Patching Deep Neural Networks for Nonstationary Environments

———————[1,2], ———————, and ———————[1]
[1] ———————
[2] ———————
———————, ———————

Abstract

In this work we present neural network patching, an approach for adapting deep neural network models to nonstationary environments. Instead of creating or updating a network to accomodate concept drift, neural network patching leverages the inner layers of a previously trained network as well as its output to learn a patch that enhances the classification. It learns (i) a predictor that estimates whether the original network will misclassify an instance, and (ii) a patching network that fixes the misclassification. Neural network patching is based on the idea that the original network can still classify a majority of instances well, and that the inner feature representations encoded in the deep network aid the classifier to cope with unseen or changed inputs. We evaluated this technique on several datasets, comparing it to similar methods. Our finding is that neural network patching is adapting quickly to concept shifts, while also maintaining long-term learning capabilities similar to more complex methods that update the whole network.

1 Introduction

Nowadays, deep neural networks (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 Hinton 2015; Schmidhuber 2015).

The main advantage of deep networks is their layered architecture, which turns out to be easier to train compared to networks with a single hidden layer, given enough training data is present. The possibility of training bigger and deeper networks has enabled neural networks to deal with more complex problems. Each of these 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 like edges or corners. Later layers comprise more complex features related to the given task, for example eyes or ears, when recognizing faces.

Due to the vast amounts of data available today, building highly capable deep neural networks for certain tasks has become feasible. However, most domains are subject to changing conditions in the long run. That means, either the data, the data distribution, or the target classification function changes. This is usually caused by concept drift or other kinds of non-stationarity. The result is that once perfectly capable systems degrade in their performance or even become unusable over time.

An example could be a piece of complex machinery, as used in productive environments such as factories. This machine might be fitted with hard- and software to finely detect its current state, and a predictive model for failures on top of it. When the next hardware revision of that machine is sold by the manufacturer, new data from the machine has to be collected and the failure predicting model has to be retrained, which can be very expensive. Another example is the need to learn new classes, that had not been there before. A final example to motivate the necessity of adaptation is the personalization setting. A product is sold with a general prediction model that covers a wide variety of users. However, personalization would help to make it even more suitable for a specific user. This is a type of adaptation that is difficult to manage with neural networks as underlying models.

In this paper, we give a solution to these problems. 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 hyperparameters. 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 solve this problem, we build upon *patching*, a framework that has recently been proposed to cope with such problems (Kauschke and Fürnkranz 2018). Contrary to many conventional techniques, this framework does not assume that it is feasible to re-train the model from scratch with newly recorded data. Instead, it tries to recognize regions where the model errs, and tries to learn local models—so-called *patches*—that repair the original model in these error regions.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper, 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 is not only fed by the output of the base network, but also leverages inner layers of the network that enhance its capabilities. Furthermore, the patching network is only activated, when the underlying base network gives erroneous results.

This paper is structured as follows. In Section 2 we discuss related work from the domain of adaptive learning. In Section 3 we elaborate on the concept of the original *Patching*, and go into detail about *NN-Patching* in Section 4. We present the experiments in Section 5 and give experimental results in Section 6. We conclude our findings in Section 7.

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/transfer learning and adaptive neural networks is generally relevant to our approach. In this section we will elaborate on related work in these fields.

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 and good process that happens gradually. However, in artificial neural networks trained by back-propagation, learning new information leads to a reconfiguration of the weights in the network, which 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.

2.2 Concept Drift

Concept drift learning is concerned with a supervised scenario, where the observed data is a nonstationary, continuous stream, which changes over time w.r.t. the data distribution or the target variable. It is supervised in the way that the true labels will become available shortly after the prediction and can then be used for enhancing the classifier for future predictions. Gama et al. (2014) give an overview on the state-of-the-art of concept drift adaptation. They elaborate on the goals and the basic assumptions that are made in this research domain. Webb et al. (2016) provide a taxonomy that classifies concept drift in multiple dimensions such as duration, type of transition, re-occurrence, and more. Recent work on concept drift is centered on classification of massive amounts of stream data, where fast processing and reduction of overall resources is paramount (Aggarwal 2014). Seminal work on concept drift was conducted by Widmer and Kubat (1996) with the FLORA family of algorithms. Other well known contributions such as Bifet and Gavaldà (2009) have carried over this work into the current decade.

However, 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*$^{++}$ by Polikar et al. (2001) and variations thereof.

2.3 Transfer Learning

In transfer learning (TL), 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), and is highly related to 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 (convolutional) neural networks 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 convolutional networks 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 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.

3 The Concept of Patching

The idea of the original *Patching* algorithm as described in Kauschke and Fürnkranz (2018) is as follows: We assume a general instance space D_0 of instances x with labels $l(x)$, as well as a black-box classifier C_0. C_0 is immutable and can classify the examples of D_0 well. Caused by a nonstationary environment, a new batch of data $D_i, i > 0$ is received, for which C_0 makes imperfect predictions. The goal is to learn a classifier C_i that approximate l_i as closely as possible in three steps:

(i) A classifier E_i is trained, that is able to identify the so-called *error region* where C_0 misclassifies data of D_i.

(ii) A new classifier C_i, a so-called *patch*, is learned on the misclassified instances.

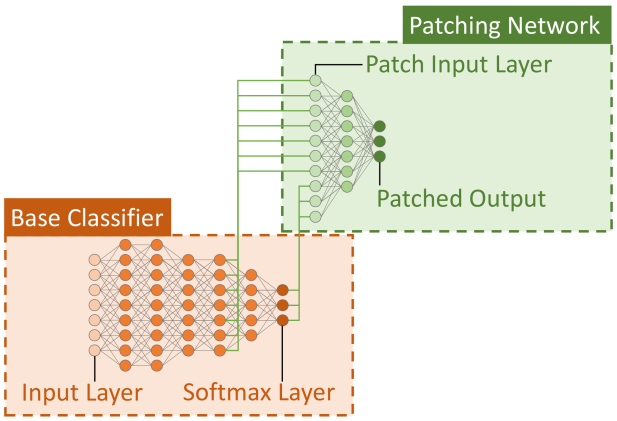

Figure 1: Patching of feed-forward networks – orange: the original network, green: the patch

(iii) At classification time, C_i is used if E_i predicts an instance to be part of the error region, otherwise C_0 is used.

The method works especially well when only part of the instance population is affected by change, since the base classifier remains static and can deal with the unchanged population as well as before. Furthermore, when the change reverts, e.g. due to seasonal effects, the errors will vanish and the underlying base classifier will take over the classification again. One of the downsides of this concept is that error region recognizer and patch are re-trained for every new batch of information that is gathered. This is sub-optimal w.r.t. training performance, because a certain amount of older instances must be kept for training. The original *Patching* can be stacked onto every type of classification model, it works on the instance attributes and the output of the base classifier, and uses classical machine learning methods such as rule learners or decision trees to learn the patches.

The goal of our work is to leverage the strengths of *Patching* and advance it for the special case of neural network classifiers. More specifically, we want to improve it so that it can leverage the inner layers of a deep neural network. In these layers, abstractions and representations are stored that may be crucial to the classification. We also want to improve the training process such that it is iterative and continuous instead of disruptive. In the next section, we describe how these advances are achieved.

4 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 Schmidhuber (2012) leverage this behavior to achieve transfer to problems with higher complexity than the original problem the network was intended for.

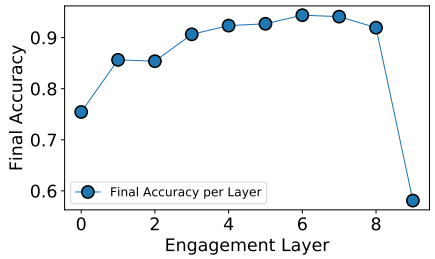

Figure 2: Engagement Layer — Final accuracy with varying engagement layers in a convolutional network with 8 hidden layers as experienced with the $NIST_{flip}$ dataset.

In summary, we make three observations: (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 next section we will explain the patching procedure for neural networks.

4.1 Patching for Neural Networks

We tailored the *Patching*-procedure by Kauschke and Fürnkranz (2018) to the specific case of neural network classifiers. The idea is depicted in Figure 1. *NN-Patching* therefore consists of three steps:

1. **Learn a classifier E that determines where C_0 errs.** In this step, when receiving a new batch of labeled data, the data is used to learn a classifier that estimates where C_0 will misclassify instances.
2. **Learn a patch network P.** The patch network engages to one inner layer of C_0 and the last layer (usually a softmax for classification), and takes the activations of these layers as its own input (Fig. 1).
3. **Divert classification from C_0 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 C_0.

In contrast to the original procedure (Sect. 3), 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 C_0. The selection of this layer is non-trivial. We established two rules of thumb for either fully connected networks and convolutional neural networks.

Fully connected neural network The engagement layer is the second layer of the network.

Convolutional neural network The engagement layer is the last pooling or convolutional layer in the network.

These rules were derived from preliminary experiments that we conducted, but follow the observations of Yosinski et al. (2014) that earlier layers in the network tend to encompass more general and later layers more specific abstractions. In our case this means that for fully connected networks, using the more general features (earlier layers) was more promising, whereas in convolutional networks using the more specific features (pooling layers) towards the end of the network worked well. An example of engaging to different layers in a convolutional network is shown in Fig. 2. Early engagement layers result in a lower accuracy than later layers, the exception being the final (in this case a Softmax) layer.

4.2 Patch Architecture

In addition to determining the engagement layer, we have to create the patch P itself as a neural network. The ideal architecture of the patch depends on the size of the engagement layer as well as on the amount of data we are dealing with. 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 problem to begin with. A patch always consists of an input layer and a softmax layer for classification, but may contain hidden layers in between. In preliminary experiments we tested patch architectures with varying input size from 2048 down to 128 units, and depths from 1–3 hidden layers. With every hidden layer, the amount of units was halved towards the size of the output layer (usually 10 or 36 units). The experiments showed that a patch with **a single hidden layer consisting of 512 units** is reasonable for all experiments. It is not optimal w.r.t. accuracy or adaptation speed, because those are problem-specific, but performed well on average. We only used fully-connected layers with ReLU activations in our patch. However, keep in mind that for certain problems, specific architectures such as convolutional layers might be much better suited. Moreover, we need to learn the error detector function E.

We created two approaches to neural network patching:

NN-Patching This variant is described in Section 4.1. It contains the error decision function E and the patching network P. Both 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 plus the classification layer of C_0, contains one hidden layer with 512 units 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$_{ms}$ This variant replaces the error decision E with a **model selection** (ms) M. The model selection not only estimates the ability of C_0 to classify an instance, but also estimates if the patch P is (already) able to do so. Especially in the first batches of adaptation when P is not yet well trained, it can often be better to use C_0, although E indicates that C_0 is flawed. M is a neural network equal to E, but contains a softmax output for estimating probabilities on the correctness of P and C_0 combined. The classification is diverted to P, when $Pr(P) > Pr(C_0)$.

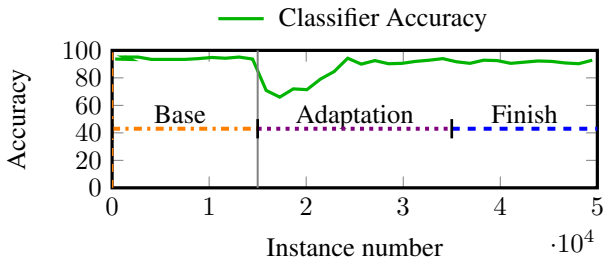

Figure 3: Example of a stream with concept drift. The drift is marked by the vertical line. Three phases can be identified.

5 Experimental Setup

In this section, we will elaborate on the datasets we used as well as the experiment setup itself. Our datasets are derived from well known datasets 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 it allows us to retrospectively evaluate the performance of the classifier, and make adaptations for the next batch. Bifet et al. (2010) describe this process more thoroughly w.r.t. their Massive Online Analysis (MOA) framework. We applied the same principles, although we implemented our solution in Python.

5.1 Evaluation Measures

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

- *Final Accuracy (F.Acc):* Classification accuracy, measured in the *Finish* phase, which consists of the last five batches of the stream.
- *Average Accuracy (Avg.Acc): Average accuracy in the Adaptation and Finish phases (after first change point).*
- *Recovery Speed (R.Spd):* Number of instances that a classifier requires during the *Adaptation* phase to achieve 90% 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.

5.2 Evaluation Datasets

We will 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 1.

Table 1: Summary of the datasets used in the experiments

Dataset	Init	CPs	Total	Chunks
MNIST Dataset				
$MNIST_{flip}$	40k	#70k	140k	100
$MNIST_{rotate}$	20k	#35k	70k	100
$MNIST_{appear}$	15k	#20.4k	57.2k	100
$MNIST_{remap}$	20k	#35.7k	70k	100
$MNIST_{transfer}$	20k	#35k	70k	100
NIST Dataset				
$NIST_{flip}$	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

Table 2: Network architecture for fully connected and convolutional network, as used in the evaluation to build the base network.

Architecture: Fully Connected
FC(786) - FC(2048) - FC(1024) - FC(1024) - FC(512) - FC(128) - Dropout(0.5) - Softmax(#classes)

Architecture: Convolutional
Conv2D(7,7) - MaxPool(2,2) - Conv2D(5,5) - Conv2D(5,5) - Conv2D(3,3) - Conv2D(3,3) - MaxPool(2,2) - Dropout(0.5) - FC(256) - Softmax(#classes)

FC(n): Fully Connected, n = #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)

The MNIST Dataset. The first dataset is the *MNIST*[1] 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 the labeling. We created the following scenarios.

$MNIST_{flip}$ The second half of the dataset consists of vertically and horizontally flipped digits.

$MNIST_{rotate}$ The digits in the dataset are rotated at a random angle from instance #35k onwards with increasing degree of rotation up to 180 degrees (at #65k).

$MNIST_{appear}$ The digits change during the stream, such that classes 5–9 do not exist in the beginning, but only start to appear at the change point (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). Here we only have 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*[2] dataset of handprinted forms and characters. It contains 810,000 digits and characters, to which we will 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 will use all digits 0-9 and uppercase characters A-Z for a total of 36 classes as data stream and draw a random sample for each scenario.

$NIST_{flip}$ The second half of the dataset consists of vertically and horizontally flipped images.

$NIST_{rotate}$ The images in the dataset start to rotate randomly at instance #40k with increasing rotation up to 180 degrees for the last 10k instances.

$NIST_{appear}$ 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 change point (mixed in between the characters A–Z).

$NIST_{remap}$ 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. Here we only have 10 classes.

$NIST_{transfer}$ The first 30k instances of the stream only consists of digits 0–9, while the following 80k are solely characters A–Z.

[1] http://yann.lecun.com/exdb/mnist/

[2] https://www.nist.gov/srd/nist-special-database-19

5.3 Compared Methods

In this section, we explain the variants of *NN-Patching* and other 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 2) 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*.

***NN-Patching*:** Neural network patching with an error estimator E that predicts, if C_0 will misclassify an instance.

***NN-Patching$_{ms}$*:** Neural network patching with a model selector M that predicts whether C_0 will misclassify and P will correctly classify an instance.

***Freezing*:** This is based on a common transfer method (Oquab et al. 2014). The layers up to and including the engagement layer are static, and the remaining layers are updated via backpropagation when possible. Whereas the patching network is randomly initialized, the transfer network is initialized with the weights from C_0.

***Baseline*:** The original network C_0, unaltered.

***Base$_{update}$*:** This variant updates all weights in the base network via backpropagation. This affects a significantly higher number of weights compared to *Freezing* or *NN-Patching* during training, but has more parameters to adjust and therefore a higher overall capability for new concepts.

Table 3: Results for *MNIST* (Base classifier is a fully connected deep network)

Classifier	F.Acc	Avg.Acc	R.Spd	Ad.Rk	F.Rk
	***MNIST**$_{flip}$*				
Baseline	29.1	31.4	—	4.78	5.0
NN-Patching	91.8	89.5	8.2	2.21	3.6
NN-Patching$_{ms}$	94.0	**91.7**	**5.6**	**1.41**	2.0
Freezing	94.1	84.1	16.8	3.39	1.9
Base$_{update}$	**94.8**	85.1	15.5	2.89	**1.0**
	***MNIST**$_{rotate}$*				
Baseline	46.6	46.5	—	4.43	4.1
NN-Patching	67.3	66.4	—	2.89	2.9
NN-Patching$_{ms}$	68.9	68.2	—	2.91	2.3
Freezing	69.9	67.9	—	2.60	**1.8**
Base$_{update}$	**70.2**	**70.0**	—	**1.45**	2.1
	***MNIST**$_{appear}$*				
Baseline	51.0	50.6	—	4.32	5.0
NN-Patching	95.2	89.0	15.8	1.36	**1.9**
NN-Patching$_{ms}$	**95.4**	**89.4**	**11.2**	**1.14**	**1.9**
Freezing	93.8	79.5	33.3	3.41	3.1
Base$_{update}$	**95.4**	78.8	32.1	3.85	1.4
	***MNIST**$_{remap}$*				
Baseline	40.5	41.3	—	4.76	5.0
NN-Patching	90.4	87.8	6.30	2.86	3.5
NN-Patching$_{ms}$	**95.8**	**92.7**	**3.1**	**1.43**	1.6
Freezing	95.2	88.6	13.6	3.12	2.1
Base$_{update}$	**95.8**	89.8	9.8	2.49	**1.3**
	***MNIST**$_{transfer}$*				
Baseline	0.0	0.0	—	4.55	5.0
NN-Patching	**95.6**	**92.1**	**3.8**	**1.59**	**1.2**
NN-Patching$_{ms}$	**95.6**	92.0	**3.8**	1.88	1.5
Freezing	93.4	72.3	24.4	3.08	3.0
Base$_{update}$	93.6	68.7	24.6	3.35	2.5

6 Experimental Results

As described in the previous section, we conducted experiments on ten problems with varying types of concept drift or transfer. 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*).

6.1 Deep Networks – The *MNIST* dataset

On the *MNIST* datasets, all adapting algorithms are able to achieve similar results for the final accuracy. Here, *NN-Patching* is applied to deep, fully connected networks. Similar to the *NIST* results, *NN-Patching* and *NN-Patching*$_{ms}$ achieve the best adaptation ranks in four out of five datasets, as can be seen from Table 3. The *Base*$_{update}$-method can profit from its higher capabilities (all weights are being trained) and achieve a high final accuracy in five datasets, albeit paired with a slow adaptation. The example of *MNIST*$_{flip}$ in Figure 4 shows that the more complex model selector in *NN-Patching*$_{ms}$ improves over *NN-Patching*. It diverts more classification decisions to the patch, which is beneficial in the long run. This can be concluded from the oscillating behavior that *NN-Patching* shows in Figure 6, which we also observed in other datasets.

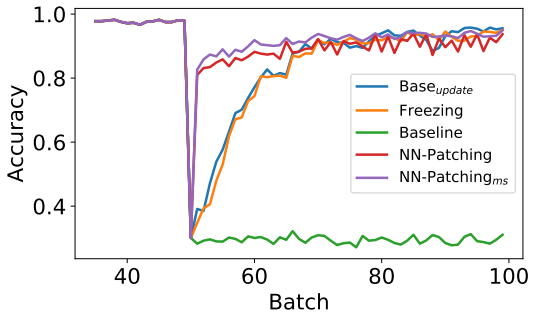

Figure 4: Results on the *MNIST*$_{flip}$ dataset as a data stream.

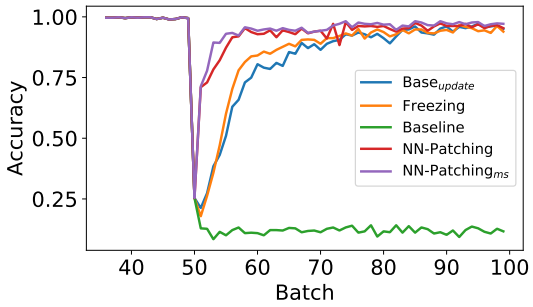

Figure 5: Results on the *NIST*$_{remap}$ dataset as a data stream. Dataset was divided into 100 batches, the first 20 batches were used for training the base classifier C_0.

6.2 Convolutional Networks – The *NIST* dataset

In the *NIST* datasets, *NN-Patching* is applied to a convolutional network. As we can see in the example plot in Figure 5, both *NN-Patching*-variants adapt significantly faster than the other methods. In Table 4, we can see that in all of the *NIST* datasets but one, *NN-Patching* or *NN-Patching*$_{ms}$ achieve the top rank in the adaptation phase. *NN-Patching*$_{ms}$ also achieves the highest average accuracy in all datasets. For the final rank and accuracy, the results are similar, only *Base*$_{update}$ is able to adapt better in *NIST*$_{appear}$. This can be observed in Figure 6, where *Base*$_{update}$ adapts slower, but in the end more successfully. It has the highest number of trainable weights of all compared methods and can therefore learn more complex concepts, which is an advantage here, and probably on all datasets in which the stream of data is of arbitrary length without additional concept changes.

6.3 Significance Testing

Demšar (2006) suggests the Friedmann test for comparison of multiple classifiers, which we conducted on the average ranks. For both adaptation rank and 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 Figures 7

Table 4: Results for *NIST* (Base classifier is a convolutional network)

Classifier	F.Acc	Avg.Acc	R.Spd	Ad.Rk	F.Rk
	NIST$_{flip}$				
Baseline	17.4	17.7	—	4.95	5.0
NN-Patching	92.5	89.6	9.2	1.64	1.7
NN-Patching$_{ms}$	**94.1**	**90.8**	**8.5**	**1.25**	**1.0**
Freezing	88.5	71.5	39.2	3.10	3.2
Base$_{update}$	89.1	66.7	41.6	3.93	2.8
	NIST$_{rotate}$				
Baseline	43.1	39.1	—	3.22	3.0
NN-Patching	59.5	59.2	—	3.57	2.2
NN-Patching$_{ms}$	**60.9**	**60.4**	—	3.31	**2.0**
Freezing	58.4	52.4	—	**2.11**	2.1
Base$_{update}$	56.2	51.2	—	2.31	2.9
	NIST$_{appear}$				
Baseline	71.2	69.8	—	4.23	5.0
NN-Patching	93.9	91.7	**5.0**	**1.55**	3.0
NN-Patching$_{ms}$	94.5	**92.2**	6.2	2.02	2.4
Freezing	94.9	91.6	9.4	2.23	1.6
Base$_{update}$	**95.1**	90.3	13.4	4.05	**1.3**
	NIST$_{remap}$				
Baseline	11.5	11.9	—	5.0	5.0
NN-Patching	95.7	93.2	6.7	1.31	2.2
NN-Patching$_{ms}$	**97.1**	**94.7**	**6.0**	**1.09**	**1.0**
Freezing	94.8	85.5	16.2	3.15	3.2
Base$_{update}$	95.7	83.9	20.3	3.45	2.1
	NIST$_{transfer}$				
Baseline	0.0	0.0	—	4.91	5.0
NN-Patching	93.9	88.1	**16.8**	1.42	**1.0**
NN-Patching$_{ms}$	**94.1**	**88.4**	17.1	**1.55**	**1.0**
Freezing	91.4	80.4	31.9	3.05	3.3
Base$_{update}$	91.8	72.3	38.3	3.81	2.9

and 8. The algorithms are ordered by average rank. A connection between two algorithms means that these two are *not* significantly different. This type of figure is based on the one proposed by Demšar for the Nemenyi post-hoc test. As the figure indicates, *NN-Patching$_{ms}$* significantly differs in adaptation rank from any non-patching method. For the final rank, only *Base$_{update}$* and *Freezing* are in the same significance group.

7 Conclusion

In this paper, we proposed a version of patching that is specifically tailored to neural networks. We found that neural 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 was small (one hidden layer with 512 neurons), it was 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$_{update}$*) can leverage the higher network complexity compared to the other methods in some datasets, but the real advantage is limited. Estimating not

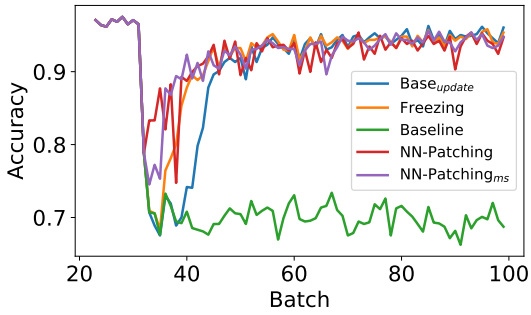

Figure 6: Results on the *NIST$_{appear}$* dataset as a data stream.

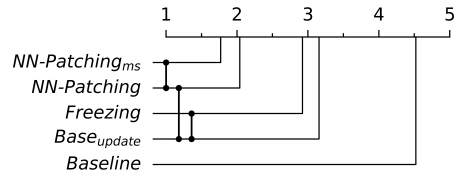

Figure 7: Wilcoxon signed-rank test on the adaptation rank.

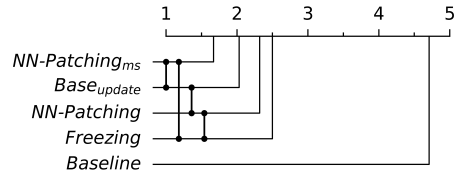

Figure 8: Wilcoxon signed-rank test on the final rank.

only the error of the base classifier, but also the current performance of the patch increases the accuracy, as the results of *NN-Patching$_{ms}$* indicate.

Because of its design, neural network patching is resilient against catastrophic forgetting, while at the same time allowing significantly faster adaptation for nonstationary 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. Opposed to many concept drift learning methods, it does not require explicit drift detection. This is handled implicitly by the error decision.

However, the method introduces new hyperparameters such as the patch architecture and engagement layer, which need to be tuned for the scenario at hand. Therefore, in future research we plan to establish heuristics to determine ideal engagement layers, and also application-specific architectures. This should improve the ease of use for neural network patching in general, and broaden its applicability.

References

Aggarwal, C. C. 2014. A Survey of Stream Classification Algorithms. In *Data Classification: Algorithms and Applications*. CRC Press. 245–273.

Ben-David, S.; Blitzer, J.; Crammer, K.; Kulesza, A.; Pereira, F.; and Vaughan, J. W. 2010. A Theory of Learning from Different Domains. *Machine Learning* 79(1-2):151–175.

Bengio, Y. 2012. Deep Learning of Representations for Unsupervised and Transfer Learning. In *JMLR: Workshop and Conference Proceedings*, volume 7, 1–20.

Bifet, A., and Gavaldà, R. 2009. Adaptive Learning from Evolving Data Streams. In *Advances in Intelligent Data Analysis VIII: 8th International Symposium on Intelligent Data Analysis – IDA'09*, 249–260.

Bifet, A.; Holmes, G.; Kirkby, R.; and Pfahringer, B. 2010. MOA Massive Online Analysis. *Journal of Machine Learning Research* 11:1601–1604.

Carpenter, G. A., and Grossberg, S. 1987. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine. *Computer Vision, Graphics and Image Processing* 37(1):54–115.

Cireşan, D. C.; Meier, U.; and Schmidhuber, J. 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*, 1–6.

Demšar, J. 2006. Statistical Comparisons of Classifiers Over Multiple Data Sets. *Journal of Machine Learning Research* 7:1–30.

French, R. 1999. Catastrophic Forgetting in Connectionist Networks. *Trends in Cognitive Sciences* 3(4):128–135.

Gama, J.; Žliobaitė, I.; Bifet, A.; Pechenizkiy, M.; and Bouchachia, A. 2014. A Survey on Concept Drift Adaptation. *ACM Computing Surveys* 46(4):44.

Ganin, Y., and Lempitsky, V. 2015. Unsupervised Domain Adaptation by Backpropagation. *Proceedings of the 2015 International Conference on Machine Learning – ICML'15* (i):1180–1189.

Geng, M.; Wang, Y.; Xiang, T.; and Tian, Y. 2016. Deep Transfer Learning for Person Re-identification. *CoRR* abs/1611.05244.

Gong, B.; Grauman, K.; and Sha, F. 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*, 222–230. PMLR.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition – CVPR'16*.

Hoffman, J.; Guadarrama, S.; Tzeng, E.; Hu, R.; Donahue, J.; Girshick, R.; Darrell, T.; and Saenko, K. 2014. LSDA: Large Scale Detection Through Adaptation.

Kauschke, S., and Fürnkranz, J. 2018. Batchwise Patching of Classifiers. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence – AAAI'18*.

Kirkpatrick, J.; Pascanu, R.; Rabinowitz, N.; Veness, J.; Desjardins, G.; Rusu, A. A.; Milan, K.; Quan, J.; Ramalho, T.; Grabska-Barwinska, A.; Hassabis, D.; Clopath, C.; Kumaran, D.; and Hadsell, R. 2017. Overcoming Catastrophic Forgetting in Neural Networks. *Proceedings of the National Academy of Sciences*.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature* 521(7553):436–444.

Long, M.; Cao, Y.; Wang, J.; and Jordan, M. I. 2015. Learning Transferable Features with Deep Adaptation Networks.

Oquab, M.; Bottou, L.; Laptev, I.; and Sivic, J. 2014. Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition – CVPR'14*, 1717–1724.

Pan, S. J., and Yang, Q. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22(10):1345–1359.

Polikar, R.; Udpa, L.; Udpa, S. S.; and Honavar, V. 2001. Learn++: An Incremental Learning Algorithm for Supervised Neural Networks. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 31(4):497–508.

Saenko, K.; Kulis, B.; Fritz, M.; and Darrell, T. 2010. Adapting visual category models to new domains. In *Proceedings of the 11th European Conference on Computer Vision – ECCV'10*, 213–226. Springer.

Schmidhuber, J. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61:85–117.

Torrey, L., and Shavlik, J. 2009. Transfer Learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, volume 1. Information Science Reference. 242–264.

Tzeng, E.; Hoffman, J.; Zhang, N.; Saenko, K.; and Darrell, T. 2014. Deep Domain Confusion: Maximizing for Domain Invariance. Technical report.

Webb, G. I.; Hyde, R.; Cao, H.; Nguyen, H. L.; and Petitjean, F. 2016. Characterizing Concept Drift. *Data Mining and Knowledge Discovery* 30(4):964–994.

Widmer, G., and Kubat, M. 1996. Learning in the Presence of Concept Drift and Hidden Contexts. *Machine Learning* 23(1):69–101.

Yosinski, J.; Clune, J.; Bengio, Y.; and Lipson, H. 2014. How Transferable are Features in Deep Neural Networks? In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc. 3320–3328.