Experiments with unsupervised domain adaptation using the gradient reversal layer

(Eksperymenty z nienadzorowaną adaptacją dziedzinową przy użyciu gradient reversal layer)

Wojciech Pratkowiecki

Praca inżynierska

Promotor: dr Jan Chorowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

15 lutego 2019

Abstract

The amount of data that is still unlabeled has been growing over recent years. With domain adaptation methods it is possible to get a model that classifies well these samples, if only a related and labeled set is available. While trying to get better and better classification results, many diverse and complex approaches were invented. This paper describes analysis and experiments aimed at understanding the problem of domain adaptation and coping with it using the *gradient reversal layer* and its modifications.

Ilość danych, które wciąż nie zostały opisane, znacznie wzrosła na przestrzeni ostatnich lat. Dzięki adaptacji dziedzinowej możemy otrzymać model, który poprawnie klasyfikuje takie dane, jeżeli tylko posiadamy zbiór powiązanych, opisanych przykładów. Aby uzyskać jak najlepsze wyniki klasyfikacji, wymyślono wiele różnorodnych i skomplikowanych metod. Ta praca opisuje analizy oraz eksperymenty, których celem jest zrozumienie samego problemu adaptacji dziedzinowej, jak i radzenia sobie z nią przy użyciu *gradient reversal layer* oraz jego modyfikacji.

Contents

Chapter 1

Introduction

Neural networks have found many applications in recent years due to their ability of learning some complex data dependencies. Top performance in various tasks have resulted in increased popularity of not only neural networks and deep learning but also of other machine learning (ML) methods. Different architectures designed for different tasks were invented. One of them is convolutional neural network (CNN) introduced by Yann LeCun [1], which caused huge improvement in the image classification field. However, obtaining top-performing models often requires a long training, with huge amount of labeled data. Such a complicated procedure yields a desired model, but it would not reach comparable results on different, even strongly related dataset. Moreover, if we have small amount of labeled samples, we need to find an alternative way to train the network. At this point domain adaptation is an attractive option.

Domain adaptation (DA) is a process of learning a distribution of samples from source domain and obtain a model that performs well on other, but related, target domain. For instance, if we have a huge (e.g. synthetic), labeled set of road signs images and a video from a ride through a city, we could get an effective model that classifies well all the signs from the video frames, if we train it properly with the labeled dataset. One of domain adaptation techniques is the gradient reversal layer (GRL) proposed by Yaroslav Ganin and Victor Lempitsky [2]. GRL is a simple, but effective, architectural trick, applied during backpropagation, designed especially for domain adaptation task. This thesis is a collection of research on the gradient reversal layer and its adjustments, with some new approaches and modifications that may not only improve its accuracy, but also help to get a detailed understanding of domain adaptation problem and the GRL behaviour.

Chapter 2

Background knowledge

2.1 Artificial neural network

Artificial neural network (ANN) is a machine learning technique inspired by the neural connections in human brain. While training the model we use a pre-defined architecture and a loss function. The architecture describes the transformation of the input data made by our model, its complexity and interpretation of the output. Each network contains neurons that are stacked in layers. Neurons are nothing but a weighted sum characterized by the neuron's weights and bias. A layer is built with many neurons and followed by non-linear activation function. Each layer transforms its input vector into an output vector. The first layer of the network is called the input layer, the last one is the output layer, while all the intermediate layers are called hidden layers. Figure 2.1 presents a sample ANN. The loss function is a way to measure and optimize the performance of the model, it is commonly the magnitude of its imprecision.

Successful learning process requires many examples of the problem we want the network to solve. At the beginning, we usually set the model's parameters (mostly denoted as Θ) to random values. The model transforms some of the input examples and we compute the loss function based on the output. As we want the model to perform as well as possible, our goal is to minimize the loss function. Therefore we use the backpropagation method - for each input sample we compute its gradient. The values of those gradients tell us, how should we tune model's parameters Θ to find the minimum of loss function. Therefore, we slightly change values of Θ in the direction of gradient, which should recall in a better performance of the model on the next examples. We repeat this process many times, improving our network during the whole learning process.

This iterative training approach is a simplified description of stochastic gradient descent (SGD) - one of the most popular and effective ANN optimization method. As our model may be defined by millions of parameters, computational cost of thousands

of training iterations is a huge drawback of neural networks. However, great growth of computer's computation power with GPU acceleration let us find solutions for some really complex problems within a decent time. Therefore neural networks are nowadays widely used for many tasks. One of them is image classification - finding for a given input image the most likely label from a fixed label set.

2.2 Convolutional neural networks

Applying neural networks to image processing became very popular. To keep reaching better results in tasks, like image classification, some new inventions were needed. The breakthrough came with convolutional neural networks (CNN). CNN, just like classical neural network, is built with neurons stacked in layers. First layers of CNN are convolution layers. Each of them is a set of learnable, small size (like 5×5) filters followed by activation function. Each filter processes an input image step by step and produces a transformed, usually smaller, one. Therefore, after each layer we get a new set of images obtained from all pictures from previous layer. A big advantage of this approach is a much smaller number of learnable parameters - if the input image has size 1000×1000 it would require 1000000 parameters for a single neuron in a fully connected layer, while convolution layer processes it with its filter, that has 25 parameters, if the size of the filter is 5×5. After a convolution layer we usually use pooling - we divide the picture into cells (e.g. of size 2×2) and we replace the whole cell with its maximum value (in case of the maximum pooling). This operation reduces the size of the image internal representation, and force the network to focus on important patterns. Figure 2.2 presents an example of CNN architecture.

The convolutional part of the model learns high-level features or descriptors of locations in the input image. Moreover it makes the network invariant for small translation of object over the image. Convolutional layers produce a vector representation of the input image. It is then fed into classical, fully-connected layers that return the model's output. If our task is image classification the output is a vector of classes probabilities typically.

CNNs caused huge improvements in both accuracy and speed of image classification. Nowadays top-performing architectures in many image processing problems are built with convolutions. Convolutional layers also found application in some non-picture field like some natural language processing tasks or speech generation.

2.3 Digit classification with CNN

MNIST dataset [5] is a collection of black and white photos of handwritten digits. Every picture's resolution is 28×28 pixels, each one is an integer in the range $[0, 255]$, that describes the brightness of the pixel. All the samples are provided with ground

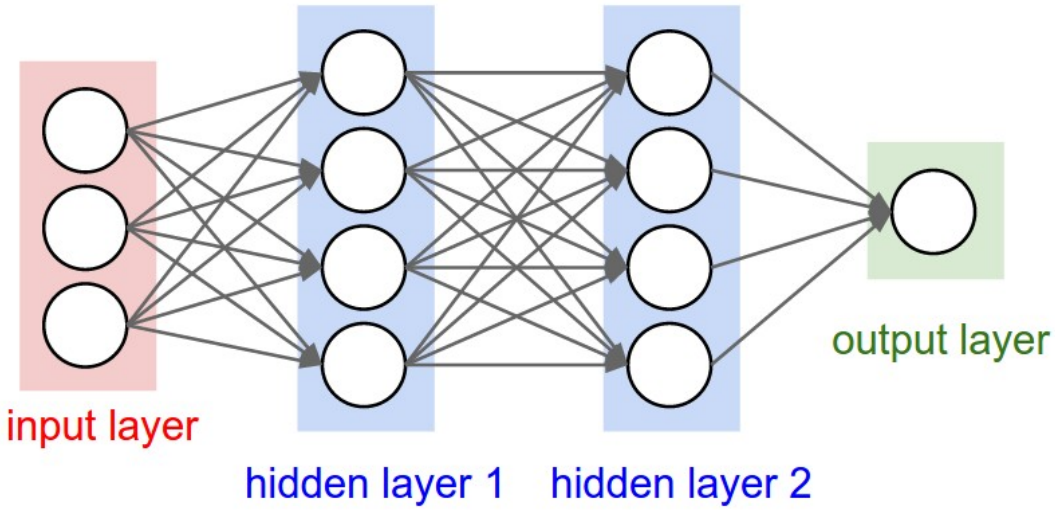


Figure 2.1: Example of neural network architecture. Picture from Stanford course website [3]

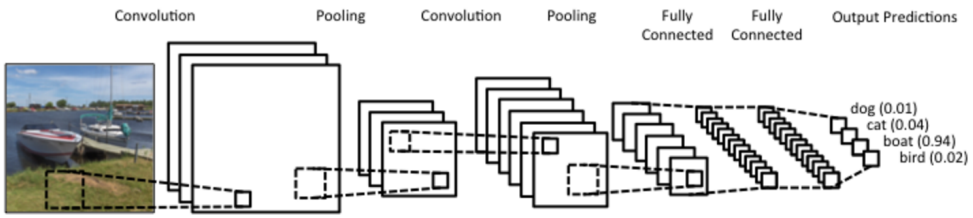


Figure 2.2: Example of convolutional neural network architecture. Picture from [4]

truth labels. Classification of digit images is therefore a task of processing the input image that dimensionality is 28×28, and assigning probabilities for each of 0-9 digits. The model prediction is the digit with highest probability. Even very ordinary convolutional architecture reaches over 99% accuracy on this dataset. A bit more challenging modification of MNIST is the MNIST-M set [6], obtained by blending example from MNIST with some color photos from BSDS500. There is probably no difference in difficulty of classification MNIST and MNIST-M digits for human, but from neural network point of view, the colorful set is much more complex. Nevertheless, obtaining top-performing model is relatively easy.

There are a lot of similar digit recognition datasets, with many that are much more complex than MNIST and its modifications. The Street View House Number (SVHN) is a good example of such collection. SVHN samples are obtained from Google Street View photos, therefore they are colorful, with diverse backgrounds, pictures are often blurry and a single image may present few digits, while its label is the one in the centre of the photo. Due to all those properties of SVHN, model that classifies well its samples must be complex and longer trained.

Even though digit classification seems not to be the most important and en-

Figure 2.3: Samples from MNIST, MNIST-M and SVHN digit images datasets, best viewed in color

tertaining problem that CNN may solve, it is still one of the most popular field of measuring a model's performance and yields experimental results with quick turn-around time. Figure 2.3 presents some examples from mentioned datasets - MNIST, MNIST-M and SVHN.

2.4 Domain adaptation

Let's suppose that we have a novel, large set of digit images, for instance photos of the back of each European footballer's jersey, preprocessed in a way, that every image contains one digit - figure 2.4 presents some possible samples. Our goal is to obtain a model that classifies digits from football kits. Unfortunately we don't have any labels for these pictures, so we are unable to go a straightforward way and train a CNN adjusted to the image set, as we cannot determinate if the model's prediction is right. In this kind of situation domain adaptation is a solution.

Domain adaptation is a specific training process, when we teach a model with examples from a source domain, but our goal is to make it perform well on different, but related, target domain. Over recent years many architectures have been proposed to reach as high target domain accuracy as possible. DA is needed when samples from target distribution are unlabeled (unsupervised domain adaptation) or we have just few labeled examples (semi-supervised domain adaptation). A model should find a mapping between domains, which would allow source domain classifier perform well on test (target domain) examples.

In our case of football jerseys, when the set is unlabeled at all, we have an unsupervised DA problem. Target domain is the distribution of jersey's photos. Now we should try to obtain a model learned with labeled examples from other dataset, like SVHN, and apply some architectural solution that would let the model classify well digits from football kits. Dozens of such tricks have been proposed in recent years. One of them is gradient reversal layer.

Figure 2.4: Football jersey dataset

2.5 Domain adaptation using GRL

Gradient reversal layer was introduced by Yaroslav Ganin and Victor Lempitsky in the paper titled "Unsupervised Domain Adaptation by Backpropagation" in 2015 [2]. The authors describe domain adaptation as a problem of finding common mapping for source and target distribution, that would transform samples to some kind of representation, that does not contain any information about the domain of mapped input. Therefore, the classifier basing on a transformed sample, would have similar accuracy for both distributions. In perfect scenario the mapping does not influence the label predictor's accuracy, but is so invariant w.r.t. domain shift, that we cannot obtain a well performing distribution classifier.

To obtain the desired representation, the model is divided into three parts - a *feature extractor*, that is denoted as G_f with its parameters Θ_f. G_f maps the input to a D-dimensional vector $f \in \mathbb{R}^D$. This representation is then used by two other components of the model - a *class predictor* G_y with parameters Θ_y, that classifies the feature vector, and a *domain predictor* G_d with its parameters denoted as Θ_d, which task is to predict the domain of the vector f.

All these three components are independent neural networks that construct our architecture. While training the model we aim to force G_f to yield domain invariant feature vectors f. Therefore, GRL is placed between the feature extractor G_f and the domain predictor G_d. Its task is to change the G_f to **maximize** the loss of the domain predictor, while G_d itself tries to minimize its own loss function to make better predictions. If the domain predictor, that wants to does its best, is unable to to predict from which domain a sample has come from, then f is getting more domain invariant. In perfect scenario GRL has such an impact on Θ_f, that G_d has the same accuracy as a random model. Concurrently G_y tries to predict the class of each sample and we try to minimize its loss function. Class predictor's gradient is used to tune Θ_y and then passed to the feature extractor. Therefore the feature extractor parameters Θ_f are tuned to minimize the loss of G_y and maximize the loss

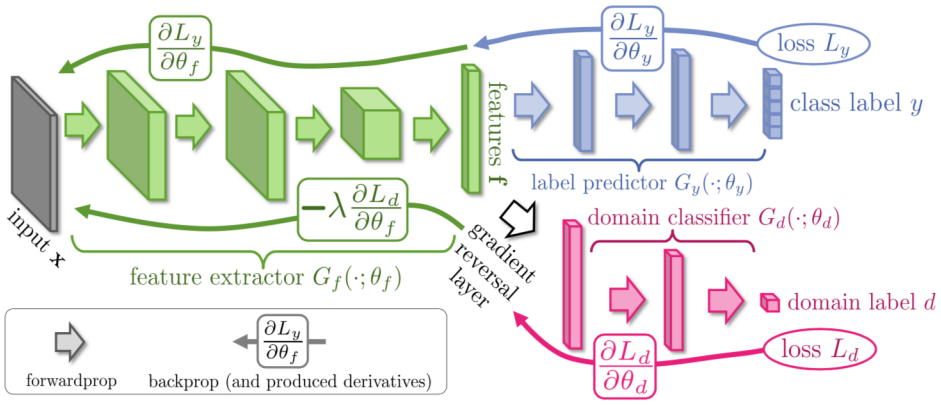


Figure 2.5: Architecture with GRL in G_d proposed by Ganin and Lempitsky - green part is a *feature extractor* G_f that transforms input sample x to feature vector f. Afterwards it is classified by *label predictor* G_y (blue) and *domain predictor* G_d (pink). Picture from the author's paper [2].

of G_d. Figure 2.5 presents the described architecture.

As the basic task of our model is to classify the digits, the parameters are tuned to make label predictor as accurate as possible. Therefore, we tune parameters Θ_y and Θ_f to minimize the loss of class predictor $L_y(G_y(G_f(x)))$. The domain predictor G_d is tuning its parameters Θ_d to minimize its own loss $L_d(G_d(G_f(x)))$, but the feature extractor, as previously mentioned, tries to remove the distribution information by tuning its parameters Θ_f to maximize the L_d value. Then, to find the proper parameters we perform the optimization updates:

- Θ_y w.r.t. the direction of $-\frac{\partial L_y}{\partial \Theta_y}$ and Θ_f w.r.t. the direction of $-\frac{\partial L_y}{\partial \Theta_f}$
- Θ_d w.r.t. the direction of $-\frac{\partial L_d}{\partial \Theta_d}$
- Θ_f w.r.t. the direction of $\frac{\partial L_d}{\partial \Theta_f}$ (no $-$, we maximize the L_d)

To perform these optimization steps we use the gradient reversal layer, as it enables us to execute this kind of updates. Gradient reversal layer itself is a simple trick used during one forward and backward pass through the network. It is a trival layer in the network's architecture, that leaves the input unchanged during forward propagation, but multiplies the gradient by negative scalar λ during backpropagation. We place it at the beginning of the domain predictor G_d. Therefore, the value of the gradient passed to the feature extractor is $-\lambda\frac{\partial L_d}{\partial \Theta_f}$. The λ coefficient grows non-linearly for 0 to 1 over the training, as we want the feature extractor to focus on good representation for class predictor firstly and making the vector f domain-invariant when G_y performs well already. The optimization may be implemented then as a stochastic gradient descent update with loss $L_y + \lambda \cdot L_d$.

GRL is therefore a simple modification of model's architecture. Implementing it with some machine learning framework requires a little effort. If a model with GRL

were creating absolutely domain-invariant representation of data the accuracy for test sets from source and target domain should be almost equal, as the only features left after G_f transformation would be associated with the sample's class. During tests we don't use domain predictor, our final model is build with feature extractor and class predictor, G_d is used only to tune parameters of the network during the training.

2.6 Conceptors

Domain adaptation is a very complex problem, therefore understanding it could be crucial to find some satisfying solutions. As previously mentioned the representation of input sample x after mapping by feature extractor G_f is a feature vector $f = G_f(x; \Theta_f) \in \mathbb{R}^D$. It may be also considered as a point in $\mathbb{R}^D$. All the samples from source domain S and target domain T form a point clouds in $\mathbb{R}^D$ space that can be denoted as F_S and F_T respectively:

$$F_S = \{G_f(x_s; \Theta_f) : x_s \in S\}$$
$$F_T = \{G_f(x_t; \Theta_f) : x_t \in T\}$$

If the transformed samples from both domain are considered as points, the feature extractor should find a mapping, that produces as similar point clouds for distributions as possible. Then, the domain classifier could not predict well, if point from $\mathbb{R}^D$ with given coordinates belongs to either point cloud F_S or F_T. Obtaining such representation would be a half way to success, as perfect mapping should also place the samples from a single class in the same region for both domains. The label predictor G_y would then have to just find a division of the space into the regions for all classes.

It is not a simple task to find a description of points cloud in $\mathbb{R}^D$ that would be understandable for a human, if the dimension D is large. Nevertheless, it is possible to approximate these points somehow. One approach is to find an ellipsoid in $\mathbb{R}^D$ that captures the majority of the points. This idea was used by Herbert Jaeger to introduce the concpetors [7]. Such ellipsoid is then used to transform a new point into the cloud of known points. The conceptor is defined as a matrix C that determines an ellipsoid that approximates a given point cloud and is then normalized, so it lies inside the unit sphere. Figure 2.6 presents a simple visualization of this idea. Singular vectors of matrix C are the principal axes of the described ellipsoid, while singular values of C are their lengths. The conceptor matrix C is obtained with correlation matrix $R = \mathbb{E}_x[xx^T] \in \mathbb{R}^{N \times N}$ of samples $x \in \mathbb{R}^N$ and the aperture parameter α, as:

$$C = R(R + \alpha^{-2} I)^{-1}$$

The conceptors can be used to move a point inside the ellipsoid. The transformation of given point x into the conceptor is made by multiplying it by the matrix

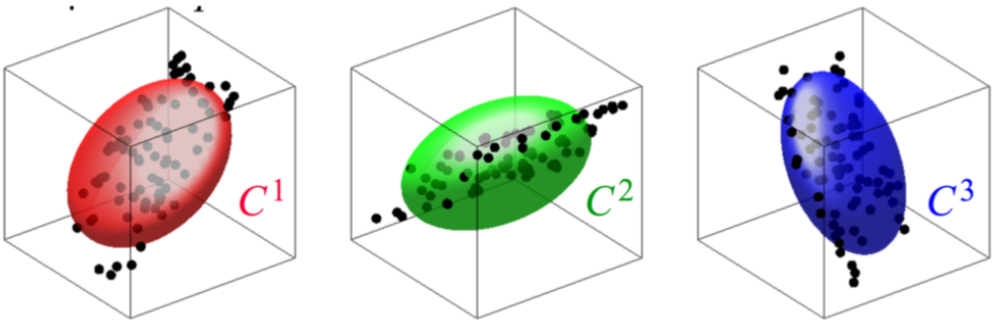


Figure 2.6: Point clouds (black) and conceptors approximating them. Best viewed in color. Picture from "Conceptors: an easy introduction" by Herbert Jaeger [8].

C. If x lies inside the ellipsoid that is described by the conceptor, the multiplication should act like identity and remain x unchanged. If x is a point from the outside of the ellipsoid then C should move it inside.

As it is possible to represent the high dimensional point cloud with well defined solid shape given by the conceptor, the analysis of F_S and F_T could be done with operations on these ellipsoids. It may be a good approach to understand the domain adaptation and the influence of GRL.

Chapter 3

Experiments

3.1 Paper reproduction

First stage of all the research made within this thesis was the paper reproduction. All implementations were made with PyTorch [9] - deep learning python framework developed by Facebook. All computations were run on Google Colab [10] notebook.

Few architectural changes in the network proposed by Ganin and Lempitsky were made, like replacing ReLU activation function with leaky-ReLU [11] and using dropout [12] during learning. When training on SVHN dataset, much more complex architecture was needed. The feature extractor was built with 5 convolutional layers, each one using batch normalization [13] and the leaky-ReLU followed by max pooling and dropout during training process. Such a complex architecture allows us to get satisfying results 10 times faster than with model proposed by Ganin and Lempitsky[1].

During the experiments two pairs of domains were used - MNIST as source and MNIST-M as target distribution, denoted as $\overline{\text{MNIST-MNIST-M}}$ and SVHN and MNIST as source and target domain respectively, which case will be denoted as $\overline{\text{SVHN-MNIST}}$.

In $\overline{\text{MNIST-MNIST-M}}$ case, the accuracy close to the result from paper (81.49%) was reached after 30 epochs of training, but already after 16 epochs the obtained results were satisfying (see table 3.1). Therefore, a long learning process is not needed to verify if the training may be successful.

The relatedness between MNIST and MNIST-M may be worrying, therefore the experiment should be verified with some other datasets, therefore it was repeated in $\overline{\text{SVHN-MNIST}}$ case. The obtained model classified well 92.42% of samples from SVHN test set and 77.1% images from MNIST test set, slightly higher than best

[1] During authors' research the classification error on SVHN dataset was high for first 150 training epochs. With proposed, complex feature extractor, high accuracy was achieved after about 12 epochs.

Source domain: Target domain:	MNIST MNIST-M	SVHN MNIST
Accuracy on source domain	99.12%	92.42%
Accuracy on target domain (training without G_d with GRL)	51.06%	68.5%
Accuracy on target domain (training with G_d with GRL)	80.57%	77.1%

Table 3.1: Classification accuracy for tested domains. Target domain test set was classified with traditional model, as well as with model that uses GRL. The improvement is significant.

model obtained by the authors (71.07%). Afterwards, the model built without a domain predictor with GRL was trained. It managed to recognize well 68.5% of target domain samples. Such an improvement (Ganin and Lempitsky reached 59.19%) is the result of the complex architecture. Table 3.1 shows the composed results of classifying test sets for different domains.

Repeating the experiment with $\overline{\text{SVHN-MNIST}}$ shows, that strong correlation between MNIST and MNIST-M was not the only reason of GRL partial success in domain adaptation. The improvement of the performance after applying GRL is then really caused by the ability of the network to focus on the semantic of the image - a digit.

3.2 Domain information remaining after GRL training

After reproducing the paper, the next step was to verify if the vector $f = G_f(x; \Theta_f)$ really was domain invariant, as claimed by the authors [2]. When model's parameters were tuned, the feature extractor G_f was frozen and a new domain predictor was constructed, this time without the gradient reversal layer. So far it was known that G_f mapping was incomprehensible for a single domain predictor G_d with parameters Θ_d, as it's accuracy was just 68%. The input for new classifier was a vector $f = G_f(x; \Theta_f)$ for all samples x from both source and target domain, and the goal was to predict the distribution of x. If the feature extractor mapping really was domain invariant we should have not been able to obtain a model with much better performance.

It could have been expected that G_f mapping is unclear for just one, used during the learning process, domain predictor. After a short training, the new distribution classifier reached over 95% accuracy in both $\overline{\text{MNIST-MNIST-M}}$ and $\overline{\text{SVHN-MNIST}}$ cases. Therefore we can certainly say, that representation f of input image x still contains a lot of information about the sample's domain. Nevertheless ability to outsmart a single distribution predictor significantly improves the model's performance.

3.3 Adding GRL to well performing domain predictor

State of the art domain adaptation approaches reach much higher results than a model with GRL. Therefore looking for some improvement is not groundless. With MNIST as source domain and MNIST-M as target distribution, over 90% accuracy on MNIST-M test set has been reached [14]. It should be mentioned that top-performing domain adaptation models apply some data augmentation techniques, which make MNIST more similar to MNIST-M. One of such transformation is adding to the dataset a color-negative version of each sample (white background, black digit) [15]. During all experiments made within this thesis no augmentation was made.

As verified before, feature extractor produces representation that is indistinguishable for a single domain predictor. After the training we can easily obtain a model G'_d that successfully predicts the distribution of the sample. It can be checked then, how difficult and effective would be changing the transformation made by G_f, so it become unclear for the well performing domain predictor G'_d. The GRL is then added at the beginning of the G'_d and the whole model is trained once again, but now the components are G_f, G_y and G'_d.

After just few epochs accuracy of G'_d has decreased significantly, as G_f mapping adjusted to the new architecture. However, classifying samples from target distribution has not got any better. Moreover, during the training G_d is not used anymore, so we can't say, that G_f produces the representation that is unclear for two domain predictors. Anyway, the gradient reversal layer is able to easily outsmart even a very good classifier.

3.4 Filtering out the domain information

As the domain information remains in the feature vector after the training with GRL, the label classifier prediction is still dependent on the domain of the input image. Checking the score of a new distribution classifier on the frozen output of the class predictor's first layer would show the magnitude of this dependency.

After the training in $\overline{\text{MNIST-MNIST-M}}$ case, the accuracy of such classifier was 88%, then a small decay of domain information happened, which improved the performance on target domain. Therefore, filtering out the distribution features may cause even better results. To remove more domain information from the output of G_y first layer, the new domain predictor with GRL was "plugged" there. The weights of the feature extractor G_f were frozen, as we only want to change the mapping made by the first layer of class predictor.

When the domain information was measured once again after the training, the new domain classifier reached only 74.75%. Also, the accuracy on the target domain test set slightly increased. Therefore, the GRL made the output of G_y first layer

Accuracy of \ model's layer	G_f output	G_y 1st layer	G_y 2nd layer	G_y 3rd layer
Domain classifier on output	99.98%	87.86%	75.13%	69.47%
Domain classifier after adding GRL	95.95%	74.75%	68.19%	68.6%
Target domain test set classification (after applying domain predictor with GRL)	78.6%	78.74%	79.06%	79.26%

Table 3.2: Filtering out the domain information - for each tested layer the distribution information is measured, then a domain predictor with GRL is applied to filter out the domain features. After the training, once again the distribution information is measured and the layer's weights are frozen, before handling the following one.

more domain invariant. This observation induces to check the result of filtering out the distribution information in deeper layers of the model.

Figure 3.1 presents the described learning process. The first layer of G_y was frozen and GRL was plugged to the following layer. The process was repeated until the output layer of G_y, as its values are probabilities of each digit, so measuring the domain information or applying GRL to the model's output would be pointless. Table 3.2 shows the classifiers results on each stage. A special, deeper architecture of the class predictor was used during this experiment, what allowed us to monitor the vanishing of the domain information more precisely.

The experiment was repeated on a smaller model with just a single hidden layer, as it was able to reach better accuracy on the test set than the deeper network. Table 3.3 presents detailed results. Obtained model's score is better than the score of the basic model, so a small improvement was made.

Accuracy of \ model's layer	G_f output	G_y 1st layer	G_y 2nd layer
Domain classifier on output	99.99%	89.69%	77.43%
Domain classifier after adding GRL	93.95%	78.28%	71.83%
Target domain test set classification (after applying domain predictor with GRL)	76.91%	80.6%	80.7%

Table 3.3: Filtering out the domain information for G_y with a single hidden layer. The information about the sample distribution is progressively removed by domain predictors with GRL.

Adding a domain classifier with GRL definitely caused the decay of distribution information in the layers of class predictor. Therefore, the output of consecutive G_y layers is more domain invariant. We can also notice, that in deeper layers of the model, the decrease of domain predictor accuracy was small. It seems like applying an extra domain classifier is more reasonable on early layers of G_y. It may be helpful observation, if the learning process lasts for a long time. To sum up, filtering out

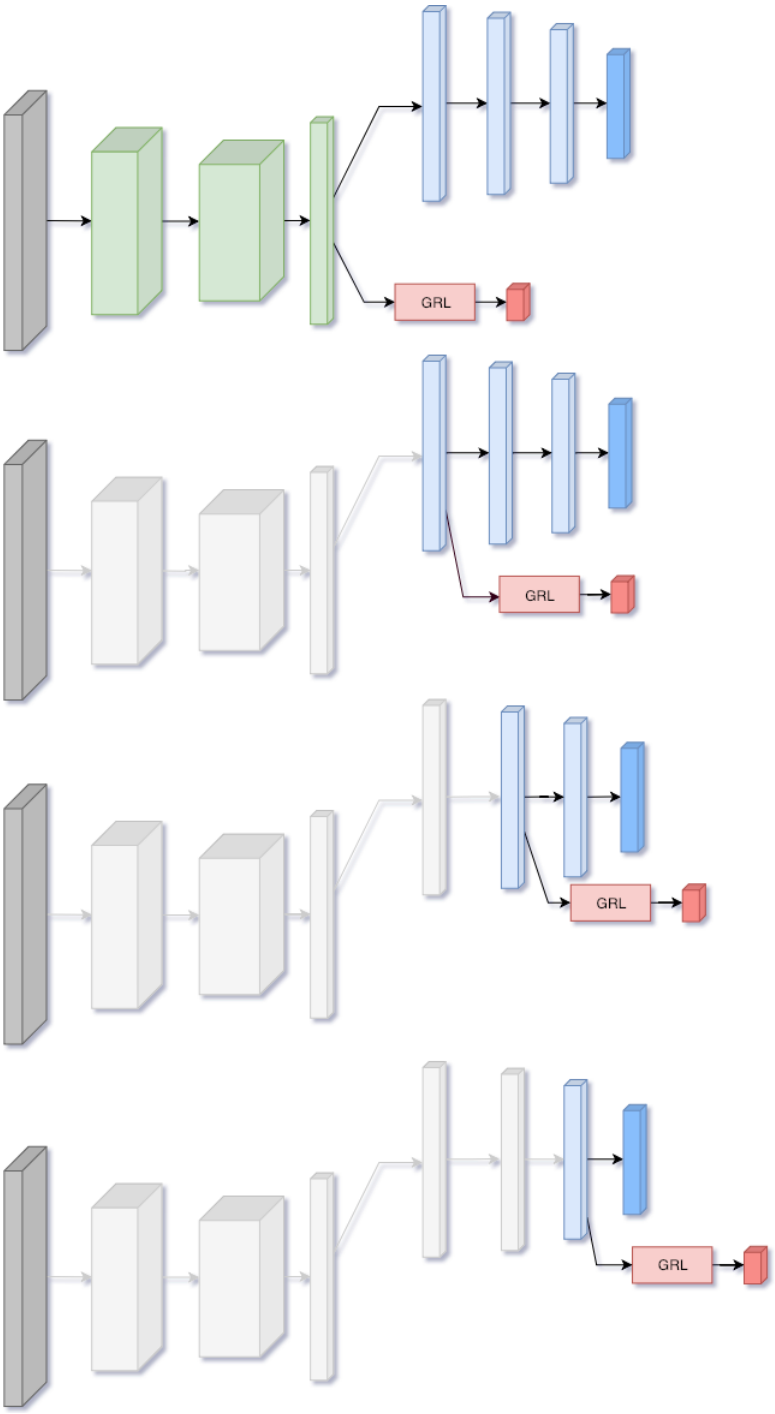


Figure 3.1: Filtering out the domain information process. Firstly the domain predictor is plugged to the output of G_f. Then G_f is frozen and a model with GRL is applied to the consecutive layers of class predictor. All previous layers are frozen. Green blocks represent G_f, blue ones G_y while pink are domain predictor G_d. Grey blocks are frozen layers. Best viewed in color.

the domain features from more than one layer in the whole model is a successful modification.

3.5 Hyperparameters impact

3.5.1 Training continuation with increased λ

During the training process the λ coefficient grows from 0 to 1, as the feature extractor's output should be firstly adjusted to a label predictor, and then made more and more domain invariant. When the final model is obtained, parameters Θ_y are well tuned, as the accuracy on source domain is high, the only limitation is not domain invariant enough transformation made by G_f. Therefore we could try to increase the G_d impact, when G_y is already well trained. The learning process was repeated with a single modification - λ had fixed value, that was greater than 1. In first attempt λ was set to 3. After a short training on $\overline{\text{MNIST-MNIST-M}}$ the model had 79.2% accuracy on MNIST-M test set. Then the λ was set to 3 for the consecutive epochs and the result increased to 80.6%. The result of basic model trained for the same time is very similar, however it seems reasonable to apply this kind of continuation, when larger epochs number does not help.

Setting λ parameter to some value grater then 1 is enhancing the impact of domain predictor on feature extractor mapping. If the coefficient would be too large, it may lead to deprecation of the model. During experiments, when $\lambda = 6$ a pre-trained model gets chaotic and does not reach the level from first training. Therefore we should worry about picking the right value for λ coefficient during the training continuation.

3.5.2 Tuning feature vector size

As previously verified, the feature vector f is not domain shift invariant. During all the experiments made, f was 320-dimensional. Maybe the size is so large, that G_y uses only its fraction, and the rest of the space is filled with valuable information for a distribution predictor. Therefore a smaller size of G_f output should be considered.

The intuition is that there may be an optimal size of the feature extractor output. It would be large enough for G_y to predict the input's label, but the space left would be too little to be sufficient for any domain predictor. The approach with a domain classifier without GRL should also be tried. If two models would compete for the space of a small feature vector f, the feature extractor could be forced to separate the data valuable for each of classifiers.

So far feature extractor was constructed with only convolution layers. During these experiments it was extended by single fully connected layer that was mapping

the output from $\mathbb{R}^{320}$ to $\mathbb{R}^{D'}$, where $D' = 280, 240, 200, 180, 160, 120$. Then the smaller feature vector was classified by class predictor G_y and domain predictor G_d with GRL.

When G_f tries to store all the features in a smaller vector, the training process gets noisy. If the dimensionality D' is too little, the value of the loss function may become NaN. As G_f returns just few values, both predictors compete for significant changes in most of them, what lead to the failure of the training.

After each architecture's training on $\overline{\text{MNIST-MNIST-M}}$ the results were not special at all. The more complex the model was, the higher accuracy it reached. Moreover, adding a fully connected layer caused a noticeable decrease of the performance. Figure 3.2 presents the accuracy of each learned model.

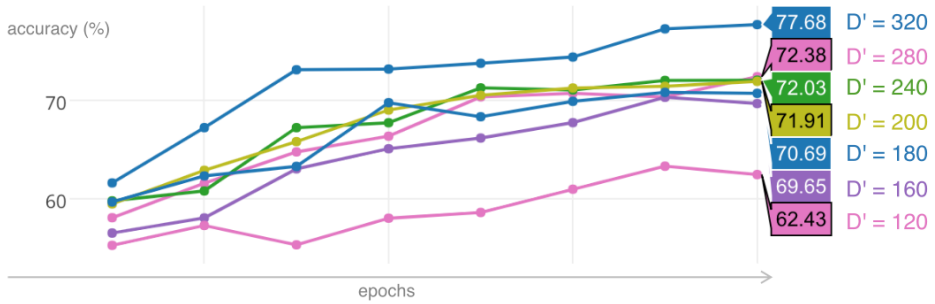


Figure 3.2: Accuracy of models with different feature extractor output size over the epochs. Model with feature extractor output size $D' = 320$ has no additional fully connected layer. Other architectures are build with feature extractor extended with single fully connected layer, which return a feature vector $f \in \mathbb{R}^{D'}$. Labels present results on MNIST-M test set.

In the second tested approach, when G_d did not have a gradient reversal layer, the feature extractor tried to return an output clear for both label and domain predictors. If it was too small, the G_f could separate the digit and domain features, as they would not fit in together.

The results of such multitasking approach were terrible. The classifiers performed well within their tasks, but accuracy on MNIST-M test-set was under 20%, more than twice less than a simple CNN trained on MNIST. Figure 3.3 presents the results for different dimensionality of feature vector $f \in \mathbb{R}^{D'}$.

Limiting the feature vector size did not force the feature extractor to separate the data. It seems that domain predictor's impact on feature extractor is so high, that the G_f mapping becomes highly correlated with the input distribution. Therefore, when we transform a sample from the target domain, its representation is meaningless. Straightforward using of a common feature extractor is not a good idea then.

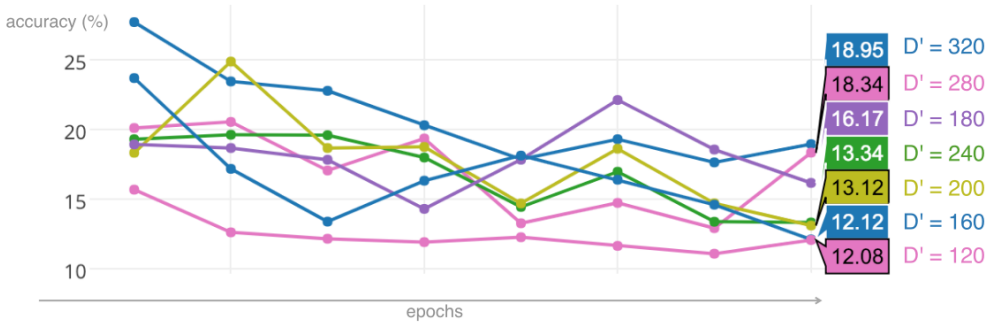


Figure 3.3: Accuracy of models with different feature extractor output size over the epochs. During the training domain predictor did not contain GRL. This kind of training led to the huge decrease of model's accuracy.

3.6 Impact of gradient modifications

Since domain information remains in feature vector, the real effect of GRL should be considered. Adding random noise is one of the gradient modifications that regularize the network [16]. Therefore, GRL itself could also be some kind of a regularization. With the following experiments we try to check, how complex the effect of GRL is, by comparing its results with some other gradient modifications.

Each of the following modifications were placed at the beginning of the domain predictor and used only during backpropagation pass, just like the transformation made by the gradient reversal layer. Experiments were made on $\overline{\text{MNIST-MNIST-M}}$ domains.

3.6.1 Random gradient replacement

First of tested gradient modifications was replacing the real gradient with some random numbers from range $(-1, 1)$ multiplied by λ parameter. If the results of obtained model were similar to the accuracy of the network with GRL, the gradient reversal layer could be seen as a very simple regularization, that adds some random noise to the gradient of the classifier. It would be equal then to use the domain predictor with GRL, as to apply the approach from mentioned paper [16].

At the very beginning of the training, the loss function became NaN. It could be caused by the magnitude of the random number, therefore the range of the random value was limited to $(-\frac{1}{10}, \frac{1}{10})$, but the failure repeated. It was needed to limit the new gradient value to $(-\frac{1}{10000}, \frac{1}{10000})$ to finish the learning process. However, with such a low value, the G_d impact on the feature extractor was insignificant. The obtained model was performing like a basic digit classifier. The G_d reached very high accuracy in domain predicting, as there was nothing that would disturb it, its gradient was ignored by the feature extractor.

Replacing gradient by some random values does not seem very effective. Firstly, the random value must be small, otherwise, the training fails. Secondly there is not any improvement of the model. Performance of the obtained model is far from the network with GRL, therefore the gradient reversal layer is something more complex than such a simple regularization.

3.6.2 Random sign inversion

As the GRL usage is not a simple added noise, the values of domain predictor's gradient are crucial to tune the parameters. We can test then, if it is necessary to change the sign of every value. The new modification could be then multiplying only fraction of gradient's value by $-\lambda$, and the rest by λ.

After the training the model's performance was very close to the network with GRL. Accuracy on source domain reached 98.58%, and samples from target distribution were classified well in 77.76%. The domain predictor, with first layer that was modifying the gradient in described way, managed to predict well the distribution of 69.58% samples. Table 3.4 presents the results of this and following tested modifications.

Multiplying just a fraction of gradient's values by a negative number causes comparable improvement to the GRL, which inverts the sign of every value. Therefore the advance of GRL may be caused by just changing the direction of the optimization step. It is not necessary then to follow exactly the opposite direction to the G_d loss gradient.

3.6.3 Random gradient multiplying

We already know that it is not needfully to multiply each of gradient values by a negative λ coefficient. Now we can check how important is to use equal λ for all parameters and keep the relative magnitude of these values. Therefore, the next gradient transformation could be multiplying it by some random numbers from range $(-1, 1)$. The increasing λ coefficient is not needed anymore. With such transformation not only a fraction of values change their sign, but their significance may be changed.

The gradient value is considerably different from the origin after the modification, therefore this model could perform comparably to the random gradient replacement. However, the results are even closer to the GRL model, than in the random sign inversion experiment. The accuracy on source domain was 98.75% and the score on target distribution was 79.12%. This result is very similar to the accuracy of the model with GRL trained for the same time. Table 3.4 presents the result in comparison to others gradient modifications.

	GRL	Random sign inversion	Random multiplication	Gradient Inverse
Accuracy on source domain	99.12%	98.58%	98.75%	98.38%
Accuracy of domain classifier	67.29%	70.49%	68.27%	59.28%
Accuracy on target domain	80.57%	77.76%	79.12%	78.87%

Table 3.4: Results of applying different gradient modifications during backpropagation in $\overline{\text{MNIST-MNIST-M}}$ case. The scores are very close to the accuracy of the model with GRL, therefore the advance of the GRL may be just avoiding the parameters update in the direction of the G_d loss minimum. The low accuracy of domain predictor when gradient inverse layer was used shows that this method may produce much more domain invariant feature vectors.

Inverting the sign of a fraction of gradient's values and changing their relative magnitude is a modification with very comparable effect to the GRL. Therefore, it is enough to modify the gradient much more randomly. With this transformation, the optimization step has its size and direction significantly changed. The advantage of the GRL may be then just avoiding the step in the direction of the domain predictor loss gradient.

3.6.4 Gradient inversion layer

The gradient not only points at the direction of the function's minimum, but also determines the magnitude of each parameter's impact on the final result, as gradient is a multi-variable derivative. Therefore, when the goal is to set such parameters Θ_f, that the transformation made by G_f is absolutely meaningless for the domain predictor, reversing the relative magnitude of gradient's value should be considered. The next transformation could then take the vector of gradient's value, and change the parameters with a small absolute value to a big ones and vice versa.

This transformation could cause that crucial for G_d changes of Θ_f parameters would be ignored, while parameters that were already well tuned from domain predictor point of view, would be significantly changed. Moreover each value would had its sign changed, so the direction of the optimization step is even more changed.

Such modification could disallow the domain predictor from finding any dependencies. If only G_d finds any pattern in some G_f parameters, they are changed much more than some irrelevant ones, so no meaningful information about domain would occur in feature extractor mapping.

The described modification takes a vector of gradient's values $x = [x_0 ... x_{D-1}]$ and transforms it to $-\lambda \cdot (\max_{0 \leqslant i < D} |x_i| \cdot sign(x) \div x)$, where $\div$ is element-wise difference. Each value of the vector x is subtracted from the largest absolute value

from the entire vector multiplied by the $sign(x)$, so the sign of x is kept until $-\lambda$ multiplication.

Accuracy of a model with such transformation placed at the beginning of the domain predictor G_d was comparable to previous modifications - it classified well 98.38% of samples from source domain and 78.87% from target distribution. The result of domain predictor was much lower - G_d score was just 59.28%. The last column of table 3.4 presents the model's accuracy.

The obtained feature vectors were much more domain invariant than in previous case, at least for the domain predictor used during the learning process. Therefore, the success of this approach shows that some significantly different from GRL gradient modifications should be considered in the domain adaptation.

3.7 Other experiments

3.7.1 Measuring orthogonality

The GRL is used to obtain a feature vector f that contains only the information valuable for the class predictor G_y, and no features required by domain classifier G_d. Therefore we want G_y and G_d to use disjoint sets of data dependencies. Transformation made by each of the classifiers is in fact a matrix multiplication. Feature extractor G_f maps the input to vector $f \in \mathbb{R}^D$. Predictors contain an input layer, that maps $f \in \mathbb{R}^D$ to some $f_i \in \mathbb{R}^I$. Multiplying the coordinates of D-dimensional point by some matrix is in fact reducing its dimensionality by projecting it onto some directions.

We can say then, that using disjoint dependencies is in fact transforming the input point with some pairwise (between predictors) orthogonal vectors. Shifting the domain is therefore moving the data along directions that are used by the domain predictor, which would be unnoticeable for class predictor. If feature vector f would contain only features used by class predictor, such move would not change its values.

We can measure the orthogonality of the vectors used by the predictors, as they are defined by the matrix $D \times I$ of each classifier input layer. If dot product of both matrices it is close to zero, the directions are more orthogonal. When we compute it after the training, it turns out to be a very small number. Therefore directions of G_y and G_d are almost orthogonal.

We can force the predictors to use even more orthogonal vectors by adding to the objective the mean value of the squared dot product, i.e. if we denote the model loss function as $L(\Theta_f, \Theta_y, \Theta_d)$ and classifiers input layer matrices as W_{I_y} and W_{I_d}, then new loss function is:

$$L'(\Theta_f, \Theta_y, \Theta_d) = L(\Theta_f, \Theta_y, \Theta_d) + \overline{(W_{I_y} \cdot W_{I_d}^T)^2}$$

The new loss function caused reducing the dot product indeed, but it had no impact on model's performance, therefore GRL itself force the predictors to use as disjoint sets of feature vector dependencies as possible. As the new loss is punishing for product of model's weights, the optimizer may force them to decay. This would be undesirable, we want the dot product to reduce not because of shorter vectors, but because of their orthogonality. However, monitoring the matrices weights shows that there is no shrinking.

3.7.2 Adam Optimizer

Ganin and Lempitsky used Stochastic gradient descent (SGD) optimizer during their research. However, in recent years some fancy optimization methods released and they can boost the training process even more. One of them is Adam Optimizer [17]. It is much more complex than SGD, with individual learning rate for each parameter and usage of more gradient's properties.

To check if Adam may improve our model, the optimizer for each network was changed. After the training, the accuracy on target domain was slightly higher than before. The learning process was repeated few times and after every iteration the highest score was reached by model tuned with Adam, so it turned out to be a good choice. Its parameters should be initialized carefully, as during the training the loss may get NaN value much easier. In our particular case, the initial learning rate was reduced from 0.01 for SGD to 0.001 for Adam.

3.8 Cloud point shape analysis and visualizations

3.8.1 Conceptors

As previously mentioned, feature vectors for all the samples from both domains can be considered as two point clouds F_S and F_T, that can be approximated with ellipsoids. These ellipsoids are described by the conceptors matrices C_S and C_T. The quality of the feature extractor G_f mapping may be measured as the similarity of ellipsoids approximating F_S and F_T, as a domain invariant transformation produces the same point cloud for both distributions. Measuring the similarity of F_S and F_T can be done by conceptors comparison.

As a conceptor ellipsoid always lies inside an unit sphere, it can be measured, how much of space does it occupies. If the point cloud is scattered over the whole space then it would not be possible to approximate it with nothing but a sphere. However, if points are located in an organized cluster, they can be captured with much more limited ellipsoid. Therefore, the *quota* value measures the fraction of the

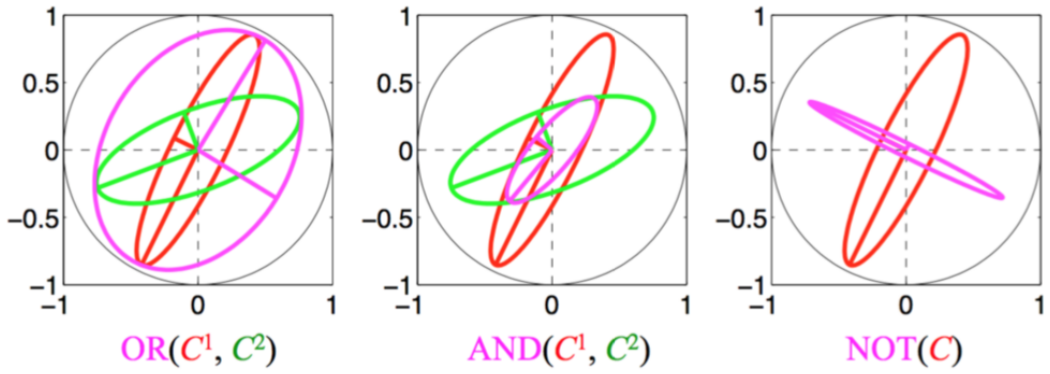


Figure 3.4: Logic operations on 2-dimensional conceptors. Picture from [18].

space that is taken by a conceptor. It can be computed as:

$$Q(C) = \frac{1}{N} \sum_{i=1}^{N} s_i$$

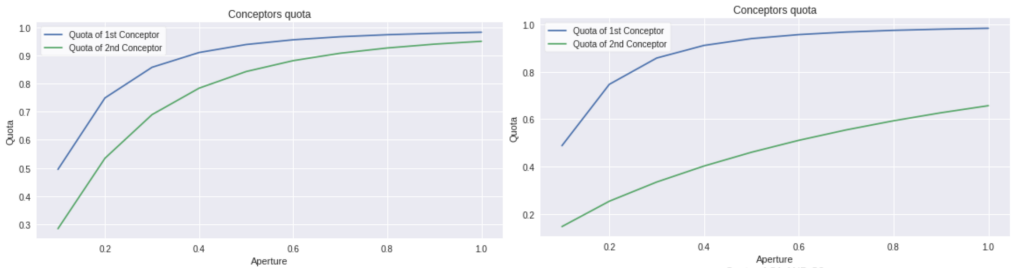
where s_i are singular values of the conceptor matrix C. Moreover, logic operations can be applied to the conceptors, such as $\neg, \vee, \wedge$. The negation of conceptor is the space orthogonal to the ellipsoid. The disjunction is a conceptor that is obtained with a sum of samples sets approximated by two conceptors, while the conjunction is the conceptor that approximates the intersection of these sets. Figure 3.4 charts these operations.

To assess the similarity of two point clouds, corresponding conceptors C_S and C_T should be computed firstly. Then $C_S \vee C_T$ is their intersection. The value $Q(C_S \vee C_T)$ determines its magnitude. If it is close to both $Q(C_S)$ and $Q(C_T)$ then ellipsoids that approximate both point clouds are similar. The quota of conjunction $Q(C_S \wedge C_T)$ should also be close to the $Q(C_S \vee C_T)$, as intersection of these point clouds should include most of them.

Reliable result demands well computed conceptors. The final values of matrix C depend on the magnitude of the scaling parameter α, called the aperture. If it is too low, the conceptor acts as zero mapping, while when the aperture is getting larger, the conceptor matrix becomes an identity matrix.

The proper aperture value could be taken from the paper published by Xu He and Herbert Jaeger "Overcoming Catastrophic Interference using Conceptor-Aided Backpropagation" [18], where conceptors are used to learn MNIST and its permutations. Authors try to obtain a model that performs well on following datasets and conceptors are used to determine the ability of the model to learn a new MNIST permutation and the still spare space, so the next task could fit there. When quota of the concpetor that approximate the already taken space gets 1, capacity of the network is full and trying to fit any other dataset fails.

As MNIST was the main dataset in the paper and received results were satis-



(a) Model trained with GRL (b) Model trained without GRL

Figure 3.5: Quota of concpetors C_{MNIST}(blue) and $C_{\text{MNIST-M}}$(green) for training with or without domain predictor with GRL.

fying, it seems reasonable to use the same α as the authors and try to measure the similarity of F_S and F_T point clouds in $\overline{\text{MNIST-MNIST-M}}$ case. Therefore, conceptor matrices $C_{\text{MNIST}} \in \mathbb{R}^{D \times D}$ and $C_{\text{MNIST-M}} \in \mathbb{R}^{D \times D}$ were computed with $\alpha = 4$, as that value was used in most experiments in the paper. The conceptors were approximating feature vectors $f \in \mathbb{R}^D$ for all samples from both distributions.

When quota values were measured, both of them were over 0.99. This was highly undesirable, as it implied that the points form both point clouds were scattered over the space, therefore the ellipsoids approximating them were not a good depiction. Even the value $Q(C_{\text{MNIST}} \vee C_{\text{MNIST-M}})$ and $Q(C_{\text{MNIST}} \wedge C_{\text{MNIST-M}})$ were close to each other and to quotas of each conceptors, it did not mean the F_{MNIST} and $F_{MNIST-M}$ were similar. We could only conclude that points from these cloud lied in many regions of the space.

For any $\alpha > 1$ the quota of conceptor was over 0.99, to obtain $Q(C_{\text{MNIST}}) < 0.9$ it was needed to set $\alpha = 0.4$. With such a low aperture it is hard to say if the results are reliable. However, for all tested values, the $Q(C_{\text{MNIST}})$ and $Q(C_{\text{MNIST-M}})$ were close.

When model was trained without a domain predictor with GRL, the $Q(C_{\text{MNIST}})$ was as high as previously, but the quota of $C_{\text{MNIST-M}}$ was significantly lower. As $F_{MNIST-M}$ may be captured with a smaller ellipsoid, then it must be different from F_{MNIST}. Figure 3.5 presents the quota values of C_{MNIST} and $C_{\text{MNIST-M}}$when aperture was in range $[0.1, 1]$.

Jaeger proposes conceptors as a way to transform a given point into a point cloud that is well known by the model [7]. This idea sounds attractive for domain adaptation. After computing the conceptor C_S for data cloud F_S, while input sample x would be classified, its feature vector $f = G_f(x; \Theta_f)$ could be multiplied by C_S before processing by class predictor G_y. This would project it into data cloud that is already well known for the label predictor. Therefore the probabilities for each class would be computed as:

$$P(x) = G_y(G_f(x; \Theta_f) \cdot C_S; \Theta_y)$$

Even though this approach was successful with echo-state network [7], it did not improve the trained model's performance. The F_S and F_T point clouds were probably similar enough to had almost equal conceptor matrices, therefore multiplying a point form F_T by C_S was close to identical mapping. Therefore, using conceptors for model's improvement was ineffective.

3.8.2 Visualization

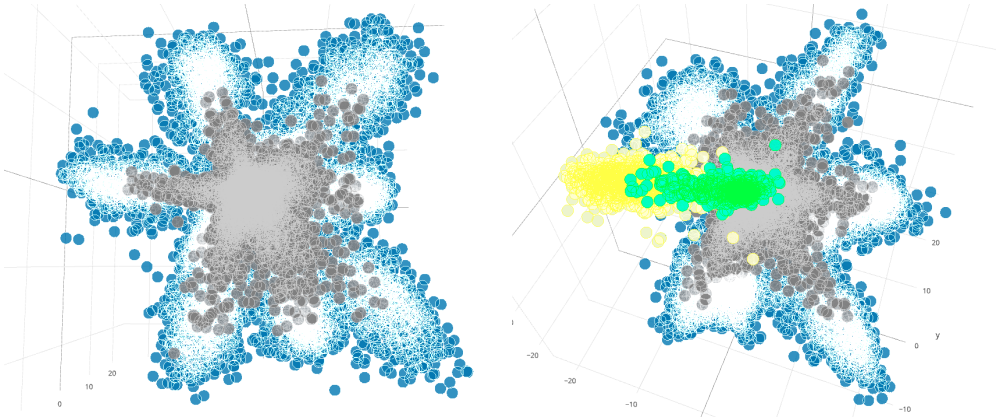
Comparing the F_S and F_T did not succeed with the conceptors. The knowledge about these point clouds could be helpful in understanding the problem of domain adaptation as well as the gradient reversal layer effect. Therefore we could try to visualize them by setting the size of feature extractor to 3. The G_f mapping would be then a function $\mathbb{R}^M \rightarrow \mathbb{R}^3$, where M is the dimensionality of input sample x. Such mapping can be obtained by extending the original feature vector by a single fully connected layer that transforms the feature vector $f \in \mathbb{R}^D$ to $\mathbb{R}^3$. This kind of feature extractor may be denoted as G_3 with parameters Θ_3. When each point in F_S and F_T is a 3-dimensional feature vector, its values may be treated as the coordinates, so the point cloud can be easily drawn.

3.8.3 MNIST and MNIST-M visualizations

Training a model with such a small layer inside is much more difficult task, as many information may be lost during mapping do $\mathbb{R}^3$. Therefore, at the beginning the domain classifier G_d with GRL was not used. The point clouds in $\overline{\text{MNIST-MNIST-M}}$ case might be denoted as F_{MNIST} and $F_{MNIST-M}$. The model achieved 98% accuracy on the MNSIT test set, but it performed poorly on MNIST-M, just 31% of digits, were classified well. As parameters of G_3 were tuned, the point clouds could be visualized. Figure 3.6(a) shows the result of mapping by G_3 all the samples from MNIST test set (blue) and MNIST-M test set (gray). The shape of both point clouds resemble a multi-armed star. Moreover it seems like F_{MNIST} is just a stretched version of $F_{MNIST-M}$. We can expect that each arm contains mapped pictures of same digit. Figure 3.6(b) presents distribution of the images of zero digit from both domains. In both cases the pictures transformations form corresponding arm of the star.

3.8.4 GRL visualizations

As the previous visualization shown, F_S and F_T are quite similar. It could be expected, that learning the model with GRL should make these point clouds even closer. The next model was composed with feature extractor G_3, label predictor G_y, and domain classifier G_d with GRL. After the training the model got 96% accuracy on source domain test set and 40% accuracy on target domain test set. The



(a) MNIST and MNIST-M (b) Zero digit samples

Figure 3.6: The visualization of samples transformed by feature extractor G_3. (a) presents two stars formed by samples from source domain (MNIST/blue) and a target domain (MNIST-M/gray) (b) shows how pictures of zero digit are mapped - yellow points are zeros from MNIST and green ones from MNIST-M

improvement was almost 30% in relation to the model without GRL.

The visualization of both point clouds mapped by G_3 is really surprising. Figure 3.7 shows how samples from source and target domain were transformed. The second picture presents samples from both distributions. The target domain point cloud $F_{MNIST-M}$ formed a blob that is not similar to the source domain distribution F_{MNIST} at all. The last picture shows the location of zero digit images.

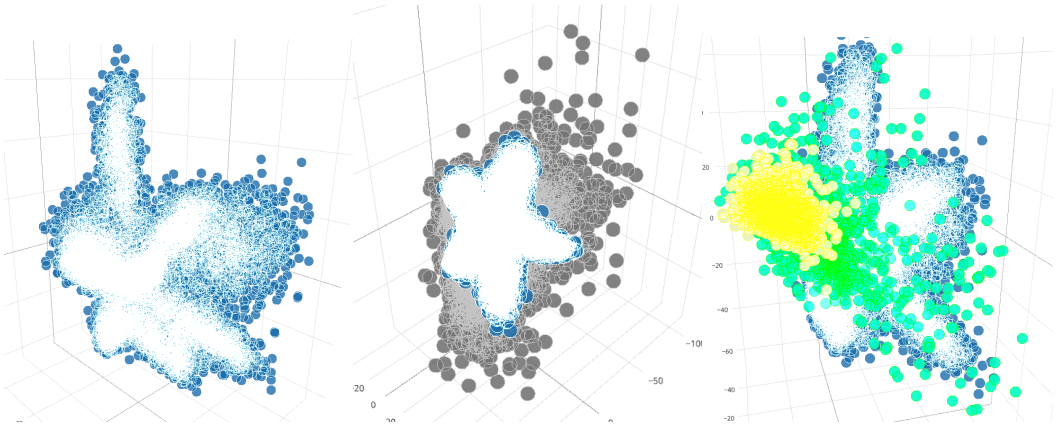
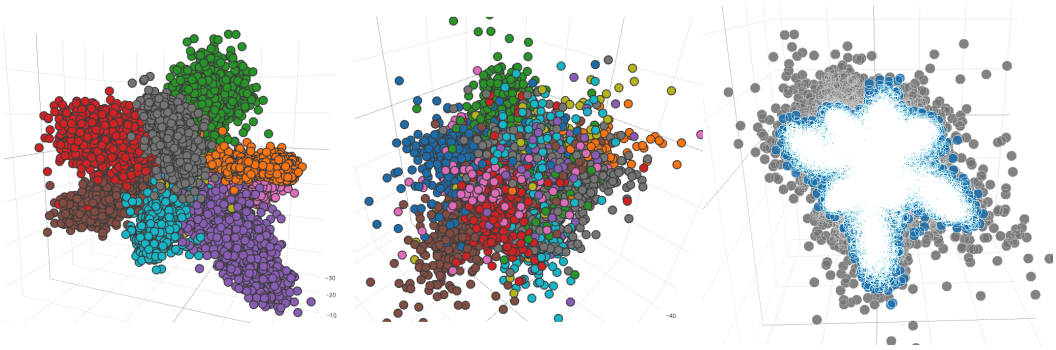


Figure 3.7: Output of G_3 mapping when GRL was used during training. First picture presents how source domain samples are transformed, second one shows the point clouds for both source (blue) and target (gray) domain. Last picture shows how points with 0 label are distributed.

Training process is very noisy, when feature extractor output is so small. The results after different runs were very diverse. The training often failed when the loss became NaN, the variance of model's accuracy over training was high. Mapping the feature vector to just 3 numbers caused the loss of many information. Therefore,



(a) F_{MNIST} (b) $F_{MNIST-M}$ (c) Both point clouds

Figure 3.8: Mapping of the input images of particular digits. The model was learned with GRL and fixed value $\lambda = 0.3$

to avoid the failure of training process, the λ was set to fixed value $\lambda = 0.3$. This allowed the model to perform better, the accuracy on MNIST-M test set was over 50%.

To make the plots even more understandable, the feature vector may returns a 2-dimensional feature vector $f \in \mathbb{R}^2$. Figure 3.9 presents the F_{MNIST} and $F_{MNIST-M}$ mapped to $\mathbb{R}^2$ with and without using the GRL. Without the GRL, the point clouds formed multi-armed stars again. Each star has less arms than number of classes in the dataset, therefore few of them are stacked in the middle. Classifying of these samples is likely much less accurate. Moreover, majority of points from $F_{MNIST-M}$ lies close to the middle of the coordinate system. As input images of different digits are represented by similar numbers, the classifier cannot be sure while predicting their label. When GRL was applied during the training, the accuracy on the MNIST-M was much higher, but the samples formed a blob, therefore the better performance is incomprehensible.

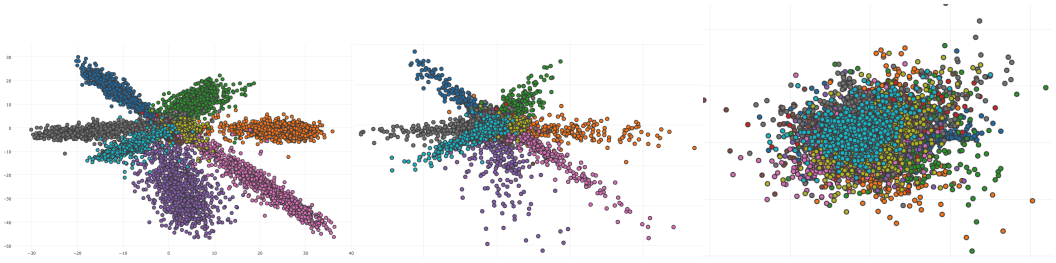
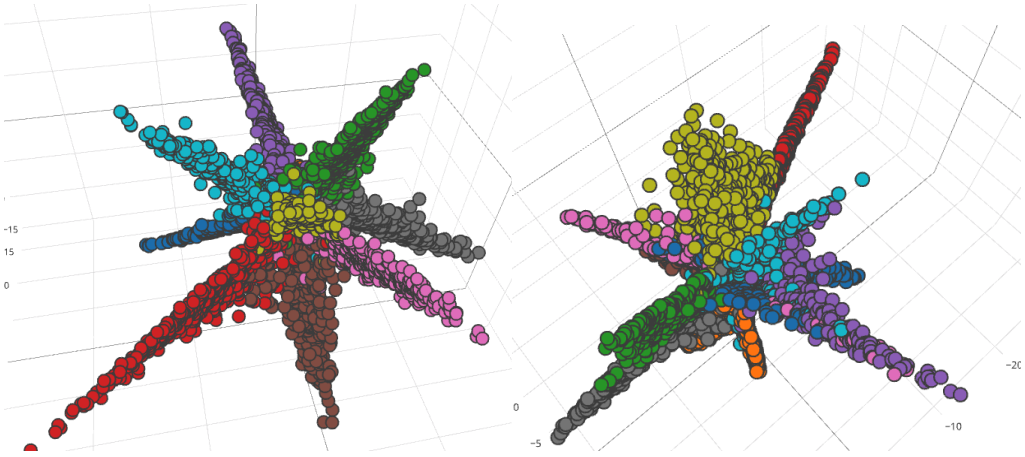


Figure 3.9: 2-dimensional feature vector mapping. First picture shows the distribution of samples from source domain, second one presents the $F_{MNIST-M}$ without applying GRL during training, while the last image shows $F_{MNIST-M}$ when GRL was used.

3.8.5 SVHN visualizations

In $\overline{\text{SVHN-MNIST}}$ case, visualizing the point cloud F_{SVHN} formed by samples from SVHN dataset mapped by feature vector G_3 is much more challenging. SVHN images are more complex, so compressing the information required to a good classification into just 3 values is a tough mission. Fortunately, a model with a complex G_3 architecture reached 92% accuracy on the SVHN test set and 65% on target domain (MNIST) samples. The performance on MNIST test set is satisfying, the majority of the information was not lost during mapping to $\mathbb{R}^3$. Figure 3.10 presents the visualization of F_{SVHN} and F_{MNIST} after mapping by G_3.

The point clouds are much more similar this time, what is related to the better performance of the model on target domain. Also images of every digit from SVHN formed a sharp, individual arm, while pictures from target domain are a bit mixed, with many of the samples close to the middle of the coordinate system. Therefore classifying a sample by its location in the star formed by the point cloud may be unsuccessful.

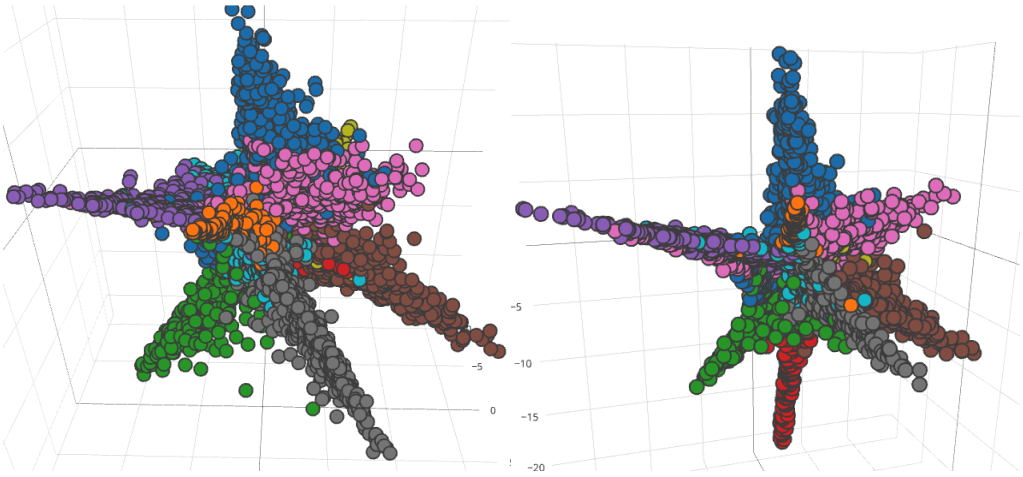


(a) F_{SVHN} (b) F_{MNIST}

Figure 3.10: Distribution of particular digit images after G_3 mapping.

Training with GRL applied was already very unstable in $\overline{\text{MNIST-MNIST-M}}$, therefore, it was not surprising that with in $\overline{\text{SVHN-MNIST}}$ applying GRL during training a fresh model caused the failure of the learning process. Even if λ was set to a fixed small value, the loss function became NaN. Nonetheless, the already trained model could be extended with a domain predictor G_d with GRL plugged to the output of the feature extractor. After that kind of training, accuracy on SVHN test set was 73%, therefore applying GRL increased the model's performance by 12%. λ parameter had to be fixed and small to avoid the training failure again. With $\lambda = 0.2$ the model managed to tune its parameters well. Figure 3.11 presents the distribution of both point clouds mapped by G_3 after the posterior training with GRL applied. This time the target domain samples did not formed a blob, however, the GRL impact on the entire model was much lower in this case. The multi-armed

star formed by F_{SVHN} is not as sharp as previously. Also samples from the F_{MNIST} are less mixed, and the point cloud itself is more similar to the F_{SVHN}.



(a) F_{SVHN} (b) F_{MNIST}

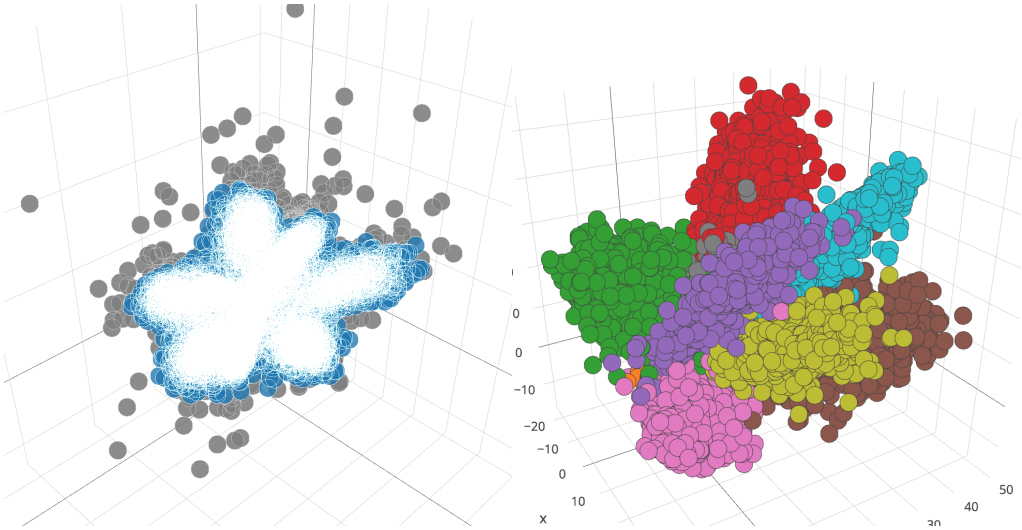
Figure 3.11: Distribution of particular digit images after G_3 mapping, when GRL was used during the training.

3.8.6 3-dimensional layer in class predictor

Such a restrictive limitation of the feature vector f dimensionality caused a serious decline of model's performance. Also using the GRL during training was unsettled, as the learning process often failed. Therefore the layer with the output size 3 could be placed in the class predictor, rather than as the last layer of the feature extractor. This approach favours also the training with GRL, as the gradients from both predictors influence a D-dimensional feature vector. The new architecture is then build with feature extractor G_f that produces the feature vector $f \in \mathbb{R}^D$, domain predictor G_d with GRL and the class predictor G_y^3 with a single hidden layer of size 3.

This architectural change cuased a significant improvement of the model's performance. In $\overline{\text{MNIST-MNIST-M}}$ case after a short training, the accuracy on MNIST-M reached 69.08%, what was much closer to the results obtained by the model with no architectural limitations. To visualize the input sample x, it is firstly transformed by G_f to feature vector $f \in \mathbb{R}^D$ and then mapped by the first layer of class predictor G_y^3 to a 3-dimensional vector $f_3 \in \mathbb{R}^3$. Figure 3.12 presents the result of such mapping. Both point clouds again formed multi-armed stars, but they are much more similar in this case. Also applying GRL did not resulted in transformation of the target domain point cloud to a blob. Using G_y^3 instead of G_3 is definitely a better approach to visualize the data.

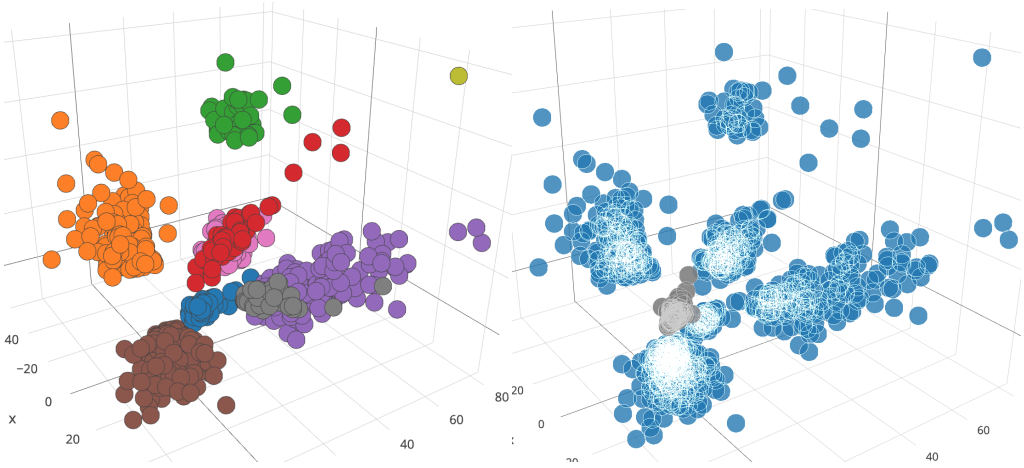
As visualizations with G_y^3 seem more reliable it can be used to some other analysis. As some of the mapped samples are mixed close to the middle of the



(a) Point clouds mapped to $\mathbb{R}^3$ (b) Digits from MNIST mapped to $\mathbb{R}^3$

Figure 3.12: Mapping of F_{MNIST} and $F_{MNIST-M}$ to $\mathbb{R}^3$ by class predictor's first layer. First picture presents both point clouds - blue is the mapping of F_{MNIST}, gray is $F_{MNIST-M}$. The second image shows the distribution of each digit from MNIST. Once again the mapped sampled formed a multi-armed star.

coordinate system, they are likely wrongly classified. Therefore, the distribution of samples with worst and best prediction could be checked. Figure 3.13 presents the location of misclassified samples and the input images that are predicted well with highest score from the model. It confirms that mapping the input into the middle area may cause the mistake during classification.



(a) Best predictions (b) Best and worst predictions

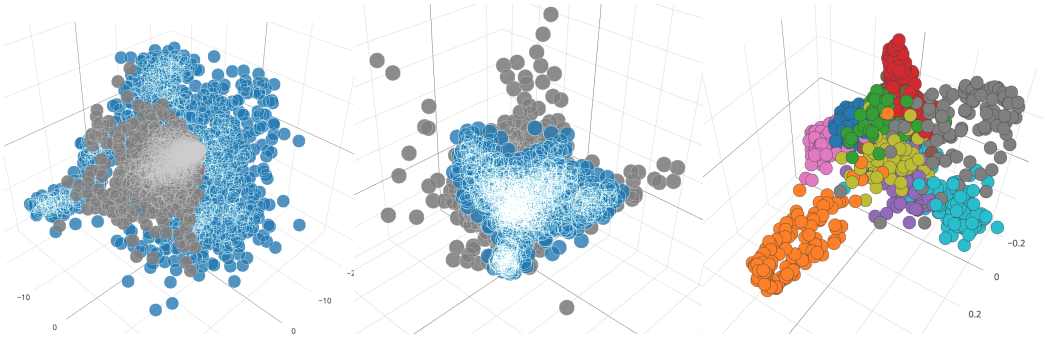
Figure 3.13: Distribution of best and worst predictions by the model. First picture shows the location of samples that are classified well, with highest score assigned by the model. Second image shows all these samples as blue ones and some of the misclassified samples (gray).

3.8.7 Visualizations with dimensionality reduction techniques

Even with 3-dimensional layer in class predictor, forcing the model to use such a small layer causes the decrease of its performance. To avoid this, some of the dimensionality reduction techniques may be used to obtain the representation of the data in $\mathbb{R}^3$. One of them is PCA, that transforms the data by projecting them onto the principal vectors of the data points correlation matrix. To obtain best possible approximation of the data, the vectors corresponding to the greatest principal values should be chosen.

Firstly, the correlation matrix was computed for both F_{MNIST} and $F_{MNIST-M}$. When the singular values of the matrices were computed, the greatest 3 of them were not much greater than many others. It confirms, that the obtaining a good feature extractor demands the dimensionality of feature extractor to be much higher than $\mathbb{R}^3$. Therefore, projecting the data onto just 3 vectors caused a huge loss of the information.

Figure 3.14 presents the visualizations of F_{MNIST} and $F_{MNIST-M}$ obtained with PCA and Kernel-PCA with RBF kernel, that firstly transforms the input data to some higher-dimensional space. The results are not as impressive as with 3-dimensional layer in the model. However, it can be clearly seen that applying GRL makes the point clouds much more similar.



(a) PCA, without GRL (b) PCA, with GRL (c) Kernel-PCA on F_{MNIST}

Figure 3.14: The visualizations with dimensionality reduction techniques. First two pictures presents the point clouds for source (blue) and target (gray) domain, transformed to $\mathbb{R}^3$ with PCA method. Applying GRL during training results in more similar point clouds. The last picture presents distribution of each digit with Kernel PCA used to reduce the dimensionality.

3.8.8 Other visualizations

The well trained feature extractor G_3 could be used to visualize the transformation made by conceptor. With $\overline{\text{MNIST-MNIST-M}}$ distribution, the ellipsoid approximating the F_{MNIST} could be computed and represented as conceptor C_{MNIST}. The

quota $Q(\ C_{\text{MNIST}})$ value was close to 1 for even very small values of aperture α. This seems reasonable, as the obtained multi-armed star cannot be captured with nothing but a sphere. Therefore, the C_{MNIST}matrix was close to the identity matrix, so multiplying the samples of $F_{MNIST-M}$ by C_{MNIST}did not make the point cloud more similar to F_{MNIST}. Figure 3.15 presents the result of such transformation.

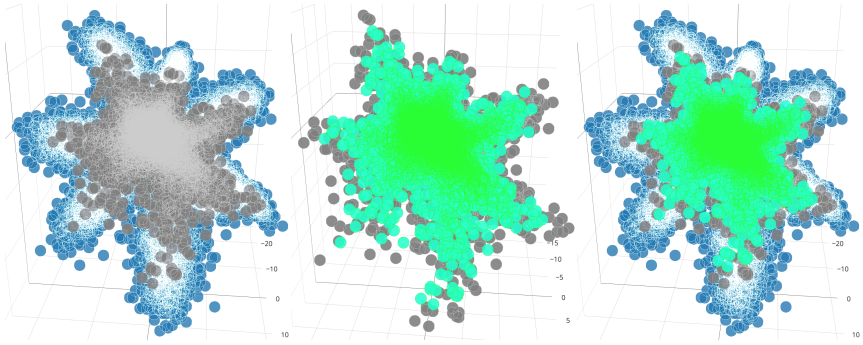
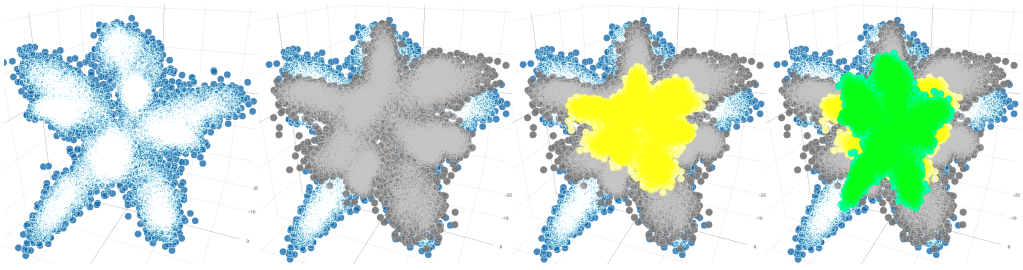


Figure 3.15: Transformation the $F_{MNIST-M}$ point cloud woth C_{MNIST}. First picture shows point clouds F_{MNIST} (blue) and $F_{MNIST-M}$ (gray) obtained after the training (domain predictor with GRL was not used). The second picture presents the transformation of the $F_{MNIST-M}$ point cloud (gray) by conceptor C_{MNIST}(green). The difference is almost imperceptible. The last picture shows all three point clouds together.

The multiplication by conceptor matrix only reduced the area of the point cloud. As every conceptor matrix C is a positive semidefined matrix [7], it can be presented as $C = U\Sigma U^T$, where U and U^T can be interpreted as rotation matrices and Σ is a diagonal matrix with singular values of C on diagonal. As all singular values of conceptor matrix are real numbers from range $[0, 1]$, the conceptor matrix multiplication is in fact rotation, shortening (as non-zero values of Σ are $\leqslant 1$ [7]) in certain directions and then re-rotation of the point cloud (see figure 3.16).



(a) F_M (b) $U^T \cdot F_M$ (c) $\Sigma \cdot U^T \cdot F_M$ (d) $U \cdot \Sigma \cdot U^T \cdot F_M$

Figure 3.16: F_{MNIST} point cloud (denoted as F_M) transformation by its own conceptor C_{MNIST}. The F_{MNIST} (blue points) is firstly rotated (gray), then scaled (yellow) and finally re-rotated (green)

Chapter 4

Conclusion

During the research, the gradient reversal layer was tested in many configurations and modifications. Unquestionably it greatly improves the model performance in domain adaptation. The different gradient transformations showed what was important during the optimization to obtain some satisfying results. Applying a domain predictor with GRL to more layers of the model turned out to be a good approach, that may improve the results. The hyperparameters selection, like optimization algorithm is also a key to success.

Including a 3-dimensional layer inside the network's architecture is a great approach to visualize the model's behaviour. The visualizations form this paper shows the reason of imprecision in target domain classification, obtained multi-armed star are really impressive and prove the relationship between datasets. Also the blob formed by target domain point cloud after applying the GRL is very surprising and shows that the gradient reversal layer considerably handicaps the learning process.

Obtaining a domain shift invariant feature vector is not possible with just the GRL. The distribution information remains in the feature vector after the training. Nonetheless, the representation that is unclear for a single domain predictor causes a significant improvement, therefore, some techniques that would increase the domain invariance would definitely be valuable in the domain adaptation.

I would like to thank my tutor dr Jan Chorowski for tons of advice, a patient guidance and many ideas how to develop the research in the best way. I do appreciate all the time and effort spent on my thesis.

Bibliography

[1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495*, 2014.

[3] Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`.

[4] Data science blog: An intuitive explanation of convolutional neural networks. `https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/`.

[5] Mnist dataset. `http://yann.lecun.com/exdb/mnist/`.

[6] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[7] Herbert Jaeger. Controlling recurrent neural networks by conceptors. *CoRR*, abs/1403.3369, 2014.

[8] Herbert Jaeger. Conceptors: an easy introduction. *CoRR*, abs/1406.2671, 2014.

[9] Pytorch website. `https://pytorch.org`.

[10] Google colab website. `https://colab.research.google.com/`.

[11] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, 2013.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[14] Jinming Cao, Oren Katzir, Peng Jiang, Dani Lischinski, Danny Cohen-Or, Changhe Tu, and Yangyan Li. Dida: Disentangled synthesis for domain adaptation. *arXiv preprint arXiv:1805.08019*, 2018.

[15] Philip Haeusser, Thomas Frerix, Alexander Mordvintsev, and Daniel Cremers. Associative domain adaptation. In *International Conference on Computer Vision (ICCV)*, volume abs/1708.00938, page 6, 2017.

[16] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[18] Xu He and Herbert Jaeger. Overcoming catastrophic interference using conceptor-aided backpropagation. In *International Conference on Learning Representations*, 2018.