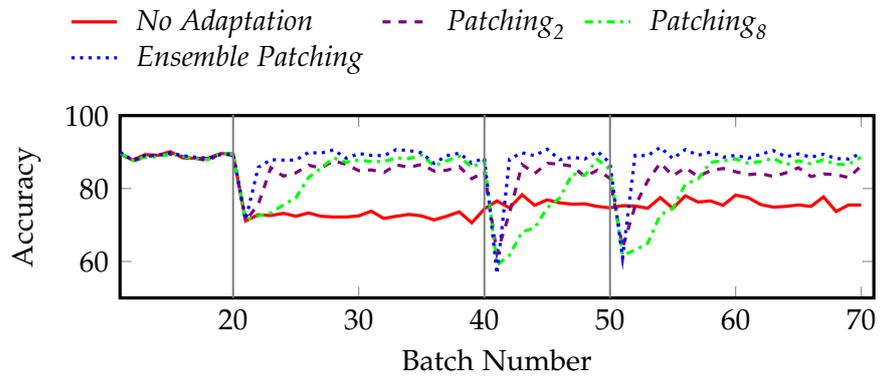


Figure 6.10: Comparison between *Ensemble Patching* and the original *Patching* on *SEA*.

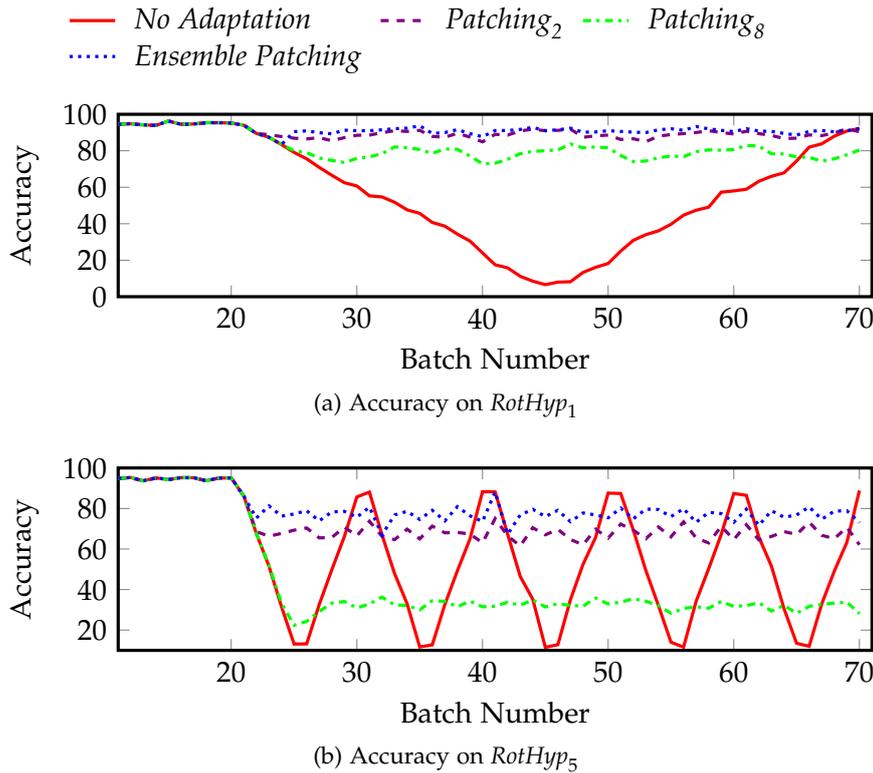


Figure 6.11: Comparison between *Ensemble Patching* and the original *Patching* on *RotHyp*.

## 6.8 Conclusion

In this chapter we have introduced *Ensemble Patching*, a method that is geared towards handling recurring concepts based on the idea of *Patching*. Opposed to *Patching*, *Ensemble Patching* does not determine specific regions in the instance space where the base classifiers errs. Instead it tracks patterns in the misclassification behavior of the base classifier, and based on these patterns decides whether a new patch should be instantiated. Patches are instantiated for new concepts and updated when matching data is seen afterwards. Concepts that are a mixture of previously learned concepts are handled by existing patches, therefore reducing the computational overhead.

We evaluated the method experimentally, and can show very good results in terms of recurring concept adaptation, but also state-of-the-art performance on singular concept drift on gradual or abruptly drifting data. Furthermore, the method is able to handle scenarios where new classes appear.

The process of creating patches based on class-error observations is advantageous towards the previous error-region learning from *Patching* in Chapter 4. It allows us to refine existing patches in situations with ongoing concept drift, and rids us of the necessity to define a certain amount of batches to keep. Since the error-pattern can be detected via an active drift-detection method, it is not as computationally

expensive as the error-region learner. The patches themselves are of simple neural architecture, which might become an issue for more complex adaptation problems. But, as in *Patching*, it is possible to use any (adaptive) learning method as a patch classifier.

However, by the use of a potent ensemble conductor, eventual downsides from these less complex patches are mitigated. The ensemble conductor is the key aspect in this ensemble: its main tasks are (i) understanding the current concept and (ii) based on that, weigh the results of the ensemble members. By this, it somewhat incorporates the same functionality as the error region learner in *Patching* did, but implicitly. We have seen that the instance encoding is helpful: the dimensionality-reduced representation gives the conductor more cues about the instance, as compared to a more passive stacking-approach without any features (Figure 6.4). Of course, this makes the RNN of the conductor more complex to train, but our tests showed no significant degradation in adaptation speed.

Since the number of resulting patches is limited by the number of encountered error-patterns, the management of the ensemble members is simple and straightforward. Members are never removed, so there need not be a mechanism that decides when removing is useful. This enables the ensemble to re-activate patches from the distant past, which can be an advantage for adapting to re-occurring concepts.

However, over very long streams of data, this advantage could turn into a disadvantage. Theoretically, a high number of ensemble members could appear over time, which could affect the ensemble conductor negatively, i. e., its ability to quickly adapt. We will have to conduct further investigations into this direction.

A potential disadvantage is the necessity of explicit drift detection, which is not present in the original *Patching* method. While it is easy to compute, it requires the original concept to remain stable for a while, such that the original model error can be sufficiently observed. It also introduces additional hyperparameters, which could require some fine-tuning for real-world applications.

We provide the source-code implementation for the scikit-multiflow framework on [github](https://github.com/Shademan/EnsemblePatching)<sup>3</sup>.

---

<sup>3</sup> <https://github.com/Shademan/EnsemblePatching>

# Conclusion

---

In this thesis we researched methods and algorithms to adapt existing classifiers for evolving scenarios, under the assumption that we can or do not want to alter the existing systems directly. For many real-world scenarios, a non-invasive approach like this could be the only feasible solution. Reasons for this being necessary could be, that the existing model is a black-box, e. g., it may be cast in hardware. Another reason could be that re-training the classifier is too complex to perform: the data may be sparse, or building a new model too expensive. Since *Machine Learning* is becoming more widely used by practitioners, it is reasonable to assume that not all have in-depth knowledge of the available technologies. A layered approach that builds on top of existing approaches can aid in widening the usage of ML methods and increase practical applicability for a broader audience.

We position our work mostly in the domain of online and data stream learning, as well as transfer learning. Here, the underlying methodology has matured over two decades, and many methods for various scenarios have been implemented and evaluated. Our approaches solve the same problems via the *Patching* principle of adapting existing classifiers.

In the following we conclude our efforts w. r. t. the initial challenges, and discuss potential future research.

## 7.1 Challenges Revisited

In Chapter 1 we stated the research challenges of this work. We now revisit these challenges and recapitulate on the solutions and contributions we have provided to manage them.

1. **Make existing systems adaptable.** We presented the *Patching-Framework* for the adaptation of existing black-box classifiers. We assume that the data is underlying concept drift, and that the original model is still capable of classifying a part of the instances correctly. The *Patching* learning algorithm identifies regions in the instance space, where erroneous classifications occur regularly. It then learns smaller classifiers, so-called patches, for those regions in order to fix these errors. The resulting ensemble consisting of base-classifier and patches can adapt any given classifier to new tasks or altered concepts. We evaluated the framework on various scenarios from concept drift and transfer learning, and showed good results w. r. t. the efficiency and speed of adaptation, as well as overall classification accuracy.

2. **Fix only aspects that are broken.** All three presented variants of *Patching* are focused on identifying errors of the underlying base classifier in order to minimize the learning effort for the adaptation mechanism. While *Patching* learns error regions in instance space, *NN-Patching* learns a more general estimator function for identifying potential errors. We discovered that this process can be as or even more complex than learning the target function itself. Therefore we introduced a compromise solution in *Ensemble Patching*, where the patches are learned in a class-wise fashion based on the actual error of the base classifier.
3. **Leverage existing models as much as possible.**  
With *NN-Patching*, we have shown a special-purpose version of the *Patching* idea. It leverages the inner layers of a neural network for enhanced feature representations, and only learns a small neural network for adapting the base classifier, in the form of a patch network. It established a general error estimator, which identifies potential errors of the base classifier, such that only those errors will be fixed by the patch network. In this way, we retain the functionality of the base classifier, as long as it is not affected by concept drift. When concept drift occurs, the complex latent features can immediately be used for classification, without the need to re-train the original network. This works well in scenarios where the concept drift or the required target task do not differ drastically from the original task.
4. **Increase model re-usability.** All *Patching* variations have the same goal of prolonging the lifetime of a given classifier. Provided that the complex task of learning to estimate potential errors of the base classifier is solvable, all our proposed algorithms can be geared towards fixing said error, hence maximizing the potential lifetime of a classification model. Should the underlying concept drift render the base classifier insufficient, it is still possible to leverage the knowledge and latent features incorporated in a neural network to make the classification task easier for similar target problems.
5. **Learn online.** In online learning scenarios, there are constraints regarding the available storage and computational effort. While the *Patching* framework from Chapter 4 suffers from the necessity to retain a certain amount of data for learning the adaptation, *NN-Patching* and *Ensemble Patching* improve upon this issue. Both can be updated iteratively with batches of new knowledge by retaining only a small amount of data. This improves the practical applicability for these methods to a level where we consider them perfectly capable of handling most online learning scenarios. However, we acknowledge that there may be certain high-frequency learning scenarios, where these methods are restricted by their learning speed. After all, training a moder-

ately complex neural network is still computationally expensive compared to other methods.

6. **Deal with recurring concepts.** Over the course of a data stream, concept drift might occur multiple times, and previously seen concepts may re-occur. Most concept drift learning algorithms do not explicitly solve this scenario, so in the case of recurring concepts, the concept has to be learned again and again. We mitigate this issue in the *Ensemble Patching* algorithm by employing a mechanism that never forgets previous concepts. We identify the concepts based on the pattern of class-wise errors and fix them with specialized patches. An ensemble decision network adjusts the total ensemble output based on the current concept, but older patches remain intact and can be easily reactivated. This way, a fast adaptation for older concepts is reached.

## 7.2 Outlook

In this work we have shown multiple ways of adapting a given classifier to a scenario that is affected by concept drift or some sort of knowledge transfer. In this regard, we have contributed multiple additions to enhance the state-of-the-art in machine learning. However, the presented methods have room for further enhancements regarding the aspects described in the following.

Although we think that the presented algorithms are capable of handling everyday scenarios, for certain high-speed classification and learning tasks the performance might need to be enhanced. This is mostly attributed to the way we deal with the adaptation. While in the basic *Patching* framework any learning algorithm can be deployed into any of the three learning aspects, it does not allow for a continuous learning in an online or incremental way. The more advanced methods *NN-Patching* and *Ensemble Patching* mitigate this issue somewhat by relying on neural networks for most of the learning aspects. However, training a neural network cannot be considered ideal for incremental high-speed tasks. It remains to be evaluated, if they can be swapped with more appropriate learners, especially for learning of the error estimator. This could yield further enhancements, and bring the whole framework one step closer to actual high-speed operation.

As one of the key aspects of the *Patching* methodology is the necessity to identify errors in the base model, investigating alternative ways of learning this estimator is of interest. For example, it would need to be evaluated, if a precise estimation as we implemented it is actually necessary, and how it reflects on the requirements of the patch learner.

In our work we have provided empirically constructed heuristics for *NN-Patching* to establish engagement points for the neural patch as well as the patch architecture itself. Although we have tried to widen the underlying basis for our evaluation, we essentially only covered convolutional and fully connected network architectures. In

preliminary experiments we could already gather some results with networks with residual connections, but they were not yet ready for publication. However, while it is convenient to have some rule-of-thumb heuristic, it would be more reasonable to conduct some optimization process for determining attachment layer and patch architecture. Based on the assumption of limited data availability, this is of course a completely new challenge. While considering such an optimization process, we assume for certain target tasks it may be beneficial to not only have one engagement layer, but multiple instead. This assumption could be naturally explored via such a process.

Regarding our *Ensemble Patching* algorithm, we see two critical points for improvement and deeper investigation. Currently, the ensemble size in *Ensemble Patching* is potentially infinite, only theoretically bounded by the number of encountered concepts. We have not yet evaluated the performance on very long data streams respectively many consecutive concept drifts. The bigger the ensemble gets, the more it might affect the performance and abilities of the ensemble conductor. It might be reasonable to remove ensemble members after prolonged periods of time, of course with the tradeoff of then losing the ability to rapidly recover to old concepts. The ensemble conductor network such as we implemented it is by itself a main point of our ongoing investigation. While the idea of deciding the ensemble as a sequence of the member results is quite novel, it is yet unclear how well this generalizes. Unfortunately we could not yet incorporate a comparison of various ensemble deciding operators such as majority voting, etc. into our evaluation.

Regardless of the potential aspects for improving the three *Patching* methods, we have also envisioned further usage scenarios. One of the more promising ideas is to leverage the patch learning process for a distributed learning scenario. We propose that the participants of the distributed learning system will each learn patches by themselves locally, like the original *Patching* scenario. These patches can then be exchanged with other participants of their peer group. Preliminary research suggests, that partners can benefit from exchanging such information to enhance their own classification experience, as long as the encountered concepts are somewhat similar. This could be especially useful when we think of patching as a kind of individualization method. People with similar interests or behaviors would be able to overcome the cold-start problem of individualization, if we can provide them with a set of proper patches from a central repository. This matching process could be conducted via establishing behavior information of users. Ideally, this information can be encoded in a privacy-preserving manner, which allows a patch selection without knowing anything about the participating users.

## 7.3 Final Remarks

By establishing the *Patching* framework, we have solved various adaptation problems for scenarios with a given, immutable classifier. The solution we provided can compete with state-of-the-art classification methods in various domains, which was not necessarily one of our expectations. We have shown this in various experiments throughout the course of this thesis.

We hope, by providing this framework of adaptation, we can inspire other researchers and practitioners to make valuable use of the technology. We think the idea is novel and appropriate in times like these, where machine learning is one of the main buzzwords in computer science and industry, but still lacks in ease of applicability. Our framework requires only little understanding of the underlying technology, but still enables users to expand their given systems with the capability to adapt to unseen scenarios.

Since developing new machine learning models is expensive, the application of *Patching* can reduce the overall cost for adaptation. The saying “never touch a running system” also applies here. Our method only adapts, where the old classifier errs. This reduces the potential for fixing problems that were not there to begin with, and focuses on mitigating the actual issues.



# Appendix

In the appendix we present additional results and figures.

## A.1 Results for NN-Patching Variants

Fully-Connected Neural Network on <i>MNIST</i>						
Dataset:	<i>MNIST<sub>appear</sub></i>			<i>MNIST<sub>flip</sub></i>		
Model	A.Acc	F.Acc	R.Spdx	A.Acc	F.Acc	R.Spdx
<i>NN-Patching(ME)<sub>incl</sub></i>	87.77	94.79	16.0	87.97	90.89	14.5
<i>NN-Patching(ME)<sub>semi</sub></i>	87.84	94.9	16.4	88.11	90.99	13.9
<i>NN-Patching(ME)<sub>excl</sub></i>	87.24	93.87	15.0	89.07	90.59	19.0
<i>NN-Patching(PE)<sub>incl</sub></i>	<b>87.87</b>	<b>95.12</b>	<b>13.4</b>	<b>90.68</b>	<b>93.68</b>	<b>7.3</b>
<i>NN-Patching(PE)<sub>semi</sub></i>	86.2	92.81	15.7	90.22	93.49	8.0
<i>NN-Patching(PE)<sub>excl</sub></i>	86.7	94.16	16.5	87.35	91.9	14.2
Dataset:	<i>MNIST<sub>remap</sub></i>			<i>MNIST<sub>rotate</sub></i>		
Model	A.Acc	F.Acc	R.Spdx	A.Acc	F.Acc	R.Spdx
<i>NN-Patching(ME)<sub>incl</sub></i>	86.65	89.45	10.6	66.7	68.74	—
<i>NN-Patching(ME)<sub>semi</sub></i>	86.65	89.72	9.7	<b>66.78</b>	<b>68.91</b>	—
<i>NN-Patching(ME)<sub>excl</sub></i>	84.44	88.81	29.5	64.47	62.2	—
<i>NN-Patching(PE)<sub>incl</sub></i>	<b>92.04</b>	<b>95.36</b>	<b>4.9</b>	64.78	68.84	—
<i>NN-Patching(PE)<sub>semi</sub></i>	91.22	95.13	6.0	62.34	62.02	—
<i>NN-Patching(PE)<sub>excl</sub></i>	87.41	91.45	18.4	61.76	61.41	—
Dataset:	<i>MNIST<sub>transfer</sub></i>					
Model	A.Acc	F.Acc	R.Spdx			
<i>NN-Patching(ME)<sub>incl</sub></i>	91.73	<b>95.39</b>	6.0			
<i>NN-Patching(ME)<sub>semi</sub></i>	91.72	95.34	6.0			
<i>NN-Patching(ME)<sub>excl</sub></i>	91.69	95.36	6.0			
<i>NN-Patching(PE)<sub>incl</sub></i>	91.63	95.26	6.0			
<i>NN-Patching(PE)<sub>semi</sub></i>	<b>91.76</b>	95.31	6.0			
<i>NN-Patching(PE)<sub>excl</sub></i>	91.74	95.34	<b>5.9</b>			

Table A.1: Comparison of neural network patching and transfer learning techniques on *MNIST* with FC-NN base classifiers.

Fully-Connected Neural Network on <i>NIST</i>						
Dataset:	<i>NIST<sub>appear</sub></i>			<i>NIST<sub>flip</sub></i>		
Model	A.Acc	F.Acc	R.Sp	A.Acc	F.Acc	R.Sp
<i>NN-Patching(ME)<sub>incl</sub></i>	82.13	85.63	<b>8.4</b>	75.78	82.16	38.5
<i>NN-Patching(ME)<sub>semi</sub></i>	82.1	85.49	9.5	<b>75.8</b>	82.1	37.5
<i>NN-Patching(ME)<sub>excl</sub></i>	80.86	84.99	13.5	74.4	81.16	43.6
<i>NN-Patching(PE)<sub>incl</sub></i>	<b>82.36</b>	<b>86.13</b>	11.7	74.81	82.59	<b>35.7</b>
<i>NN-Patching(PE)<sub>semi</sub></i>	81.93	85.75	10.8	74.7	<b>82.65</b>	35.7
<i>NN-Patching(PE)<sub>excl</sub></i>	79.04	81.37	17.4	72.41	80.92	47.8

Dataset:	<i>NIST<sub>remap</sub></i>			<i>NIST<sub>rotate</sub></i>		
Model	A.Acc	F.Acc	R.Sp	A.Acc	F.Acc	R.Sp
<i>NN-Patching(ME)<sub>incl</sub></i>	78.02	83.69	—	38.49	<b>40.37</b>	—
<i>NN-Patching(ME)<sub>semi</sub></i>	<b>78.18</b>	83.76	—	<b>38.56</b>	40.33	—
<i>NN-Patching(ME)<sub>excl</sub></i>	76.18	82.65	—	34.76	33.31	—
<i>NN-Patching(PE)<sub>incl</sub></i>	77.65	<b>84.57</b>	—	33.54	35.76	—
<i>NN-Patching(PE)<sub>semi</sub></i>	77.09	83.97	—	32.91	35.5	—
<i>NN-Patching(PE)<sub>excl</sub></i>	74.76	82.55	—	32.19	34.46	—

Dataset:	<i>NIST<sub>transfer</sub></i>		
Model	A.Acc	F.Acc	R.Sp
<i>NN-Patching(ME)<sub>incl</sub></i>	64.55	<b>72.9</b>	—
<i>NN-Patching(ME)<sub>semi</sub></i>	<b>64.65</b>	72.85	—
<i>NN-Patching(ME)<sub>excl</sub></i>	64.6	72.96	—
<i>NN-Patching(PE)<sub>incl</sub></i>	60.16	71.8	—
<i>NN-Patching(PE)<sub>semi</sub></i>	61.7	71.81	—
<i>NN-Patching(PE)<sub>excl</sub></i>	61.95	71.95	—

Table A.2: Comparison of neural network patching and transfer learning techniques on *NIST* with FC-NN base classifiers.

Convolutional Neural Network on <i>MNIST</i>						
Dataset:	<i>MNIST<sub>appear</sub></i>			<i>MNIST<sub>flip</sub></i>		
Model	A.Acc	F.Acc	R.Spdl	A.Acc	F.Acc	R.Spdl
<i>NN-Patching(ME)<sub>incl</sub></i>	92.07	98.25	8.1	90.34	93.37	6.1
<i>NN-Patching(ME)<sub>semi</sub></i>	91.91	<b>98.32</b>	9.1	90.3	93.21	6.0
<i>NN-Patching(ME)<sub>excl</sub></i>	90.1	97.83	10.4	89.66	92.91	8.2
<i>NN-Patching(PE)<sub>incl</sub></i>	<b>93.01</b>	98.26	<b>6.4</b>	<b>94.56</b>	<b>97.9</b>	<b>5.6</b>
<i>NN-Patching(PE)<sub>semi</sub></i>	89.48	96.47	12.0	93.54	97.68	5.9
<i>NN-Patching(PE)<sub>excl</sub></i>	90.14	97.81	10.7	90.89	95.43	7.3

Dataset:	<i>MNIST<sub>remap</sub></i>			<i>MNIST<sub>rotate</sub></i>		
Model	A.Acc	F.Acc	R.Spdl	A.Acc	F.Acc	R.Spdl
<i>NN-Patching(ME)<sub>incl</sub></i>	88.42	91.77	6.7	73.24	72.73	—
<i>NN-Patching(ME)<sub>semi</sub></i>	88.27	91.79	6.3	<b>73.4</b>	73.85	—
<i>NN-Patching(ME)<sub>excl</sub></i>	88.19	90.87	9.1	71.75	69.95	—
<i>NN-Patching(PE)<sub>incl</sub></i>	<b>94.89</b>	<b>98.71</b>	<b>1.84</b>	72.48	<b>76.97</b>	—
<i>NN-Patching(PE)<sub>semi</sub></i>	94.52	98.69	4.0	70.4	72.84	—
<i>NN-Patching(PE)<sub>excl</sub></i>	92.9	96.65	5.3	68.79	65.54	—

Dataset:	<i>MNIST<sub>transfer</sub></i>		
Model	A.Acc	F.Acc	R.Spdl
<i>NN-Patching(ME)<sub>incl</sub></i>	94.49	98.67	4.7
<i>NN-Patching(ME)<sub>semi</sub></i>	<b>94.7</b>	<b>98.71</b>	<b>4.0</b>
<i>NN-Patching(PE)<sub>incl</sub></i>	94.04	98.67	4.4
<i>NN-Patching(PE)<sub>semi</sub></i>	94.41	98.69	4.6
<i>NN-Patching(ME)<sub>excl</sub></i>	94.65	98.68	4.7
<i>NN-Patching(PE)<sub>excl</sub></i>	94.36	98.67	4.4

Table A.3: Comparison of neural network patching and transfer learning techniques on *MNIST* with CNN base classifiers.

Convolutional Neural Network on <i>NIST</i>						
Dataset:	<i>NIST<sub>appear</sub></i>			<i>NIST<sub>flip</sub></i>		
Model	A.Acc	F.Acc	R.Spdx	A.Acc	F.Acc	R.Spdx
<i>NN-Patching(ME)<sub>incl</sub></i>	91.55	93.85	4.7	89.18	92.29	8.4
<i>NN-Patching(ME)<sub>semi</sub></i>	<b>91.59</b>	93.96	<b>3.9</b>	89.15	92.18	8.5
<i>NN-Patching(ME)<sub>excl</sub></i>	90.35	93.17	6.4	87.48	91.2	12.4
<i>NN-Patching(PE)<sub>incl</sub></i>	91.38	<b>94.38</b>	6.5	<b>90.52</b>	93.75	<b>7.6</b>
<i>NN-Patching(PE)<sub>semi</sub></i>	90.81	94.36	7.4	90.32	<b>93.79</b>	8.0
<i>NN-Patching(PE)<sub>excl</sub></i>	87.96	89.79	4.5	87.3	91.6	13.1

Dataset:	<i>NIST<sub>remap</sub></i>			<i>NIST<sub>rotate</sub></i>		
Model	A.Acc	F.Acc	R.Spdx	A.Acc	F.Acc	R.Spdx
<i>NN-Patching(ME)<sub>incl</sub></i>	92.63	95.85	5.5	<b>59.83</b>	61.22	—
<i>NN-Patching(ME)<sub>semi</sub></i>	92.79	95.79	<b>4.9</b>	59.75	61.25	—
<i>NN-Patching(ME)<sub>excl</sub></i>	92.52	95.6	6.0	53.91	49.78	—
<i>NN-Patching(PE)<sub>incl</sub></i>	93.46	96.9	5.2	58.53	<b>61.93</b>	—
<i>NN-Patching(PE)<sub>semi</sub></i>	<b>93.83</b>	<b>96.96</b>	5.2	58.13	61.11	—
<i>NN-Patching(PE)<sub>excl</sub></i>	92.79	96.21	5.3	52.65	55.54	—

Dataset:	<i>NIST<sub>transfer</sub></i>		
Model	A.Acc	F.Acc	R.Spdx
<i>NN-Patching(ME)<sub>incl</sub></i>	87.88	93.67	18.5
<i>NN-Patching(ME)<sub>semi</sub></i>	<b>88.07</b>	93.76	<b>17.3</b>
<i>NN-Patching(ME)<sub>excl</sub></i>	88.05	93.68	<b>17.3</b>
<i>NN-Patching(PE)<sub>incl</sub></i>	87.9	93.71	17.8
<i>NN-Patching(PE)<sub>semi</sub></i>	87.72	<b>93.8</b>	17.8
<i>NN-Patching(PE)<sub>excl</sub></i>	87.72	93.76	18.4

Table A.4: Comparison of neural network patching and transfer learning techniques on *NIST* with CNN base classifiers.

## A.2 Adaptation Process in Ensemble Patching

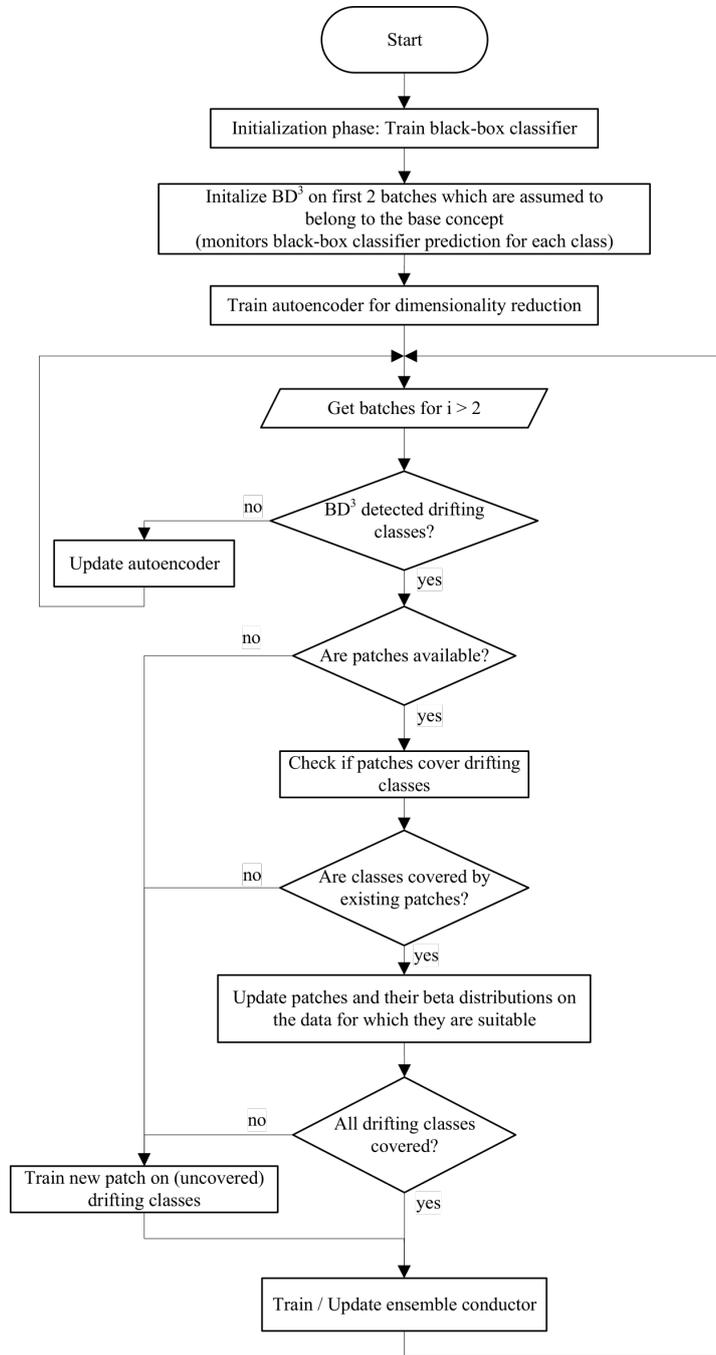


Figure A.1: Adaptation process in the *Ensemble Patching* algorithm.

### A.3 Decay Convergence Proof

In order to proof the convergence of the sum  $\alpha + \beta$  it can be assumed that all samples are misclassified, hence  $k = n$  and  $\alpha + \beta = \alpha$ . Each step of the recursive parameter update can be replaced by a sum as follows:

$$\alpha_i = \frac{\alpha_{i-1}}{d} + n = \frac{\sum_{j=1}^i (d^{j-1} \cdot n) + \frac{\alpha_0}{d}}{d^{i-1}}$$

where  $d$  is an abbreviation for the *decay* parameter. Consequently, the limit can be computed by:

$$\begin{aligned} \lim_{i \rightarrow \infty} \alpha_i + \beta_i &= \lim_{i \rightarrow \infty} \frac{\sum_{j=1}^i (d^{j-1} \cdot n) + \frac{\alpha_0}{d}}{d^{i-1}} \\ &= \lim_{i \rightarrow \infty} \frac{\sum_{j=1}^i (d^{j-1} \cdot n)}{d^{i-1}} \\ &= \lim_{i \rightarrow \infty} \frac{d^{1-i} (d^i - 1)n}{d - 1} \\ &= \frac{n}{d - 1} \lim_{i \rightarrow \infty} d^{1-i} (d^i - 1) \\ &= \frac{n}{d - 1} \lim_{i \rightarrow \infty} d - d^{1-i} \\ &= \frac{n}{d - 1} \lim_{i \rightarrow \infty} d - \lim_{i \rightarrow \infty} d^{1-i} \\ &= \frac{n}{d - 1} d \\ &= n + \frac{n}{d - 1} \end{aligned}$$

# Bibliography

---

- Aimone, James B, Yan Li, Star W Lee, Gregory D Clemenson, Wei Deng, and Fred H Gage (2014). "Regulation and Function of Adult Neurogenesis: From Genes to Cognition." In: *Physiological reviews* 94.4, pp. 991–1026.
- Ba, Jimmy and Rich Caruana (2014). "Do Deep Nets Really Need to be Deep?" In: *Proceedings of the 2014 Conference on Neural Information Processing Systems – NIPS'14*. Curran Associates, Inc., pp. 2654–2662.
- Baena-Garcia, Manuel, José Campo-Avila, Raúl Fidalgo-Merino, Albert Bifet, Ricard Gavaldà, and Rafael Morales-Bueno (2006). "Early Drift Detection Method." In: *Proceedings of the 4th International Workshop on Knowledge Discovery from Data Streams – IWKDDs'06*.
- Basseville, Michèle and Igor V Nikiforov (1993). "Detection of Abrupt Changes: Theory and Application." In: *Change* 2.4, pp. 729–730.
- Ben-David, Shai, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan (2010). "A Theory of Learning from Different Domains." In: *Machine Learning* 79.1-2, pp. 151–175.
- Bengio, Yoshua (2012). "Deep Learning of Representations for Unsupervised and Transfer Learning." In: *JMLR: Workshop and Conference Proceedings*. Vol. 7, pp. 1–20.
- Bifet, Albert and Ricard Gavaldà (2007). "Learning from Time-Changing Data with Adaptive Windowing." In: *Proceedings of the 7th SIAM International Conference on Data Mining – SDM'07*, pp. 443–448.
- (2009). "Adaptive Learning from Evolving Data Streams." In: *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis – IDA'09*, 249–260 Vol. 8.
- Bifet, Albert, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer (2010). "MOA Massive Online Analysis." In: *Journal of Machine Learning Research* 11, pp. 1601–1604.
- Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Praseen Goyal, Lawrence D Jackel, Mathew Monfort, Urs Müller, Jiakai Zhang, et al. (2016). "End to End Learning for Self-driving Cars." In: *arXiv preprint arXiv:1604.07316*.
- Breiman, Leo (Aug. 1996a). "Bagging Predictors." In: *Machine Learning* 24.2, pp. 123–140.
- (1996b). *Bias, Variance, and Arcing Classifiers*. Tech. rep. Statistics Department, University of California at Berkeley.
- (2001). "Random Forests." In: *Machine Learning* 45.1, pp. 5–32.
- Breiman, Leo, Jerome Friedman, Charles J. Stone, and Richard A. Olshen (1984). *Classification and Regression Trees*. Taylor & Francis.
- Brzezinski, Dariusz and Jerzy Stefanowski (2011). "Accuracy Updated Ensemble for Data Streams with Concept Drift." In: *Proceedings of*

- the 6th International Conference on Hybrid Artificial Intelligent Systems – HAIS'11*, 155–163 Vol. 2.
- Canny, J (1986). "A Computational Approach To Edge Detection." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. PAMI-8 (6), pp. 679–698.
- Carpenter, Gail A. and Stephen Grossberg (Jan. 1987). "A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine." In: *Computer Vision, Graphics and Image Processing* 37.1, pp. 54–115.
- Caruana, Rich (July 1997). "Multitask Learning." In: *Machine Learning* 28.1, pp. 41–75.
- Cho, Kyunghyun, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio (2014). "On the Properties of Neural Machine Translation: Encoder-decoder Approaches." In: *arXiv preprint arXiv:1409.1259*.
- Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014). "Learning Phrase Representations using RNN Encoder-decoder for Statistical Machine Translation." In: *arXiv preprint arXiv:1406.1078*.
- Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." In: *arXiv preprint arXiv:1412.3555*.
- Cireşan, Dan C., Ueli Meier, and Jürgen Schmidhuber (2012). "Transfer Learning for Latin and Chinese Characters with Deep Neural Networks." In: *Proceedings of the 2012 International Joint Conference on Neural Networks – IJCNN'12*, pp. 1–6.
- Cohen, William W (1995). "Fast Effective Rule Induction." In: *Proceedings of the 12th International Conference on Machine Learning – ICML '95*, pp. 115–123.
- Dai, Wenyuan, Gui-Rong Xue, Quiang Yang, and Yong Yu (2007). "Transferring Naive Bayes Classifiers for Text Classification." In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence – AAAI'07*, pp. 540–545.
- Dasu, Tamraparni, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi (2006). "An Information-theoretic Approach to Detecting Changes in Multi-dimensional Data Streams." In: *Proceedings of the 2006 Symposium on the Interface of Statistics, Computing Science, and Applications*.
- Demšar, Janez (2006). "Statistical Comparisons of Classifiers Over Multiple Data Sets." In: *Journal of Machine Learning Research* 7, pp. 1–30.
- Domingos, Pedro and Geoff Hulten (2000). "Mining High-Speed Data Streams." In: *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'00*. ACM, pp. 71–80.
- Draeos, Timothy J., Nadine E. Miner, Christopher C. Lamb, Jonathan A. Cox, Craig M. Vineyard, Kristofor D. Carlson, William M. Severa, Conrad D. James, and James B. Aimone (2017). "Neurogenesis Deep Learning: Extending Deep Networks to Accommodate New

- Classes." In: *Proceedings of the 2017 International Joint Conference on Neural Networks – IJCNN'17*, pp. 526–533.
- Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *The Journal of Machine Learning Research* 12, pp. 2121–2159.
- Duda, Richard O, Peter E Hart, and David G Stork (2012). *Pattern Classification*. John Wiley & Sons.
- Elwell, Ryan and Robi Polikar (2011). "Incremental learning of concept drift in nonstationary environments." In: *IEEE Transactions on Neural Networks* 22.10, pp. 1517–1531.
- Esbel, Ousama, Sebastian Kauschke, and Stephan Rinderknecht (May 2019). "Predicting and Forecasting the Lifetime of Automotive Vehicle Components." In: *29. VDI-Fachtagung Technische Zuverlässigkeit 2019*. VDI-Berichte 2345. Düsseldorf: VDI Wissensforum GmbH, pp. 321–336.
- Fleckenstein, Lukas, Sebastian Kauschke, and Johannes Fürnkranz (Apr. 2019). "Beta Distribution Drift Detection for Adaptive Classifiers." In: *Proceedings of the 2019 European Symposium on Artificial Neural Networks – ESANN'19*.
- French, Robert (1999). "Catastrophic Forgetting in Connectionist Networks." In: *Trends in Cognitive Sciences* 3.4, pp. 128–135.
- Freund, Yoav and Robert E. Schapire (1996). "Experiments with a New Boosting Algorithm." In: *Proceedings of the 13th International Conference on International Conference on Machine Learning – ICML'96*, pp. 148–156.
- Frias-Blanco, Isvani, José del Campo-Avila, Gonzalo Ramos-Jiménez, Rafael Morales-Bueno, Agustín Ortiz-Díaz, and Yailé Caballero-Mota (2015). "Online and Non-Parametric Drift Detection Methods Based on Hoeffding's Bounds." In: *IEEE Transactions on Knowledge and Data Engineering* 27.3, pp. 810–823.
- Frias-Blanco, Isvani, Alberto Verdecia-Cabrera, Agustín Ortiz-Díaz, and Andre Carvalho (2016). "Fast Adaptive Stacking of Ensembles." In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing – SAC'16*, pp. 929–934.
- Friedman, Jerome H (2002). "Stochastic Gradient Boosting." In: *Computational statistics & data analysis* 38.4, pp. 367–378.
- Friedman, Jerome H., Trevor Hastie, and Robert Tibshirani (2000). "Additive Logistic Regression: A Statistical View of Boosting." In: *The Annals of Statistics* 38.2, pp. 337–407.
- Friedman, Milton (1937). "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance." In: *Journal of the American Statistical Association* 22, pp. 675–701.
- Fukushima, Kunihiko (Apr. 1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." In: *Biological Cybernetics* 36.4, pp. 193–202.
- Fürnkranz, Johannes (Feb. 1999). "Separate-and-Conquer Rule Learning." In: *Artificial Intelligence Review* 13.1, pp. 3–54.

- Gama, João and Petr Kosina (Sept. 2014). "Recurrent Concepts in Data Streams Classification." In: *Knowledge and Information Systems* 40.3, pp. 489–507.
- Gama, João, Pedro Medas, Gladys Castillo, and Pedro Rodrigues (2004). "Learning with Drift Detection." In: pp. 286–295.
- Gama, João, Raquel Sebastião, and Pedro Pereira Rodrigues (2013). "On Evaluating Stream Learning Algorithms." In: *Machine learning* 90.3, pp. 317–346.
- Gama, João, Indrè Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia (2014). "A Survey on Concept Drift Adaptation." In: *ACM Computing Surveys* 46.4, p. 44.
- Ganin, Yaroslav and Victor Lempitsky (2015). "Unsupervised Domain Adaptation by Backpropagation." In: *Proceedings of the International Conference on Machine Learning – ICML'15*, pp. 1180–1189.
- Gao, Jing, Wei Fan, Jing Jiang, and Jiawei Han (2008). "Knowledge Transfer via Multiple Model Local Structure Mapping." In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'08*, pp. 283–291.
- Geng, Mengyue, Yaowei Wang, Tao Xiang, and Yonghong Tian (2016). *Deep Transfer Learning for Person Re-identification*. Tech. rep.
- Gomes, Heitor Murilo, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet (Mar. 2017). "A Survey on Ensemble Learning for Data Stream Classification." In: *ACM Computing Surveys* 50.2, 23:1–23:36.
- Gomes, João Bártolo, Ernestina Menasalvas, and Pedro AC Sousa (2011). "Learning Recurring Concepts from Data Streams with a Context-aware Ensemble." In: *Proceedings of the 2011 ACM symposium on applied computing*. ACM, pp. 994–999.
- Goncalves Jr, Paulo Mauricio and Roberto Souto Maior De Barros (2013). "RCD: A Recurring Concept Drift Framework." In: *Pattern Recognition Letters* 34.9, pp. 1018–1025.
- Gong, Boqing, Kristen Grauman, and Fei Sha (2013). "Connecting the Dots with Landmarks: Discriminatively Learning Domain-Invariant Features for Unsupervised Domain Adaptation." In: *Proceedings of the 30th International Conference on Machine Learning – ICML'13*. PMLR, pp. 222–230.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press.
- Harries, Michael (1999). *SPLICE-2 Comparative Evaluation: Electricity Pricing*. Tech. rep. The University of South Wales.
- Hashemian, H. M. (2011). "State-of-the-Art Predictive Maintenance Techniques." In: *IEEE Transactions on Instrumentation and Measurement* 60.1, pp. 226–236.
- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (June 2016). "Deep Residual Learning for Image Recognition." In: *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition – CVPR'16*.
- Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). "Reducing the Dimensionality of Data With Neural Networks." In: *Science* 313.5786, pp. 504–507.

- Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. (2001). *Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-term Dependencies*.
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-term Memory." In: *Neural computation* 9.8, pp. 1735–1780.
- Hodges, Cameron, Senjian An, Hossein Rahmani, and Mohammed Bennamoun (2019). "Deep Learning for Driverless Vehicles." In: *Handbook of Deep Learning Applications*. Springer, pp. 83–99.
- Hoeffding, Wassily (1963). "Probability Inequalities for Sums of Bounded Random Variables." In: *Journal of the American Statistical Association* 58.301, pp. 13–30.
- Hoffman, Judy, Sergio Guadarrama, Eric S Tzeng, Ronghang Hu, Jeff Donahue, Ross Girshick, Trevor Darrell, and Kate Saenko (2014). "LSDA: Large Scale Detection Through Adaptation." In: *Advances in Neural Information Processing Systems*, pp. 3536–3544.
- Hornik, Kurt (1991). "Approximation Capabilities of Multilayer Feed-forward Networks." In: *Neural Networks* 4.2, pp. 251–257.
- Hubel, David and Torsten Wiesel (1962). "Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex." In: *Journal of Physiology* 160, pp. 106–154.
- Hulten, Geoff, Laurie Spencer, and Pedro Domingos (2001). "Mining Time-Changing Data Streams." In: *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'01*, pp. 97–106.
- Jaber, Ghazal, Antoine Cornuéjols, and Philippe Tarroux (2013). "Online Learning: Searching for the Best Forgetting Strategy Under Concept Drift." In: *Proceedings of the 20th International Conference on Neural Information Processing – ICONIP'13*. Springer, pp. 400–408.
- Katakis, Ioannis, Grigorios Tsoumakas, and Ioannis Vlahavas (2010). "Tracking Recurring Contexts Using Ensemble Classifiers: An Application to Email Filtering." In: *Knowledge and Information Systems* 22.3, pp. 371–391.
- Kauschke, Sebastian, Lukas Fleckenstein, and Johannes Fürnkranz (July 2019). "Mending is Better than Ending: Adapting Immutable Classifiers to Nonstationary Environments using Ensembles of Patches." In: *Proceedings of the 2019 International Joint Conference on Neural Networks – IJCNN'19*.
- Kauschke, Sebastian and Johannes Fürnkranz (Feb. 2018). "Batchwise Patching of Classifiers." In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence – AAAI'18*.
- Kauschke, Sebastian and David Hermann Lehmann (Dec. 2018). "Towards Neural Network Patching: Evaluating Engagement-Layers and Patch-Architectures." In: *arXiv preprint arXiv:1812.03468*.
- Kauschke, Sebastian, David Hermann Lehmann, and Johannes Fürnkranz (July 2019). "Patching Deep Neural Networks for Nonstationary Environments." In: *Proceedings of the 2019 International Joint Conference on Neural Networks – IJCNN'19*.
- Kingma, Diederik P. and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization." In: *arXiv preprint arXiv:1412.6980*.

- Kirkpatrick, James, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell (2017). "Overcoming Catastrophic Forgetting in Neural Networks." In: *Proceedings of the National Academy of Sciences*.
- Klinkenberg, Ralf and Ingrid Renz (1998). "Adaptive Information Filtering: Learning in the Presence of Concept Drifts." In: *Learning for Text Categorization*, pp. 33–40.
- Kolter, Zico and Marcus Maloof (2007). "Dynamic Weighted Majority: An Ensemble Method for Drifting Concepts." In: *The Journal of Machine Learning Research* 8, pp. 2755–2790.
- Kosina, Petr and João Gama (2015). "Very Fast Decision Rules for Classification in Data Streams." In: *Data Mining and Knowledge Discovery* 29.1, pp. 168–202.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "Imagenet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*.
- Kubat, Miroslav (1989). "Floating Approximation in Time-Varying Knowledge Bases." In: *Pattern Recognition Letters* 10.4, pp. 223–227.
- Lanquillon, Carsten (2001). "Enhancing Text Classification to Improve Information Filtering." PhD thesis. Otto-von-Guericke-Universität Magdeburg, Universitätsbibliothek.
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). "Deep Learning." In: *Nature* 521.7553, pp. 436–444.
- LeCun, Yann, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (Dec. 1989). "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Comput.* 1.4, pp. 541–551.
- Lecun, Yann, Leon Bottou, Yoshua Bengio, and Patrick Haffner (1998). "Gradient-based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.
- Long, Mingsheng, Yue Cao, Jianmin Wang, and Michael I Jordan (2015). "Learning Transferable Features with Deep Adaptation Networks." In: arXiv: 1502.02791.
- Mitchell, Thomas M. (1997). *Machine Learning*. 1st ed. McGraw-Hill, Inc.
- Montiel, Jacob, Jesse Read, Albert Bifet, and Talel Abdesslem (2018). "Scikit-Multiflow: A Multi-output Streaming Framework." In: *Journal of Machine Learning Research* 19.72, pp. 1–5.
- Nemenyi, Peter (1963). "Distribution-Free Multiple Comparisons." PhD thesis. Princeton University.
- Nishida, Kyosuke, Koichiro Yamauchi, and Takashi Omori (2005). "ACE: Adaptive Classifiers-ensemble System for Concept-drifting Environments." In: *Proceedings of the 6th International Conference on Multiple Classifier Systems – MCS'05*, pp. 176–185.
- Oquab, Maxime, Leon Bottou, Ivan Laptev, and Josef Sivic (2014). "Learning and Transferring Mid-Level Image Representations using

- Convolutional Neural Networks." In: *IEEE Conference on Computer Vision and Pattern Recognition – CVPR'14*, pp. 1717–1724.
- Ortega, Julio, Moshe Koppel, and Shlomo Argamon (2001). "Arbitrating Among Competing Classifiers Using Learned Referees." In: *Knowledge and Information Systems 3.4*, pp. 470–490.
- Ortíz Díaz, Agustín, José del Campo-Ávila, Gonzalo Ramos-Jiménez, Isvani Frías Blanco, Yailé Caballero Mota, Antonio Mustelier Hechavarría, and Rafael Morales-Bueno (2015). "Fast Adapting Ensemble: A New Algorithm for Mining Data Streams with Concept Drift." In: *The Scientific World Journal 2015*.
- Oza, Nikunj C. (2005). "Online Bagging and Boosting." In: *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics – SMC'05*, pp. 2340–2345.
- Pan, Sinno Jialin and Qiang Yang (2010). "A Survey on Transfer Learning." In: *IEEE Transactions on Knowledge and Data Engineering 22.10*, pp. 1345–1359.
- Peng, Ying, Ming Dong, and Ming Jian Zuo (2010). "Current Status of Machine Prognostics in Condition-based Maintenance: A Review." In: *International Journal of Advanced Manufacturing Technology (1-4)*, pp. 297–313.
- Polikar, Robi, Lalita Udpa, Satish S. Udpa, and Vasant Honavar (2001). "Learn++: An Incremental Learning Algorithm for Supervised Neural Networks." In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews 31.4*, pp. 497–508.
- Quinlan, J. Ross (1986). "Induction of Decision Trees." In: *Machine Learning 1.1*, pp. 81–106.
- (1993). *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Rosenblatt, Frank (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review 65*, pp. 386–408.
- Ross, Gordon J., Niall M. Adams, Dimitris K. Tasoulis, and David J. Hand (2012). "Exponentially Weighted Moving Average Charts for Detecting Concept Drift." In: *Pattern Recognition Letters 33.2*, pp. 191–198.
- Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning Representations by Back-propagating Errors." In: *Nature 323.6088*, p. 533.
- Saenko, Kate, Brian Kulis, Mario Fritz, and Trevor Darrell (2010). "Adapting Visual Category Models to New Domains." In: *Proceedings of the 11th European Conference on Computer Vision – ECCV'10*. Springer, pp. 213–226.
- Schlimmer, Jeffrey C. and Richard H. Granger (1986). "Incremental Learning from Noisy Data." In: *Machine Learning 1.3*, pp. 317–354.
- Schmidhuber, Jürgen (2015). "Deep Learning in Neural Networks: An Overview." In: *Neural Networks 61*, pp. 85–117.
- Seewald, Alexander K. and Johannes Fürnkranz (2001). "An Evaluation of Grading Classifiers." In: *Advances in Intelligent Data Analysis: Proceedings of the 4th International Conference – IDA'01*, pp. 115–124.

- Sipos, Ruben, Dmitriy Fradkin, Fabian Moerchen, and Zhuang Wang (2014). "Log-based Predictive Maintenance." In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'14*. ACM, pp. 1867–1876.
- Street, W Nick and YongSeog Kim (2001). "A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification." In: *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'01*. ACM, pp. 377–382.
- Ting, Kai Ming and Ian H. Witten (1999). "Issues in Stacked Generalization." In: *Journal of Artificial Intelligence Research* 10, pp. 271–289.
- Torrey, Lisa and Jude Shavlik (2009). "Transfer Learning." In: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. Vol. 1. Information Science Reference, pp. 242–264.
- Tsymbol, Alexey (2004). "The Problem of Concept Drift: Definitions and Related Work." In: *Computer Science Department, Trinity College Dublin* 106.2.
- Tzeng, Eric, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell (2014). *Deep Domain Confusion: Maximizing for Domain Invariance*. Tech. rep.
- Valiant, Leslie G (1984). "A Theory of The Learnable." In: *Proceedings of the 16th annual ACM symposium on theory of computing*. ACM, pp. 436–445.
- Wang, Haixun, Wei Fan, Philip S Yu, and Jiawei Han (2003). "Mining Concept-Drifting Data Streams Using Ensemble Classifiers." In: *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD'03*. AcM, pp. 226–235.
- Webb, Geoffrey I., Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean (July 2016). "Characterizing Concept Drift." In: *Data Mining and Knowledge Discovery* 30.4, pp. 964–994.
- Widmer, Gerhard and Miroslav Kubat (1992). "Learning Flexible Concepts from Streams of Examples: FLORA2." In: *Proceedings of the 10th European Conference on Artificial Intelligence – ECAI'92*, pp. 463–467.
- (1993). "Effective Learning in Dynamic Environments by Explicit Context Tracking." In: *Proceedings of the European Conference on Machine Learning – EMCL'93*, pp. 227–243.
- (1996). "Learning in the Presence of Concept Drift and Hidden Contexts." In: *Machine Learning* 23.1, pp. 69–101.
- Wilcoxon, Frank (1945). "Individual Comparisons by Ranking Methods." In: *Biometrics bulletin* 1.6, pp. 80–83.
- Witten, Ian H. and Eibe Frank (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.
- Wolpert, David H. (1992). "Stacked Generalization." In: *Neural Networks* 5.2, pp. 241–260.
- Wolpert, David H. and William G. Macready (1996). *Combining Stacking with Bagging to Improve a Learning Algorithm*. Tech. rep. SFI-TR-96-03-123. Santa Fe, New Mexico: Santa Fe Institute.

- Yosinski, Jason, Jeff Clune, Yoshua Bengio, and Hod Lipson (2014). "How Transferable are Features in Deep Neural Networks?" In: *Proceedings of the 27th International Conference on Neural Information Processing Systems – NIPS'14*, pp. 3320–3328.
- Zeiler, Matthew D. and Rob Fergus (2013). "Visualizing and Understanding Convolutional Networks." In: *CoRR* abs/1311.2901.
- Zhou, Yi-Tong and Rama Chellappa (1988). "Computation of Optical Flow Using a Neural Network." In: *IEEE International Conference on Neural Networks*. Vol. 1998, pp. 71–78.



# Erklärung zur Dissertation

---

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

*Darmstadt, 16.09.2019*

---

Sebastian Kauschke