University of the Aegean

Undergraduate Thesis

Applications of Convolutional Neural Networks in Image and Natural Language Processing

---

Author:
Apostolos Dimoulakis

Supervisor:
Dr. Nicholaos Ampazis

Department of
Financial and Management Engineering

November 28, 2021

Abstract

In this thesis we experimentally investigate the possibility of effectively transferring properties of a Convolutional Neural Network (CNN) to another one, where the latter one is associated with a different type of problem from the problem which the former one solves. Tuning a deep CNN architecture for either image or text classification can become a tedious and time consuming task. So if there is a way in which our hypothesis is valid, then this might be a promising way to facilitate that task. We propose an approach to the CNN training for image and text classification, with the aim to examine that for a given CNN which performs well in one of these two areas, it can perform well in the other area too. We test the performances on their respective tasks accordingly and discuss the effectiveness of such transferences. Two CNN architectures were selected, where each is tuned for a specific image and a text dataset respectively. Initially, in each case, we have transferred only each architecture's properties without taking into account the trainable parameters. We have swapped the tasks for each of these architectures by also changing 1D layers (e.g. 1D convolutional layer) to 2D for the training of the image classifier and the converse for the text classifier. The results of training such modified architectures exhibit the high performance of these models as we demonstrate by extensive experiments.

Additionally, there was a second approach to this problem, only this time we have invoked transfer learning techniques using trained CNNs based on the same tuned architectures. Our experiments have showed that the target CNNs perform as random classifiers, thus our attempt to utilize transfer learning for this purpose has failed. It seems that part of the reason for the poor performance for such transferences, is that the parameters which were trained on the source dataset might not support the training on the target dataset. Furthermore, using 2D layer with small scanning shape (e.g. small filter shape in the case of 2D convolutional layer) on text data, must be contributing to this issue because each embedded vector represents a document's token and producing feature maps based on small parts of such vectors makes little sense.

Περίληψη

Σε αυτή την διπλωματική εξετάζουμε πειραματικά το ενδεχόμενο της αποτελεσματικής μεταφοράς ιδιοτήτων ενός Συνελικτικού Νευρωνικού Δικτύου (ΣΝΔ) σε ένα άλλο ΣΝΔ, όπου το τελευταίο επιλύει ένα πρόβλημα το οποίο διαφέρει από το πρόβλημα που επιλύει το πρώτο ΣΝΔ. Για την κατηγοριοποίηση εικόνας ή κειμένου, η προσαρμογή των υπερπαραμέτρων μίας αρχιτεκτονικής ενός βαθιού ΣΝΔ είναι συνήθως μία κουραστική και δύσκολη διαδικασία. Οπότε, εάν υπάρχει τρόπος μέσω του οποίου η υπόθεσή μας είναι έγκυρη, τότε υπάρχει η δυνατότητα για την διευκόλυνση αυτής της διαδικασίας. Εξετάζουμε αν ένα ΣΝΔ το οποίο είναι αποδοτικό σε ένα από τα δύο αυτά πεδία, είναι εξίσου αποδοτικό στο άλλο πεδίο. Ελέγχουμε τις αποδόσεις στα αντίστοιχα προβλήματα και συζητάμε για την αποτελεσματικότητα αυτών των μεταφορών. Δύο αρχιτεκτονικές ΣΝΔ επιλέχθηκαν, όπου η μία είναι προσαρμοσμένη σε ένα συγκεκριμένο σύνολο δεδομένων με εικόνες και σε ένα με κείμενα αντίστοιχα. Αρχικά, για κάθε περίπτωση, έχουμε μεταφέρει μόνο τις ιδιότητες της αρχιτεκτονικής χωρίς να λάβουμε υπόψη μας τις εκπαιδεύσιμες παραμέτρους. Συγκεκριμένα έχουμε μεταθέσει τα προβλήματα που αντιστοιχίζονται στις δύο αρχιτεκτονικές, με τέτοιο τρόπο ώστε εάν η Αρχιτεκτονική Α αντιστοιχίζεται στο πρόβλημα κατηγοριοποίησης εικόνας και η Αρχιτεκτονική Β αντιστοιχίζεται στο πρόβλημα κατηγοριοποίησης κειμένου, μέσω αυτής της μετάθεσης καταλήγουμε να χρησιμοποιούμε την Αρχιτεκτονική Α για το πρόβλημα κατηγοριοποίησης κειμένου και την Αρχιτεκτονική Β για το πρόβλημα κατηγοριοποίησης εικόνας. Κατά την μετάθεση αυτή, αντικαθιστούμε τις στοιβάδες τύπου 1Δ (π.χ. 1Δ συνελικτική στοιβάδα) με τις αντίστοιχες στοιβάδες τύπου 2Δ για την εκπαίδευση του κατηγοριοποιητή εικόνας καθώς επίσης κάνουμε και το αντίστροφο για την εκπαίδευση του κατηγοριοποιητή κειμένου. Τα αποτελέσματα των εκπαιδεύσεων των μοντέλων που βασίστηκαν σε αυτές τις τροποποιημένες αρχιτεκτονικές δείχνουν πως είναι υψηλής αποδοτικότητας όπως παρουσιάζουμε εκτενώς στα πειράματα.

Επιπρόσθετα, υπήρξε και μία δεύτερη προσέγγιση σε αυτό το πρόβλημα της μεταφοράς ιδιοτήτων από το ένα ΣΝΔ στο άλλο, ωστόσο αυτή την φορά επικαλεστήκαμε τεχνικές του transfer learning χρησιμοποιώντας εκπαιδευμένα ΣΝΔ τα οποία βασίστηκαν στις ίδιες προσαρμοσμένες (tuned) αρχιτεκτονικές. Τα πειράματά μας, δείχνουν πως τα ΣΝΔ στόχοι (target CNNs) αποδίδουν όπως οι τυχαίοι κατηγοριοποιητές, οπότε η χρήση του transfer learning για αυτόν το σκοπό έχει αποτύχει. Φαίνεται πως εν μέρει η κακή απόδοση σε περιπτώσεις τέτοιων μεταφορών, οφείλεται στις παραμέτρους που εκπαιδεύτηκαν στο σύνολο δεδομένων πηγή (source dataset) οι οποίες παράμετροι είναι αισθητά ακατάλληλες για την εκπαίδευση του ΣΝΔ στόχος (target CNN) στο σύνολο δεδομένων στόχο (target dataset). Τέλος, η χρήση στοιβάδας 2Δ με μικρό scanning shape (π.χ. μικρό shape φίλτρου στη περίπτωση της 2Δ συνελικτικής στοιβάδας) σε δεδομένα κειμένου, πρέπει να συμβάλλει σε αυτή την προβληματική συμπεριφορά διότι κάθε embedded διάνυσμα εκπροσωπεί ένα token ενός κειμένου και παράγωντας feature maps βασισμένα σε μικρά μέρη τέτοιων διανυσμάτων δεν έχει πολύ νόημα.

Thesis Committee Members

Nicholaos Ampazis, Ph.D.	Department of Financial and Management Engineering
Supervisor	University of the Aegean
Dimosthenis Drivaliaris, Ph.D.	Department of Financial and Management Engineering
	University of the Aegean
Nikolaos Passalis, Ph.D.	Department of Financial and Management Engineering
	University of the Aegean

Acknowledgements

I would like to express my sincere gratitude to the members of this committee, for their time, their invaluable help, guidance and inspiration during my studies as an undergraduate student, and additionally for their contribution to this undergraduate research project. I also thank everyone else that has supported and believed in me, especially my hardworking parents.

Contents

Introduction **1**

1 Convolutional Neural Networks **3**
1.1 Overview . . . 3
1.2 Application in Image Processing . . . 4
1.2.1 Simple Preprocessing . . . 5
1.2.2 2D Convolutional Layer . . . 5
1.2.3 2D Pooling Layers . . . 11
1.2.4 Image Data Augmentation . . . 12
1.3 Application in Natural Language Processing . . . 14
1.3.1 Preprocessing . . . 15
1.3.2 Word Embedding . . . 17
1.3.3 1D Convolutional Layer . . . 19
1.3.4 1D Pooling Layers . . . 21

2 Hypothesis and Methodology **23**
2.1 Hypothesis . . . 23
2.2 Transfer Learning . . . 23
2.3 Problems and Training Algorithms Part I . . . 25
2.3.1 Training Framework of the Original Image Classifier . . . 25
2.3.2 Training Framework of the Original Document Classifier . . . 26
2.3.3 Switching The Tasks . . . 27
2.3.4 Tables and Diagrams . . . 28
2.4 Problems and Training Algorithms Part II . . . 34
2.4.1 Training Framework of the Source Image Classifier . . . 34
2.4.2 Training Framework of the Source Document Classifier . . . 34
2.4.3 Training Framework of the Target Models . . . 35
2.4.4 Tables and Diagrams . . . 36

3 Experimental Evaluation **42**
3.1 Evaluation Metrics . . . 42
3.2 Datasets . . . 42
3.2.1 The MNIST Dataset . . . 42
3.2.2 The LMRDv1.0 Dataset . . . 43
3.3 Experiments Part I . . . 44
3.3.1 Experiments Based on MNIST's Architecture . . . 44
3.3.2 Experiments Based on LMRDv1.0's Architecture . . . 48
3.3.3 Training Time . . . 52
3.4 Experiments Part II . . . 52
3.4.1 Experiments Based on the `mnistsrc` Source Model . . . 53
3.4.2 Experiments Based on the `imdbsrc` Source Model . . . 55
3.4.3 Training Time . . . 57

4 Discussion **57**

5 Conclusions **59**

Appendices **61**
Appendix I 61
Appendix II 71
Appendix III 73
Appendix IV 89

References **114**

Symbols

$\mathbb{N}$	The formal set of natural numbers, defined such that $\min(\mathbb{N}) = 0$.
$\mathbb{R}$	The set of real numbers.
$T_{\mathbf{p}}(S)$	For a set S, a number $k \in \mathbb{N}$ and $\mathbf{p} \in \mathbb{N}^k$, $T_{\mathbf{p}}(S)$ is the set of k rank tensors with $\mathbf{p}$ shape and elements that belong to the set S.
$\mathbb{D}$	The dataset.
$\mathbb{D}_{\text{tr}}$	The training set.
$\mathbb{D}_{\text{va}}$	The validation set.
$\mathbb{D}_{\text{te}}$	The test set.
$\mathbb{D}_{\text{mtr},s}$	The minibatch s, of instances taken from the training set. s is the number that specifies which exact minibatch is taken from the training set after it is split into multiple minibatches.
m	The total number of instances found on the dataset.
m_{tr}	The total number of instances found on the training set.
m_{va}	The total number of instances found on the validation set.
m_{te}	The total number of instances found on the test set.
n	The total number of instance features.
c	The total number of classes in a classification problem.
$\mathbf{X}$	This is a $(n+1)^{\text{th}}$ rank tensor that carries all of the dataset's instances. Each element is indexed by the vector $(i, \mathbf{j}_{\text{feat}})$ where $i \in \{0, \ldots, m-1\}$ specifies a training instance from the dataset, and $\mathbf{j}_{\text{feat}}$ specifies the feature.
$\mathbf{Y}$	This is a vector that carries all of the dataset's target values.
$\mathbf{W}$	The weight term.
$\mathbf{b}$	The bias term.
L	Total number of layers.
$\mathbf{n}$	The set holding the shapes of each layer's output. Indexing: Layer $l \in \{0, \ldots, L-1\}$ has shape $\mathbf{n}^{[l]}$.
n_{tot}	The set holding the total number of neurons of each layer's output. Indexing: Layer $l \in \{0, \ldots, L-1\}$ has totally $n_{\text{tot}}^{[l]}$ neurons.
$\mathbf{g}$	The set with elements the activation functions. Indexing: Layer $l \in \{0, \ldots, L-1\}$ has activation $\mathbf{g}^{[l]}$.

z The set with elements the output neuron values layerwise, without these neuron values being subjected to their respective activation functions. Indexing: Layer $l \in \{0, \ldots, L-1\}$, with neuron index $\mathbf{j} \in \{0, \ldots, n_0^{[l]} - 1\} \times \cdots \times \{0, \ldots, n_{\texttt{number_of_axes}-1}^{[l]} - 1\}$ corresponds to the linear neuron $z_{\mathbf{j}}^{[l]}$.

a The set with elements the output neuron values layerwise. Has the same indexing rules as **z**.

Slice indexing for a tensor **B** is specified in either its superscript or subscript. When we want to take slices we are free to use the same indexing as the object `ndarray` of `Python`'s `NumPy` library. When we want to specify every element across an axis we prefer using the symbol "$*$" instead of ":". The slice index object, which is a generalized version of the regular tensor index, is still denoted by **j** and determines a specific slice $(\mathbf{B})_{\mathbf{j}}$ or $\mathbf{B}_{\mathbf{j}}$. Also the shape of each tensor is a vector of size equal to its tensor rank k.

Introduction

In our current, highly computerized world, an astonishing amount of digital data consisting of image and text is being generated and processed on a daily basis, which brings about a considerably increasing demand for automation in a wider variety of applications that involve data. Unstructured information that exists inside data holds great potential to elevate technology, industry, science, and prosperity by orders of magnitude. The primary focus of data science is to decode information from a set of data, in order to reveal the underlying nature of a problem or phenomenon. This is usually achieved in conjunction with mathematical models. On the other hand, machine learning overlaps with data science in terms of their purpose and the tools that are used to achieve it. Machine learning is the art of designing systems which are capable of learning from data with minimal human intervention. These systems are able to transform a finite set of digital data items, into prediction models which, through their *knowledge*, are able to perform robustly in new data which hasn't been exploited before in any direct way by these trained models. This is similar to what inferential statistics is concerned with, in the sense that information is pulled from a sample, with the goal of making predictions about a wider population in the context of a given task or problem. Additionally, machine learning is also taking into account the available computational resources for the training process. Also the measurement of the predictive potential, which is more formally known as the trained model's generalizability, is an inseparable part of the field. Sacrificing performance for more data interpretability is usually not a matter of primary concern, which contrasts the purpose of statistics, in which data interpretability is deemed necessary. Of course the measurement and actual representation of data in a good enough digital format is essential for the final model's performance.

In deep learning, which is a subfield of machine learning, models are capable of learning complex patterns from large pools of data, where many of these patterns are undetectable by humans and regular measurement instruments. These models are able to break down a task into multiple simpler subtasks, and take upon the completion of these subtasks instead. These subtasks are parameterized by the data during the training process of the model. The convolutional neural network is a type of one such a deep learning model, which has been proven to be incredibly robust in image and text related problems. Able to be trained on diverse and large pools of image or text data, CNNs are capable of learning extremely complex patterns from these types of data. However, the setup of the training process for deep CNNs is a difficult and time consuming task.

We will firstly establish the notion of CNNs. Then we will explain how a CNN can become the right tool for either image or text classification. For the examination of the hypothesis, we will utilize two CNN architectures: One tuned for image classification on a dataset consisting of image data, and the other tuned for text classification on a dataset consisting of text data. In order to check if the hypothesis is true for each one of these architectures, we will exchange the problems they were tuned to solve,

with some necessary adjustments to the architecture and the dataset preprocessing. In each case, we will train the respective CNNs and then we will evaluate their effectiveness by interpreting the evaluation metrics for each of these models.

It is assumed that the reader is familiar with some basic theory of linear algebra, matrix operations like matrix multiplication, multivariate differential calculus along with the Python programming language along with its NumPy, Keras and TensorFlow libraries. Also being familiar with probability theory and statistics is a plus. Chapter 1 offers some foundational knowledge for the reader's mathematical perspective and intuition about CNNs. The reader that is familiar with CNNs along with text and image data may omit chapter 1. In the other chapters we discuss the hypothesis and experiments. The code applied for the experiments is presented in the appendices at the end of this thesis.

1 | Convolutional Neural Networks

In this chapter we will set the theoretical basis regarding the architecture of *Convolutional Neural Networks*. Firstly we make a brief overview about these models, then we specify them for image classification, and then for document classification.

1.1 | Overview

Recent developments of the **Convolutional Neural Network** (**CNN**) [1], have led into major breakthroughs in image and natural language processing tasks [2]. These breakthroughs would not be possible without the usage of processing units capable of a high number of parallel operations, and without our access to vast amounts of image and text data. This type of neural network was inspired by the visual cortex of animals [3].

The CNN is a special type of feedforward neural network (FNN), which is associated with the **convolutional layer**. The convolutional layer is a trainable layer that is based on the operation of *2D discrete convolution*, as the linear map of the layer's input. This layer is not fully connected like the dense layer, in the sense that not every input neuron of the layer necessarily participates in the computation of every neuron in its output. It's also capable of extracting spatial information in a more direct way from the layer's input tensor. The computation of each output neuron is a mapping of a specific group of neurons also called the **receptive field**, whose neurons happen to be closer together inside the input tensor of the convolutional layer. The mapping of the output tensor is the result of the convolution between the input tensor and a weight tensor, which holds the trainable parameters of the convolutional layer. If the receptive fields were to be translated in different parts of the same input tensor, it would have detrimental influence in the way that the tensor is processed by the convolutional layer [4, 5]. This property is also referred to as the **translation invariance**, which is profoundly a consequence of the convolution operation. Additionally, features that are more far apart inside the input tensor don't influence one another as much (if at all), this is the **locality** property [6]. Compared to the training of densely connected FNN, the training of a CNN on image and text data, typically occupies lower computational resources and comes with a lower exposure to the curse of dimensionality and overfitting, that can result in higher performance models in reduced training time, given that the trainings are computed by processing units, capable of parallel computation (e.g. GPUs, TPUs). The convolutional layer may be considered as a regularized version of the dense layer due to the partial connection property.

The architectures of CNNs that are intended to be used for a classification task, can be seen as pipelines of 3 vital components whose purpose is different from one another. These are the *feature extractor*, the *flattening* and *classifier* components (see Figure 1), where each of these is consisted of a stacked group of layers, except from the flattening component which only contains one layer. To describe the role of

each one:

- **Feature Extractor Component**: Its role is to reduce the input tensor into a more refined representation, with minimal disposal of spatial information. The convolutional layer is an important part of this component. Some of the appropriate types of layers used for this component are the *2D Convolutional*, *2D Pooling* and the *Dropout* layers.
- **Flattening Component**: Containing exactly one layer with index $L_f - 1$, the **flattening layer**, which simply stretches any input tensor into a vector.
- **Classifier Component**: This is a dense FNN. The flattening layer is its input, and the CNN's output layer is the *classifier component*'s output layer.

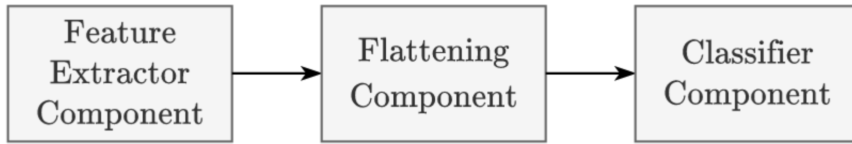


Figure 1: A high abstraction of the CNN's architecture or pipeline.

1.2 | Application in Image Processing

Any coloured *image*, can be represented by a 3 rank tensor $\mathbf{x}$ with shape

$$(\texttt{height},\ \texttt{width},\ \texttt{channels})$$

where the final axis is the *channel* axis. In this thesis we will use the **RGB** *additive color model* [7], in which the total number of channels is 3, which are sorted as Red, Green and Blue. Each **channel** corresponds to the intensity of a unique color among `channels` unique colors. So the values of the matrix $\mathbf{x}_{*,*,0}$ correspond to intensities of the red color of the image, values of $\mathbf{x}_{*,*,1}$ to intensities of green color and values of $\mathbf{x}_{*,*,2}$ to intensities of the blue color for each position $(\texttt{height},\ \texttt{width})$ of an image. Images that are gray are **grayscale** images, and these can be represented by a plain matrix or a tensor of shape $(\texttt{height},\ \texttt{width},\ 1)$.

An image **pixel** is the smallest possible part of a colored image located in some position $(\texttt{height},\ \texttt{width})$, which is also a vector of size `channels` and may be expressed as $\mathbf{x}_{\texttt{height},\texttt{width},*}$. Higher values of a pixel element correspond to higher intensity to the respective channel on that pixel. On the *feature extractor component* a layer's l output, is a tensor with shape $(n_0^{[l]}, n_1^{[l]}, n_2^{[l]})$, where $n_0^{[l]}$ is the height, $n_1^{[l]}$ is the width and $n_2^{[l]}$ is the number of channels.

Training on image datasets often requires a large amount of data due to the larger number of features per image instance. The dense FNN can be effective for image processing tasks, however CNNs have proven to be far better due to the good properties that the convolutional layer has on ordered arrays such as images.

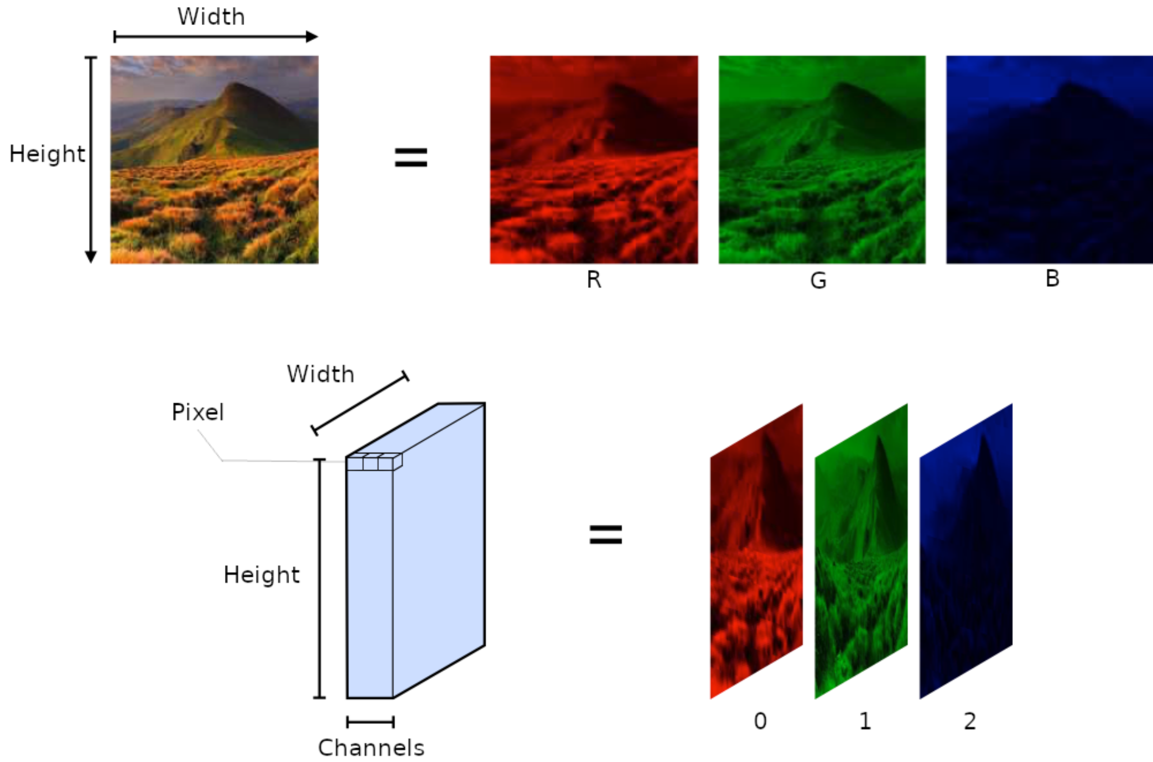


Figure 2: This image was taken from [8]. It shows how an RGB image can be seen by segmenting it into its respective color channel matrices.

1.2.1 | Simple Preprocessing

Given the feature $\mathbf{X}$ and target $\mathbf{y}$ tensors of a raw instance set S, firstly we need to make sure that each of the images denoted by $\mathbf{X}_{i,*}$ or $\mathbf{X}_{i,*,*,*}$ with index $i \in \{0,\ldots,p-1\}$, needs to have the same exact shape. If the height and width differs per image then we need to somehow downsample each image channel into some predetermined height and width say $n_0^{[0]}$ and $n_1^{[0]}$ respectively.

1.2.2 | 2D Convolutional Layer

In the case of image data, the convolutional layer l accepts a 3 rank tensor $\mathbf{a}^{[l-1]}$ where the final axis determines the channel. Now like in the case of the linear mapping of the dense layer, the linear mapping of the 2D convolutional layer has some weight $\mathbf{W}^{[l]} \in T_{k_0^{[l]},k_1^{[l]},n_2^{[l-1]},n_2^{[l]}}(\mathbb{R})$ and bias $\mathbf{b}^{[l]} \in T_{n_0^{[l]},n_1^{[l]},n_2^{[l]}}(\mathbb{R})$ terms. Given an output channel $\nu_2^{[l]} \in \{0,\ldots,n_2^{[l]}-1\}$, the term $\mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}}$ is called the $\nu_2^{[l]}$-th **filter** of the layer, or the layer's **output channel**. Each filter corresponds to a different channel of the layer's output. The natural numbers $k_0^{[l]}$ and $k_1^{[l]}$ are the height and width of the filters, and these two variables along with $n_2^{[l]}$ are crucial hyperparameters for that layer. The *multichannel convolution* operation $\circledast$ is the convolution map between the input $\mathbf{a}^{[l-1]}$ and the $\nu_2^{[l]}$-th filter $\mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}}$, and this mapping is expressed as $\mathbf{a}^{[l-1]} \circledast \mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}}$. For

any given output height $\nu_0^{[l]}$, output width $\nu_1^{[l]}$ and input channel $\nu_2^{[l-1]}$, this binary operation produces

$$(\mathbf{a}^{[l-1]} \circledast \mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}})_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l-1]}} = \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{k_1^{[l]}-1} (a^{[l-1]}_{\nu_0^{[l]}+\kappa_0^{[l]},\nu_1^{[l]}+\kappa_1^{[l]},\nu_2^{[l-1]}} \cdot W^{[l]}_{\kappa_0^{[l]},\kappa_1^{[l]},\nu_2^{[l-1]},\nu_2^{[l]}}) \quad (1.1)$$

where

$$n_j^{[l]} = n_j^{[l-1]} - k_j^{[l]} + 1 \ (\forall j \in \{0,1\}) \ . \quad (1.2)$$

The linear output of this layer is defined as,

$$\begin{aligned} z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} &= \sum_{\nu_2^{[l-1]}=0}^{n_2^{[l-1]}-1} (\mathbf{a}^{[l-1]} \circledast \mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}})_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l-1]}} + b_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \\ &= \sum_{\nu_2^{[l-1]}=0}^{n_2^{[l-1]}-1} \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{k_1^{[l]}-1} (a^{[l-1]}_{\nu_0^{[l]}+\kappa_0^{[l]},\nu_1^{[l]}+\kappa_1^{[l]},\nu_2^{[l-1]}} \cdot W^{[l]}_{\kappa_0^{[l]},\kappa_1^{[l]},\nu_2^{[l-1]},\nu_2^{[l]}}) + b_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \ . \end{aligned} \quad (1.3)$$

Sometimes we want to skip some rows and columns from the convolution operation to reduce the computations done or to add some regularization effect. One way to do this is through the **strided convolution** operation [9]. This operation is determined by two additional hyperpameters. These are the **height stride** $s_0^{[l]} (s_0^{[l]} \in \mathbb{N}\backslash\{0\})$, and the **width stride** $s_1^{[l]} (s_1^{[l]} \in \mathbb{N}\backslash\{0\})$. The symbol of the strided convolution is now denoted by $\circledast_{s_0^{[l]},s_1^{[l]}}$ or $\circledast_{\mathbf{s}^{[l]}}$ (where $\mathbf{s}^{[l]} = (s_0^{[l]}, s_1^{[l]})$) and is a generalization of the convolution defined by (1.1) where $\circledast_{1,1} = \circledast$. The height stride determines how many rows will be skipped for each convolution iteration on each individual height vs width slice of the layer's input (it is demonstrated in Figure 3). Additionally, to control how these convolution operations are handling the elements with indices that are close to the boundaries of the input tensor, and to be able to prevent losing the rightmost and leftmost $\lfloor k_0^{[l]}/2 \rfloor$ dimensions, along with the upmost and downmost $\lfloor k_1^{[l]}/2 \rfloor$ dimensions from the layer's output tensor per channel, we can surround the layer's input tensor per channel with $p_0^{[l]} (p_0^{[l]} \in \mathbb{N})$ rows and $p_1^{[l]} (p_1^{[l]} \in \mathbb{N})$ columns of 0 values [9]. This is the **padded** input tensor and we denote it by $\mathbf{a}^{[l-1],\text{pad}}$. The analogous is true for the width stride for the columns on the same matrices. Thus, now (1.3) is computed by

$$\begin{aligned} z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} &= \sum_{\nu_2^{[l-1]}=0}^{n_2^{[l-1]}-1} (\mathbf{a}^{[l-1],\text{pad}} \circledast_{s_0^{[l]},s_1^{[l]}} \mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}})_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l-1]}} + b_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \\ &= \sum_{\nu_2^{[l-1]}=0}^{n_2^{[l-1]}-1} \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{k_1^{[l]}-1} (a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s_0^{[l]}+\kappa_0^{[l]},\nu_1^{[l]}\cdot s_1^{[l]}+\kappa_1^{[l]},\nu_2^{[l-1]}} \cdot W^{[l]}_{\kappa_0^{[l]},\kappa_1^{[l]},\nu_2^{[l-1]},\nu_2^{[l]}}) + b_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \ . \end{aligned} \quad (1.4)$$

$(\forall \nu_j^{[l]} \in \{0, \ldots, n_j^{[l]} - 1\})(\forall j \in \{0, 1, 2\})$ where

$$n_{j'}^{[l]} = \left\lfloor \frac{n_{j'}^{[l-1]} + 2 \cdot p_{j'}^{[l]} - k_{j'}^{[l]}}{s_{j'}^{[l]}} + 1 \right\rfloor (\forall j' \in \{0, 1\}) \ . \tag{1.5}$$

$\mathbf{z}^{[l]}_{*,*,\nu_2^{[l]}}$ for some $\nu_2^{[l]}$, is also called the **feature map** of the convolutional layer l. Also in (1.4) we will use sometimes $C_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}$ to refer to the $\sum_{\nu_2^{[l-1]}=0}^{n_2^{[l-1]}-1}(\mathbf{a}^{[l-1],\text{pad}} \circledast_{s_0^{[l]},s_1^{[l]}} \mathbf{W}^{[l]}_{*,*,*,\nu_2^{[l]}})_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l-1]}}$ quantity. And to make the feature maps nonlinear, we use a nonlinear activation $\mathbf{g}^{[l]}$ to produce the output $\mathbf{a}^{[l]}$. We can see how the 2D convolutional layer works from Figure 4. Additionally, the example of 2D convolution presented in Figure 3, demonstrates the strided convolution operation of the 2D convolutional layer with index say l for a specific padded input $\nu_2^{[l-1]} \in \{0, \ldots, n_2^{[l-1]} - 1\}$ and output $\nu_2^{[l]} \in \{0, \ldots, n_2^{[l]} - 1\}$ channels. The strides here are $s_0^{[l]} := 2, s_1^{[l]} := 2$, and the paddings are $p_0^{[l]} := 1$ and $p_1^{[l]} := 1$. The matrix represented by the blue array of numbers is $\mathbf{a}^{[l-1]}_{*,*,\nu_2^{[l-1]}}$ with dimensions $n_0^{[l]} := 5$ and $n_1^{[l]} := 5$, which is surrounded by the padding and the matrix represented by the darker array with shape $(3, 3)$ overlapping with the padded input, is the filter $\mathbf{W}^{[l]}_{*,*,\nu_2^{[l-1]},\nu_2^{[l]}}$ where $k_0^{[l]} := 3$ and $k_1^{[l]} := 3$. Each step produces an output neuron, via element wise multiplication between values of the filter and the padded input matrix, where the elements of the resulting matrix are added together to produce the respective output neuron's signal. We start from up left (Step 1) and go towards the right, skipping $s_1^{[l]} - 1$ columns each time, until we exhast all columns (Step 3). Then we return to the leftmost area again and we shift downwards one time by skipping $s_0^{[l]} - 1$ rows (Step 4). The same procedure applies on this group of rows, again going towards the right side, and then again we go downwards by skipping again $s_0^{[l]} - 1$ rows. This happens until the entire padded matrix is exhausted. The output neuron's location in the output tensor is determined by the overlapping of the filter with the padded input.

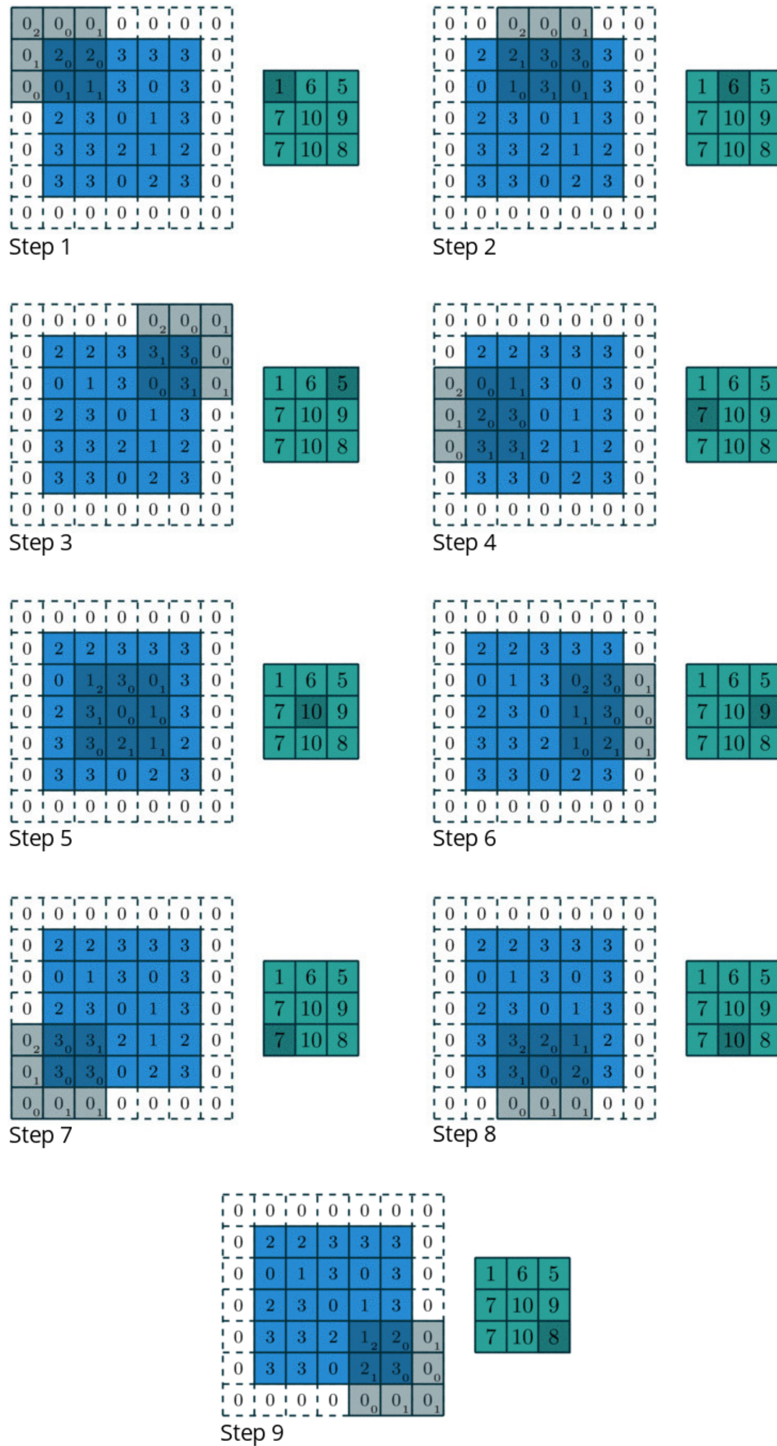


Figure 3: The image was taken from [10]. This is a demonstration of a strided 2D Convolution per filter for a padded input matrix.

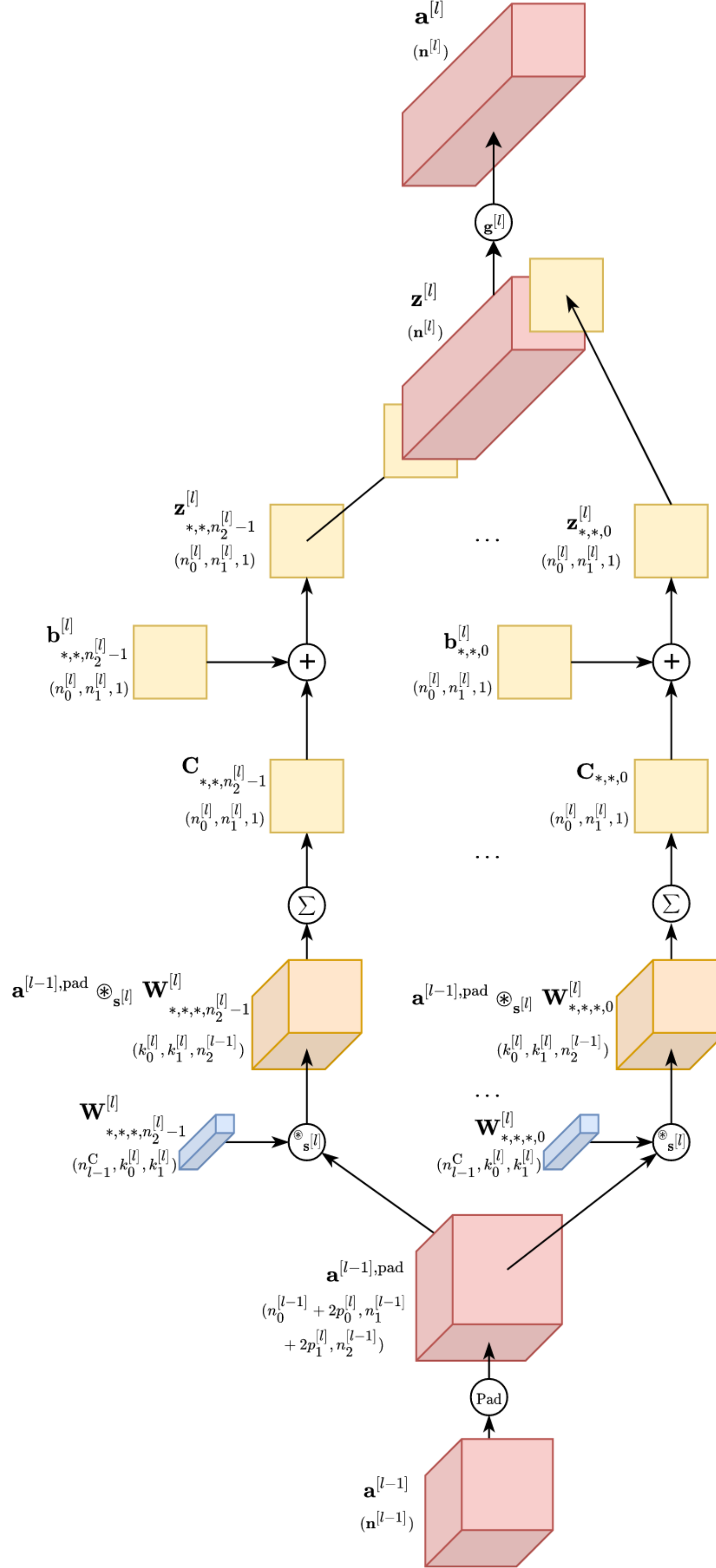


Figure 4: The pipeline of the 2D convolutional layer.

The weight parameters that need to be learned are the filters that are convolved

around the image. During the training, as these filters are in a state of constant adjustment in each epoch for a better performing model, they become more able to extract useful information about the image, and pass that information to the next layer. In the first convolutional layers that are closer to the input image, the filters may improve at sharpening the image for edges with a specific angle or direction. As the image is processed by deeper convolutional layers, the kernels extract more complex patterns, and every time these patterns are associated with larger areas of the input image. Due to the translation invariance property, the convolutional layer does not take into account the position of local areas inside the image. This is a useful property in cases where the object of interest inside an image, should be free to translate wherever possible. At least for CNNs applied for image classification tasks, this property is normally useful [4, 5]. For a task of image classification between boats and cars, it would be expected that the background of most of the boat images would mostly contain water areas and the background of car images would mostly consist of road areas. Assume that a dense FNN and a CNN were trained on the same dataset until the same training accuracies were reached. Let an image that was not included in the training set, include a boat very similar to other boats of the training set, but instead of being located on top of water, the boat is located in the middle of a road with no cars. A dense layer of the dense FNN would be prone to pass to the next layers more irrelevant information about the relation of the object of interest, which is the boat, with the background. This irrelevant information would contribute to larger errors which means that the classifier might fail to classify the image correctly. On the other hand, the convolutional layer of a trained CNN on the same dataset, would not pass as much irrelevant information about the location of the boat inside the image. The pixels closer to the boat would surely be taken into account by the convolutional layer of the CNN, but the dense layer of the dense FNN would be free to take into account every pixel in that image to determine the entire stack of neurons in the output. Thus, the dense FNN would be more prone to overfit and it would surely need a larger training set in order to additionally learn how to ignore the location of the object of interest inside an image whereas, the CNN would naturally not need this additional training, because the translation invariance property is hard-wired on the 2D convolutional layer, while the dense FNN has to learn this property from scratch by the provided training set. Consequently, the translation invariance property of the convolutional layer, gives CNNs the advantage, as it reduces the need of training on a larger training set.

Using small shapes for each filter especially in the initial convolutional layers, results in a drastic reduction in the total number of parameters, while the representations or output tensors of these convolutional layers become translation invariant and our layer only incorporates local information, when determining the value of each hidden activation [6]. Smaller filter shapes mean that the convolutional layer will capture smaller, more simple features that do not vary much between other images containing that same object. This is good because the convolutional layer is less prone to capture

high variation[1] patterns which is usually accompanied by noise. Compared to a dense layer that produces an output tensor of the same size, this is a smart way that effectively reduces the total number of parameters and decreases the total training time.

For $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x}$ is an image in the RGB format, and l the index of the convolutional layer, the error term from, [11] becomes

$$\boldsymbol{\delta}^{[l]} = \sum_{\nu_2^{[l+1]}=0}^{n_2^{[l]}-1} \sum_{\nu_1^{[l+1]}=0}^{n_1^{[l]}-1} \sum_{\nu_0^{[l+1]}=0}^{n_0^{[l]}-1} (\delta^{[l+1]}_{\nu_0^{[l+1]},\nu_1^{[l+1]},\nu_2^{[l+1]}} \cdot \nabla_{\mathbf{z}^{[l]}} h^{[l+1]}_{\nu_0^{[l+1]},\nu_1^{[l+1]},\nu_2^{[l+1]}}(\mathbf{g}^{[l]}(\mathbf{z}^{[l]}))) \tag{1.6}$$

where if $\nu_0^{[l]} + p_0^{[l+1]} - \nu_0^{[l+1]} \cdot s_0^{[l+1]} \in \{0, \ldots, k_0^{[l+1]}\}$ and $\nu_1^{[l]} + p_1^{[l+1]} - \nu_1^{[l+1]} \cdot s_1^{[l+1]} \in \{0, \ldots, k_1^{[l+1]}\}$ it's

$$\begin{aligned}&\frac{\partial}{\partial z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}} h^{[l+1]}_{\nu_0^{[l+1]},\nu_1^{[l+1]},\nu_2^{[l+1]}}(g^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}(z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}})) = \\ &\frac{\partial}{\partial z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}} g^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}(z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}) \cdot W^{[l+1]}_{\nu_0^{[l]}+p_0^{[l+1]}-\nu_0^{[l+1]}\cdot s_0^{[l+1]},\nu_1^{[l]}+p_1^{[l+1]}-\nu_1^{[l+1]}\cdot s_1^{[l+1]},\nu_2^{[l]},\nu_2^{[l+1]}}\end{aligned} \tag{1.7}$$

else it's

$$\frac{\partial}{\partial z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}} h^{[l+1]}_{\nu_0^{[l+1]},\nu_1^{[l+1]},\nu_2^{[l+1]}}(g^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}(z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}})) = 0 \tag{1.8}$$

$$(\forall \nu_0^{[l]} \in \{0, \ldots, n_0^{[l]} - 1\})(\forall \nu_1^{[l]} \in \{0, \ldots, n_1^{[l]} - 1\})(\forall \nu_2^{[l]} \in \{0, \ldots, n_2^{[l]} - 1\})$$
$$(\forall \nu_0^{[l+1]} \in \{0, \ldots, n_0^{[l+1]} - 1\})(\forall \nu_1^{[l+1]} \in \{0, \ldots, n_1^{[l+1]} - 1\})(\forall \nu_2^{[l+1]} \in \{0, \ldots, n_2^{[l+1]} - 1\})$$

The loss gradients [12] become

$$\nabla_{\mathbf{W}^{[l]}} \mathcal{L}_{\{(\mathbf{x},\mathbf{y})\}}(\hat{\mathbf{y}}) = \sum_{\nu_2^{[l]}=0}^{n_2^{[l]}-1} \sum_{\nu_1^{[l]}=0}^{n_1^{[l]}-1} \sum_{\nu_0^{[l]}=0}^{n_0^{[l]}-1} (\delta^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \cdot \nabla_{\mathbf{W}^{[l]}} h^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}(\mathbf{a}^{[l-1]})) \tag{1.9}$$

$$\nabla_{\mathbf{b}^{[l]}} \mathcal{L}_{\{(\mathbf{x},\mathbf{y})\}}(\hat{\mathbf{y}}) = \sum_{\nu_2^{[l]}=0}^{n_2^{[l]}-1} \sum_{\nu_1^{[l]}=0}^{n_1^{[l]}-1} \sum_{\nu_0^{[l]}=0}^{n_0^{[l]}-1} (\delta^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} \cdot \nabla_{\mathbf{b}^{[l]}} h^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}}(\mathbf{a}^{[l-1]}))\ . \tag{1.10}$$

1.2.3 | 2D Pooling Layers

One way to deal with large numbers of computations using a convolutional layer is through the usage of **pooling layers** [13], in layers before that convolutional layer.

[1] Variation between different images.

Two of the most common ones are the **max-pooling** and the **average-pooling** layers. Pooling layers are similar to the convolutional layer in the sense that they too scan the image in the same way as the convolutional layer, but instead this time we have a window that scans an image slice. 2D pooling layers can also have stride and padding. The height and width of the window is $k_0^{[l]}$ and $k_1^{[l]}$ respectively. We use the same notation for stride and padding as in the case of the 2D convolutional layer.

The max pooling takes $\mathbf{a}^{[l-1]}$, which is a tensor of size $(n_0^{[l-1]}, n_1^{[l-1]}, n_2^{[l-1]})$ and converts it into the output tensor $\mathbf{a}^{[l]}$ which is

$$\begin{aligned} a^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} = \max(\{a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s_0^{[l]}+\kappa_0^{[l]},\nu_1^{[l]}\cdot s_1^{[l]}+\kappa_1^{[l]},\nu_2^{[l-1]}}(\kappa_0^{[l]} \in \{0,\dots,k_0^{[l]}-1\}, \\ \kappa_1^{[l]} \in \{0,\dots,k_1^{[l]}-1\})\}) \ . \end{aligned} \tag{1.11}$$

On the other hand, the average pooling layer converts the input into

$$z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]},\nu_2^{[l]}} = \frac{1}{k_0^{[l]}\cdot k_1^{[l]}} \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{k_1^{[l]}-1} a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s_0^{[l]}+\kappa_0^{[l]},\nu_1^{[l]}\cdot s_1^{[l]}+\kappa_1^{[l]},\nu_2^{[l-1]}} \ . \tag{1.12}$$

In either case, the resulting output tensor has shape

$$\left(\left\lfloor \frac{n_0^{[l-1]} + 2\cdot p_0^{[l]} - k_0^{[l]}}{s_0^{[l]}} + 1\right\rfloor, \left\lfloor \frac{n_1^{[l-1]} + 2\cdot p_1^{[l]} - k_1^{[l]}}{s_1^{[l]}} + 1\right\rfloor, n_2^{[l-1]}\right) \ . \tag{1.13}$$

Note that the new height and width are computed in the same way as in the case of the 2D convolutional layer.

The max pooling layer can lead to faster convergence, select superior features and improve generalization [14]. Thus we can use this type of layer to reduce the overfitting behaviour of trainings, while reducing the total training time as the number of output neurons is decreased by this mapping. Additionally, max pooling is better compared to other downsampling methods such as average pooling [14].

1.2.4 | Image Data Augmentation

As we have already seen, the convolutional layer is translation invariant, which means that moving the object of interest in an image does not make much of a difference on where a CNN *sees* that object, for the classification. However, there are more image transformations of the training set that can correspond to the same class, which implies that the overfitting phenomenon still has ways to create training issues. If there are indications of overfitting in the trainings on the given training set, we can collect additional external instance images for the training set to improve training and resolve part of the overfitting issue. Finding additional images may prove to be difficult so we can follow other practices to resolve overfitting like regularizing the training. An alternative choice we have, is the option to create more instances by

copying instances of the training set, and then making some minor adjustments to them, depending on the task.

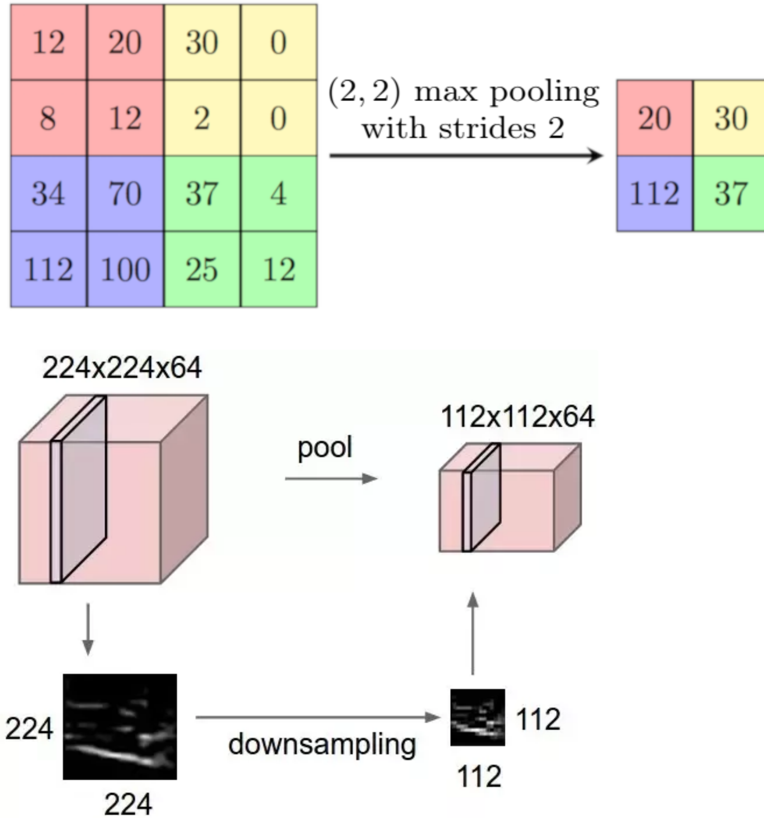


Figure 5: This image was taken from [15], showing the operation of 2D max pooling on two different example inputs.

Such techniques are referred to as **data augmentation** techniques and they have proven to be a reliable solution to image related tasks [16]. Such image transformations may be a random rotation, flipping, color intensity change, cropping, injecting noise etc. [16]. After the data augmentation, all these instance transformations can be included to the training set, thus now having additional, slightly diverse instances, that force the model to be less sensitive to the respective augmentation transformations.

Rotation Augmentation

To describe how data augmentation can be implemented in the case of random image rotations, assuming we have a training instance, say $(\mathbf{x}, y)$ where $\mathbf{x}$ is a given grayscale image in the form of a matrix with shape $(\texttt{height}, \texttt{width})$, we firstly need to define a maximum angle for the rotation tranformation, say $\theta_{\max}(\theta_{\max} > 0)$ (in rads) and then we sample a random angle θ from the continuous uniform distribution $\mathcal{U}(0, \theta_{\max})$ and we rotate x by θ rads. It is advised by [16] for $\theta_{\max}$ to be small. This time we will use a matrix with different indexing whose origin is located on $(\lfloor\texttt{height}/2\rfloor, \lfloor\texttt{width}/2\rfloor)$

when we look at the same image $\mathbf{x}$. Assume that new matrix is $\boldsymbol{\xi}$, so we have

$$x_{\alpha+i,\beta+j} = \xi_{i,j} \tag{1.14}$$

where $\alpha = \lfloor \texttt{height}/2 \rfloor$ and $\beta = \lfloor \texttt{width}/2 \rfloor$, $(\forall i \in \{0, \dots, \texttt{height} - 1\})$ $(\forall j \in \{0, \dots, \texttt{width} - 1\})$. The new index of an image pixel that belongs to $\boldsymbol{\xi}$ with index (i, j) is

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \tag{1.15}$$

where i' and j' are rounded to the closest integers after this operation. A simple way to avoid problems with exceeding index values, is to just dispose of the pixels corresponding to them, and replace empty pixels with 0 values. Another simple workaround is to surround all of these images with a common padding of zeros, so that the resulting image $\mathbf{x}$ will have height and width equal to $\lceil (\alpha^2 + \beta^2)^{1/2} \rceil$.

1.3 | Application in Natural Language Processing

The field of **natural language processing** (NLP) is concerned with the interactions between computers and humans through *natural language* [17]. **Sentiment analysis** is the field of study that analyzes people's opinions, sentiments, evaluations, attitudes, and emotions from written language [18]. This thesis will be limited in the case of **document classification** tasks using sentiment analysis techniques, in which a machine learning model has to classify **documents**. CNNs have proven to be capable of performing well on NLP tasks [19, 20, 21], including document classification tasks [22].

The representation of an instance's document is the feature $\mathbf{x}$ and an initial raw form of this representation may be stored in a machine's memory as a string, say

```
"They␣arrived␣late,␣and␣they␣couldn't␣find␣good␣seats!"
```

where the "␣" is set to be identical to the *space* character, which acts as a visible separator of words in sentences. A document holds some meaning that needs to be decoded in correspondence with the NLP task. That initial representation of each feature-document is incompatible with the input of an FNN, as FNNs accept tensors of continuous numbers and not strings. This means that it is imperative that each document string is firstly properly mapped into a tensor of numbers in a similar way as images are encoded, before being processed by such a model. The layers used in the *feature extractor component* of a CNN's architecture in the case of images, can be applied to sequence data like documents[2] with some minor modifications. We refer to these layers by 1D or 1 dimensional layers, because sequence data is scanned through one axis instead of two like in the case of image data. Additionally we'll introduce an optional, but extremely useful layer called the *embedding layer* (see Figure 6).

[2] Documents can be seen as sequences of words.

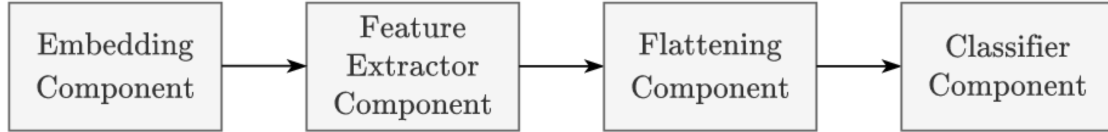


Figure 6: A high abstraction of the CNN's architecture or pipeline with the embedding component which is defined to contain the embedding layer only.

1.3.1 | Preprocessing

In order to use a FNN for document classification, we need to find a numerical representation of an instance's document $\mathbf{x}$, with *length* which will be shared with the other documents as well. In order to map any document as such, we invoke the following preprocessing rules.

Tokenization

Each document's string can be broken down into a vector containing a number of fundamental elements called **tokens**. A document's token may be defined to be an individual character. So we can represent the document as a vector of these tokens, while maintaining the ordering of the tokens found in that document string. However in this thesis we define tokens to be the words of a document, instead of individual characters. A document's size or length, denoted by `doc_length`, is defined to be the multitude of tokens it is associated with. Each instance's document in its raw form, may vary in size and is consisted of alphanumeric characters, along with other potential symbols that are used in sentences like punctuation marks (e.g. "`.`", "`,`", "`!`", "`;`"). We will use an approach similar to the one shown in [23] for the *document tokenization*. Each word in a document is separated by the space character "␣" between them, so the $\mathbf{x}$ of our previous example now becomes the vector of word strings

("They", "arrived", "late", "and", "they", "couldn't", "find", "good", "seats!") .

Uppercase alphabetical characters offer detrimental information so they are substituted by their respective lowercase and assuming that the special symbols do not significantly contribute to the meaning of a document, we are free to drop them as well. Therefore $\mathbf{x}$ now becomes

("they", "arrived", "late", "and", "they", "couldnt", "find", "good", "seats") .

Vocabulary

Looking at a training set's tokenized documents, most of the documents' words repeat themselves. The vocabulary [24] is denoted by `vocab` and is defined to be a vector. Every element of this vector corresponds to a unique word's string found in the training set's tokenized document representations. The words inside `vocab` are sorted

in a decreasing order by the frequency of their appearances inside the training set. So the elements corresponding to initial indices of `vocab` are the more frequent ones, while the elements corresponding to later indices are the less frequent ones. For example `vocab` may look like this (showing only the first 4 elements)

("the", "a", "and", "of", ...)

meaning that "the" is the most frequent word found in the training set, "a" is the second most frequent word etc. Furthermore, an additional symbol will be used for the padding of each document, we define that to be "<PAD/>", and include it as a special *word* in the `vocab`'s zeroth position. Additionally an external document may contain words that do not exist inside the `vocab`, in this case we can substitute these with "<UNK/>". So the above `vocab` example becomes

("<PAD/>", "<UNK/>", "the", "a", "and", "of", ...) .

Thus now we can represent every unique word with a unique natural number, so the tokenized document example may be encoded as

(32, 4407, 511, 4, 32, 20799, 164, 49, 6617)

where each number is an index of `vocab` that points to the corresponding document words located inside the `vocab` vector. For example, "they" has index 32, meaning that it is located on the 32-th position inside `vocab`. We can drop some of the rarest words found in the training set to reduce complexity [24]. The number of words we hold is set to be the first, or most frequent `max_words` words. For each removed word from the vocabulary, the removed word inside each document is substituted by the value 1 (the encoding of "<UNK/>") in the training set documents. We also define `vocab_length` to be the final length of the `vocab` vector.

Trimming and Padding Sequences

Now, each document can be represented by a sequence of numbers using the training set's dictionary. But the problem of varying document size remains. In order to fix this, we can adjust any document to a fixed sequence size `seq_size` $\in \mathbb{N}$. If a tokenized document exceeds `seq_size` in size, we trim the document by removing the rightmost words with document indices more than `seq_size` $- 1$. On the other hand, if a tokenized document's size is lower than `seq_size` we add <PAD/> tokens to the right, until the size of the document totals to `seq_size` tokens. So if we would select `seq_size` to be equal to 7, our document example would be converted into

(32, 4407, 511, 4, 32, 20799, 164)

or we could select `seq_size` to be equal to 10, in which case the example now becomes

(32, 4407, 511, 4, 32, 20799, 164, 49, 6617, 0, 0) .

The documents that have undergone the preprocessing until up to this point, are denoted as $\mathbf{x}^{\text{int}}$.

One Hot Encoding

The training process of regression models, have a natural way of absorbing information from the order between the training instances' values. Words are categorical feature objects and the ordering by the frequency of their appearance in the training set, carries information that forces FNNs to converge into low performance classifiers [25]. Such an ordering is trivial for the document classification task, so this is a low quality representation of words. In order to improve this representation, we'll need to convert the integer encodings, into their respective OHE vectors [26]. For this representation, instead of integers, each of the words is encoded into a vector with size equal to the length of the `vocab` vector. Assuming a word encoded to integer is symbolized by $\texttt{word_int}(\texttt{word_int} \in \{0, \ldots, \texttt{vocab_length} - 1\})$, then the equivalent OHE word `word_ohe` has 0 values in every one of its positions except from the position `word_int`, where it takes the value 1. Thus an OHE document tensor has shape $(\texttt{doc_length}, \texttt{vocab_length})$.

1.3.2 | Word Embedding

The OHE document has a compatible format to become an input into a CNN model, as a single channeled, grayscale image would. But is one hot encoding the proper encoder for a document? Different OHE words are pairwise orthogonal[3], which roughly reveals any information about the similarity between them (e.g. cosine similarity) [27]. On top of that, if the vocabulary is large, say with 5000 unique words, then it would be computationally demanding (assuming it would be feasible at all) for a CNN to process larger document datasets effectively [28]. We can resolve this by shrinking each OHE word into a smaller vector, while retaining the information which the former was holding relative to the vocabulary, and at the same time by harnessing contextual information from the training set.

Word2vec

According to [27], a robust way to learn more compact word encodings of these OHE word vectors, so that the encoding captures information about their similarity with one another inside the training set, is through the utilization of a **Word2vec** unsupervised model [29, 30]. Word2vec models are FNNs. The result of such word representations are vectors. One property of this kind of encoding is that the angles between the word vectors are negatively correlated with the frequency at which these were found closer together inside the training set's documents. If `word_0` was found more frequently closer to `word_1` than with word `word_2`, then the angle between vectors `word_0` and `word_1` is expected to be lower than the angle between `word_0` and `word_2`. Words that share more common semantic meanings inside a sentence, are more likely to be clustered closer together. Word2vec captures the semantic meaning of a word, by taking into account the distance of a word's occurrence relative to other words' occurrences inside a document. The latter words are also called *context words*

[3] With respect to the Euclidean inner product.

[29, 30]. For a given word and its occurrence inside the training set's documents, the number of words that will be taken into account as the context words is constant and is determined by the hyperparameter called *window size* (see Figures 7 and 8). These *embedded* representations of words which are projected in lower dimensions than `vocab_length` have length equal to **embedding dimension**, and we denote this length using the symbol `embedding_dim`.

The Embedding Layer

We set the **embedding layer** [28] as the first hidden layer of a CNN's architecture. The embedding layer is there to essentially assign a meaningful representation to each word. So while the document classifier is being trained, this layer refines improved encoded representations of these OHE words per document, in direct relation with the classification task based on information that exists inside the training set, and passes those representations into the next layers (see Figure 6). Word2vec algorithms can be utilized prior to the classifier's training as a pre-training step for the trainable parameters of the embedding layer, with the constraint that the embedding layer of the classifier has to have the same embedding dimension (or word size)[4] with the one used in Word2vec, and after the Word2vec trainings, we initialize the weights of the embedding layer, using the trainable parameters of a trained Word2vec's layer. An additional option would be to avoid using the embedding layer, however, as we've already explained, using arbitrary representations or OHE of words inside the documents would be impractical, so it is recommended to train those representations beforehand using a Word2vec algorithm, and then pass the learned representations as inputs to an appropriate input layer of the CNN classifier.

Technically speaking, the embedding layer is a linear dense layer with no bias term, which instead of accepting a vector as an instance, it accepts the document matrix where each line is a OHE row vector word. $\mathbf{X}^{\mathrm{OHE}}$ is defined to be such a document which we express as $\mathbf{a}^{[0]}$ in terms of FNN notation, with shape (`doc_length`, `vocab_length`). Now the embedding layer transforms $\mathbf{X}^{\mathrm{OHE}}$ or $\mathbf{a}^{[0]}$ into a matrix $\mathbf{a}^{[1]}$ with shape (`doc_length`, `embedding_dim`) through the simple matrix multiplication

$$\mathbf{a}^{[1]} = \mathbf{X}^{\mathrm{OHE}} \cdot \mathbf{W}^{[1]} \tag{1.16}$$

where $\mathbf{W}^{[1]}$ obviously has shape (`vocab_length`, `embedding_dim`). In the following part of the thesis, we will be based on the architecture that includes the embedding layer, so (1.16) will always be true.

[4] The size of each individual word vector representation.

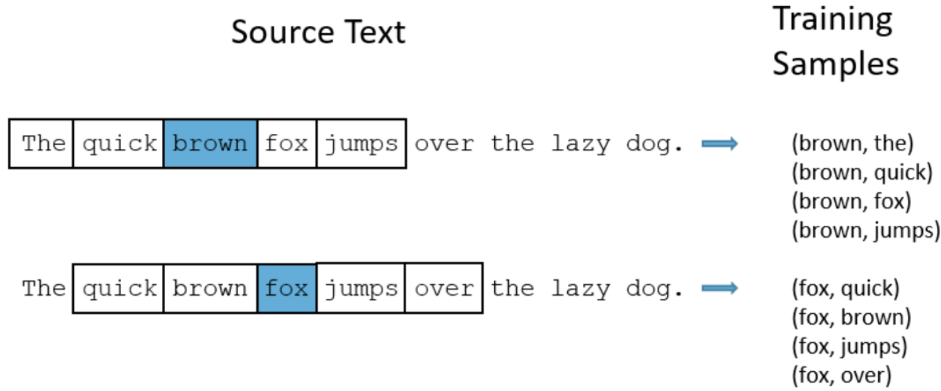


Figure 7: This image was taken from [31]. It shows that the training set is being harnessed iteratively moving from the previous to the next word's occurrence inside a training document instance. The size of the window here is set to take into account 2 word occurrences from the left and 2 word occurrences from the right. So the words found in the *window* corresponding to the word `brown`, which in the case of the example shown in the image, and specifically in the first row, are in $\{\texttt{The}, \texttt{quick}, \texttt{fox}, \texttt{jumps}\}$, so all of these words, the context words, are considered to be close to the word `brown`.

1.3.3 1D Convolutional Layer

We can see the embedded layer's output as a grayscale image where instead, each line corresponds to the given document's word. The **1D convolutional layer**, in case the 1D convolutional layer is next to the embedding layer, it refines information from local rows of document words, instead of local image pixels (located next to the embedding layer) [22]. So, it makes sense for each scan to take into account entire rows of these embedded matrices, taking into account words that are stacked close together to form new more refined features or feature maps. If we would stack more of these layers, then the deeper we go, the wider the area of the initial document that will be refined, in the same way as 2D convolutional layers are acting on images. So initial 1D convolutional layers are taking into account more basic patterns in smaller areas of the document, while the deeper convolutional layers are refining more complex documentwise features associated with more words of the given document.

Based on the modeling of [22], given a layer index l which corresponds to a layer that belongs to the feature extractor component, we can define $n_1^{[l]}$ different filters, each with shape $(k_0^{[l]}, n_1^{[l-1]})$. We define $\mathbf{W}^{[l]}$ to be the weight tensor with shape $\mathbf{k}^{[l]}$ or $(k_0^{[l]}, k_1^{[l]}, k_2^{[l]})$ which is also equal to $(k_0^{[l]}, n_1^{[l-1]}, n_1^{[l]})$, where each slice $\mathbf{W}^{[l]}_{*,*,\nu_1^{[l]}} (\nu_1^{[l]} \in \{0, \ldots, n_1^{[l]} - 1\})$ is a filter belonging to this layer. Additionally, the bias term $\mathbf{b}$ is a matrix of shape $\mathbf{n}^{[l]}$.

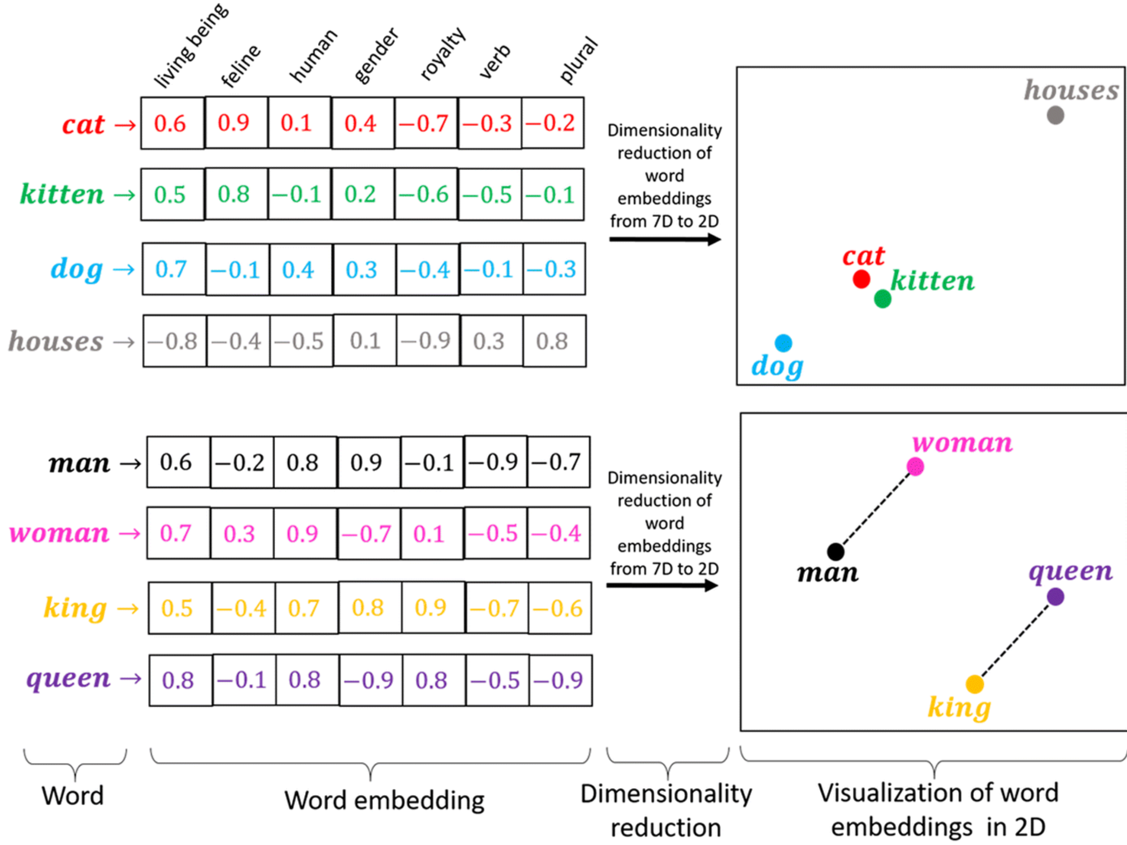


Figure 8: This image was taken from [32]. The example demonstrated here shows a potential case of a trained Word2vec model. Each encoded word's value is expressed in terms of {`living_being`, `feline`, `human`, `gender`, `gender`, `royalty`, `verb`, `plural`}, and depending on how high or low some of the scores are on each of these features for the given word, it is correlated more or less to other words, strictly in terms of these features. In the rightmost visualization of the shown figure, that correlation is expressed as the vector *distance* between words.

We can include a row stride $s^{[l]}$ and also add the $p^{[l]}$ row padding in an analogous way as per the construction of the 2D convolutional layer for image data, with $\mathbf{a}^{[l-1],\text{pad}}$ being the padded input (see Figure 9). Obviously the column stride and padding would have no meaning in the case of this type of layer, so it is assumed that $s^{[l]}$ and $p^{[l]}$ are the row stride and the row padding configurations respectively. The 1D convolution is defined (filter-wise) as

$$(\mathbf{a}^{[l-1],\text{pad}} \circledast_{s^{[l]}} \mathbf{W}^{[l]}_{*,*,\nu_1^{[l]}})_{\nu_0^{[l]}} = \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{n_1^{[l]}-1} (a^{[l-1],\text{pad}}_{\nu_0^{[l]} \cdot s^{[l]} + \kappa_0^{[l]}, \kappa_1^{[l]}} \cdot W^{[l]}_{\kappa_0^{[l]}, \kappa_1^{[l]}, \nu_1^{[l]}}) \,. \tag{1.17}$$

and the linear output is

$$\begin{aligned} z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]}} =& (\mathbf{a}^{[l-1],\text{pad}} \circledast_{s^{[l]}} \mathbf{W}^{[l]}_{*,*,\nu_1^{[l]}})_{\nu_0^{[l]}} + b_{\nu_0^{[l]},\nu_1^{[l]}} \\ =& \sum_{\kappa_0^{[l]}=0}^{k_0^{[l]}-1} \sum_{\kappa_1^{[l]}=0}^{n_1^{[l]}-1} (a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s^{[l]}+\kappa_0^{[l]},\kappa_1^{[l]}} \cdot W^{[l]}_{\kappa_0^{[l]},\kappa_1^{[l]},\nu_1^{[l]}}) + b_{\nu_0^{[l]},\nu_1^{[l]}} \ . \end{aligned} \tag{1.18}$$

which applies for every output row $\nu_0^{[l]}(\nu_0^{[l]} \in \{0,\dots,n_0^{[l]}-1\})$ and filter $\nu_1^{[l]}(\nu_1^{[l]} \in \{0,\dots,n_1^{[l]}-1\})$. The multitude of the output's columns are equal to the number of filters as we've defined them to be and the multitude of the output's rows are

$$n_0^{[l]} = \left\lfloor \frac{n_0^{[l-1]} + 2 \cdot p^{[l]} - k_0^{[l]}}{s^{[l]}} + 1 \right\rfloor \ . \tag{1.19}$$

1.3.4 | 1D Pooling Layers

Analogous properties of 2D pooling layers apply to **1D pooling layers** for document data. 1D pooling layers are defined so that each window is a vector with size $k^{[l]}$.

$$z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]}} = \max(\{a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s^{[l]}+\kappa^{[l]},\nu_1^{[l]}}(\kappa^{[l]} \in \{0,\dots,k^{[l]}-1\})\}) \tag{1.20}$$

and the **average pooling** to be

$$z^{[l]}_{\nu_0^{[l]},\nu_1^{[l]}} = \frac{1}{k^{[l]}} \sum_{\kappa^{[l]}=0}^{k^{[l]}-1} a^{[l-1],\text{pad}}_{\nu_0^{[l]}\cdot s^{[l]}+\kappa^{[l]},\nu_1^{[l]}} \tag{1.21}$$

for every output's row $\nu_0^{[l]} \in \{0,\dots,n_0^{[l]}-1\}$ and column $\nu_1^{[l]} \in \{0,\dots,n_0^{[l]}-1\}$ with (row) stride $s^{[l]}$ and padding $p^{[l]}$. The output's columns and rows are $n_1^{[l]} = n_1^{[l-1]}$ and

$$n_0^{[l]} = \left\lfloor \frac{n_0^{[l-1]} + 2 \cdot p^{[l]} - k^{[l]}}{s^{[l]}} + 1 \right\rfloor \tag{1.22}$$

respectively.

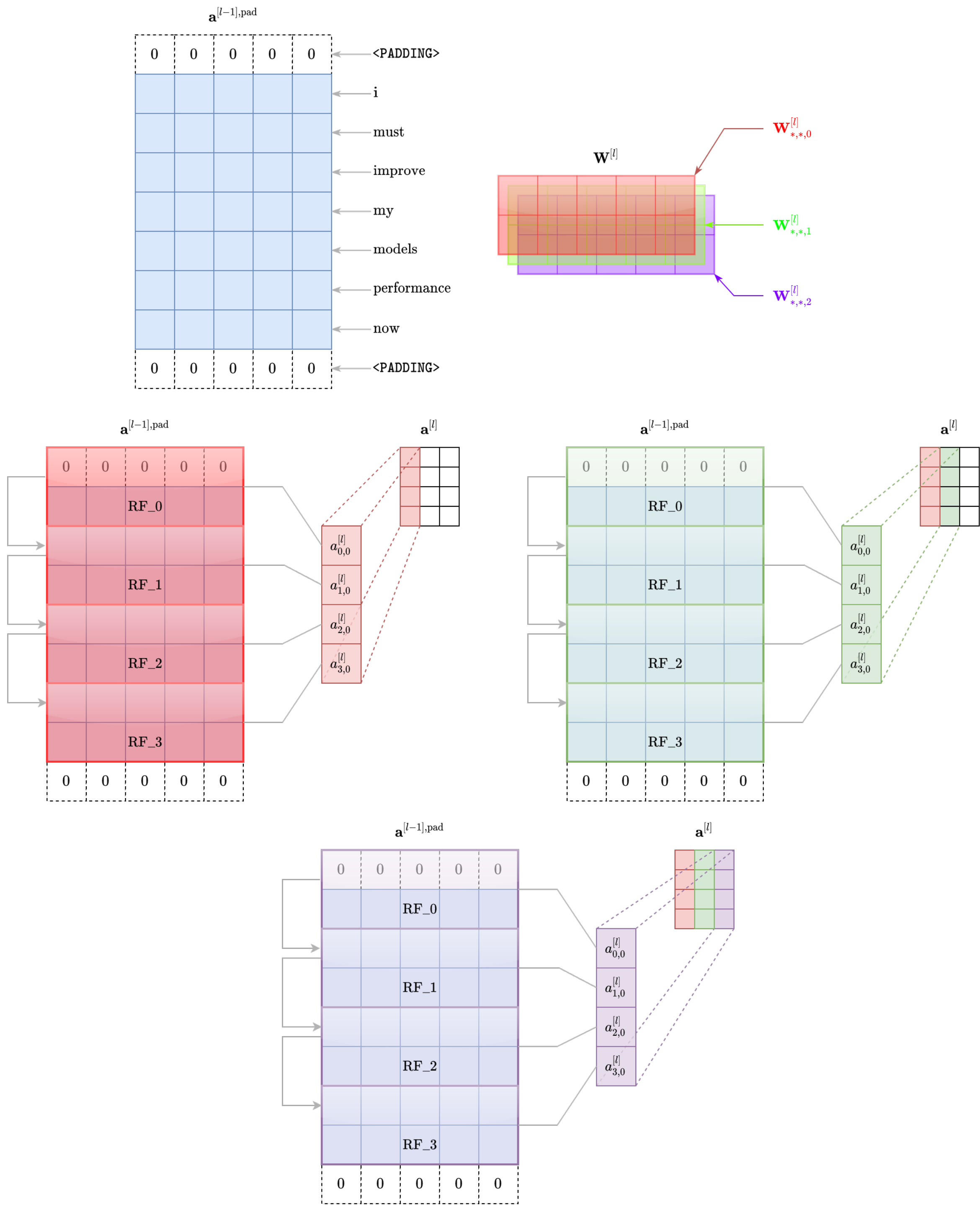


Figure 9: An example of a 1D convolutional layer with index l, input shape $(7, 5)$, $k_0^{[l]} = 2$, stride $s^{[l]} = 2$ and padding $p^{[l]} = 1$. The input layer happens to be the embedded input with `doc_length` $= 7$ and `embedding_dim` $= 5$. In the upper part we can see the padded embedded input $\mathbf{a}^{[l-1],\text{pad}}$ next to the weight tensor $\mathbf{W}^{[\mathbf{l}]}$. The rest of this depiction shows the convolution operation per filter for the red, lime and purple filters, separately, to produce the output $\mathbf{a}^{[l]}$ with shape $(4, 3)$. Each of the 4 receptive fields, is labeled as `RF_`c where $c \in \{0, 1, 2, 3\}$.

2 | Hypothesis and Methodology

We start off by introducing the scientific hypothesis, then we move on to the description of the learning algorithms which were utilized to test this hypothesis in the case of two specific tasks, one task is from the field of image processing and one from the field of natural language processing. Finally we test the hypothesis using *transfer learning*.

2.1 | Hypothesis

Deep CNN architectures are undeniably tedious and time consuming to manually tune for image and natural language related tasks, due to the overwhelmingly large set of potential hyperparameter selections. As for automatic model selection algorithms (e.g. [33, 34]), these require a considerable amount of computational resources and time. So both practices have their own challenges and issues that need to be addressed in order to find appropriate architectures that are capable of leading trainings into high performance models. We investigate a different approach to the tuning problem, by utilizing a deep CNN architecture that works well for a sentiment analysis problem, to solve a problem in the field of image processing and vice versa. By this, the following questions arise: Can architectures that are optimized to work well for NLP tasks, also work well for image processing tasks? Is the converse also true? If so, what are the limitations of this hypothesis? Obviously these questions cannot be experimentally investigated for every problem in these fields, consequently in this thesis we chose to test the validity of this hypothesis under some specific circumstances[5], as an attempt to provide solid evidence that it is possible for CNN transfers from task to task to be effective at least in some way.

2.2 | Transfer Learning

In the field of deep learning, the **transfer learning** methodology is concerned with the replacement of the *target model*'s hidden layers, with the hidden layers which belong to a trained *source model*, in order to transfer knowledge or information extracted from the *source dataset*, that can also be applied to the *target task*, as part of the target model [35]. In order for these methodologies to be effective, it is assumed that the source and target tasks and datasets share some similarities between them [36]. It has been confirmed that transfer learning applied for image and natural language related tasks, can significantly contribute to a target model's performance with a lower generalization error [36, 37].

Specifically regarding classification tasks using FNNs, given a source model's architecture along with a source dataset, we firstly train this model on that dataset. Now given another task (the target task), transfer learning can be utilized by copying the

[5] By saying *specific circumstances* we practically mean to limit our experimentation to specific problems, datasets and architectures.

same hidden layers of the source model as the hidden layers of the target model, and initializing the hidden layer parameters of the target model as the trained hidden layer parameters of the source model, and then we train the target model on the target dataset. The output layers between the source and target models may differ because the number of classes can vary between the source and target tasks, in which case the output layer of the target model has to be randomly initialized before its training starts. The input data feature tensor obviously has to have the same shape in each case, so if the input of the target task has a different shape, it has to be preprocessed in order to match the input's shape from the source task.

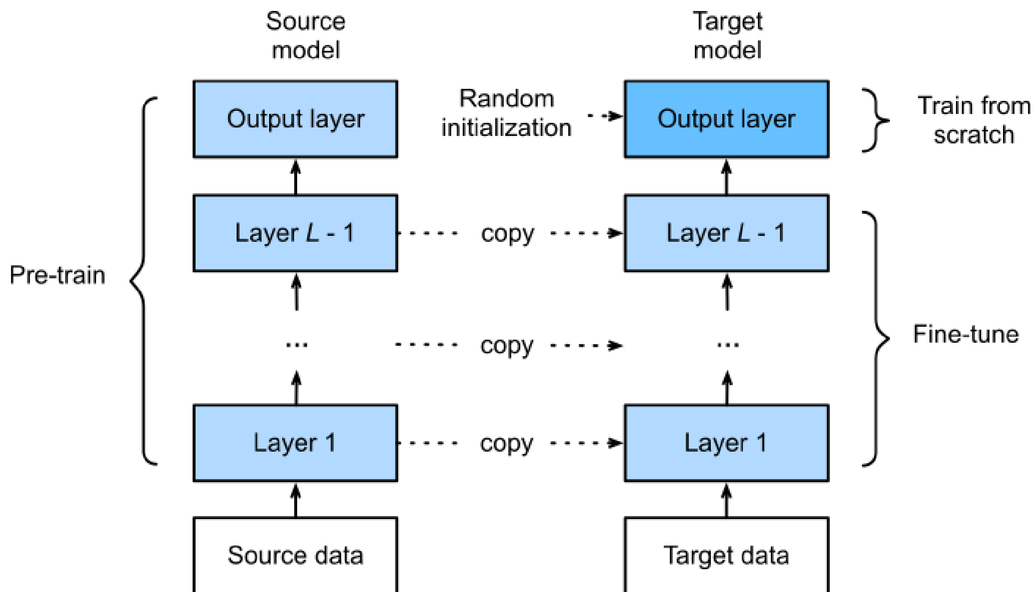


Figure 10: This image was taken from [38]. It shows how transfer learning can be accomplished, by the transference of all the hidden layers (layer hyperparameters and parameters) of an already trained neural network, to a neural network that is supposed to be trained for the target task, which may differ from the source task.

The process of training a source model for transfer learning is called **pre-training** and when the source model has completed its training for the purpose of transfer learning, it is a **pre-trained** model. Training parts of a pre-trained model on the target dataset is also called **fine-tuning**.

Assume that the source task is the image classification between the classes {"car", "bike", "bus"}, an input image which strictly contains exactly one of these objects, and a high performance neural network classifier which will be the source model, that is trained for this task. This source model has obviously been trained on a dataset consisted of car, bike and bus images. For a target task that differs from the source task, say the image classification of dog image vs cat image, a slightly different neural network classifier is trained, where its hidden layers are the same hidden layers as the ones used in the source model. The initial hidden layer parameters are the already

trained hidden layer parameters of the source model. The hidden layers closer to the input layer, will already be able to extract edges, colors and more basic geometrical characteristics from an input image. Thus before the training of the target model even starts, these initial layers will be more capable in identifying features that are already important for the target task, without being explicitly trained on the target dataset. As a consequence, the target model will not need as much training compared to a model whose hidden layers have been randomly initialized, in order to reach higher performances. Finally there are less overfitting risks involved for the target model on the target's training set, as the trainable parameters of hidden layers have already captured relevant information from a different dataset (the source dataset) [36].

2.3 | Problems and Training Algorithms Part I

For our experiments, we have selected to inspect the training algorithms found on the repositories [39] and [40], along with the tasks they are intended to be trained on. Both of these are CNN classifier trainers, one tuned for a specific labeled dataset that holds information regarding an image classification task, and the other tuned for another labeled dataset that holds information regarding a document classification task. For our experiments, we have trained both of these on the tasks they were tuned for, respectively. Furthermore, we have swapped the tasks they solve along with the datasets, with minimal modifications in their architectures and trained these too on the swapped datasets. One noticeable modification is the conversion of 2D layers into 1D layers and the converse. 2D layers were used for the image dataset and 1D layers were used for the text dataset, where the parameters (e.g. filters, and window size) of each of these layers were maintained. All 4 of these training processes and architectures were implemented in a total of 10 epochs, we have selected 64 as the size of each minibatch and the gradient descent algorithm we have used AdaDelta [41]. For the image classification task we have used the categorical cross entropy as the loss function, and in the case of the document classification we have used the binary cross entropy as the loss function. Obviously, the number of output neurons was adjusted for each of the 4 architectures, depending on the task. All of the CNNs were trained from scratch, in the sense that the trainable parameters were initialized by some simple process which allows for a proper training (excluding the embedding layer). Biases were all initialized to 0, and weights across all layers, except from the embedding one, were all randomly initialized by sampling from the Xavier/Glorot (normal) distribution. The following subchapters thoroughly specify each of the discussed training processes and architectures.

2.3.1 | Training Framework of the Original Image Classifier

The training process and architecture we have selected that was intended to solve the image classification task is completely based on [39]. The specific image classification task, which the respective learning process was tuned to handle, is the classification of grayscale images among all the decimal digits or classes $c(c\{0, \ldots, 9\})$. The raw input tensor's shape is equal to $(28, 28)$.

Preprocessing

We firstly normalize this dataset by using Min-Max Normalization. Then we shuffle the instances inside the dataset. The dataset itself has no ordering, but when it is stored in a `Python`'s `list` or `NumPy`'s `ndarray` (which is essentially a `list` with more structure in it), then it gains some non-random ordering, so this is why a prior shuffling of the dataset's elements inside the list is imperative. Finally the dataset is split into a training and validation[6] sets so that $m_{\text{tr}} \approx [6/7 \cdot m]$ and $m_{\text{va}} \approx [1/7 \cdot m]$.

Architecture

The architecture of [39] is tuned for the MNIST dataset [42], and will be denoted as

`arch_opt4mnist_dataset_mnist` .

Table 1 specifies it, and Figure 11 offers a visual sense about the output shapes found across its layers.

2.3.2 | Training Framework of the Original Document Classifier

The training process and architecture we have selected that was intended to solve the document classification task is completely based on [40]. The document classification task, which this learning process was tuned to handle, is the classification of movie review documents into either a positive, in which case it's $c = 0$ or a negative review in which case it's $c = 1$. We set the size of the vocabulary to be equal to 95 489.

Preprocessing

We begin by setting tokens to be words, then we tokenize each document, removing any potential non-alphanumeric symbols and we split each document to word strings, so that every document is converted into a vector of its word strings. Then we create the vocabulary `vocab` which is consisted of the words found in the dataset. The vocabulary will not be reduced, and remains as is. Seeing the vocabulary as a vector of words, the first word is the most frequent one beginning from index 1 and goes on towards the less frequent ones as the `vocab`'s index gets incremented, where in the position with index 0 we assign it to hold the value `"<PAD/>"` for the paddings. Now we adjust each document to hold exactly 400 words, trimming the ones that exceed this length, and padding the ones with lower length using the `"<PAD/>"` string. After that, we substitute each word string with the respective `vocab`'s index. As a final step, we create word embeddings by using a word2vec algorithm once again using the entire dataset, and the trained weights of the hidden word2vec FNN are the initial weights of the trainable embedding layer which is part of the document classifier's architecture. The embedding dimension is set to be 50 and the window size to 10. We split the dataset into a training and validation set so that $m_{\text{tr}} \approx [1/2 \cdot m]$ and $m_{\text{va}} \approx [1/2 \cdot m]$.

[6] As we have not tuned any of the CNNs, the role of the validation set perfectly aligns with the role of a test set.

Architecture

The architecture of [40] is tuned for the LMDRv1.0 dataset [43], and will be denoted as

`arch_opt4imdb_dataset_imdb` .

Table 2 specifies it, and Figure 12 offers a visual sense about the output shapes found across its layers.

2.3.3 | Switching The Tasks

To test if `arch_opt4mnist_dataset_mnist` can itself process a document dataset instead of an image dataset, we have modified it into

`arch_opt4mnist_dataset_imdb`

which includes an embedding layer and where every 2D layer was substituted by a corresponding 1D layer in the feature extractor component of the architecture. This modified architecture is specified by Table 3, and the layerwise output shape is shown in Figure 13. The preprocessing of `arch_opt4mnist_dataset_imdb`'s learning algorithm uses the same preprocessing that corresponds to the architecture `arch_opt4imdb_dataset_imdb`. Analogously, in order for the architecture `arch_opt4imdb_dataset_imdb` to be able to process an image dataset, it was adjusted into

`arch_opt4imdb_dataset_mnist`

by removing the embedding layer and converting 1D layers into their corresponding 2D ones. The preprocessing of `arch_opt4imdb_dataset_mnist`'s learning algorithm uses the preprocessing that corresponds to the architecture `arch_opt4mnist_dataset_mnist`. This modified architecture is specified in Table 4 and the layerwise output shape is shown in Figure 14.

2.3.4 Tables and Diagrams

arch_opt4mnist_dataset_mnist				
Index	Layer Type	Layer hyperparameters	Inp. Shape	Out. Shape
[0]	Input	-	(28,28)	(28,28)
[1]	2D Convolutional	Filter Shape: (3,3) # of Filters: 32 Stride: (1,1) Padding: (0,0) Activation: ReLU	(28,28)	(26,26,32)
[2]	2D Convolutional	Filter Shape: (3,3) # of Filters: 64 Stride: (1,1) Padding: (0,0) Activation: ReLU	(26,26,32)	(24,24,64)
[3]	2D Max Pooling	Pooling Shape: (2,2) Stride: (2,2) Padding: (0,0)	(24,24,64)	(12,12,64)
[4]	Dropout	Drop Rate: 25%	(12,12,64)	(12,12,64)
[5]	Flattening	-	(12,12,64)	(9216)
[6]	Dense	Output Size: 128	(9216)	(128)
[7]	Dropout	Drop Rate: 50%	(128)	(128)
[8]	Dense	Output Size: 10 Activation: Softmax	(128)	(10)

Table 1: Total # of trainable parameters: 1 199 882.

arch_opt4imdb_dataset_imdb				
Index	Layer Type	Layer hyperparameters	In. Shape	Out. Shape
[0]	Input	-	(400)	(400)
[1]	Embedding	Embedding dim.: 50 Vocab. Length: 95489	(400)	(400,50)
[2]	Dropout	Drop Rate: 50%	(400,50)	(400,50)
[3;0]	1D Convolutional	Filter Shape: 3 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(400,50)	(398,10)
[3;1]	1D Convolutional	Filter Shape: 8 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(400,50)	(393,10)
[4;0]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(398,10)	(199,10)
[4;1]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(393,10)	(196,10)
[5;0]	Flattening	-	(199,10)	(1990)
[5;1]	Flattening	-	(196,10)	(1960)
[6]	Concatenate	Inputs: $(\mathbf{a}^{[5;0]}, \mathbf{a}^{[5;1]})$	-	(3950)
[7]	Dropout	Drop Rate: 80%	(3950)	(3950)
[8]	Dense	Output Size: 50 Activation: ReLU	(3950)	(50)
[9]	Dense	Output Size: 1 Activation: Sigmoid	(50)	(1)

Table 2: Total # of trainable parameters: 4 977 571.

arch_opt4mnist_dataset_imdb				
Index	Layer Type	Layer hyperparameters	Inp. Shape	Out. Shape
[0]	Input	-	(400)	(400)
[1]	Embedding	Embedding dim.: 50 Vocab. Length: 95489	(400)	(400,50)
[2]	1D Convolutional	Filter Shape: 3 # of Filters: 32 Stride: 1 Padding: 0 Activation: ReLU	(400,50)	(398,32)
[3]	1D Convolutional	Filter Shape: 3 # of Filters: 64 Stride: 1 Padding: 0 Activation: ReLU	(398,32)	(396,64)
[4]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(396,64)	(198,64)
[5]	Dropout	Drop Rate: 25%	(198,64)	(198,64)
[6]	Flattening	-	(198,64)	(12672)
[7]	Dense	Output Size: 128	(12672)	(128)
[8]	Dropout	Drop Rate: 50%	(128)	(128)
[9]	Dense	Output Size: 1 Activation: Sigmoid	(128)	(1)

Table 3: Total # of trainable parameters: 6 407 763.

arch_opt4imdb_dataset_mnist				
Index	Layer Type	Layer hyperparameters	In. Shape	Out. Shape
[0]	Input	-	(28,28)	(28,28)
[1]	Dropout	Drop Rate: 50%	(28,28)	(28,28)
[2;0]	2D Convolutional	Filter Shape: (3,3) # of Filters: 10 Stride: (1,1) Padding: (0,0) Activation: ReLU	(28,28)	(26,26,10)
[2;1]	2D Convolutional	Filter Shape: (8,8) # of Filters: 10 Stride: (1,1) Padding: (0,0) Activation: ReLU	(26,26,10)	(21,21,10)
[3;0]	2D Max Pooling	Pooling Shape: (2,2) Stride: (2,2) Padding: (0,0)	(26,26,10)	(13,13,10)
[3;1]	2D Max Pooling	Pooling Shape: (2,2) Stride: (2,2) Padding: (0,0)	(21,21,10)	(10,10,10)
[4;0]	Flattening	-	(13,13,10)	(1690)
[4;1]	Flattening	-	(10,10,10)	(1000)
[5]	Concatenate	Inputs: $(\mathbf{a}^{[4;0]}, \mathbf{a}^{[4;1]})$	-	(2690)
[6]	Dropout	Drop Rate: 80%	(2690)	(2690)
[7]	Dense	Output Size: 50 Activation: ReLU	(2690)	(50)
[8]	Dense	Output Size: 10 Activation: Softmax	(50)	(10)

Table 4: Total # of trainable parameters: 135 810.

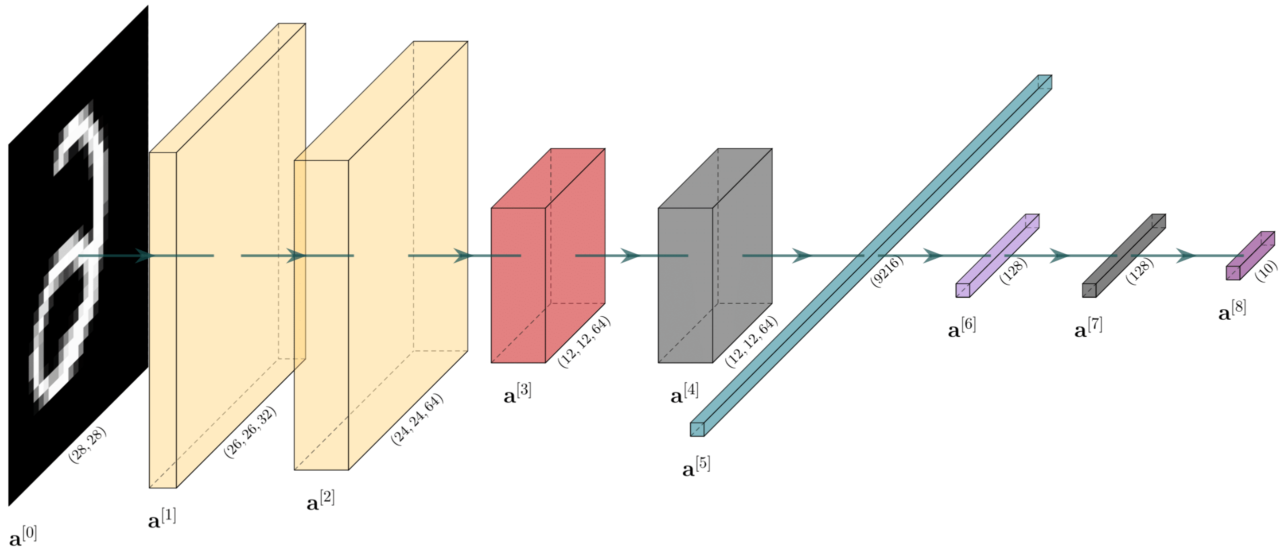


Figure 11: This figure was obtained using the software [44]. It shows `arch_opt4mnist_dataset_mnist`'s output shapes layerwise.

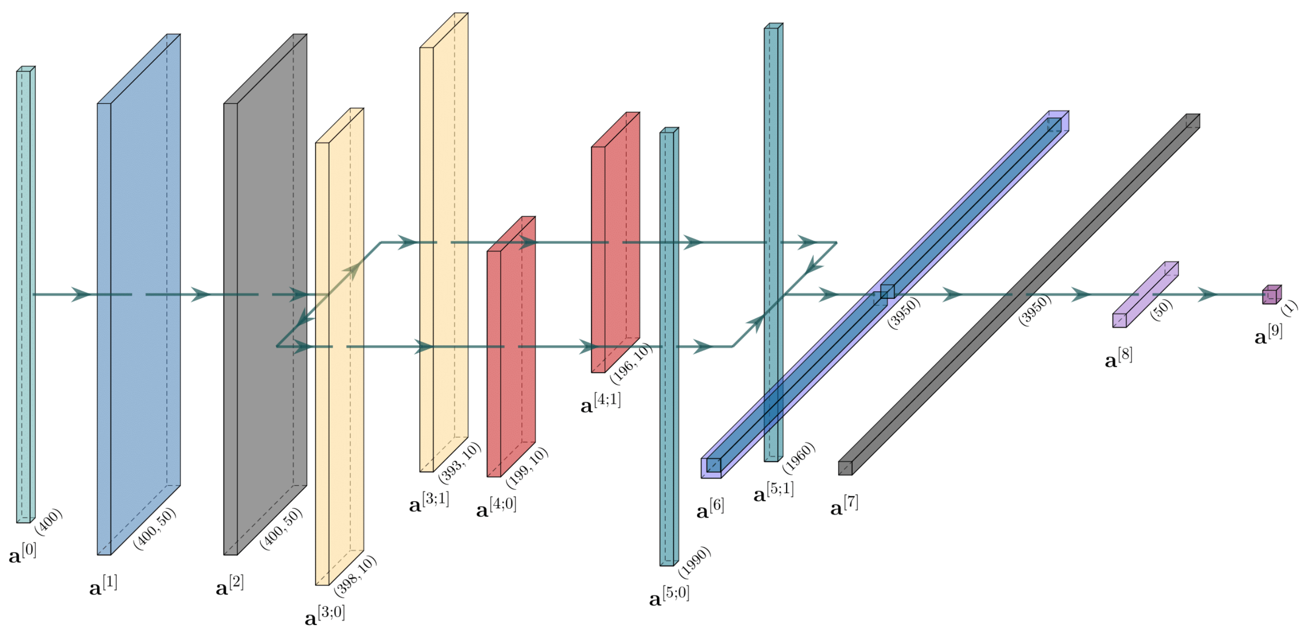


Figure 12: This figure was obtained using the software [44]. It shows `arch_opt4imdb_dataset_imdb`'s output shapes layerwise.

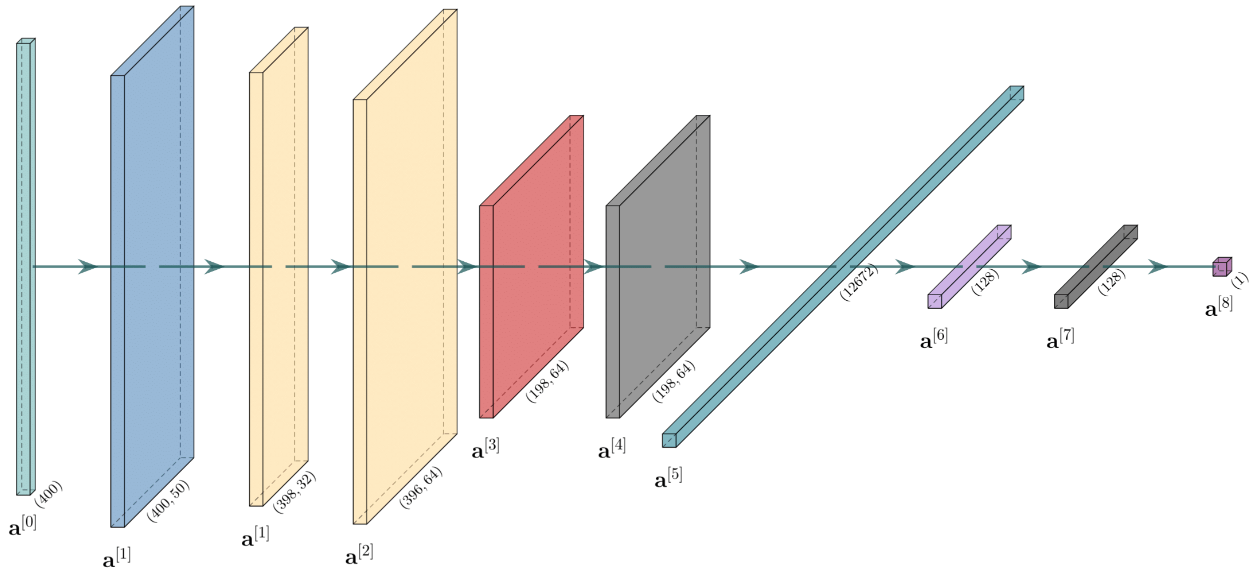


Figure 13: This figure was obtained using the software [44]. It shows `arch_opt4mnist_dataset_imdb`'s output shapes layerwise.

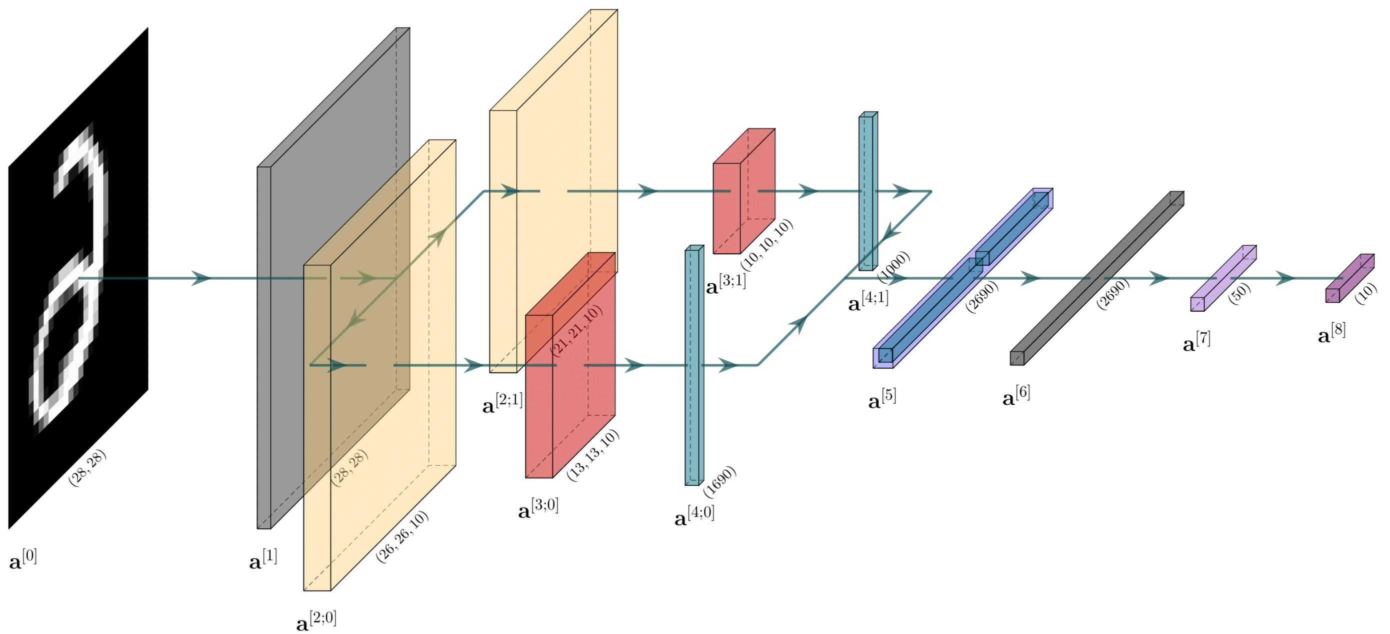


Figure 14: This figure was obtained using the software [44]. It shows `arch_opt4imdb_dataset_mnist`'s output shapes layerwise.

2.4 | Problems and Training Algorithms Part II

We have utilized transfer learning for image and document classification tasks. The source model architectures are identical to `arch_opt4mnist_dataset_mnist` and `arch_opt4imdb_dataset_imdb`, where the first is tuned for a specific image classification task, and the latter is tuned for a specific document classification task. The source models were trained on the respective problems they were tuned to solve. When the training of the source models finished, their respective hidden layers were used as the hidden layers of the target models (excluding the embedding layer). For the fine-tuning part, the same datasets were used as the target datasets, but they were swapped for the training of each of the target models. Additionally, the target models were trained on the same datasets as their respective source models with the same hyperparameters. Thus there are 2 source models and for each of these source models, there are 2 target models. The training processes share the same number of epochs, minibatch size, gradient descent algorithm and trainable parameter initialization as in the case of 2.3.

2.4.1 | Training Framework of the Source Image Classifier

The training process that trains the source image classifier is based on [39]. It is tuned to produce high performance classifiers for the same task as the one described in 2.3.1 with the same raw input shape and preprocessing.

Architecture

The source architecture of the image classification task is set to be equal to `arch_opt4 mnist_dataset_mnist` and will be denoted as

`mnistsrc`

Table 5 specifies it, and Figure 15 offers a visual sense about the output shapes found across its layers.

2.4.2 | Training Framework of the Source Document Classifier

The training process that trains the source document classifier is based on [40]. It is tuned to produce high performance classifiers for the same task as the one described in 2.3.2 with the same vocabulary size.

Preprocessing

The same preprocessing as in 2.3.2 was used here, except this time the document size was set to be equal to 28 instead of 400, and the embedding dimension was set to be equal to 28 instead of 50. This is one way to make it possible for transfer learning to work properly.

Architecture

The hidden layers' hyperparameters of the source architecture of the image classification task, are set to be equal to the respective ones as in the `arch_opt4imdb_dataset_imdb` and will be denoted as

`imdbsrc` .

The shapes of each layer's output is different from `arch_opt4imdb_dataset_imdb`, as the preprocessing leads to different output shapes on the input and embedding layers. Table 6 specifies it, and Figure 16 offers a visual sense about the output shapes found across its layers.

2.4.3 | Training Framework of the Target Models

The target model which was based on the source model that was originally trained on the image dataset, was trained on the document dataset where the respective trained source model was trained on. The respective architecture of that target model, is denoted as

`mnistsrc_imdbtgt` ,

Table 7 specifies it, and Figure 17 shows its layers' outputs. For the target architecture, an embedding layer was added prior to all of the source architecture's hidden layers, while the number of output neurons in the output layer was set to be 1 which is equipped with the sigmoid activation function. On the other hand, the target model associated with the source model that was trained on the document dataset, was trained on the image dataset where the respective trained source model was trained on. The respective architecture of that target model, is denoted as

`imdbsrc_mnisttgt` ,

Table 8 specifies it, and Figure 18 shows its layers' outputs. The embedding layer of the source architecture was not used in the target model, while the number of output neurons in the output layer was set to be 10, while the layer was equipped with the softmax activation function. In both of these target model cases, the output layer's initial trainable parameters were sampled from the Xavier/Glorot distribution.

The same source models were also trained on the same source datasets, with the same hyperparameter configurations in each of the 2 cases, which is basically the same as training the same source models for the double amount of epochs as part of the pre-training stage. The architecture of the target model associated with the source model where both of these models were trained on the same image dataset, is denoted as

`mnistsrc_mnisttgt` ,

and the architecture of the target model associated with the source model where both of these models were trained on the same document dataset, is denoted as

`imdbsrc_imdbtgt` .

where obviously

$$\texttt{mnistsrc} = \texttt{mnistsrc_mnisttgt} \ ,$$

and

$$\texttt{imdbsrc} = \texttt{imdbsrc_imdbtgt} \ .$$

2.4.4 Tables and Diagrams

mnistsrc or mnistsrc_mnisttgt				
Index	Layer Type	Layer hyperparameters	Inp. Shape	Out. Shape
[0]	Input	-	(28,28)	(28,28)
[1]	2D Convolutional	Filter Shape: (3,3) # of Filters: 32 Stride: (1,1) Padding: (0,0) Activation: ReLU	(28,28)	(26,26,32)
[2]	2D Convolutional	Filter Shape: (3,3) # of Filters: 64 Stride: (1,1) Padding: (0,0) Activation: ReLU	(26,26,32)	(24,24,64)
[3]	2D Max Pooling	Pooling Shape: (2,2) Stride: (2,2) Padding: (0,0)	(24,24,64)	(12,12,64)
[4]	Dropout	Drop Rate: 25%	(12,12,64)	(12,12,64)
[5]	Flattening	-	(12,12,64)	(9216)
[6]	Dense	Output Size: 128	(9216)	(128)
[7]	Dropout	Drop Rate: 50%	(128)	(128)
[8]	Dense	Output Size: 10 Activation: Softmax	(128)	(10)

Table 5: Total # of trainable parameters: 1 199 882.

imdbsrc or imdbsrc_imdbtgt				
Index	Layer Type	Layer hyperparameters	In. Shape	Out. Shape
[0]	Input	-	(28)	(28)
[1]	Embedding	Embedding dim.: 28 Vocab. Length: 95489	(28)	(28,28)
[2]	Dropout	Drop Rate: 50%	(28,28)	(28,28)
[3;0]	1D Convolutional	Filter Shape: 3 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(28,28)	(26,10)
[3;1]	1D Convolutional	Filter Shape: 8 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(28,28)	(21, 10)
[4;0]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(26,10)	(13, 10)
[4;1]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(21, 10)	(10, 10)
[5;0]	Flattening	-	(199,10)	(130)
[5;1]	Flattening	-	(196,10)	(100)
[6]	Concatenate	Inputs: $(\mathbf{a}^{[5;0]}, \mathbf{a}^{[5;1]})$	-	(230)
[7]	Dropout	Drop Rate: 80%	(230)	(230)
[8]	Dense	Output Size: 50 Activation: ReLU	(230)	(50)
[9]	Dense	Output Size: 1 Activation: Sigmoid	(50)	(1)

Table 6: Total # of trainable parameters: 1 086 625.

mnistsrc_imdbtgt				
Index	Layer Type	Layer hyperparameters	Inp. Shape	Out. Shape
[0]	Input	-	(28)	(28)
[1]	Embedding	Embedding dim.: 28 Vocab. Length: 95489	(28)	(28,28)
[2]	2D Convolutional	Filter Shape: (3,3) # of Filters: 32 Stride: (1,1) Padding: (0,0) Activation: ReLU	(28,28)	(26,26,32)
[3]	2D Convolutional	Filter Shape: (3,3) # of Filters: 64 Stride: (1,1) Padding: (0,0) Activation: ReLU	(26,26,32)	(24,24,64)
[4]	2D Max Pooling	Pooling Shape: (2,2) Stride: (2,2) Padding: (0,0)	(24,24,64)	(12,12,64)
[5]	Dropout	Drop Rate: 25%	(12,12,64)	(12,12,64)
[6]	Flattening	-	(12,12,64)	(9216)
[7]	Dense	Output Size: 128	(9216)	(128)
[8]	Dropout	Drop Rate: 50%	(128)	(128)
[9]	Dense	Output Size: 1 Activation: Sigmoid	(128)	(1)

Table 7: Total # of trainable parameters: 2 270 645.

imdbsrc_mnisttgt				
Index	Layer Type	Layer hyperparameters	In. Shape	Out. Shape
[0]	Input	-	(28,28)	(28,28)
[1]	Dropout	Drop Rate: 50%	(28,28)	(28,28)
[2;0]	1D Convolutional	Filter Shape: 3 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(28,28)	(26,10)
[2;1]	1D Convolutional	Filter Shape: 8 # of Filters: 10 Stride: 1 Padding: 0 Activation: ReLU	(28,28)	(21, 10)
[3;0]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(26,10)	(13, 10)
[3;1]	1D Max Pooling	Pooling Shape: 2 Stride: 2 Padding: 0	(21, 10)	(10, 10)
[4;0]	Flattening	-	(199,10)	(130)
[4;1]	Flattening	-	(196,10)	(100)
[5]	Concatenate	Inputs: $(\mathbf{a}^{[5;0]}, \mathbf{a}^{[5;1]})$	-	(230)
[6]	Dropout	Drop Rate: 80%	(230)	(230)
[7]	Dense	Output Size: 50 Activation: ReLU	(230)	(50)
[8]	Dense	Output Size: 10 Activation: Softmax	(50)	(10)

Table 8: Total # of trainable parameters: 15 160.

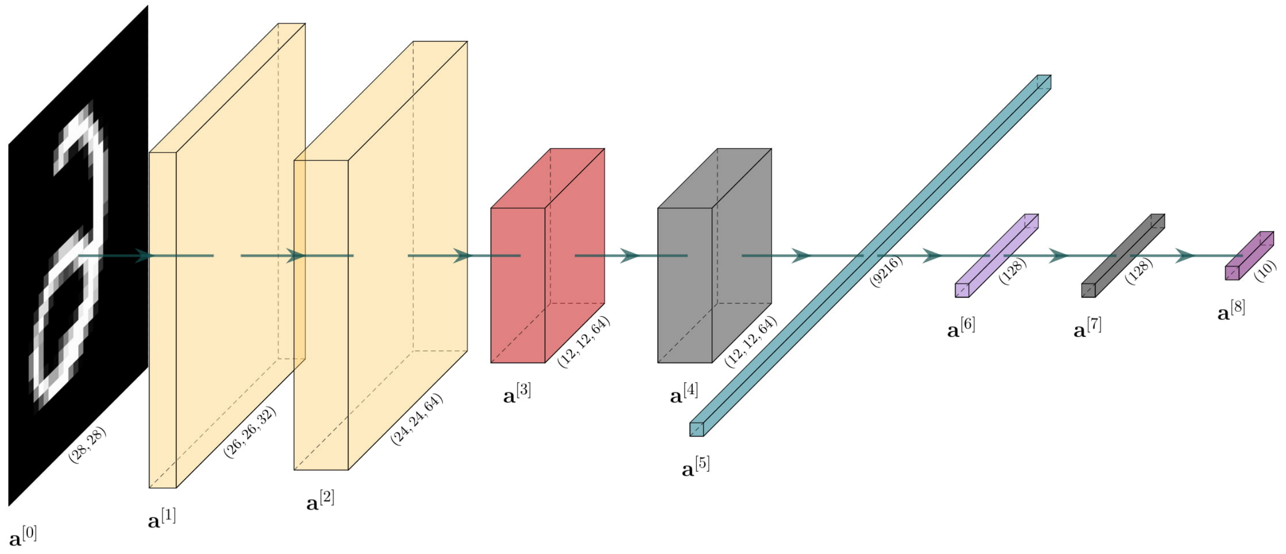


Figure 15: This figure was obtained using the software [44]. It shows `mnistsrc`'s or `mnistsrc_mnisttgt`'s output shapes layerwise.

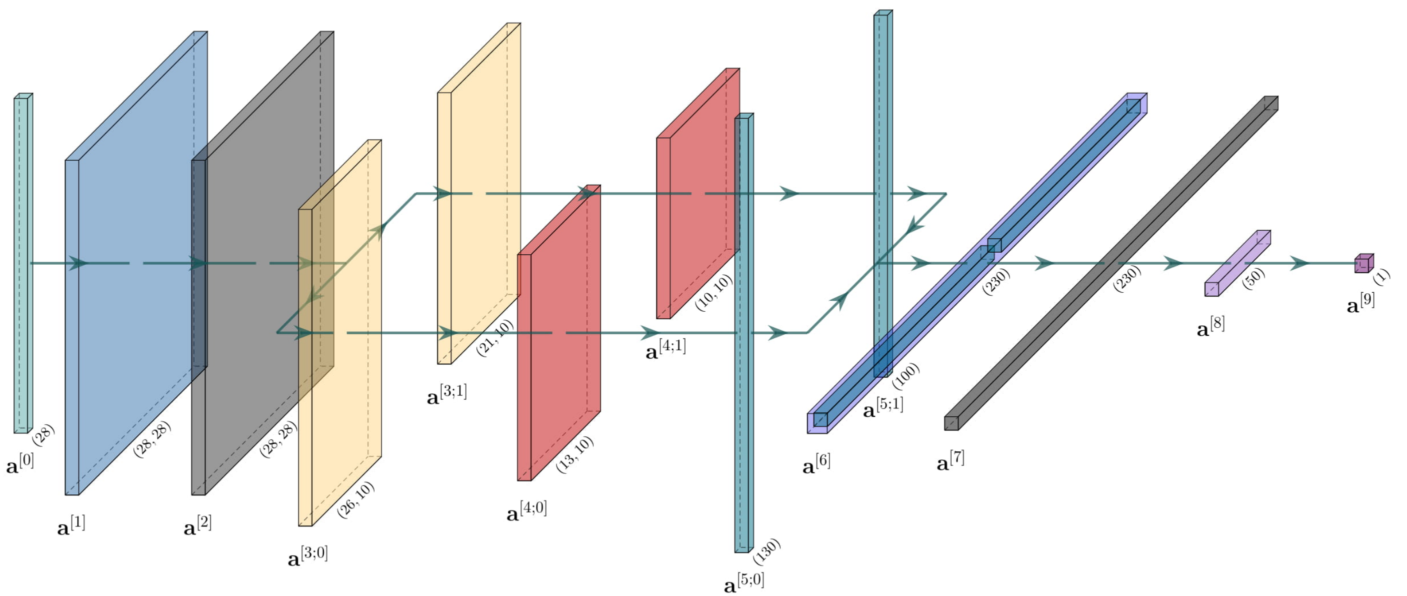


Figure 16: This figure was obtained using the software [44]. It shows `imdbsrc`'s or `imdbsrc_imdbtgt`'s output shapes layerwise.

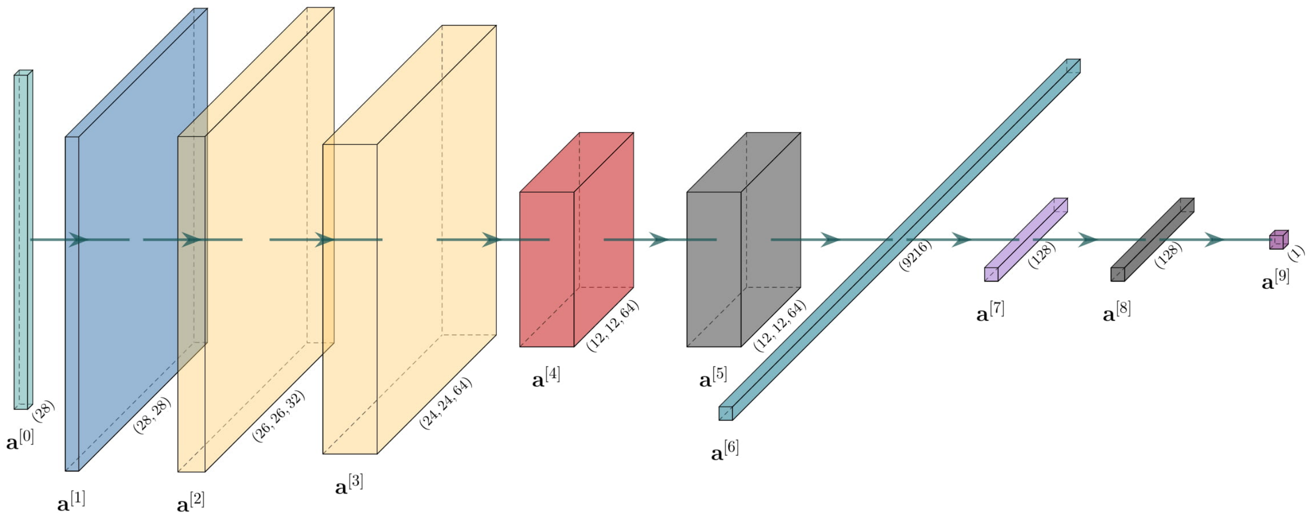


Figure 17: This figure was obtained using the software [44]. It shows `mnistsrc_imdbtgt`'s output shapes layerwise.

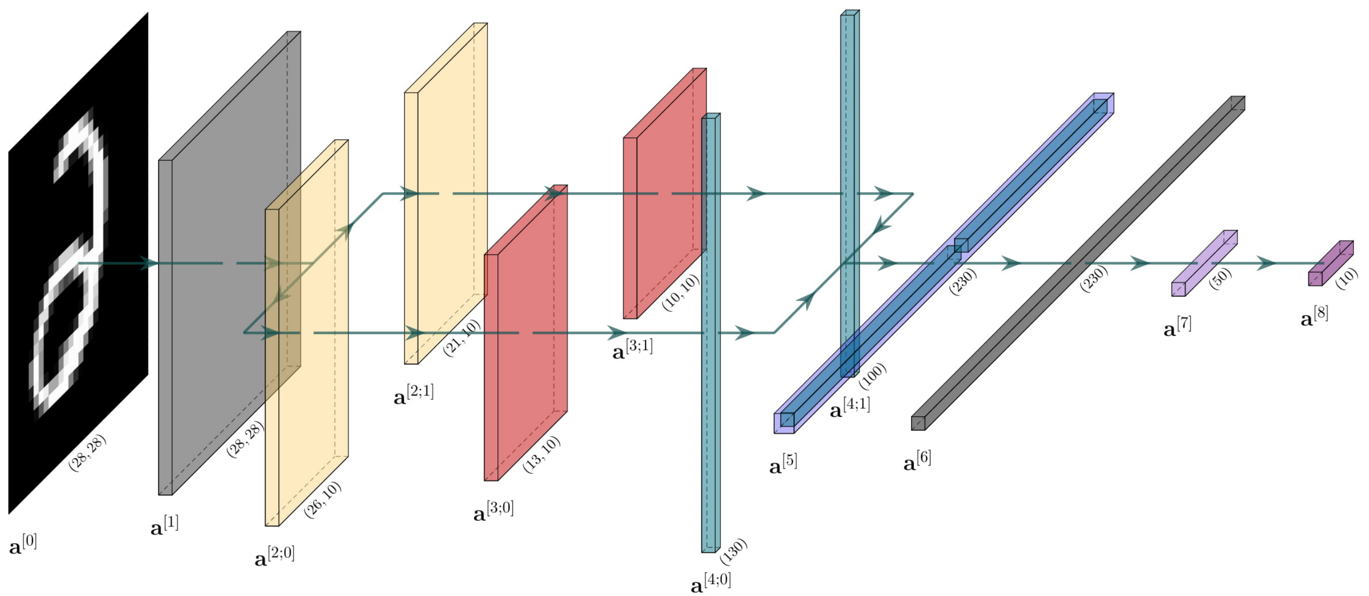


Figure 18: This figure was obtained using the software [44]. It shows `imdbsrc_mnisttgt`'s output shapes layerwise.

3 | Experimental Evaluation

For each of the training algorithms specified in the Subchapter 2.3 corresponding to one of the mentioned architectures, we have trained a classifier and here we present the produced classifiers' evaluation metrics per epoch, which were collected in tables followed by their respective plots. The experiments were conducted in a Debian 9 (Linux) equipped with Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and with total memory ~132GB. All the experiments were implemented using `Python` version 3. `Python` version 2 was also used in some cases. We have utilized the open source library of `Keras` [45], using `TensorFlow` [46] as its *backend*.

- For `Python2` we have used `Keras` version 2.3.0 and `TensorFlow` version 1.13.1.
- For `Python3` we have used `Keras` version 2.3.0 and `TensorFlow` version 2.0.0.

The experiments are reproducible per machine.

3.1 | Evaluation Metrics

The evaluation metrics we have collected for all of the trained classifiers are that from the respective loss functions which corresponds to each learning process and accuracy. The measured numbers presented in the tables were rounded to the 4th fractional digit so that the tables could fit inside the pages, given that the rest of the digits do not play a significant role. Regarding the evaluation tables, we have highlighted the cell corresponding to the epoch with the optimal value among the values of all epochs per metric. So for example, if epoch $3 \in \{0, \dots, 9\}$ of the rounded validation set's loss has the lowest value among all epochs, then the cell containing that value will be highlighted. The cells which have a non-white background are the highlighted ones. Also, it should be noted that `Keras/TensorFlow`'s evaluation metrics parameterized on the respective training set per epoch, are estimators of the real evaluation metrics. These estimators are computed as an average of the evaluation metrics parameterized on every minibatch per epoch [47]. Because of that, we will observe excessively lower *estimated* performance on the training set compared to the validation/test set in initial epochs. That is because during the initial minibatch updates in early epochs, the loss is way higher, and as a result the average value of all loss estimations is directly affected by these minibatch losses.

3.2 | Datasets

As we have already mentioned in the Subchapter 2.3.1, each of the original training algorithms are tuned on the MNIST and the LMRDv1.0 datasets.

3.2.1 | The MNIST Dataset

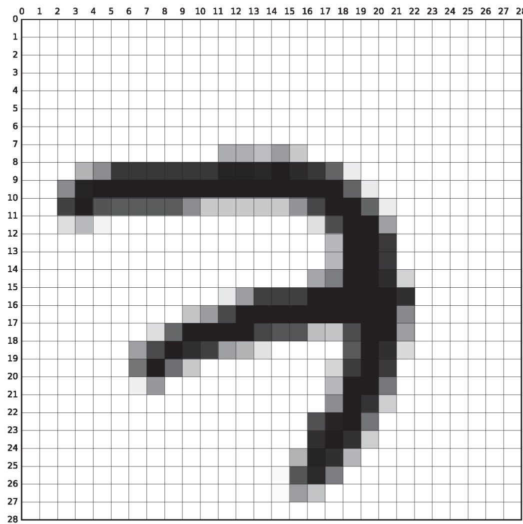
The **MNIST** dataset, formally known as the Modified National Institute of Standards and Technology database [42], is used to solve the task of image classification

of grayscale images consisted of handwritten digits. It is a balanced dataset with 10 unique labels, where each label index $c (c \in \{0, 1, \ldots, 9\})$ corresponds to the handwritten decimal digit c. So an image $\mathbf{x}$ with label index c (or y) corresponds to the number c. The dataset contains 70 000 grayscale images of handwritten digits where every image has a fixed $(28, 28)$ shape (see Figure 19). Each image's pixel positioned at (j_0, j_1) corresponds to an integer number $x_{j_0,j_1} (x_{j_0,j_1} \in \{0, \ldots, 255\})$.

3.2.2 | The LMRDv1.0 Dataset

The **LMRDv1.0**, formally known as the Large Movie Review Dataset v1.0 [43], containing comments of movie reviews, originated from the IMDb's official website[7] [43]. We focus on 50 000 of these comments which are labeled either as positive or negative reviews, each corresponding to a specific movie. This balanced dataset is suitable for sentiment analysis and specifically for the binary classification problem of a positively versus negatively reviewed movie from multiple users among a wide variety of movies. Each comment is a specific document instance. The dataset we invoke for the experiments is a slightly modified version of the mentioned one, pulled from the repository [48]. We may also refer to the same dataset as the **IMDb** dataset. Below a dataset's instance of these comments is shown in string format.

```
"i␣really␣liked␣this␣movie␣despite␣one␣scene␣that␣was␣pretty␣bad␣the␣
one␣when␣samantha␣and␣nick␣are␣flirting␣in␣the␣hotel␣the␣story␣is␣
so␣cool␣and␣can␣t␣wait␣to␣read␣the␣book␣bravo␣for␣the␣super␣station"
```



(a) MNIST sample belonging to the digit '7'.

(b) 100 samples from the MNIST training set.

Figure 19: This image was taken from [49], and it shows a sample of images from the MNIST dataset. In this demonstration, the color intensity of each image pixel was inverted.

[7] https://www.imdb.com

3.3 | Experiments Part I

These experiments were conducted using the methodology proposed in 2.3.

3.3.1 | Experiments Based on MNIST's Architecture

Table 9 (for `Python2`), Table 11 (for `Python3`) along with their respective plots shown in Figure 20 (a), (b) (for `Python2`), and Figure 21 (a), (b) (for `Python3`) show the evaluation of the classifier that was produced by the architecture which was tuned for the problem of MNIST, this architecture was named as `arch_opt4mnist_dataset_mnist`. This classifier was trained on the MNIST dataset by the training algorithm corresponding to that architecture.

On the other hand, Table 10 (for `Python2`), Table 12 (for `Python3`) along with their respective plots shown in Figure 20 (c), (d) (for `Python2`), and Figure 21 (c), (d) (for `Python3`) show the evaluation of the classifier that was produced by the architecture which was tuned for the problem of MNIST, but is modified in a way that it is able to process the IMDb dataset, this architecture was named as `arch_opt4mnist_dataset_imdb`, which is a slightly modified version of the architecture `arch_opt4mnist_dataset_mnist` that allows for the processing of the IMDb dataset's documents. This classifier was trained on the IMDb dataset by the training algorithm corresponding to that architecture.

Training `arch_opt4mnist_dataset_mnist` on MNIST using `Python2`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.2107	0.0809	0.0614	0.0523	0.0472	0.0425	0.0395	0.0370	0.0357	0.0336
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.0601	0.0468	0.0434	0.0375	0.0349	0.0329	0.0318	0.0359	0.0300	0.0338
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.9360	0.9761	0.9817	0.9848	0.9865	0.9877	0.9882	0.9892	0.9893	0.9906
$\text{acc}(\mathbb{D}_{\text{va}})$	0.9837	0.9867	0.9866	0.9897	0.9900	0.9905	0.9910	0.9898	0.9917	0.9918

Table 9: Evaluation metrics during the training of the classifier based on the MNIST dataset, that was produced by the MNIST architecture using `Python2`.

Training `arch_opt4mnist_dataset_imdb` on IMDb using `Python2`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.5286	0.2913	0.2445	0.2087	0.1867	0.1599	0.1385	0.1124	0.0856	0.0632
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.3115	0.2644	0.2744	0.2855	0.2742	0.2759	0.2938	0.3256	0.3846	0.4264
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.7111	0.8802	0.9041	0.9198	0.9283	0.9400	0.9485	0.9600	0.9702	0.9784
$\text{acc}(\mathbb{D}_{\text{va}})$	0.8706	0.8916	0.8914	0.8915	0.8973	0.8956	0.8940	0.8975	0.8901	0.8948

Table 10: Evaluation metrics during the training of the classifier based on the IMDb dataset, that was produced by the MNIST architecture after its modifications to allow for such a training, using `Python2`.

Training `arch_opt4mnist_dataset_mnist` on MNIST using `Python3`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.2112	0.0804	0.0606	0.0513	0.0462	0.0413	0.0384	0.0361	0.0354	0.0333
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.0609	0.0462	0.0437	0.0354	0.0350	0.0333	0.0309	0.0347	0.0302	0.0361
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.9360	0.9761	0.9817	0.9847	0.9862	0.9877	0.9887	0.9895	0.9891	0.9905
$\text{acc}(\mathbb{D}_{\text{va}})$	0.9829	0.9869	0.9864	0.9901	0.9894	0.9905	0.9909	0.9906	0.9919	0.9915

Table 11: Evaluation metrics during the training of the classifier based on the MNIST dataset, that was produced by the MNIST architecture using `Python3`.

Training `arch_opt4mnist_dataset_imdb` on IMDb using `Python3`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.4854	0.2876	0.2429	0.2087	0.1837	0.1572	0.1337	0.1049	0.0772	0.0567
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.2946	0.2613	0.2943	0.2611	0.2718	0.2877	0.3009	0.4066	0.3555	0.4279
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.7500	0.8835	0.9046	0.9199	0.9298	0.9415	0.9510	0.9626	0.9747	0.9812
$\text{acc}(\mathbb{D}_{\text{va}})$	0.8792	0.8921	0.8869	0.8991	0.8970	0.8924	0.8894	0.8774	0.8966	0.8933

Table 12: Evaluation metrics during the training of the classifier based on the IMDb dataset, that was produced by the MNIST architecture after its modifications to allow for such a training, using `Python3`.

`Python2`, Trainings Based on the Architecture Tuned for MNIST

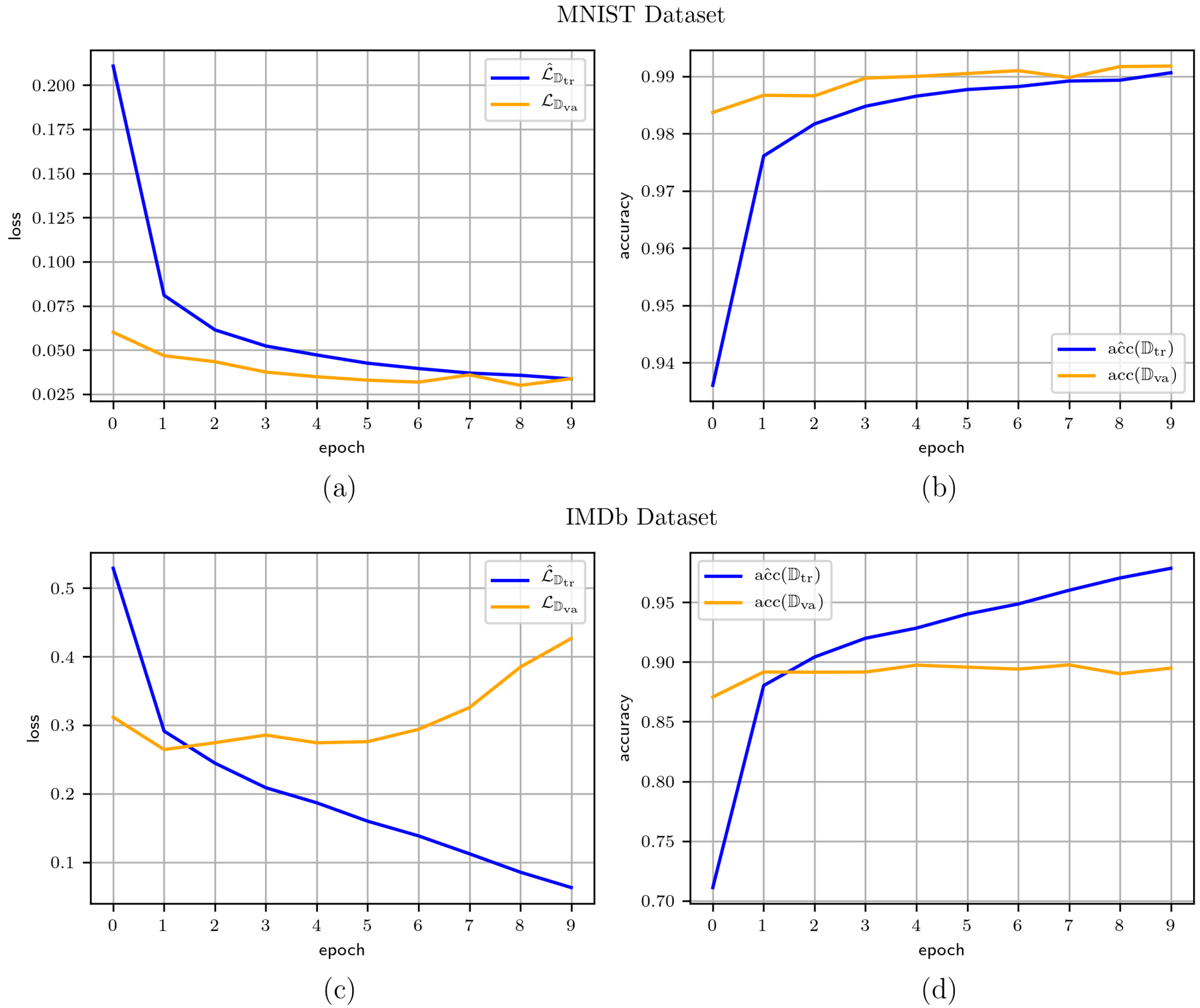


Figure 20: Performance graphs of the classifiers' trainings produced using the architectures which were originally tuned for the MNIST task in `Python2`. (a) and (b) correspond to the produced model using `arch_opt4mnist_dataset_mnist` which was trained on the MNIST dataset. (c) and (d) correspond to the produced model using `arch_opt4mnist_dataset_imdb` which was trained on the IMDb dataset.

`Python3`, Trainings Based on the Architecture Tuned for MNIST

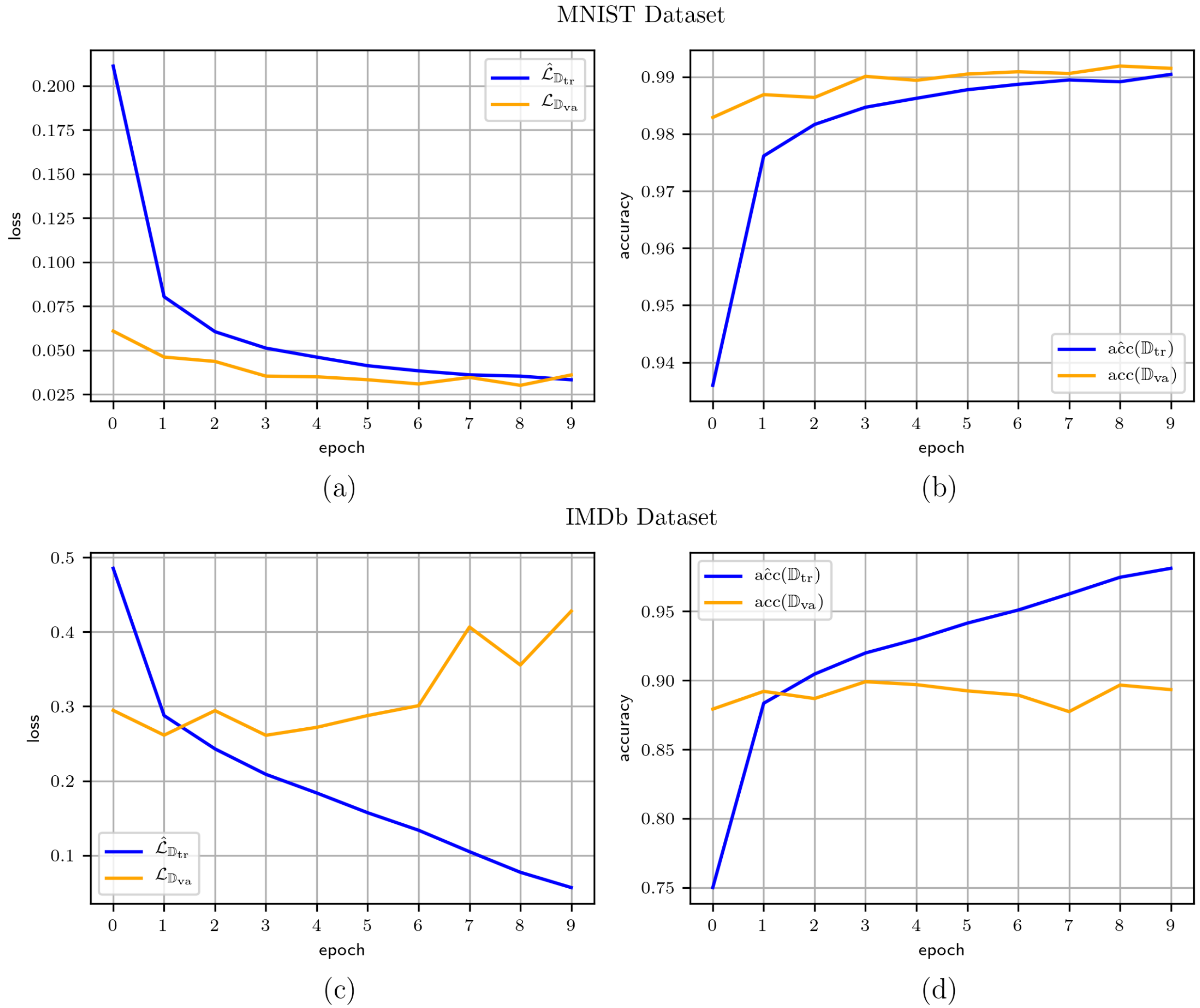


Figure 21: Performance graphs of the classifiers' trainings produced using the architectures which were originally tuned for the MNIST task in `Python3`. (a) and (b) correspond to the produced model using `arch_opt4mnist_dataset_mnist` which was trained on the MNIST dataset. (c) and (d) correspond to the produced model using `arch_opt4mnist_dataset_imdb` which was trained on the IMDb dataset.

3.3.2 | Experiments Based on LMRDv1.0's Architecture

Table 14 (for `Python2`), Table 16 (for `Python3`) along with their respective plots shown in Figure 22 (c), (d) (for `Python2`), and Figure 23 (c), (d) (for `Python3`) show the evaluation of the classifier that was produced by the architecture which was tuned for the problem of IMDb, this architecture was named as `arch_opt4imdb_dataset_imdb`. This classifier was trained on the IMDb dataset by the training algorithm corresponding to that architecture.

On the other hand, Table 13 (for `Python2`), Table 15 (for `Python3`) along with their respective plots shown in Figure 22 (a), (b) (for `Python2`), and Figure 23 (a), (b) (for `Python3`) show the evaluation of the classifier that was produced by the architecture which was tuned for the problem of IMDb[8] but is modified in a way that it is able to process the MNIST dataset, this architecture was named as `arch_opt4imdb_dataset_mnist`, which is a slightly modified version of the architecture `arch_opt4imdb_dataset_imdb` that allows for the processing of the MNIST dataset's images. This classifier was trained on the MNIST dataset by the training algorithm corresponding to that architecture.

[8] Reminding that the IMDb and LMRDv1.0 are referring to the same movie review dataset in this thesis.

Training `arch_opt4imdb_dataset_mnist` on MNIST using `Python2`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\mathrm{tr}}}$	0.5527	0.3076	0.2615	0.2377	0.2248	0.2137	0.2044	0.1999	0.1912	0.1884
$\mathcal{L}_{\mathbb{D}_{\mathrm{va}}}$	0.2889	0.2297	0.1835	0.1684	0.1622	0.1757	0.1413	0.1334	0.1507	0.1382
$\hat{\mathrm{acc}}(\mathbb{D}_{\mathrm{tr}})$	0.8219	0.9028	0.9188	0.9255	0.9294	0.9308	0.9346	0.9360	0.9391	0.9395
$\mathrm{acc}(\mathbb{D}_{\mathrm{va}})$	0.9438	0.9580	0.9655	0.9663	0.9695	0.9696	0.9701	0.9726	0.9703	0.9741

Table 13: Evaluation metrics during the training of the classifier based on the MNIST dataset, that was produced by the IMDb architecture after its modifications to allow for such a training, using `Python2`.

Training `arch_opt4imdb_dataset_imdb` on IMDb using `Python2`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\mathrm{tr}}}$	0.6102	0.4099	0.3522	0.3188	0.2999	0.2828	0.2705	0.2614	0.2543	0.2445
$\mathcal{L}_{\mathbb{D}_{\mathrm{va}}}$	0.4088	0.3205	0.3282	0.2793	0.2727	0.2693	0.2574	0.2680	0.2513	0.2499
$\hat{\mathrm{acc}}(\mathbb{D}_{\mathrm{tr}})$	0.6376	0.8165	0.8488	0.8684	0.8751	0.8840	0.8889	0.8961	0.8989	0.9022
$\mathrm{acc}(\mathbb{D}_{\mathrm{va}})$	0.8331	0.8692	0.8682	0.8901	0.8936	0.8964	0.8988	0.8916	0.9001	0.9005

Table 14: Evaluation metrics during the training of the classifier based on the IMDb dataset, that was produced by the IMDb architecture using `Python2`.

Training `arch_opt4imdb_dataset_mnist` on MNIST using `Python3`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\mathrm{tr}}}$	0.5527	0.3071	0.2622	0.2382	0.2247	0.2138	0.2049	0.1998	0.1904	0.1886
$\mathcal{L}_{\mathbb{D}_{\mathrm{va}}}$	0.2893	0.2291	0.1819	0.1677	0.1639	0.1755	0.1430	0.1352	0.1586	0.1444
$\hat{\mathrm{acc}}(\mathbb{D}_{\mathrm{tr}})$	0.8215	0.9032	0.9184	0.9252	0.9293	0.9312	0.9342	0.9361	0.9392	0.9398
$\mathrm{acc}(\mathbb{D}_{\mathrm{va}})$	0.9447	0.9577	0.9662	0.9666	0.9690	0.9700	0.9704	0.9724	0.9691	0.9734

Table 15: Evaluation metrics during the training of the classifier based on the MNIST dataset, that was produced by the IMDb architecture after its modifications to allow for such a training, using `Python3`.

Training `arch_opt4imdb_dataset_imdb` on IMDb using `Python3`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\mathrm{tr}}}$	0.5849	0.4025	0.3456	0.3170	0.3031	0.2835	0.2708	0.2639	0.2565	0.2471
$\mathcal{L}_{\mathbb{D}_{\mathrm{va}}}$	0.3976	0.3204	0.3300	0.2783	0.2855	0.2737	0.2557	0.2664	0.2496	0.2488
$\hat{\mathrm{acc}}(\mathbb{D}_{\mathrm{tr}})$	0.6707	0.8224	0.8543	0.8680	0.8738	0.8862	0.8888	0.8920	0.8975	0.9031
$\mathrm{acc}(\mathbb{D}_{\mathrm{va}})$	0.8437	0.8735	0.8642	0.8912	0.8902	0.8944	0.8998	0.8940	0.9022	0.9012

Table 16: Evaluation metrics during the training of the classifier based on the IMDb dataset, that was produced by the IMDb architecture using `Python3`.

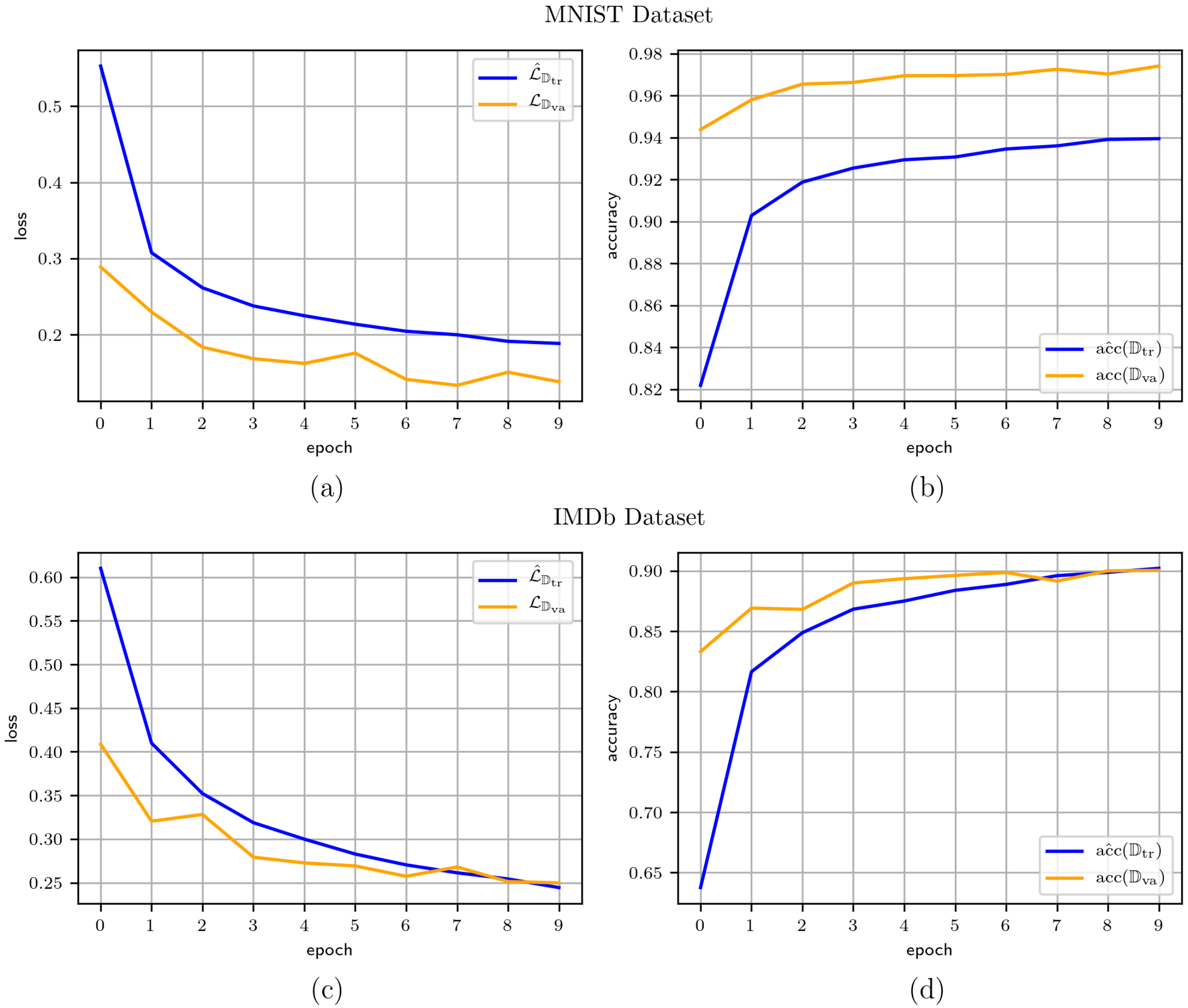


Figure 22: Performance graphs of the classifiers' trainings produced using the architectures which were originally tuned for the IMDb task in `Python2`. (a) and (b) correspond to the produced model using `arch_opt4imdb_dataset_mnist` which was trained on the MNIST dataset. (c) and (d) correspond to the produced model using `arch_opt4imdb_dataset_imdb` which was trained on the IMDb dataset.

`Python3`, Trainings Based on the Architecture Tuned for IMDb

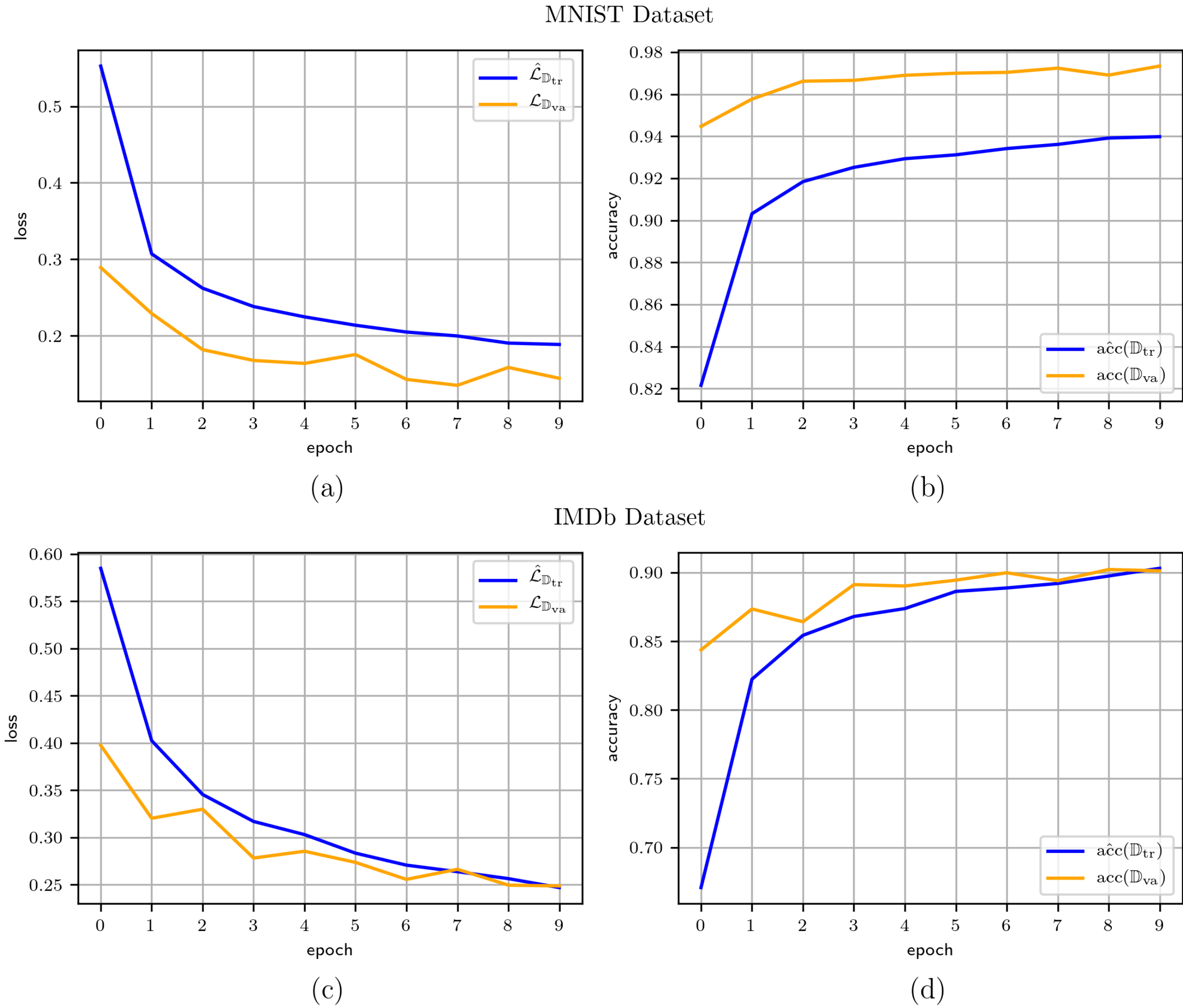


Figure 23: Performance graphs of the classifiers' trainings produced using the architecture which were tuned for the IMDb task in `Python3`. (a) and (b) correspond to the produced model using `arch_opt4imdb_dataset_mnist` which was trained on the MNIST dataset. (c) and (d) correspond to the produced model using `arch_opt4imdb_dataset_imdb` which was trained on the IMDb dataset.

3.3.3 Training Time

Training Time using Python2

- `arch_opt4mnist_dataset_mnist` takes approximately 265 seconds to complete an epoch.
- `arch_opt4mnist_dataset_imdb` takes approximately 122 seconds to complete an epoch.
- `arch_opt4imdb_dataset_mnist` takes approximately 54 seconds to complete an epoch.
- `arch_opt4imdb_dataset_imdb` takes approximately 132 seconds to complete an epoch.

Training Time using Python3

- `arch_opt4mnist_dataset_mnist` takes approximately 43 seconds to complete an epoch.
- `arch_opt4mnist_dataset_imdb` takes approximately 42 seconds to complete an epoch.
- `arch_opt4imdb_dataset_mnist` takes approximately 9 seconds to complete an epoch.
- `arch_opt4imdb_dataset_imdb` takes approximately 40 seconds to complete an epoch.

The execution time is notably longer in the `Python2` trainings mainly because of the additional configuration through the function

`tf.ConfigProto`

where `tf` is the alias for the `TensorFlow` v1.13.1 module, invoked for the `Python2` implementations. This additional RNG configuration was necessary for the experiments' reproducibility[9].

3.4 Experiments Part II

These experiments were conducted using the methodology proposed in 2.4 using `Python` 3.

[9] https://www.tensorflow.org/api_docs/python/tf/compat/v1/ConfigProto

3.4.1 | Experiments Based on the `mnistsrc` Source Model

The evaluation metrics during the training of the source model and its target models produced by the transfer learning associated with the `mnistsrc` architecture which was tuned for the MNIST dataset, are shown in the following tables and diagrams. The source architecture was tuned for the MNIST task. Table 17 demonstrates the performance of the pre-training part which is the training of the source model with architecture `mnistsrc`, on the MNIST dataset. Table 18 demonstrates the performance of the fine-tuning part on the same dataset as the source model with no changes in the architecture `mnistsrc_mnisttgt` relative to the `mnistsrc` architecture. Table 19 demonstrates the performance of the fine-tuning part on the IMDb dataset, where the target architecture is `mnistsrc_imdbtgt`. Figure 24 shows the respective diagrams.

Training `mnistsrc` on MNIST										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.2112	0.0804	0.0606	0.0513	0.0462	0.0413	0.0384	0.0361	0.0354	0.0333
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.0609	0.0462	0.0437	0.0354	0.0350	0.0333	0.0309	0.0347	0.0302	0.0361
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.9360	0.9761	0.9817	0.9847	0.9862	0.9877	0.9887	0.9895	0.9891	0.9905
$\text{acc}(\mathbb{D}_{\text{va}})$	0.9829	0.9869	0.9864	0.9901	0.9894	0.9905	0.9909	0.9906	0.9919	0.9915

Table 17: Evaluation metrics during the training of `mnistsrc` on the MNIST dataset.

Training `mnistsrc_mnisttgt` on MNIST										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.0320	0.0302	0.0311	0.0290	0.0289	0.0270	0.0263	0.0255	0.0256	0.0248
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.0297	0.0304	0.0322	0.0304	0.0309	0.0302	0.0319	0.0322	0.0354	0.0316
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.9899	0.9908	0.9904	0.9913	0.9915	0.9919	0.9920	0.9919	0.9923	0.9923
$\text{acc}(\mathbb{D}_{\text{va}})$	0.9918	0.9912	0.9924	0.9916	0.9916	0.9929	0.9921	0.9922	0.9914	0.9927

Table 18: Evaluation metrics during the training of `mnistsrc_mnisttgt` on the MNIST dataset, where the initial trainable parameters were copied from the respective trained source model of `mnistsrc`.

Training `mnistsrc_imdbtgt` on IMDb										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.7266	0.7180	0.7116	0.7076	0.7049	0.7028	0.6993	0.6990	0.6988	0.6988
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.7091	0.7036	0.6997	0.6971	0.6955	0.6946	0.6940	0.6936	0.6934	0.6933
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.4959	0.4985	0.5010	0.4954	0.4954	0.4965	0.4993	0.5024	0.5009	0.4951
$\text{acc}(\mathbb{D}_{\text{va}})$	0.5030	0.5030	0.5030	0.5030	0.5030	0.5030	0.5030	0.5030	0.5030	0.5030

Table 19: Evaluation metrics during the training of `mnistsrc_imdbtgt` on the IMDb dataset, where the initial trainable parameters were copied from the respective trained source model of `mnistsrc`.

Trainings Based on the Source Model with Architecture mnistsrc

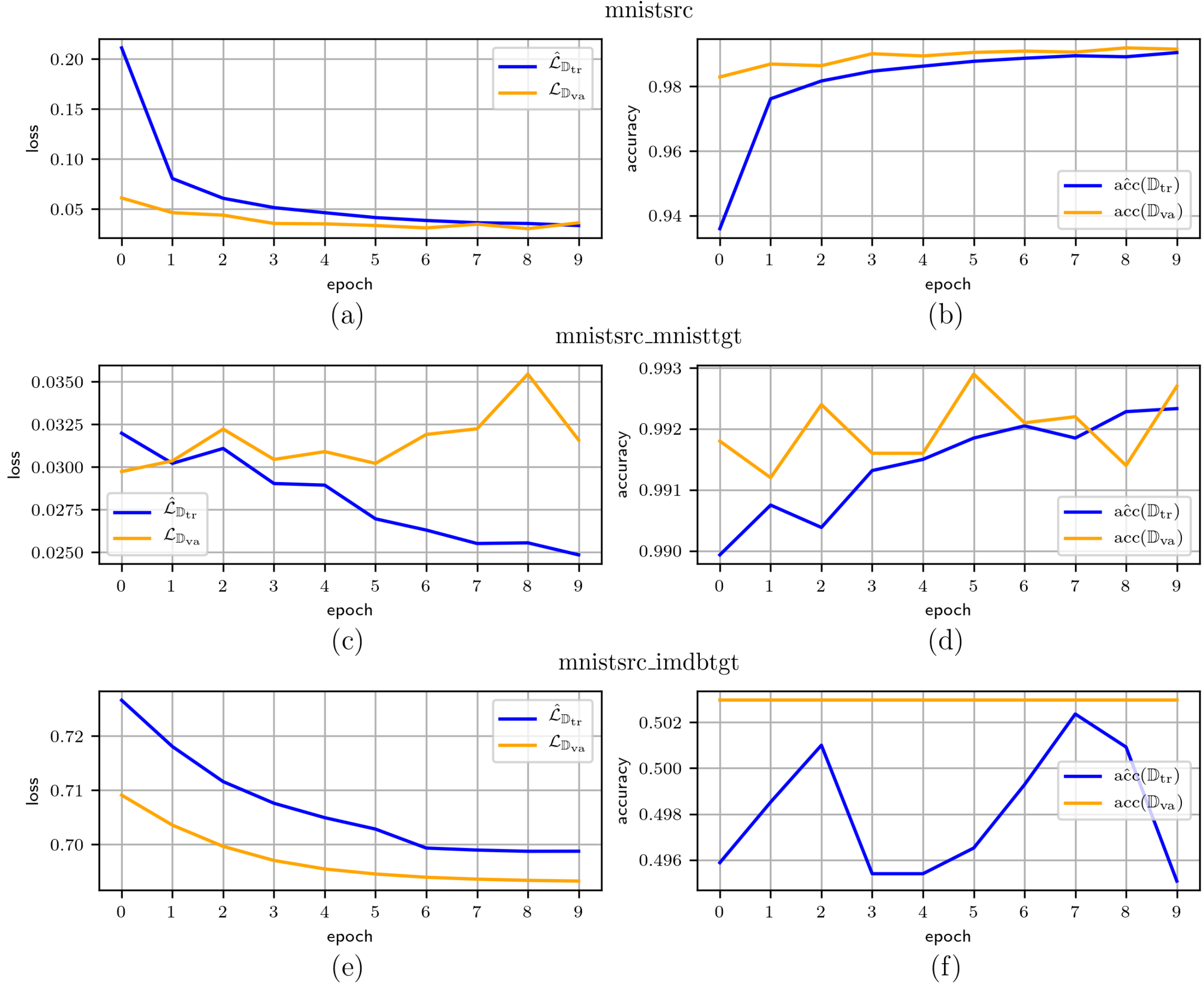


Figure 24: Performance graphs of the classifiers' trainings based on the `mnistsrc`'s trained source model. (a) and (b) correspond to the training of the source model with architecture `mnistsrc`, on the MNIST dataset. (c) and (d) correspond to the training of the target model with architecture `mnistsrc_mnisttgt` on the MNIST dataset, while (e) and (f) correspond to the training of the target model with architecture `mnistsrc_imdbtgt` on the IMDb dataset. Both of the target models were based on the mentioned source model.

3.4.2 | Experiments Based on the `imdbsrc` Source Model

The evaluation metrics during the training of the source model and its target models produced by the transfer learning associated with the **`imdbsrc`** architecture which was tuned for the IMDb dataset, are shown in the following tables and diagrams. The source architecture was tuned for the IMDb task. Table 20 demonstrates the performance of the pre-training part which is the training of the source model with architecture **`imdbsrc`**, on the IMDb dataset. Table 21 demonstrates the performance of the fine-tuning part on the same dataset as the source model with no changes in the architecture **`imdbsrc_imdbtgt`** relative to the **`imdbsrc`** architecture. Table 22 demonstrates the performance of the fine-tuning part on the MNIST dataset, where the target architecture is **`imdbsrc_mnisttgt`**. Figure 25 shows the respective diagrams.

Training `imdbsrc`										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.6971	0.6726	0.6496	0.6303	0.6089	0.5973	0.5856	0.5746	0.5667	0.5581
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.6784	0.6544	0.6277	0.6054	0.5884	0.5746	0.5678	0.5589	0.5511	0.5521
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.5328	0.5830	0.6157	0.6419	0.6584	0.6723	0.6830	0.6957	0.6984	0.7076
$\text{acc}(\mathbb{D}_{\text{va}})$	0.6092	0.6423	0.6718	0.6819	0.6910	0.6990	0.7080	0.7041	0.7149	0.7097

Table 20: Evaluation metrics during the training of `imdbsrc` on the IMDb dataset.

Training `imdbsrc_imdbtgt` on IMDb										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	0.6084	0.5969	0.5899	0.5844	0.5770	0.5752	0.5654	0.5632	0.5551	0.5499
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	0.5909	0.5802	0.5743	0.5748	0.5697	0.5675	0.5649	0.5577	0.5550	0.5495
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.6566	0.6678	0.6734	0.6794	0.6896	0.6907	0.6970	0.7010	0.7071	0.7132
$\text{acc}(\mathbb{D}_{\text{va}})$	0.6724	0.6825	0.6843	0.6899	0.6932	0.6950	0.6984	0.7030	0.7068	0.7087

Table 21: Evaluation metrics during the training of `imdbsrc_imdbtgt` on the IMDb dataset, where the initial trainable parameters were copied from the respective trained source model of `imdbsrc`.

Training `imdbsrc_mnisttgt` on MNIST										
	0	1	2	3	4	5	6	7	8	9
$\hat{\mathcal{L}}_{\mathbb{D}_{\text{tr}}}$	6.7460	6.1296	5.6088	5.1579	4.7634	4.4514	4.1984	3.9790	3.7935	3.6294
$\mathcal{L}_{\mathbb{D}_{\text{va}}}$	4.9574	4.4590	4.0436	3.7057	3.4350	3.2169	3.0429	2.9036	2.7920	2.7019
$\hat{\text{acc}}(\mathbb{D}_{\text{tr}})$	0.1010	0.1014	0.1027	0.1039	0.1046	0.1072	0.1088	0.1089	0.1092	0.1113
$\text{acc}(\mathbb{D}_{\text{va}})$	0.1047	0.1045	0.1021	0.0976	0.0885	0.0860	0.0851	0.0880	0.0946	0.0992

Table 22: Evaluation metrics during the training of `imdbsrc_mnisttgt` on the MNIST dataset, where the initial trainable parameters were copied from the respective trained source model of `imdbsrc`.

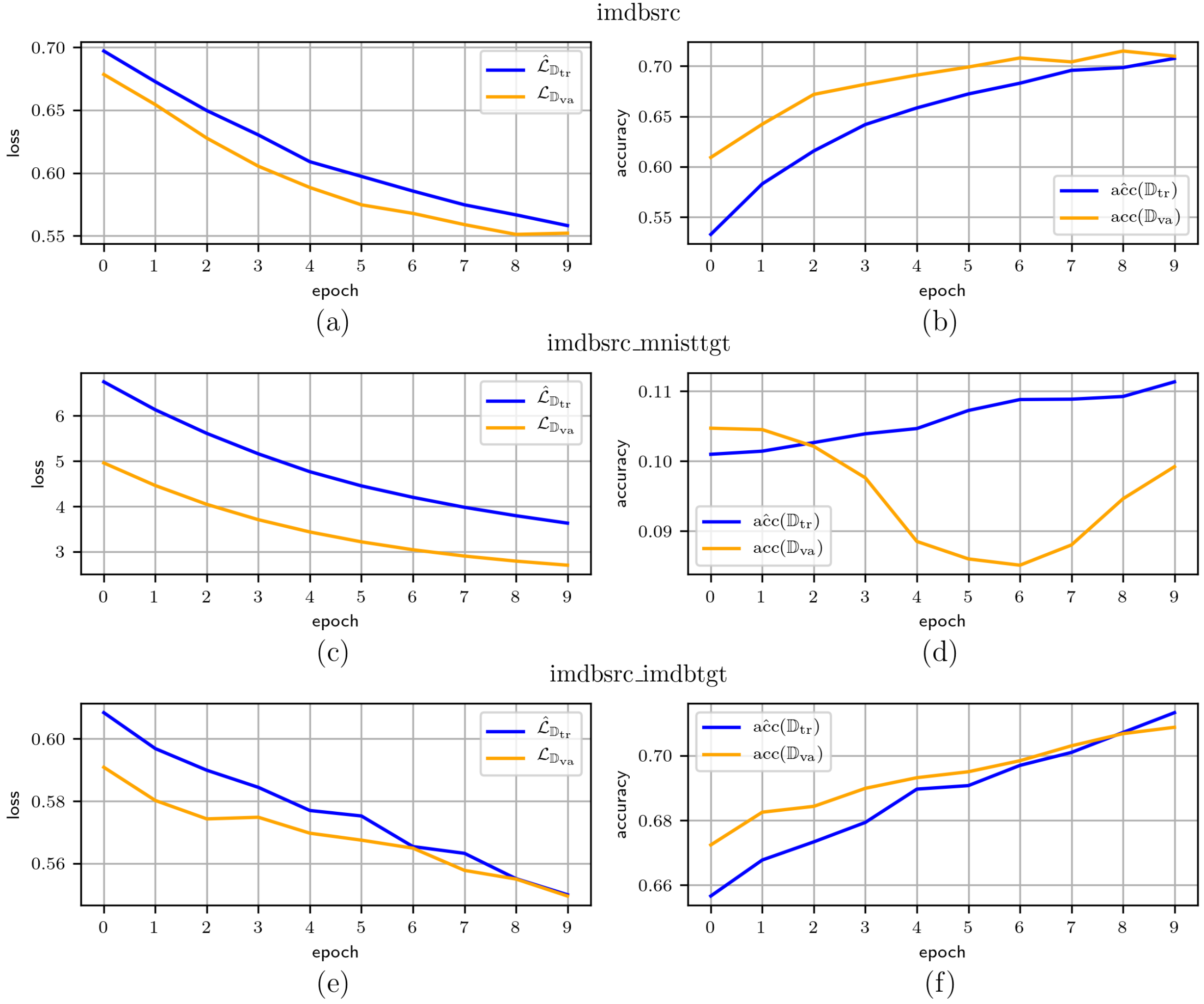


Figure 25: Performance graphs of the classifiers' trainings based on the `imdbsrc`'s trained source model. (a) and (b) correspond to the training of the source model with architecture `imdbsrc`, on the IMDb dataset. (c) and (d) correspond to the training of the target model with architecture `imdbsrc_mnisttgt` on the MNIST dataset, while (e) and (f) correspond to the training of the target model with architecture `imdbsrc_imdbtgt` on the IMDb dataset. Both of the target models were based on the mentioned source model.

3.4.3 | Training Time

Training Time of Pre-Training

- `mnistsrc` takes approximately 37s seconds to complete an epoch.
- `imdbsrc` takes approximately 6s seconds to complete an epoch.

Training Time of Fine-Tuning

- `mnistsrc_mnisttgt` takes approximately 29s seconds to complete an epoch.
- `mnistsrc_imdbtgt` takes approximately 16s seconds to complete an epoch.
- `imdbsrc_mnisttgt` takes approximately 4s seconds to complete an epoch.
- `imdbsrc_imdbtgt` takes approximately 3s seconds to complete an epoch.

4 | Discussion

About Experiments Part I

For this group of experiments there are no considerable differences between the Python2 and Python3 implementations.

Using an architecture which was tuned for the MNIST dataset, for the training on the IMDb dataset after the adjustment, part of which was the conversion of the 2D to 1D layers in the feature extraction component (Architecture's Table 3), the trained model performs well enough but is slightly worse than the model trained for the IMDb dataset where the final one was based on the architecture (Architecture's Table 2) which was tuned for the IMDb dataset (Performance Tables 10, 14, 12, 16). Judging by the loss metric, the adjusted model has a mild tendency to overfit after epoch 2 in the case of Python2, while in Python3 this happens after epoch 3. This makes sense because the adjusted architecture is set to have many more parameters (6 407 763 parameters) compared to the number of parameters of the architecture that was tuned on the IMDb dataset and used for trainings on the same dataset (4 977 571 parameters). The model with architecture tuned for the IMDb dataset does not show any signs of overfitting behavior during its training as expected.

Conversely, taking the architecture which was tuned for the IMDb dataset and applying it to the MNIST dataset with the necessary adjustments (Architecture's Table 4), the model performs slightly worse than the model corresponding to the architecture (Architecture's Table 1) which was tuned for the MNIST dataset (Performance Tables 13, 9, 15, 11). The adjusted model has validation accuracy 0.9734 which is noticeably lower than the 0.9918 validation accuracy of the model that was training on the MNIST dataset whose architecture was tuned for the MNIST dataset. A possible reason for this performance reduction is because the total number of parameters

is lower in the case of the adjusted architecture (135 810 parameters) compared to the architecture tuned for the MNIST dataset which is used for the training on that same dataset (1 199 882 parameters), and potentially because of the dropout layer immediately after the input layer with a drop rate of 50% which may ruin important patterns inside the input's image. The combination of these two facts imply that the adjusted model's architecture learns a lot less features with the initial dropout causing a bottleneck in the information received by the following layers.

About Experiments Part II

Considering the architecture which was tuned for the MNIST dataset (Architecture's Table 5), the training (Table 17) of the model seems to be optimal at epoch 8 with validation accuracy 0.9919, after which it drops by a trivial amount to 0.9915. Continuing the training on the same dataset (Table 18) the resulting model has a trivially higher validation accuracy 0.9927 where the optimal one is at epoch 5 with validation accuracy 0.9929. The training shows a minor sign of overfitting because the optimal validation loss remains at epoch 0 after which it fluctuates around the same values while the training loss continues to improve. On the other hand, using the same source model for the training on the IMDb dataset, with the appropriate adjustments (Architecture's Table 7), the transfer learning has failed. The reason is that during the training (Table 19), in all epochs the validation and training accuracy oscillates around values close to $0.5 = 1/c$, where $c = 2$ is the number of classes for the IMDb's task. This means that the trained model in each epoch performs like a random classifier and uses excessive computational resources. This classifier is obviously impractical under all circumstances, thus at this point we have accepted that the transfer learning from an image dataset to a text dataset has failed. Compared to the architecture which was tuned for the IMDb dataset, the reduction of the input vector's size from 400 to 28 with the embedding dimension reduced from 50 to 28 could've had an impact on that, but it seems like a major reason was the usage of 2D layers to a text dataset, instead of 1D. It makes little sense to extract the features of a very small part of each embedding vector (e.g. 3 coordinates) every time and this is probably why such CNNs cannot work on embedded text datasets.

Conversely, having the architecture which was tuned for the IMDb dataset (Architecture's Table 6), the training (Table 20) of the model should have stopped at epoch 8 where the validation accuracy is the highest 0.7149 with the final one being 0.7097. The validation accuracy is notably lower compared to the 0.9012 (Table 16) produced by the original architecture for the IMDb dataset. The reason for the big gap is because `imdbsrc` has 28 input neurons with 28 being the size of the embedding dimension while `arch_opt4imdb_dataset_imdb` has 400 input neurons and 50 is the size of the embedding dimension, thus this accuracy reduction makes sense in that regard. Additionally, this source architecture contains a lower amount of parameters (1 086 625 parameters) compared to the original architecture for IMDb trainings (4 977 571 parameters). Continuing the training for the same number of epochs (Table 21) the final model with validation accuracy 0.7087 doesn't show any signs of significant improvement which is a strong sign of underfitting. On the other hand,

using this model as a source model for the training on the MNIST dataset and by adjusting the architecture (Architecture's Table 8), we observe an identical behavior during the training, as in the case of the other transfer learning procedure (Table 22). Validation and training accuracy once again are oscillating around $0.1 = 1/c$. This time the number of classes c is obviously 10, thus this model behaves like a random classifier as in the other case. Another possible justification for the bad performance is the number of parameters which is 15 160, and compared to the 1 199 882 of the original architecture's for the MNIST dataset, it is extremely lower. We can observe that the validation loss is decreasing after each epoch which raises the hope for more successful models produced by trainings with more epochs.

5 | Conclusions

We made an attempt to test if we can successfully train a CNN using some properties of another CNN whose architecture was tuned for a different problem based on a different type of dataset. Moreover, we decided to use the MNIST and IMDb datasets along with their tuned CNN architectures and their respective training algorithms from external sources. We then trained these models and have found out that they work fine on the datasets they were supposed to be trained on. Then we have switched the tasks for each of these architectures. 2D layers and 1D layers have been also swapped while maintaining the rest of their hyperparameters. We have also included the embedding layer in the case of the training on the IMDb dataset and removed the embedding layer in the case of the training on the MNIST dataset. The reason for these changes is because 2D layers are supposed to work on image data and 1D layers on text data and the embedding layer has been designed to work for text data. These modifications resulted in well performing final models with some mild overfitting and underfitting behaviors during their trainings. This transference has worked relatively fine, as the trainings of the swapped models showed high enough validation accuracies with minor differences compared to the default trainings. We have also used transfer learning, which has prevented us from swapping 1D with 2D layers. The target models performed as random classifiers. We suspect that this issue is caused because the transfered 1D kernels learned document features which are not applicable to image data, and because the transfered 2D kernels learned image features which are not applicable to document data, ending up *trapping* the initial target model in a state where it has to unlearn irrelevant noise in each case. Additionally, the processing of a 2D convolutional layer with a small kernel has little meaning, as each embedded word is supposed to be treated as a whole thing. That is because each word in a document is the equivalent of a pixel in an image.

A potential future work may be focused on investigating why such transfer learning methods from image to text data (and vice versa) fail. Was this training issue caused by the transference of 1D to 2D trained layers? Is the usage of 2D (1D) layers to text (image) data the problem? Are the dense layers contributing to the problem? In order to reveal more about the hypothesis, all these questions should be

answered first. Additionally, increasing the number of epochs for the fine tuning may be another approach for a more successful application of transfer learning. Finally we could investigate the following transfer learning methodology: use 2D (1D) layers to train a source model on text (image) data and proceed for the fine tuning on image (text) data using these trained hidden layers.

Appendices

The code described here can be found in [50].

Appendix I

Here we present the Python code responsible for the parsing and preprocessing of the MNIST and IMDb datasets' files. The file containing these parsing and preprocessing functions is named as `data_helpers.py` . The chunk of code that imports all of the necessary libraries for the `data_helpers.py` file can be found in Algorithm 1.

`load_local_mnist_dataset` (MNIST Dataset)

The function `load_local_mnist_dataset` (Algorithm 2) loads the MNIST dataset from a local directory. The inputs are

- <num_classes>: int, total number of classes for the classification task.
- <img_rows>: int, total number of the feature's image matrix rows.
- <img_cols>: int, total number of the feature's image matrix columns.

and it returns

- <dataset>: tuple, contains 2 tuples each containing 2 elements. It is the dataset after it has been preprocessed. <dataset> = (<x_train>, <y_train>), (<x_test>, <y_test>):
- <x_train>: numpy.ndarray, the training set's feature tensor, has shape (<predefined_number_of_training_instances>, <predefined_number_of_rows>, <predefined_number_of_cols>, 1).
- <y_train>: numpy.ndarray, ground truth of the test set with shape (<predefined_number_of_training_instances>, <predefined_number_of_classes>).
- <x_test>: numpy.ndarray, the test set's feature tensor, has shape (<predefined_number_of_test_instances>, <predefined_number_of_feature_rows>, <predefined_number_of_feature_cols>, 1).
- <y_test>: numpy.ndarray, ground truth of the test set with shape (<predefined_number_of_test_instances>, <predefined_number_of_classes>).

`mnist_channel_axis_add` (MNIST Dataset)

The function `mnist_channel_axis_add` (Algorithm 3) adds another axis to the input, and returns that expanded input. The inputs are

- <x>: numpy.ndarray, the dataset's feature tensor in which there is going to be assigned an additional axis to it. Has shape (<number_of_instances>, <number_of_feature_rows>, <number_of_feature_cols>, 1).

- <img_rows>: int, total number of the features' image matrix rows.
- <img_cols>: int, total number of the features' image matrix columns.

and it returns

- <x_expanded_axis>: numpy.ndarray, the input tensor with an additional axis.

`preprocess_image_dataset` (MNIST Dataset)

The function `preprocess_image_dataset` (Algorithm 4) preprocesses the given dataset, and returns that expanded input. This is where the main preprocessing of the dataset is implemented. The feature images are normalized, the ground truth variables are mapped to their OHE equivalents, the instances are shuffled inside their respective tensors and the dataset is split into a training and test set respectively. The inputs are

- <x>: numpy.ndarray, the dataset's feature tensor. Has shape (<number_of_instances>, <number_of_feature_rows>, <number_of_feature_cols>, 1).
- <y>: numpy.ndarray, the dataset's ground truth tensor. Has shape (<number_of_instances>, <number_of_classes>). <num_classes>: int, total number of classes.

and it returns

- <dataset>: tuple, 2 tuples each containing 2 elements. It is the dataset after it has been preprocessed. <dataset> = (<x_train>, <y_train>), (<x_test>, <y_test>):
    - <x_train>: numpy.ndarray, the training set's feature tensor, has shape (<predefined_number_of_training_instances>, <predefined_number_of_rows>, <predefined_number_of_cols>, 1).
    - <y_train>: numpy.ndarray, ground truth of the test set in OHE with shape (<predefined_number_of_training_instances>, <predefined_number_of_classes>).
    - <x_test>: numpy.ndarray, the test set's feature tensor, has shape (<predefined_number_of_test_instances>, <predefined_number_of_feature_rows>, <predefined_number_of_feature_cols>, 1).
    - <y_test>: numpy.ndarray, ground truth of the test set in OHE with shape (<predefined_number_of_test_instances>, <predefined_number_of_classes>).

`load_local_imdb_dataset` (IMDb Dataset)

The function `load_local_imdb_dataset` (Algorithm 5) loads and preprocesses data from the IMDb dataset. Returns input vectors, labels, vocabulary, and the inverse vocabulary. The inputs are

- <sequence_length>: int, the resulting feature instances are all going to have an equal <sequence_length> length. This is the sequence length.

and it returns

- <dataset_vocabulary>: list, contains 3 elements where the first two elements are tuples each composed by 2 elements. <dataset_vocabulary> = [(<x_train>, <y_train>), (<x_test>, <y_test>), <vocabulary_inv>]:
  - <x_train>: numpy.ndarray, the training feature matrix with shape (<number_of_training_instances>, <sequence_length>).
  - <y_train>: numpy.ndarray, the training ground truth vector with shape (<number_of_training_instances>).
  - <x_test>: numpy.ndarray, the test feature matrix with shape (<number_of_test_instances>, <sequence_length>).
  - <y_test>: numpy.ndarray, the test ground truth vector with shape (<number_of_test_instances>).
  - <vocabulary_inv>: dict, the inverted vocabulary defined as per the function <build_vocab>.

`clean_str` (IMDb Dataset)

The function `clean_str` (Algorithm 6) is a tokenization/string cleaning program. Original taken from [51]. The inputs are

- <string>: str.

and it returns

- <string_cleaned>: str.

`pad_sentences` (IMDb Dataset)

The function `pad_sentences` (Algorithm 7) pads all sentences to the same length which is set to be equal to the maximum number of tokens among all instance texts. The length is defined by the longest sentence and returns padded sentences. The inputs are

- <sentences>: list, is consisted of all of the dataset's features. Contains <number_of_instances> elements which are all lists, where each of these lists is a feature instance that contains token (word) strings. These are splitted tokens from the text corresponding to that particular instance.

- <maxlen>: int, default value: -1, after each text instance has been padded, -(<maxlen>+1) tokens from the right towards the left are dropped.
- <padding_word>: str, default value: '<PAD/>', this is the padding token.

and it returns

- <padded_reduced_sentences>: list, contains the same text elements as <sentences> but each text element is now padded and trimmed from the right so that all text elements share the same number of token elements, which is set to be equal to <maxlen>.

`build_vocab` (IMDb Dataset)

The function `build_vocab` (Algorithm 8) builds a vocabulary mapping from word to index based on the sentences and returns vocabulary mapping and inverse vocabulary mapping. The inputs are

- <sentences>: list, contains the text features. All the text features share the same length <maxlen>. Also note that some of the raw feature set's tokens may be missing.

and it returns

- <vocabulary_pair>: list, contains 2 elements, <vocabulary_pair> = [<vocabulary>, <vocabulary_inv>]:
    - <vocabulary>: dict, each key is the vocabularies token in str type and each value is the unique token's index which is an int type, beginning from 0 and ending to the total number of unique tokens found inside the feature set, each time incrementing the key value by 1. The zeroth value contains the padding token, and beginning from key with value 1 until the maximum dictionary key value, the tokens are sorted in descending order by the frequency of their token appearances inside the feature set.
    - <vocabulary_inv>: list, each element is a vocabularies token in str. Each list index versus string word pair is the same as in <vocabulary>. It can be seen as the inverse of <vocabulary> in regards to its key-value pair.

`build_input_data` (IMDb Dataset)

The function `build_input_data` (Algorithm 9) generates the vocabulary based on the dataset. The tokens are substituted by their respective vocabulary indices inside the feature set, and because text instances have equal size, the feature list can be converted into a feature tensor, which is invoked here. The inputs are

- <sentences>: list, contains the text features. All the text features share the same length <maxlen>.

- <labels>: numpy.ndarray, the ground truth vector.

and it returns

- <dataset>: list, consisted of 2 elements. <dataset> = [x, y]:
    - <x>: numpy.ndarray, the feature matrix, has shape (<number_of_instances>, <maxlen>).
    - <y>: numpy.ndarray, the ground truth vector, has shape (<number_of_instances>).

Algorithm 1 Libraries (`data_helpers.py`)

```
1: import io
2: import re
3: import itertools
4: from collections import Counter
5: import idx2numpy
6: import numpy as np
7: from tensorflow.python.keras import backend as K
8: from keras.datasets import mnist
9: from keras.utils import to_categorical
```

Algorithm 2 The `load_local_mnist_dataset` function

```
1: def load_local_mnist_dataset(num_classes, img_rows, img_cols):

2:     rel_path_dir = '../../datasets/MNIST'
3:     rel_path_x_train = rel_path_dir+'/train/train-images-idx3-ubyte'

4:     rel_path_y_train = rel_path_dir+'/train/train-labels-idx1-ubyte'

5:     rel_path_x_test = rel_path_dir+'/test/t10k-images-idx3-ubyte'
6:     rel_path_y_test = rel_path_dir+'/test/t10k-labels-idx1-ubyte'

7:     x_train = idx2numpy.convert_from_file(rel_path_x_train)
8:     y_train = idx2numpy.convert_from_file(rel_path_y_train)
9:     x_test = idx2numpy.convert_from_file(rel_path_x_test)
10:    y_test = idx2numpy.convert_from_file(rel_path_y_test)

11:    x = mnist_channel_axis_add(x=np.concatenate((x_train, x_test),
   axis=0), img_rows=img_rows, img_cols=img_rows)
12:    y = np.concatenate((y_train, y_test), axis=0)

13:    (x_train, y_train), (x_test, y_test) = preprocess_image_dataset(x,
   y, num_classes=num_classes)

14:    return (x_train, y_train), (x_test, y_test)
```

Algorithm 3 The `mnist_channel_axis_add` function

```
1: def mnist_channel_axis_add(x, img_rows, img_cols):

2:     if K.image_data_format() == 'channels_first':
3:         print('Warning:  Channels axis is before rows and columns.')
4:         x = x.reshape(x.shape[0], 1, img_rows, img_cols)
5:     else:
6:         x = x.reshape(x.shape[0], img_rows, img_cols, 1)

7:     return x
```

Algorithm 4 The `preprocess_image_dataset` function

```
1:  def preprocess_image_dataset(x, y, num_classes):

2:      ## Dataset format modifier
3:      x.astype('float32')

4:      ## Feature normalizer
5:      x = x/255.

6:      ## One hot encoding for classes
7:      y = to_categorical(y, num_classes)

8:      ## Dataset shuffler
9:      shuffle_indices = np.random.permutation(np.arange(len(y)))
10:     x = x[shuffle_indices]
11:     y = y[shuffle_indices]

12:     ## Dataset splitter
13:     train_len = int(len(y) * 6./7.)
14:     x_train = x[:train_len]
15:     y_train = y[:train_len]
16:     x_test = x[train_len:]
17:     y_test = y[train_len:]

18:     return (x_train, y_train), (x_test, y_test)
```

Algorithm 5 The `load_local_imdb_dataset` function

```
1: def load_imdb_data_and_labels():

2:    # Load data from files
3:    positive_training_examples = io.open("../../datasets/IMDb/train
   -pos.txt", encoding='iso-8859-1').readlines()
4:    positive_training_examples = [s.strip() for s in
   positive_training_examples]
5:    negative_training_examples = io.open("../../datasets/IMDb/train
   -neg.txt", encoding='iso-8859-1').readlines()
6:    negative_training_examples = [s.strip() for s in
   negative_training_examples]

7:    positive_testing_examples = io.open("../../datasets/IMDb/test-
   pos.txt", encoding='iso-8859-1').readlines()
8:    positive_testing_examples = [s.strip() for s in
   positive_testing_examples]
9:    negative_testing_examples = io.open("../../datasets/IMDb/test-
   neg.txt", encoding='iso-8859-1').readlines()
10:   negative_testing_examples = [s.strip() for s in
   negative_testing_examples]

11:   ## ! Dataset concat: Begin

12:   positive_examples = positive_training_examples +
   positive_testing_examples
13:   negative_examples = negative_training_examples +
   negative_testing_examples

14:   ## Split by words
15:   x_text = positive_examples + negative_examples

16:   x_text = [clean_str(sent) for sent in x_text]
17:   x_text = [s.split(" ") for s in x_text]

18:   ## Generate labels
19:   positive_labels = [1 for _in positive_examples]
20:   negative_labels = [0 for _in negative_examples]

21:   y = np.concatenate([positive_labels, negative_labels], 0)

22:   ## ! Dataset concat: End

23:   return [x_text, y]
```

Algorithm 6 The `clean_str` function

```
1: def clean_str(string):

2:    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
3:    string = re.sub(r"\'s", " \'s", string)
4:    string = re.sub(r"\'ve", " \'ve", string)
5:    string = re.sub(r"n\'t", " n\'t", string)
6:    string = re.sub(r"\'re", " \'re", string)
7:    string = re.sub(r"\'d", " \'d", string)
8:    string = re.sub(r"\'ll", " \'ll", string)
9:    string = re.sub(r",", " , ", string)
10:   string = re.sub(r"!", " !  ", string)
11:   string = re.sub(r"\(", " \( ", string)
12:   string = re.sub(r"\)", " \) ", string)
13:   string = re.sub(r"\?", " \?  ", string)
14:   string = re.sub(r"\s{2,}", " ", string)

15:   return string.strip().lower()
```

Algorithm 7 The `pad_sentences` function

```
1: def pad_sentences(sentences, maxlen=-1, padding_word="<PAD/>"):

2:    sequence_length = max(len(x) for x in sentences)
3:    padded_sentences = []
4:    for i in range(len(sentences)):
5:       sentence = sentences[i]
6:       num_padding = sequence_length - len(sentence)
7:       new_sentence = sentence + [padding_word] * num_padding
8:       padded_sentences.append(new_sentence)

9:    ## A sentence has length equal to the length of the sentence
   with the greatest length of <sentence>, and the next line shrinks
   this length to <maxlen> size.

10:   padded_reduced_sentences = \
11:      [
12:         padded_sentences[sentence_idx][:maxlen]
13:         for sentence_idx in range(len(padded_sentences))
14:      ]

15:   return padded_reduced_sentences
```

Algorithm 8 The `build_vocab` function

```
1: def build_vocab(sentences):

2:    ## Counts the number of encounters of each word on the dataset,
   it resembles a dictionary where you input the word, and it returns
   it's number of encounters.
3:    word_counts = Counter(itertools.chain(*sentences))
4:    ## Mapping from index to word
5:    vocabulary_inv = [x[0] for x in word_counts.most_common()]
6:    ## Mapping from word to index
7:    vocabulary = {x:  i for i, x in enumerate(vocabulary_inv)}

8:    return [vocabulary, vocabulary_inv]
```

Algorithm 9 The `build_input_data` function

```
1: def build_input_data(sentences, labels, vocabulary):

2:    ## Converts a given sentence, which is a list consisted of
   string elements, to their respective vocabulary indices (0 is the
   padding or <PAD/>)
3:    x = np.array([[vocabulary[word] for word in sentence] for
   sentence in sentences])
4:    y = np.array(labels)

5:    return [x, y]
```

Appendix II

Here we present the Python code responsible for the word embeddings' training using a Word2Vec model based on the `gensim` library [52]. The embeddings are trained on the IMDb's feature set. The file is named as `w2v.py` . The chunk of code that imports all of the necessary libraries for the `w2v.py` file can be found in Algorithm 10.

Algorithm 10 Libraries (`w2v.py`)

```
1: import io
2: from __future__ import print_function
3: import os
4: from os.path import join, exists, split
5: import numpy as np
6: from gensim.models import word2vec
```

`train_word2vec`

The function `train_word2vec` (Algorithm 11) trains, saves, loads a Word2Vec model. The inputs are

- <sentence_matrix>: numpy.ndarray, has shape (<dataset_size>, <sequence_length>), where <dataset_size> is the size of the dataset and <sequence_length> is the size of each sequence or instance text.
- <vocabulary_inv>: dict int: str, inverted vocabulary.
- <num_features>: int, word vector dimensionality.
- <min_word_count>: int, minimum word count.
- <context>: int, context window size.

and it returns

- <embedding_weights>: dict, It contains the embedding vectors of each dictionary word. Each dictionary key is an identifier of each vocabulary word in int, and dictionary value is the embedding vector of the word it corresponds to.

Algorithm 11 The `train_word2vec` function

```
1: def train_word2vec(sentence_matrix, vocabulary_inv,
   num_features=300, min_word_count=1, context=10):

2:    model_dir = 'models'
3:    model_name = "{:d}features_{:d}minwords_{:d}context.mdl".format(
   num_features, min_word_count, context)
4:    model_name = join(model_dir, model_name)
5:    if exists(model_name):
6:       embedding_model = word2vec.Word2Vec.load(model_name)
7:       print('Load existing Word2Vec model \'%s\'' %
   split(model_name)[-1])
8:    else:
9:       ## Set values for various parameters
10:      num_workers = 2 # Number of threads to run in parallel
11:      downsampling = 1e-3 # Downsample setting for frequent words

12:      ## Initialize and train the model
13:      print('Training Word2Vec model...')
14:      sentences = [[vocabulary_inv[w] for w in s] for s in
   sentence_matrix]
15:      embedding_model = word2vec.Word2Vec(sentences,
   workers=num_workers,
16:      size=num_features, min_count=min_word_count,
17:      window=context, sample=downsampling)

18:      ## If we don't plan to train the model any further, calling
   init_sims will make the model much more memory-efficient.
19:      embedding_model.init_sims(replace=True)

20:      # Saving the model for later use.  You can load it later
   using Word2Vec.load()
21:      if not exists(model_dir):
22:      os.mkdir(model_dir)
23:      print('Saving Word2Vec model \'%s\'' % split(model_name)[-1])
24:      embedding_model.save(model_name)

25:   ## Add unknown words
26:   embedding_weights = {key:  embedding_model[word] if
   word in embedding_model else np.random.uniform(-0.25,
   0.25, embedding_model.vector_size) for key, word in
   vocabulary_inv.items()}

27:   return embedding_weights
```

Appendix III

Here we present the files containing each executable code that produces each of the experiments described in Experiments Part I. The chunk of code that imports all of the necessary libraries for these files can be found in Algorithm 12. In order for each of these codes to be reproducible, the code chunk in Algorithms 13 and 14 for Python 2 and Python 3 respectively is read by the Python interpreter beforehand, prior to any other line of code presented in this thesis. That is because most of the code injects pseudo-randomness into the respective preprocessing and training process. The rest of the code presented here applies for both Python 2 and Python 3.

Final Code Chunk

Every executable code of Experiments Part I, ends with the code chunk in Algorithm 15. The `model.fit` is the function that trains the already defined neural network. The `model.save` saves the model after it is trained and the final line of code saves the evaluation metrics into a spreadsheet file.

Algorithm 12 Libraries used by the executable files of Experiments Part I

```
1: from copy import deepcopy
2: import os
3: import random
4: import numpy as np
5: import pandas as pd
6: import tensorflow as tf
7: from keras.models import Sequential, Model
8: from keras.layers import Input, Dense, Dropout, Flatten, Conv2D,
   MaxPooling2D, Convolution1D, MaxPooling1D, Embedding, Concatenate
9: from keras.datasets import mnist, imdb
10: from keras.preprocessing import sequence
11: from keras.losses import binary_crossentropy,
    categorical_crossentropy
12: from keras.optimizers import Adadelta
13: from w2v import train_word2vec
14: import data_helpers
```

Algorithm 13 RNG seed for the experiments implemented in Python 2

```
1: random.seed(1)
2: np.random.seed(1)
3: session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
   inter_op_parallelism_threads=1)
4: sess = tf.Session(graph=tf.get_default_graph(),
   config=session_conf)
5: K.set_session(sess)
6: tf.set_random_seed(1)
```

Algorithm 14 RNG seed for the experiments implemented in Python 3

```
1: random.seed(1)
2: np.random.seed(1)
3: tf.random.set_seed(1)
```

Algorithm 15 Final code chunk for the executable code files of Experiments Part I

```
 1: model.summary()

 2: # Train the model
 3: model.fit\
 4: (
 5:    x_train,
 6:    y_train,
 7:    batch_size=batch_size,
 8:    epochs=num_epochs,
 9:    validation_data=(x_test, y_test),
10:    verbose=1
11: )

12: model.save(''.join((model_path, '.mdl')))

13: score = model.evaluate(x_test, y_test, verbose=0)

14: print('Test loss:  %.30f'%score[0])
15: print('Test accuracy:  %.30f'%score[1])

16: ## Store training output
17: metrics_history = pd.DataFrame.from_dict(model.history.history)
18: metrics_history = pd.concat([metrics_history['loss'],
    metrics_history['accuracy'], metrics_history['val_loss'],
    metrics_history['val_accuracy']], axis=1)
19: metrics_history.T.to_excel(''.join((model_path, '.xlsx')))

20: model.summary()
```

The Executable `arch_opt4_mnist_dataset_mnist.py`

This executable file contains the code through which the CNN with architecture `arch_opt4_mnist_dataset_mnist` can be trained on the MNIST dataset. The code is presented in Algorithms 16, 17 and 15. Table 1 shows the architecture and Figure 11 gives an visual intuition about its outputs layerwise. The source of this code is [39]. The training process and CNN architecture are tuned for the MNIST dataset.

The Executable `arch_opt4_mnist_dataset_imdb.py`

This executable file contains the code through which the CNN with architecture `arch_opt4_mnist_dataset_imdb` can be trained on the IMDb dataset. The code is presented in Algorithms 18, 19 and 15. Table 3 shows the architecture and Figure 13 gives an visual intuition about its outputs layerwise.

This code is a modified version of `arch_opt4_mnist_dataset_mnist.py`. The modifications allow for the training on the IMDb dataset instead. We use the `data_helpers.load_local_imdb_dataset` function in line 19 in Algorithm 16, which parses and preprocesses the IMDb dataset using the preprocessing functions of [53]. The lines 10 until 28 in Algorithm 20 have been added to make preparations for the initialization of the embedding weights which will be used in the newly added embedding layer 4 in Algorithm 21. The feature extractor component of the architecture specified by lines 5 until 10 in Algorithm 17 are modified to the lines 5 until 8 in Algorithm 21. This change converted all the 2D layers to their 1D equivalents, on the feature extractor component of the architecture. The cost function has been modified from the categorical cross entropy in line 18 in Algorithm 17 to the binary cross entropy in line 22 in Algorithm 21 and the output's layer activation function has been modified from the softmax in line 14 in Algorithm 17 to the sigmoid function in line 12 in Algorithm 21 with 1 output neuron.

Algorithm 16 The `arch_opt4_mnist_dataset_mnist.py` trainer file: Chunk 1

```
1: ## ! Configuration: Begin

2: ## Output model
3: model_name = 'arch_opt4_mnist_dataset_mnist'
4: model_path = './models/'+model_name

5: batch_size = 64
6: num_classes = 10
7: num_epochs = 10

8: ## input image dimensions
9: img_rows, img_cols = 28, 28

10: ## Data source
11: data_source = "local_dir" # "keras_data_set" or "local_dir"

12: ## ! Configuration: End

13: def load_data(data_source):
14:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
15:    if data_source == "keras_data_set":
16:       (x_train, y_train), (x_test, y_test) = mnist.load_data()
17:    else:
18:       (x_train, y_train), (x_test, y_test) = \
19:       data_helpers.load_local_mnist_dataset\
20:       (
21:          num_classes=num_classes,
22:          img_rows=img_rows,
23:          img_cols=img_cols
24:       )

25:    return x_train, y_train, x_test, y_test

26: # Data Preparation
27: print("Load data...")
28: x_train, y_train, x_test, y_test = load_data(data_source)
```

Algorithm 17 The `arch_opt4_mnist_dataset_mnist.py` Trainer File: Chunk 2

```
1: ## !  Build model:  Begin

2: input_shape = x_train.shape[1:]

3: print('Input shape:  '  + str(input_shape))

4: model = Sequential()
5: model.add(Conv2D(32, kernel_size=(3, 3),
6:             activation='relu',
7:             input_shape=input_shape))
8: model.add(Conv2D(64, (3, 3), activation='relu'))
9: model.add(MaxPooling2D(pool_size=(2, 2)))
10: model.add(Dropout(0.25))
11: model.add(Flatten())
12: model.add(Dense(128, activation='relu'))
13: model.add(Dropout(0.5))
14: model.add(Dense(num_classes, activation='softmax'))

15: ## !  Build model:  End

16: model.compile\
17: (
18:    loss=categorical_crossentropy,
19:    optimizer=Adadelta(),
20:    metrics=["accuracy"]
21: )
```

Algorithm 18 The `arch_opt4_mnist_dataset_imdb.py` Trainer File: Chunk 1

```
1: ## ! Configuration: Begin

2: ## Output model
3: model_name = 'arch_opt4_mnist_dataset_imdb'
4: model_path = './models/'+model_name

5: ## Model type. See Kim Yoon's Convolutional Neural Networks for
   Sentence Classification, Section 3
6: model_type = "CNN-non-static" # CNN-rand or CNN-non-static or
   CNN-static

7: ## Data source
8: data_source = "local_dir" # "keras_data_set" or "local_dir"

9: padding_word = "<PAD/>"

10: batch_size = 64
11: num_epochs = 10

12: ## input image dimensions
13: img_rows, img_cols = 28, 28

14: embedding_dim = 50

15: ## Prepossessing parameters
16: sequence_length = 400
17: max_words = 5000

18: ## Word2Vec parameters (see train_word2vec)
19: min_word_count = 1
20: context = 10

21: ## ! Configuration: End
```

Algorithm 19 The `arch_opt4_mnist_dataset_imdb.py` Trainer File: Chunk 2

```
1: def load_data(data_source):
2:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
3:    if data_source == "keras_data_set":
4:       print('Using the keras IMDb dataset')
5:       (x_train, y_train), (x_test, y_test) = \
6:       imdb.load_data\
7:       (
8:          num_words=max_words,
9:          start_char=None,
10:         oov_char=None,
11:         index_from=None
12:      )

13:      x_train = sequence.pad_sequences(x_train,
   maxlen=sequence_length, padding="post", truncating="post")
14:      x_test = sequence.pad_sequences(x_test,
   maxlen=sequence_length, padding="post", truncating="post")

15:      vocabulary = imdb.get_word_index()
16:      vocabulary_inv = dict((v, k) for k, v in vocabulary.items())
17:      vocabulary_inv[0] = deepcopy(padding_word)
18:   else:
19:      print('Using a local IMDb dataset.')
20:      (x_train, y_train), (x_test, y_test), vocabulary_inv =
   data_helpers.load_local_imdb_dataset(sequence_length)

21:   return x_train, y_train, x_test, y_test, vocabulary_inv


22: # Data Preparation
23: print("Load data...")
24: x_train, y_train, x_test, y_test, vocabulary_inv =
   load_data(data_source)

25: input_shape = x_train.shape[1:]
```

Algorithm 20 The `arch_opt4_mnist_dataset_imdb.py` Trainer File: Chunk 3

```
 1: print('Input shape:  ' + str(input_shape))

 2: if sequence_length != x_test.shape[1]:
 3:    print("Adjusting sequence length for actual size")
 4:    sequence_length = x_test.shape[1]

 5: print("x_train shape:  " + str(x_train.shape))
 6: print("x_test shape:  " + str(x_test.shape))
 7: print("Vocabulary Size:  {:d}".format(len(vocabulary_inv)))

 8: # Prepare embedding layer weights and convert inputs for static
    model
 9: print("Model type is:  " + str(model_type))
10: if model_type in ["CNN-non-static", "CNN-static"]:
11:    embedding_weights = \
12:    train_word2vec\
13:    (
14:       np.vstack((x_train, x_test)),
15:       vocabulary_inv,
16:       num_features=embedding_dim,
17:       min_word_count=min_word_count,
18:       context=context
19:    )
20:    if model_type == "CNN-static":
21:       x_train = np.stack([np.stack([embedding_weights[word] for
    word in sentence]) for sentence in x_train])
22:       x_test = np.stack([np.stack([embedding_weights[word] for
    word in sentence]) for sentence in x_test])
23:       print("x_train static shape:  " + str(x_train.shape))
24:       print("x_test static shape:  " + str(x_test.shape))

25: elif model_type == "CNN-rand":
26:    embedding_weights = None
27: else:
28:    raise ValueError("Unknown model type")
```

Algorithm 21 The `arch_opt4_mnist_dataset_imdb.py` Trainer File: Chunk 4

```
1: ## !  Build model:  Begin

2: model = Sequential()

3: input_shape = (sequence_length,)
4: model.add(Embedding(len(vocabulary_inv), embedding_dim,
   input_length=sequence_length, name="embedding_",
   input_shape=input_shape))
5: model.add(Convolution1D(32, kernel_size=3, activation='relu'))

6: model.add(Convolution1D(64, 3, activation='relu'))
7: model.add(MaxPooling1D(pool_size=2))
8: model.add(Dropout(0.25))
9: model.add(Flatten())
10: model.add(Dense(128, activation='relu'))
11: model.add(Dropout(0.5))
12: model.add(Dense(1, activation='sigmoid'))

13: # Initialize weights with word2vec
14: if model_type == "CNN-non-static":
15:    weights = np.array([v for v in embedding_weights.values()])
16:    print("Initializing embedding layer with word2vec weights,
   shape" + str(weights.shape))
17:    embedding_layer = model.get_layer(name="embedding_")
18:    embedding_layer.set_weights([weights])

19: ## !  Build model:  End

20: model.compile\
21: (
22:    loss=binary_crossentropy,
23:    optimizer=Adadelta(),
24:    metrics=["accuracy"]
25: )
```

The Executable `arch_opt4_imdb_dataset_imdb.py`

This executable file contains the code through which the CNN with architecture `arch_opt4_imdb_dataset_imdb` can be trained on the IMDb dataset. The code is presented in Algorithms 22, 23, 24, 25 and 15. Table 2 shows the architecture and Figure 11 gives an visual intuition about its outputs layerwise. The source of this code is [40]. The training process and CNN architecture are tuned for the MNIST dataset.

The Executable `arch_opt4_imdb_dataset_mnist.py`

This executable file contains the code through which the CNN with architecture `arch_opt4_imdb_dataset_mnist` can be trained on the MNIST dataset. The code is presented in Algorithms 26, 27 and 15. Table 4 shows the architecture and Figure 14 gives an visual intuition about its outputs layerwise.

This code is a modified version of `arch_opt4_imdb_dataset_imdb.py`. The modifications allow for the training on the MNIST dataset instead. We use the `data_helpers.load_local_mnist_dataset` function in line 19 in Algorithm 16, which parses and preprocesses the IMDb dataset. The preprocessing method used is the same ones found in [39]. The lines 9 until 27 in Algorithm 24 have been removed as the embedding weights have no meaning for the training on the MNIST dataset. The feature extractor component of the architecture specified by lines 7 until 15 in Algorithm 25 are modified to the lines 13 until 21 in Algorithm 27. This change converted all the 1D layers to their 2D equivalents, on the feature extractor component of the architecture. The cost function has been modified from the binary cross entropy in line 31 in Algorithm 25 to the categorical cross entropy in line 32 and the output's layer activation function has been modified from the sigmoid in line 20 in Algorithm 25 to the softmax function in line 27 in Algorithm 27 with 10 output neurons.

Algorithm 22 The `arch_opt4_imdb_dataset_imdb.py` Trainer File: Chunk 1

```
1: ## ! Configuration: Begin

2: ## Output model
3: model_name = 'arch_opt4_imdb_dataset_imdb'
4: model_path = './models/'+model_name

5: ## Model type. See Kim Yoon's Convolutional Neural Networks for
   Sentence Classification, Section 3
6: model_type = "CNN-non-static" # CNN-rand or CNN-non-static or
   CNN-static

7: ## Data source
8: data_source = "local_dir" # keras_data_set or local_dir

9: padding_word = "<PAD/>"

10: ## Model Hyperparameters
11: embedding_dim = 50
12: filter_sizes = (3, 8)
13: num_filters = 10
14: dropout_prob = (0.5, 0.8)
15: hidden_dims = 50

16: ## Training parameters
17: batch_size = 64
18: num_epochs = 10

19: ## Prepossessing parameters
20: sequence_length = 400
21: max_words = 5000

22: ## Word2Vec parameters (see train_word2vec)
23: min_word_count = 1
24: context = 10

25: ## ! Configuration: End
```

Algorithm 23 The `arch_opt4_imdb_dataset_imdb.py` Trainer File: Chunk 2

```
1: def load_data(data_source):
2:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
3:    if data_source == "keras_data_set":
4:       print('Using the keras IMDb dataset')
5:       (x_train, y_train), (x_test, y_test) = \
6:       imdb.load_data\
7:       (
8:          num_words=max_words,
9:          start_char=None,
10:         oov_char=None,
11:         index_from=None
12:      )

13:      x_train = sequence.pad_sequences(x_train,
   maxlen=sequence_length, padding="post", truncating="post")
14:      x_test = sequence.pad_sequences(x_test,
   maxlen=sequence_length, padding="post", truncating="post")

15:      vocabulary = imdb.get_word_index()
16:      vocabulary_inv = dict((v, k) for k, v in vocabulary.items())
17:      vocabulary_inv[0] = deepcopy(padding_word)
18:   else:
19:      print('Using a local IMDb dataset.')
20:      (x_train, y_train), (x_test, y_test), vocabulary_inv =
   data_helpers.load_local_imdb_dataset(sequence_length)

21:   return x_train, y_train, x_test, y_test, vocabulary_inv


22: # Data Preparation
23: print("Load data...")
24: x_train, y_train, x_test, y_test, vocabulary_inv =
   load_data(data_source)
```

Algorithm 24 The `arch_opt4_imdb_dataset_imdb.py` Trainer File: Chunk 3

```
1: if sequence_length != x_test.shape[1]:
2:    print("Adjusting sequence length for actual size")
3:    sequence_length = x_test.shape[1]

4: print("x_train shape:  " + str(x_train.shape))
5: print("x_test shape:  " + str(x_test.shape))
6: print("Vocabulary Size:  {:d}".format(len(vocabulary_inv)))

7: # Prepare embedding layer weights and convert inputs for static
   model
8: print("Model type is:  " + str(model_type))
9: if model_type in ["CNN-non-static", "CNN-static"]:
10:    embedding_weights = \
11:    train_word2vec\
12:    (
13:       np.vstack((x_train, x_test)),
14:       vocabulary_inv,
15:       num_features=embedding_dim,
16:       min_word_count=min_word_count,
17:       context=context
18:    )
19:    if model_type == "CNN-static":
20:       x_train = np.stack([np.stack([embedding_weights[word] for
   word in sentence]) for sentence in x_train])
21:       x_test = np.stack([np.stack([embedding_weights[word] for
   word in sentence]) for sentence in x_test])
22:       print("x_train static shape:  " + str(x_train.shape))
23:       print("x_test static shape:  " + str(x_test.shape))

24: elif model_type == "CNN-rand":
25:    embedding_weights = None
26: else:
27:    raise ValueError("Unknown model type")

28: ## !  Build model:  Begin

29: input_shape = (sequence_length,)
30: model_input = Input(shape=input_shape)
```

Algorithm 25 The `arch_opt4_imdb_dataset_imdb.py` Trainer File: Chunk 4

```
1: z = Embedding(len(vocabulary_inv), embedding_dim,
   input_length=sequence_length, name="embedding")(model_input)
2: z = Dropout(dropout_prob[0])(z)

3: # Convolutional block
4: conv_blocks = []
5: for sz in filter_sizes:
6:    conv = \
7:    Convolution1D\
8:    (
9:       filters=num_filters,
10:      kernel_size=sz,
11:      padding="valid",
12:      activation="relu",
13:      strides=1
14:   )(z)
15:   conv = MaxPooling1D(pool_size=2)(conv)
16:   conv = Flatten()(conv)
17:   conv_blocks.append(conv)
18: z = Concatenate()(conv_blocks) if len(conv_blocks) > 1 else
   conv_blocks[0]

19: z = Dropout(dropout_prob[1])(z)
20: z = Dense(hidden_dims, activation="relu")(z) model_output = Dense(1,
   activation="sigmoid")(z)

21: model = Model(model_input, model_output)

22: # Initialize weights with word2vec
23: if model_type == "CNN-non-static":
24:    weights = np.array([v for v in embedding_weights.values()])
25:    print("Initializing embedding layer with word2vec weights,
   shape:  " + str(weights.shape))
26:    embedding_layer = model.get_layer("embedding")
27:    embedding_layer.set_weights([weights])

28: ## !  Build model:  End

29: model.compile\
30: (
31:    loss=binary_crossentropy,
32:    optimizer=Adadelta(),
33:    metrics=["accuracy"]
34: )
```

Algorithm 26 The `arch_opt4_imdb_dataset_mnist.py` Trainer File: Chunk 1

```
1: ## ! Configuration: Begin

2: ## Output model
3: model_name = 'arch_opt4_imdb_dataset_mnist'
4: model_path = './models/'+model_name

5: num_classes = 10

6: ## input image dimensions
7: img_rows, img_cols = 28, 28

8: ## Model Hyperparameters
9: filter_sizes = (3, 8)
10: num_filters = 10
11: dropout_prob = (0.5, 0.8)
12: hidden_dims = 50

13: ## Training parameters
14: batch_size = 64
15: num_epochs = 10

16: ## Data source
17: data_source = "local_dir" # "keras_data_set" or "local_dir"

18: ## ! Configuration: End


19: def load_data(data_source):
20:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
21:    if data_source == "keras_data_set":
22:       (x_train, y_train), (x_test, y_test) = mnist.load_data()
23:    else:
24:       (x_train, y_train), (x_test, y_test) = \
25:       data_helpers.load_local_mnist_dataset\
26:       (
27:          num_classes=num_classes,
28:          img_rows=img_rows,
29:          img_cols=img_cols
30:       )

31:    return x_train, y_train, x_test, y_test
```

Algorithm 27 The `arch_opt4_imdb_dataset_mnist.py` Trainer File: Chunk 2

```
1: # Data Preparation
2: print("Load data...")
3: x_train, y_train, x_test, y_test = load_data(data_source)

4: ## ! Build model: Begin

5: input_shape = x_train.shape[1:]

6: print('Input shape: ' + str(input_shape))

7: model_input = Input(shape=input_shape)
8: z = Dropout(dropout_prob[0])(model_input)

9: # Convolutional block
10: conv_blocks = []
11: for sz in filter_sizes:
12:    conv = \
13:    Conv2D\
14:    (
15:       filters=num_filters,
16:       kernel_size=sz,
17:       padding="valid",
18:       activation="relu",
19:       strides=1
20:    )(z)
21:    conv = MaxPooling2D(pool_size=2)(conv)
22:    conv = Flatten()(conv)
23:    conv_blocks.append(conv)
24: z = Concatenate()(conv_blocks) if len(conv_blocks) > 1 else
    conv_blocks[0]

25: z = Dropout(dropout_prob[1])(z)
26: z = Dense(hidden_dims, activation="relu")(z)
27: model_output = Dense(num_classes, activation='softmax')(z)

28: model = Model(model_input, model_output)

29: ## ! Build model: End

30: model.compile\
31: (
32:    loss=categorical_crossentropy,
33:    optimizer=Adadelta(),
34:    metrics=["accuracy"]
35: )
```

Appendix IV

Here we present the files containing each executable code that produces each of the experiments described in the Experiments Part II. The chunk of code that imports some of the necessary libraries for the these files can be found in Algorithm 28. In order for each of these codes to be reproducible, the code chunk in Algorithm 14 is read by the Python interpreter beforehand, prior to any other line of code presented in this thesis. All of the code presented here applies for Python 3.

Algorithm 28 Libraries used by the executable files of Experiments Part II

```
1: from copy import deepcopy
2: import os
3: import random
4: import numpy as np
5: import pandas as pd
6: import tensorflow as tf
7: from keras.layers import Dropout, Flatten, Convolution1D,
   MaxPooling1D, Conv2D, MaxPooling2D, Concatenate
8: from keras.datasets import mnist, imdb
9: from keras.preprocessing import sequence
10: from keras.losses import binary_crossentropy,
    categorical_crossentropy
11: from w2v import train_word2vec
12: import data_helpers
```

The Executables `mnistsrc_train.py`, `imdbsrc_train.py`

The executable files `mnistsrc_train.py` and `imdbsrc_train.py` train the same CNNs as `arch_opt4_mnist_dataset_mnist.py` (`mnistsrc` source architecture) on the MNIST dataset and `arch_opt4_imdb_dataset_imdb.py` (`imdbsrc` source architecture) on the IMDb dataset respectively. Each of these files train the source models which will in turn will continue being trained by the `mnistsrc_mnisttgt_train.py` (`mnistsrc_mnisttgt`), `mnistsrc_imdbtgt_train.py` (`mnistsrc_imdbtgt`), `imdbsrc_imdbtgt_train.py` (`imdbsrc_imdbtgt`) and `imdbsrc_mnisttgt_train.py` (`imdbsrc_mnisttgt`) for the fine tuning part.

Algorithm 29 The `mnistsrc_train.py` Trainer File: Chunk 1

```
1: from keras.models import Sequential
2: from keras.layers import Dense
3: from keras.optimizers import Adadelta

4: ## ! Configuration: Begin

5: batch_size = 64
6: num_classes = 10
7: num_epochs = 10

8: ## input image dimensions
9: img_rows, img_cols = 28, 28

10: ## Data source
11: data_source = "local_dir" # "keras_data_set" or "local_dir"

12: ## Output model's name
13: src_model_name = "mnistsrc"

14: ## Output model's file path
15: src_model_path = "models/"+src_model_name+".mdl"

16: ## Output model's evaluation history file path
17: src_model_hist = "models/"+src_model_name+".xlsx"

18: ## ! Configuration: End

19: def load_data(data_source):
20:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
21:    if data_source == "keras_data_set":
22:       (x_train, y_train), (x_test, y_test) = mnist.load_data()
23:    else:
24:       (x_train, y_train), (x_test, y_test) = \
25:       data_helpers.load_local_mnist_dataset\
26:       (
27:          num_classes=num_classes,
28:          img_rows=img_rows,
29:          img_cols=img_cols
30:       )

31:    return x_train, y_train, x_test, y_test
```

Algorithm 30 The `mnistsrc_train.py` Trainer File: Chunk 2

```
1: ## Data Preparation
2: print("Load data...")
3: x_train, y_train, x_test, y_test = load_data(data_source)

4: input_shape = x_train.shape[1:]

5: print('Input shape:  ' + str(input_shape))

6: src_model = Sequential()
7: src_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
   input_shape=input_shape))
8: src_model.add(Conv2D(64, (3, 3), activation='relu'))
9: src_model.add(MaxPooling2D(pool_size=(2, 2)))
10: src_model.add(Dropout(0.25))
11: src_model.add(Flatten())
12: src_model.add(Dense(128, activation='relu'))
13: src_model.add(Dropout(0.5))
14: src_model.add(Dense(num_classes, activation='softmax'))

15: src_model.compile\
16: (
17:    loss=categorical_crossentropy,
18:    optimizer=Adadelta(),
19:    metrics=["accuracy"]
20: )

21: src_model.summary()

22: ## Train the model
23: src_model.fit\
24: (
25:    x_train,
26:    y_train,
27:    batch_size=batch_size,
28:    epochs=num_epochs,
29:    validation_data=(x_test, y_test),
30:    verbose=1
31: )
```

Algorithm 31 The `mnistsrc_train.py` Trainer File: Chunk 3

```
1: src_model.save(src_model_path)

2: score = src_model.evaluate(x_test, y_test, verbose=0)

3: print('Test loss:  %.30f'%score[0])
4: print('Test accuracy:  %.30f'%score[1])

5: ## Store training output
6: metrics_history = pd.DataFrame.from_dict(src_model.history.history)

7: metrics_history = pd.concat([metrics_history['loss'],
   metrics_history['accuracy'], metrics_history['val_loss'],
   metrics_history['val_accuracy']], axis=1)
8: metrics_history.T.to_excel(src_model_hist)
```

Algorithm 32 The `imdbsrc_train.py` Trainer File: Chunk 1

```
1: from keras.models import Model
2: from keras.layers import Input, Dense, Embedding
3: from keras.optimizers import Adadelta

4: ## ! Configuration: Begin

5: ## Model type. See Kim Yoon's Convolutional Neural Networks for
   Sentence Classification, Section 3
6: model_type = "CNN-non-static" # CNN-rand or CNN-non-static or
   CNN-static

7: ## Data source
8: data_source = "local_dir" # keras_data_set or local_dir

9: padding_word = "<PAD/>"

10: ## Model Hyperparameters
11: embedding_dim = 28 # 50
12: filter_sizes = (3, 8)
13: num_filters = 10
14: dropout_prob = (0.5, 0.8)
15: hidden_dims = 50

16: ## Training parameters
17: batch_size = 64
18: num_epochs = 10

19: ## Prepossessing parameters
20: sequence_length = 28 # 400
21: max_words = 5000

22: ## Word2Vec parameters (see train_word2vec)
23: min_word_count = 1
24: context = 10

25: ## Stage 1 model name
26: src_model_name = "imdbsrc"

27: ## Stage 1 model file path
28: src_model_path = "models/"+src_model_name+".mdl"

29: ## Stage 1 model evaluation history file path
30: src_model_hist = "models/"+src_model_name+".xlsx"
```

Algorithm 33 The `imdbsrc_train.py` Trainer File: Chunk 2

```
1: ## !  Configuration:  End

2: def load_data(data_source):
3:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
4:    if data_source == "keras_data_set":
5:       print('Using the keras IMDb dataset')
6:       (x_train, y_train), (x_test, y_test) = \
7:       imdb.load_data\
8:       (
9:          num_words=max_words,
10:         start_char=None,
11:         oov_char=None,
12:         index_from=None
13:      )

14:      x_train = sequence.pad_sequences(x_train,
   maxlen=sequence_length, padding="post", truncating="post")
15:      x_test = sequence.pad_sequences(x_test,
   maxlen=sequence_length, padding="post", truncating="post")

16:      vocabulary = imdb.get_word_index()
17:      vocabulary_inv = dict((v, k) for k, v in vocabulary.items())
18:      vocabulary_inv[0] = deepcopy(padding_word)
19:   else:
20:      print('Using a local IMDb dataset.')
21:      (x_train, y_train), (x_test, y_test), vocabulary_inv =
   data_helpers.load_local_imdb_dataset(sequence_length)

22:   return x_train, y_train, x_test, y_test, vocabulary_inv


23: ## Data Preparation
24: print("Load data...")
25: x_train, y_train, x_test, y_test, vocabulary_inv =
   load_data(data_source)
```

Algorithm 34 The `imdbsrc_train.py` Trainer File: Chunk 3

```
 1: if sequence_length != x_test.shape[1]:
 2:    print("Adjusting sequence length for actual size")
 3:    sequence_length = x_test.shape[1]

 4: print("x_train shape:  " + str(x_train.shape))
 5: print("x_test shape:  " + str(x_test.shape))
 6: print("Vocabulary Size:  {:d}".format(len(vocabulary_inv)))

 7: ## Prepare embedding layer weights and convert inputs for static
    model
 8: print("Model type is:  " + str(model_type))
 9: if model_type in ["CNN-non-static", "CNN-static"]:
10:    embedding_weights = \
11:    train_word2vec\
12:    (
13:       np.vstack((x_train, x_test)),
14:       vocabulary_inv,
15:       num_features=embedding_dim,
16:       min_word_count=min_word_count,
17:       context=context
18:    )
19:    if model_type == "CNN-static":
20:       x_train = np.stack([np.stack([embedding_weights[word] for
    word in sentence]) for sentence in x_train])
21:       x_test = np.stack([np.stack([embedding_weights[word] for
    word in sentence]) for sentence in x_test])
22:       print("x_train static shape:  " + str(x_train.shape))
23:       print("x_test static shape:  " + str(x_test.shape))

24: elif model_type == "CNN-rand":
25:    embedding_weights = None
26: else:
27:    raise ValueError("Unknown model type")
```

Algorithm 35 The `imdbsrc_train.py` Trainer File: Chunk 4

```
1: # Build model
2: if model_type == "CNN-static":
3:     input_shape = (sequence_length, embedding_dim)
4: else:
5:     input_shape = (sequence_length,)

6: a0 = Input(shape=input_shape)

7: ## Static model does not have embedding layer
8: if model_type == "CNN-static":
9:     a = a0
10: else:
11:     a = Embedding(len(vocabulary_inv), embedding_dim,
   input_length=sequence_length, name="embedding")(a0)

12: a = Dropout(dropout_prob[0])(a)

13: ## Convolutional block
14: conv_blocks = []
15: for sa in filter_sizes:
16:     conv = \
17:     Convolution1D\
18:     (
19:         filters=num_filters,
20:         kernel_size=sa,
21:         padding="valid",
22:         activation="relu",
23:         strides=1
24:     )(a)
25:     conv = MaxPooling1D(pool_size=2)(conv)
26:     conv = Flatten()(conv)
27:     conv_blocks.append(conv)
28: a = Concatenate()(conv_blocks) if len(conv_blocks) > 1 else
   conv_blocks[0]
```

Algorithm 36 The `imdbsrc_train.py` Trainer File: Chunk 5

```
1: a = Dropout(dropout_prob[1])(a)
2: a = Dense(hidden_dims, activation="relu")(a)
3: a = Dense(1, activation="sigmoid")(a)

4: src_model = Model(a0, a)
5: src_model.compile\
6: (
7:    loss=binary_crossentropy,
8:    optimizer=Adadelta(),
9:    metrics=["accuracy"]
10: )

11: ## Initialize weights with word2vec
12: if model_type == "CNN-non-static":
13:    weights = np.array([v for v in embedding_weights.values()])
14:    print("Initializing embedding layer with word2vec weights,
   shape:  " + str(weights.shape))
15:    embedding_layer = src_model.get_layer("embedding")
16:    embedding_layer.set_weights([weights])

17: src_model.summary()

18: ## Train the model
19: src_model.fit\
20: (
21:    x_train,
22:    y_train,
23:    batch_size=batch_size,
24:    epochs=num_epochs,
25:    validation_data=(x_test, y_test),
26:    verbose=1,
27: )

28: src_model.save(src_model_path)
29: score = src_model.evaluate(x_test, y_test, verbose=0)
30: print('Test loss:  %.30f'%score[0])
31: print('Test accuracy:  %.30f'%score[1])
32: ## Store training output
33: metrics_history = pd.DataFrame.from_dict(src_model.history.history)

34: metrics_history = pd.concat([metrics_history['loss'],
   metrics_history['accuracy'], metrics_history['val_loss'],
   metrics_history['val_accuracy']], axis=1)
35: metrics_history.T.to_excel(src_model_hist)
```

The Executable `mnistsrc_mnisttgt_train.py`

The `mnistsrc_mnisttgt_train.py` file fine tunes a model trained on the MNIST dataset with architecture `mnistsrc`. The fine tuning of the model is based on the MNIST dataset. Line 13 in Algorithm 37 loads the source model trained by `mnistsrc_train.py` (Algorithms 29, 30 and 31), and `mnistsrc_mnisttgt_train.py` (Algorithms 37, 38 and 39) produces the target model `mnistsrc_mnisttgt`.

The Executable `mnistsrc_imdbtgt_train.py`

The `mnistsrc_imdbtgt_train.py` file fine tunes a model trained on the MNIST dataset with architecture `mnistsrc`. The fine tuning of the model is based on the IMDb dataset. Line 15 in Algorithm 40 loads the source model trained by `mnistsrc_train.py` (Algorithms 29, 30 and 31), and `mnistsrc_imdbtgt_train.py` (Algorithms 40, 41, 42, 43 and 44) and produces the target model corresponding to the architecture `mnistsrc_imdbtgt`. However before the training, `mnistsrc_imdbtgt_train.py` prepares the embedding layer in lines 11 until 29 in Algorithm 42. In line 4 in Algorithm 43 the embedding layer is added prior to the rest of the layers. Line 7 in Algorithm 43 is responsible for the transference of each of the hidden layers of the source model to the target architecture and initial target model. The output's layer activation function is defined to be the sigmoid and the number of output neurons is 1, this is done in line 10 in Algorithm 43. The cost function is defined to be the binary cross entropy in line 13 in Algorithm 43.

Algorithm 37 The `mnistsrc_mnisttgt_train.py` Trainer File: Chunk 1

```
1: ## ! Configuration: Begin

2: ## Stage 1 model name
3: src_model_name = "mnistsrc"

4: ## Stage 1 model file path
5: src_model_path = "models/"+src_model_name+".mdl"

6: ## Stage 2 model name
7: tgt_model_name = src_model_name+"_"+"mnisttgt"

8: ## Stage 2 model file path
9: tgt_model_path = "models/"+tgt_model_name+".mdl"

10: ## Stage 2 model evaluation history file path
11: tgt_model_hist = "models/"+tgt_model_name+".xlsx"

12: ## Pulling the trained Stage 1 model
13: tgt_model = tf.keras.models.load_model(src_model_path)

14: batch_size = 64
15: num_classes = 10
16: num_epochs = 10

17: ## input image dimensions
18: img_rows, img_cols = 28, 28

19: ## Data source
20: data_source = "local_dir" # "keras_data_set" or "local_dir"

21: ## ! Configuration: End
```

Algorithm 38 The `mnistsrc_mnisttgt_train.py` Trainer File: Chunk 2

```
1: def load_data(data_source):
2:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
3:    if data_source == "keras_data_set":
4:       (x_train, y_train), (x_test, y_test) = mnist.load_data()
5:    else:
6:       (x_train, y_train), (x_test, y_test) = \
7:       data_helpers.load_local_mnist_dataset\
8:       (
9:          num_classes=num_classes,
10:         img_rows=img_rows,
11:         img_cols=img_cols
12:      )

13:   return x_train, y_train, x_test, y_test

14: ## Data Preparation
15: print("Load data...")
16: x_train, y_train, x_test, y_test = load_data(data_source)

17: input_shape = x_train.shape[1:]

18: print('Input shape:  '  + str(input_shape))

19: tgt_model.summary()

20: ## Train the model
21: tgt_model.fit\
22: (
23:    x_train,
24:    y_train,
25:    batch_size=batch_size,
26:    epochs=num_epochs,
27:    validation_data=(x_test, y_test),
28:    verbose=1
29: )
```

Algorithm 39 The `mnistsrc_mnisttgt_train.py` Trainer File: Chunk 3

```
1: tgt_model.save(tgt_model_path, save_format='h5')

2: score = tgt_model.evaluate(x_test, y_test, verbose=0)

3: print('Test loss:  %.30f'%score[0])
4: print('Test accuracy:  %.30f'%score[1])

5: ## Store training output
6: metrics_history = pd.DataFrame.from_dict(tgt_model.history.history)

7: metrics_history = pd.concat([metrics_history['loss'],
   metrics_history['accuracy'], metrics_history['val_loss'],
   metrics_history['val_accuracy']], axis=1)
8: metrics_history.T.to_excel(tgt_model_hist)
```

Algorithm 40 The `mnistsrc_imdbtgt_train.py` Trainer File: Chunk 1

```
1: from tensorflow.keras import Model, Sequential
2: from tensorflow.keras.layers import Input, Dense, Embedding,
   Reshape
3: from tensorflow.keras.optimizers import Adadelta


4: ## ! Configuration: Begin

5: ## Stage 1 model name
6: src_model_name = "mnistsrc"
7: ## Stage 1 model file path
8: src_model_path = "models/"+src_model_name+".mdl"

9: ## Stage 2 model name
10: tgt_model_name = src_model_name+"_"+"imdbtgt"
11: ## Stage 2 model file path
12: tgt_model_path = "models/"+tgt_model_name+".mdl"
13: ## Stage 2 model evaluation history file path tgt_model_hist = "mod-
    els/"+tgt_model_name+".xlsx"

14: ## Pulling the trained Stage 1 model
15: src_model = tf.keras.models.load_model(src_model_path)

16: ## The Stage 1 input layer's hyperparameters
17: src_img_rows, src_img_cols = tuple(src_model.layers[0].input
    .shape[1:-1])

18: ## Model type. See Kim Yoon's Convolutional Neural Networks for
    Sentence Classification, Section 3
19: model_type = "CNN-non-static" # CNN-rand or CNN-non-static or
    CNN-static

20: ## Data source
21: data_source = "local_dir" # keras_data_set or local_dir

22: padding_word = "<PAD/>"

23: ## Model Hyperparameters
24: embedding_dim = src_img_cols # 50
25: filter_sizes = (3, 8)
26: num_filters = 10
27: dropout_prob = (0.5, 0.8)
28: hidden_dims = 50
```

Algorithm 41 The `mnistsrc_imdbtgt_train.py` Trainer File: Chunk 2

```
 1: ## Training parameters
 2: batch_size = 64
 3: num_epochs = 10

 4: ## Prepossessing parameters
 5: sequence_length = src_img_rows # 400
 6: max_words = 5000

 7: ## Word2Vec parameters (see train_word2vec)
 8: min_word_count = 1
 9: context = 10

10: ## ! Configuration: End

11: def load_data(data_source):
12:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
13:    if data_source == "keras_data_set":
14:       print('Using the keras IMDb dataset')
15:       (x_train, y_train), (x_test, y_test) = \
16:       imdb.load_data\
17:       (
18:          num_words=max_words,
19:          start_char=None,
20:          oov_char=None,
21:          index_from=None
22:       )

23:       x_train = sequence.pad_sequences(x_train,
   maxlen=sequence_length, padding="post", truncating="post")
24:       x_test = sequence.pad_sequences(x_test,
   maxlen=sequence_length, padding="post", truncating="post")

25:       vocabulary = imdb.get_word_index()
26:       vocabulary_inv = dict((v, k) for k, v in vocabulary.items())
27:       vocabulary_inv[0] = deepcopy(padding_word)
28:    else:
29:       print('Using a local IMDb dataset.')
30:       (x_train, y_train), (x_test, y_test), vocabulary_inv =
   data_helpers.load_local_imdb_dataset(sequence_length)

31:    return x_train, y_train, x_test, y_test, vocabulary_inv
```

Algorithm 42 The `mnistsrc_imdbtgt_train.py` Trainer File: Chunk 3

```
1: ## Data Preparation print("Load data...")
2: x_train, y_train, x_test, y_test, vocabulary_inv =
   load_data(data_source)

3: if sequence_length != x_test.shape[1]:
4:     print("Adjusting sequence length for actual size")
5:     sequence_length = x_test.shape[1]

6: print("x_train shape:  " + str(x_train.shape))
7: print("x_test shape:  " + str(x_test.shape))
8: print("Vocabulary Size:  {:d}".format(len(vocabulary_inv)))

9: ## Prepare embedding layer weights and convert inputs for static
   model
10: print("Model type is:  " + str(model_type))
11: if model_type in ["CNN-non-static", "CNN-static"]:
12:     embedding_weights = \
13:     train_word2vec\
14:     (
15:         np.vstack((x_train, x_test)),
16:         vocabulary_inv,
17:         num_features=embedding_dim,
18:         min_word_count=min_word_count,
19:         context=context
20:     )
21:     if model_type == "CNN-static":
22:         x_train = np.stack([np.stack([embedding_weights[word] for
   word in sentence]) for sentence in x_train])
23:         x_test = np.stack([np.stack([embedding_weights[word] for
   word in sentence]) for sentence in x_test])
24:         print("x_train static shape:  " + str(x_train.shape))
25:         print("x_test static shape:  " + str(x_test.shape))

26: elif model_type == "CNN-rand":
27:     embedding_weights = None
28: else:
29:     raise ValueError("Unknown model type")
```

Algorithm 43 The `mnistsrc_imdbtgt_train.py` Trainer File: Chunk 4

```
1: tgt_model = Sequential()

2: if model_type != "CNN-static":
3:    input_shape = (sequence_length,)
4:    tgt_model.add(Embedding(len(vocabulary_inv),
   embedding_dim, input_length=sequence_length, name="embedding",
   input_shape=input_shape))
5:    tgt_model.add(Reshape(target_shape=tgt_model.get_layer(index=0)
   .output.shape[1:]+1))
6:    for src_model_idx in range(len(src_model.layers)-1):
7:       tgt_model.add(src_model.get_layer(index=src_model_idx))
8: else:
9:    exit("Exception:  The current version does not support static
   CNN trainings.")

10: tgt_model.add(Dense(1, activation="sigmoid"))

11: tgt_model.compile\
12: (
13:    loss=binary_crossentropy,
14:    optimizer=Adadelta(),
15:    metrics=["accuracy"]
16: )

17: tgt_model.summary()

18: ## Train the model
19: tgt_model.fit\
20: (
21:    x_train,
22:    y_train,
23:    batch_size=batch_size,
24:    epochs=num_epochs,
25:    validation_data=(x_test, y_test),
26:    verbose=1,
27: )
```

Algorithm 44 The `mnistsrc_imdbtgt_train.py` Trainer File: Chunk 5

```
1: tgt_model.save(tgt_model_path, save_format='h5')

2: score = tgt_model.evaluate(x_test, y_test, verbose=0)

3: print('Test loss:  %.30f'%score[0])
4: print('Test accuracy:  %.30f'%score[1])

5: ## Store training output
6: metrics_history = pd.DataFrame.from_dict(tgt_model.history.history)

7: metrics_history = pd.concat([metrics_history['loss'],
   metrics_history['accuracy'], metrics_history['val_loss'],
   metrics_history['val_accuracy']], axis=1)
8: metrics_history.T.to_excel(tgt_model_hist)
```

The Executable `imdbsrc_imdbtgt_train.py`

The `imdbsrc_imdbtgt_train.py` file fine tunes a model trained on the IMDb dataset with architecture `mnistsrc`. The fine tuning of the model is based on the IMDb dataset. Line 13 in Algorithm 45 loads the source model trained by `imdbsrc_train.py` (Algorithms 32, 33, 34, 35 and 36), and `imdbsrc_imdbtgt_train.py` (Algorithms 45, 46 and 47) produces the target model `imdbsrc_imdbtgt`.

The Executable `imdbsrc_mnisttgt_train.py`

The `imdbsrc_mnisttgt_train.py` file fine tunes a model trained on the IMDb dataset with architecture `mnistsrc`. The fine tuning of the model is based on the MNIST dataset. Line 16 in Algorithm 48 loads the source model trained by `imdbsrc_train.py` (Algorithms 32, 33, 34, 35 and 36), and `imdbsrc_mnisttgt_train.py` (Algorithms 48, 49 and 50) and produces the target model corresponding to the architecture `imdbsrc_mnisttgt`. The embedding layer will not be included in this architecture. Lines 20 until 5 in Algorithm 49 is responsible for the transference of each of the hidden layers of the source model to the target architecture and initial target model. The output's layer activation function is defined to be the softmax and the number of output neurons is 10, this is done in line 6 in Algorithm 50. The cost function is defined to be the categorical cross entropy in line 10 in Algorithm 50.

Algorithm 45 The `imdbsrc_imdbtgt_train.py` Trainer File: Chunk 1

```
1: ## !  Configuration:  Begin

2: ## Stage 1 model name
3: src_model_name = "imdbsrc"

4: ## Stage 1 model file path
5: src_model_path = "models/"+src_model_name+".mdl"

6: ## Stage 2 model name
7: tgt_model_name = src_model_name+"_"+"imdbtgt"

8: ## Stage 2 model file path
9: tgt_model_path = "models/"+tgt_model_name+".mdl"

10: ## Stage 2 model evaluation history file path
11: tgt_model_hist = "models/"+tgt_model_name+".xlsx"

12: ## Pulling the trained Stage 1 model
13: src_model = tgt_model = tf.keras.models.load_model(src_model_path)

14: ## The Stage 1 input and embedding layers' hyperparameters
15: src_embedding_dim, src_sequence_length = tuple(src_model.layers[1]
    .output.shape[1:])

16: ## Model type.  See Kim Yoon's Convolutional Neural Networks for
    Sentence Classification, Section 3
17: model_type = "CNN-non-static" # CNN-rand or CNN-non-static or
    CNN-static

18: ## Data source
19: data_source = "local_dir" # keras_data_set or local_dir

20: padding_word = "<PAD/>"

21: ## Model Hyperparameters
22: embedding_dim = src_embedding_dim
23: filter_sizes = (3, 8)
24: num_filters = 10
25: dropout_prob = (0.5, 0.8)
26: hidden_dims = 50
```

Algorithm 46 The `imdbsrc_imdbtgt_train.py` Trainer File: Chunk 2

```
1: ## Training parameters
2: batch_size = 64
3: num_epochs = 10

4: ## Prepossessing parameters
5: sequence_length = src_sequence_length
6: max_words = 5000

7: ## Word2Vec parameters (see train_word2vec)
8: min_word_count = 1
9: context = 10

10: ## !  Configuration:  End

11: def load_data(data_source):
12:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
13:    if data_source == "keras_data_set":
14:       print('Using the keras IMDb dataset')
15:       (x_train, y_train), (x_test, y_test) = \
16:       imdb.load_data\
17:       (
18:          num_words=max_words,
19:          start_char=None,
20:          oov_char=None,
21:          index_from=None
22:       )

23:       x_train = sequence.pad_sequences(x_train,
   maxlen=sequence_length, padding="post", truncating="post")
24:       x_test = sequence.pad_sequences(x_test,
   maxlen=sequence_length, padding="post", truncating="post")

25:       vocabulary = imdb.get_word_index()
26:       vocabulary_inv = dict((v, k) for k, v in vocabulary.items())
27:       vocabulary_inv[0] = deepcopy(padding_word)
28:    else:
29:       print('Using a local IMDb dataset.')
30:       (x_train, y_train), (x_test, y_test), vocabulary_inv =
   data_helpers.load_local_imdb_dataset(sequence_length)

31:    return x_train, y_train, x_test, y_test, vocabulary_inv
```

Algorithm 47 The `imdbsrc_imdbtgt_train.py` Trainer File: Chunk 3

```
1: ## Data Preparation
2: print("Load data...")
3: x_train, y_train, x_test, y_test, vocabulary_inv =
   load_data(data_source)

4: if sequence_length != x_test.shape[1]:
5:    print("Adjusting sequence length for actual size")
6:    sequence_length = x_test.shape[1]
7: print("x_train shape:  " + str(x_train.shape))
8: print("x_test shape:  " + str(x_test.shape))
9: print("Vocabulary Size:  {:d}".format(len(vocabulary_inv)))

10: tgt_model.summary()

11: ## Train the model
12: tgt_model.fit\
13: (
14:    x_train,
15:    y_train,
16:    batch_size=batch_size,
17:    epochs=num_epochs,
18:    validation_data=(x_test, y_test),
19:    verbose=1,
20: )

21: tgt_model.save(tgt_model_path, save_format='h5')

22: score = tgt_model.evaluate(x_test, y_test, verbose=0)

23: print('Test loss:  %.30f'%score[0])
24: print('Test accuracy:  %.30f'%score[1])

25: ## Store training output
26: metrics_history = pd.DataFrame.from_dict(tgt_model.history.history)

27: metrics_history = pd.concat([metrics_history['loss'],
    metrics_history['accuracy'], metrics_history['val_loss'],
    metrics_history['val_accuracy']], axis=1)
28: metrics_history.T.to_excel(tgt_model_hist)
```

Algorithm 48 The `imdbsrc_mnisttgt_train.py` Trainer File: Chunk 1

```
1: from tensorflow.keras.models import Model
2: from tensorflow.keras.layers import Input, Dense
3: from tensorflow.keras.optimizers import Adadelta

4: ## ! Configuration: Begin

5: ## Stage 1 model name
6: src_model_name = "imdbsrc"

7: ## Stage 1 model file path
8: src_model_path = "models/"+src_model_name+".mdl"

9: ## Stage 2 model name
10: tgt_model_name = src_model_name+"_"+"mnisttgt"

11: ## Stage 2 model file path
12: tgt_model_path = "models/"+tgt_model_name+".mdl"

13: ## Stage 2 model evaluation history file path
14: tgt_model_hist = "models/"+tgt_model_name+".xlsx"

15: ## Pulling the trained Stage 1 model
16: src_model = tf.keras.models.load_model(src_model_path)

17: ## Input layer hyperparameters
18: src_sequence_length, src_embedding_dim = tuple(src_model
    .layers[1].output.shape[1:])

19: batch_size = 64
20: num_classes = 10
21: num_epochs = 10

22: ## input image dimensions
23: img_rows, img_cols = src_sequence_length, src_embedding_dim

24: ## Data source
25: data_source = "local_dir" # "keras_data_set" or "local_dir"

26: ## ! Configuration: End
```

Algorithm 49 The `imdbsrc_mnisttgt_train.py` Trainer File: Chunk 2

```
1: def load_data(data_source):
2:    assert data_source in ["keras_data_set", "local_dir"], "Unknown
   data source"
3:    if data_source == "keras_data_set":
4:       (x_train, y_train), (x_test, y_test) = mnist.load_data()
5:    else:
6:       (x_train, y_train), (x_test, y_test) = \
7:       data_helpers.load_local_mnist_dataset\
8:       (
9:          num_classes=num_classes,
10:         img_rows=img_rows,
11:         img_cols=img_cols
12:      )

13:   return x_train, y_train, x_test, y_test

14: ## Data Preparation print("Load data...")
15: x_train, y_train, x_test, y_test = load_data(data_source)
16: x_train, x_test = x_train[..., -1], x_test[..., -1]

17: input_shape = x_train.shape[1:]

18: print('Input shape:  '  + str(input_shape))

19: a0 = Input(shape=input_shape)
20: a = src_model.get_layer(index=2)(a0)

21: ## !  2 Conv Blocks:  Begin

22: ## To simplify the code below, it was assumed that the
   architecture has exactly 1 part consisted of 2 blocks where each
   block contains 3 sequential layers.  So (2 blocks)*(3 layers per
   block) = (6 layers) meaning that 6 layers have to be iterated
   until the blocks merge again into a single block.

23: conv1 = src_model.get_layer(index=3)(a)
24: conv1 = src_model.get_layer(index=5)(conv1)
25: conv1 = src_model.get_layer(index=7)(conv1)
26: conv2 = src_model.get_layer(index=4)(a)
27: conv2 = src_model.get_layer(index=6)(conv2)
28: conv2 = src_model.get_layer(index=8)(conv2)
29: conv_blocks = [conv1, conv2]

30: a = src_model.get_layer(index=9)(conv_blocks)
```

Algorithm 50 The `imdbsrc_mnisttgt_train.py` Trainer File: Chunk 3

```
1: ## !  2 Conv Blocks:  End

2: tgt_model.summary()

3: ## The final merged part of the hidden layers.
4: for src_mode_idx in range(10, len(src_model.layers)-1):
5:    a = src_model.get_layer(index=src_mode_idx)(a)

6: a = Dense(num_classes, activation="softmax")(a)

7: tgt_model = Model(a0, a)

8: tgt_model.compile\
9: (
10:    loss=categorical_crossentropy,
11:    optimizer=Adadelta(),
12:    metrics=["accuracy"]
13: )

14: ## Train the model
15: tgt_model.fit\
16: (
17:    x_train,
18:    y_train,
19:    batch_size=batch_size,
20:    epochs=num_epochs,
21:    validation_data=(x_test, y_test),
22:    verbose=1
23: )

24: tgt_model.save(tgt_model_path, save_format='h5')

25: score = tgt_model.evaluate(x_test, y_test, verbose=0)

26: print('Test loss:  %.30f'%score[0])
27: print('Test accuracy:  %.30f'%score[1])

28: ## Store training output
29: metrics_history = pd.DataFrame.from_dict(tgt_model.history.history)

30: metrics_history = pd.concat([metrics_history['loss'],
   metrics_history['accuracy'], metrics_history['val_loss'],
   metrics_history['val_accuracy']], axis=1)
31: metrics_history.T.to_excel(tgt_model_hist)
```

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *PROCEEDINGS OF THE IEEE*, 1998, pp. 2278–2324.

[2] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, Dec 2020. [Online]. Available: https://doi.org/10.1007/s10462-020-09825-6

[3] A. C. Ian Goodfellow, Yoshua Bengio, *Deep Learning*. MIT Press, 2017, pp. 364–371.

[4] W. Zhang, "Shift-invariant pattern recognition neural network and its optical architecture," 1988.

[5] R. Zhang, "Making convolutional networks shift-invariant again," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 7324–7334. [Online]. Available: http://proceedings.mlr.press/v97/zhang19a.html

[6] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, p. 228, https://d2l.ai.

[7] R. Hirsch, *Exploring Colour Photography: A Complete Guide*. Laurence King, 2004. [Online]. Available: https://books.google.gr/books?id=4Gx2WItWGYoC

[8] M. L. Andrei Bursuc, Florent Krzakala. Neural networks, Convolutions, Architectures. Lecture Slides. [Online]. Available: https://www.di.ens.fr/~lelarge/dldiy/slides/lecture_6/index.html#114

[9] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, pp. 237–241, https://d2l.ai.

[10] Convolution arithmetic tutorial. Theano Tutorial. [Online]. Available: https://theano-pymc.readthedocs.io/en/latest/tutorial/conv_arithmetic.html

[11] M. Kaushik. Backpropagation for convolution with strides: Loss gradient with respect to the input tensor. [Online]. Available: https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710

[12] ——. Backpropagation for convolution with strides: Loss gradient with respect to the filter (weight) tensor. [Online]. Available: https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-fb2f2efc4faa

[13] D. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification." 07 2011, pp. 1237–1242.

[14] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Artificial Neural Networks – ICANN 2010*, K. Diamantaras, W. Duch, and L. S. Iliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101.

[15] C. S. Wiki, "Max-pooling / pooling — computer science wiki,," 2018. [Online]. Available: https://computersciencewiki.org/index.php?title=Max-pooling_/_Pooling&oldid=7839

[16] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, Jul 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0197-0

[17] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, p. 663, https://d2l.ai.

[18] B. Liu, *Sentiment Analysis and Opinion Mining*, 2012, sourced from Microsoft Academic, https://academic.microsoft.com/paper/2108646579.

[19] W.-t. Yih, X. He, and C. Meek, "Semantic parsing for single-relation question answering," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 643–648. [Online]. Available: https://www.aclweb.org/anthology/P14-2105

[20] Y. Shen, X. He, J. Gao, l. Deng, and G. Mesnil, "Learning semantic representations using convolutional neural networks for web search," 04 2014, pp. 373–374.

[21] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *CoRR*, vol. abs/1404.2188, 2014. [Online]. Available: http://arxiv.org/abs/1404.2188

[22] Y. Kim, "Convolutional neural networks for sentence classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 08 2014.

[23] P. R. Christopher D. Manning and H. Schütze. Tokenization. [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

[24] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, p. 310, https://d2l.ai.

[25] ——, *Dive into Deep Learning*, 2020, p. 108, https://d2l.ai.

[26] ——, *Dive into Deep Learning*, 2020, p. 327, https://d2l.ai.

[27] ——, *Dive into Deep Learning*, 2020, p. 664, https://d2l.ai.

[28] R. Ruizendaal. Deep learning 4: Why you need to start using embedding layers. [Online]. Available: https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

[29] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[31] J. Qadrud-Din. Transform anything into a vector. [Online]. Available: https://blog.insightdatascience.com/entity2vec-dad368c5b830

[32] H. Gautam. Word embedding: Basics. [Online]. Available: https://medium.com/@hari4om/word-embedding-d816f643140

[33] M. M. Hossain, D. Talbert, S. Ghafoor, and R. Kannan, "Fawca: A flexible-greedy approach to find well-tuned cnn architecture for image recognition problem," 08 2018.

[34] D. Laredo, Y. Qin, O. Schütze, and J. Q. Sun, "Automatic model selection for neural networks," 05 2019.

[35] S. Bozinovski. (1976) Reminder of the first paper on transfer learning in neural networks. http://www.informatica.si/index.php/informatica/article/view/2828.

[36] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, p. 574, https://d2l.ai.

[37] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, "Transfer learning in natural language processing," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 15–18. [Online]. Available: https://www.aclweb.org/anthology/N19-5004

[38] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, p. 575, https://d2l.ai.

[39] "Keras," 2018, GitHub, Commit: 1a3ee8441933fc007be6b2beb47af67998d50737. [Online]. Available: https://github.com/keras-team/keras/blob/1a3ee8441933fc007be6b2beb47af67998d50737/examples/mnist_cnn.py

[40] A. Rakhlin, "Convolutional neural networks for sentence classification," 2017, GitHub, Commit: 0c7a61f428470d014177ad4a98fef7ad4b86d387. [Online]. Available: https://github.com/alexander-rakhlin/CNN-for-Sentence-Classification-in-Keras/tree/0c7a61f428470d014177ad4a98fef7ad4b86d387

[41] M. Zeiler, "Adadelta: An adaptive learning rate method," vol. 1212, 12 2012.

[42] C. J. B. Yann LeCun, Corinna Cortes. The mnist database. Http://yann.lecun.com/exdb/mnist/.

[43] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[44] H. Iqbal, "Plotneuralnet," 2020, GitHub, Commit: e96bc852189c2089dd500527a0a01a5a36e8977e. [Online]. Available: https://github.com/HarisIqbal88/PlotNeuralNet/tree/e96bc852189c2089dd500527a0a01a5a36e8977e

[45] F. Chollet *et al.*, "Keras," 2015, GitHub. [Online]. Available: https://github.com/fchollet/keras

[46] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[47] M. Cerliani. Tensorflow 2.0: How are metrics computed when the ouput is sequential? [Online]. Available: https://stackoverflow.com/questions/61732344/tensorflow-2-0-how-are-metrics-computed-when-the-ouput-is-sequential#:~:text=Tensorflow%2FKeras%20by%20default%20computes,calculated%20on%20all%20the%20batches.

[48] linanqiu. word2vec-sentiments. [Online]. Available: https://github.com/linanqiu/word2vec-sentiments

[49] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with mnist and emnist," *Applied Sciences*, vol. 9, no. 15, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/15/3169

[50] fl0wer, "Code to produce experiments," 2021. [Online]. Available: https://gitlab.com/fl0wer-roots/undergrad_thesis

[51] "String cleaner python code." [Online]. Available: https://github.com/yoonkim/CNN_sentence/blob/master/process_data.py

[52] R. Řehůřek. Gensim, topic modelling for humans. [Online]. Available: https://radimrehurek.com/gensim/

[53] A. Rakhlin, "Convolutional neural networks for sentence classification," 2017, GitHub, Commit: 0c7a61f428470d014177ad4a98fef7ad4b86d387. [Online]. Available: https://github.com/alexander-rakhlin/CNN-for-Sentence-Classification-in-Keras/blob/0c7a61f428470d014177ad4a98fef7ad4b86d387/data_helpers.py