Patching Deep Neural Networks for Nonstationary Environments

Sebastian Kauschke[1,2], David Lehmann and Johannes Fürnkranz[1]
[1] Knowledge Engineering Group, [2] Telecooperation Group
TU Darmstadt, Germany
{kauschke, fuernkranz}@ke.tu-darmstadt.de

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 learn the concept drift, neural network patching leverages the existing networks' inner layers as well as its output to learn an additive patch that enhances the classification. It learns (i) a predictor that estimates, if 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 help in aiding the classification for unseen or drastically altered inputs.

1 Introduction

Nowadays, neural networks are everywhere in machine learning research. Although they have been around since the 1940s, it took a long time to leverage their potential, mostly because of the computational complexity involved. This changed in the mid 2000s, when new methods and hardware emerged that allowed bigger networks to be trained faster, opening up new possibilities of application. Today, 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.

le

The main advantage of deep networks is the 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 therefore enabled neural networks to deal with more complex problems.

probably geschwafel from here on:

The current understanding of this is, that each layer of a network represents a different stage of abstraction from the input data, similar to how we believe the human brain processes information. Besides from the abstraction, specific units in artificial neural networks such as convolutional or long-short-term-memory units provide functionality that is beneficial to certain tasks, for example when dealing with image data. These units can also leverage a hierarchical abstraction. A typical network for image classification consists of multiple layers of convolutional units, each representing feature detectors with different grades of abstraction. Early layers detect simple structures like edges or corners, while later layers comprise more complex features such as eyes or ears, if the task is to recognize faces.

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

This could occur, for example, when we deal with an image classification task, and new classes need to be detected. Another 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. A final example to motivate the necessity of adaptation would be a personalization setting. A product is sold with a general prediction model that covers a wide variety of users. However, personalization would be beneficial to make it even more suitable for a specific user. A kind of adaptation that is difficult to manage with neural networks as underlying models.

In this paper, we propose 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 hyperparameters. Once such a network is established and properly trained, a prolonged use of it is usually appreciated. However, it is not guaranteed that a problem domain remains stationary.

We do not assume that it is feasible to re-train the model from the ground up with newly recorded data because of expenses and duration. However, adapting the already existing model to account for those changes is likely more efficient.

This is why we propose a new kind of adaptation, the so called **Neural Network Patching** (*NN-Patching*).

NN-Patching 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.

This paper is structured as follows. In Section 2 we discuss related work from the domain adaptive learning. In Section 3 we elaborate on the concept of the *Patching* family of adaptation algorithms, and go into detail about *NN-Patching* in Section 4. Furthermore, 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 general, forgetting is a natural and good process that happens gradually, and is also found in human brains. However, in artificial neural networks trained by backpropagation, learning new information leads to a reconfiguration of the weights in the network, hence catastrophically forgetting concepts that had been learned before. Although researchers have provided solutions to the problem via specialized network architectures (cf. Kirkpatrick et al. 2017), the problem is not generally solved.

Another issue of neural networks is, that continuous learning of new concepts over extended periods could require an increase in the size of the network for both the amount of layers as well as the number of neurons per layer. One approach to solve this is called neurogenesis, and it is also inspired by nature. Neurogenesis is the act of creating new neurons and integrating them into an existing network in order to account for novel information. Draelos et al. (2017) applied this principle to artificial neural networks, and managed to create a deep autoencoder that is able to adjust the number of neurons in a layer according to the complexity of the problem. They avoid catastrophic forgetting by executing a process called *intrinsic replay*, which leverages the properties of the autoencoder to approximately reproduce instances from previously learned classes, and re-train on those reproductions. Intrinsic replay is also a process that we believe to happen in the hippocampus of the human brain. Although they could achieve some promising results, they themselves claim that more research has to be conducted to find general solutions of how and when to apply intrinsic replay.

2.2 Concept Drift and Transfer Learning

Concept drift learning is concerned with a supervised scenario, where the observed data is a nonstationary, continuous stream and changes over time w.r.t. data distribution or the target variable. It is supervised in such a way, that the assumption is that the true labels will be available some time after the classification 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 being made in this research domain. Webb et al. provide a taxonomy that classifies concept drift in multiple dimensions such as duration, type of transition, re-occurrence, etc. (Webb et al. 2016). 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, in the domain of adaptive learning under concept drift, neural networks do not play a huge role as of today. Most of the state-of-the-art methods 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.

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 supervised scenarios such as we are interested in, the existing knowledge is combined with additional information from the target task—or even a third, related task—to help 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.)). Yosinski et al. (2014) elaborate on how deep convolutional networks learn layers of representations which tend to develop from being more general to being more specific w.r.t. the given task.

Our work is related to both concept drift and transfer learning in such a way, 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 a few annotated 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.

A general instance space D of instances x with labels $l(x)$ exists, as well as a black-box classifier C_0. C_0 is immutable and can classify the examples of D well. Caused by a non-stationary environment, a new batch $D_i, i > 0$ is received, for which C_0 makes imperfect predictions. The goal is to learn classifiers C_i that approximate l_i as close as possible in two steps:

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

(ii) A new classifier $C_{i,j}$ is learned on the misclassified instances, a so-called *patch*.

At classification time, a check occurs if the instance falls into the so called error region. If not, C_0 is used as usual. Otherwise, the classification is forwarded to the patch classifier, that was specifically trained with data from only that error region.

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 the patches 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 is kept for the 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 build the patches.

We want to leverage the strengths of the concept and advance it, so that it can also be applied to neural networks. More specifically, we want to improve it, that it can leverage the inner layers of the (deep) neural network, because in these layers abstractions and representations are stored that may be crucial to the classification output. 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,

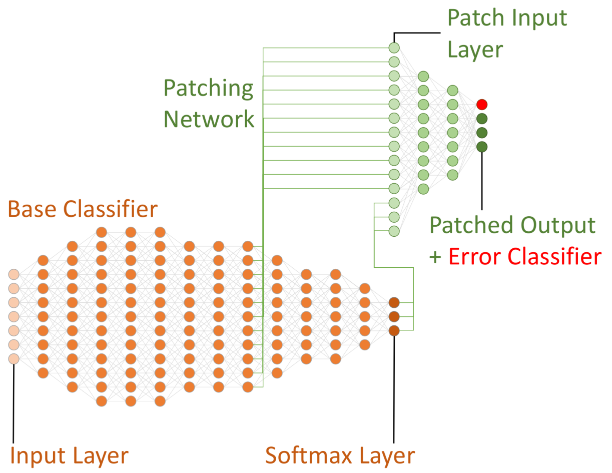

Figure 1: Patching of Feed-Forward Networks – Orange: the original network, Green: the patch

Cirean, Meier, and Schmidhuber (2012) leverage this behavior to achieve transfer to problems with higher complexity than the original problems the network was intended for.

In summary, we make three observations: (i) Neural Networks are useful towards adaptation tasks, based on 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. With this paper, we want to leverage the advantages in (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 adapted the *Patching*-procedure by Kauschke and Fürnkranz (2018) to work with neural networks. The idea is depicted in Figure 1. *Patching* consists of three steps:

1. **Learn a classifier E that determines where $\mathbf{C_0}$ errs.** In this step, when receiving a new batch of labeled data, the data is used to learn a classifier that estimates where $\mathbf{C_0}$ will misclassify instances.
2. **Learn a patch-network P.** The patch-network engages to one inner layer of $\mathbf{C_0}$ 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 $\mathbf{C_0}$ 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.

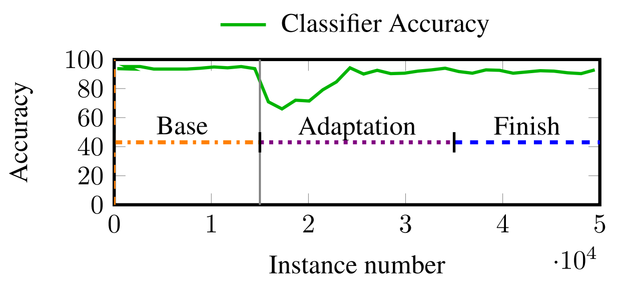

Figure 2: Example of a stream with concept drift. The drift is marked by the vertical line. Three phases can be identified.

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 they follow the argumentation of (Yosinski et al. 2014), where earlier layers in the network tend to be more general, and later layers are more specific. In our case this means, that for fully-connected networks we tend to use the more general features were more promising, whereas in convolutional networks we use the more specific features towards the end of the network.

5 Experiment 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.
- *Recovery Speed (R.Spd):* Number of instances that a classifier requires during the *Adaptation* phase to achieve 90% of its final accuracy.
- *Average Accuracy (Avg.Acc.): Average Accuracy from the change point until the end of the stream.*
- *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.

We ran each algorithm 10 times on every dataset with different random seeds for the network and calculated the average results for the metrics. The standard deviation of accuracy in those 10 runs was smaller than ????%.

ausrechne

5.2 The Evaluation Datasets

We will evaluate our findings on three datasets in 13 scenarios, each of them representing a different type of concept drift with varying severity up until a complete transfer of knowledge to an unknown problem.

achtung, a
passen

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 for the following variants.

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

$MNIST_{rotate}$ The digits in the dataset start to rotate randomly at instance #35k with increasing degree of rotation up to 180 degrees (at #65k).

$MNIST_{appear}$ The labels of the digits change during the stream so that classes 5-9 do not exist in the beginning, but only start to appear at the change point (mixed with 0-4).

$MNIST_{reappear}$ Digits 0-4 will disappear from the stream after one third, and then re-appear for the last third of the stream.

$MNIST_{remap}$ In the first half, only the digits 0-4 exist. The input images are then replaced by the digits 5-9 for the second half, but the 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 has the 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 of 100,000 instances for each variation.

$NIST_{flip}$ 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 number 40k with linearly increasing rotation up to 180 degrees for the last 10k instances.

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

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

NIST_{appear} The labels of the images change 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).

NIST$_{reappear}$ Instances with classes 0-9 will disappear from the stream after one third, and then re-appear for the last third of the stream.

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 half of the stream only consists of digits 0-9, while the second half has the characters A-Z.

5.3 Compared Methods

In this section we explain the variants of *NN-Patching*and other algorithms we compared against. We did not compare to the original *Patching*, since the *NIST*-dataset is too complex to be handled by the algorithm in reasonable time.

NN-Patching This is the variant as described in section 4.1. It contains the error decision as well as the patching network. Both are trained with new examples after each new batch of instances via backpropagation, and are initialized with random weights.

NN-Patching$_{noerr}$ This variant does not contain the error decision, it always relies on the learned patch for classification.

Freezing This is based on a usual transfer method. The layers up to and including the engagement layer are static, and the remaining layers are updated with new information when available. Whereas the patching network is randomly initialized, the transfer network is initialized with the initial weights from C_0.

Baseline The original network C_0, unchanged. This is used for comparison.

Table 1: Summary of the datasets used in the experiments

Dataset	Init	CPs	Total	Chunks
		MNIST Dataset		
MNIST$_{flip}$	20k	#70k	140k	100
MNIST$_{rotate}$	20k	#35k	70k	100
MNIST$_{appear}$	20k	#30k	50.2k	100
MNIST$_{reappear}$	20k	#25k	60k	100
MNIST$_{remap}$	20k	#35k	70k	100
MNIST$_{transfer}$	20k	#35k	70k	100
		NIST Dataset		
NIST$_{flip}$	30k	#40k	100k	100
NIST$_{rotate}$	30k	#40k	100k	100
NIST$_{appear}$	16.9k	#24.4k	84.4k	100
NIST$_{reappear}$	15k	#40k, #64.4k	100k	100
NIST$_{remap}$	?	#35k	70k	100
NIST$_{transfer}$	20k	#30k	80k	100

Baseline$_{update}$ This variant updates all weights in the network via backpropagation. It changes a significantly higher number of weights compared to *Freezing* or *NN-Patching*.

6 Experimental Results

7 Conclusion

Pros: Our methods allows application of methods from transfer learning to be used on concept drift scenarios and avoids the problems that usually occur then.

Acknowledgements. [redacted]

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.

Cirean, D. C.; Meier, U.; and Schmidhuber, J. 2012. Transfer Learning for Latin and Chinese Characters with Deep Neural Networks. *The 2012 International Joint Conference on Neural Networks (IJCNN)* 1–6.

Draelos, T. J.; Miner, N. E.; Lamb, C. C.; Cox, J. A.; Vineyard, C. M.; Carlson, K. D.; Severa, W. M.; James, C. D.; and Aimone, J. B. 2017. Neurogenesis Deep Learning: Extending Deep Networks to Accommodate New Classes. *Proceedings of the 2017 International Joint Conference on Neural Networks – IJCNN'17* 2017-May:526–533.

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 International Conference on Machine learning (ICML)* (i):1180–1189.

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

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.

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*.

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)*, 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.

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. 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?