Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Ανίχνευση εκφράσεων προσώπου με χρήση Βαθιών Συνελικτικών Νευρωνικών Δικτύων** **Facial expression recognition using Deep Convolutional Neural Network techniques**
Ονοματεπώνυμο Φοιτητή	**Αλέξανδρος Ζάγκος**
Πατρώνυμο	**Δημήτριος**
Αριθμός Μητρώου	**ΜΠΣΠ/ 16040**
Επιβλέπων	**Γεώργιος Τσιχριντζής, Καθηγητής**

Ημερομηνία Παράδοσης **Δεκέμβριος 2019**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Γεώργιος Τσιχριντζής Καθηγητής	Διονύσιος Σωτηρόπουλος Επίκουρος Καθηγητής	Ευθύμιος Αλέπης Επίκουρος Καθηγητής

Στο γιο μου Άρη,
πηγή έμπνευσης και απέραντης χαράς για εμένα.

Περίληψη

Η παρούσα διπλωματική εργασία αποτελεί μία μελέτη της ακρίβειας ταξινόμησης του μοντέλου μας στο σύνολο δεδομένων που εξετάζουμε. Για τους σκοπούς της συγκεκριμένης εργασίας χρησιμοποιήθηκε μοντέλο επιβλεπόμενης μάθησης (supervised learning) πρόσθιας τροφοδότησης (feed-forward) με χρήση του αλγόριθμου ανάστροφης μετάδοσης (backpropagation). Τα μοντέλα στα οποία εστιάζουμε είναι τα Συνελικτικά Νευρωνικά Δίκτυα (CNN) τύπου VGG, εκπαιδευόμενα και αξιολογημένα στο σύνολο δεδομένων με το όνομα *FER-2013*. Ο αλγόριθμος μας ταξινομεί σε επτά κατηγορίες συναισθημάτων (Θυμό, Αηδία, Φόβο, Ευτυχία, Θλίψη, Έκπληξη και Ουδετερότητα). Το μοντέλο που χρησιμοποιείται για τη συγκεκριμένη εργασία αναπτύχθηκε σε Python και τρέχει στη CPU. Όλοι οι ταξινομητές μας εφαρμόστηκαν στην διεπαφή προγραμματισμού εφαρμογών Keras (Keras API) με τη χρήση της βιβλιοθήκης TensorFlow. Στόχος μας είναι να βελτιώσουμε την ακρίβεια ταξινόμησης στο σύνολο δεδομένων μας επιλέγοντας διαφορετικές αρχιτεκτονικές και βελτιστοποιώντας τις παραμέτρους του μοντέλου.

Abstract

This dissertation constitutes a study of the classification accuracy of our model on the given dataset. Our supervised learning model uses a feed-forward neural network which we train with the backpropagation algorithm. The models we focus on are VGG-like Convolutional Neural Networks, trained and evaluated on the *FER-2013* dataset. The algorithm classifies on seven emotion categories (Anger, Disgust, Fear, Happiness, Sadness, Surprise, and Neutral). We implemented this model with Python code on the CPU. All of our classifiers were implemented in Keras neural network API using TensorFlow backend. Our goal is to improve the classification accuracy on our dataset by choosing different architectures and by optimizing the model hyperparameters.

List of Figures

List of Tables

List of Abbreviations in Alphabetical Order

1-D	One-Dimensional
2-D	Two-Dimensional
3-D	Three-Dimensional
AAM	Active Appearance Model
ACT	Activation Function
AI	Artificial Intelligence
API	Application Programming Interface
AU	Action Unit
BDBN	Boosted Deep Belief Network
BN	Batch Normalization
BoW	Bag of Words
C3D	3-Dimensional Convolutional Neural Network
CNN	Convolutional Neural Network
C-LSTM	Convolutional Long Short-Term Memory
CONV	Convolutional Layer
CPU	Central Processing Unit
CSM	Cosine Similarity Measure
DAE	Deep Autoencoder
DBN	Deep Belief Network
DCT	Discrete Cosine Transform
DO	Dropout Layer
DR-GAN	Disentangled Representation Generative Adversarial Network
ELU	Exponential Linear Unit
FACS	Facial Action Coding System
FC	Fully Connected Layer
FCP	Facial Characteristic Point
FER	Facial Expression Recognition
FF-GAN	Face Frontalization Generative Adversarial Network
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HoG	Histogram of Gradients
ICA	Independent Component Analysis
ICML	International Conference on Machine Learning
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IP	Internet Protocol
LBP	Local Binary Patterns

LDA	Linear Discriminant Analysis
LSTM	Long Short-Term Memory
MKL	Multiple Kernel Learning
NBC	Naive Bayesian Classifier
NN	Neural Network
NMF	Nonnegative Matrix Factorization
PB- DBN	Pseudo Boost Deep Belief Network
PCA	Principal Component Analysis
POOL	Pooling Layer
QDC	Quadratic Discriminant Classifier
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SIFT	Scale Invariant Feature Transform
SVC	Support Vector Classifier
SVM	Support Vector Machine
TAN	Tree-Augmented-Naïve
TCDCN	Tasks Constrained Deep Convolutional Network
T-LSTM	Time Aware Long Short-Term Memory
TOP	Three Orthogonal Planes
TP-GAN	Two-Pathway Generative Adversarial Network
VGG	Visual Geometry Group

Contents

1
Introduction

It has long been an objective of the research computer vision community to have an imprint and representation of the visual world; capable of recognizing objects in complex scenes. Since the facial expression is one of the most important means for human beings to communicate and interact with the environment and other humans; automated analysis of nonverbal facial behavior has attracted considerable attention in the last decades. Creating an intelligent system similar to human perception system is still an active area of research.

Facial expression recognition (FER) is a process which consists of three main steps:

1. Pre-processing the data.
2. Learn the algorithm to extract facial features from the detected face region (feature learning and feature extraction).
3. Analyzing the motion of these features or the changes in their appearance and finally classifying them into some categories (facial expression classification and interpretation).

Deep neural networks can perform FER in an **end-to-end** way, unlike the traditional methods, where the feature extraction step and the feature classification step are independent [22].

In this work, we implemented several deep learning models with different architecture on our dataset. We use a dataset of 35,887 images, called *FER-2013*, as described in Chapter 3. The models we focus on are Visual Geometry Group (**VGG**) [1] Convolutional Neural Networks (**CNN**) [2, 3]. We train the classifier on a labelled subset of *FER-2013* and evaluate the model on the test and validation set, respectively. The model classifies on the six basic emotion categories (Anger, Disgust, Fear, Happiness, Sadness, Surprise) as indicated by Paul Ekman [4] along with the addition of a seventh Neutral emotion category.

Conclusions regarding the spotted accuracy and the reliability of the examined architectures depict in Chapter 5 and Chapter 6.

2
Literature Review

The study of visual data (Computer Vision) has become ubiquitous in our modern world since visual data have increased to a significant extent in the last years. In a recent study [133] carried out by Cisco on February 27, 2019, the estimation is that by 2022, globally IP video traffic will be 82% of all IP traffic on the internet, 7% greater than 2017. We can understand that the majority of the bits on the internet are visual data. Developing algorithms that can analyze and interpret these data is crucial. Computer Vision touches a lot of other research fields such as Computer Science, Mathematics, Engineering, Physics, Biology and Psychology (Figure 2.1). In Computer Vision field, a lot of facial expression recognition models have been investigated to encode expression information from facial expressions.

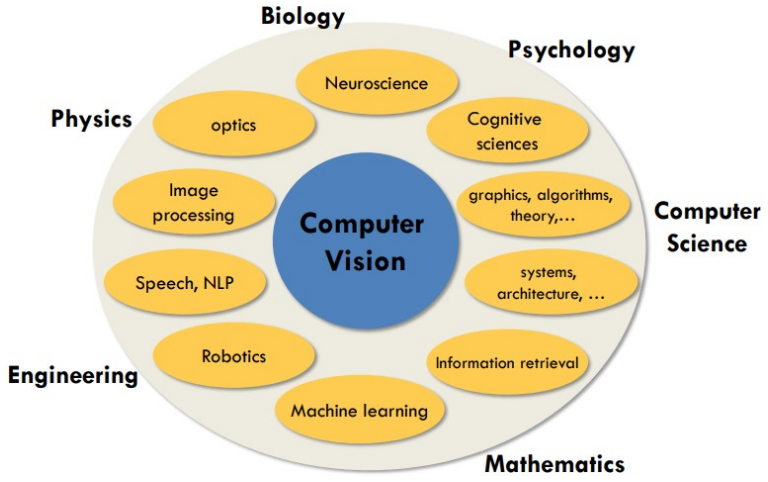


Figure 2.1: Computer vision regarding other research fields.
Retrieved from http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture01.pdf

2.1 History of Computer Vision

One of the first research that influenced Computer Vision community comes from the Biology field in the early 60s. More specifically, Hubel and Wiesel [5], [6] studied visual processing in mammals. They inserted microelectrodes in the primary visual cortex of a cat (striate cortex) and observed a distinct pattern by which specific neurons of mammals detect edges of the image seen by the eye. The complete map of these receptive fields of the cortical unit, stimulated by the image seen by the eye, consisted of a vertically oriented region. By all, they discovered that visual processing starts with a simple structure of oriented edges and as the information moves along to the structured area of the brain, the mammals begin to build the information they receive and finally recognize the visual world.

In 1963, Lawrence Roberts [7] in his thesis built a program which processes a two-dimensional (2-D) photograph into a line drawing and transforms it into a three-dimensional array (3-D). This geometric shape recognition and reconstruction model is one of the first in the field of computer vision.

In the Summer Vision Project [8] of the MIT in 1966, the researches of the AI group set the bar high. This summer project attempted to effectively construct a significant part of a visual system complex enough to be a landmark in the development of pattern recognition.

David Marr, with his work [9], described a general framework for studying and understanding the visual world and devised a systematic approach to edge detection. With his work in the 70s, he explained in detail the theory of early vision as proceeding from the input image to the three-dimensional output representation of objects. David Marr's model consists of three stages. The researcher implemented the so-called primal sketch for feature extraction of components as edges, ends, curves, boundaries etc. In the next step, we piece together these components to analyze surfaces, textures, depth information etc. of the visual input ending up with a three-dimensional representation as an output of the model.

In the early study of object recognition from images it worth mentioning the work of Fischler and Elschlager [10] and David Lowe [11] in 1973 and 1987 respectively. In both works, every object is analyzed in basic geometric shapes and structures.

Since a holistic approach of object recognition from images had major bottlenecks at the time, another method changed the course of the history of vision. That is feature-based object recognition. These approaches depict that regardless of the variation, deformation, occlusion or illumination condition changes in the environment or the object we are studying, some feature-based characteristics remain the same. Pattern matching studies like David Lowe's [12] in 1999 identify these features that stay the same and match these features to another object. David Lowe presented a new method for image feature generation called the Scale Invariant Feature Transform (SIFT).

It is worth mentioning the crucial work of Paul Viola and Michael Jones in 2001 [13], which used the AdaBoost algorithm [14] along with statistical methods for real-time face recognition based on a set of rectangle features.

The recognition of holistic scenes along with human recognition started to grow in the early 00s, with studies [15], [16], that used feature-based algorithms for feature extraction and Support Vector Machines (SVM) on top, for classification.

2.2 Facial Expression Recognition

Facial expression analysis and recognition refer to computer systems that attempt to automatically analyze and recognize facial feature changes and facial motions from visual information. There are various ways to represent the face: as a set of features (analytic representation), as a whole unit (holistic representation), or as a combination of these [17]. From a temporal point of view, facial expression recognition models can divide into two main categories, static-based methods and dynamic-based methods [18].

Feature extraction remains a crucial problem in pattern recognition, such as facial expression recognition. From a feature extraction point of view, the techniques aiming to recognize facial expressions are categorized into methods that use geometric features and methods that use appearance features [19] although some hybrid-based approaches are also possible. With geometric-based methods, facial features present the shape and locations of facial components (eyes, mouth, eyebrows, nose) and the location of salient facial points (corners of the eyes, mouth). In geometric feature extraction system, shape, angles, distances, or the coordinates of facial fiducial points are extracted, forming a feature vector representing the face geometry. A typical example of a geometric-based model is this of Kotsia and Pitas [36] in 2007, who used a 3-D face model named Candide, initially proposed by Jorgen Ahlberg [38].

Appearance representations use texture-based methods, considering the intensity values of the pixels. With appearance-based techniques, image filters such as Gabor wavelets, for

example, are applied on the whole face or some regions of the face to extract changes in facial appearance, forming a feature vector. In the appearance-based method, features represent the facial texture, including bulges, wrinkles and furrows. A typical example of appearance-based models is this of Bartlett et al. [37], who used Gabor wavelets.

As previously stated, a hybrid approach can be used, using both geometric and appearance methods. An example of such an approach is this of Lucey et al. [39] who used an Active Appearance Model (AAM) to record the shape of facial expressions and the characteristics of facial appearance.

The feature vectors formed from either geometric or appearance-based methods are used for facial expression classification. The classifier divides the extracted features into the relevant categories according to the corresponding classification mechanism.

In terms of facial expression classification, Facial Action Coding System (FACS), which was developed by Paul Ekman and Wallace Friesen [20], is one of the most known studies on facial activity and has been considered as a foundation for describing facial expression classification. This cross-cultural study on the existence of universal categories of emotional expressions, defined six categories, referred to as the basic emotions: Anger, Disgust, Fear, Happiness, Sadness and Surprise. It also provides a description of all, visually detectable, facial changes in terms of 44 so-called Action Units (AUs) [17]. Although recent research on psychology and neuroscience [21] argued that this model is culture-specific and not universal, this categorical model is still the most popular perspective for Facial Expression Recognition, due to its pioneering investigations along with the direct and intuitive definition of facial expressions [22]. The main advantage of this categorical emotion representation is that people use this scheme to describe observed emotional displays in real life. This labelling scheme is very instinctive and thus matches people experience [25].

2.3 Non-Deep Approaches of Facial Expression Recognition

Complete surveys on automatic facial expression analysis have published in recent years of Facial Expression Recognition [17, 19, 23, 24, 25, 26]. These surveys established a set of standard algorithmic pipelines for Facial Expression Recognition. In this section, will be summarized some of the traditional state-of-the-art models of this early era of automatic facial expression analysis.

Early efforts toward Facial Expression Recognition include studies like Kobayashi and Hara in 1991 [27]. In this feature-based study, researchers proposed a method of back-propagation Neural Network (NN) for recognizing human emotions through a CCD camera. With this camera-acquired method, they extracted data of Facial Characteristic Points (FCP) from 3 components of the face (eyebrows, eyes and mouth). The classification is based on the six basic categories, as proposed by Paul Ekman and Wallace Friesen [20].

Padgett and Cottrell [28], in 1996, use a holistic face representation with Principal Component Analysis (PCA) combined with a back-propagation NN. The input to the NN was the normalized projection of the seven extracted blocks of the PCA. The hidden layer of the NN employs a nonlinear Sigmoid activation function. The output layer of the NN contains seven units, each of which corresponds to one emotion category. Padgett and Cottrell used the images of six basics plus a neutral expression.

Essa and Pentland [29], in 1997, applied the eigenspace method of Pentland et al. [30], using PCA to automatically locate the faces in an arbitrary scene and extract the positions of the eyes, nose, and mouth. This method applied to frontal-view facial image sequences. Essa and Pentland use the Optical Flow computation method proposed by Simoncelli [31]. The method is real-time and has been successfully tested on a database of people, having different head

positions, illumination changes, headwear, facial hair and/or eyeglasses. In this holistic motion model, Essa and Pentland, employed sophisticated 3-D motion and muscle models for facial expression recognition and increased tracking stability. They also proposed FACS+, an extension to FACS which consists of a set of control parameters using vision-based observations, also describing the dynamics of facial expressions.

In 1998, Hong et al. [32] proposed an online facial expression recognition system in order to perform real-time tracking of faces. He used the PersonSpotter system by Steffens et al. [33]. The face dimensions were obtained by fitting a labelled graph into the input facial image. The face was previously detected by the PersonSpotter system utilizing the method of elastic graph matching.

Zhang et al. [34], in 1998, use a hybrid approach to face representation. They use FCP for which a set of Gabor wavelet coefficients extracted. A NN with backpropagation algorithm has used for classification. A similar face representation was used by Lyons et al. [35], in 1999, for expression classification into the basic emotion categories. They used a Fiducial Grid with a set of Gabor wavelet coefficients. For classification, PCA and Linear Discriminant Analysis (LDA) used on the labelled graph vectors.

Cohen et al. [40], in 2003, described a real-time face tracking system used for feature extraction developed by Tao and Huang [41]. Cohen et al. described several different classifiers developed for recognizing the facial expressions and introduced the Tree-Augmented-Naive Bayes (TAN) classifier as introduced by Friedman et al. [42] in 1997. They also introduced a multi-level Hidden Markov Model (HMM) architecture for automatic segmentation and recognition of emotions.

In 2004, Ma and Khorasani [43], proposed a new technique for Facial Expression Recognition using a 2-D Discrete Cosine Transform (DCT) as a feature detector, implemented over the entire face images. As a facial expression classifier, they used a constructive one-hidden-layer feedforward NN providing improved generalization and recognition performance capabilities.

Shan et al. [44] used Local Binary Patterns (LBP) for feature extraction, showing that LBP features are robust to low resolution. For facial expression classification, template matching and SVM has adopted.

A little work has been done until that time in 3-D facial expression recognition model. Wang et al. [45], in 2006, demonstrated the advantages of a 3-D geometric based approach over 2-D texture-based approaches for Facial Expression Recognition. For classification, four classifiers used and tested, the Quadratic Discriminant Classifier (QDC), LDA, Naive Bayesian Classifier (NBC), and Support Vector Classifier (SVC). Another 3-D approach at this time is the approach of Kotsia and Pitas [36], with the Candide model, in 2007.

Axel Panning et al. [46] in 2008, proposed a novel approach for facial feature detection in color image sequences using Haar-like classifiers since they had already been utilized successfully for face detection by Paul Viola and Michael Jones in 2001 [13]. This research group built a combination of Haar-like-Feature detection and skin color detection for face detection. For classification, they trained and used a full connected feed-forward NN with sigmoid nodes.

Buciu et al. [47], take Independent Component Analysis (ICA) as the baseline for feature extraction and test another five ICA approaches. For classification, they use either a Cosine Similarity Measure (CSM) classifier or SVM.

In 2012, Valstar and Pantic [48], proposed a method that enables the detection of a much more extensive range of facial behavior by recognizing AUs, and models their temporal characteristics (temporal segments) as neutral, onset, apex, and offset. For feature extraction

and thus localization of the fiducial points, they use a Gabor-based facial point detector. For classification, a combination of GentleBoost, SVM, and HMM applied.

Most recent approaches aim to obtain high-level data-driven representations to encode features [26]. Nonnegative Matrix Factorization (NMF) algorithm is such an approach. NMF, based on the idea that negative numbers are meaningless in a physical way in a lot of data-processing tasks; it thus finds a non-negative decomposition of the original data matrix in non-negative matrices. This method represents a facial image as a linear combination of basis images consisting of basis vectors. These basis vectors represent eyes, nose, mouth, etc. One NMF technique is Graph-Preserving Sparse Nonnegative Matrix Factorization (GS-NMF) [49]. Another NMF approach is Subclass Discriminant Nonnegative Matrix Factorization (SD-NMF) [50].

Sparse Representation for facial expressions recognition is based on the idea that every image is sparse in some areas. In this domain, most coefficients of the transformed image are zero. The transformation can be adaptive or non-adaptive and is based on a so-called dictionary [53]. The computational complexity of these algorithms depends on the optimization algorithm and the size of the dictionary. The approaches of Mahoor et al. [51] along with Zafeiriou and Petrou [52] describe this method.

2.3.1 Discussion

In this section, we will attempt to categorize the methods denoted in the previous section depending on feature representation, feature extraction and classification-recognition.

Spatial Representations:
Spatial representations encode the input image sequences frame-by-frame. Appearance-based representations are more common and encode low or high-level subspaces of the image.

Low-level representations extract local features, such as edges, encoding them in a transformed image. They perform clustering of the local features into uniform regions. In the final stage, they pool the features of each area with local histograms and concatenate all local histograms to extract the final representation [26]. Representation such as LBP [44, 54] that describes local texture variation along a region with an integer, and the Histogram of Gradients (HoG) [15] that extracts local features and representing images by the direction of their edges are prevalent. Another low-level feature representation is the Gabor representation [37, 48].

High-level representations aim to obtain high-level data-driven semantic representations of objects, faces, and scenes. High-level approaches include NMF [49, 50] and Sparse Representation [51, 52].

Bag of Words (BoW) representation extracts SIFT [12] descriptors either from the whole image or from a spatial pyramid dividing the image into subregions. This approach represents images as normalized presence vectors of visual words. Radu Ionescu, Marius Popescu, and Cristian Grozea [106] provided a submission using a feature extraction BoW model and got the fourth place in *FER-2013* competition.

Haar-like features [46], utilized initially by Paul Viola and Michael Jones [13] for face detection, consider of rectangular regions at a specific location in an image window. These features categorize the subsections of an image by summing up the pixel intensities in each region, calculating the difference between these sums. The position of the rectangles is defined relative to the detection image window that acts like a bounding box to the target object (the face).

Spatio-Temporal Representations:
Spatio-temporal representations enable modelling of the temporal variation to represent subtle expressions more efficiently, considering a range of frames within a window as a single entity [26]. Most spatio-temporal representations are appearance-based representations.

Geometric Features from Tracked Facial Points is a representation describing facial activity through fiducial points and localize these points. Valstar and Pantic [48] used this method to recognize AUs with their temporal segments (neutral, onset, apex, and offset).

Low-level representations such as Three Orthogonal Planes (TOP) representation, is a popular approach of low-level feature extraction and initially emerged when extending LBP to LBP-TOP [55].

Feature Extraction:
Feature extraction methods extract features from the initial representations mapping them onto a lower dimensional space in order to discover their structure. Most popular linear transformations are DCT [43], PCA [28, 29, 34] and LDA [34, 45].

Classification-Recognition:
Most of traditional facial expression recognition models use machine learning techniques for expression recognition. Most of these methods rely on SVMs for classification [44, 47, 48]. In order to improve model prediction, statistical models such as HMM and Boosting techniques are combined with SVM [48].

2.4 Entering a New Era for Object Recognition

In the early 00s, the field of Computer Vision has defined a significant problem to solve -the object recognition problem. At this time, we began to have benchmarked datasets. From the first attempts for benchmarked data [56] to the influential datasets such as PASCAL [57, 134] and ImageNet [58, 135], one thing is for sure, that benchmarks pushed forward the algorithm development for object recognition. These datasets play a vital role since they enable to measure the progress in the object recognition problem.

Another key point that took the object recognition problem to another level is international recognition challenges that helped to measure the progress of Computer Vision algorithms by checking the classification results. The year 2012, was written in the history of Computer Vision and object recognition. Alex Krizhevsky et al. [128] trained a deep CNN to classify the 1.2 million high-resolution images of the ILSVRC-2010. This winning model beat all the other models in this contest, dropping the error rate significantly regarding all previous classifications efforts. The current intensity of commercial interest in deep learning began in 2012, but CNNs had been used to win other machine learning and computer vision contests with less impact for years earlier [83]. It is evident that benchmarked data and international challenges, along with the dramatically increased chip processing abilities and the use of GPUs, changed the history of object recognition. Well-designed network architecture studies have begun to transfer to deep learning methods. These deep learning methods have achieved state-of-the-art recognition accuracy and exceeded previous results by a wide margin

2.5 Deep Approaches of Facial Expression Recognition

In this section, there will be an attempt to describe the three main steps of deep Facial Expression Recognition models and survey the deep approaches used for FER.

Pre-processing:
Most of the existing traditional approaches on FER are based on engineered features (e.g. HOG, LBP, Gabor filters), as depicted in Section 2.3, where the classifier's hyperparameters are tuned to give best classification accuracies across the dataset [59].

Several models use cascade function in order to map the images to a landmark location. The prediction and localization of landmarks using cascaded CNNs, proposed originally in 2013 by Yi Sun et al. with the design of a three-level CNN [60]. Later works proposed multi-task algorithms for facial landmark detection, such as Tasks Constrained Deep Convolutional Network (TCDCN) [61] and Multi-task Cascaded Convolutional Networks [62].

In 2014, Goodfellow et al. [63], proposed a new framework called Generative Adversarial Network (GAN) applied for Data Augmentation. As denoted in this work, GAN could improve the performance of classifiers when limited labelled data is available. Later works use GAN methods to generate realistic faces, and other types of images, varying in poses and expressions (image synthesis) improving recognition tasks [64, 65]. In 2017, researchers proposed three different models for frontal view synthesis. The models are the Two-Pathway Generative Adversarial Network (TP-GAN) [66], the Face Frontalization Generative Adversarial Network (FF-GAN) [67] and the Disentangled Representation Generative Adversarial Network (DR-GAN) [68].

Feature Learning:
Deep learning approaches promise to discover rich, hierarchical models that represent probability distributions over the kinds of data encountered in Computer Vision and Artificial Intelligence (AI) applications [63]. The research theory for facial expression recognition based on deep learning mainly focuses on four methods: Convolutional Neural Networks (CNN), Deep Belief Networks (DBN), Deep Autoencoders (DAE) and Recurrent Neural Networks (RNN).
Convolutional Neural Networks proposed in the late 80s [2, 3] and have their roots the Neocognitron [69] Neural Network. CNN is a deep, feedforward network, more comfortable to train with much better generalization than other networks [70]. Many works from the early 00s to the present point depict the predominance of CNN models compared to traditional methods in FER [129, 130, 131, 132]. Further information regarding CNN architecture listed in Section 4.9.

Since objects of interest in the image might have different spatial locations and different aspect ratios, a vast number of regions might be selected for feature learning. Finding these regions is computationally expensive. Therefore, algorithms like R-CNN [71], Fast R-CNN [72] and Faster R-CNN [73] have been developed to find these regions in a fast way. The region-based CNNs (R-CNN, Fast R-CNN and Faster R-CNN) identify facial expressions by generating region proposals. In the YOLO approach [74], the algorithm does not look at the entire image searching for regions. It predicts multiple bounding boxes and class probabilities for those boxes with a single CNN and extracts features from the image.

Regarding 3-Dimensional Convolutional Neural Networks (C3D), in 2013, Ji et al. [75] developed and proposed a novel 3-D CNN model for 3-D spatio-temporal feature learning and action recognition. This model generates multiple channels of information from the input, and the final representation combines information from all these channels. Tran et al. [76], in 2015, proposed a C3D for the same purpose. This model was trained on a large-scale supervised video dataset using 3-D convolutional kernels with shared weights, instead of the traditional method of 2-D kernels.

Deep Belief Networks (DBN) originally introduced in 2006 by Geoffrey Hinton et al. [77]. DBNs are composed of multiple layers of stochastic hidden units (or feature detectors) with binary values in their typical form. The top two layers have undirected, connections between them while the lower layers receive top-down, directed connections from the layer above [136].

A DBN can be viewed as a stack of Restricted Boltzmann machines (RBM) [78, 79] that contains a visible unit layer that represents the data and a hidden unit layer representing features that capture higher-order correlations in the data. This layer-by-layer top-down procedure helps the model to learn the weights that determine how the variables in one layer depend on the variables in the layer above. Modern studies propose algorithms like Boosted Deep Belief Network (BDBN) [80], that performs feature learning, feature selection and classifier construction in a unified loopy framework, and Pseudo Boosted Deep Belief Network (PB-DBN) [81] where the top layers of the DBN boosted while the lower layers of the base classifiers share weights for feature extraction.

Deep Autoencoders (DAE) [82] objective is to minimize the reconstruction error between input and output. DAE can convert high-dimensional data into low-dimensional codes by using a small central layer to reconstruct high-dimensional input vectors.

Recurrent Neural Networks (RNN), first introduced in 1986 [84], are a family of neural networks for processing sequential data [83]. RNNs include recurrent edges that share the same parameters across every step and cover neighbouring time steps [22]. RNNs process an input sequence one element at a time, thus are better for language and speech processing and other sequential inputs. They maintain in their hidden units a vector that contains information about the history of all the past elements of the sequence. RNNs are robust systems but have a problem in the training procedure because the back-propagated gradients either grow or shrink at each time step leading them to either vanish or explode [70]. Long Short-Term Memory (LSTM) is a type of gated (controlled by another hidden unit) RNN introduced in 1997 by Hochreiter and Schmidhuber [85] to solve gradient problems regarding RNNs. Hochreiter and Schmidhuber introduced self-loops to produce paths where the gradient can flow for long durations. In 1999, Gers et al. [86] made another important addition to LSTM model by making the weight on this self-loop conditioned on the context, rather than fixed. LSTM models are proved very successful in speech recognition [90], machine translation [87, 88, 89], handwriting recognition and generation [91, 92], and image captioning [93, 94].

Neural Network Ensemble:

Ensemble learning is the technique of training multiple neural networks models instead of one and finally combine the predictions from these models. Researchers suggest that Neural Network Ensembles can improve generalization performance [95] and classification accuracy [96] of the model.

In 2015, Hamester at al. [97], proposed a model of a channel of unsupervised DAE combined with a standard CNN channel. Results show that the addition of this unsupervised channel improved test accuracy and overall training time. Liu et al., in 2016, proposed an ensemble of different structured CNNs for improving test classification accuracy on *FER-2013* [99].

In 2018, an end-to-end convolutional architecture proposed for spatio-temporal FER [100]. Researchers coupled a C3D network with a Nested LTSM. The Nested-LSTM composed of two sub-LSTMs, the Time Aware LSTM (T-LSTM) and the Convolutional LSTM (C-LSTM).

Cascaded Networks:

Cascade networks are similar to feed-forward networks, including a connection every previous layer to the following layers. In this type of NN, various modules are combined sequentially, forming a deeper network.

Lv et al., in 2014, trained a model with DBN for component detectors. These component detectors first detect the face and then detect nose, eyes and mouth in a hierarchically way. For FER, A deep architecture with stacked autoencoder is applied [101].

Baccouche et al. [102], proposed a spatio-temporal convolutional sparse autoencoder model with LSTM for sequence classification.

Facial Expression Classification:
The final step of FER algorithms is the classification of the given facial image into one of the basic expression (emotion) categories. There are two approaches to deep learning models.

In the end-to-end approach, feature extraction and feature classification steps are not independent. In CNNs, a loss function layer is added inside the model. Softmax classifier and SVM are the two most used functions to minimize this loss function. Finally, the model extracts the prediction probabilities for each category.

Besides the above end-to-end approach, we can use a deep network as a feature extractor and apply an independent classifier for classification.

2.5.1 Discussion

Deep learning approaches, the most recent developments in neural networks, have significantly advanced the performance of visual recognition systems. These models not only emphasize in depth (as its name implies) but highlight the performance of feature learning and feature extraction [22]. Deep learning models have made some research achievements in image recognition tasks and especially in Facial Expression Recognition.

When we are implementing a facial expression recognition model, usually we do not have a lot of training data. Thus, the model cannot generalize very well, and overfitting happens too fast because of the complexity of image data. With pre-processing techniques like Data Augmentation, we strengthen the network's robustness to common distractions (head pose variations, illumination and occlusion) forcing the model to focus in more facial areas with valuable information.

Neural network ensembles integrate various networks at the feature or decision level for both spatial and temporal information to help boost their performance. These methods usually enlarge the computational cost and the storage requirement since they use different kinds of networks [22].

Cascaded networks can train sequentially various networks in a hierarchical way. In general, this method can reduce overfitting problem and disentangling factors that are irrelevant to facial expressions [22].

3
Dataset Description

The ICML 2013 Workshop on Challenges in Representation Learning [137], organized by LISA lab of Montreal University, focused on three challenges [103]. The facial expression recognition challenge was one of them. In this challenge hosted on Kaggle [138], the competitors were invited to design the best system for emotion recognition. *FER-2013* dataset introduced to the contest as an entirely new dataset. *FER-2013* was prepared by Pierre Luc Carrier and Aaron Courville, as part of an ongoing research project.

The images of faces collected using the Google image search API (Figure 3.1). OpenCV [139] used for face recognition in the collected images. Human labelers then approved the correctly labeled, filtered out some duplicate images, corrected the cropping if necessary and rejected the incorrect ones. These 35,887 images resized in 48x48 pixels and transformed into grayscale. The entire set consists of 35,887 grayscale images divided into three sets, the training set, the public set and the private set. The training set consists of 28,709 examples; the public validation set consists of 3,589 examples and the private test set consists of 3,589 examples. The *fer2013.csv* file contains three columns, “emotion”, “pixels” and “usage” respectively. The “emotion” column contains a numeric code from 0 to 6, depending on the emotion in the image. The “pixels” column contains a list of 48x48=2,304 pixels representing each face. The “usage” column refers to the usage of the image, whether it is for training, validation or testing. The task is to categorize the facial expression of each face into one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). The set contains 4,953 “Angry” images, 547 “Disgust” images, 5,121 “Fear” images, 8,989 “Happy” images, 6,077 “Sad” images, 4,002 “Surprise” images and 6,198 “Neutral” images.

Figure 3.1: Example images from the FER-2013 dataset

Ian Goodfellow, one of the organizers of ICML 2013 found that human accuracy on *FER-2013* was 65±5% [103]. James Bergstra determined the best performance using an ensemble of “Null Models” (this model presented after the contest was over) obtaining an accuracy of 65.5% on the test data [104]. This model uses an ensemble construction method called SVM HyperBoost, based on Boosting method [98]. Among the 56 challenge participating teams submitted on the final test, only four beats this “Null” model (Table 3.1). All top three teams used Convolutional

Neural Networks, including the winner Yichuan Tang who submitted the winning solution with a private test score of 71.2% [105].

Table 3.1: Dataset accuracy on FER-2013 for the four first teams

Team Members	Private Set Accuracy [138]	Public Set Accuracy [138]
·YichuanTang [105]	71.161%	69.768%
·YingboZhou, ChetanRamaiah	69.267%	69.072%
·MaximMilakov	68.821%	68.152%
·Radu Ionescu, Marius Popescu, Cristian Grozea [106]	67.483%	67.316%

Radu Ionescu, Marius Popescu, and Cristian Grozea [106] provided a submission using a hand-engineered feature extraction model instead of a feature learning one. Their approach used a BoW model [107], [108] with SIFT features [12] and linear kernels, combined into weighted sums for Multiple Kernel Learning (MKL). This approach represents images as normalized presence vectors of visual words.

4
Basic Principles of Deep Learning

Deep learning algorithms, is a specific type of machine learning algorithms. To continue with the CNN architecture implemented in this thesis, we must cite some of the basic principles of machine learning and deep learning algorithms, respectively.

Machine learning algorithms are algorithms that can learn from data and have the ability to find patterns and generalize to new data. The different settings we make when we design our algorithm, that controls the learning process, are called **hyperparameters**. One of the critical issues for accurate and effective deep learning algorithms is to optimize (tune) the hyperparameters by choosing the optimal set.

The **optimization** of our **supervised learning** algorithm is based on an extension of the gradient descent algorithm called Stochastic Gradient Descent (SGD) and on an adaptive learning rate optimization algorithm called Adam. The task of these optimization algorithms is to minimize a function called the **loss function** in order to perform an accurate mapping of the input images to a specific category.

In this section, we discuss feed-forward neural networks which we train with the **backpropagation** algorithm.

Finally, we discuss techniques to perform well in **generalization**, including methods to avoid overfitting and underfitting and **regularization** techniques such as L2 regularization, dropout and Data Augmentation. The theory provided in this chapter is from the Deep Learning book from Ian Goodfellow et al. [83], the Neural Networks and Deep Learning [140] online book by Michael Nielsen and CS231n: Convolutional Neural Networks for Visual Recognition class notes [141] from Stanford University.

4.1 Image Classification

The task of this thesis is to classify with accuracy the given input images into seven categories. Our learning algorithm is asked to produce a function $y = f(x)$ and assign an input vector ***x*** to a category identified by a numeric code (label) $\hat{y}$.

Image Classification Pipeline:

- Our **input** consists of a training set with 28,709 images; each labelled with one of the seven classes (Angry, Disgust, Fear, Happy, Sad, Surprise and Neutral). The pixels vector **x** contains a list of 48x48x1=2,304 pixels representing each face.
- The job of the algorithm is to **train a classifier** (learn a model) by using the training dataset to learn what every one of these seven classes looks like.
- Finally, the algorithm will **evaluate** the classifier by predicting labels (Angry, Disgust, Fear, Happy, Sad, Surprise and Neutral) for a new unseen dataset called the testing dataset. The predicted labels will be compared with the ground truth (correct labels) of these images to evaluate the accuracy of the model.

The input of our model consists of a vector with 2,304 pixels per image, and the output consists of a distinct price from the set {0,1,…,6} with 0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral; thus the score function we want to produce is: $f : \mathbb{R}^{2304} \rightarrow \mathbb{R}^{7}$.

4.2 Supervised Learning

Supervised learning is the machine learning task of learning a function, from a pre-labelled training dataset, that maps an input to an output. We assume a mapping $f : X \to Y$, where X is the input space and Y the output space. Every example in the dataset (in our case in the image dataset) is a data point. Each data point is a pair $(x, y) \in X \times Y$ consisting of an input vector and the desired output value (a label). The algorithm analyzes the training data, discovers underlying patterns and produces a function. This function is used for mapping new examples. The optimal scenario will allow the algorithm to correctly classify unseen data points and thus, to generalize.

In this section, we will describe the three basic functions we use in our supervised learning algorithm. The function that maps examples to predicted labels (**score function**), the function that quantifies the mismatch between the predicted labels and the ground truth labels (**loss function**), and the function that evaluates the complexity of the mapping (**regularization function**).

Loss Function:
The function that maps an event or an input to an output, as described above, representing some cost associated with the event is called a loss function or a cost function. A loss function tells how good our current classifier is.

We assume a training dataset $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i \in X$ are the inputs and $y_i \in Y$ the labels associated with these inputs, made up of independent and identically distributed data (i.i.d. assumptions) from a data generating distribution. In our model, we have a training set of *N*=28,709 images. Since the input consists of a flattened 2,304-pixel vector, $x_i \in \mathbb{R}^{2304}$. Our distinct label categories are seven, thus $y_i \in \mathbb{R}^7$. Note that we use $\hat{y}$ to denote the predicted label values of the model instead of ***y*** which are the ground truth labels of the dataset. Our **scalar-valued** loss function measures the disagreement between $\hat{y}_i$ and the ground truth label y_i. By optimizing the loss only over the available training dataset, we consider the loss over the entire dataset (total loss) ***L*** as an average of the loss over the examples (data loss) L_i as we can see in Equation 3.1.

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i\left(\hat{y}_i, y_i\right) \tag{3.1}$$

A given $x_i \in X$ is classified correctly if $L_i = 0$.

Multiclass SVM Classifier and Hinge Loss:
Support Vector Machines [109, 110] are **non-probabilistic** supervised algorithms used for classification. Although they designed for binary classification, recent approaches extend SVM to handle multiclass classification problems [111, 112].

Given an example $(x_i, y_i),\ \mathrm{i} = 1,\ldots,N$, where $x_i \in X$ are the input images and $y_i \in Y$ the ground truth labels associated with these inputs (as described above), a linear **score function**

$$\hat{y}_i \equiv s = f\left(x_i, \mathrm{w}\right) = wx_i \tag{3.2}$$

takes the pixels and computes the vector of class scores. The parameters in matrix ***w*** are called the **weights** of this function.

The SVM is set up in a way that the score of the correct classes should be higher than the sum of scores of all incorrect categories by some safety margin Δ set to Δ=1 in all cases. The SVM loss for the $j-th$ element has the form:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & if\ s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1, & otherwise \end{cases} = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right) \tag{3.3}$$

Since we work with a linear score function $s = f\left(x_i, \mathrm{w}\right)$, we can rewrite Equation 3.3:

$$L_i \overset{(3.2)}{=} \sum_{j \neq y_i} \max\left(0, f\left(x_i, \mathrm{w}\right)_j - f\left(x_i, \mathrm{w}\right)_{y_i} + 1\right) \tag{3.4}$$

It is worth mentioning that this threshold at zero $\max\left(0, -\right)$ function used for maximum-margin classification is also called **hinge loss** function. Essentially, the hinge loss function is summing across all incorrect classes and compares the output of the scoring function ***s*** returned for the $j-th$ class label (the incorrect class) and the $y_i - th$ ground truth label (the correct class). We apply the max operation to stabilize values at zero, which is essential to ensure we do not sum negative values. To derive the loss across the entire training set as described by Equation 3.1, we take the average over each L_i:

$$\begin{aligned} L &= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right) \overset{(3.2)}{\Longrightarrow} \\ \Rightarrow L &= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max\left(0, f\left(x_i, \mathrm{w}\right)_j - f\left(x_i, \mathrm{w}\right)_{y_i} + 1\right) \end{aligned} \tag{3.5}$$

Softmax Classifier and Cross-entropy Loss:

The most commonly used function for classification is the Softmax classifier with a cross-entropy loss. This function classifies scores as **probabilities**. While SVM classifier faces the output of the model as scores for each class, Softmax classifier normalizes the output into a probability distribution for each class.

The Softmax function is often used to predict the probabilities associated with a Categorical distribution, which is a generalization of the Bernoulli distribution. The sample space in this distribution is taken to be a finite set of integers. Given a score function $s = f\left(x_i, \mathrm{w}\right)$, we use the Softmax function for the $j-th$ element (Equation 3.6):

$$P\left(Y = y_i \,|\, \mathrm{X} = \mathrm{x}_i\right) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \tag{3.6}$$

Since we do not know the probability distribution ***p*** a priori, we use the estimated probability ***q*** and use the Cross-entropy (Equation 3.7) to minimize the loss between the estimated and the real probability distribution.

$$H(\mathrm{p}, \mathrm{q}) = -\sum_x p(x) \log \mathrm{q}(x) \tag{3.7}$$

If we put it all together by minimizing the Cross-entropy between the estimated class probabilities and the correct distribution, we get the loss function (Equation 3.8)

$$L_i = -\log P\left(Y = y_i \,|\, \mathrm{X} = \mathrm{x}_i\right) = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \tag{3.8}$$

Or equivalently:

$$L_i = -s_{y_i} + \log\sum_j e^{s_j} \tag{3.9}$$

Regularization:
Regularization is the process of making modifications (add information) to our learning model to reduce its generalization error, thus prevent overfitting. However, too much regularization can limit the capacity of the model, thus underfit the training data (Figure 4.1).

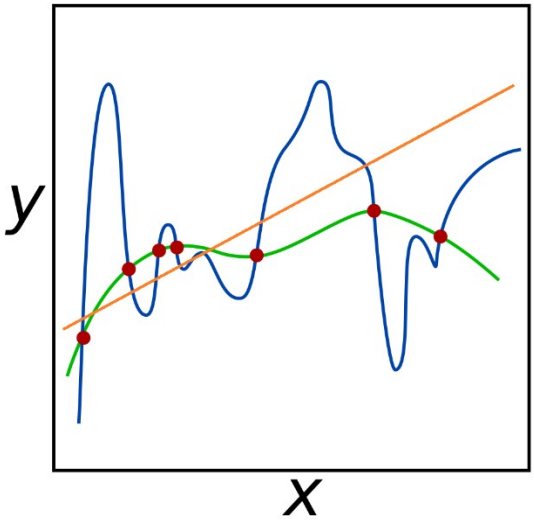


Figure 4.1: The blue line is an example of overfitting; the algorithm cannot generalize. The green line may generalize well on the data points. It is worth mentioning that both blue and green functions give zero loss on the given dataset. The orange line is an example of underfitting since our model cannot learn. The objective is to build functions like the green line, with the ability to generalize on new data points.
Figure inspired by Nicoguaro - Own work, CC BY 4.0,
https://commons.wikimedia.org/w/index.php?curid=46259145

Let us consider a set of parameters ***w*** (weights) that correctly classifies the examples, thus $L_i = 0$. The problem is that this solution is not unique and there might be many sets of ***w*** that classify the examples correctly and have zero loss function (e.g. all $\lambda w, \quad \lambda > 1$ will have this ability). We need a process to regulate the loss function from undesirable weight explosions. This process is the addition of a regularization penalty $R(w)$ to the loss function. The regularization penalty is a function of ***w*** only.

The most common regularization is ***L2* regularization** or weight decay (Equation 3.10). The L2 regularization will force the parameters to be relatively small. ***λ*** is a regularization hyperparameter (that needs to be tuned) which determines how much to penalizes the weights.

$$\lambda R(w) = \lambda \sum_i \sum_j w_{i,j}^2 \tag{3.10}$$

Total Loss Function:
The total loss function consists of the combination of the data loss (Equation 3.1) and the regularization loss (Equation 3.10):

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i\left(\hat{y}_i, y_i\right) + \lambda R(w) \tag{3.11}$$

Where ***N***: number of training examples and ***λ***: regularization strength.
We can visualize the diagram of the data flow of a supervised learning algorithm in Figure 4.2.

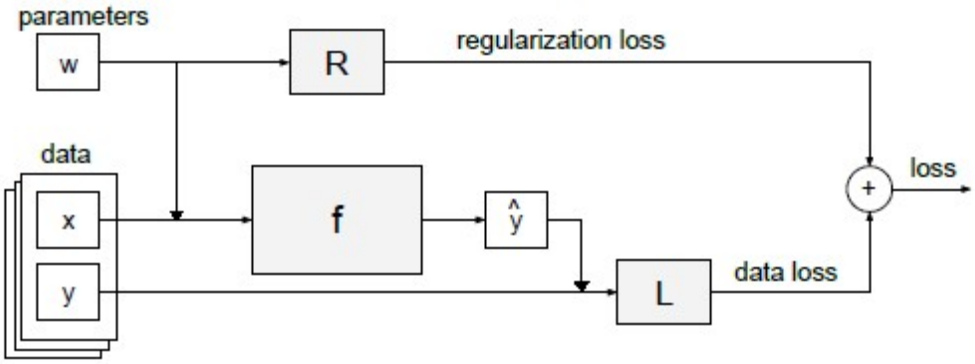


Figure 4.2: Data flow diagram of a supervised learning algorithm where ***x*** are the inputs and ***y*** the labels associated with these inputs. There are three basic functions we use in our supervised learning algorithm: 1) The score function ***f*** that maps examples ***x*** to predicted labels $\hat{y}$, 2) the data loss function $L\left(\hat{y}_i, y_i\right)$ that quantifies the mismatch between the predicted labels and the ground truth labels and 3) the regularization function $R(w)$ that evaluates the complexity of the mapping. These functions make up the overall equation of the total loss $L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(w)$.
Retrieved from the dissertation of Andrej Karpathy (https://purl.stanford.edu/wf528qt3314), licensed under a Creative Commons Attribution-Noncommercial 3.0 Unported License.

4.3 Advanced Regularization Techniques for Deep Learning

In Section 4.2, we described the basic concepts of the total loss function, including the score function, the data loss function and the regularization function. Our objective is to make an algorithm that will perform well on new input data points, not just on the training data. We need to control the capacity of the model and ensure our model generalizes well.

We need to be concerned with overfitting and underfitting. **Overfitting** (Figure 4.3) occurs when the model fails to generalize; thus, it models too well the training dataset. When the model

is overfitting, the gap between the training loss and the validation loss is too large (similar behavior will also exist between train accuracy and validation accuracy). We can **control overfitting** by decreasing the capacity of the algorithm (remove layers of the model) and apply regularization techniques.

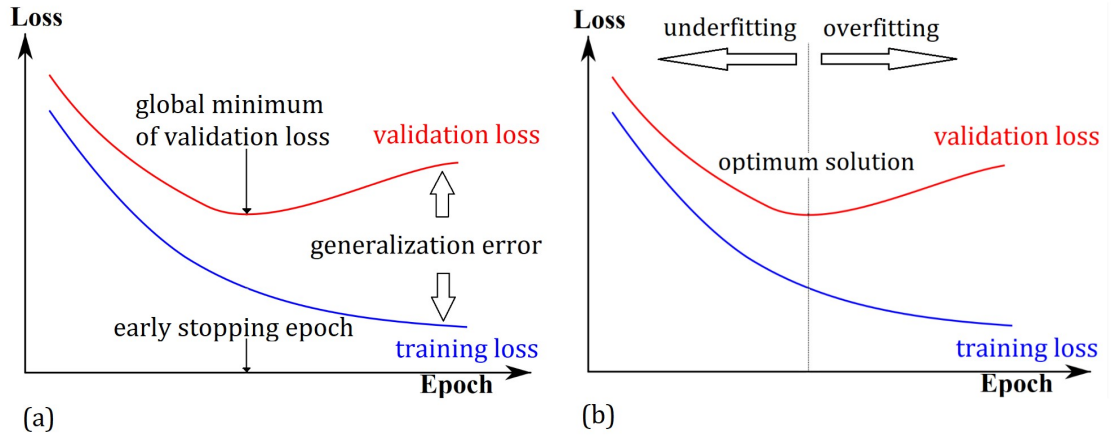


Figure 4.3: We evaluate how the model generalizes by comparison of the training and validation curves during the training procedure. The best model would be where the validation loss curve has its global minimum (**image a**). This optimal solution separates the curves in the overfitting and underfitting area (**image b**).

Underfitting occurs when the model fails to obtain a low error value; thus, the model is not learning. Another indicator for underfitting is when the validation accuracy tracks the training accuracy reasonably well (Figure 4.4). We can **control underfitting** by increasing the capacity of the algorithm (use a deeper model). We want to ensure our algorithm reduces the loss and increases the accuracy while ensuring that the gap between training and validation loss and accuracy, respectively, is relatively small.

In this section, we describe regularization techniques for deep learning models such as Data Augmentation, dropout and early stopping.

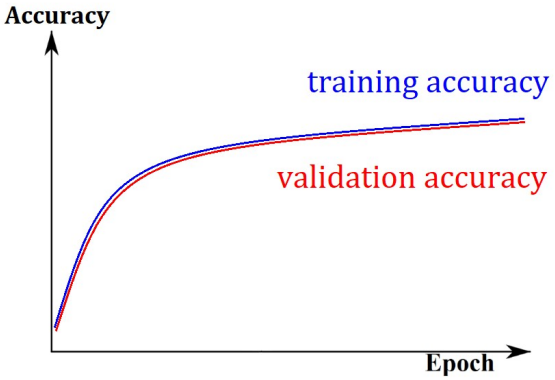


Figure 4.4: Signs of overfitting at accuracy curve. No gap between training and validation accuracy curves indicates underfitting. We have to increase the capacity of the model and keep the gap between training and validation accuracy relatively small.

Data Augmentation:
Data Augmentation is an effective technique, especially for the object recognition classification problem. The amount of input data we have for our classification problem is limited. The basic idea of Data Augmentation is that it generates new training fake data points and adds them to the training dataset. This technique applies simple geometric transformations such as rotations, zooms, scale changes and flips to the data points and creates new data points. Let us consider the input of our model as a pair $(x, y) \in X \times Y$ where ***x*** is the input images and ***y*** the ground truth labels associated with these inputs. Data Augmentation creates new pairs by transforming the ***x*** input without changing its label ***y***. However, each augmented image is considered as a new training data point for the algorithm.

Dropout:
Dropout, proposed initially by Srivastava et al. [113], is a technique for addressing the problem of overfitting in NNs. Dropout layers, with probability ***p***, randomly drop units (hidden and visible) along with their connection during the training of the model. This method prevents overfitting and provides a way of combining many NN architectures. By dropping a unit, we mean temporarily removing it (with its incoming and outcoming connection) from the model, as shown in Figure 4.5. The dropping of the connection ensures that no single node is responsible for activating in a given pattern.

By using dropout, we randomly alter the architecture of the NN by producing a thinner network from the nodes that survived dropout. After the forward and backward pass (Section 4.5), the dropped connections are re-connected, and the algorithm samples another set.

The hyperparameter we want to tune is ***p*****,** which is the probability of retaining a unit in the network. Typical values of ***p*** for hidden units are in the range of 0.5 to 0.8 [113].

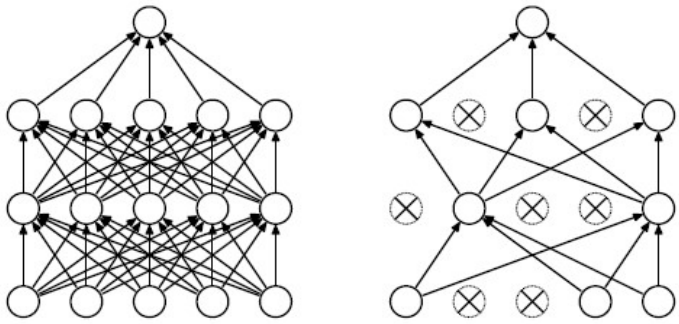


Figure 4.5: A simple three-layer Neural Network (**image a**). After dropout, all crossed nodes have been dropped, producing a thinner network (**image b**).
Retrieved from the paper of Srivastava et al.
(http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf)

Early Stopping:
One of the most effective and simple regularization strategies in deep learning is early stopping [83]. Validation-based early stopping [114] is the strategy we use by selecting a stopping criterion of the learning procedure. We evaluate how the model generalizes by comparison of

the training and validation curves during the training procedure. If there are signs of overfitting either on loss or on accuracy curves (Figure 4.3 a), we stop the training procedure.

We update the parameters with an optimized set or return to the parameter setting at the point in time with the lowest validation set error or highest validation accuracy respectively, and continue the training.

4.4 Optimization

The objective of the optimization process is to find an efficient way to minimize the loss function (Equation 3.11). We denote the value that minimizes a function with the superscript *. Thus, the problem of optimizing our loss function takes the form of $f^* = \arg\min L$.

Optimization algorithms use training examples to optimize the loss function. If the model is using the entire set for a single pass, it is called a batch method. In our case, we use subsets with more than one but fewer than all the training examples for the pass; thus, we use minibatches. We typically use the term **batch size** to describe the size of the **minibatch**. The size of the minibatch is a hyperparameter typically set to 32, 64, 128 or other powers of 2.

For this thesis, we only use **first-order optimization methods** such as SGD, along with advanced first-order optimization methods such as Momentum, Nesterov Acceleration and ADAM.

Gradient Descent:
The basic technique used to minimize a function ***f*** is called gradient descent introduced by Cauchy in 1847 [115]**.** The derivative $f'(x)$ or $\frac{dy}{dx}$ of a function gives the slope of $f(x)$ at point ***x***. The slope of $f(x)$ tells us how to change ***x*** in order to improve ***y***. Let us define $\nabla_x f(x)$ as the gradient of $f(x)$ with respect to ***x.*** The gradient $\nabla_x f(x)$ is the vector containing all the partial derivatives $\frac{\partial}{\partial x_i} f(x)$, measuring how $f(x)$ changes with respect to x_i. To find a local minimum of a function using gradient descent, we have to take steps proportional to the negative of the gradient of the function at the current point (Figure 4.6). The gradient descent method described above proposes a new point (Equation 3.12), where $-\nabla_x f(x_t)$ is the direction of maximum decrease of ***f*** at the point x_t with respect to ***x*** and $a > 0$ is a positive scalar step size called **learning rate.** Learning rate $a > 0$ is a hyperparameter that we can tune.

$$x_{t+1} = x_t - a\nabla_x f(x_t) \quad (3.12),$$

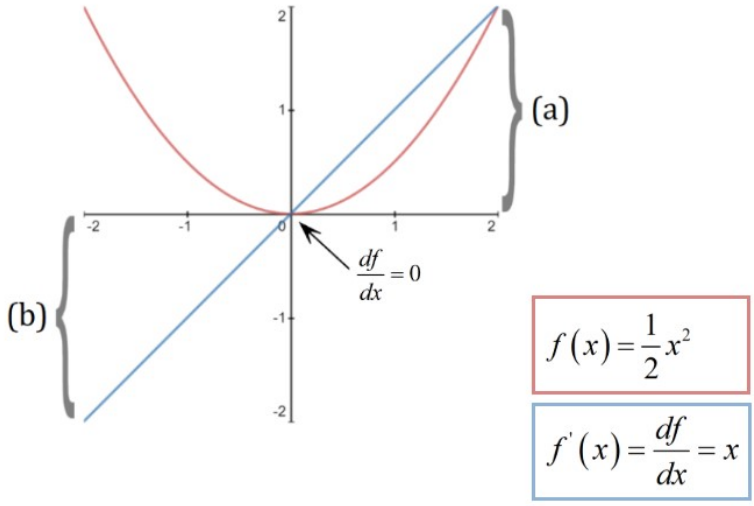


Figure 4.6: An example of how we can use the gradient descent to minimize a function. For (a), if $x\in[0,+\infty)$ then $\frac{df}{dx}>0$, so we can decrease $f(x)$ by moving leftward. For (b), if $x\in(-\infty,0]$ then $\frac{df}{dx}<0$, so we can decrease $f(x)$ by moving rightward.

Stochastic Gradient Descent:
SGD is an extension of the algorithm described above. SGD technically refers to using a single example at a time to evaluate the gradient. In this thesis, we use minibatch SGD since we compute the gradient over **batches** of the training data. The basic algorithm of our SGD method, based on Equation 3.12 is summarized below:

1. Sample a minibatch, from the training data, of ***n*** examples $\{(x_i,y_i)\}_{i=1}^{N}$ along with their labels y_i.

2. Estimate the gradient: $\nabla_w g(w)\approx\nabla_w L(w)=\nabla_w\left[\frac{1}{N}\sum_{i=1}^{N}L_i\left(f(x_i,w),y_i\right)+\lambda R(w)\right]$.

3. Compute the direction that minimizes the function with the negative gradient: $\Delta w=-a\nabla_w g(w)$.

4. Follow the estimated gradient downhill ($\Delta_w=-a\nabla_w g(w)$) and update the parameter $w\leftarrow w+\Delta_w$ via the backpropagation algorithm (Section 4.5).

The SGD update may be written as $w_{t+1}=w_t-a\nabla_w g(w_t)$. For more details about SGD tricks, we can refer to the work of Leon Bottou [116].

While SGD remains a very popular algorithm for optimization, researchers suggest that this learning scheme can be very slow. There are two primary extensions for SGD. The first is momentum [117, 118], and the other is Nesterov momentum [119, 120].

Momentum Update:
The momentum algorithm aggregates an exponentially-decaying average of past gradients and continues to move in their direction (Figure 4.7 a). The algorithm uses a variable ***v*** that plays the role of velocity. The momentum update may be written as:

$$v_{t+1} = \gamma v_t - a\nabla_w g\left(w_t\right)$$
$$w_{t+1} = w_t + v_{t+1}$$

Where $a > 0$ and $\gamma \in \left[0,1\right]$.

The basic algorithm of the SGD method with momentum is summarized below:

1. Sample a minibatch, from the training data, of ***n*** examples $\left\{\left(x_i, y_i\right)\right\}_{i=1}^{N}$ along with their labels y_i.

2. Estimate the gradient: $\nabla_w g\left(w\right) \approx \nabla_w L\left(w\right) = \nabla_w \left[\frac{1}{N}\sum_{i=1}^{N} L_i\left(f\left(x_i, w\right), y_i\right) + \lambda R\left(w\right)\right]$.

3. Compute velocity ***v*** update: $v \leftarrow \gamma v - a\nabla_w g\left(w\right)$.

4. Update our parameter ***w***: $w \leftarrow w + v$.

Momentum term ***γ*** is a hyperparameter called momentum coefficient, usually set to 0.5, 0.9, or 0.99 [83].

Nesterov Acceleration Momentum Update:
Nesterov momentum is a variant of the momentum algorithm inspired by the accelerated gradient method by Nesterov [119]. The Nesterov momentum update (Figure 4.7 b) may be written as:

$$v_{t+1} = \gamma v_t - a\nabla_w g\left(w_t + \gamma v_t\right)$$
$$w_{t+1} = w_t + v_{t+1}$$

The only difference between the momentum and the Nesterov momentum is the update of the velocity vector ***v***. They both apply a gradient-based correction of the velocity vector but while momentum algorithm computes the update from the current position w_t, Nesterov momentum first updates w_t by $w_t + \gamma v_t$; changing ***v*** in a quicker way.

The basic algorithm of the SGD method with Nesterov momentum is summarized below:

1. Sample a minibatch, from the training data, of ***n*** examples $\left\{\left(x_i, y_i\right)\right\}_{i=1}^{N}$ along with their labels y_i.

2. Partial update of our parameter ***w***: $w \leftarrow w + \gamma v$.

3. Estimate the gradient: $\nabla_w g(w) \approx \nabla_w L(w) = \nabla_w \left[\frac{1}{N} \sum_{i=1}^{N} L_i \left(f(x_i, w+\gamma v), y_i \right) + \lambda R(w) \right]$.

4. Compute velocity ***v*** update: $v \leftarrow \gamma v - a \nabla_w g(w)$.

5. Update our parameter ***w***: $w \leftarrow w + v$.

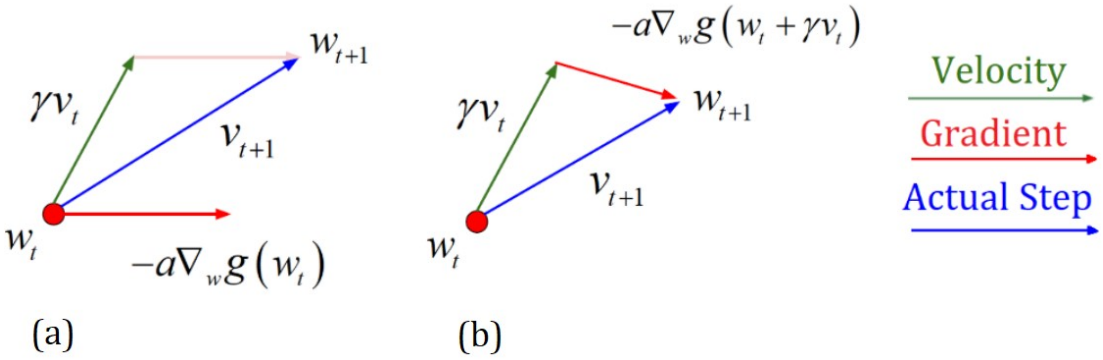


Figure 4.7: A Momentum update (**image a**) in comparison with a Nesterov acceleration momentum update (**image b**).

Adam:

Adam algorithm, proposed by Kingma and Ba [121] is a gradient-based optimization method. The momentum is embedded in the algorithm along with bias correction for the mean and the uncentered variance of the moving averages.

The basic algorithm of the Adam method is summarized below:

1. Sample a minibatch, from the training data, of ***n*** examples $\{(x_i, y_i)\}_{i=1}^{N}$ along with their labels y_i.

2. Estimate the gradient: $\nabla_w g(w) \approx \nabla_w L(w) = \nabla_w \left[\frac{1}{N} \sum_{i=1}^{N} L_i \left(f(x_i, w), y_i \right) + \lambda R(w) \right]$.

3. Update timestep: $t \leftarrow t+1$.

4. Update biased first moment estimate: $c \leftarrow \beta_1 c + (1-\beta_1) \nabla_w g(w)$.

5. Update biased second moment estimate: $k \leftarrow \beta_2 k + (1-\beta_2) \left(\nabla_w g(w) \odot \nabla_w g(w) \right)$.

6. Correct first moment bias: $\hat{c} \leftarrow c \left(1-\beta_1^t\right)^{-1}$.

7. Correct second moment bias: $\hat{k} \leftarrow k \left(1-\beta_2^t\right)^{-1}$.

8. Compute the update: $\Delta w = -a \hat{c} \left(\sqrt{\hat{k}} + \varepsilon \right)^{-1}$.

9. Update our parameter ***w***: $w \leftarrow w + \Delta w$.

All operations on vectors are element-wise. The hyperparameters $\beta_1, \beta_2 \in [0,1)$, control the exponential decay rates of the moving averages of ***c*** and ***k***. $a > 0$ is the positive scalar step size called **learning rate** and ε is a very small number to avoid dividing by zero in the gradient update. Good default setting for the hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\varepsilon = 10^{-8}$ [121].

4.5 Backpropagation Algorithm

The main objective of the backpropagation algorithm in NN applications is to find how the input parameters influence the loss function, and in what way we shall change them in order to reduce the loss.

The backpropagation algorithm [84] is the method we use to efficiently compute gradients of our functions with respect to their inputs by using the chain rule of Calculus. The backpropagation algorithm consists of two phases, the forward propagation and the backward propagation.

In the **forward propagation** (forward pass), the information flows forward through the NN. The network accepts an input ***x*** and produces an output $\hat{y}$. In our case, ***x*** is the pixel vector of an image and $\hat{y}$ the predicted label for this image. The information provided by ***x*** propagates to the layers of the NN and finally produces the predicted label $\hat{y}$. The final product of the forward pass is the scalar-valued loss function ***L***.

In order to perform the parameter update of our functions and continue the learning process, we have to compute the gradient $\nabla_w L$ with respect to the parameters ***w*** and update the parameters by applying the chain rule of Calculus. This process takes place during the **backward propagation** (backward pass).

Since we use a scalar-valued loss function, we have many values as an input and a single output. Straightforwardly computing the gradient could be computationally expensive. It is more computationally efficient to use reverse-mode differentiation by going back to front and, thus using the backward pass for gradient computation.

The backpropagation algorithm is responsible only for computing the gradient and update the parameters of our functions. The actual learning process of our algorithm takes place by the use of the optimized algorithms, as described in Section 4.4.

Chain Rule of Calculus:

The chain rule of Calculus is used to compute the derivatives of a composite function. The order by which the backpropagation algorithm computes the chain rule proves to be highly efficient [83].

Let us suppose we have a vector u_0 that we want to transform into a scalar u_k through a series of functions $u_i = g_i(u_{i-1}), \quad i = 1,2,...,k$. We assume that the nodes are arranged in a way we can compute their output one after the other (Figure 4.8).

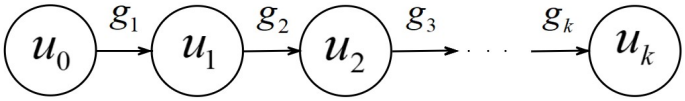


Figure 4.8: A simple computational graph. Every node is arranged in a way that we can compute their output one after the other.

The Jacobian matrix of all first-order partial derivatives of our vector-valued function tells us how each u_i depends on the changes of u_{i-1}. Finally, with the use of the product operator of the Jacobian matrixes $\frac{\partial u_k}{\partial u_0} = \prod_{i=1}^{k} \frac{\partial u_i}{\partial u_{i-1}}$, we can compute the gradient we are interested in. Backpropagation algorithm performs such Jacobian products, using the chain rule, for each operation in the computational graph.

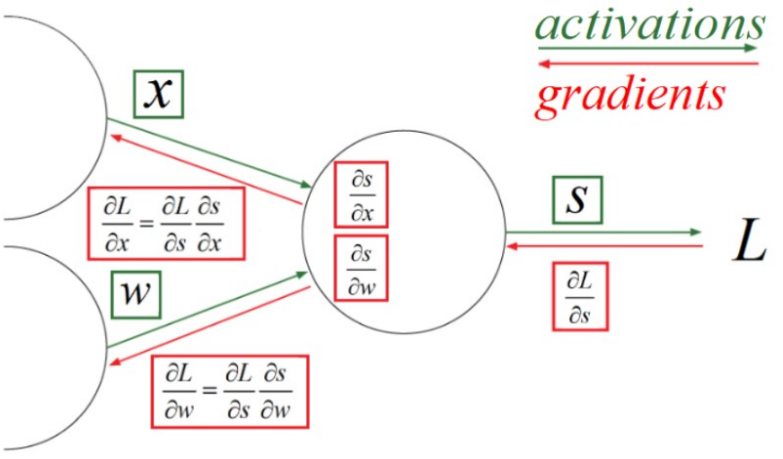


Figure 4.9: Backpropagation along a simple computational graph. We have to know what function ***s*** is computing on the forward propagation. We assume that ***s*** is computed by a fixed function $s = x \odot w$. The value of ***s*** continues into the graph until the total loss ***L*** is produced. All the intermediate functions are fixed. The backward propagation proceeds in the opposite direction. The objective is to know how ***x*** and ***w*** influence ***L.*** We can compute all the Jacobian matrixes $\frac{\partial L}{\partial s}, \frac{\partial s}{\partial x}, \frac{\partial s}{\partial w}$ that tell us what is the influence of ***s*** to ***L, x*** to ***s*** and ***w*** to ***s*** respectively. We continue travelling backwards through the functions, multiplying by Jacobians until the inputs are reached.

4.6 Neural Networks

In the previous section of this chapter, we mentioned a score function ***f*** that transforms the input vector ***x*** into a prediction $\hat{y}$ and optimizes this differentiable function with respect to any loss function. In this section, we will make a more extensive reference to this function***.***

Feedforward networks have introduced the notion of hidden layers (Figure 4.10 a). In order to compute the hidden layer value, we have to choose an **activation function**. An activation function derives the output of a neuron given a set of inputs and represents the firing rate of a neuron.

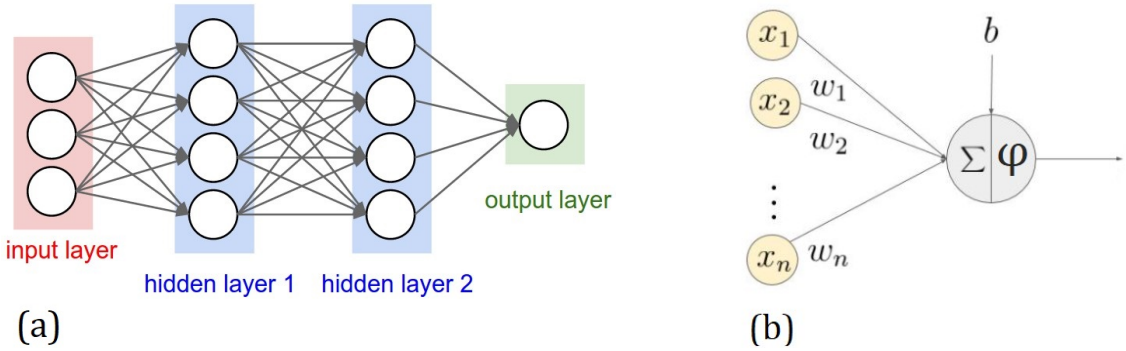


Figure 4.10: (**image a**): A two-hidden Fully Connected layer Neural Network. The network is fully pairwise connected between two adjacent layers. Neurons between a single layer are not connected to each other. If W_1, W_2, W_3 the parameter matrixes and φ the non-linearity element-wise function, this network will have the form of $f(x) = W_3\varphi(W_2\varphi(W_1 x))$. (**image b**): To pass a signal through a node, we compute the sum of the weighted inputs from previous nodes; we add a bias vector, fire the signal after the activation function and propagate the output to the next layer.

Let us consider a simple linear classifier with a **bias** vector embedded into the design matrix. Thus, this linear function takes the form $f(x_i; w, b) = wx_i + b$. The bias vector ***b*** does not interact with the actual data but influences the output of this function. In the absence of any input, the output of the transformation is biased to being ***b***.

To convert an input signal of a node into an output signal, we have to compute the weighted sum of each connection pointing to this specific neuron (Figure 4.11 b). We then pass that sum to a non-linear activation function that transforms it into a number between some limits.

The activation function has to be **differentiable** to perform the backpropagation algorithm through the chain rule, as described in Section 4.5. The activation function has to be **non-linear** to provide a complex solution and be able to model any data by adding curvature. A linear equation is easy to solve but is limited in its complexity with less power to learn complex functional mappings from data.

Every non-linear activation function performs a specific fixed mathematical operation. There are several activation functions we may encounter in practice. For this thesis, we use two activation functions, the ReLU activation function and the ELU activation function.

Rectified Linear Unit (ReLU):

The Rectified Linear Unit [122], depicted in Figure 4.11, is defined by the function (Equation 3.13):

$$\varphi(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{3.13}$$

Or equivalently:

$$\varphi(x) = \max\{0, x\} \tag{3.14}$$

This function transforms the input to the max of either 0 or the input itself. The more positive the neuron, the more activated it is. The derivative of these functions takes the form:

$$\varphi'(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases} \tag{3.15}$$

One problem with this approach is the dying ReLU problem. For activations $x \in (-\infty, 0]$, the gradient will be zero (Equation 3.15), and neurons will stop responding to variations of the loss function in the backpropagation procedure.

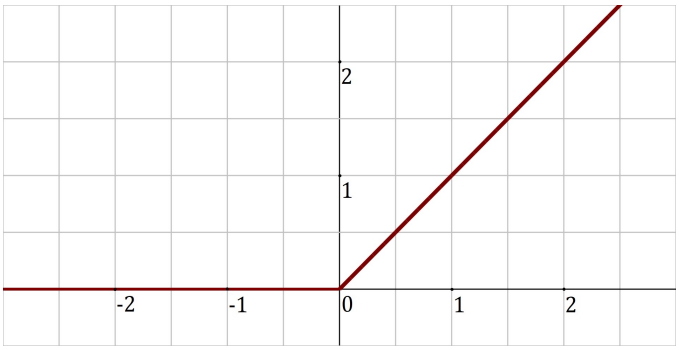


Figure 4.11: The Rectified Linear Unit (ReLU) activation function.

Exponential Linear Unit (ELU):

The Exponential Linear Unit [123], depicted in Figure 4.12, is defined by the function (Equation 3.16):

$$\varphi(x) = \begin{cases} \alpha(\exp(x) - 1), & x < 0 \\ x, & x \geq 0 \end{cases} \tag{3.16}$$

The hyperparameter $\alpha > 0$ controls the value to which an ELU saturate for negative net inputs. The derivative of Equation 3.16 takes the form:

$$\varphi'(x) = \begin{cases} \varphi(x) + \alpha, & x < 0 \\ 1, & x \geq 0 \end{cases} \tag{3.17}$$

Exponential Linear Unit function has negative values; thus, it can produce negative outputs, allowing them to push activations closer to zero.

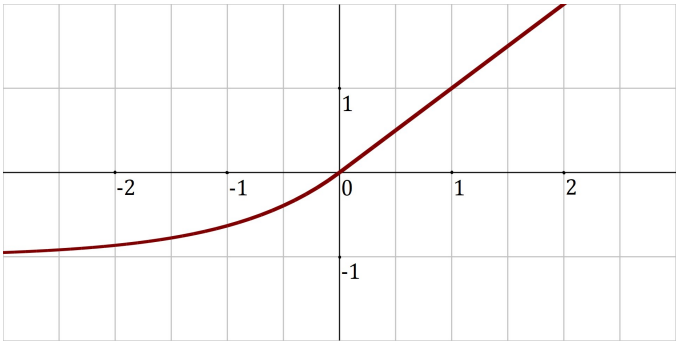


Figure 4.12: The Exponential Linear Unit (ELU) activation function.

4.7 Parameter Initialization

In this section, we will discuss **weight initialization** along with the effect of this initialization to the training process of our model. We will then mention a technique for parameter normalization, called **Batch Normalization**.

Let us suppose that the values of our weights are random numbers distributed with a standard Normal (Gaussian) distribution (with mean $\mu = 0$ and standard deviation $\sigma = 1$). If we focus on the inputs of a specific node in the hidden layer (Figure 4.10 b), we presume that the input of this hidden neuron is the weighted sum:

$$z_j = \sum_i w_{ij} x_i + b_j \tag{3.18}$$

Thus, ***z*** is itself distributed with a Normal distribution with mean $\mu = 0$. The problem in this procedure is with the prices of the variance and standard deviation because they grow with the number of inputs. The value of the standard deviation in the input of the neuron is the sum of the values of the standard deviation of each weight pointing at this node, along with the bias:

$$\sigma_z = \sqrt{\sum_i \sigma_{w_i} + b_j} \tag{3.19}$$

That is, ***z*** has a broader Normal distribution not sharply peaked (Figure 4.13). That leads to saturated neurons, which means that every small change in the value of the weights will have a minuscule change in the activation of the neuron. This change will negatively affect the ability of the NN to learn through the backpropagation algorithm.

To avoid learning slowdown along with saturated neurons, we have to choose another way to initialize the parameters. We have to squash the Gaussians down, making the neurons less likely to saturate. The proposed methods initialize the weights as random Normal distributed variables with mean $\mu = 0$ and alter the standard deviation.

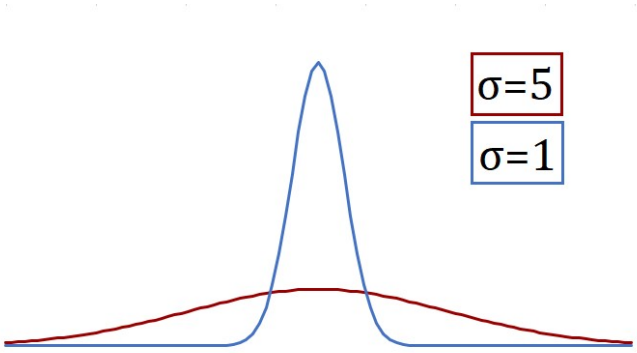


Figure 4.13: An example of two Normally distributed curves with mean $\mu = 0$. The distribution with the more significant standard deviation has a much broader curve not sharply peaked and squashes down all the prices.

Xavier Glorot Normal Initialization:

In 2010, Xavier Glorot and Yoshua Bengio [124] proposed a new initialization scheme. Let us we define as F_{in} and F_{out} as the number of the input weights along with the number of the output weights of a node. The Glorot normal distribution method draws samples from a Normal distribution centered on 0 with standard deviation:

$$\sigma = \sqrt{\frac{2}{F_{in} + F_{out}}}$$

He et al. (MSRA) Normal Initialization:
In 2015, He et al. [125] proposed a similar method. The He et al. normal distribution method draws samples from a Normal distribution centered on 0 with standard deviation:

$$\sigma = \sqrt{\frac{2}{F_{in}}}$$

Batch Normalization:
Another method that makes normalization a part of the model architecture, performing for every minibatch, is Batch Normalization. This recently developed technique [126], allow us to be less careful about initialization and can be interpreted as doing preprocessing at every layer of the network, in a differentiable manner. It maintains the mean activation close to 0 and the activation standard deviation close to 1, protecting the weights from becoming imbalanced (extraordinarily high or low prices).

The basic algorithm of the Batch normalization method, for a minibatch of activations, is summarized below:

1. Sample a minibatch of activations $H = \{x_i\}_{i=1}^{N}$.

2. Compute the mean of the given minibatch: $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$.

3. Compute the standard deviation of the given minibatch: $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$.

4. Normalize the output from activation function: $H' = \frac{H - \mu}{\sqrt{\sigma^2 + \varepsilon}}$. ε is a very small number to avoid dividing by zero.

5. Set a new standard deviation and mean for the data: $y_i = \gamma x_i' + \beta$. The β and the γ, are called beta and gamma weights hyperparameters.

At test time, we can replace ***μ*** and ***σ*** by running averages collected during the training time. This technique allows the model to be evaluated without being biased by ***μ*** and ***σ*** from the minibatch that passed through the network during training time.

4.8 Discussion

In the previous sections, we described how the information propagates inside a NN designed for machine learning. The basic idea is summarized below:

- Vectorize an input data (x, y).

- Feed the input data to the network. The input data propagates with matrix operations layer by layer. The input of every node of the hidden layers is a weighted sum $z_j = \sum_i w_{ij} x_i + b_j$.
- Apply an activation function $\varphi(z_j)$ to the previous result and pass it to the next layer to make the same procedure.
- The final product of this procedure is an output value depicting a prediction $\hat{y}_i$. This prediction is a combination of a linear function of the input ***x*** with a non-linear function (through the activations) of the weights w_i.
- Quantify the mismatch between the predicted labels and the ground truth labels via the loss function $L_i(\hat{y}_i, y_i)$.
- Use that error value to compute all the Jacobian matrixes of the partial derivatives with respect to the weights during the backward pass.
- Update the weights with their new values
- Repeat the procedure until the error is minimized as much as possible.

4.9 Convolutional Neural Networks

Convolutional Neural Networks [2] are specialized networks for processing data with a grid-like topology and can scale such models to considerable size. Convolutional Neural Networks are analogous to traditional Neural since they are made up of neurons with learnable parameters. These neurons receive input data, perform an operation and pass it to the next layers. From the input image vector to the output predicted score, the network will still express a single differentiable scalar-valued score function. We still quantify the mismatch between the predicted labels and the ground truth labels via the loss function. Since CNNs are primarily designed for image-based pattern recognition, there are specific properties encodes to their architecture that reduces the overall parameters.

As the name indicates, these types of networks employ a linear operation called **convolution**. Let us suppose we have a weighting function ***w*** (or a kernel**)** and an input function **x** which are defined only on integer ***t***. We define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{\tau=-\infty}^{+\infty} x(\tau) w(t-\tau) \tag{3.20}$$

That means that we:

- Reflect the weighting function $w(\tau) \rightarrow w(-\tau)$.
- Add a time-offset ***t***, which allows $w(t-\tau)$ to slide along $\tau - axis$.
- Starting ***t*** at $-\infty$ to $+\infty$, we slide the kernel. Wherever the two functions interact we compute a weighted sum of $x(\tau)$ with the weighting function $w(-\tau)$.

In practice, we implement that summation over a finite number of elements.

We can generalize this approach over a 2-D input along with a 2-D kernel with a finite number of elements. If ***I*** is a 2-D image and ***K*** a 2-D kernel then:

$$S(i,j) = \sum_p \sum_q I(i-p, j-q) K(p,q) \tag{3.21}$$

In Equation 3.21, we have flipped the kernel relative to the input. That means if we increase ***p,*** the index into the kernel increases but the index into the input decreases. Instead of this approach, many deep learning libraries implement a **cross-correlation** function:

$$S(i,j) = \sum_p \sum_q I(i+p, j+q) K(p,q),$$

that gives similar results with (3.21). In either way, convolution is the sum of element-wise multiplication between the kernel areas that the kernel covers by sliding across the input data.

CNN Architecture:
The neurons inside the CNN are tensors arranged in three dimensions, the **width**, the **height** and the **depth**. Depth refers to the number of channels in the image (an RGB image will have three color channels). In contrast with the Fully Connected layers, the neurons inside a given CNN layer are connected to a small region of the previous layer. There are three main types of layers in a CNN, the **Convolutional layer**, the **Pooling layer** and the **Fully Connected layer**.

4.9.1 Convolutional Layer

The Convolutional layer plays a vital role in the operation of a CNN since it is the core building block of a CNN. It consists of convolving filters acting as learnable **kernels**. Every one of these filters spreads through the full depth of the input tensor. Each kernel slides across the spatial dimensionality of the input tensor, producing scalar product between the parameters of the filter and the input volume. These products produce a 2-D activation map during the forward pass. The network will learn the kernels that fire when they see a specific type of feature in the given image. Every one of our learnable kernels will have a corresponding activation map. These maps are stacked along the full depth of the input tensor to produce the output volume of the Convolutional layer.

Local Receptive Fields:
In Convolutional Neural Networks, we do not connect every input pixel to a neuron. Instead, we only make connections in a small, localized region of the input volume called the **receptive field** (or filter size). These fields share local connections across their spatial dimensionality but always full along the depth on the input tensor. The neurons learn the weights from each one of these connections along with an overall bias for the receptive field. We then slide the receptive field across the entire input volume.

Output Volume:
To understand the output volume of the Convolutional layer, we have to talk about the spatial arrangement inside the Convolutional layer along with the three hyperparameters that control it. The output is optimized through the hyperparameters called **depth**, **stride** and **zero-padding** respectively.

The depth of the output volume controls the number of filters we use to connect to a local region of the input.

To control the output spatially, we must specify the stride (***S***) by which we slide the kernel and the amount of zero-padding (***P***) of this kernel. When we set $S = 1$, we move the filters from left-to-right and top-to-down one pixel at a time. When we set $S = 2$ $S = 1$, we move the filters two pixels at a time. Only by changing the stride, we can reduce the spatial dimensions of the input volume. In order to preserve the spatial dimensions and ensure that the input volume and output volume of the layer will have the same size spatially, we pad the input along the borders

with zeros. Without the padding, the spatial dimension of the input would decrease too quickly, causing problems to the training procedure.

If we would like to visualize the convolution of a kernel over an input tensor, let us think of an input $48\times48\times3$ image (48 width, 48 height, 3 depth) and a $3\times3\times3$ kernel (3 width, 3 height, 3 depth) as we can see in Figure 4.14.

A Convolution layer requires four hyperparameters -the filter size F, the stride S, the number of filters K and the zero-padding P.

A Convolution layer:

- Accepts an input volume $W_{in}\times H_{in}\times D_{in}$
- Produces an output volume $W_{out}\times H_{out}\times D_{out}$ where:
  - $W_{out}=\frac{(W_{in}-F+2P)}{S}+1$
  - $H_{out}=\frac{(H_{in}-F+2P)}{S}+1$
  - $D_{out}=K$

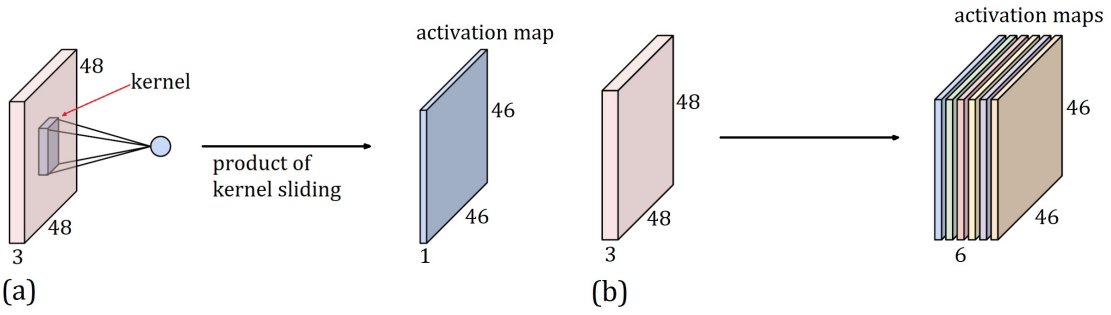


Figure 4.14: Visualization of a convolving $3\times3\times3$ kernel over a $48\times48\times3$ tensor with stride $S=1$ and zero-padding $P=0$. The filter spreads through the full depth of the input tensor since depth=3. There are 46×46 unique positions for the 3×3 filter in this input. Thus, the convolution products an 46×46 activation map (**image a**). If we suppose that the Convolutional layer has a set of 6 different filters, each applied in the same way, we have 6 activation maps. The activation maps are stacked together to produce the $46\times46\times6$ output of the layer. This $46\times46\times6$ new layer, proceeds as an input for the next operation (**image b**).

Parameter Sharing:

If we consider the example above, we see that there are $46\times46\times6=12,696$ neurons in this Convolutional layer, and each one of these neurons has $3\times3\times3=27$ weights along with one bias. If we compute the total parameters, we have $12,696\cdot(27+1)=355,488$ parameters on the first Convolutional layer. This number is high if we consider that this is an example with small numbers.

The Convolutional Neural Networks are designed to reduce the overall parameters with **parameter sharing**. This scheme works with the assumption that if a feature region is useful to compute in a spatial region, then it is likely to be useful in another spatial region. In every

different 2-D activation map, the model constrains the neuron to use the same weights and bias. In the given example, with this parameter sharing scheme, this Convolutional layer would have only six sets of weights (each one for every **slice** of the stacked activation map) along with their biases, for a total $(3\times3\times3)\times6=162$ unique set of weights and $162+6=168$ overall parameters.

With the parameter sharing, the Convolutional layer produces $F\times F\times D_{in}$ weights per filter, for a total $(F\times F\times D_{in})\times K$ set of weights along with K biases. Thus, it produces $(F\times F\times D_{in})\times K+K$ parameters.

Backpropagation:
With the parameter sharing assumption, during the backpropagation, each neuron computes the total gradient of its weights. These gradients are summed up per slice, thus, updating a single set of weights in the backward pass.

4.9.2 Pooling Layer

The purpose of the Pooling layer is to reduce the spatial dimensionality of the representation, reduce the number of parameters and the computational complexity; thus, control overfitting. It implements a fixed function without the need for any parameter. This downsampling technique operates in any activation map.

A Pooling algorithm requires two hyperparameters -the filter size F by which it downscales (vertical and horizontal) the representation and the stride S.

A Pooling operator:

- Accepts an input volume $W_{in}\times H_{in}\times D_{in}$
- Produces an output volume $W_{out}\times H_{out}\times D_{out}$ where:
  - $W_{out}=\frac{(W_{in}-F)}{S}+1$
  - $H_{out}=\frac{(H_{in}-F)}{S}+1$
  - $D_{out}=D_{in}$

In this thesis, we use the **Max** Pooling operation [127]. Max Pooling is applying a max filter over the representation and creates a new output matrix where each element is the maximum of a region in the original input determined by the filter size and the stride.

During the forward pass in the backpropagation algorithm, the model keeps track of the index of the max activation. In the backward pass, the algorithm uses that index to rout the gradient to this input with the highest value.

4.9.3 Fully Connected Layer

The Fully Connected layers are analogous to the neurons arranged in a traditional NN with full connections to all activations in the previous layer. The neurons are directly connected to the neurons of the two adjacent layers, but the neurons between a single layer are not connected to each other.

4.9.4 VGG Case Study

Initially proposed by Karen Simonyan and Andrew Zisserman [1] the VGG type of CNN uses only a small (3×3) Convolutional filter along with a (2×2) Pooling layers across the whole network. Their main contribution was in showing that the depth of the network is one of the critical components regarding a good overall performance.

4.10 Summary

In this chapter, we described the workflow for applying an image classification task in an end-to-end way using Neural Networks. The necessary ingredients we need to put together in our deep learning algorithm are the following:

Dataset Preprocessing:
The dataset itself, along with the task we are trying to solve; define our goals. Each data point in our dataset is a $(x, y) \in X \times Y$ where ***x*** is a vectorized input and ***y*** a label associated with this input. We split our dataset into three folds, the training, validation and testing fold. We use the training dataset for parameter optimization, the validation dataset for hyperparameter optimization and the testing dataset for the final evaluation of the model. In terms of **data preprocessing**, conventional techniques include data resizing, data normalization and data standardizing.

Model Architecture:
The architecture of our model is a critical point for the performance of the algorithm. Since our data consists of images, we process the pixel data with convolutions. The Convolutional Neural Networks are made up of different type stacked layers. Some of these layers perform a fixed mathematical operation and do not need parameters (Activation layer, Pooling layer) in contrast with other types that contain parameters (Convolutional layer, Fully Connected layer). Each layer may or may not have additional hyperparameters (Activation layers do not use hyperparameters).

Optimization Method:
We have to define an optimization method for the optimization of our parameters through the backpropagation algorithm. Since we use advanced first-order optimization methods, we must decide how to tune the other hyperparameters along with the number of epochs and the batch size we will use.

Evaluation:
With the help of the validation dataset, we define the best model for the task we study. The final step is to evaluate the model. We evaluate the model with a single pass of the testing dataset.

5
Experimentation

In this chapter, we first describe the generic architecture of our CNN model (Section 5.1). In Section 5.2, we analyze the details of our classification framework, followed by the experimental procedure in Section 5.3.

5.1 Architecture

During the training process, the input of the VGG-like [1] type of CNN [2] is a fixed-size 48x48 grayscale image. The data have been prepared and preprocessed by the organizers of the ICML 2013 Workshop on Challenges in Representation Learning [103]. The ICML 2013 organizers pre-defined a split into three folds, training (80% of the total images), validation (10% of the total images) and test (10% of the total images). In terms of data preprocessing, the images are grayscaled and resized into 48x48 pixel images.

Every image is passed through a stack of Convolutional layers with a fixed receptive field and a fixed stride of 1. The spatial dimensionality is preserved after each convolution with the use of a zero-padding set to 1 for the Convolutional layers. Reduce of spatial dimensionality is achieved through the max-pooling layers. Max pooling is performed over a filter with a stride of two (2).

The hidden layers are equipped with either the ELU [123] or the ReLU [122] non-linearities. We used a Batch normalization layer [126] before or after the non-linearity, depending on the experiment. Furthermore, we add a Dropout layer [113] after each Convolutional layer. The multi-dimensional volume of the stack of Convolutional blocks is flattened into a 1-D array, and the Fully Connected layers follow the stack. The configuration of the Fully Connected layers is not the same for all the networks. The final layer is a Softmax layer.

Consider we have M Convolutional blocks of N Convolutional layers stacked together, followed by L Fully Connected layers. We can derive the most common Convolutional block architectures, of this thesis, using the following patterns:

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow ACT \Rightarrow BN\right)\times N\right] \Rightarrow POOL \Rightarrow DO\right\}\times M \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow ACT \Rightarrow BN \Rightarrow DO\right)\times L \Rightarrow Last\ FC \Rightarrow SOFTMAX,$$

or

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow BN \Rightarrow ACT\right)\times N\right] \Rightarrow POOL \Rightarrow DO\right\}\times M \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow BN \Rightarrow ACT \Rightarrow DO\right)\times L \Rightarrow Last\ FC \Rightarrow SOFTMAX,$$

where $CONV \equiv$ Convolutional layer, $ACT \equiv$ Activation layer, $BN \equiv$ Batch Normalization layer, $POOL \equiv$ Max pooling layer, $DO \equiv$ Dropout layer and $FC \equiv$ Fully Connected layer.

The CNN architectures evaluated in this thesis are outlined in Tables 5.1 and 5.2. All the architectures follow the design presented above and differ in the dimension of depth and the configuration of some of their layers, as we will describe in Section 5.3. In the following, we will refer to the CNNs by their names (A-F). For each one of the different configurations, we report the number of trainable parameters in Table 5.3.

Table 5.1: Types A-D of Different VGG-like Convolutional Neural Networks. The depth of the architectures increases from left to right. The Convolutional layers are denoted as *conv2d* followed by the depth of the channels (*conv2d-Depth)*. Fully Connected layers are denoted as *dense* followed by the number of their nodes (*dense- # of nodes)*. The Activation layers, along with the Batch Normalization and Dropout layers are not shown for brevity.

<table>
<tr><th>Type A</th><th>Type B</th><th>Type C</th><th>Type D</th></tr>
<tr><td>7 weight layers</td><td>7 weight layers</td><td>9 weight layers</td><td>10 weight layers</td></tr>
<tr><td rowspan="2">conv2d - 32</td><td rowspan="2">conv2d - 64</td><td>conv2d - 32</td><td>conv2d - 32</td></tr>
<tr><td>conv2d - 32</td><td>conv2d - 32</td></tr>
<tr><td colspan="4">max_pooling2d</td></tr>
<tr><td rowspan="2">conv2d - 64</td><td rowspan="2">conv2d - 128</td><td>conv2d - 64</td><td>conv2d - 64</td></tr>
<tr><td>conv2d - 64</td><td>conv2d - 64</td></tr>
<tr><td colspan="4">max_pooling2d</td></tr>
<tr><td>conv2d - 128</td><td>conv2d - 256</td><td>conv2d - 128</td><td>conv2d - 128</td></tr>
<tr><td>conv2d - 128</td><td>conv2d - 256</td><td>conv2d - 128</td><td>conv2d - 128</td></tr>
<tr><td colspan="4">max_pooling2d</td></tr>
<tr><td colspan="3">dense - 64</td><td>dense - 64</td></tr>
<tr><td colspan="3">dense - 64</td><td>dense - 64</td></tr>
<tr><td colspan="3" rowspan="2">dense - 7</td><td>dense - 32</td></tr>
<tr><td>dense - 7</td></tr>
<tr><td colspan="4">Softmax</td></tr>
</table>

Table 5.3: Number of Trainable Parameters (in thousands).

Type of Network	A	B	C	D	E	F
Number of Trainable Parameters	541 K	1,556 K	587 K	589 K	1,326 K	1,917 K

Table 5.2: Types E and F of Different VGG-like Convolutional Neural Networks. The depth of the architectures increases from left to right. The Convolutional layers are denoted as *conv2d* followed by the depth of the channels (*conv2d-Depth*). Fully Connected layers are denoted as *dense* followed by the number of their nodes (*dense- # of nodes*). The Activation layers, along with the Batch Normalization and Dropout layers are not shown for brevity.

Type E	Type F
11 weight layers	12 weight layers
conv2d – 32	conv2d – 32
conv2d – 32	conv2d – 32
max_pooling2d	
conv2d – 64	conv2d – 64
conv2d – 64	conv2d – 64
max_pooling2d	
conv2d – 128	conv2d – 128
conv2d – 128	conv2d – 128
max_pooling2d	
conv2d – 256	conv2d – 256
conv2d – 256	conv2d – 256
	conv2d – 256
max_pooling2d	
dense1 – 64	
dense2 – 64	
dense3 – 7	
Softmax	

5.2 Classification Framework

In this section, we present details regarding the classification framework we used during our training and testing procedure.

5.2.1 Implementation

We implemented this model with Python code on CPU. All of our classifiers were implemented in Keras using TensorFlow backend. The original *fer2013.csv* file converted into HDF5 [142] format so we can train a CNN on top of it. With the use of Keras callbacks, we saved the model's weights in HDF5 format, every five epochs.

5.2.2 Data Augmentation

In order to prevent from overfitting, due to the small amount of training data, we apply Data Augmentation techniques. To increase the number of training samples, we employ Data Augmentation in the form of:

- random rotations with 10° range,
- random zooms with 0.1 range,

- random horizontal flips,
- points outside the boundaries of the created images are filled with the *nearest* mode.

After applying all other transformations, we rescale all images. The original input pixel images are unnormalized, in range $[0,255]$; thus we scale down to range $[0,1]$ with a rescaling factor of $1/255$.

This rescaling factor is also performed in validation and testing dataset.

5.2.3 Training

We train every model for up to 100 epochs with the maximum size of the generator queue set to 10. The loss function is optimized using the Adam update or minibatch SGD with momentum and Nesterov momentum updates. We set the batch size to 32, 64 or 128 depending on the experiment. All weights are initialized with the Xavier Glorot or He (MSRA) normal distribution methods.

5.3 Experiments

In this section, we review the experiments we performed in order to improve the classification accuracy on *FER-2013* dataset.

5.3.1 Effectiveness of Data Augmentation

We aim to test whether Data Augmentation improves the classification accuracy of our model. We start with an SGD optimizer with 128 batch size, learning rate $1e-1$, Nesterov and Momentum update with $\gamma = 0.9$ and decay of 0.001 (Figure 5.3: Bottom Right). We use a Softmax classifier and Cross-entropy Loss- the Data Augmentation techniques applied in this section, described in Section 5.2.2.

We can visualize this architecture in the schematics in Figures 5.1 and 5.2. We use a Type C VGG-like CNNs with architecture using the following pattern:

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow \mathrm{ReLU} \Rightarrow BN\right)\times 2\right] \Rightarrow POOL \Rightarrow DO\right\}\times 3 \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow \mathrm{ReLU} \Rightarrow BN \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX.$$

The results of this experiment are depicted in Figure 5.3. We used a model with all Data Augmentation arguments applied (Red line), along with a model we did not use a rescaling factor (Yellow line). At the end of the 90th epoch, the private set accuracy for the red lined model is 63.39. The corresponding result for the yellow lined model is 62.95. There is a slight superiority of the model that uses all Data augmentation arguments. The absence of any Data Augmentation technique leads to overfitting if we continue the training process (Bottom Left image).

- We can conclude that Data Augmentation significantly improves the model's accuracy; thus, we use Data Augmentation by default in all the next experiments. The *FER-2013* dataset is a small dataset, so it seems essential to apply these regularization techniques in order to generate new training data points and avoid overfitting. The Data Augmentation techniques applied in all of our future experiments, described in Section 5.2.2.

5.3.2 Lack of the Dropout Layer

In this section, we aim to test whether the lack of any Dropout technique affects the classification accuracy of our model. We refer to the results on the Dropout rate experiments in Section 5.3.7.

We use the same architecture along with the hyperparameter optimization as described in the previous section (5.3.1). The Dropout rates are 0.25 for the Convolutional layers and 0.5 for the in-between Fully connected Layers (Figure 5.4: Bottom Right). It is worth denoting that in Keras documentation [143], the hyperparameter ***p*** of the Dropout layer stands for the probability of the element to be dropped (to be zeroed), in contrast with the original paper by Srivastava et al. [113].

The results of this experiment are depicted in Figure 5.4. There is no need to compute the training and validation accuracy since the overfitting is massive (Top Right image).

- We conclude that Dropout layer improves the overall architecture protecting from overfitting.
- The use of Dropout layers affects the overall training time since it takes a bit longer to train.

5.3.3 Experiments on the Momentum Updates

By using the same Type C VGG-like type and architecture as described in 5.3.1, we conducted some experiments on the usefulness of the momentum updates along with the Momentum rate.

As figure 5.5 depicts, we can see that Momentum update along with Nesterov accelerated momentum update help the SGD to converge much faster to find the global optimum. By the end of the 30th epoch, the momentum accelerated method (Figure 5.5: Red-lined model) succeeds a test accuracy of 58.43, same as the test accuracy of the non-accelerated method (Figure 5.5: Yellow-lined model) on the 85th epoch.

A standard set of value for the Momentum update hyperparameter is 0.9. Srivastava et al. [113], claimed that Momentum rates at around 0.95 to 0.99 work a lot better combined with Dropout techniques. In the experiments we conducted, we found that the loss curves are better if we set the Momentum rate to $\gamma = 0.95$ (Figure 5.6). A Momentum rate set to $\gamma = 0.99$ does not improve train loss and accuracy.

- We conclude that in order to achieve better results with the SGD method, we have to combine it with the Momentum and Nesterov accelerated updates.
- With a momentum rate of $\gamma = 0.95$, we can outperform the performance of the standard default setting of $\gamma = 0.9$.

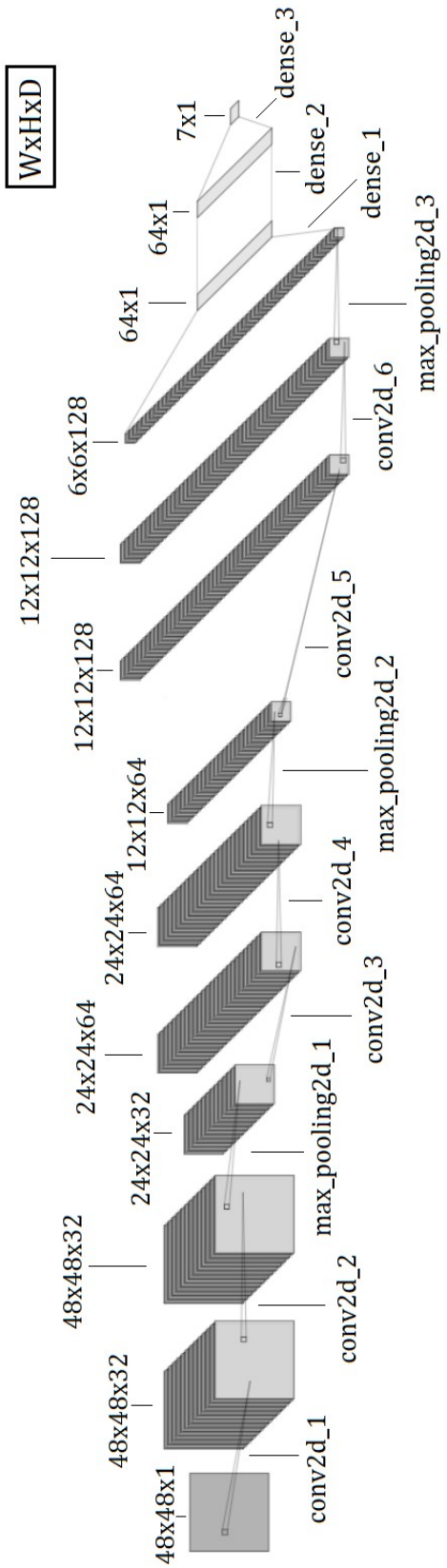


Figure 5.1: Architecture schematic of the Type C VGG-like network used in this model. We visualize every one of the activation map slices, forming the output of the Convolutional layer

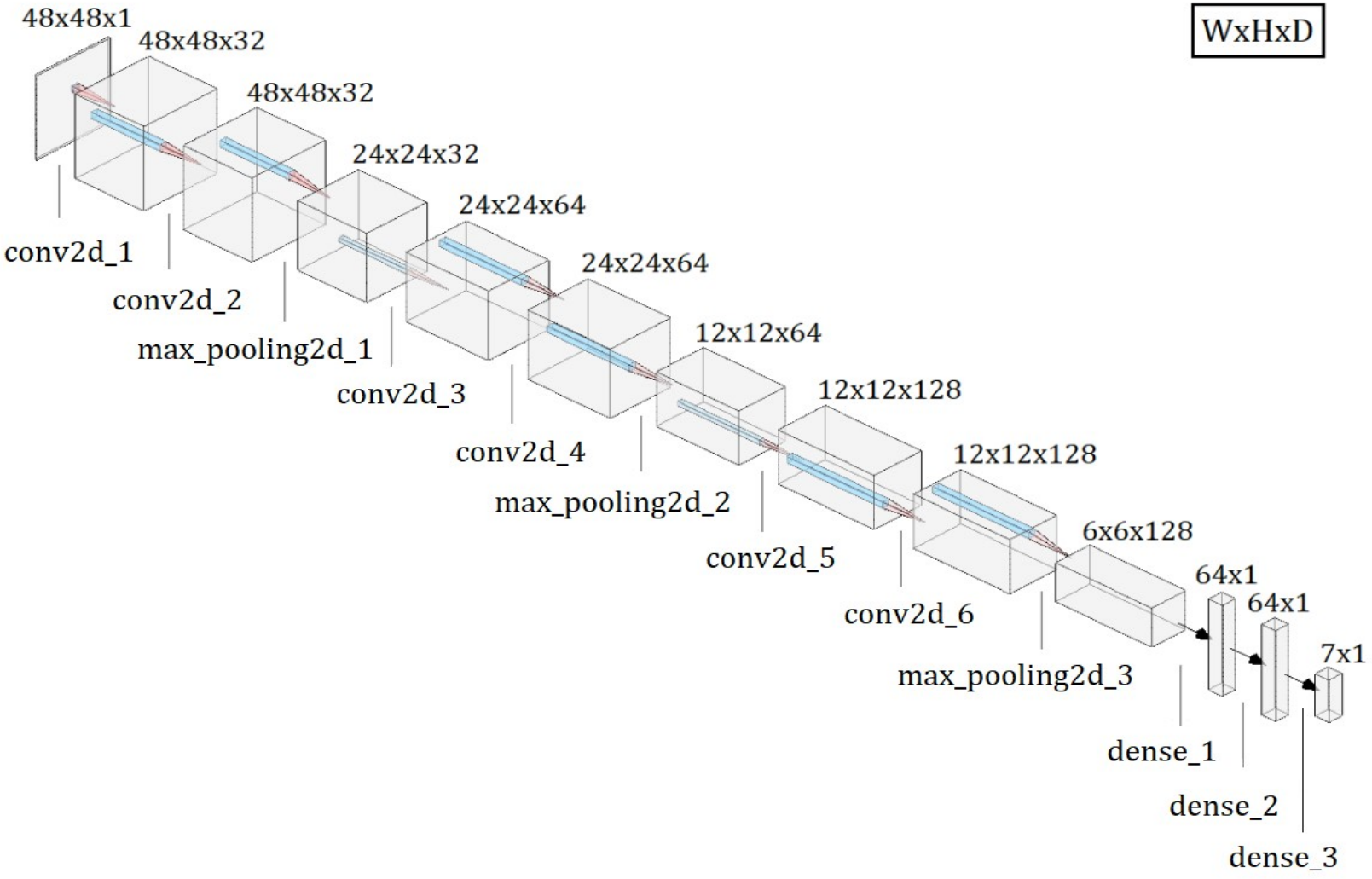
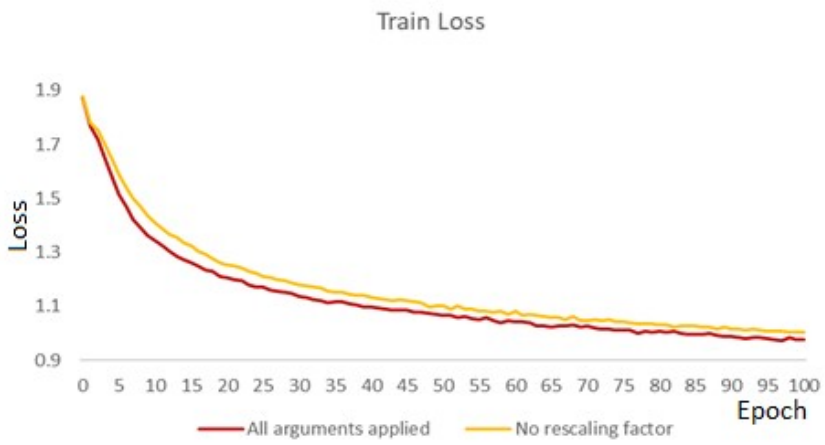
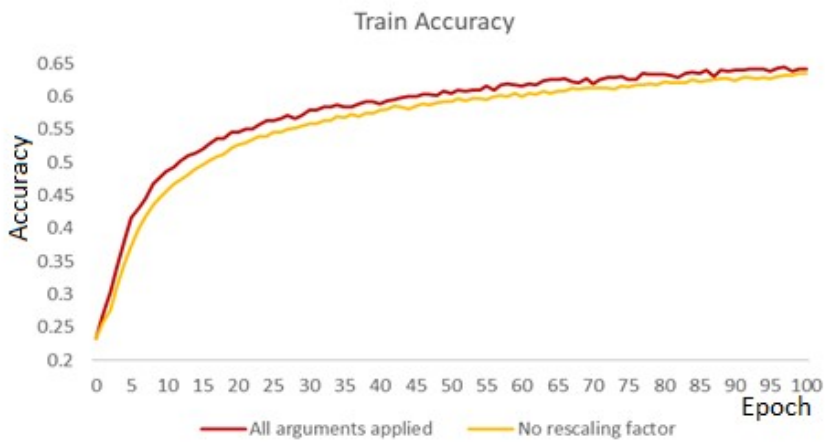
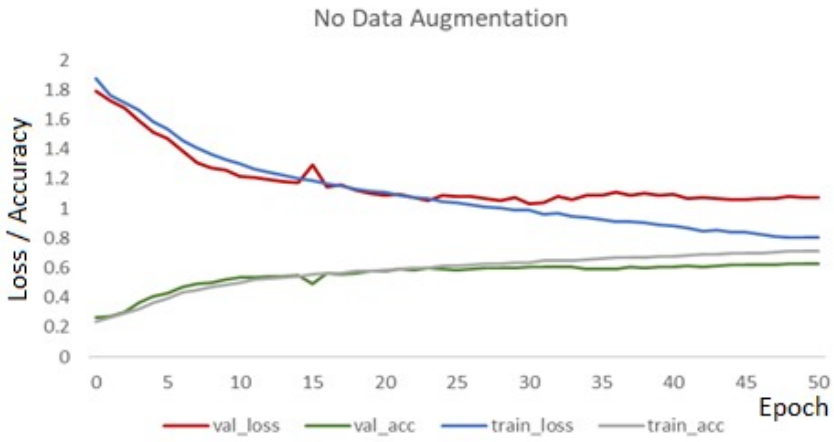


Figure 5.2: Architecture schematic of the Type C VGG-like network we use in this model. Every one of the activation map slices is stacked together forming a single block that represents the output of the Convolutional layer. This is the default architecture schematic we use in our future experiments.

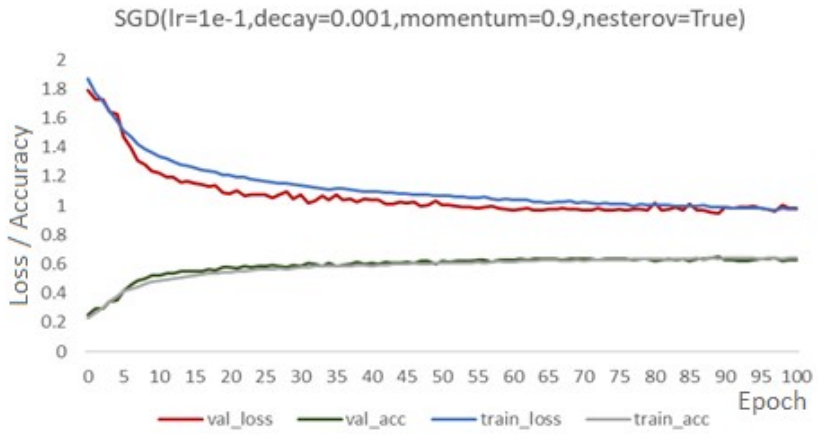


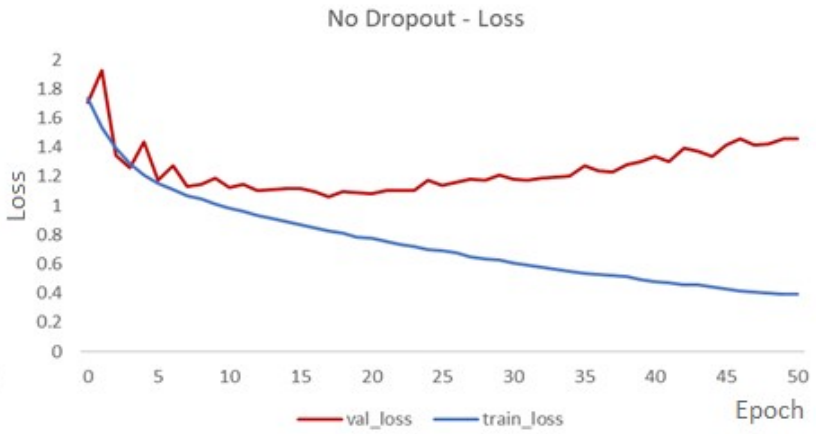


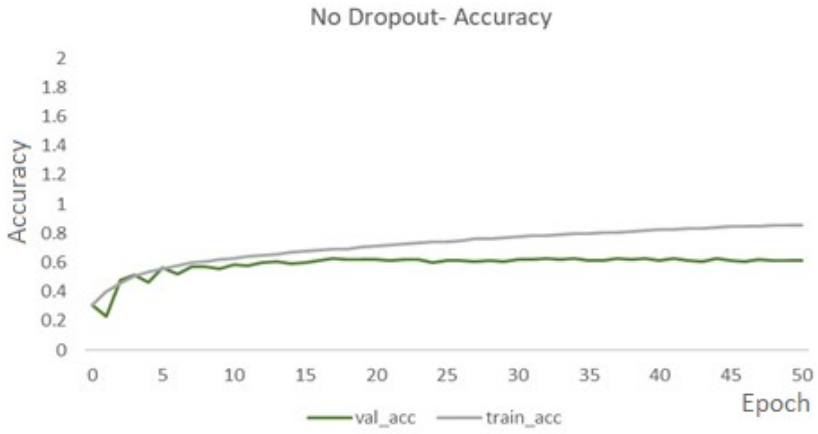


Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Momentum Update	$\gamma = 0.9$
Nesterov Update	True
Decay	0.001
Weight Initialization	Glorot normal

Figure 5.3: Top Left: Illustration of the training loss comparison between the two models. **Top Right**: Illustration of the training accuracy comparison between the two models. **Bottom Left**: No Data Augmentation techniques applied to the dataset. **Bottom right**: The hyperparameters of this experiment.

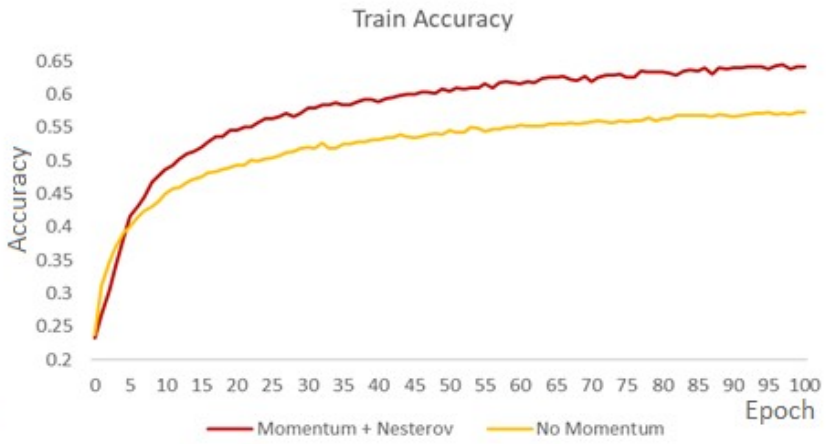






Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Momentum Update	$\gamma = 0.9$
Nesterov Update	True
Decay	0.001
Weight Initialization	Glorot normal
Dropout Rates	• 0.25 for the Convolutional layers • 0.5 for the Fully Connected layers

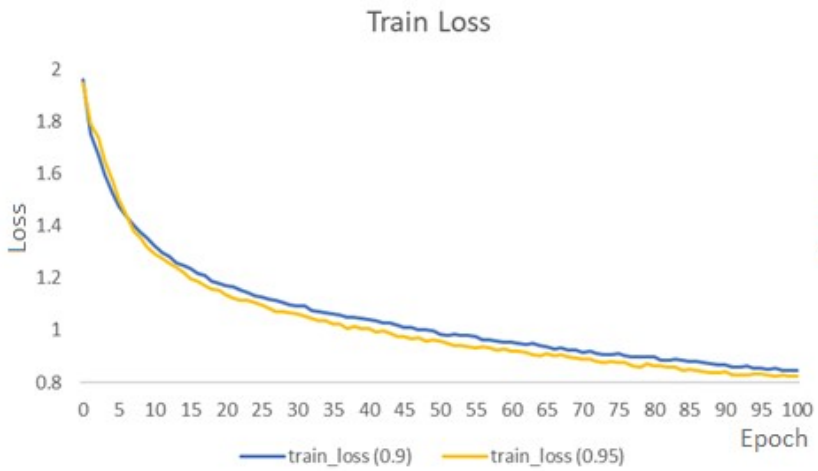
Figure 5.4: Top Left: The SGD optimizer along with its hyperparameters. **Top Right**: The training and validation loss in the absence of any Dropout technique. **Bottom Left**: The training and validation accuracy in the absence of any Dropout technique. **Bottom right**: The hyperparameters of this experiment.



Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Momentum Update	$\gamma = 0.9$
Nesterov Update	True
Decay	0.001
Weight Initialization	Glorot normal
Dropout Rates	• 0.25 for the Convolutional layers • 0.5 for the Fully Connected layers

Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Decay	0.001
Weight Initialization	Glorot normal
Dropout Rates	• 0.25 for the Convolutional layers • 0.5 for the Fully Connected layers

Figure 5.5: **Top Left:** Training loss comparison between the two models. **Top Right**: Training accuracy comparison between the two models. **Bottom Left**: The hyperparameters of the accelerated red-lined experiment. **Bottom right**: The hyperparameters of the non-accelerated yellow-lined experiment.



Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-2$
Momentum Update	$\gamma = 0.9$ (Yellow) $\gamma = 0.95$ (Blue)
Nesterov Update	True
Weight Initialization	Glorot normal
Dropout Rates	• 0.25 for the Convolutional layers • 0.5 for the Fully Connected layers

Figure 5.6: **Top Left:** The train loss comparison between the two hyperparameter Momentum updates. **Top Right**: The train accuracy comparison between the two hyperparameter Momentum updates. **Bottom**: The hyperparameters of the experiment.

5.3.4 Batch Normalization, before or after the Activation Function?

The authors of the Batch Normalization technique [126], suggest that the Batch Normalization transform is added before the non-linearity in the algorithm.

From a statistical point of view, this may cause a problem depending on the activation function we use. In this thesis, we apply either a ReLU or an ELU activation function (Section 4.6). The ReLU activation function is thresholded at zero, giving non-negative numbers as an output. This feature makes ReLU fragile during training. On the other hand, the ELU function can produce negative values.

The Batch Normalization technique normalizes the distribution of features. It first computes the mean of the given minibatch forwarding it to the normalized output function (Section 4.7). Some of these features might have negative values, giving a negative mean (Table 5.4); thus, a negative normalized output. If we normalize before the ReLU activation function, we zero the normalized values (kill the activation) by passing them through the non-negative ReLU. If we use a Batch Normalization layer after the ReLU function, we normalize all the features and pass them to the next Convolutional layer.

In this section, we use two different network architectures, a 7-weight layer Type A (Table 5.1, Figure 5.7) and a deeper 9-weight layer Type C (Table 5.1, Figures 5.1 & 5.2).

Regarding the Type A network (Figure5.8), we use two patterns:

$$\begin{aligned} INPUT \Rightarrow & \left(CONV \Rightarrow RELU \Rightarrow BN \Rightarrow POOL \Rightarrow DO\right)\times 2 \Rightarrow \\ & \Rightarrow \left[\left(CONV \Rightarrow RELU \Rightarrow BN\right)\times 2\right] \Rightarrow POOL \Rightarrow DO \Rightarrow \\ & \Rightarrow \left(FC \Rightarrow RELU \Rightarrow BN \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX, \end{aligned}$$

for the *blue line* curve, and

$$\begin{aligned} INPUT \Rightarrow & \left(CONV \Rightarrow BN \Rightarrow RELU \Rightarrow POOL \Rightarrow DO\right)\times 2 \Rightarrow \\ & \Rightarrow \left[\left(CONV \Rightarrow BN \Rightarrow RELU\right)\times 2\right] \Rightarrow POOL \Rightarrow DO \Rightarrow \\ & \Rightarrow \left(FC \Rightarrow BN \Rightarrow RELU \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX, \end{aligned}$$

for the *yellow line* curve.

The use of Batch Normalization layer after the Activation layer seems to increase the training accuracy of the model. The blue-lined model achieved a training accuracy of 62.72 at the end of the 80th epoch, while the yellow-lined one achieved a 61.44 accuracy by the end of the 95th epoch. These are the best training accuracies of the model.

Regarding the Type C network (Figure5.9), we use two patterns:

$$\begin{aligned} INPUT \Rightarrow & \left\{\left[\left(CONV \Rightarrow ELU \Rightarrow BN\right)\times 2\right] \Rightarrow POOL \Rightarrow DO\right\}\times 3 \Rightarrow \\ & \Rightarrow \left(FC \Rightarrow ELU \Rightarrow BN \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX, \end{aligned}$$

for the *blue line* curve, and

$$\begin{aligned} INPUT \Rightarrow & \left\{\left[\left(CONV \Rightarrow BN \Rightarrow ELU\right)\times 2\right] \Rightarrow POOL \Rightarrow DO\right\}\times 3 \Rightarrow \\ & \Rightarrow \left(FC \Rightarrow BN \Rightarrow ELU \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX, \end{aligned}$$

for the *yellow line* curve.

The use of Batch Normalization layer after the Activation layer seems to favor the training accuracy of the model. By the end of the 75th epoch, both models achieved their best training

accuracies. The blue line curved model achieved a training accuracy of 65.54, compared to 63.39 for the yellow line model.

- We cannot make a safe conclusion on the effect of the ReLU activation function on the Batch Normalization negative moving mean values.
- By placing the Batch Normalization after the activation function, we achieve lower loss and slightly higher accuracy.

Table 5.4: Moving mean weights from the HDF5 file, regarding the 65th epoch of the two models depicted in Figure 5.8. We represent the first 15 means out of 128 from the 4th Batch Normalization layer of the models. We can see that the ReLU activation function culls the negative values.

Filters	ReLU --> BN	BN --> ReLU
1	0.7136606	-5.32962
2	0.31214255	-7.736952
3	7.26E-01	-6.048093
4	8.41E-01	-5.617975
5	0.43556514	-8.206996
6	4.91E-01	-4.919324
7	2.18E-01	-7.189156
8	4.22E-01	-4.636477
9	0.50856704	-4.176944
10	4.16E-01	-7.798513
11	1.25E+00	-6.208605
12	1.0797383	-4.22E+00
13	1.6704437	-2.262339
14	8.31E-01	-5.12595
15	1.122856	-4.007105

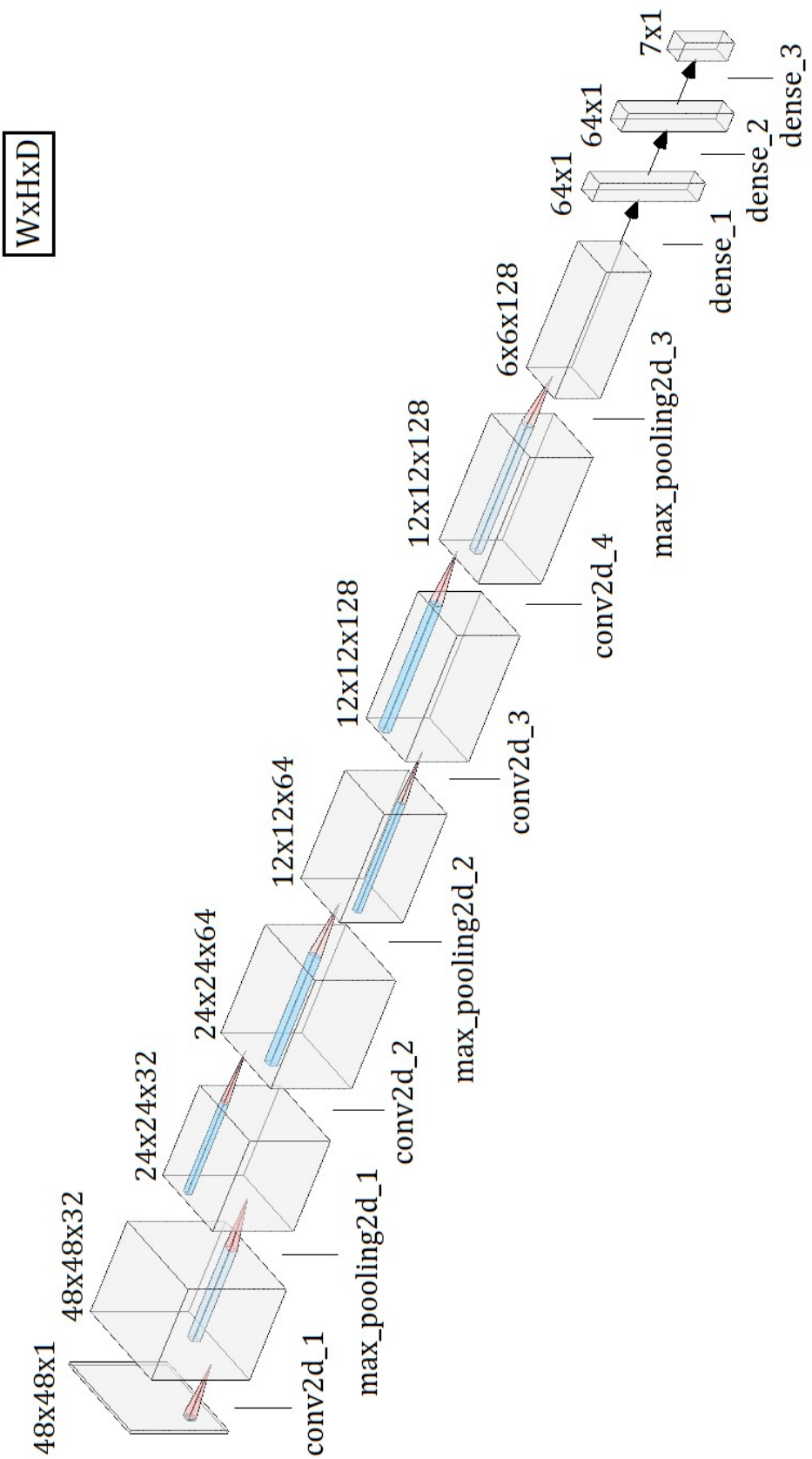
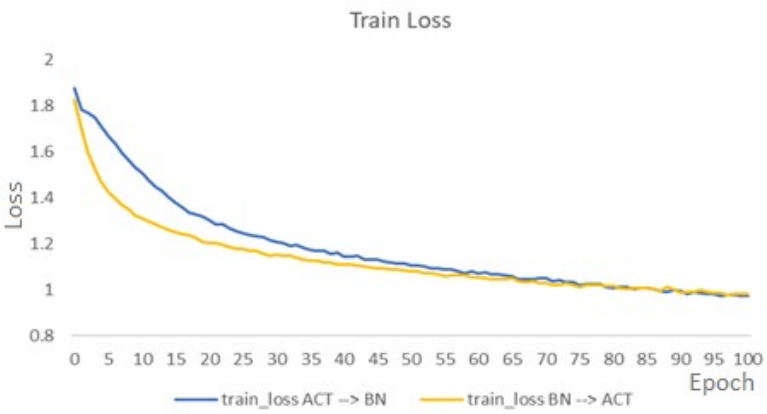
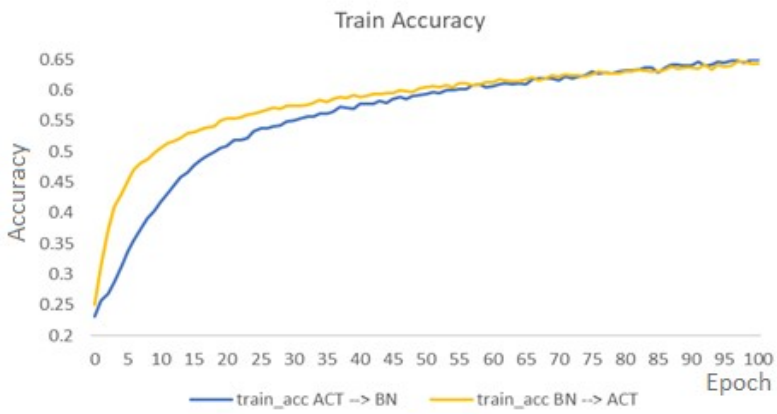


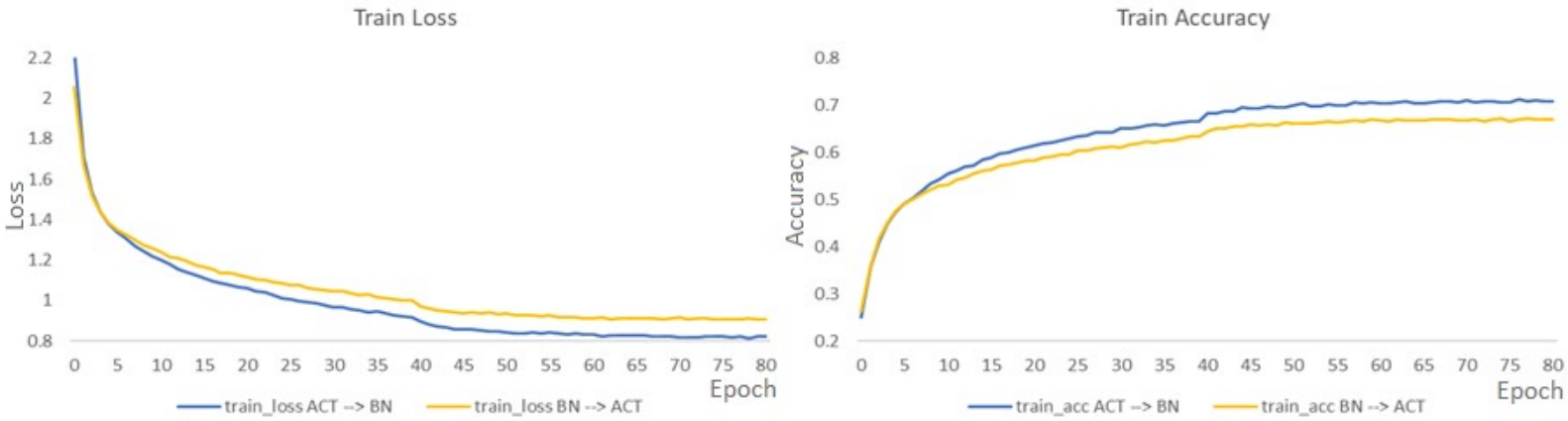
Figure 5.7: Architecture schematic of the Type A VGG-like network we use in this model. Every one of the activation map slices is stacked together, forming a single block that represents the output of the Convolutional layer





Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Momentum Update	$\gamma = 0.9$
Nesterov Update	True
Activation Function	ReLU
Weight Initialization	Glorot normal
Dropout Rates	• 0.25 for the Convolutional layers • 0.5 for the Fully Connected layers

Figure 5.8: Comparison between the training loss (**Top Left**) and accuracy (**Top Right**) of the two patterns, regarding Type A architecture



Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	Adam
Learning Rate	$1e-3$, for the first 40 epochs. $1e-4$, for the next 20 epochs. $1e-5$, for the last 20 epochs
Activation Function	ELU
Weight Initialization	He (MSRA) normal
Dropout Rates	0.25 for the Convolutional layers 0.5 for the Fully Connected layers

Figure 5.9: Comparison between the training loss (**Top Left**) and accuracy (**Top Right**) of the two patterns, regarding Type C architecture

5.3.5 Finding the Best Architecture

We conducted experiments regarding the architecture that fits better on our data. We depict some of them in Figures 5.10 and 5.11. Every model uses the following pattern:

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow RELU \Rightarrow BN\right)\times N\right] \Rightarrow POOL \Rightarrow DO\right\}\times M \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow RELU \Rightarrow BN \Rightarrow DO\right)\times L \Rightarrow Last\ FC \Rightarrow SOFTMAX,$$

In the experiments described by Figure 5.11, we used a decayed SGD (decay=0.001) with the same hyperparameter configuration as in Figure 5.10.

Type A and Type B models do not seem to follow a stable training procedure with fluctuations on the validation curves. If we lower our learning rate and add a regularization L2(0.01) factor on the FC layers, the curve seems to make smoother improvements.

In any case, we can make some conclusions regarding different model architectures:

- We conclude the predominance of Type C model architecture in any case.
- Deeper networks do not necessarily have more parameters, as shown in Table 5.3.
- The architecture design is more an art than a science. Many hyperparameters can be tuned and improve the given architectures at a competitive level.
- Deeper models perform worse, in most of the cases, without overfitting. It seems like deeper models are harder to optimize.

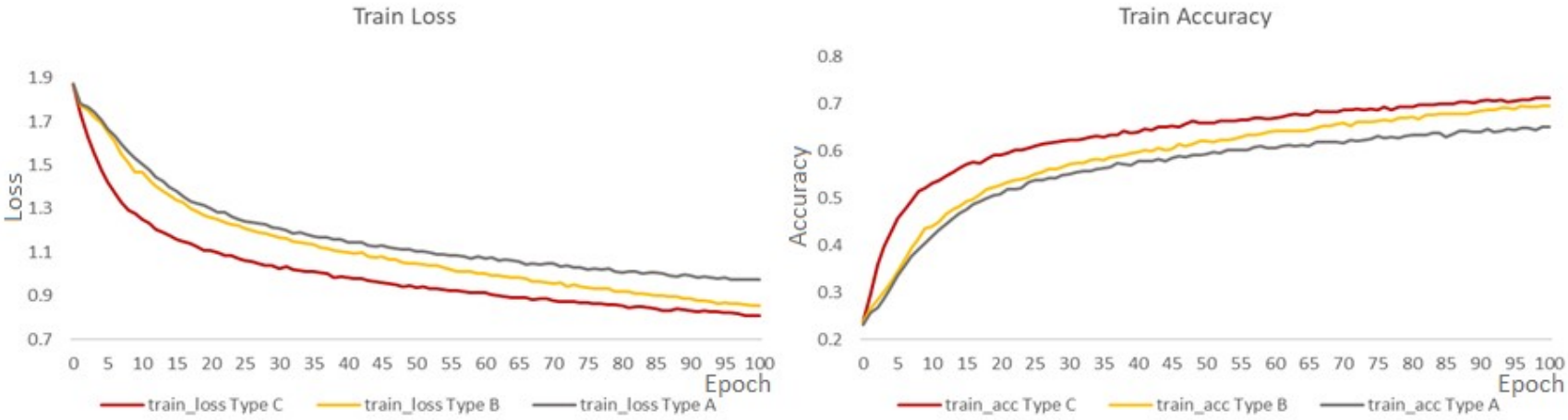
5.3.6 Learning Rate Decay

Choosing the learning rate that fits better on our data is one of the most critical settings in training a NN. A higher learning rate is faster with bigger risk. A lower learning rate avoids fluctuations during the training procedure but is slower. One of the best practices is to start with a high learning rate and decay over time. If we decay the learning rate too slowly, we will have a computationally expensive procedure that takes more time to deliver acceptable results. Using a very high decay will lead to an early stagnation of the training procedure.

By babysitting the training procedure, we can early stop when we do not have the desired results, adjust the hyperparameters and continue the training.

We used two different types of learning decay, the **time-based** decay and the **step decay**. By using the step-based decay, we adjust the learning rate every epoch using the formula $\alpha = \frac{\alpha_0}{1 + k \cdot e}$, where α is the updated learning rate, α_0 the learning rate we use , k the decay rate and e the epoch number. With the step decayed manual method, we drop the learning rate during training by some factor after specific epochs that we choose.

- Step decayed method models were easier to tune and worked better on this experiment, giving the best classification accuracy results (Section 5.3.8).



Loss	Cross-entropy
Classifier	Softmax
Batch Size	128
Optimizer	SGD
Learning Rate	$1e-1$
Momentum Update	$\gamma = 0.9$
Nesterov Update	True
Activation Function	ReLU
Weight Initialization	Glorot normal
Dropout Rates	0.25 for the Convolutional layers 0.5 for the Fully Connected layers

Figure 5.10: **Top Left**: Training loss comparison between different types of architecture. **Top Right**: Training accuracy comparison between different types of architecture. **Bottom**: The hyperparameters of this experiment

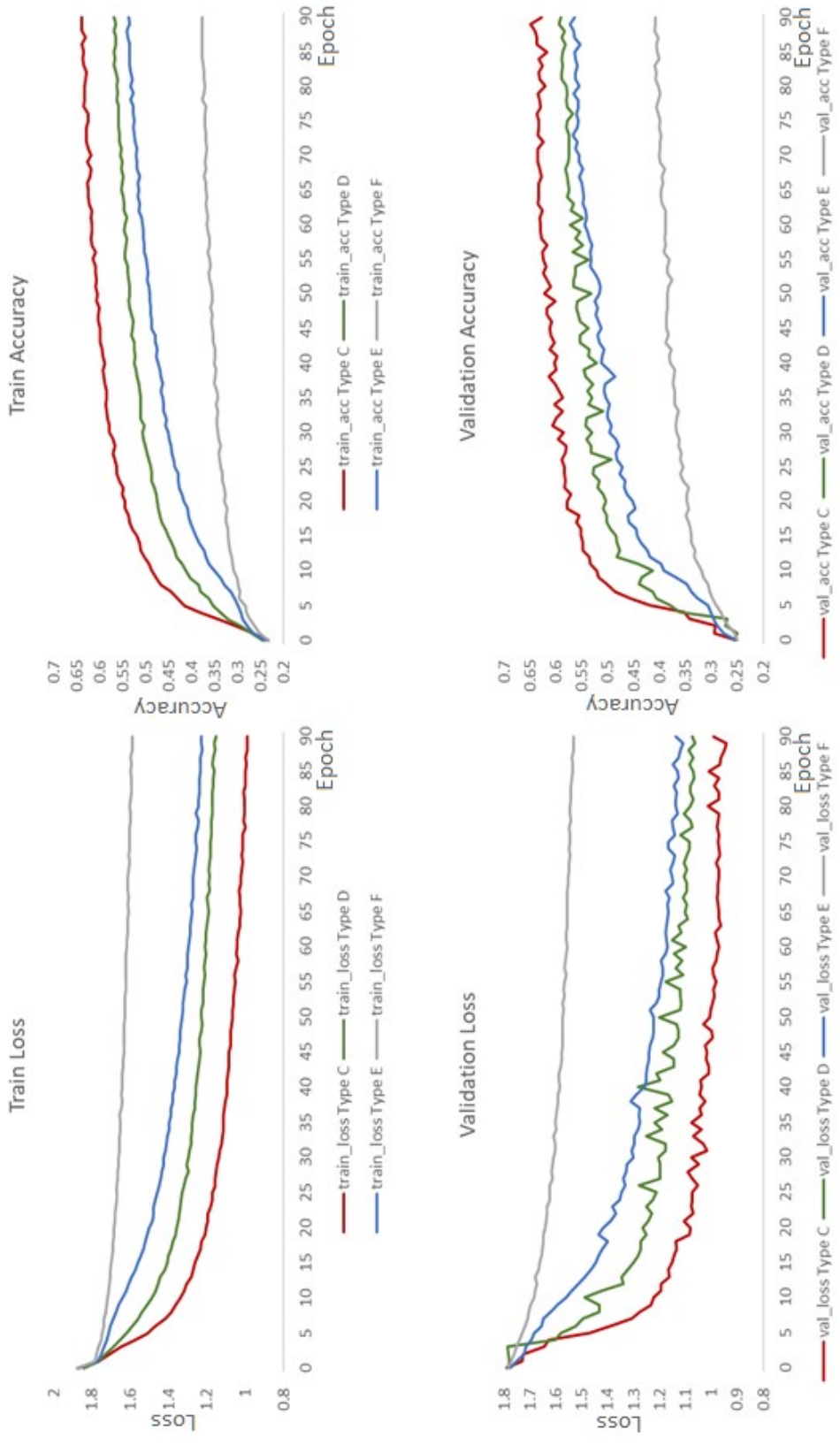


Figure 5.11: Training and validation plots between the different model architectures. We can see the predominance of Type C model in any case

5.3.7 On the Dropout Rates

During the training procedure, we experimented with the values of the Dropout rates. In Keras, the hyperparameter ***p*** of the Dropout layer stands for the probability of the element to be dropped (to be zeroed). A common technique is to place the Dropout layer only in between the Fully Connected layers with $p = 0.5$. We may apply Dropout with lower rates after every Convolutional layer, including the Fully Connected layers.

In this experiment, we use Type C CNNs following the pattern:

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow ELU \Rightarrow BN\right)\times 2\right] \Rightarrow POOL \Rightarrow DO\right\}\times 3 \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow ELU \Rightarrow BN \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX.$$

The hyperparameters for every model are the same (Table 5.5).

The best curves are given by a mixed architecture that uses $p = 0.25$ after the Convolutional layers and $p = 0.5$ (Red line: Figure 5.12) in the in-between Fully Connected layers; along with an architecture that uses $p = 0.3$ in the whole network (Blue line: Figure 5.12). The use of a Dropout rate $p = 0.5$ only in the in-between FC layers leads to massive overfitting by the end of the 40th epoch.

Table 5.5: Hyperparameters for the experiment on the Dropout rates.

Loss	Cross-entropy
Classifier	Softmax
Batch Size	64
Optimizer	Adam
Learning Rate (step decayed)	$1e-3$, for the first 40 epochs $1e-4$, for the next 20 epochs $1e-5$, for the last 20 epochs
Activation Function	ELU
Weight Initialization	He (MSRA) normal
Dropout Rates	• Red line: 0.25 for the Convolutional layers 0.5 for the Fully Connected layers • Blue line: 0.3 in the whole network • Grey line: 0.5 in the whole network • Yellow line: 0.8 in the whole network • Green line: 0.8 only on FC layer

- We conclude that adding Dropout to the Convolutional layers as well reduces the training and validation loss leading to higher classification accuracy.
- To achieve better results, we apply lower Dropout rates on the Convolutional layers, increasing them on the FC in between layers or a low Dropout constant rate on the whole network.

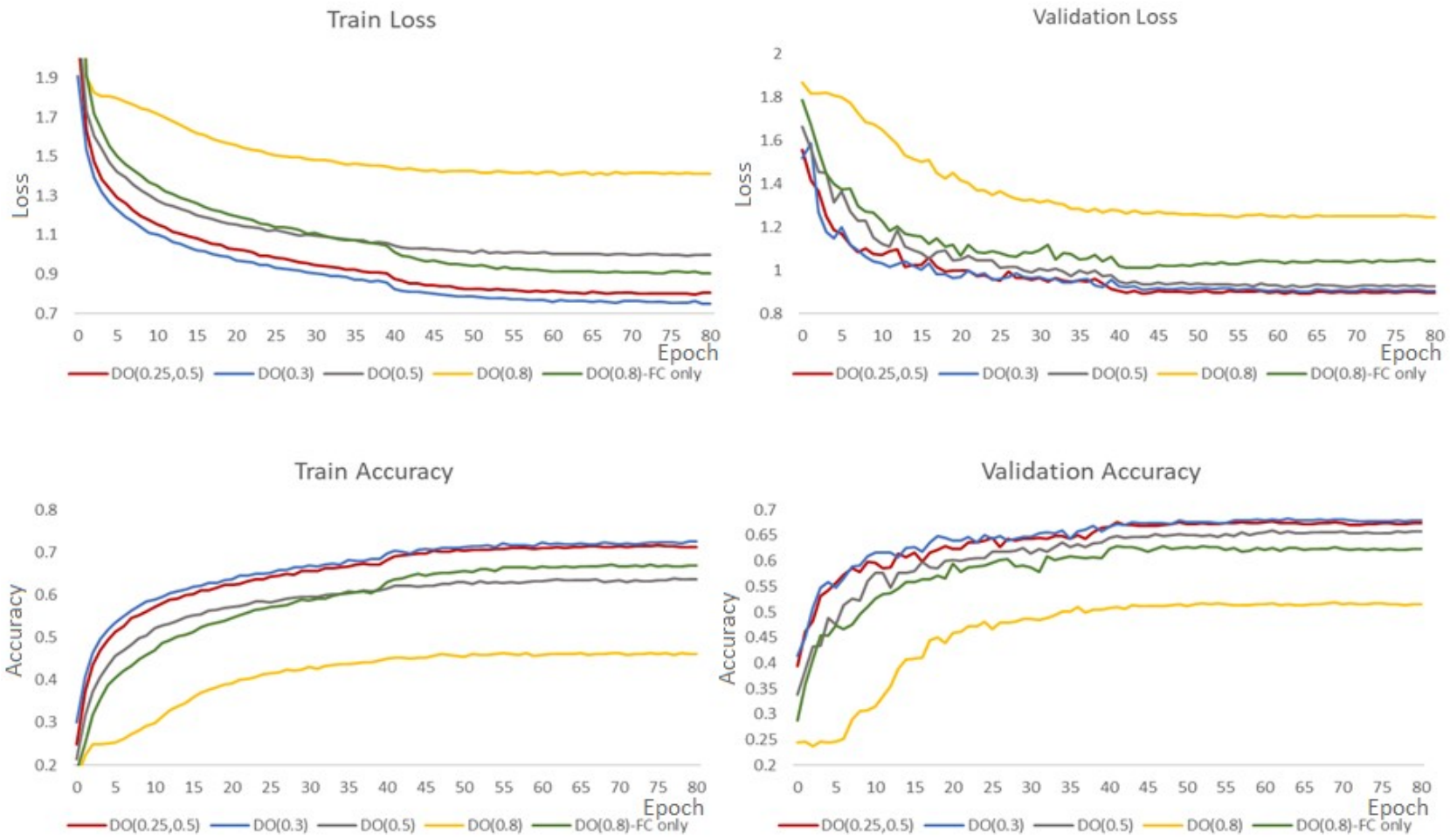


Figure 5.12: Results on the experimentation on the Dropout rates. The Red and Blue models give better results regarding the error and accuracy of the training model

5.3.8 Top 3 Classification Accuracies

In order to succeed high classification accuracy results, we started with an SGD optimizer with momentum as a baseline. During the experimental process, we found that SGD optimizer requires more tuning on its hyperparameters, thus is harder to optimize. We swapped to Adam, an adaptive optimizer that proved a better choice for optimization. In order to avoid the dying ReLU problem (Section 4.6), we used the ELU activation function. He (MSRA) weight initialization method proved a better choice for this type of data.

The top 3 classification accuracies succeeded with a Type C model using the pattern

$$INPUT \Rightarrow \left\{\left[\left(CONV \Rightarrow ELU \Rightarrow BN\right)\times 2\right] \Rightarrow POOL \Rightarrow DO\right\}\times 3 \Rightarrow$$
$$\Rightarrow \left(FC \Rightarrow ELU \Rightarrow BN \Rightarrow DO\right)\times 2 \Rightarrow Last\ FC \Rightarrow SOFTMAX.$$

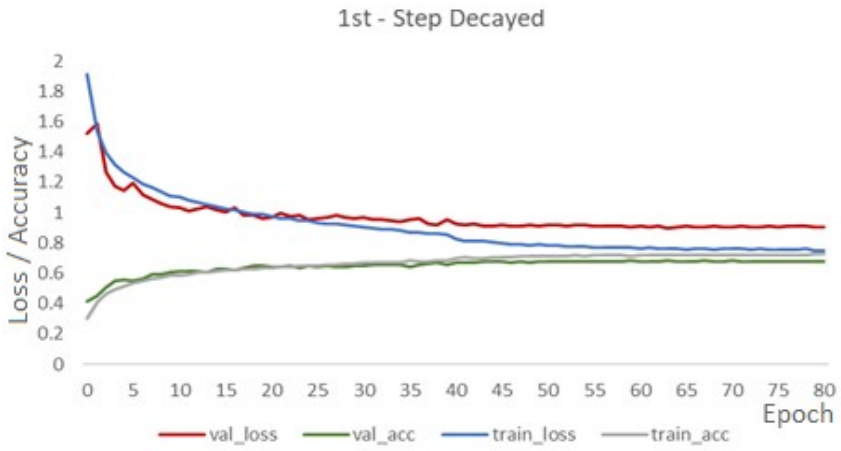
Table 5.6 illustrates the hyperparameters for each of these top 3 models (Figure 5.13). The achieved accuracies for the public set are outlined in Table 5.7

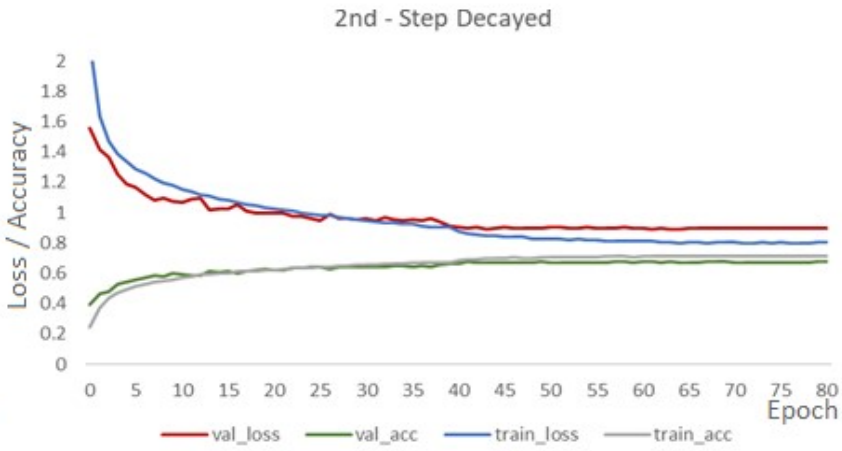
Table 5.6: Illustration of the hyperparameter setting for the best models.

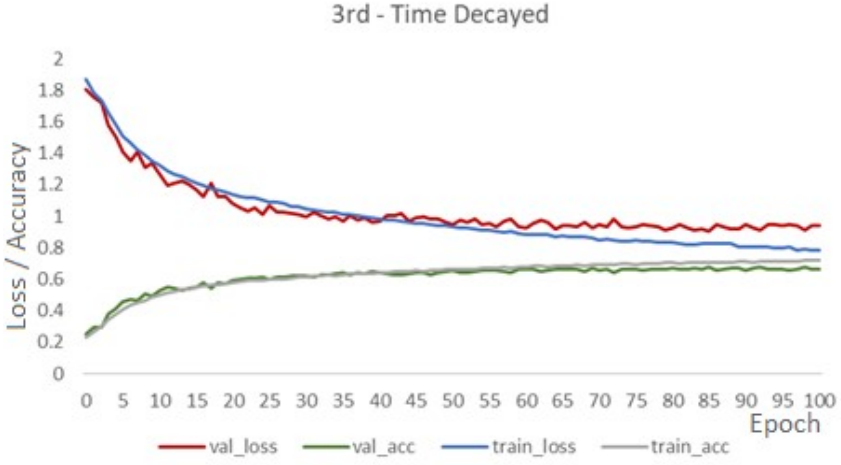
	1st	2nd	3rd
Loss	Cross-entropy	Cross-entropy	Cross-entropy
Classifier	Softmax	Softmax	Softmax
Batch Size	64	64	128
Optimizer	Adam	Adam	Adam
Learning Rate and Other Optimizer Parameters	$1e-3$, for the first 40 epochs $1e-4$, for the next 20 epochs $1e-5$, for the last 20 epochs	$1e-3$, for the first 40 epochs $1e-4$, for the next 20 epochs $1e-5$, for the last 20 epochs	$lr = 1e-1,$ $\text{decay} = 0.001,$ $\beta_1 = 0.9,$ $\beta_2 = 0.999,$ $\varepsilon = 1e-8$
Activation Function	ELU	ELU	ELU
Weight Initialization	He (MSRA) normal	He (MSRA) normal	He (MSRA) normal
Dropout Rates	0.3 in the whole network	0.25 for the Convolutional layers 0.5 for the Fully Connected layers	0.25 for the Convolutional layers 0.5 for the Fully Connected layers

Table 5.7: Private set accuracies for the top 3 model of our experiment.

Model	Private Set Accuracy	Epoch
1st	66.55	80th
2nd	65.88	80th
3rd	65.82	100th







Model	Private Set Accuracy
1st	66.55%
2nd	65.88%
3rd	65.82%

Figure 5.13: This figure illustrates the top three models regarding their classification accuracy, along with their private set accuracies

5.4 Discussion

In the previous section, we implemented several models by optimizing our hyperparameters using different model architectures. The purpose of this section is to summarize the concrete conclusions we reached.

On the effectiveness of Data Augmentation:

1. Data Augmentation techniques significantly improves the model's accuracy. The *FER-2013* dataset is a small dataset, so it seems essential to apply these regularization techniques and avoid overfitting.

On the Dropout layer and the Dropout rates:

2. Dropout layer improves the overall architecture protecting from overfitting.
3. The use of Dropout layers affects the overall training time since it takes a bit longer to train.
4. Adding Dropout to the Convolutional layers reduces the training and validation loss leading to higher classification accuracy.
5. To achieve better results, we apply lower Dropout rates on the Convolutional layers, increasing them on the Fully Connected in between layers or a low Dropout constant rate on the whole network.

On the Momentum Updates:

6. In order to achieve better results with the SGD method, we have to combine it with the Momentum and Nesterov accelerated updates.
7. With a momentum rate of $\gamma = 0.95$, we can outperform the performance of the standard default setting of $\gamma = 0.9$.

On the relative position between Batch Normalization and the Activation Function:

8. We cannot make a safe conclusion on the effect of the ReLU activation function on the Batch Normalization negative moving mean values.
9. By placing the Batch Normalization after the activation function, we achieve lower loss and slightly higher accuracy.

On the Learning Rates:

10. Step decayed method models were easier to tune and worked better on this experiment, giving the best classification accuracy results.

The Best Architecture:

11. We conclude the predominance of Type C model architecture in any case.
12. Deeper networks do not necessarily have more parameters.
13. The architecture design is more an art than a science. Many hyperparameters can be tuned and improve the given architectures at a competitive level.
14. Deeper models perform worse, in most of the cases, without overfitting. It seems like deeper models are harder to optimize.

6
Conclusions and Future Work

Deep learning approaches, the most recent developments in neural networks, have significantly advanced the performance of visual recognition systems. Convolutional Neural Networks are specialized networks for processing data with a grid-like topology and can scale such models to considerable size. They are primarily designed for image-based pattern recognition; thus, they succeed in solving object recognition problems, including facial expression recognition problems.

In this dissertation, we applied a VGG-like CNN model in order to achieve the highest classification accuracy on the *FER-2013* dataset. We tested different model architectures and optimized a lot of the model hyperparameters.

One of the critical issues for accurate and effective deep learning algorithms proved to be hyperparameter optimization. By choosing the optimal set, we can improve many given model architectures to a competitive level.

Our classification results are encouraging since the analysis of faces and expressions performs well with deep learning approaches, but this model is subject to a fundamental limitation. Since we implemented this model on the CPU, it is computationally expensive not allowing us to conduct as many experiments as we desire.

Future Work:
Convolutional Neural Networks has seen exponential growth in grid-based pattern recognition challenges. These types of networks have recently been extended to work on irregularly sampled data, such as graphs; opening the door to a new set of potential applications. Deep convolutional neural networks are actively used medical image analysis proving to be an active research field. Machine learning with CNNs plays a vital role in medical imaging with its applications in automated segmentation (for example, automated tumor segmentation), detection and classification of abnormalities (for example, cancer), image-guided therapy (computer detection or diagnosis) and medical image annotation. The development and use of networks for the needs of medical science is a necessity that needs further work by the scientific community.

Bibliography

[1] Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.
(arXiv:1409.1556)

[2] LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE,* 86(11), 2278-2324.

[3] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. &Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems* 2, NIPS 1989, 396–404. Morgan Kaufmann Publishers.

[4] Ekman, P. (1994). Strong evidence for universals in facial expressions: a reply to Russell's mistaken critique. *Psychological bulletin,* 115 (2)*,* 268-287.

[5] Hubel, D.H., & Wiesel, T.N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology,* 148*,* 574-591.

[6] Hubel, D.H., & Wiesel, T.N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology,* 160*,* 106-54.

[7] Roberts, L.G. (1963). Machine Perception of Three-Dimensional Solids. *Outstanding Dissertations in the Computer Sciences*.

[8] Seymour, P. (1966). The Summer Vision Project.

[9] Marr, D. (1982). Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.

[10] Fischler, M.A., & Elschlager, R.A. (1973). The Representation and Matching of Pictorial Structures. *IEEE Transactions on Computers,* C-22, 67-92.

[11] Lowe, D.G. (1987). Three-Dimensional Object Recognition from Single Two-Dimensional Images. *Artificial Intelligence,* 31, 355-395.

[12] Lowe, D. G. (1999). Object recognition from local scale-invariant features. *Proceedings of the Seventh IEEE ICCV.*

[13] Viola, P. A. & Jones, M. J. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. *IEEECVPR ,*1, 511-518.

[14] Freund, Y. & Schapire, R. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55, 119-139.

[15] Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *IEEE CVPR 2005,* 1(1)*,* 886-893.

[16] Lazebnik, S., Schmid, C., & Ponce, J. (2006). Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. *IEEECVPR 2006,* 2*,* 2169-2178.

[17] Pantic, M., & Rothkrantz, L.J. (2000). Automatic Analysis of Facial Expressions: The State of the Art. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 22(12)*,* 1424-1445.

[18] Caleanu, C. (2013). Face expression recognition: A brief overview of the last decade. *2013 IEEE 8th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 157-161.

[19] Tian, Y.-L., Kanade, T., & Cohn, J. F. (2005). Facial Expression Analysis. In *Handbook of Face Recognition,* 247–275.

[20] Ekman, P., & Friesen, W. V. (1976). Measuring facial movement. *Environmental Psychology and Nonverbal Behavior,* 1(1), 56–75.

[21] Jack, R. E., Garrod, O. G. B., Yu, H., Caldara, R., & Schyns, P. G. (2012). Facial expressions of emotion are not culturally universal*. Proceedings of the National Academy of Sciences,* 109(19), 7241–7244.

[22] Li, S., & Deng, W. (2018). Deep Facial Expression Recognition: A Survey. (arXiv:1804.08348)

[23] Samal, A., & Iyengar, P. A. (1992). Automatic recognition and analysis of human faces and facial expressions: a survey. *Pattern Recognition,* 25(1), 65–77.

[24] Fasel, B., & Luettin, J. (2003). Automatic facial expression analysis: a survey. *Pattern Recognition,* 36(1), 259–275.

[25] Zhihong Zeng, Pantic, M., Roisman, G. I., & Huang, T. S. (2009). A Survey of Affect Recognition Methods: Audio, Visual, and Spontaneous Expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 31(1), 39–58.

[26] Sariyanidi, E., Gunes, H., & Cavallaro, A. (2015). Automatic Analysis of Facial Affect: A Survey of Registration, Representation, and Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 37(6), 1113–1133.

[27] Kobayashi H, Hara F (1991) The recognition of basic facial expressions by neural network. *Proc IEEE International Joint Conference on Neural Networks*, 1, 460– 466.

[28] Padgett, C. & Cottrell, G. W. (1996). Representing Face Images for Emotion Classification. In *M. Mozer, M. I. Jordan & T. Petsche (eds.), NIPS'96* 894-900*:* MIT Press.

[29] Essa, I. A., & Pentland, A. P. (1997). Coding, analysis, interpretation, and recognition of facial expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 19 (7), 757–763.

[30] Pentland, Moghaddam, & Starner. (1994). View-based and modular eigenspaces for face recognition. *Proceedings of IEEE CVPR-94.*

[31] Simoncelli, E. P. (1993). *Distributed representation and analysis of visual motion.* Unpublished doctoral dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA.

[32] Hong, H., Neven, H. & von der Malsburg, C. (1998). Online Facial Expression Recognition Based on Personalized Galleries, 354-359.

[33] Steffens, J., Elagin, E. & Neven, H. (1998). PersonSpotter - Fast and Robust System for Human Detection, Tracking and Recognition, 516-521.

[34] Zhang, Z., Lyons, M.J., Schuster, M., & Akamatsu, S. (1998). Comparison Between Geometry-Based and Gabor-Wavelets-Based Facial Expression Recognition Using Multi-Layer Perceptron.

[35] Lyons, M. J., Budynek, J., & Akamatsu, S. (1999). Automatic classification of single facial images. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 21(12), 1357–1362.

[36] Kotsia, I., & Pitas, I. (2007). Facial Expression Recognition in Image Sequences Using Geometric Deformation Features and Support Vector Machines. *IEEE Transactions on Image Processing,* 16(1), 172–187.

[37] Bartlett, M. S., Littlewort, G., Braathen, B., Sejnowski, T. J. & Movellan, J. R. (2002). A Prototype for Automatic Recognition of Spontaneous Facial Actions. In S. Becker, S. Thrun & K. Obermayer (eds.), *NIPS*, 1271-1278: MIT Press.

[38] Ahlberg, J. (2001). CANDIDE-3 - An Updated Parameterised Face.

[39] Lucey, S., Ashraf, A.B., & Cohn, J.F. (2007). Investigating Spontaneous Facial Action Recognition through AAM Representations of the Face.

[40] Cohen, I., Sebe, N., Garg, A., Chen, L.S., & Huang, T.S. (2003). Facial expression recognition from video sequences: temporal and static modeling. *Computer Vision and Image Understanding,* 91, 160-187.

[41] Tao, H., & Huang, T.S. (1998). Connected Vibrations: A Modal Analysis Approach for Non-Rigid Motion Tracking. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 735-740

[42] Friedman, N., Geiger, D., & Goldszmidt, M. (1997). Bayesian Network Classifiers. *Machine Learning,* 29, 131-163.

[43] Ma, L., & Khorasani, K. (2004). Facial Expression Recognition Using Constructive Feedforward Neural Networks. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics), 34(3),* 1588–1595.

[44] Shan, C., Gong, S., & McOwan, P.W. (2005). Robust facial expression recognition using local binary patterns. *IEEE International Conference on Image Processing* 2005, *2*, 370-373.

[45] Wang, J., Yin, L., Wei, X., & Sun, Y. (2006). 3D Facial Expression Recognition Based on Primitive Surface Feature Distribution. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06*), 2*,* 1399-1406.

[46] Panning, A., Al-Hamadi, A.K., Niese, R., & Michaelis, B. (2008). Facial expression recognition based on Haar-like feature detection. *Pattern Recognition and Image Analysis,* 18, 447-452.

[47] Buciu, I., Kotropoulos, C., & Pitas, I. (2009). Comparison of ICA approaches for facial expression recognition. *Signal, Image and Video Processing,* 3*,* 345-361.

[48] Valstar, M.F., & Pantic, M. (2012). Fully Automatic Recognition of the Temporal Phases of Facial Actions. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics),* 42, 28-43.

[49] Zhi, R., Flierl, M., Ruan, Q., & Kleijn, W.B. (2011). Graph-Preserving Sparse Nonnegative Matrix Factorization with Application to Facial Expression Recognition. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics),* 41*,* 38-52.

[50] Nikitidis, S., Tefas, A., Nikolaidis, N., & Pitas, I. (2012). Subclass discriminant Nonnegative Matrix Factorization for facial image analysis. *Pattern Recognition,* 45*,* 4080-4091.

[51] Mahoor, M.H., Zhou, M., Veon, K.L., Mavadati, S.M., & Cohn, J.F. (2011). Facial action unit recognition with sparse representation. *Face and Gesture 2011*, 336-342.

[52] Zafeiriou, S.P., & Petrou, M. (2010). Sparse representations for facial expressions recognition via l1 optimization. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, 32-39.

[53] Candes, E.J., & Wakin, M.B. (2008). An Introduction to Compressive Sampling. *IEEE Signal Processing Magazine,* 25, 21-30.

[54] Ahonen, T., Hadid, A., & Pietikäinen, M. (2006). Face Description with Local Binary Patterns: Application to Face Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 28, 2037-2041.

[55] Zhao, G., & Pietikäinen, M. (2007). Dynamic Texture Recognition Using Local Binary Patterns with an Application to Facial Expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 29*, 915-928.

[56] Fei-Fei, L., Fergus, R., & Perona, P. (2004). Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. *CVPR Workshops*.

[57] Everingham, M., Eslami, S.M., Gool, L.V., Williams, C.K., Winn, J.M., & Zisserman, A. (2014). The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision, 111*, 98-136.

[58] Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248-255.

[59] Mollahosseini, A., Chan, D., & Mahoor, M.H. (2016). Going deeper in facial expression recognition using deep neural networks. *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 1-10.
(arXiv:1511.04110)

[60] Sun, Y., Wang, X., & Tang, X. (2013). Deep Convolutional Network Cascade for Facial Point Detection. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 3476-3483.

[61] Zhang, Z., Luo, P., Loy, C. C. & Tang, X. (2014). Facial landmark detection by deep multi-task learning. *European Conference on Computer Vision*, 94-108.

[62] Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks. *IEEE Signal Processing Letters, 23*, 1499-1503.

[63] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. C. & Bengio, Y. (2014). Generative Adversarial Nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence & K. Q. Weinberger (eds.), *NIPS*, 2672-2680.

[64] Lai, Y., & Lai, S. (2018). Emotion-Preserving Representation Learning via Generative Adversarial Network for Multi-View Facial Expression Recognition. *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, 263-270.

[65] Zhang, F., Zhang, T., Mao, Q., & Xu, C. (2018). Joint Pose and Expression Modeling for Facial Expression Recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 3359-3368.

[66] Huang, R., Zhang, S., Li, T., & He, R. (2017). Beyond Face Rotation: Global and Local Perception GAN for Photorealistic and Identity Preserving Frontal View Synthesis. *2017 IEEE International Conference on Computer Vision (ICCV)*, 2458-2467.
(arXiv:1704.04086)

[67] Yin, X., Yu, X., Sohn, K., Liu, X., & Chandraker, M.K. (2017). Towards Large-Pose Face Frontalization in the Wild. *2017 IEEE International Conference on Computer Vision (ICCV)*, 4010-4019.
(arXiv:1704.06244)

[68] Tran, L., Yin, X., & Liu, X. (2017). Disentangled Representation Learning GAN for Pose-Invariant Face Recognition. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1283-1292.

[69] Fukushima, K., & Miyake, S. (1982). Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition, 15*, 455-469.

[70] LeCun, Y., Bengio, Y., & Hinton, G. (2015). *Deep learning. Nature, 521(7553),* 436–444.

[71] Girshick, R.B., Donahue, J., Darrell, T., & Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 580-587.
(arXiv:1311.2524)

[72] Girshick, R.B. (2015). Fast R-CNN. *2015 IEEE International Conference on Computer Vision (ICCV)*, 1440-1448.
(arXiv:1504.08083)

[73] Ren, S., He, K., Girshick, R.B., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 39*, 1137-1149.
(arXiv:1506.01497)

[74] Redmon, J., Divvala, S.K., Girshick, R.B., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779-788.
(arXiv:1506.02640)

[75] Ji, S., Xu, W., Yang, M.W., & Yu, K. (2010). 3D Convolutional Neural Networks for Human Action Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 35*, 221-231.

[76] Tran, D., Bourdev, L.D., Fergus, R., Torresani, L., & Paluri, M. (2015). Learning Spatiotemporal Features with 3D Convolutional Networks. *2015 IEEE International Conference on Computer Vision (ICCV)*, 4489-4497.
(arXiv:1412.0767)

[77] Hinton, G.E., Osindero, S., & Teh, Y.W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation, 18*, 1527-1554.

[78] Hinton, G. E. & Sejnowski, T. E. (1986). Learning and Relearning in Boltzmann Machines. In *Parallel Distributed Processing* ,1, 282-317. MIT Press.

[79] Hinton, G. E. (2012). A Practical Guide to Training Restricted Boltzmann Machines.. In G. Montavon, G. B. Orr & K.-R. Müller (ed.), *Neural Networks: Tricks of the Trade (2nd ed.)*, 7700, 599-619. Springer.

[80] Liu, P., Han, S., Meng, Z., & Tong, Y. (2014). Facial Expression Recognition via a Boosted Deep Belief Network. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 1805-1812.

[81] Duan, T. & Srihari, S. N. (2016). Pseudo Boosted Deep Belief Network. In A. E. P. Villa, P. Masulli & A. J. P. Rivero (eds.), *ICANN (2)* (p./pp. 105-112): Springer.

[82] Hinton, G. E. & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313, 504-507.

[83] Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press

[84] Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature, 323*, 533-536.

[85] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*, 1735-1780.

[86] Gers, F.A., Schmidhuber, J., & Cummins, F.A. (2000). Learning to Forget: Continual Prediction with LSTM. *Neural Computation, 12*, 2451-2471.

[87] Sutskever, I., Vinyals, O. & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 3104-3112.

[88] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.

[89] Bahdanau, D., Cho, K. & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.
(arXiv:1409.0473)

[90] Graves, A. & Jaitly, N. (2014). Towards End-To-End Speech Recognition with Recurrent Neural Networks. *ICML*, 1764-1772.

[91] Graves, A. & Schmidhuber, J. (2008). Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In D. Koller, D. Schuurmans, Y. Bengio & L. Bottou (eds.), *NIPS,* 545-552: Curran Associates, Inc.

[92] Graves, A. (2013). Generating Sequences with Recurrent Neural Networks. (arXiv:1308.0850)

[93] Kiros, R., Salakhutdinov, R. & Zemel, R. S. (2014). Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models. (arXiv:1411.2539)

[94] Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and tell: A neural image caption generator. (arXiv:1411.4555)

[95] Yang, J., Zeng, X., Zhong, S. & Wu, S. (2013). Effective Neural Network Ensemble Approach for Improving Generalization Performance. *IEEE Trans. Neural Netw. Learning Syst.*, 24, 878-887.

[96] Tao, S. (2019). Deep Neural Network Ensembles. (arXiv:1904.05488)

[97] Hamester, D., Barros, P.V., & Wermter, S. (2015). Face expression recognition with a 2-channel Convolutional Neural Network. *2015 International Joint Conference on Neural Networks (IJCNN)*, 1-8.

[98] Demiriz, A., Bennett, K. & Shawe-Taylor, J. (2002). Linear Programming Boosting via Column Generation. *Machine Learning*, 46, 225-254.

[99] Liu, K., Zhang, M., & Pan, Z. (2016). Facial Expression Recognition with CNN Ensemble. *2016 International Conference on Cyberworlds (CW)*, 163-166.

[100] Yu, Z., Liu, G., Liu, Q., & Deng, J. (2018). Spatio-temporal convolutional features with nested LSTM for facial expression recognition. *Neurocomputing, 317*, 50-57.

[101] Lv, Y., Feng, Z., & Xu, C. (2014). Facial expression recognition via deep learning. *2014 International Conference on Smart Computing*, 303-308.

[102] Baccouche, M., Mamalet, F., Wolf, C., Garcia, C. & Baskurt, A. (2012). Spatio-Temporal Convolutional Sparse Auto-Encoder for Sequence Classification. In R. Bowden, J. P. Collomosse & K. Mikolajczyk (eds.), *BMVC* (p./pp. 1-12): BMVA Press.

[103] Goodfellow I.J. et al. (2013) Challenges in Representation Learning: A Report on Three Machine Learning Contests. In: Lee M., Hirose A., Hou ZG., Kil R.M. (eds) *Neural Information Processing. ICONIP 2013,* 117-124. *Lecture Notes in Computer Science, vol 8228*. Springer, Berlin, Heidelberg

[104] Bergstra, J. & Cox, D. D. (2013). Hyperparameter Optimization and Boosting for Classifying Facial Expressions: How good can a "Null" Model be?

[105] Tang, Y. (2013). Deep Learning using Support Vector Machines. (arXiv:1306.0239)

[106] Ionescu, R.T., & Grozea, C. (2013). Local Learning to Improve Bag of Visual Words Model for Facial Expression Recognition.

[107] Sivic, J., Russell, B.C., Efros, A.A., Zisserman, A., & Freeman, W.T. (2005). Discovering object categories in image collections. *IEEE ICCV 2005*,*1*, 370-377.

[108] Zhang, J., Marszałek, M., Lazebnik, S., & Schmid, C. (2006). Local Features and Kernels for Classification of Texture and Object Categories: A Comprehensive Study. *International Journal of Computer Vision, 73(2),* 213–238.

[109] Boser, B. E.., Guyon, I. M. & Vapnik, V. N. (1992). A training algorithm for optimal margin classifier. *Proceedings of the fifth Annual ACM Workshop on Computational Learning Theory*, 144-152.

[110] Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning, 20*, 273-297.

[111] Weston, J. & Watkins, C. (1999). Support vector machines for multi-class pattern recognition. *ESANN*, 219-224.

[112] Lee, C.-P. & Lin, C.-J. (2013). A Study on L2-Loss (Squared Hinge-Loss) Multiclass SVM. *Neural Computation*, 25, 1302-1323.

[113] Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R.R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*, 1929-1958.

[114] Prechelt, L. (1996). Early Stopping-But When? *Neural Networks: Tricks of the Trade*.

[115] Lemsrechal, C. (2012). Cauchy and the Gradient Method. Documenta Mathematica. · ISMP, 251–254

[116] Bottou, L. (2012). Stochastic Gradient Descent Tricks. In G. Montavon, G. B. Orr & K.-R. Müller (ed.), *Neural Networks: Tricks of the Trade (2nd ed.)*, 7700, 421-436. Springer.

[117] Polyak, B. T. (1964). *Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics, 4(5), 1–17.*

[118] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12, 145-151.

[119] Nesterov, Y.E. (1983). A method for solving the convex programming problem with convergence rate O(1/k^2). *Proceedings of the USSR Academy of Sciences, 269*, 543-547.

[120] Sutskever, I., Martens, J., Dahl, G. E. & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. *ICML,3,* 1139-1147.

[121] Kingma, D.P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. (arXiv:1412.6980)

[122] Nair, V. & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In J. Fürnkranz & T. Joachims (eds.), *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 807-814.

[123] Fast and Accurate Deep Network Learning by Exponential Linear Units. (arXiv:1511.07289)

[124] Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh & D. M. Titterington (eds.), *AISTATS*, 249-256.

[125] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, 1026-1034.

[126] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ICML*. (arXiv:1502.03167)

[127] Zhou, Y.T., & Chellappa, R. (1988). Computation of optical flow using a neural network. *IEEE 1988 International Conference on Neural Networks*, 2, 71-78.

[128] Krizhevsky, A., Sutskever, I., & Hinton, G.E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM, 60*, 84-90.

[129] Sang, D.V., Dat, N.V., & Thuan, D.P. (2017). Facial expression recognition using deep convolutional neural networks. *2017 9th International Conference on Knowledge and Systems Engineering (KSE)*, 130-135.

[130] Raghuvanshi, A., & Choksi, V. (2016). Facial Expression Recognition with Convolutional Neural Networks.

[131] Parkhi, O.M., Vedaldi, A., & Zisserman, A. (2015). Deep Face Recognition. *BMVC*.

[132] Alizadeh, S., & Fazel, A. (2017). Convolutional Neural Networks for Facial Expression Recognition. (arXiv:1704.06756)

[133] cisco.com (2019). Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. Retrieved from https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

[134] host.robots.ox.ac.uk. The PASCAL Visual Object Classes Homepage. Retrieved from http://host.robots.ox.ac.uk/pascal/VOC/

[135] Stanford Vision Lab (2016). ImageNet. Retrieved from http://www.image-net.org/

[136] Geoffrey E. Hinton (2009). Deep belief networks. Retrieved from http://www.scholarpedia.org/article/Deep_belief_networks

[137] deeplearning.net (2013). ICML 2013 Challenges in Representation Learning.
Retrieved from http://deeplearning.net/icml2013-workshop-competition/

[138] Kaggle Inc (2013). Challenges in Representation Learning: Facial Expression Recognition Challenge.
Retrieved from https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge

[139] OpenCV team (2019). OpenCV.
Retrieved from https://opencv.org/

[140] Nielsen, A. (2015). Neural Networks and Deep Learning.
Retrieved from http://neuralnetworksanddeeplearning.com/

[141] Stanford Vision Lab (2019). CS231n: Convolutional Neural Networks for Visual Recognition.
Retrieved from http://cs231n.stanford.edu/

[142] The HDF Group (2019). The HDF5 Library and File Format.
Retrieved from https://www.hdfgroup.org/solutions/hdf5/

[143] Keras Documentation (2019). Core Layers.
Retrieved from
https://keras.io/layers/core/