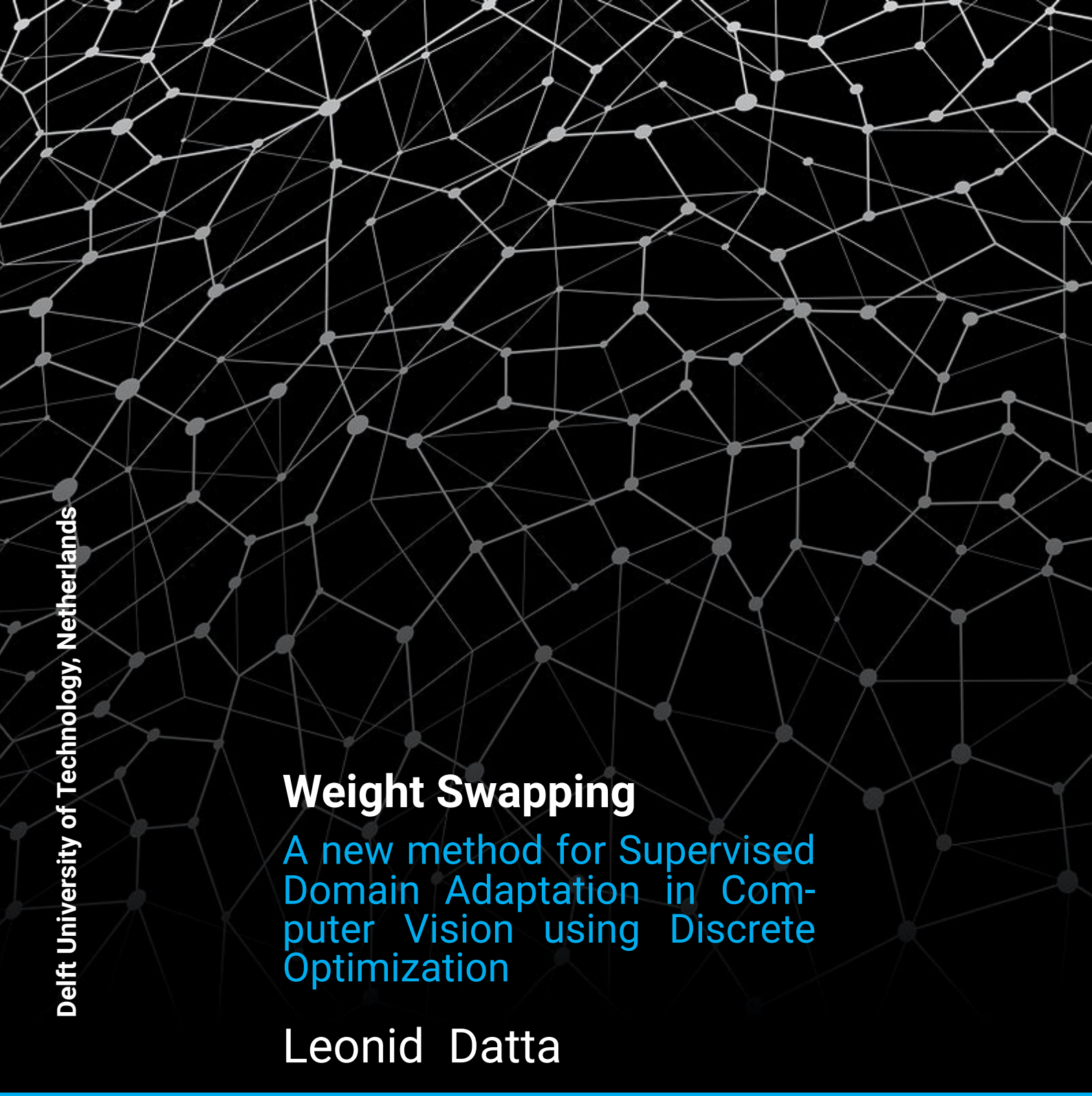

Delft University of Technology, Netherlands

Weight Swapping

A new method for Supervised Domain Adaptation in Computer Vision using Discrete Optimization

Leonid Datta

TU Delft

Weight Swapping

A new method for Supervised Domain Adaptation in Computer Vision using Discrete Optimization

by

Leonid Datta

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday August 19, 2020 at 10:00 AM.

Student number:	4817842	
Project duration:	September, 2020 – August, 2020	
Thesis committee:	Prof. dr. Jan van Gemert,	TU Delft, Chair of the thesis committee and Supervisor
	Prof. dr. David Tax	TU Delft, Internal thesis committee member
	Prof. dr. Matthijs Spaan	TU Delft, External thesis committee member

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This report presents the work done for my master's thesis for the master of science degree program at the Delft University of Technology (TU Delft). The research is conducted at the Computer Vision Lab of the Pattern Recognition and Bioinformatics Group of TU Delft, under the supervision of Dr. J.C. van Gemert. Robert-Jan Bruintjes is my daily supervisor.

Coming from a different part of the world with a different culture, the two years of master's was a difficult journey for me. I thank all the professors and friends who have helped me to overcome the challenges. The eleven-months-long journey of my thesis has given me the passion for computer vision and CNN. I would like to thank *Jan* for accepting me as a graduate student and supervising me for my thesis. I would like to thank $Robert-Jan$ for giving your critical views and offering all your support. I would like to thank $Shampa\ ma'am$ for inspiring me to become a researcher. I would like to thank the thesis committee members Dr. David Tax and Dr. Matthijs Spaan, for your interest in my thesis and for evaluating my work.

Finally, I will not thank my parents $Baba$ and Maa but I can say only this much that without your support, this journey would not have been possible.

Leonid Datta
Delft, August 2020

Contents

1 Scientific Paper 1

2 Background on Neural Network and Convolutional Neural Network 3
2.1 Neural Network 4
2.2 Convolutional Neural Nwtwork 6

Bibliography 11

1

Scientific Paper

Weight Swapping: a new method for Supervised Domain Adaptation in Computer Vision using Discrete Optimization

Leonid Datta
Jan van Gemert (supervisor)
Robert-Jan Bruintjes (daily supervisor)
Computer Vision Lab, Pattern Recognition and Bioinformatics Group
Delft University of Technology, Netherlands

leoniddatta@gmail.com

Abstract

Training Convolutional Neural Network (CNN) models is difficult when there is a lack of labeled training data and no unlabeled data is available. A popular method for this is domain adaptation where the weights of a pre-trained CNN model are transferred to the problem setup. The model is pre-trained on the same task but in a different domain that has plenty of labeled data samples available. In a CNN model, we can rearrange the weights of a convolutional layer by permuting them along the input channel dimension. This work shows that certain weights that are learned in the pre-trained model work well in the problem setup when the weights are rearranged in this manner. Computing the set of all possible rearrangements of the weights is computationally intractable. This work proposes two algorithms to find a good rearrangement of the weights in reasonable computation time. The solutions from the algorithms perform equally well or better than fine-tuning in the domain adaptation between SVHN and MNIST data.

1. Introduction

Convolutional Neural Networks (CNN) has shown promising performance to solve real-world challenges [8] [11] [14]. Training CNN models becomes difficult when there is a lack of labeled data and there is no unlabeled data available [35]. To fill the gap of knowledge in the model, knowledge of solving a similar task is transferred to the current task . This is known as transfer learning. To train CNN models with small training data, transfer learning is one of the most used methods [25] [28] [31]. In transfer learning, when the task from where the knowledge is transferred and the task where the knowledge is transferred is the same, but there is a difference in the distribution of data between the two domains, it is known as domain adaptation [17]. For example, we have two problem settings of image classification tasks for identifying handwritten lowercase alphabets. In one domain the alphabets are written with colors on a white background and in the other domain, the alphabets are written with white color on black background. The task in both of the settings is to classify the lowercase alphabets and the classes are a, b, c, ... z. But there are differences between the distribution of data between the two domains. Here, the knowledge of solving the task in one domain can be used for solving the same task in the other domain. It is an example of domain adaptation. The domain from where the knowledge is transferred is known as the source domain and the domain where the knowledge is transferred is known as the target domain. Domain adaptation is performed when there is either a lack of training data or there is no training data available in the target domain whereas there is plenty of data available in the source domain. When there are a few data samples available in the target domain, it is known as supervised domain adaptation [35]. When there is no training data available in the target domain, it is known as unsupervised domain adaptation [35].

In a CNN model, there are filters or weights that generate the representations at the output channels of a CNN layer. The representations are generated through convolution between the inputs received at the input channels and the weights. We can rearrange the weights of a CNN layer by permuting them along the input channels. In our research, the hypothesis is when a CNN model is trained on the source domain, certain weights that are learned in the source domain work well on the target domain when the weights are rearranged. The set P that contains all of the possible solutions through the rearrangements of the weights is computationally intractable. It makes the problem a discrete optimization problem where the global op-

timal solution is not known. Since P is a computationally intractable, a practical way of solving it is the approximation of a local optimal solution. Determining whether this local optimal solution is the global optimum solution is not tractable. The 'improvement algorithm' is a class of algorithms that starts with an initial solution and then leads to an improved solution through iterations [2] [9]. For the problems where the path that leads to an optimal solution is irrelevant, iterative improvement algorithms often provide the most practical approach [29]. This work approaches the problem by applying an iterative improvement algorithm called 'local search' that can find an optimal solution using reasonable computation time. Local search has been a successful optimization algorithm for these problems despite its simplicity [1] [3] [26]. It can find an optimal solution in reasonable computation time [5] [12]. This work proposes a new method 'weight swapping' using two implementations of the local search algorithm. A popular method for supervised domain adaptation is fine-tuning [6] [30]. In fine-tuning, the weights transferred from the source domain are re-trained on the target domain training data. Weight swapping can find a rearrangement of the weights that performs equally well or better than fine-tuning.

1.1. Contribution

The main contributions of this work are:

1. We show that when a CNN model is trained on the source domain, certain weights that are learned in the source domain work well on the target domain when the weights are rearranged.

2. We propose two implementations of the local search algorithm for finding a rearrangement of the weights that perform equally well or better than fine-tuning.

3. Weight swapping can find a good solution using reasonable computation time.

2. Related work

For the problems with a lack of training data, fine-tuning has been a popular method [6] [7] [15] [30] [37]. [7] and [37] have used fine-tuning using transfer learning. [6] and [30] have used fine-tuning for domain adaptation. [15] proposes a two-step progressive domain adaptation technique by fine-tuning. [4] and [34] use features from the fully connected layers of the trained model and use separate classifiers for the final task. All of these works mentioned use fine-tuning but this work makes use of the already present weights and looks for a rearrangement of the weights instead of fine-tuning them.

Designing a neural network for a specific task requires good expertise in the field. But recent research in neural architecture search (NAS) has made progress on automating the task of neural network design. Recent NAS research [16] uses no back-propagation for training a neural network. [20] searches for optimal structures through learning a surrogate model for guiding the search. [23] and [27] use evolutionary algorithms for searching optimal architecture and use stochastic gradient descent for parameter elimination in NAS. [39] uses reinforcement learning for neural architecture search. [10] treats the weights as variables randomly sampled from a fixed distribution and apply topological search operators to search for an architecture that can perform without any explicit weight training. In contrast, weight swapping uses the already present weights that are transferred from the source domain and instead of updating the weights using gradient descent or searching for optimal architecture, it uses discrete optimization methods to search for an optimal rearrangement of the learned weights.

[22] uses discrete optimization for estimating dense optical flow in computer vision. [9] demonstrates an example of binary image restoration using discrete optimization. As far as we know, this is the first work exploring discrete optimization for domain adaptation in computer vision.

3. Method

In a supervised domain adaptation setting, we have plenty of data samples available in the source domain and few data samples available in the target domain for training. The CNN model is first trained on the source domain and then the weights of the trained model are transferred to the target domain. The transferred weights are then adapted to the target through domain adaptation.

In a CNN model with m layers and first layer being the layer closest to the input layer, each layer generates representations or features of the input images at the output channels of that layer. The representations are generated through convolution between the weights, also known as filters, and the input images received at the input channel of that layer. The representations generated at that layer is the input to the next layer of the model. For example, if layer $(x-1)$ has N_x output channels then the number of input channels in xth layer is N_x. In layer x of a CNN model with N_x input channels, we can rearrange the weights of that layer by permuting them along the input channels. The number of all the possible rearrangement of the weights across the N_x input channels is $N_x!$. For the CNN model with m layers, the first layer being the layer closest to the input, the number of all possible rearrangements of the weights across the m layers is

$$\prod_{i=1}^{m} N_i! = N_1! \times N_2! \times \ldots N_m!. \tag{1}$$

This is computationally intractable. One possible solution can be trying all the possible rearrangements for each layer

separately. Using this heuristic, the number of possible rearrangements is $N_1! + N_2! + \ldots N_m!$. This is still computationally intractable. Our approach is aimed at finding better solutions at a layer and gradually moving to other layers. After all the m layers are looked over for better solutions, the process is repeated over the m layers in similar manner a number of times. The total number of times the process is repeated is the $number\ of\ iterations$. To look for better solutions at each layer in reasonable computation time, 'local search' is applied at every layer.

3.1. Local Search

'Local Search' is an optimization algorithm that is based on a concept of neighborhood. [3] defines the neighborhood as "a set of solutions that are in some sense close to [the current solution] p, for example, because they can be easily computed from p or because they share a significant amount of structure with p". The theory behind the local search is an iterative improvement: it starts with an initial solution, then it looks for solutions at its defined neighborhood. If a better solution is found in the neighborhood, the current solution is replaced with the better solution and the search continues [3].

3.2. Neighborhood

A cheap way of generating a neighbor of the present solution is by swapping two elements of the present solution. So, in the algorithms, the neighborhood of the current state s is defined as the set S, each element of which is generated by swapping weights of two input channels out of the N_x input channels in layer x of the model. An example of the generated neighborhood by swapping two channels is shown in table 1. This work proposes implementing two varieties of the local search algorithm

1. Best legal neighbor
2. First legal neighbor.

Current state	[a b c d]	
No.	Two channels to be swapped	Neighbor
1	[a b]	[b a c d]
2	[a c]	[c b a d]
3	[a d]	[d b c a]
4	[b c]	[a c b d]
5	[b d]	[a d c b]
6	[c d]	[a b d c]

Table 1. Generated neighbors of [a b c d]

3.3. Starting Point

Local search works on the concept of neighborhood [2]. Weight swapping generates neighbor at each layer. There can be two starting points of looking for neighbors. One is starting at the first layer and gradually moving forward to the last layer (referred as 'first to last'). The other one is starting at the last layer and gradually moving back to the first layer (referred as 'last to first'). In a neural network, the later layers are functions of the previous layers. This leads to a hypothesis that weight swapping using first to last leads to a better solution than weight swapping using last to first.

3.4. Best legal neighbor

This algorithm is based on the concept of 'best improvement' [3]. In 'best improvement', the algorithm looks at all the neighbors based on the defined neighborhood and moves the current state to the neighbor that generates the best and improved solution. The terminologies used in the algorithm description are described in table 2. Algorithm 'Best legal neighbor' is described in Algorithm 1.

Terminology	Description
acc	Accuracy when the model trained on source domain is validated on the target domain data before domain adaptation
$iterations$	A list of all integers between 1 and the $number\ of\ iterations$ (including both) in ascending order
$layers$	A list of all integers between 1 and m (including both) either in ascending or descending order

Table 2. Description of terminologies in algorithms

Algorithm 1: Best legal neighbor:

for $iterate\ in\ iterations$ **do**
  for $l\ in\ layers$ **do**
    From the input channels of layer number l , generate neighbor set S ;
    for $c\ in\ S$ **do**
      Measure the accuracy of the neighbor solution c on target domain data ;
      Remove c from S;
    end
    Select the neighbor c^* that produces best accuracy acc_max on target domain data ;
    if $acc\_max > acc$ **then**
      Move the current state to c^* ;
      $acc = acc\_max$;
    end
  end
end

3.5. First legal neighbor

This algorithm is based on the concept of 'first improvement' [3]. In 'first improvement', the algorithm looks at a single neighbor chosen randomly from the neighborhood and moves the current state to that neighbor if the neighbor is an improved solution. First legal neighbor is a more greedy approach than best legal neighbor. The terminologies used in the algorithm description are described in table 2. Algorithm 'First legal neighbor' is described in Algorithm 2.

Algorithm 2: First legal neighbor

```
for iterate ⩽ iterations do
    for l in layers do
        From the input channels of layer number l ,
          generate neighbor set S;
        Shuffle the elements of S randomly ;
        for c in S do
            Measure the accuracy val_acc of the
              neighbor solution c on target domain
              data ;
            Remove c from S;
            if val_acc > acc then
                Move the current state to c ;
                acc = val_acc;
            end
        end
    end
end
```

3.6. Computational Complexity

The number of all possible rearrangements across the input channels of all the layers is

$$\prod_{i=1}^{m} N_i! = N_1! \times N_2! \times \ldots N_m!. \tag{2}$$

In weight swapping, the required number of computations is

$$(number\ of\ iterations \times \sum_{i=1}^{m} (N_i \times (N_i - 1))/2). \tag{3}$$

Weight swapping is computationally reasonable.

4. Experiments

4.1. RGB MNIST experiment

4.1.1 Data and Architecture

The RGB MNIST experiment uses the MNIST handwritten digits data [18]. The MNIST handwritten digits data contains 60000 train images and 10000 test images that represent handwritten 0 to 9 digits (10 classes). Each sample of MNIST contains exactly one digit. The image samples are of size 28×28. For the experiment, the greyscale images are converted to RGB ($3 \times 28 \times 28$) images where exactly one channel out of the three are copied from MNIST and the rest of the channels are zeros. The experiment uses three datasets:

1. Source domain training data: red colored MNIST images as shown in figure 1.
2. Source domain test data: red colored MNIST images as shown in figure 1.
3. Target domain test data: green colored MNIST images as shown in figure 2.

Figure 1. Samples from red MNIST data

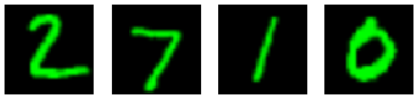

Figure 2. Samples from green MNIST data

A fully convolutional network (FCN) is defined as the broad architecture class which outputs a grid [36]. FCN was popularized by [21]. FCN models are usually followed by a global average pooling layer that outputs the average of all the values of the grid and a softmax layer [13] [19]. The architecture used in this experiment is a FCN inspired from [13] and it is shown in figure 3.

4.1.2 Experiment

The model is first trained on the source domain training data and then tested on the source domain test data and the target domain test data. The input channels at the first layer of the model are the red (R), green (G), and blue (B) channels of the input image. We generate all the possible solutions

through permuting over the three input channels [R G B] of layer 1. All the possible solutions are shown in table 3. The five generated neighbors of the current model are then tested on the green test data.

solution name	solution
current state	[R G B]
solution 1	[B G R]
solution 2	[G R B]
solution 3	[R B G]
solution 4	[G B R]
solution 5	[B R G]

Table 3. all possible solutions of [R G B]

4.1.3 Result

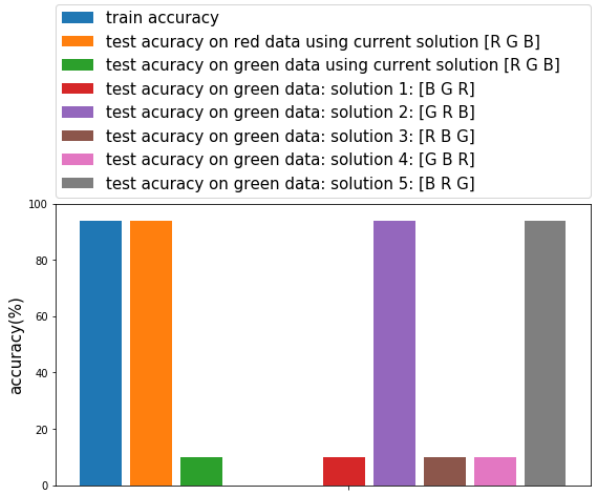

Figure 6. Result of RGB MNIST experiemnt

Figure 6 reports the results of the RGB MNIST experiment. It is evident from the result that solution 2 [G R B] and solution 5 [B R G] are the optimal solutions among the five generated solutions. What is noticeable here is in both of the optimal solutions, the G input channel is replaced by R. Since, the values at B and G channels of the train images were zeros, the weights in the input channels did not get any gradients and as a result, they did not get updated. When the weights of channel G is swapped with that of R, it produces good accuracy because the R channel weights got updated during training. Among the two solutions 2 and 5, solution 2 [G R B] is reachable from the current solution [R G B] with a single swap. The result of this experiment shows that an optimal solution can be reached with swaps if the weights already exist in the model.

4.2. Toy Data Experiment

4.2.1 Data and Architecture

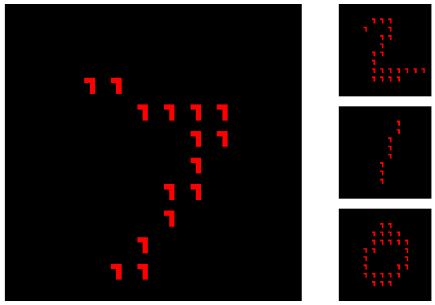

Figure 7. Samples from R7 data

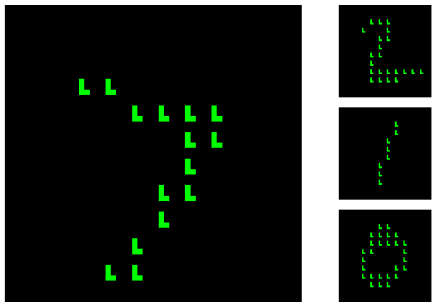

Figure 8. Samples from GL data

The toy data experiment is carried out to judge how weight swapping performs when there are two variations between the target and the source domain. The toy data experiment makes use of the MNIST handwritten digits data [18]. For the experiment, two domains are generated from the MNIST data termed as 'R7' and 'GL'. For better clarity of data, the MNIST data are resized to 56×56 binary images. From this base set up, the R7 domain is generated by converting the 56×56 binary images to RGB images of size $3 \times 56 \times 56$ where the digits are red in color and are written with small '7' shapes. Sample images of R7 domain are shown in figure 7. Similarly, the 'GL' domain, shown in figure 8, is generated where the digits are green in color and the digits are written with small 'L' shapes. R7 and GL domains have their training data (60000 samples) and test data (10000 samples). The test data of both the domains are further split into two parts: target domain data (first 1000

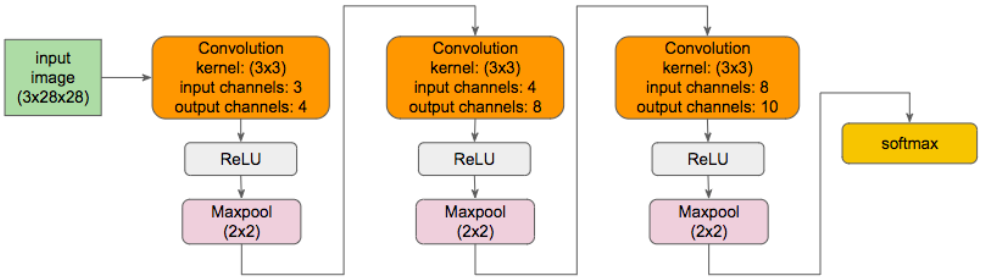

Figure 3. Architecture for RGB MNIST experiment

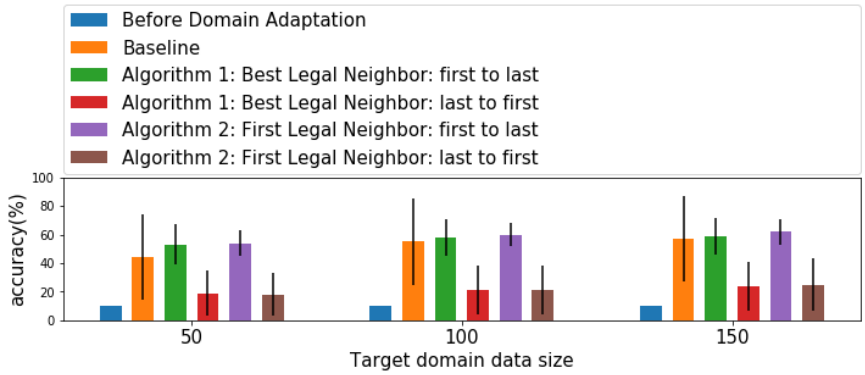

Figure 4. Result of toy data experiment R7 $\rightarrow$ GL. All the experiments are carried out on five different random seeds. The reported results are the mean (shown as bar plots) and standard deviation (shown as error bars) of the test accuracies achieved from the model over the five random seeds.

samples) and test data (the rest of the 9000 samples). We have two experimental setups of data now:

1. R7 $\rightarrow$ GL: Source domain is R7 training data, target domain is GL target domain data and test data is GL test data.

2. GL $\rightarrow$ R7: Source domain is GL training data, target domain is R7 target domain data and test data is R7 test data.

The toy experiment is carried out on an FCN architecture which is shown in figure 9. The architecture is inspired from [33] and [32].

4.2.2 Baseline

The most commonly used method for domain adaptation is fine-tuning [6] [30]. The performance of the weight swapping method is compared with fine-tuning. The models trained on the source domain are fine-tuned on the target domain. The best practice for fine-tuning is to use a lower learning rate (usually 10 times lower) than the learning rate used for training [38]. The trained model is fine-tuned using the same learning rate and a 10 times lesser learning rate and the better fine-tuned (producing better fine-tuning accuracy) model is tested on the test data. The accuracy achieved using fine-tuning on the test data is set as the baseline.

4.2.3 Experiment

After the model is trained on the source domain, we perform weight swapping for domain adaptation. For domain adaptation, three different data sizes are chosen for target domain data. They are 50 (5 samples per class), 100 (10 samples per class), and 150 (15 samples per class). The motivation behind choosing the data size was using a small number of samples per class and equal distribution of samples across the 10 classes. The data samples for the target domain data are randomly sampled from the 1000 data samples of the target domain data. After domain adapta-

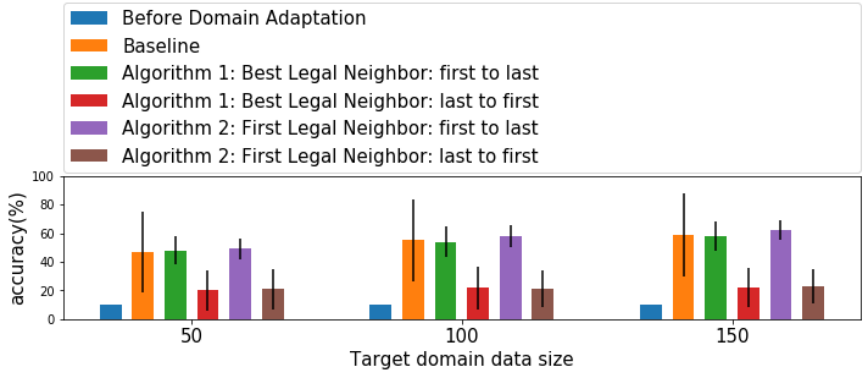

Figure 5. Result of toy data experiment GL $\rightarrow$ R7. All the experiments are carried out on five different random seeds. The reported results are the mean (shown as bar plots) and standard deviation (shown as error bars) of the test accuracies achieved from the model over the five random seeds.

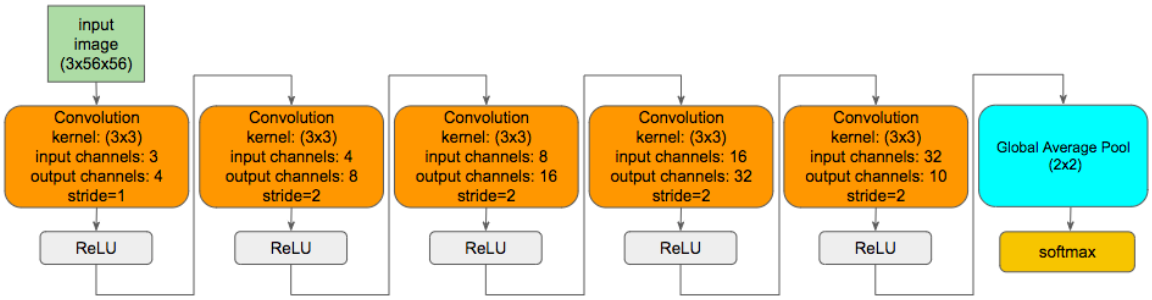

Figure 9. Architecture for toy data experiment

tion is performed on the model using weight swapping, it is tested on the test data. This is done for both of the experimental setups R7 $\rightarrow$ GL, and GL $\rightarrow$ R7. For both of the experimental setups, the experiments are carried out once using first to last and the other time using last to first. The $number\ of\ iterations$ used in this experiment is 10.

4.2.4 Result

Figure 4 reports the result of the toy data experiment R7 $\rightarrow$ GL using first to last and last to first. Figure 5 reports the result of the toy data experiment GL $\rightarrow$ R7 using first to last and last to first. The outcomes of both of the experimental setups are similar. In both of the experiments, best legal neighbor and first legal neighbor can improve the representations over the target domain when they use first to last and last to first. But the accuracy produced by the algorithms using last to first is quite lower than using first to last. Another noticeable result is the accuracy from the algorithms using last to first has a higher standard deviation than the same using first to last. This shows that the algorithms find a better solution more consistently and are less effected by change over initialization when they use first to last. These insights from the results validate our hypothesis that weight swapping using first to last finds a better solution than weight swapping using last to first.

Comparing the results from figure 4 and figure 5, it is seen that weight swapping using first to last slightly outperforms the baseline. The accuracy of weight swapping using first to last has lower standard deviations than the baseline. It shows that weight swapping using first to last finds a better solution more consistently and is less affected by the change over different initialization. This result validates our hypothesis that certain weights that are learned in the source domain work well on the target domain when the weights are rearranged. It also shows that weight swapping can overcome two variations (the color and the '7' and 'L' shapes) between the source domain and the target domain and can find a solution that slightly outperforms the baseline with lower standard deviation.

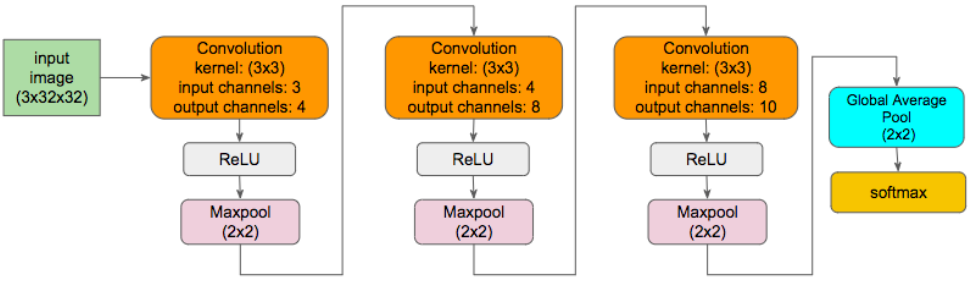

Figure 10. Architecture for real data experiment

4.3. Real Data Experiment

4.3.1 Data and Architecture

After the successful validation of the hypotheses on a domain adaptation setup between two toy domains, the real data experiment is carried out to judge how weight swapping performs on a more realistic domain adaptation setup than the toy data. For the real data experiment, Street View House Number (SVHN) data and MNIST handwritten digits data are used [18] [24]. SVHN data contains 73257 train images and 26032 test images of size $3 \times 32 \times 32$ for an image classification setting. They represent house numbers of houses from street view and contain digits 0 to 9 (10 classes). Each image may contain more than one digit but the digit at the center/focus is the ground truth label. SVHN data samples are shown in figure 11. For matching the dimensions of the MNIST data with SVHN data, the MNIST images have been resized to 32×32 using zero padding and further converted to RGB images of size $3 \times 32 \times 32$ where R, G, B channel values have been copied from the greyscale MNIST images. Data samples of MNIST is shown in figure 12. We have four data sets: MNIST training data (60000 samples), MNIST test data (10000 samples), SVHN training data (73257 samples), and SVHN test data (26032 samples). The test data are further split into two parts: target domain data (first 1000 samples) and test data (the rest of the 25032 samples of SVHN test data and the rest of the 9000 samples of MNIST test data). We have two experimental setups of data:

1. SVHN $\rightarrow$ MNIST: Source domain is SVHN training data, target domain is MNIST target domain data and test data is MNIST test data.
2. MNIST $\rightarrow$ SVHN: Source domain is MNIST training data, target domain is SVHN target domain data and test data is SVHN test data.

For the real data experiment, an FCN architecture has been used which is shown in figure 10. It is adapted from [13].

Figure 11. Samples from SVHN data

Figure 12. Samples from MNIST data

4.3.2 Baseline

To judge the performance of weight swapping, we use the same baseline setting of toy data experiment (described in section 4.2.2) using the real data experiment's source and target domain.

4.3.3 Experiment

Similar to the toy data experiment, the model is first trained on the source domain data and then we perform weight swapping for domain adaptation to the target domain data. In the real data experiment SVHN $\rightarrow$ MNIST, the chosen

data sizes for the target domain are 30 (3 samples per class), 50 (5 samples per class), 100 (10 samples per class), 150 (15 samples per class). In the real data experiment MNIST $\rightarrow$ SVHN, the chosen data sizes for the target domain are 50 (5 samples per class), 100 (10 samples per class), 150 (15 samples per class), 200 (20 samples per class), 250 (25 samples per class). The data samples for the target domain are randomly sampled from the target domain data. It is evident from the results of the toy data experiment that weight swapping finds a better solution when it uses first to last instead of last to first. That is why in the real data experiment, the experiments are carried out using first to last only. The $number\ of\ iterations$ used in this experiment is 10.

4.3.4 Result

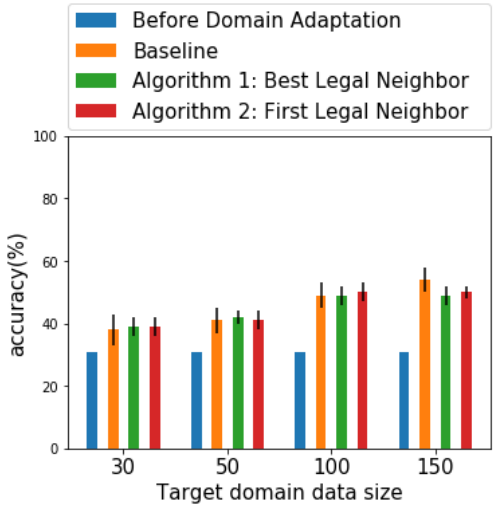

Figure 13. Result of real data experiment SVHN $\rightarrow$ MNIST. All the experiments are carried out on five different random seeds. The reported results are the mean (shown as bar plots) and standard deviation (shown as error bars) of the test accuracies achieved from the model over the five random seeds.

Figure 13 reports the test accuracy values achieved before domain adaptation, the baseline, after domain adaptation with best legal neighbor and after domain adaptation with first legal neighbor for experiment SVHN $\rightarrow$ MNIST. Figure 14 reports the same but for the experiment MNIST $\rightarrow$ SVHN. Results from both of the experiments show that weight swapping can improve the representations over the target domain. The found solutions perform equally well or slightly better when compared to the baseline. But as the size of the target domain data increases, the performance of weight swapping is slightly lower than the baseline. The standard deviation of the accuracies of the baseline is slightly higher than that of weight swapping. It shows that weight swapping is less affected than the baseline by the change over random seeds for initialization. This experiment shows that the hypothesis that certain weights that are learned in the source domain work well on the target domain when the weights are rearranged is valid for more realistic data than toy data.

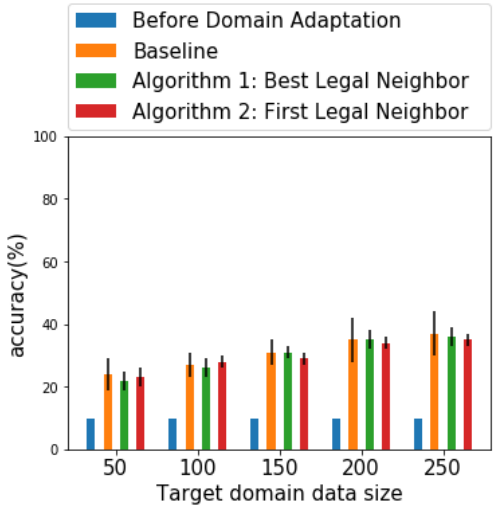

Figure 14. Result of real data experiment MNIST $\rightarrow$ SVHN. All the experiments are carried out on five different random seeds. The reported results are the mean (shown as bar plots) and standard deviation (shown as error bars) of the test accuracies achieved from the model over the five random seeds.

5. Discussion

In this work, we investigate an alternative approach for supervised domain adaptation. It is often the case that weights of some layers are transferred to a new architecture for domain adaptation and the rest of the layers are initialized randomly. The whole network is then fine-tuned on the target domain data. Since the weights are not updated in weight swapping, it is difficult to use weight swapping where the source domain and the target domain use different architecture.

The experiments are designed to first test weight swapping in a toy setup and later on a more realistic setup than the toy setup. Results suggest that weight swapping can take care of two variations in toy data and slightly beat the baseline. On a more realistic data than toy data, weight swapping performs equally well as the baseline. The research can be further proceeded by studying how the performance of weight swapping varies with the distance between the source domain and the target domain.

References

[1] Emile aarts and jan karel lenstra. local search in combinatorial optimization. john wiley and sons, journal=Artificial Intelligence, volume=118, number=1-2, pages=115–161, year=2000.

[2] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[3] J. L. Ambite and C. A. Knoblock. Planning by rewriting. *Journal of Artificial Intelligence Research*, 15:207–261, 2001.

[4] Y. Bar, I. Diamant, L. Wolf, and H. Greenspan. Deep learning with non-medical training used for chest pathology identification. In *Medical Imaging 2015: Computer-Aided Diagnosis*, volume 9414, page 94140V. International Society for Optics and Photonics, 2015.

[5] J. Boyan and A. W. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1(Nov):77–112, 2000.

[6] V. Campos, B. Jou, and X. Giro-i Nieto. From pixels to sentiment: Fine-tuning cnns for visual sentiment prediction. *Image and Vision Computing*, 65:15–22, 2017.

[7] S. Chopra, S. Balakrishnan, and R. Gopalan. Dlid: Deep learning for domain adaptation by interpolating between domains. In *ICML workshop on challenges in representation learning*, volume 2, 2013.

[8] W. Fang, P. E. Love, H. Luo, and L. Ding. Computer vision for behaviour-based safety in construction: A review and future directions. *Advanced Engineering Informatics*, 43:100980, 2020.

[9] P. Felzenszwalb and R. Zabih. Discrete optimization algorithms in computer vision. *Tutorial at CVPR*, 2007.

[10] A. Gaier and D. Ha. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*, pages 5364–5378, 2019.

[11] J. Guo, H. He, T. He, L. Lausen, M. Li, H. Lin, X. Shi, C. Wang, J. Xie, S. Zha, et al. Gluoncv and gluonnlp: Deep learning in computer vision and natural language processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020.

[12] P. V. Hentenryck and L. Michel. *Constraint-based local search*. The MIT press, 2009.

[13] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[14] M. R. Ibrahim, J. Haworth, and T. Cheng. Understanding cities with machine eyes: A review of deep computer vision in urban analytics. *Cities*, 96:102481, 2020.

[15] N. Inoue, R. Furuta, T. Yamasaki, and K. Aizawa. Cross-domain weakly-supervised object detection through progressive domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5001–5009, 2018.

[16] A. S. E. J. C. Joseph Mellor, Jack Turner. Neural architecture search without training. *arXiv preprint arxiv.2006.04647v1*, 2020.

[17] W. M. Kouw and M. Loog. An introduction to domain adaptation and transfer learning. *arXiv preprint arXiv:1812.11806*, 2018.

[18] Y. LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[19] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[20] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[21] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[22] M. Menze, C. Heipke, and A. Geiger. Discrete optimization for optical flow. In *German Conference on Pattern Recognition*, pages 16–28. Springer, 2015.

[23] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[24] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[25] H.-W. Ng, V. D. Nguyen, V. Vonikakis, and S. Winkler. Deep learning for emotion recognition on small datasets using transfer learning. In *Proceedings of the 2015 ACM on international conference on multimodal interaction*, pages 443–449, 2015.

[26] C. H. Papadimitriou. Combinatorial optimization. *Algorithms and Complexity*, 1982.

[27] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[28] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[29] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 2002.

[30] T. Schlegl, J. Ofner, and G. Langs. Unsupervised pre-training across image domains improves lung tissue classification. In *International MICCAI Workshop on Medical Computer Vision*, pages 82–93. Springer, 2014.

[31] A. Shafaei, J. J. Little, and M. Schmidt. Play and learn: Using video games to train computer vision models. *arXiv preprint arXiv:1608.01745*, 2016.

[32] T. A. Soomro, A. J. Afifi, J. Gao, O. Hellwich, L. Zheng, and M. Paul. Strided fully convolutional neural network for boosting the sensitivity of retinal blood vessels segmentation. *Expert Systems with Applications*, 134:36–52, 2019.

[33] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[34] B. Van Ginneken, A. A. Setio, C. Jacobs, and F. Ciompi. Off-the-shelf convolutional neural network features for pulmonary nodule detection in computed tomography scans. In *2015 IEEE 12th International symposium on biomedical imaging (ISBI)*, pages 286–289. IEEE, 2015.

[35] M. Wang and W. Deng. Deep visual domain adaptation: A survey. *Neurocomputing*, 312:135–153, 2018.

[36] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 129–137, 2017.

[37] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[38] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *Unpublished Draft. Retrieved*, 19:2019, 2019.

[39] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

2

Background on Neural Network and Convolutional Neural Network

2.1. Neural Network

Neural network (NN) is a computing method that learns pattern from data. It has small interconnected working units of it known as perceptrons. A perceptron is composed of four elements:

- Input node
- Weights vector
- Activation function
- Output node

The input vector $x = [x_1\ x_2\ \dots x_m\ ]$ is received at the input node and an weighted sum of the input vector x is calculated. The weighted sum is $x^\top w$ where $w = [w_1\ w_2\ \dots w_m\ ]$ is the weight vector. To the input vector x, an extra term 1 is added so that the weighted sum can be shifted easily using its corresponding weight w_0. This is called bias. The weighted sum calculated using $x^\top w$ is passed through a function. This function f is non-linear in nature. It is known as the 'activation function'. The output from this function is generated at the output node. This is how the input data flows from the input node to the output node of a perceptron. Since the input is passed through only one set of weighted sum and activation function, it is known as single layer perceptron. In practice, the perceptrons have multiple layers i.e. the input is passed through a number of sets of weighted sums and activation functions before the output is generated. There are layers which are present between the input and the output layers. These layers are called the hidden layers. For a single layer perceptron that has input vector $x = [1\ x_1\ x_2\ \dots x_m\ ]$ and the weight vector $w = [w_0\ w_1\ w_2\ \dots w_m\ ]$, the structure is shown in figure 2.1. For a NN that has two hidden layers, the diagram is shown in figure 2.2.

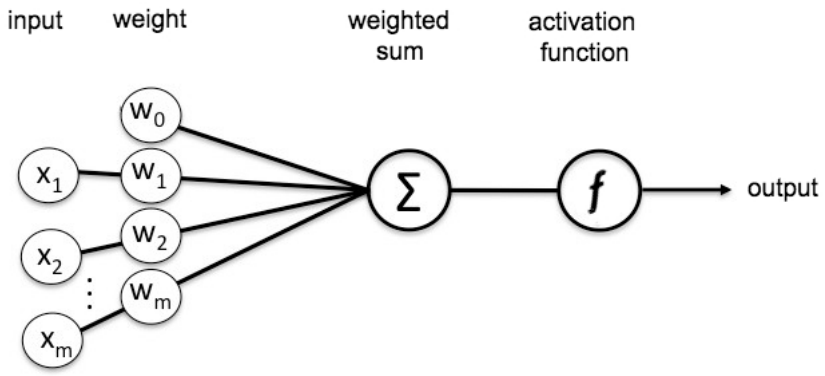

Figure 2.1: Structure of a perceptron with zero hidden layer or simply a single layer perceptron [1]

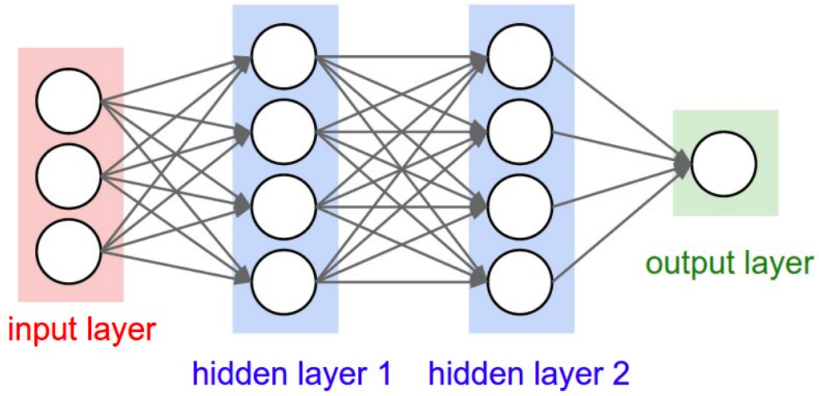

Figure 2.2: Structure of a neural network with two hidden layers [4]

In real-world problems, the pattern from data can be linear or non-linear. A linear pattern can be approximated using a non-linear function but a non-linear pattern cannot be approximated using a linear function. If the activation is a linear function, then the whole set of layers in the NN can be compressed to a single layer because the linear representation of another linear representation can be compressed to a single linear representation. Because of this, activation functions are typically non-linear.

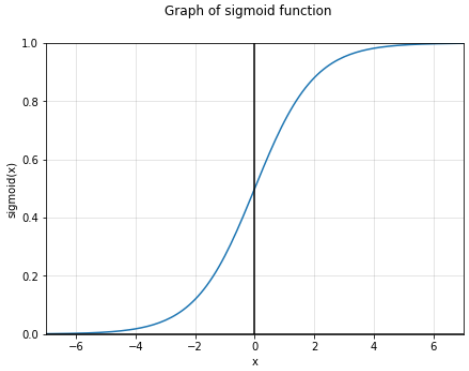

Figure 2.3: The Graph of the logistic sigmoid or the sigmoid activation function [1]

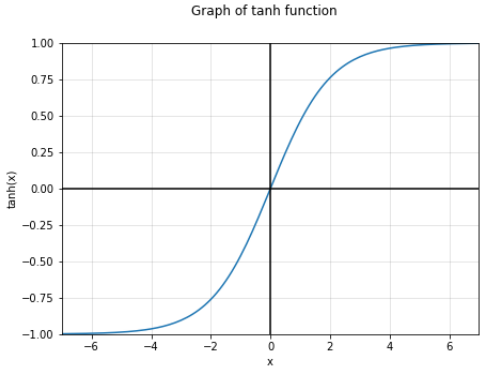

Figure 2.4: The Graph of the tanh activation function [1]

Some of the widely used activation functions are

- Sigmoid: The logistic-sigmoid function is commonly referred as the sigmoid function. It is defined as

$$f(x) = \frac{1}{1+e^{-x}}$$

for x being the input of the function. The sigmoid function is bounded, continuous, differentiable and not a zero centered activation function. Calculation of the sigmoid function is not cheap because it is exponential in nature. The graph of the sigmoid function is shown in figure 2.3.

- Tanh: The tanh activation function is defined as

$$f(x) = \frac{1-e^{-x}}{1+e^{-x}}$$

for x being the input to the function. The Tanh function is bounded, continuous, differentiable and zero centered activation function. Calculation of the tanh function is not cheap because it is exponential in nature. The graph of the tanh function is shown in figure 2.4.

- ReLU: The rectifier linear unit activation function is usually referred as the ReLU. It is defined as

$$f(x) = max(0, x)$$

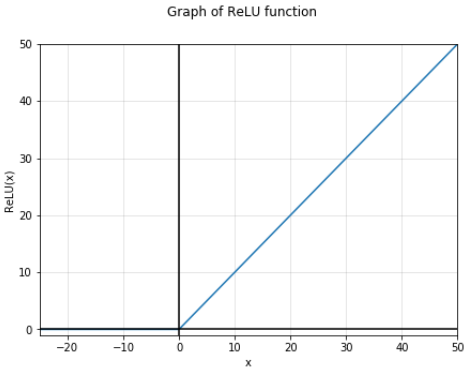

Figure 2.5: Graph of ReLU activation function [1]

for x being the input to the function. It was introduced in [5]. The ReLU function is continuous, differentiable (except at $x = 0$) and very cheap to calculate. The graph of the ReLU function is shown in figure 2.5.

After the output is calculated at the output layer, it is then compared to the ground truth or the desired output. A function that quantifies the difference between the calculated output and the ground truth is known as the loss function. Some of the most used loss functions are negative log-likelihood, L2 loss, mean squared loss, etc.

After the loss is calculated, the weights are updated such that the loss function value decreases. It is known as optimization. The most used method in optimization is gradient descent. Gradient descent work on the principle of moving on the opposite direction of the gradient of the loss function. The gradient is calculated with respect to the weights. The gradient is calculated as

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w} \tag{2.1}$$

where L is the loss, $z = x^\top w$ and $y = f(z)$ where f is the activation function. Through the gradient calculation, the weights are updated at the opposite direction of the gradient at a rate of α known as the learning rate. The weight update equation is defined as

$$w = w - \alpha \times \frac{\partial L}{\partial w}. \tag{2.2}$$

During optimization, the gradients flow from the last layer to the input layer. The flow of data from the input layer to the output layer is known as the forward pass. The flow of data from the output layer to the input layer for weight update is known as backward pass or back-propagation. A forward pass and a backward pass of all the data is known as an epoch. The iterative process of update of weights through forward and backward propagation for decreasing the loss value is known as the training of a NN.

2.2. Convolutional Neural Nwtwork

[2] defines convolutional neural network (CNN) as a NN that uses convolution operation instead of matrix multiplication. In computer vision problems, the convolution operation is done between an image/image frame and a kernel. The kernel (also known as filters) is a $k \times k$ grid that works as feature/representation generator from the image. An example convolution of a 2×2 kernel on a 4×3 matrix is shown in figure 6.

Convolution has three important properties that help in pattern learning especially for image and video data. They are

- sparse interactions: In a traditional NN that works on the principle of matrix multiplication, the layers have separate parameters for the interactions between input and output units. Thus, every input unit interacts with every output unit. On the other hand, CNN has the filters or weights which are smaller than the input image size. When an image is passed through that filter or weights, the filter extracts only meaningful features or representations that help to learn the pattern. This leads to less memory consumption and faster update of filters because they are small in size.

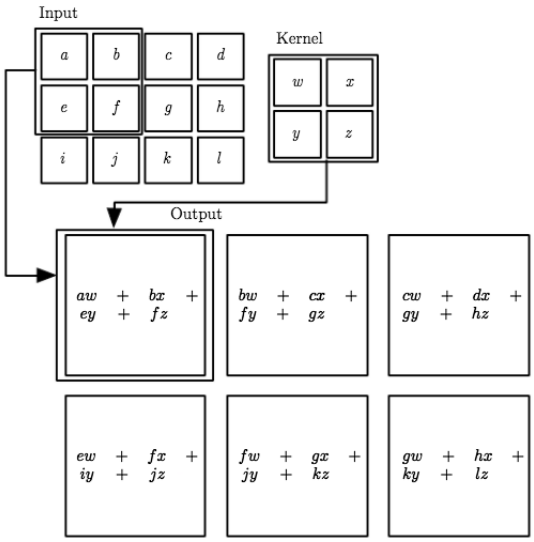

Figure 6: Example of convolution [2]

- parameter sharing: In a NN without any convolution layer, every element of a weight vector corresponds to exactly one element of an input vector. In CNN, the corresponding kernel of size $k \times k$ belongs to the whole image and thus they traverse over the whole image during convolution operation to generate its' representations. Thus, the parameters or weights are shared which reduces memory consumption and instead of having separate weights for every input pixel, the parameters are shared.

- equivariance: A function is said to have equivariance property if its' output changes in the same way as its' input changes. If f is a mapping from one image space $A1$ to $A0$, so that it shifts pixels by n pixels to left using $A1(i, j) = A0(i - n, j - n)$, then applying convolution on $A1$ and then performing the shift will be equivalent to first shifting the pixels and then applying the convolution.

Demonstration of a convolution layer with 3 input channels (R, G , and B of image size $7 \times 7 \times 3$), 2 output channels, and 3×3 kernel size is shown in figure 7.

A CNN layer is usually followed by an activation function to introduce non-linearity and a pooling layer. The pooling layer downsamples the generated representations to a smaller size which is easier to process and takes out features/information that are necessary for designing the decision boundary. The pooling layer outputs a statistical summary of a region of the input grid. The most common form of pooling is max-pooling that outputs the maximum value of a certain grid size. It makes the output invariant to small changes over input. An example of a (2x2) max-pooling is shown in figure 8.

For downsampling the representations, another method of pooling is average pooling. In average pooling, the average value of a $m \times m$ grid is replaced by a single value which is average of all the values in the $m \times m$ grid. Average pooling is less popular than maxpooling and more effected than maxpooling by small changes over grid values. Another method used for downsampling is strided CNN introduced by [6]. In strided CNN, the convolution operation uses a stride more than 1 instead of stride size 1.

CNN architectures usually consist of stacks of convolution layers followed by activation function and pooling layer. At the very end of the stacks of the convolution layers, a fully connected (fc) layer or a global average pooling is introduced which is followed by a softmax layer. This pattern of architecture was popularized by [3]. An example structure of CNN architecture with convolution layers followed by fc layers and softmax is shown in figure 9.

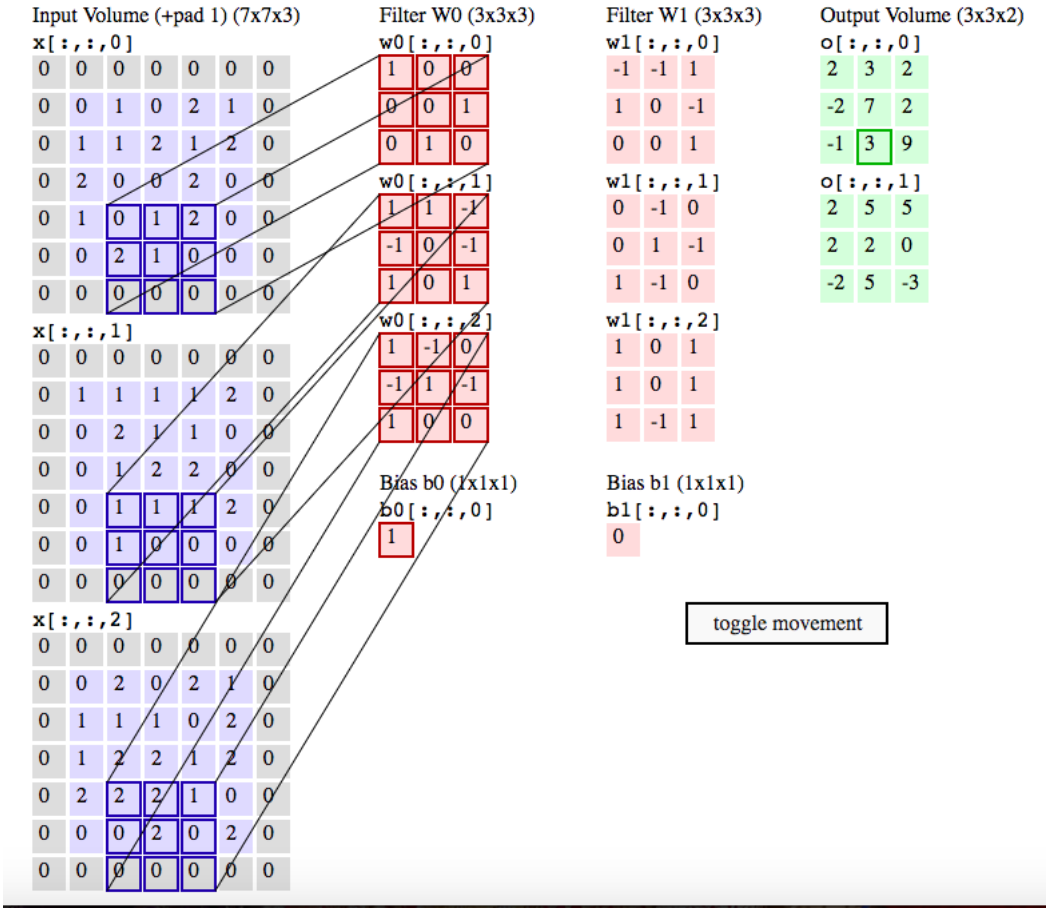

Figure 7: Demonstration of a convolution on a 7 x 7 x3 image at a convolution layer with 3 input channels, 2 output channels, and 3 x 3 kernel size [4]

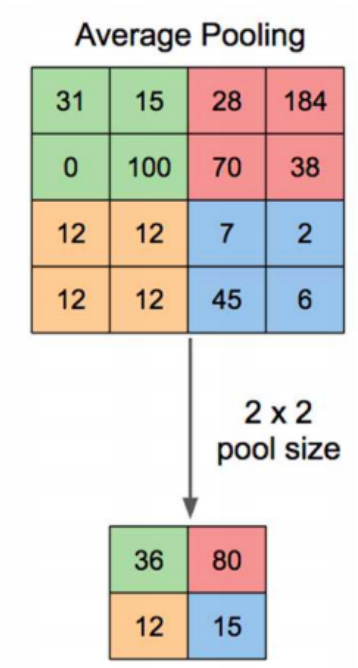

Figure 10: Example of 2 × 2 average pooling on a representation of size 4 × 4 [8]

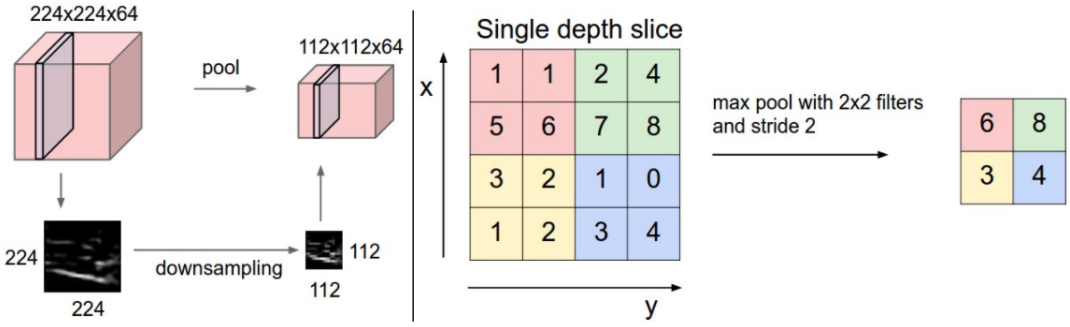

Figure 8: Example of maxpooling [4]

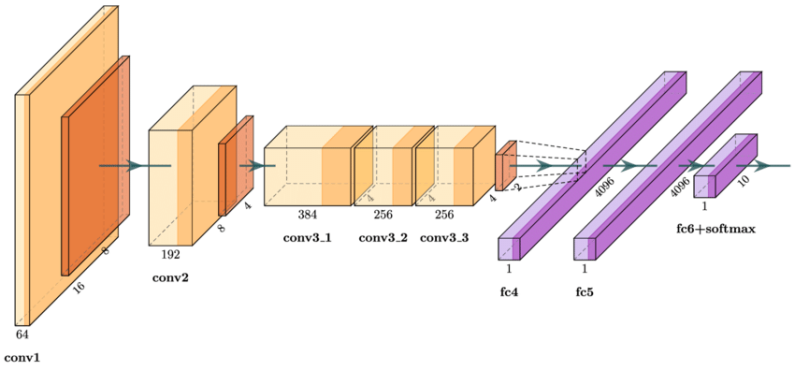

Figure 9: Example of CNN architecture [7]

The typical structure of CNN architecture, shown in figure 9, is not always followed strictly. A popular version of the other architectures is a fully convolutional neural network (FCN) which does not contain any fc layer. FCN usually contains stacks of convolutional layers, activation functions, and downsampling. At the end of the architecture, there is a global average pooling followed by a softmax.

Bibliography

[1] Leonid Datta. A survey on activation functions and their relation with xavier and he normal initialization. *arXiv preprint arXiv:2004.06632*, 2020.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Stanford cs class cs231n: Convolutional neural networks for visual recognition, 2017.

[5] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[6] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[7] Nicola Strisciuglio, Manuel Lopez-Antequera, and Nicolai Petkov. Enhanced robustness of convolutional networks with a push–pull inhibition layer. *Neural Computing and Applications*, pages 1–15, 2020.

[8] Muhamad Yani et al. Application of transfer learning using convolutional neural network method for early detection of terry's nail. In *Journal of Physics: Conference Series*, volume 1201, page 012052. IOP Publishing, 2019.