IMPROVED ARTIFICIAL NEURAL NETWORK TRAINING WITH ADVANCED METHODS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL AND ELECTRONICS ENGINEERING

By

Burak Çatalbaş

September 2018

IMPROVED ARTIFICIAL NEURAL NETWORK TRAINING WITH ADVANCED METHODS

By Burak Çatalbaş

September 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ömer Morgül(Advisor)

Süleyman Serdar Kozat

Ramazan Gökberk Cinbiş

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

IMPROVED ARTIFICIAL NEURAL NETWORK TRAINING WITH ADVANCED METHODS

Burak Çatalbaş
M.S. in Electrical and Electronics Engineering
Advisor: Ömer Morgül
September 2018

Artificial Neural Networks (ANNs) are used for different machine learning tasks such as classification, clustering etc. They have been utilized in important tasks and offering new services more and more in our daily lives. Learning capabilities of these networks have accelerated significantly since 2000s, with the increasing computational power and data amount. Therefore, research conducted on these networks is renamed as Deep Learning, which emerged as a major research area - not only in the neural networks, but also in the Machine Learning discipline. For such an important research field, the techniques used in the training of these networks can be seen as keys for more successful results. In this work, each part of this training procedure is investigated by using of different and improved - sometimes new - techniques on convolutional neural networks which classify grayscale and colored image datasets. Advanced methods included the ones from the literature such as He-truncated Gaussian initialization. In addition, our contributions to the literature include ones such as SinAdaMax Optimizer, Dominantly Exponential Linear Unit (DELU), He-truncated Laplacian initialization and Pyramid Approach for Max-Pool layers. In the chapters of this thesis, success rates are increased with the addition of these advanced methods accumulatively, especially with DELU and SinAdaMax which are our contributions as upgraded methods.

In result, success rate thresholds for different datasets are met with simple convolutional neural networks - which are improved with these advanced methods and reached promising test success increases - within 15 to 21 hours (typically less than a day). Thus, better performances are obtained by those different and improved techniques are shown using well-known classification datasets.

Keywords: Artificial Neural Networks, Convolutional Neural Networks, Deep Learning, Neural Network Training, CIFAR-10, MNIST.

ÖZET

İLERİ YÖNTEMLERLE GELİŞTİRİLMİŞ YAPAY SİNİR AĞI EĞİTİMİ

Burak Çatalbaş
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Ömer Morgül
Eylül 2018

Yapay Sinir Ağları (YSA) sınıflandırma, kümelendirme vb. farklı makine öğrenmesi görevleri için kullanılmaktadır. Bu ağlar günlük hayatımızda önemli görevler yapıyor ve yeni hizmetler sunmaktadırlar. Bu ağların öğrenme kapasiteleri, artan hesaplama gücü ve veri miktarıyla, 2000li yıllardan beri önemli ölçüde ivmelenmektedir. Bu nedenle, bu ağlar üzerinde yapılan araştırmalar Derin Öğrenme ismiyle yeniden adlandırılmış ve, sadece sinir ağlarında değil Makine Öğrenmesi disiplini için de, önemli bir araştırma sahası olarak doğmuştur. Böylesi önemli bir araştırma alanı için, bu ağların eğitiminde kullanılan teknikler daha başarılı sonuçlar için anahtar olarak görülebilir. Bu çalışmada, bu eğitim prosedürünün her kısmı farklı ve geliştirilmiş – kimi zaman yeni – teknikler siyah-beyaz ve renkli imaj veri setlerini sınıflandıran evrişimsel sinir ağları üzerinde araştırılmıştır. İleri yöntemler, He-budanmış Gauss ön değer atama gibi literatürde var olanları içermiştir. Buna ek olarak, bizim literatüre katkımız olan SinAdaMax iyileştiricisi, Dominant Olarak Üstel ve Doğrusal Birim (ing. DELU), He-budanmış Laplasyen ve Maksimum-Bölütleme katmanları için Piramit Yaklaşımı gibileri de içermiştir. Bu tezin bölümlerinde, başarı oranları bu ileri yöntemlerin - özellikle bizim katkımız olan DELU ve SinAdaMax iyileştirilmiş metotlarının - kümülatif olarak eklenmesiyle arttırılmıştır.

Sonuç olarak, farklı veri setleri için başarı oranı eşikleri - bu yöntemlerle geliştirilmiş ve önemli test başarı artışlarına ulaşmış - temel evrişimsel sinir ağları ile 15-21 saat içinde (tipik olarak bir günden daha az sürede) karşılanmıştır. Böylece, bu farklı ve ileri yöntemlerle elde edilmiş daha iyi performanslar, tanınmış sınıflandırma veri setleri kullanılarak gösterilmiştir.

Anahtar sözcükler: Yapay Sinir Ağları, Evrişimsel Sinir Ağları, Derin Öğrenme, Sinir Ağı Eğitimi, CIFAR-10, MNIST.

Acknowledgement

My Master's Thesis program became a two-year adventure in which I enjoyed the support of my friends, teachers and my precious family. In here, I would like to thank them as I can.

My first thanking goes to my supervisor Prof. Dr. Ömer Morgül for his trust and support, in addition to his invaluable guidance throughout my academic career. Without all these, it would not be possible for me to constitute this thesis. I also thank to Assoc. Prof. Süleyman Serdar Kozat and Asst. Prof. Ramazan Gökberk Cinbiş for approving my work, paving the way for qualification as thesis.

Another appreciation goes to my friends from Bilkent. During my undergraduate and graduate programs, I always relied on their friendship and support to be successful in my academic and personal life. I want to thank all of them.

I also thank to staff of our department, namely Mürüvet Parlakay, Ergün Hırlakoğlu, Onur Bostancı, and Ufuk Tufan for their kind understanding and charitable behaviors during my undergraduate and graduate career.

I appreciate the financial support given by Scientific and Technological Research Council of Turkey (TÜBİTAK). I was supported by 2210-A program for my graduate study.

I appreciate NVIDIA Corporation for their donation of NVIDIA Titan Xp GPU Card and also their software support, which was a major component of my working environment which I used to constitute my thesis work.

My last thank-you will go to my irreplacable ones. I would like to thank my mother Zehra Çatalbaş, my father Sezai Çatalbaş, and my brother Bahadır Çatalbaş for their moral and material support, from the beginning of my life. Without any one of them, I could not be here at all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) has expanded to every aspect of our lives with the rapid developments of electronical technologies took place, such as mobile phones etc. As of today, the trailblazer of the machine learning field is Artificial Neural Networks with applications for tasks such as classification [1], data generation [2], even locomotion [3] etc. With these applications surround our daily lives, any improvement of neural network structure has a potential of high quality research.

With this motivation, we suggest some improvements in various aspects of neural networks in this thesis. From initialization of network parameters to modifying the well-known activation functions, our work focused on methodical improvements in each part of artificial neural networks. Results are obtained from convolutional neural network structures, but the improvements are capable of being applied to the most of other neural network types as well.

Results are taken with well-known datasets CIFAR-10 and MNIST, with mostly similar networks which are specialized to classify image samples. Comparing to current state-of-the-art results with our percentages, the success of our methodical improvements are demonstrated. For starting to present the thesis work, however, a brief introduction to history of artificial neural networks is required.

First artificial neural networks, inspired by the biological neuron structures, was able to be formed in the real world thanks to scientific improvements after 1940's [4]. After standard percepton structure is proposed by McCulloch and Pitts, ADALINE (then MADALINE) structures are proposed by Widrow and Hoff, and their employment to the field started quickly afterwards, around 1960's [4]. Convolutional neural networks are proposed in 1980s with a different name [5], but failed to be successful due to lack of computational power at those times. Instead, feedforward neural networks started to dominate the neural network field; especially after the proof of the Universal Approximation Theorem of a Multi-Layer Feedforward Neural Networks' ability to approximate any arbitrary function [6]. During the 2000's, some other machine learning methods such as Support Vector Machines (SVMs) started to take over the classification field with higher accuracies, however neural networks were about to gain from increasing computational power and expanding data amount throughout the globe. The revival of convolutional neural networks, with support of significant techniques such as Dropout and L-norm regularization, caused the neural networks the recapture the flag of machine learning after a short period. Not only surpassing the current state-of-the-art results, but also introducing new phenomenons such as Generative Networks made the artificial neural networks the trailblazer of the machine learning, and an inseparable part of Artifical Intelligence (AI) research. Currently, convolutional, recurrent and generative neural networks dominate the machine learning and pattern recognition areas of the engineering and science, used in many different qualitative and quantitative sciences and our daily lives.

The neuron models used in the field of neural networks can be different variants, but a standard neuron model used in this thesis, which is the most common structure, can be described as follows: Multiple input values (x) are one-by-one multiplied by different weight values (w), and then summed up for being the input value (v) of the activation function (f), which will produce the output value of the neuron (o). The graphical description of this structure is shown in Figure 1.1, and it is used in every neural network in this thesis work as the basic neuron model. After that, feedforward layers are displayed as an example, using a full feedforward neural network illustration in Figure 1.2 from work [7].

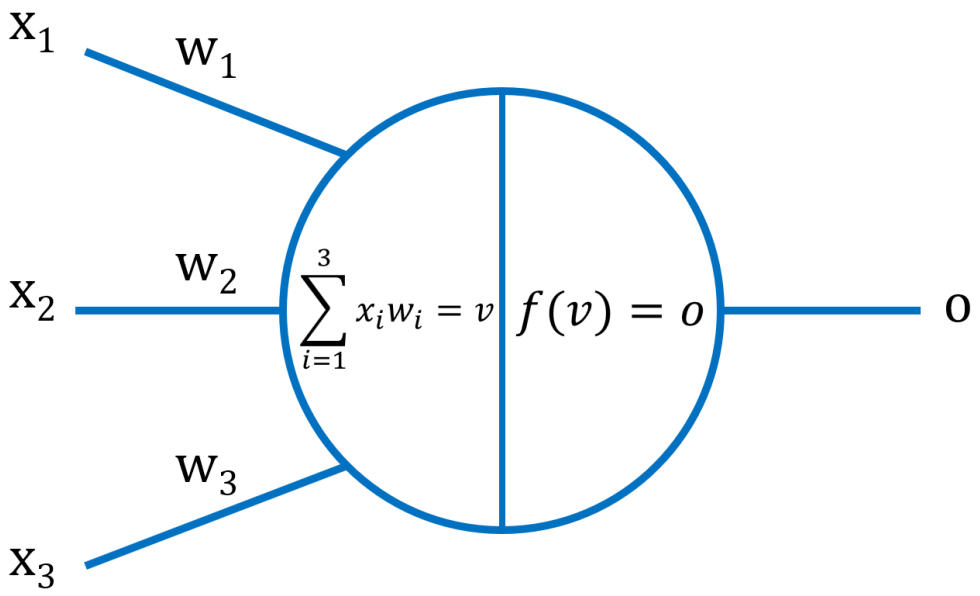

Figure 1.1: A generic neuron used in the thesis is displayed.

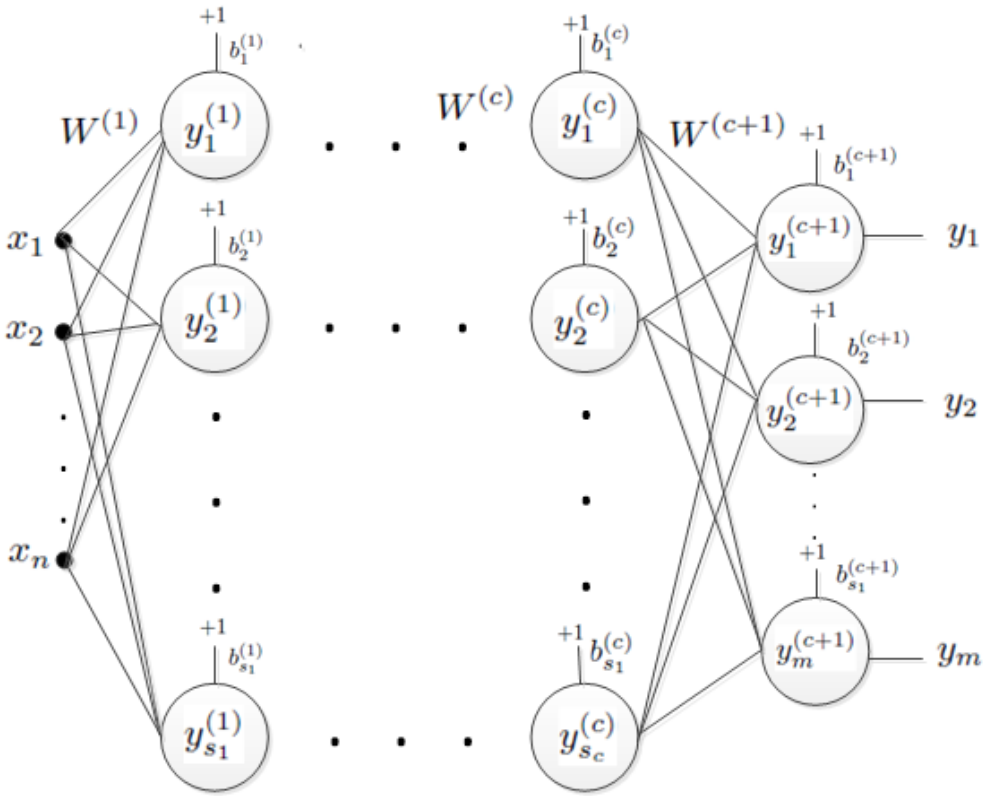

Figure 1.2: Example feedforward layer structure is illustrated.

Feedforward layers of the artificial neural networks are consisting of multiple neurons which are identical in itself. Input, internal of output layers are capable of transforming input values and propagating them to generate output values, for a specific layer. This happens by taking the input vector (X) and weight matrix (W, consisting of several weight vectors), making a matrix multiplication (*) and after the activation function (f) resulting an interior output vector (Y), which goes to the next layer as its input vector. This process generally repeats itself until the last layer of the network which produces, by the output activation function (f_o), the final output vector (O). The mathematical explanation of this procedure is given in (1.1).

$$f(X * W^1) = Y^1, \quad f(Y^1 * W^2) = Y^2, \quad ... \quad , \quad f_o(Y^{last} * W^{last}) = O \tag{1.1}$$

In addition, to train those layers, final output values (O) are compared with desired output values (D) and the difference in propagated back to these layers, and this learning technique is called as "Backpropagation". It modifies the weight parameters of the layer neurons, and teaches them to learn patterns feeded to neural network. This also means that the newer parameters will make the loss function smaller after backpropagation. The algorithm which generally explains this procedure is given in Algorithm 1.1, which is partially taken and also highly influenced from the work in [8]. In addition to feedforward layers, convolutional layers are also able to learn with backpropagation. Although convolutional layers consists of mostly smaller, square matrices that also contain modifiable weights, learning by backpropagation is still appliable to them. This time instead of the forward propagation-backpropagation reversal, convolution is reversed for a similar backpropagation process. Lastly, Max-Pooling layers do not contain any modifiable parameters as they take maximum values of some input (image or output of convolutional layers) regions - 2x2, 3x3 etc. - and constitute outputs: the distilled representations of original images or inputs. Briefly, those are the neural network layers which can be seen in place in Figure 1.3 with their specialties, which are used in the thesis work with below algorithm.

***Algorithm* 1.1 :** Generic backpropagation algorithm.

repeat
  for For every pattern (X) in the training set **do**
    Present the pattern to the network
    for Each layer (W) in the network **do**
      for Every node (w_i) in the layer **do**
        1. Calculate the weight sum of the inputs to the node
        2. Add the threshold (b_i) to the sum
        3. Calculate the activation for the node ($f(x_i w_i + b_i)$)
      end for
    end for
    for Every node in the output layer (W^{last}) **do**
      Calculate the error signal (D - O)
    end for
    Calculate the Loss Function ($\boldsymbol{\mathcal{L}}(D, O)$)
    for All hidden layers (Ws) **do**
      for Every node (w_i) in the layer **do**
        1. Calculate the node's signal error ($\frac{\partial \boldsymbol{\mathcal{L}}}{\partial w_{ji}}$)
        2. Update each node's weight in network ($w_{ji} = w_{ji} - \mu \frac{\partial \boldsymbol{\mathcal{L}}}{\partial w_{ji}}$)
      end for
    end for
    Show the Error Function ($E = \boldsymbol{\mathcal{L}}$)
  end for
until (maximum number of iterations are higher than specified)

To furthermore enhance these successful results obtained and continue to the journey of neural networks, this thesis aimed on improved artificial neural network training with advanced methods. For this purpose, primarily a convolutional neural network is proposed for CIFAR-10 dataset. Taken from the work [9], this network is applied to training set by constituting a validation set from within, and modifications of different network parts are experimented with this configuration. The network scheme can be seen in Figure 1.3 and its details are given at Structure 1.1, which obtains a success rate around 85-86% on the CIFAR-10 dataset.

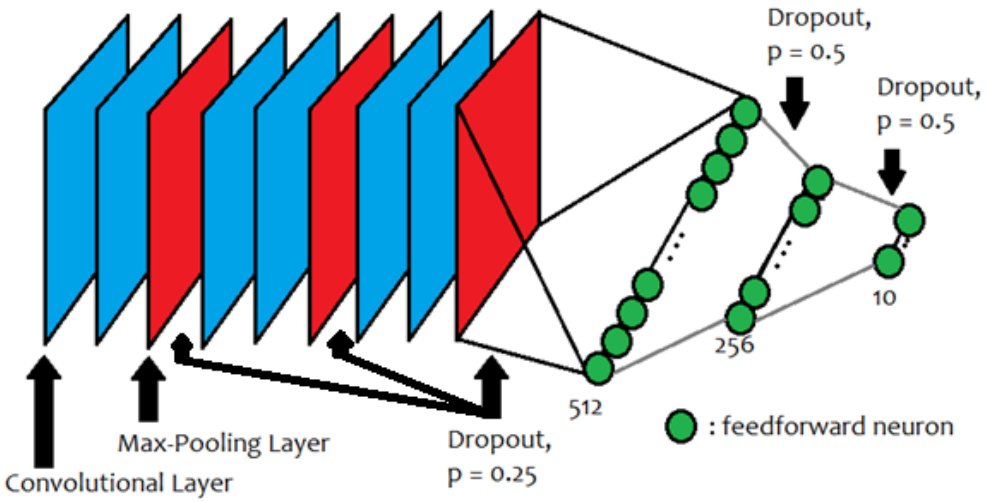

Figure 1.3: The convolutional neural network's general structure is displayed.

***Structure* 1.1 :** The main convolutional neural network structure used.

Input Layer: 32x32x3
Convolutional Layer 1: 3x3x48 (Padding with Keeping the Size Same)
Convolutional Layer 2: 3x3x48
Max-Pool Layer 1: 2x2 Pool Size
Dropout Layer with p = 0.25
Convolutional Layer 3: 3x3x96 (Padding with Keeping the Size Same)
Convolutional Layer 4: 3x3x96
Max-Pool Layer 2: 2x2 Pool Size
Dropout Layer with p = 0.25
Convolutional Layer 5: 3x3x192 (Padding with Keeping the Size Same)
Convolutional Layer 6: 3x3x192
Max-Pool Layer 3: 2x2 Pool Size
Dropout Layer with p = 0.25
(Reshaping By 'Flatten' Command Here, For Feedforward Layers)
Feedforward Layer 1: 512 Neurons
Dropout Layer with p = 0.5
Feedforward Layer 2: 256 Neurons
Dropout Layer with p = 0.5
Feedforward Layer 3: 10 Neurons

This network structure are modified for employing advanced methods throughout the thesis work, to reach over 90% test success rate for the CIFAR-10 dataset, and to obtain highest test success rate possible in MNIST dataset. Although these modifications, some important aspects of the neural network are kept constant throughout the thesis work. Network input size, size of convolutional and max-pooling layers and number of neurons in feedforward layers are given in this structure. Regarding the differences; only for the MNIST dataset, input size shrinks to 28x28x1 from 32x32x3 volume. In addition to that adaptation, given Dropout probabilities are modified in the Chapter 7 and set as p = 0.375.

Networks with this structure and modified components are compared by obtaining 10 trials for each configuration. Detailed explanation is given in Algorithm 1.2, where each step is explained.

***Algorithm* 1.2 :** The process of neural network comparison.

1. 10 independent, unmodified neural networks are trained
2. Their mean and maximum successes on validation data are found
3. Modification is done on the neural network structure
4. 10 independent but modified neural networks are trained
5. Their mean and maximum successes on validation data are found
6. Results of **2** and **5** are compared in the same table
7. Their comparison shows the success of the modification, record it
8. Change the modification and return to **3**

The rest of this thesis includes primarily this network structure with modifications which are explained in each chapter. Starting from a validation success rate around 80% for 40000 samples training and 10000 samples validation, this percentage is step-by-step increased by the different, improved and sometimes new methods applied to the network structure.

The main contributions of the thesis to literature mainly 4 new methods:

- A new initialization method, Laplacian

- A new optimizer algorithm, SinAdaMax

- A new activation function, Dominantly Exponential Linear Unit (DELU)

- A new approach for Max-Pool layers, Pyramid Approach

Especially SinAdaMax, DELU and Pyramid Approach increased the success rates of the neural network significantly, and most of these methods are applicable to other types of neural networks (feedforward, adversarial etc.) as well.

After this first chapter, the thesis is organized as follows: In second chapter we consider initialization methods for neural networks with different parameter initialization techniques. In third chapter, we consider loss functions of neural networks with different function candidates. In fourth chapter, we consider optimizer algorithms for neural networks with different learning algorithms. In fifth chapter, activation functions are investigated with different mathematical functions proposed. In sixth chapter, regularizers are considered with different combinations suggested. In seventh chapter, network layers are considered with different alternatives proposed. In eighth chapter, different data preprocessing and data augmentation techniques are experimented with convolutional neural networks. In ninth chapter, learning conditions of neural networks are modified and different approaches are tried. In tenth chapter, we present the results obtained with networks which use these advanced techniques to classify MNIST dataset and discuss possible implementations to the networks which classify the ImageNet dataset. Finally, we give some concluding remarks in the last chapter, Conclusion and Future Works.

Chapter 2

Initialization Methods

Every artificial neural network consists of network parameters. These parameters are modified during the training process in different ways. However, before starting to train the neural network, network parameters should be initialized first. These initial network parameters are important as they set the starting position of the network in the error surface throughout the training.

From the beginning of artificial neural networks, different initializations are used. The first networks founded in 1960s employed integers or rational numbers as the network parameters, such as MADALINE, Perceptron networks etc. Later, Gaussian or Uniform distributed random numbers are used for initializing the network parameters, as they were more successful. Finally, after the millenium, network-based methods are proposed such as Xavier [10] and He [11] initialization. Currently, this 3rd generation methods are widely in use at recent feedforward, convolutional and other types of neural networks.

In this chapter, different initialization methods are employed for convolutional neural networks with same configuration introduced before. Namely Gaussian, Uniform and Laplacian initialization methods are used with Xavier and Glorot truncation techniques, in addition to their untruncated versions. This chapter will start with these untruncated probability distributions and their results.

The neural networks in this chapter worked with following attributes only (except different initialization methods): Adam optimizer (a learning algorithm that uses backpropagation to modify network parameters, see [12]), Categorical Cross-Entropy loss (a mathematical formula that calculates loss values from desired and actual output values to be used in backpropagation, see example at [13]), ReLU activation function (a function that is used as seen in Chapter 1, for ReLU specifically see [14]) except Softmax output layer (see [15]), Mini-Batch Learning (a technique calculates losses per each batch and modifies the network after the batch, see [16]) with 128 samples in each batch, and training is completed in 200 epochs. Training is done with network at Figure 1.3 (Structure 1.1) using first 4 data batches of CIFAR-10 dataset, and validation is done with remaining fifth data batch (each data batch contains 10000 samples and shuffled randomly). From 10 trials conducted with this configuration, mean and maximum success rates for validation are obtained for comparison of success rates and seeing the change of results after switching to different initializations.

2.1 Untruncated Initialization Methods

In this part, network parameters are taken from standard probability distribution functions (pdfs) which are predetermined and same for all network layers. Different distribution functions with different specifications are constituted for this purpose. Function shapes are also important as they set the size of network parameters. We will first consider Uniform distribution and its results.

2.1.1 Initialization with Uniform Probability Distribution Function

Uniform probability distribution function (pdf) was widely used during the 19th and 20th century, but their usage in neural network field intensified during 1990s. This probability distribution, without any truncation, is tried with convolutional

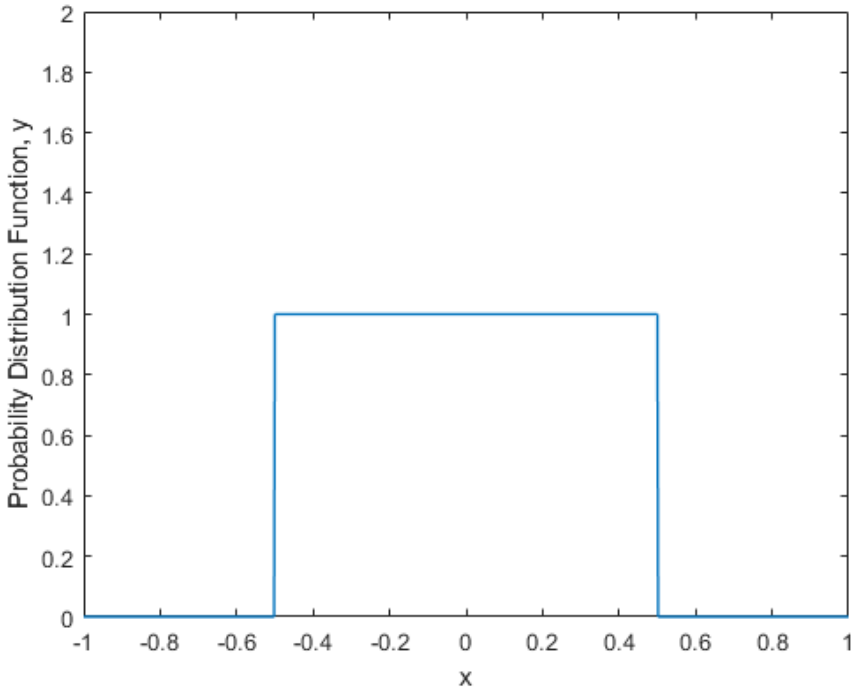

Figure 2.1: Uniform pdf used to generate samples is shown.

neural network and results are obtained. Mean is taken as zero and the limits of uniform pdf is taken as ± 0.05, makes the non-zero value of the distribution one, as seen in the Figure 2.1 and below formula, where the possible output of the random variable with this pdf, and y is the respective probability for corresponding x value.

$$y = \begin{cases} 1 & \text{if } |x| < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

Table 2.1: Results obtained from 10 trials for Uniform distribution.

Initialization Type	**Uniform**
Mean Success	79.36%
Maximum Success	80.02%

As seen from above, this initialization method yields a success rate close to 80%

for convolutional neural network. However, these rates are bounded by another well-known initialization technique: Gaussian.

2.1.2 Initialization with Gaussian Probability Distribution Function

Gaussian probability distribution, as known as Normal distribution, is well-known in probability theory. It is also very frequently used to set the neural network parameter values. Again without any truncation, a Gaussian distributed random variable with zero mean and 0.05 standard deviation is used for generating initial network parameters. Figure 2.2 shows the shape of used probability distribution function. Mathematical formula of this pdf is given below, where x is the possible output of the random variable with this pdf and y is the respective probability for corresponding x value.

$$y = e^{-x^2/2\sigma^2} / \left(\sigma\sqrt{2\pi}\right) \tag{2.2}$$

Table 2.2: Results obtained from 10 trials for Gaussian distribution.

Initialization Type	**Gaussian**
Mean Success	80.70%
Maximum Success	81.37%

As seen here, samples drawn from Gaussian probability distribution works better than uniformly distributed ones as convolutional neural network parameters. Different standard deviation values such as 0.025 and 0.01 are used but their results were not better than 0.05 standard deviation value. Another initialization type which gives similar results is our new suggestion, Laplacian initialization which is inspired from the change of shape, similar to transformation of Uniform-to-Gaussian distributions.

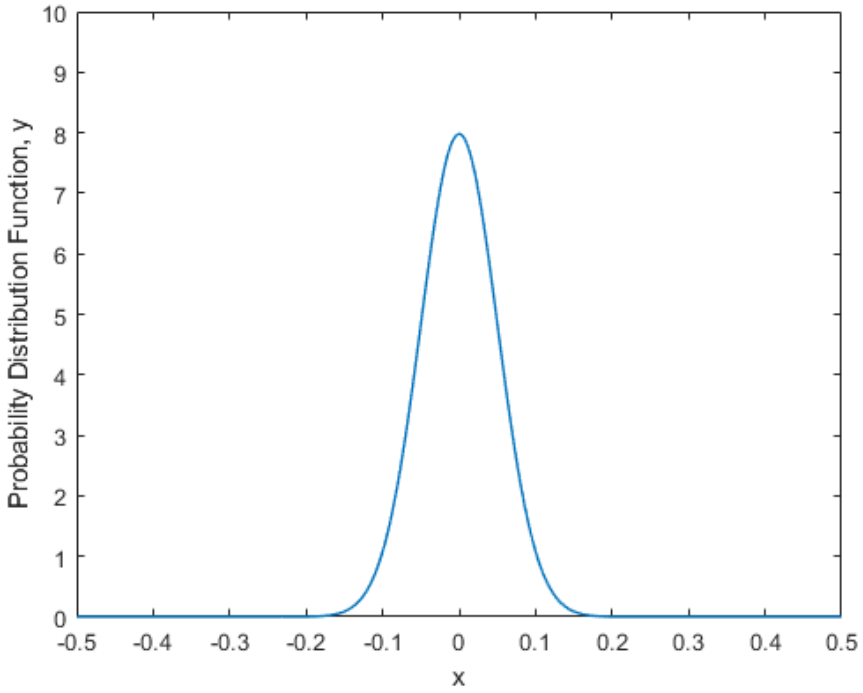

Figure 2.2: Gaussian pdf used to generate samples is shown.

2.1.3 Initialization with Laplacian Probability Distribution Function

Laplacian function is founded during 18th century and used for some regression analysis techniques, statistical databases, even for hydrology. In this context, Laplacian probability distribution is employed for generating random numbers as the network parameters. This time zero mean and 0.05 scale value (b) is used in this distribution without any truncation. Scale value is equal to standard deviation divided by $\sqrt{2}$ (i.e. $b = \sigma/\sqrt{2}$), for the case of Laplacian probability distribution function. Figure 2.3 shows the shape of used probability distribution function. Mathematical formula of this pdf is given below, where x is the possible output of random variable with this pdf and y is the respective probability for corresponding x value.

$$y = e^{-|x|/b}/(2b) \tag{2.3}$$

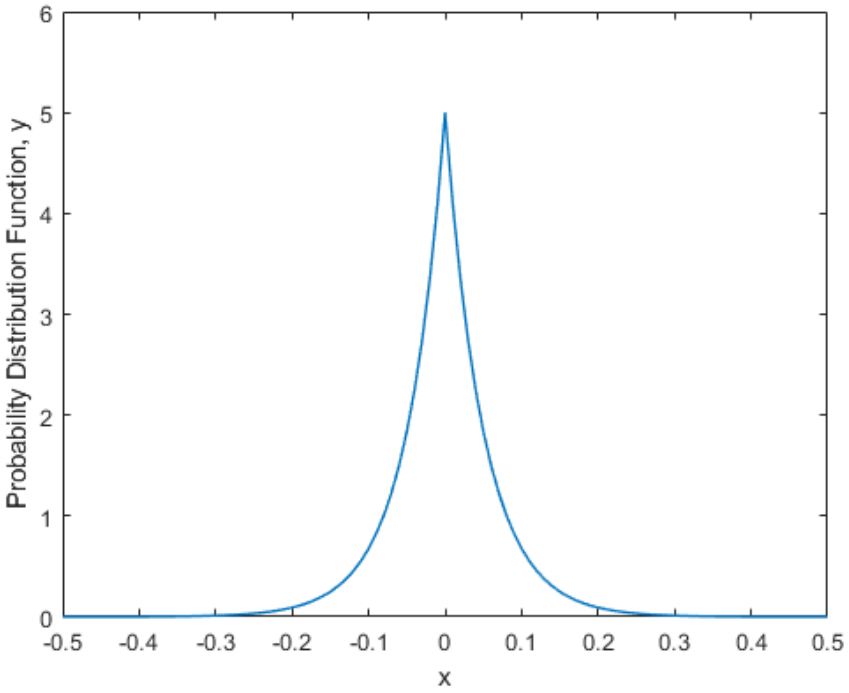

Figure 2.3: Laplacian pdf used to generate samples is shown.

Table 2.3: Results obtained from 10 trials for Gaussian distribution.

Initialization Type	**Laplacian**
Mean Success	80.51%
Maximum Success	81.59%

Results are pretty close to Gaussian ones, and the maximum success rate obtained from 10 trials exceed the maximum one obtained by both Gaussian and Uniform initializers. However, as the mean success rate is lower than Gaussian initializer, next section will contain both Gaussian and Laplacian-based probability distributions - using network parameter initializers as can be seen from Table 2.4 completely.

Table 2.4: Results obtained from 10 trials for all distributions.

Initialization Type	**Uniform**	**Gaussian**	**Laplacian**
Mean Success	79.36%	80.70%	80.51%
Maximum Success	80.02%	81.37%	81.59%

2.2 Initialization Methods with Xavier Truncation

In this part, network parameters are taken from truncated probability distributions which are shaped according to layer specialties. Dependent on input and output neuron numbers of a layer, the probability distribution is truncated by cutting off the tails from calculated locations of the probability distribution function, as limits. In this way, layer-based pdfs are generated which are used to generate parameter value specific to each layer. Xavier Glorot and his colleague proposed using input and output neuron numbers to set this limit value [10], with different formulas for each separate probability distribution function. This section will start with Xavier-truncated (a.k.a. Glorot-truncated) Uniform pdfs.

2.2.1 Initialization with Xavier-Truncated Uniform PDF

The uniform probability distribution function is by nature bounded with limits. Contrary to previous uniform distribution, however, these limits will be dependent on the input and output neuron numbers for each network layer. The formula used for this purpose is given below, where x is the possible output of the random variable with this pdf and y is the respective probability for corresponding x value (n_i for number of input and n_o for number of output neurons, for each layer):

$$y = \begin{cases} 1/2\sqrt{(6/(n_i + n_o))} & \text{if } |x| < \sqrt{(6/(n_i + n_o))} \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

The shape of this pdf is very similar to the previous uniformly distributed function, where only the limits are different. Therefore, figure of the pdf is not needed in this part. The results obtained with same configuration as before with only initialization technique is differing, and shown in Table 2.5. It can be observed from below.

Table 2.5: Results obtained from 10 trials for Xavier Uniform distribution.

Initialization Type (All Xavier)	**Uniform**
Mean Success	79.62%
Maximum Success	81.78%

As seen from above, this initialization method gives a mean success rate again close to 80% for convolutional neural network. Indeed, maximum success rate is the best acquired until this point also. However, mean success rate is lower than untruncated Gaussian and Laplacian initializations. We'll proceed to Xavier-truncated Gaussian initialization and the results obtained with this initialization technique.

2.2.2 Initialization with Xavier-Truncated Gaussian PDF

In this part, a Gaussian pdf is truncated from both sides by a limit defined with input and output neuron numbers. Its details are given below where x the possible output of the random variable with this pdf, and y is the respective probability for corresponding x value, in which n_i is used for number of input and n_o is used for number of output neurons. Example pdf shape is also shown in the Figure 2.5, where cutting of edges and enlarging of remaining curve are evident (figure is taken according to limits equal to -0.05 and +0.05, the standard deviation).

$$y = \begin{cases} (e^{-x^2/2\sigma^2}/(\sigma\sqrt{2\pi}))/(P(|x| < \sqrt{(2/(n_i+n_o))})) & |x| < \sqrt{(2/(n_i+n_o))}. \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

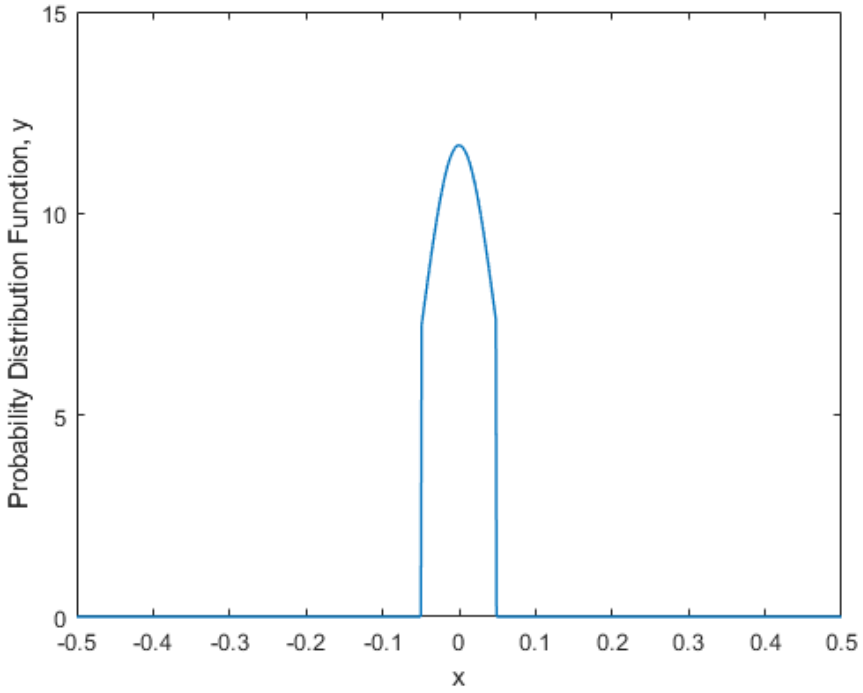

Figure 2.4: Xavier-truncated Gaussian pdf used to generate samples is shown.

With this truncation technique, mean and maximum success rates are taken for 10 trials of the same convolutional neural network as described before. These results are shown in the Table 2.6 alongside the Xavier Uniform ones.

Table 2.6: Results obtained from 10 trials for Xavier-truncated distributions.

Initialization Type (All Xavier)	**Uniform**	**Gaussian**
Mean Success	79.62%	78.94%
Maximum Success	81.78%	81.07%

As seen here, mean and maximum success rates are lower than uniform ones. In addition, it is the lowest rates obtained in this section. So long as Xavier-truncated pdfs performed worse than untruncated versions, this chapter will continue with a different truncation method.

2.3 Initialization Methods with He Truncation

In this method, probability distribution functions are calculated again to generate parameters specific to each layer, similar to Xavier. Kaiming He and his colleagues proposed using only input neuron numbers to set limit values [11], with different formulas for each separate pdf. We'll start with He-truncated Uniform.

2.3.1 Initialization with He-Truncated Uniform PDF

Similar to Xavier-truncated uniform probability distribution, these limits will be dependent on the neuron numbers for each network layer. The formula used for this purpose is given below where x the possible output of the random variable with this pdf, and y is the respective probability for corresponding x value, which contains only input neuron numbers (n_i) for each specific layer:

$$y = \begin{cases} 1 \;/\; \left(2\sqrt{(6/(n_i))}\right) & \text{if } |x| < \sqrt{(6/(n_i))} \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

The shape of this pdf is again very similar to the previous uniformly distributed function, where only the limits are different. So, the figure of the pdf is not needed in this part. The results obtained with same configuration as before with only initialization technique is differing, and shown in Table 2.7. It can be observed from below.

Table 2.7: Results obtained from 10 trials for He Uniform distribution.

Initialization Type (All He)	**Uniform**
Mean Success	81.06%
Maximum Success	81.74%

As seen from above, this initialization method gives the best results until this point. Mean success rate exceed the 81% threshold and maximum success

rate is close to 82%, performing well above the Xavier truncation and better than untruncated probability distribution functions. Thus, other initialization methods such as Gaussian and Laplacian can be seen as promising with this truncation method also. Therefore, this section will continue with He-truncated Gaussian initialization.

2.3.2 Initialization with He-Truncated Gaussian PDF

In this part, a Gaussian probability distribution function is truncated from both sides by a limit defined with input neuron numbers. Its details are given below, in which n_i is used for number of input neurons. Example pdf shape is also demostrated in this figure, where cutting of edges and enlarging of remaining curve are evident (figure is taken according to a limit equal to -0.05 and +0.05, the standard deviation). Figure is exactly same with Figure 2.4.

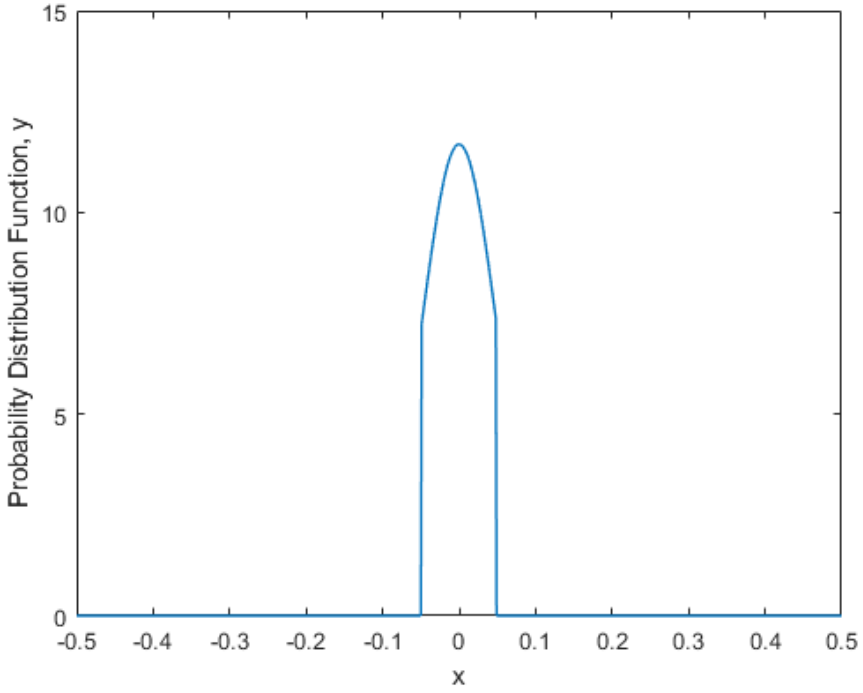

Figure 2.5: He-truncated Gaussian pdf used to generate samples is shown.

The formula used to generate the He-truncated pdfs is given below, where x

Table 2.8: Results obtained from 10 trials for He Gaussian distribution.

Initialization Type (All He)	**Gaussian**
Mean Success	81.11%
Maximum Success	82.36%

is the possible output of the random variable with this pdf, y is the respective probability for corresponding x value and n_i for number of input neurons for each specific layer. The results taken with this probability distribution is given in the above table.

$$y = \begin{cases} (e^{-x^2/2\sigma^2}/(\sigma\sqrt{2\pi}))/(P(\ |x| < \sqrt{(2/(n_i))}\ )) & \text{if } |x| < \sqrt{(2/(n_i))} \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

However, it is important to note that shape of the Gaussian changes for each specific layer dependent on the size of input neuron numbers.

As seen from here, He-truncated Gaussian initialization performs better than He-truncated Uniform initialization, making itself best initialization obtained in this chapter. From the 10 trials conducted, mean and maximum success rates are the best acquired until this point. Therefore, this initialization technique is chosen as the one that will be carried to other chapters as the primary initialization method of the convolutional neural network.

2.3.3 Initialization with He-Truncated Laplacian PDF

Laplacian probability distribution is employed for generating random numbers as the network parameters. Again, zero mean and 0.05 scale value (b) is used in this distribution with He truncation (scale value is equal to standard deviation divided by $\sqrt{2}$; $b = \sigma/\sqrt{2}$). Figure 2.6 in the next page shows the shape of used probability distribution function. As can be seen from there, example truncation

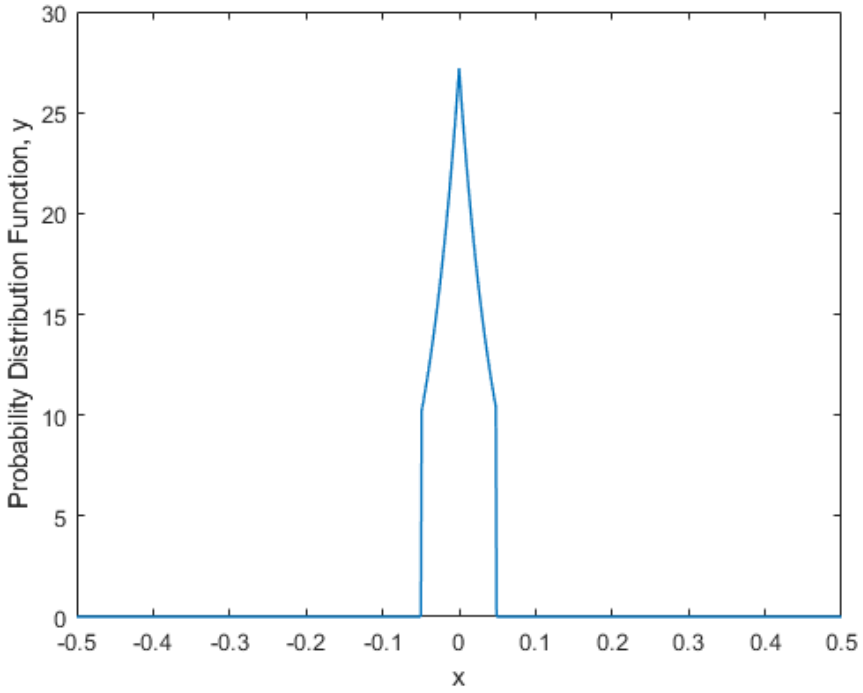

Figure 2.6: Laplacian pdf used to generate samples is shown.

occurs at -0.05 and +0.05 locations. For the Laplacian, truncation constant (TC) is selected as 12 for this experiment. Thus $\sqrt{12/n_i}$ is chosen as limit formula (n_i is the input neuron number). Detailed mathematical formula of this pdf is given below, where x is the possible output of the random variable with this pdf, y is the respective probability for corresponding x value:

$$y = \begin{cases} e^{-|x|/b} / \left(2bP(|x| < \sqrt{(12/(n_i))})\right) & \text{if } |x| < \sqrt{(12/(n_i))} \\ 0 & \text{otherwise.} \end{cases} \tag{2.8}$$

Table 2.9: Results obtained from 10 trials for He-truncated Laplacian pdf.

Initialization Type	**Laplacian**
Mean Success	80.70%
Maximum Success	81.96%

Results can be seen from the Table 2.9. They are pretty close to Gaussian

ones, but their mean and maximum success rates are not exceeded by Laplacian. However, as the second best results are taken with He-truncated Laplacian initialization method, therefore it can also be considered as an alternative initialization method for the cases where an increase of the success could be obtained. In Table 2.10, all He-truncated initialization methods' success rates are present for comparison.

Table 2.10: Results obtained from 10 trials for all He-truncated pdfs.

Initialization Type	**Uniform**	**Gaussian**	**Laplacian**
Mean Success	81.06%	**81.11**%	80.70%
Maximum Success	81.74%	**82.36**%	81.96%

To sum up this chapter, different initialization methods are tried with the convolutional neural network which classifies CIFAR-10 dataset, and according to obtained success rates best initialization methods are chosen for increasing the success of modified convolutional neural network. The results based on Laplacian initialization has been reported in [17]. As a future work, Laplacian methods can be applied with Xavier initialization and results can be obtained as a research. In result, He-truncated Gaussian (primarily) and Laplacian (secondarily) are selected in this chapter.

Chapter 3

Loss Functions

Artificial neural networks learn after their mistakes. The loss functions they use, therefore, are crucial for their learning. Without specifying a loss function, it is impossible to train any neural network. According to mathematical formulas used as loss functions, neural network moves in the error surface throughout the optimization process.

From the beginning of artificial neural networks, different loss functions are used. The first neural networks simply used the difference between actual output and desired output as the loss value [4] (primarily integers) and trained themselves accordingly. Later, more complex ones are generated which give better results such as Mean Squared Error, Categorical Cross-Entropy, Kullback-Leibler Divergence, Poisson loss etc. which are not necessarily integer, but generally closer to zero value also.

However, there are few numbers of well-known and good-performing loss functions in use. During the working process which constituted this chapter, another loss functions such as Categorical Hinge, LogCosh, Cosine Proximity, Mean Absolute Error and Mean Absolute Percentage Error are tried but not performed well with the CIFAR-10 convolutional neural network. This chapter will start with a relatively classic one: Mean Squared Error.

The neural networks in this chapter worked with following attributes only (except different loss metrics): Adam optimizer (see [12]), He-truncated Gaussian and He-truncated Laplacian initializations (see Chapter 2), ReLU activation function (see [14]) except Softmax output layer (see [15]), Mini-Batch Learning (see [16]) with 128 samples in each batch, and training is completed in 150 epochs for proceeding faster. Training is done with network at Figure 1.3 (Structure 1.1) using with first 4 data batches of CIFAR-10 dataset, and validation is done with remaining fifth data batch (each data batch contains 10000 samples and shuffled randomly). From 10 trials conducted with this configuration, mean and maximum success rates for validation are obtained for comparison of success rates and seeing the change of results after switching to different loss functions.

3.1 Mean Squared Error Loss Function

Mean Squared Error (MSE) is relatively older than other loss metrics, but it is frequently used in feedforward and convolutional neural networks. Here, we compare the desired and actual output values of the neural network, and using this comparison we calculate the loss metric which is used at the example backpropagation at output layer (weight modifications), which is given below (μ is the learning rate constant, w_{ji} is the output neuron weights (from jth neuron at previous layer to ith neuron of output layer), $\boldsymbol{\mathcal{L}}$ - equivalently E, error function - is the loss function in terms of input value v and output value of the neuron o). Throughout this chapter, we will use different loss functions for an equation that can be resembled by the following part of Gradient Descent algorithm, essentially.

$$w_{ji} = w_{ji} - \mu \frac{\partial \boldsymbol{\mathcal{L}}}{\partial w_{ji}} \tag{3.1}$$

The basic formulation of this loss metric is given in the next page, as used in [18], where $\hat{y}$ is the actual and y is the desired output for each i^{th} output neuron (total number of output neurons is n):

$$\boldsymbol{\mathcal{L}} = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{3.2}$$

With this formulation and He-truncated Laplacian initialization, several results are obtained. However, mean success rate is resulted around 75% and maximum success rate did not exceed 80% threshold with CIFAR-10 convolutional neural network. Because of these results, trainings with this loss metric is interrupted and used loss metric is switched to Kullback-Leibler Divergence loss function.

3.2 Kullback-Leibler Divergence Loss Function

Kullback-Leibler Divergence (KL Divergence), as known as relative entropy, is used as loss function for neural network training comparably recently, when Mean Squared Error and some others are considered. In formula below, mathematical expression and its details are given (from work [19]), where $\hat{y}$ is actual and y is desired output for each i^{th} output neuron (total number of output neurons is n):

$$\boldsymbol{\mathcal{L}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{D}_{KL}(y^{(i)}||\hat{y}^{(i)}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\cdot\log(y^{(i)})\right)}_{\boldsymbol{entropy}}\underbrace{-\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\cdot\log(\hat{y}^{(i)})\right)}_{\boldsymbol{cross-entropy}} \tag{3.3}$$

With this loss function, again 50, 100 and 150 epochs of training are conducted for obtaining validation success rates respectively. Mean and maximum success rates are taken for these convolutional networks which are initialized with He-truncated Laplacian (Truncation Constant was taken as 12). The results are given in the Table 3.1.

As seen from the below results, mean and maximum success rates for convolutional neural network, which classifies CIFAR-10 dataset samples, performs better than Mean Squared Error loss-using neural networks. However, there are ones that perform better than this loss function.

Table 3.1: Results from 10 trials, with He-truncated pdf and KL Divergence.

Number of Epochs	**50 Epochs**	**100 Epochs**	**150 Epochs**
Mean Success	79.94%	80.52%	80.39%
Maximum Success	80.53%	81.13%	81.05%

3.3 Poisson Loss Function

Poisson loss function, created as a variation of Poisson distribution, can be used as a loss metric for neural networks similar to Mean Squared Error and KL Divergence loss. In the formula below, mathematical expression and its details are given (again from the work [19]), where $\hat{y}$ is the actual and y is the desired output for each i^{th} output neuron (total number of output neurons is n):

$$\boldsymbol{\mathcal{L}} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}^{(i)} - y^{(i)} \cdot \log(\hat{y}^{(i)})\right) \tag{3.4}$$

With this loss function, again 50, 100 and 150 epochs of training are conducted for obtaining validation success rates respectively. Mean and maximum success rates are taken for these convolutional networks which are initialized with He-truncated Laplacian (Truncation Constant was taken as 12). The results are given in Table 3.2.

Table 3.2: Results from 10 trials, with He-truncated pdf and Poisson loss.

Number of Epochs	**50 Epochs**	**100 Epochs**	**150 Epochs**
Mean Success	79.93%	80.46%	80.91%
Maximum Success	80.59%	81.09%	81.75%

As seen in Table 3.2, although some maximum success rates exceed the previous ones, the mean success rates obtained are not the best acquired in this chapter. Seeing a possible potential in this loss function, we modified the Poisson loss as seen in the (3.5) and generated a custom Poisson loss (where r is a tuning value, i.e. offset).

$$\boldsymbol{\mathcal{L}} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(\hat{y}^{(i)} - y^{(i)} \cdot \log(\hat{y}^{(i)})\right)\left(|\hat{y}^{(i)} - y^{(i)}| + r\right)\right) \tag{3.5}$$

Table 3.3 contains the results with this modified loss function. Tuning value is tuned as 0.25 at first but it performed badly, therefore an interval from 0.5 to 1 is scanned with 0.1 resolution to find best tuning value. Best performance is obtained with tuning value 0.9 and its results are given below.

Table 3.3: Results from 10 trials, with He-truncated pdf & custom Poisson loss.

Number of Epochs	**50 Epochs**	**100 Epochs**	**150 Epochs**
Mean Success	80.41%	80.62%	81.01%
Maximum Success	81.32%	81.36%	**81.89**%

These results exceed the original Poisson loss mean success rates and include maximum success rates close to 82%. However, in terms of mean success rate, it is still not the best ones in this chapter - the Categorical Cross-Entropy loss function.

3.4 Categorical Cross-Entropy Loss Function

Categorical Cross-Entropy (Categorical CE) which is given in (3.6) is tried for 10 class CIFAR-10 dataset's classification by convolutional neural network, which was described before, as the loss function. Formula used for mathematical calculation is given below from same source [19]. Again, where $\hat{y}$ is actual and y is desired output for each i^{th} output neuron (total number of output neurons is n):

$$\boldsymbol{\mathcal{L}} = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right] \tag{3.6}$$

With this loss function, results for 50, 100 and 150 epochs are taken, which can be seen in Table 3.4. Mean and maximum success rates align conveniently

compared to result of 200 epochs, although He-truncated Laplacian is common in all networks. Table 3.4 shows the results in detail:

Table 3.4: Results from 10 trials, with He-truncated pdf and Categorical CE.

Number of Epochs	**50 Epochs**	**100 Epochs**	**150 Epochs**
Mean Success	80.83%	**81.27**%	81.19%
Maximum Success	81.16%	81.56%	81.28%

Compared to previous Mean Squared Error's results, these are higher than those success rates and indeed, these are best amongst this chapter with He-truncated Laplacian initialization.

To sum up this chapter, different loss functions are tried with convolutional neural networks classifying CIFAR-10 dataset and not only the best amongst these loss functions are found, but also existing ones are attempted to be modified, and the custom Poisson loss became the second best loss function in this chapter. Because of these outcomes, Categorical Cross-Entropy loss function will be carried to other chapters as the primary loss function to be used in neural networks.

Chapter 4

Optimizer Algorithms

Every artificial neural network is supposed to go through a learning process. Optimizer algorithms modify the neural network parameters to make the network learn the training samples and explore the boundaries between classes, in case of classification tasks. In general, the neural network travels the error surface and looks for an optimal location with these algorithms. Therefore, these techniques are indivisible from neural networks' training and crucial for increasing their success rates.

From the beginning of artificial neural networks, different optimizer algorithms, as known as learning algorithms, are employed for different neural networks. The first networks -consisting of several neurons- used the output error value to basically compare and primarily alter the weights of the neurons to make them learn input-output relationship patterns. Afterwards, more complex techniques such as gradient descent are used as a base for modifying the neural network parameters. The learning rate is taken as a variable parameter in new techniques such as AdaGrad (see [20]) and Adam (see [12]) optimizers; while some of them started to store previous gradient values for each neural network parameter to learn more stably. A different one, AdaDelta (see [21]) optimizer, got rid of learning rate constants. Adam and its variant AdaMax (see [12]), however, hold

several momentum information of neural network parameters and used this information. In the other side, batch-based learning algorithms such as Limited Memory Broyden–Fletcher–Goldfarb–Shanno algorithm (shortly called L-BFGS, for more information see [22]) followed a different approach by computing the Hessian matrices from input samples and formulating update values for neural network parameters. Each optimizer algorithm has its difficulties and successes, and in this section we decided to use mini-batch based learning techniques rather than batch-based learning techniques such as L-BFGS.

In this chapter, different optimizer algorithms are tried with the convolutional neural network that classifies the CIFAR-10 dataset. During this chapter, several learning algorithms from Keras libraries are used but did not perform well: Stochastic gradient descent (classical gradient descent learning algorithm), rmsProp (see [23]), AdaGrad, AdaDelta, and Nadam (with Nesterov variant, for more information see [24]). So long as these ones perform badly and their mean success rate do not exceed 81% and in general stuck around 80%, Adam optimizer and AdaMax optimizer from Keras library is mainly used in this chapter. In addition to these, a custom optimizer algorithm is developed, by augmenting the AdaMax optimizer, and experimented in this part. The work in this chapter will start with Adam optimizer (after the review of basic learning techniques section below) and its results with the convolutional neural networks.

Neural networks in this chapter worked with following attributes only (except different optimizer algorithms): He-truncated Laplacian initialization (Truncation Constant is an improved one, set as 13.5; see Chp. 2) and He-truncated Gaussian initialization (at the end of this chapter, see Chp. 2), Categorical Cross-Entropy loss (see [13]), ReLU activation function (see [14]) except Softmax output layer (see [15]), Mini-Batch Learning (see [16]) with 128 samples in each batch, and training is completed in 150 or 300 epochs. Training is done with network at Figure 1.3 (Structure 1.1) using first 4 data batches of CIFAR-10 dataset, and validation is done with fifth data batch (each data batch contains 10000 samples and shuffled randomly). From 10 trials conducted with this configuration, mean and maximum success rates for validation are obtained for comparison of success rates and seeing the change of results after switching to different initializations.

4.1 Review of Basic Learning Techniques

Before starting to go into the details of Adam optimizer, it will be helpful to review the mathematical details of basic optimizers. The Stochastic Gradient Descent (SGD), is the first example to be examined in this manner. The algorithm that describes this process is given below. In this algorithm, θ is the parameter set which is used to minimize the loss function $\boldsymbol{\mathcal{L}}(\theta)$.

***Algorithm* 4.1 :** Stochastic Gradient Descent is shown as an algorithm.

1. Select the input pattern to the network (patterns, if mini-batch)
2. Send the input and forward-propagate it, store the produced values
3. Find out the values of the output neurons
4. Compared desired values and actual values of the network
5. Calculate the loss function $\boldsymbol{\mathcal{L}}(\theta)$
6. Take partial derivatives (gradients, g) wrt weights (parameter set, θ)

if 7. Momentum learning will be done **then**

Multiply them with learning rate (μ) and subtract them from parameter set (θ) with by a fraction of old parameter values (γv_{t-1})

else

Multiply them with learning constants and sum them with θ

end if

8. Return to the Step 1

Another type of optimizers are Adaptive ones. These optimizers adjust the learning (and sometimes learning rates) during the training period adaptively, where AdaGrad [20] constantly decreases the learning rate according to previously calculated gradients. AdaDelta [21], instead, uses a window of past gradients to calculate the size of error-based network updates, which corresponds to root mean square (RMS) of those gradient values. Based on RMS values, another method called rmsProp [23] proposes another update formula as given in the sequel, where gradient terms (g) squared and its expectation is taken ($E[.]$) per time frame (t), so $E[g^2]_t$ points out the needed term for updating the neural network parameters (ϵ is 10^{-8}, Keras epsilon):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{4.1}$$

In the next sections, Adam and AdaMax optimizer variants -based on this adaptive learning algorithms- will be investigated in detail. However, this section will be concluded after briefly introducing a different type of learning algorithms: Batch learning.

Based on Newton's quadratic approximation and Taylor expansion, this algorithms calculate the inverse Hessian matrix (by inverting the matrix that contains the second derivatives of the loss function) in case of BFGS (or part of it, in case of L-BFGS [22]) and other Quasi-Newton optimization algorithms. Taking whole dataset and computing gradient values alongside with inverse Hessian matrix is mostly impossible for large datasets, so partial solutions are used for most of deep learning problems. Even in this way, large memory and long training time are needed for good training of the neural networks. Because of this fact and capabilities of the computer hardware used for our research, batch learning algorithms are not applied in this thesis.

4.2 Adam Optimization Algorithm

Adam is published in [12], is an optimization algorithm with adaptiveness similar to its discussed previously precedents, and less memory requirement of memory compared to previous ones.

To explain, Adam algorithm calculates learning rates for each neural network parameter -again- by using stored average of gradients and their squared values. Therefore, first momentum (mean, m) and second momentum (uncentered variance, v) of the gradients (g) are calculated with (4.2)-(4.6), where arbitrary constants β_1 is selected as 0.9 and β_2 is selected as 0.999 in [12]. t subscript implies the value of the term at t^{th} iteration.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.2}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4.3}$$

The bias of those estimates are modified as given below:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4.5}$$

Finally, these values are used for the mathematical calculation of neural networks' parameter updates as described as given below (ϵ is 10^{-8}):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{4.6}$$

Thus, the Adam optimizer algorithm update function is constituted. In addition to this, a different version of this algorithm named 'Nadam' (with addition of Nesterov momentum) in a similar way, but due to its poor performance only Adam optimizer will be used at this section.

These results are taken from previous chapter, specifically Categorical Cross-Entropy loss function results. It contains 50, 100 and 150 Epochs only (and results with Truncated Laplacian initialization with 12 as Truncation Constant), but the other optimizer results are far more successful than Adam optimizer so this does not pose any problem. The results are given in the Table 4.1.

This is the optimizer used in thesis chapters up to this point. In next section, a variant of Adam named as AdaMax will be introduced.

Table 4.1: Results from 10 trials, with Adam optimizer.

Number of Epochs	**50 Epochs**	**100 Epochs**	**150 Epochs**
Mean Success	80.83%	81.27%	81.19%
Maximum Success	81.16%	81.56%	81.28%

4.3 AdaMax Optimizer Algorithm

AdaMax is again published in [12], and it is a variant of Adam optimizer. Instead of using second moment (uncentered variance) in the calculation of neural network updates, this optimizer makes use of 'infinite moment' (or infinite norm) in the following way:

$$u_t = \beta_2^{\infty} v_{t-1} + (1 - \beta_2^{\infty})|g_t|^{\infty} = \max(\beta_2 \cdot v_{t-1}, |g_t|) \tag{4.7}$$

After this calculation, the update formula becomes the below equation by just replacing $\sqrt{\hat{v}_t} + \epsilon$ with u_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t \tag{4.8}$$

Thus, AdaMax also retains its important specialties of Adam while increasing its performance: Being an optimization algorithm with adaptiveness similar to its discussed previously precedents, and less memory requirement of memory compared to previous ones.

The results obtained by replacing Adam with AdaMax optimizer is given below, which contains 50, 100, 150, 200, 250 and 300 epochs results. To verify its success over Adam optimizer and show the behaviors of optimizers in detail during the training phase, these longer training durations are preferred, beginning from this part of the chapter and the thesis. We present these results in the Table 4.2.

Table 4.2: Results from 10 trials, with AdaMax optimizer.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.70%	82.39%	82.67%	82.66%	82.58%	82.81%
Maximum Success	82.01%	83.15%	83.18%	83.08%	83.34%	83.41%

As seen from Table 4.2, both mean and maximum success rates of AdaMax are better than Adam results for 50, 100 and 150 epochs' results. Given that the margin between two optimizers is around 1% and Adam optimizer results' falling trend at mean success rates, its results for 200, 250 and 300 epochs are not generated. Given that AdaMax is the best optimizer found in the in-built Keras library for the convolutional neural network that classifies the CIFAR-10 dataset, we started to modify this optimizer and enhance its learning strength by using different techniques. In the next sections, those techniques and their results will be shown, which ultimately results in a new network optimizer: SinAdaMax.

4.4 Improving AdaMax: SinAdaMax Optimizer Algorithm

In this chapter, different attempts to modify original AdaMax function will be presented in detail with their respective results on CIFAR-10 dataset. Thus, the foundation of SinAdaMax optimizer algorithm is demonstrated with all the work done. Before starting to present these techniques, it is important to understand what understanding and mathematics lie behind the SinAdaMax optimizer.

During the training epochs, initial learning rate of the AdaMax optimizer (0.002) set for each network parameter gradually lowers, until the length of stored gradients' window reaches its mature length. Afterwards, it fluctuates according to stored gradient values, differently for each different neural network parameter. In the graph at the next page, a typical learning rate (μ) trajectory is shown according to this mechanism where iterations are shown with symbol t.

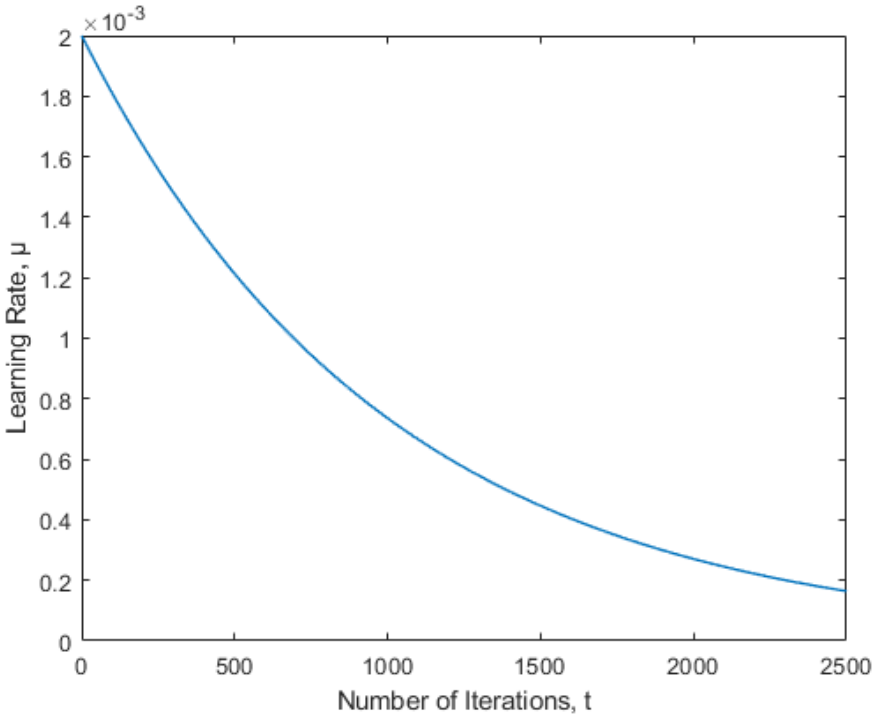

Figure 4.1: Artificial but figurative learning rate curve part seen in optimizers.

As this learning rate illustrates, the rate constantly drops until the window is completed; and after that it becomes stable with little fluctuation. This may cause neural network parameters to act 'shy', which means that learning may diminish during the training iterations with very small learning rates. To deal with this, a sinusoid is placed on top of this learning rate curve as shown below.

$$\mu_t = \mu_{constant} + A\sin(\omega t) \tag{4.9}$$

As seen in the graph at the next page, red curve has a fluctuating learning rate that differs from the blue one, which may result in a different learning success for this network parameter and whole network itself. Indeed, size of the sinusoid (A) was half of the initial learning rate, 0.001, and its frequency (ω) was 0.01 (sinusoid was $sin(0.01t)$ where t is the iteration number). Indeed, during the construction of SinAdaMax optimizer, different sinusoid magnitudes, frequency values and initial learning rates are tried. In the Tables 4.3-4.5 containing different frequency values

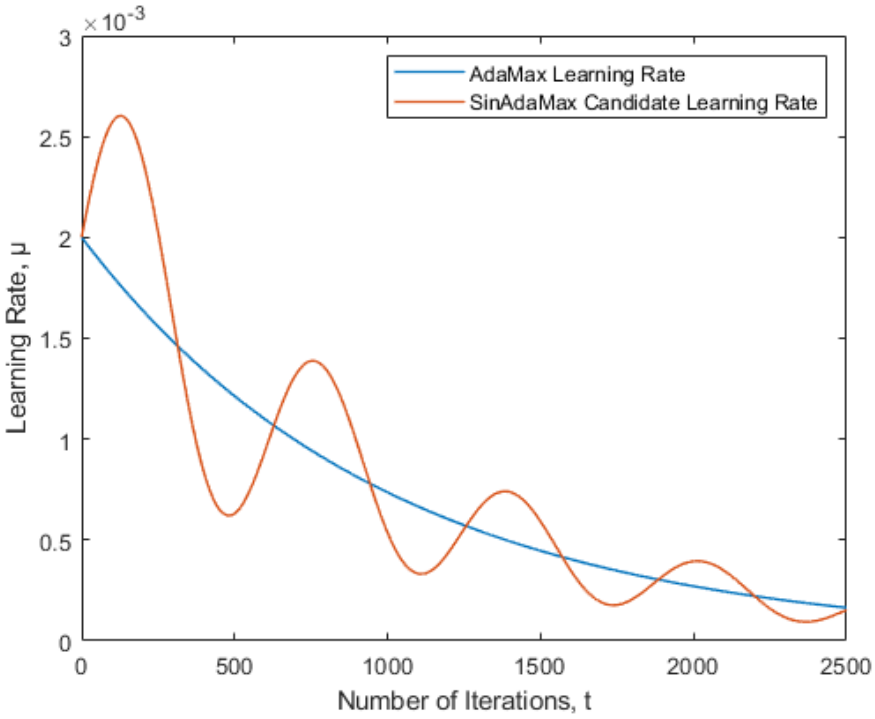

Figure 4.2: SinAdaMax candidate learning rate curve (red) vs. old blue curve.

and their results, where 0.002 initial learning rate and 0.001 sinusoid magnitude is set for this trial. Best of all results was 0.01 frequency (ω), which is given here:

Table 4.3: Results from 10 trials, for 0.01 frequency value.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.92%	82.60%	82.61%	82.73%	82.67%	82.61%
Maximum Success	82.80%	83.28%	83.18%	83.40%	83.28%	83.18%

As seen from Table 4.3, making the training phase longer showed increases and decreases of the success rates obtained from validation phase. In next page, results of the neural networks are given with other frequency values: 1 and 100.

To compare all the results obtained in this section, 0.01 is the fittest frequency value amongst all the SinAdaMax candidates in this part, as the best results are obtained with this value. However, so long as the original AdaMax function

Table 4.4: Results from 10 trials, for frequency value 1.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.50%	81.90%	82.26%	82.41%	82.49%	82.52%
Maximum Success	82.06%	83.32%	83.78%	83.55%	83.56%	84.01%

Table 4.5: Results from 10 trials, for frequency value 100.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.46%	82.02%	82.47%	82.52%	82.69%	82.59%
Maximum Success	82.03%	82.61%	83.28%	83.10%	83.27%	83.47%

are still better than these SinAdaMax candidates, modification of the optimizer function should continue; the magnitude of added sinusoid is varied up and down for this purpose.

Initial magnitude of the sinusoid placed over the learning rate curve was 0.001. Tables 4.6-4.8 which belong to different magnitude values respectively: 0.0005, 0.001 (same as previous), and 0.0015. Note that frequency value was set to 1 for this simulation, which is not the ideal value but with mediocre success; therefore the comparison between these magnitude values are still meaningful. Initial learning rate value was again 0.002. First, already presented results are given below, in Table 4.6, for comparison.

The next magnitude value, which is presented to compare with results of 'original' ones, is 0.0015. The results of this value is given in a separate table as below (Table 4.7) and they are pretty lower than the 0.001 results.

The last magnitude value is 0.0005. The results of this value is -again- given in a separate table as below (Table 4.8) and they are the best ones amongst all the magnitude values tried in this part. In addition, its performance also exceeds the original AdaMax optimizer results in mean success rates (shown in bold).

Table 4.6: Results from 10 trials, for magnitude value 0.001.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.50%	81.90%	82.26%	82.41%	82.49%	82.52%
Maximum Success	82.06%	83.32%	83.78%	83.55%	83.56%	84.01%

Table 4.7: Results from 10 trials, for magnitude value 0.0015.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	80.61%	81.47%	81.76%	81.97%	82.25%	82.23%
Maximum Success	81.86%	82.32%	82.95%	82.85%	83.04%	83.35%

Table 4.8: Results from 10 trials, for magnitude value 0.0005.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.69%	82.40%	82.39%	82.72%	82.75%	82.82%
Maximum Success	82.35%	83.03%	83.18%	83.45%	83.62%	83.53%

Briefly in this part, the added magnitude of the sinusoid is varied and obtained mean and maximum success rates are compared. Magnitude value 0.0005 yielded in 82.82% in mean success rate, which is best amongst all other SinAdaMax candidates and even better than the original AdaMax mean success rate result slightly (82.81%). In the final design of SinAdaMax, these values will be taken into account. However, given that the initial learning rate can also be varied in these experiments, the goal for best SinAdaMax optimizer with higher success rates (both in mean and maximum ones) could be pursued.

This part start with varying the initial learning rate constant (0.002) up and down, and observing the situation resulted in with this change. While designing the final SinAdaMax optimizer, these clues will pave the way of designing the best optimizer possible. In the next page, there are the tables of different initial learning rate constant values except the default 0.002 value, again given for frequency value 1 and sinusoid magnitude 0.001.

As the tables demonstrate, 0.0025 value gives best mean success rate result (82.89%) and a maximum success rate of 84.00% until this point of the chapter, even with a mediocre -but not most successful- frequency and sinusoid magnitude. As this results show, another clue of designing the best sinusoid value to place over the learning curve is obtained; 0.002 and 0.0025 are good numbers for initial learning rate constant value. With these takeaways are in mind, let us move to the last step before finalizing the SinAdaMax optimizer by tuning numerical values: Addition of absolute sinusoid function which is given in (4.10).

Table 4.9: Results from 10 trials, for initial learning rate constant 0.0015.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.09%	81.79%	82.14%	82.21%	82.26%	82.42%
Maximum Success	82.27%	83.23%	83.38%	83.16%	83.56%	83.56%

Table 4.10: Results from 10 trials, for initial learning rate constant 0.0025.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.56%	82.09%	82.63%	82.43%	82.40%	82.89%
Maximum Success	82.71%	82.91%	83.71%	83.20%	83.21%	84.00%

Table 4.11: Results from 10 trials, for initial learning rate constant 0.003.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.18%	82.01%	82.09%	82.43%	82.59%	82.69%
Maximum Success	82.66%	83.16%	83.21%	83.66%	83.66%	83.80%

$$\mu_t = \mu_{constant} + |A \sin(\omega t)| \tag{4.10}$$

A basic characteristic of sinusoid function (sin(.)) is that it has both positive and negative parts in a symmetric fashion. However, as these negative parts may decrement the learning rate and slow down the learning process, it is proposed to use an absolute sinusoid instead of an original one. The graph which demonstrates the resulting learning rate curve is given in Figure 4.3, where red curve shows the curve with absolute sinusoid is placed on top of it.

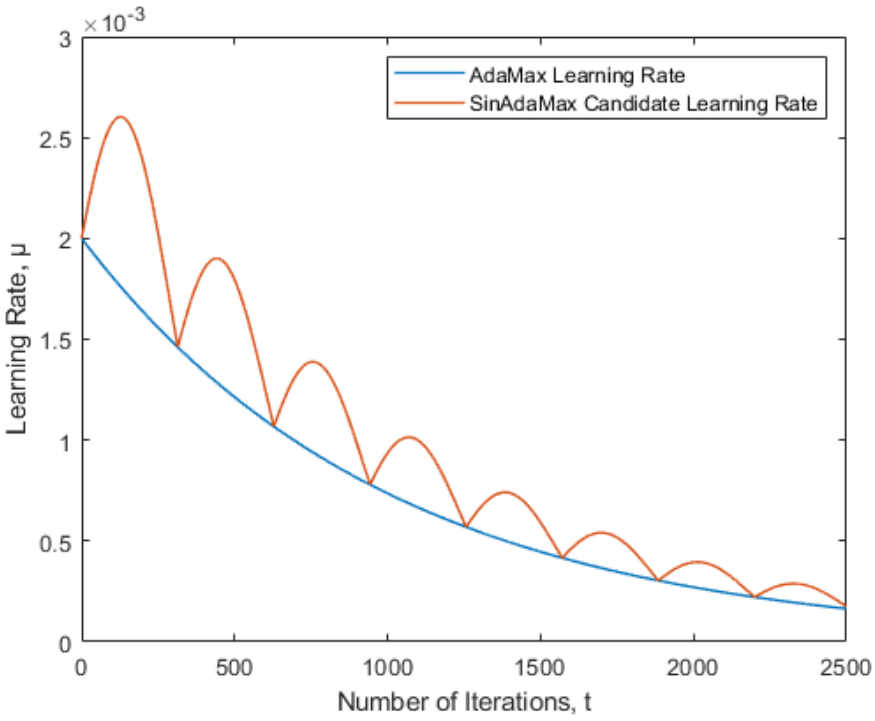

Figure 4.3: New SinAdaMax candidate learning rate curve (red) vs. old blue one.

For this purpose, sinusoids with two different frequency values but same initial learning rate constants and sinusoid magnitudes are prepared to compare with previous results. First, let us start with frequency value 1 and the comparative results obtained; with original sinusoid and absolute sinusoid (initial learning rate constant is 0.002, sinusoid magnitude is 0.001):

As seen in Tables 4.12-4.13, replacing the original sinusoid with an absolute

Table 4.12: Results from 10 trials, for frequency value 1 and original sinusoid.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.50%	81.90%	82.26%	82.41%	82.49%	82.52%
Maximum Success	82.06%	83.32%	83.78%	83.55%	83.56%	84.01%

Table 4.13: Results from 10 trials, for frequency value 1 and absolute sinusoid.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.72%	82.28%	82.28%	82.80%	82.74%	82.84%
Maximum Success	82.57%	83.33%	83.03%	83.60%	83.56%	84.02%

sinusoid (which has same magnitude -of course before the absolute operation-, frequency and initial learning rate constant) increased the mean success rates in each step of the table, where maximum success rates are also increased. Given this successful outcomes, it can be claimed that the use of absolute sinusoid function gave the network a good specialty: Combining the coarse tuning and fine tuning of neural network in the error surface and thus, better classification results. To increase the mean and maximum success rates, frequency value is decreased to 0.01 and same comparison is made again in Tables 4.14-4.15:

Table 4.14: Results from 10 trials, for frequency value 0.01 and original sinusoid.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.92%	82.60%	82.61%	82.73%	82.67%	82.61%
Maximum Success	82.80%	83.28%	83.18%	83.40%	83.28%	83.18%

Table 4.15: Results from 10 trials, for frequency value 0.01 and absolute sinusoid.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.47%	82.00%	82.49%	82.56%	82.70%	82.90%
Maximum Success	82.90%	82.98%	83.31%	83.09%	83.99%	83.68%

As mean success results imply, this combination is slightly better than the previous higher frequency (1) version for classifying the CIFAR-10 dataset with the convolutional neural network. Maximum success rates still contain close percentages such as 83.99% in comparison, but in the end of this part 82.90% is the highest mean success rate obtained in this thesis until this part.

To increase the success rates in this section, networks with another initialization method named as He-truncated Gaussian initialization is also used in parallel for absolute sinusoid version of our optimizer function. So long as this candidate of SinAdaMax was more successful than the other optimizers, for He-truncated Laplacian initialized neural networks, it is tried in He-truncated Gaussian initialized neural networks too. Results are given in the Tables 4.16-4.17, where same absolute sinusoid magnitude (0.001) and same initial learning rate (0.002) is used in all networks aside of different frequency values for absolute sinusoids.

Table 4.16: Results from 10 trials: Used 0.01 frequency & He-truncated Gaussian.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.07%	82.32%	82.55%	82.96%	83.20%	**83.21%**
Maximum Success	82.60%	82.71%	83.43%	83.44%	84.05%	**84.15%**

Table 4.17: Results from 10 trials: Used 1 frequency & He-truncated Gaussian.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.66%	82.41%	82.44%	82.63%	82.79%	82.98%
Maximum Success	82.40%	83.09%	83.18%	83.68%	83.44%	83.62%

In result, mean success rates are increased in order of 0.3% in comparison by replacing He-truncated Laplacian with He-truncated Gaussian in this combination. This situation is not special for our SinAdaMax optimizer candidate; as this success increase occurs not only for our custom optimizer, but also for standard AdaMax optimizer mean success rates. Their comparative results are given below for showing this reality, where first table (Table 4.18) is same as Table 4.2 and second table is created for the comparison.

Table 4.18: Results from 10 trials, with AdaMax (He-truncated Laplacian initialization).

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.70%	82.39%	82.67%	82.66%	82.58%	82.81%
Maximum Success	82.01%	83.15%	83.18%	83.08%	83.34%	83.41%

Table 4.19: Results from 10 trials, with AdaMax (He-truncated Gaussian initialization).

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.62%	82.24%	82.64%	82.68%	82.92%	83.10%
Maximum Success	82.23%	82.77%	83.41%	83.85%	83.51%	83.69%

As seen in these tables, regardless of the optimizer modification, in this phase of this thesis He-truncated Gaussian initialization performs better (around 0.3% for mean success rates) than He-truncated Laplacian initialization. Because of this outcome, remaining work will mainly contain He-truncated Gaussian initialization except the final chapters of thesis. Now, last part of this chapter, the final tuning of the SinAdaMax optimizer is completed an the resulting material is demonstrated.

Before finishing the tuning of SinAdaMax, different approaches were also considered for increasing the success of the optimizer more. One of them was to allow negative learning rates for getting out of local minimas and finding better ones gradually in time, the other was making the magnitude of added sinusoid

time-invariant (i.e. iteration-invariant). All those ideas are tried with convolutional neural networks, but so long as they did not contributed an increase of success, this work is not added to this chapter. After those attempts, the sinusoid magnitude and initial learning rate constant is modified altogether to figure out the most successful combination, while holding the frequency value 0.01 constant with He-truncated Gaussian initialization. In result, 0.0025 initial learning rate constant and 0.00075 sinusoid magnitude for absolute sinusoid are picked up as the best values for finalizing the SinAdaMax optimizer. Resulting hypothetical but representative learning rate curve (shown in red) is shown in below graph (Figure 4.4) to compare with original AdaMax optimizer's learning rate curve.

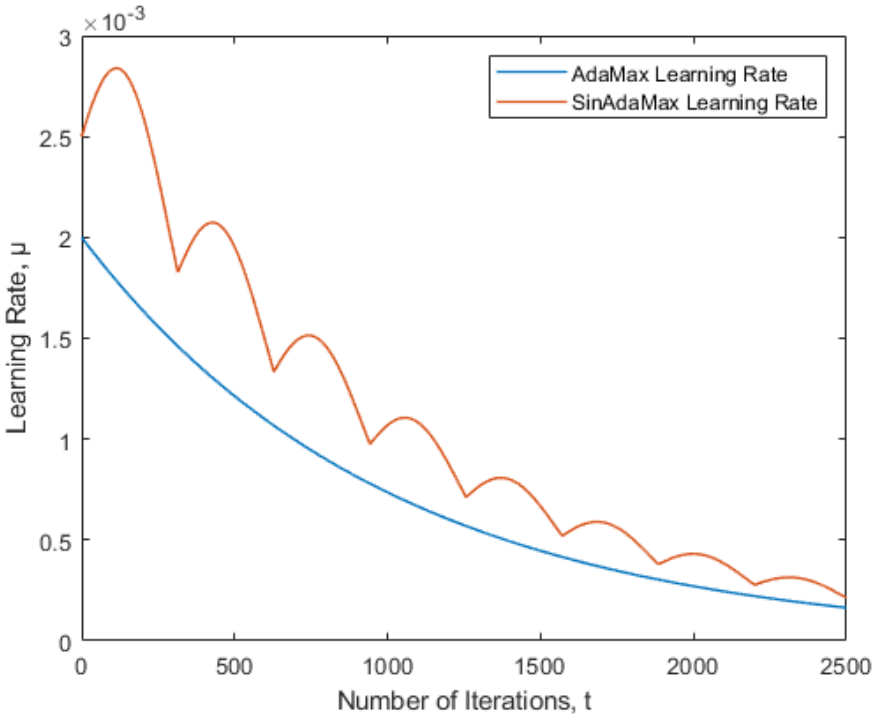

Figure 4.4: Final SinAdaMax learning rate curve (red) vs. old blue curve.

To sum up, as represented in this figure, SinAdaMax optimizer's learning rate curve starts from 0.0025 instead of AdaMax optimizer's 0.002. The absolute sinusoid placed on top of the learning rate curve has a 0.00075 sinusoid magnitude, which is zero in the AdaMax optimizer as it does not contain one. This sinusoid has a 0.01 frequency value, which is naturally zero in the AdaMax optimizer's learning curve. And as the obtained results during the training phase imply,

SinAdaMax optimizer algorithm performs better than the AdaMax optimizer algorithm.

To conclude this chapter, several optimizer algorithms - from most simple and primitive ones to most recent and complex ones - are tried with the convolutional neural network which classifies the CIFAR-10 dataset. Including Stochastic Gradient Descent (SGD) based optimizers - with or without Nesterov momentum - and Adaptive Optimizer Algorithms such as AdaGrad and AdaDelta, many optimizer algorithms are tried but fell below the default optimizer algorithm of the neural network: Adam. However, its variant AdaMax managed to exceed the mean and maximum success rates of its predecessor Adam. After this result, we achieved to modify the AdaMax optimizer algorithm and this modification yielded in SinAdaMax, which became the most successful optimizer algorithm in this chapter. However, as the research of the next thesis chapter, Activation Functions, went in parallel with this chapter's research, not only final version of SinAdaMax optimizer is used; but instead some other optimizers (such as AdaMax and preceding SinAdaMax versions) are also used for the experiments at the Activation Functions chapter.

Chapter 5

Activation Functions

Every neural network must include activation functions. Activation function is the thing that makes standard neural networks differ from simple matrix sets and their multiplications. Without using an activation function, it is impossible to learn nonlinear problems with ease. These activation functions greatly affect the depth of learning and proceeding of the neural network in the error surface.

From the beginning of neural networks, different activation functions are used. Sigmoid, and then hyperbolic tangent ($tanh(.)$) function were early used ones, with inspiration came from real neurons and their activities. After that, Softmax (alternatively Softsign for [1, -1] problems) activation function is employed in output layer by many researchers alongside with Sigmoid or $tanh(.)$ functions, through 2000s [10]. With the Nair and Hinton's article at 2010 [14], Rectified Linear Units (ReLUs) increased the strength of neural networks significantly, and opened the way of solving more complex learning problems. After this breakthrough, many other variants of ReLU are proposed in this field, such as Exponential Linear Unit (ELU), Selective Exponential Linear Unit (SELU) and so on. The research, for upgrading the specialties and success of these activation functions, continues as of today. Currently, these are some of the well-known and good-performing activation functions which are popular and used in the experiments conducted in this chapter.

In this chapter, the activation functions used in the convolutional neural networks are separated into two different classes. First part consists of output activation functions and their results, which are used at the last layer of neural networks only. Second part includes internal activation functions and their results, more in the literature with many different alternatives. Although some output activation functions were used as internal activation functions in the past, recent feedforward and convolutional neural networks use separate functions in each network section for better results. Therefore, thesis will separate these function kinds.

The neural networks in this chapter worked with following attributes only (except different activation functions): AdaMax, SinAdaMax and several SinAdaMax optimizer candidates (see Chapter 4), Categorical Cross-Entropy loss (see [15]), He-truncated Gaussian or Laplacian initialization (see Chapter 2), Mini-Batch Learning (see [16])with 128 samples in each batch, and training is completed in 300 epochs with 50 epoch intervals. Training is done with network at Figure 1.3 (Structure 1.1) using first 4 data batches of CIFAR-10 dataset, and validation is done with remaining fifth data batch (each data batch contains 10000 samples and shuffled randomly). From 10 trials with this configuration, mean and maximum success rates are obtained for comparison.

5.1 Output Activation Functions

These activation functions constitute the inseparable partition of neural networks: Output layer. In this part, different functions for output layer are used instead of default Softmax activation function (a.k.a. transformation) and the results are observed. But first, let's introduce the Softmax. It covers the $[0, 1]$ interval, and by referring to neuron at Figure 1.1 mathematical expression of Softmax is given below for jth output neuron (raw output is v_j, activation function output is named as o_j), where all the output neurons are summed up with sigma (Σ) operator (ith output neuron's raw output is v_i, activation function output is named as o_i). However, it is not possible to visualize Softmax function.

$$softmax(v_j) = o_j = \sigma(v_j) = e^{v_j}/(\Sigma_{i=1}^{all} e^{v_i}) \tag{5.1}$$

The other functions to replace Softmax activation function include ones which performed poorly during the training, and include ones that can compete with Softmax - but still cannot exceed its success rates. This part will start by presenting the signature activation function: Sigmoid. This function's graph is given in Figure 5.1. As seen in the below graph easily, although Sigmoid function is satisfies the differentiability and other properties of an output activation function, its saturation in the beginning and end parts are the defects of this function. This reality reflects to the results obtained by replacing default Softmax with Sigmoid, which can be expressed with formula given in (5.2) (v is input, o output value).

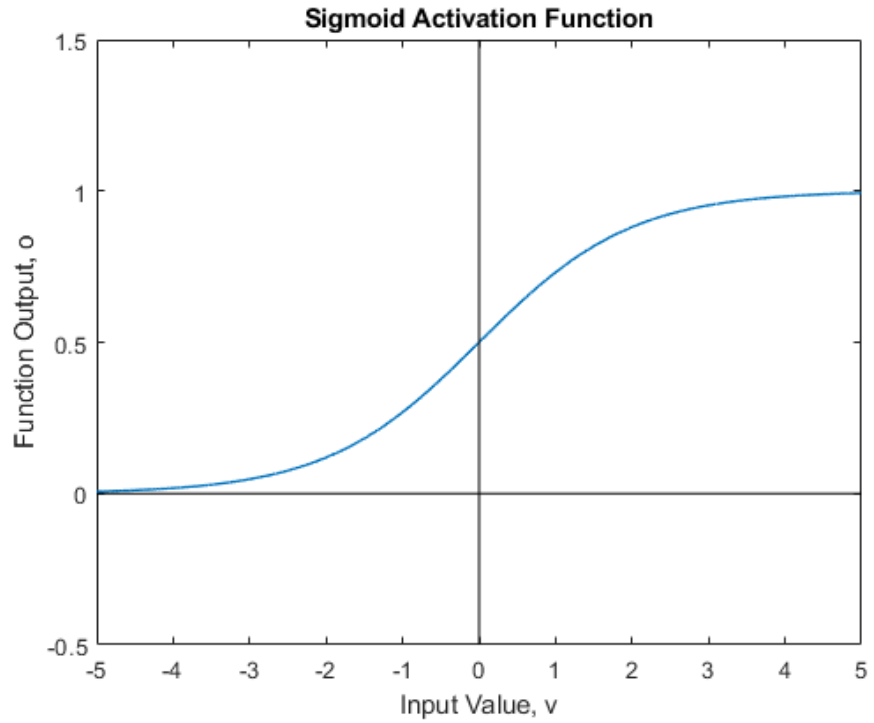

Figure 5.1: The Sigmoid activation function is displayed.

$$sigmoid(v) = o = \frac{1}{1+e^{-v}} = (1+e^{-v})^{-1} \tag{5.2}$$

From the 10 trials conducted, networks containing Sigmoid output activation

function performed very badly compared to networks with Softmax output activation function. 5 of the Sigmoid trials is stuck around 10% percent success rates, which means that they failed to learn anything. The mean success rates of other trials was slightly over 80%, but not one of them managed to reach or exceed 83% success rate. In addition to that, 3 of those networks failed after a certain number of epochs, falling to 10% in validation success rates. Because of this evident bad performance, Sigmoid is discarded as a candidate output activation function.

Another proposed candidate to replace the Softmax function was Softsign. Softsign has a different formula. The graph of the Softsign function and its mathematical formula are given in (5.3), where v is the input value and o is the output value of the neuron.

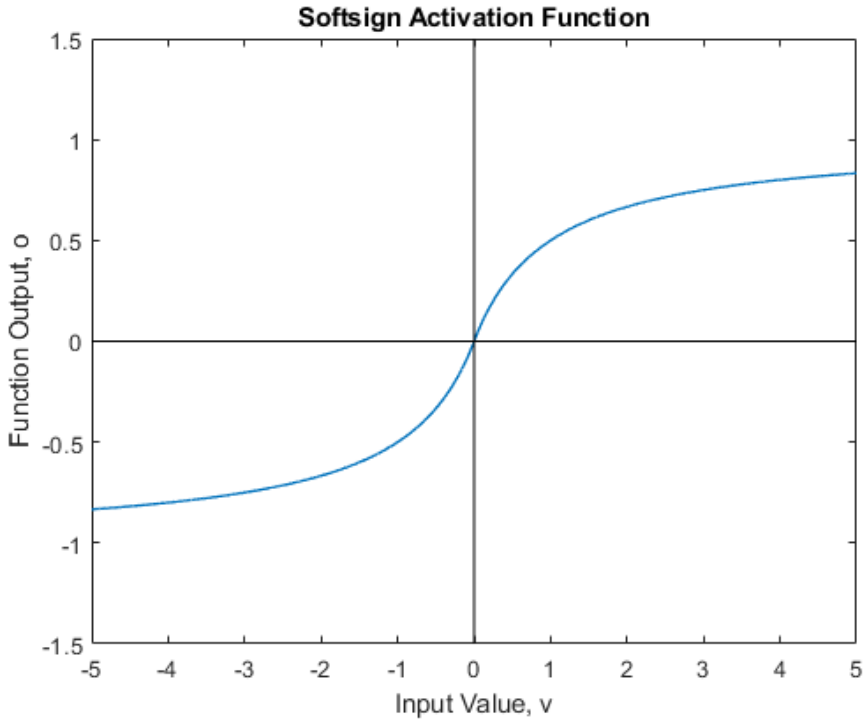

Figure 5.2: The Softsign activation function is displayed.

$$softsign(v) = o = v/(|v| + 1) \tag{5.3}$$

To work with this output activation function candidate, the labels of both

training and validation samples are adjusted properly. However, compared to Softmax results - even the Sigmoid function's results - Softsign performed worst. All 10 trials performed very badly, around 10% success rates, right from the start of neural network training. This may be because of the reason that the input values were positive, and pumping negative output values to the network parameter via backpropagation prevented the neural network to learn anything which would translate into success rates near 80%. Because of this reason Softsign function did not work, although output values were adjusted according to function.

Another proposed candidate during the research was the hyperbolic tangent function. However, so long as Softsign performed very bad due to its negativity and Sigmoid performance was clearly below the Softmax output activation function, no experiment is conducted with this function as output activation function in this chapter. However, its formula is given in (5.4) and its graph is given in Figure 5.3 for the sake of completeness of this part (v is input, o output value).

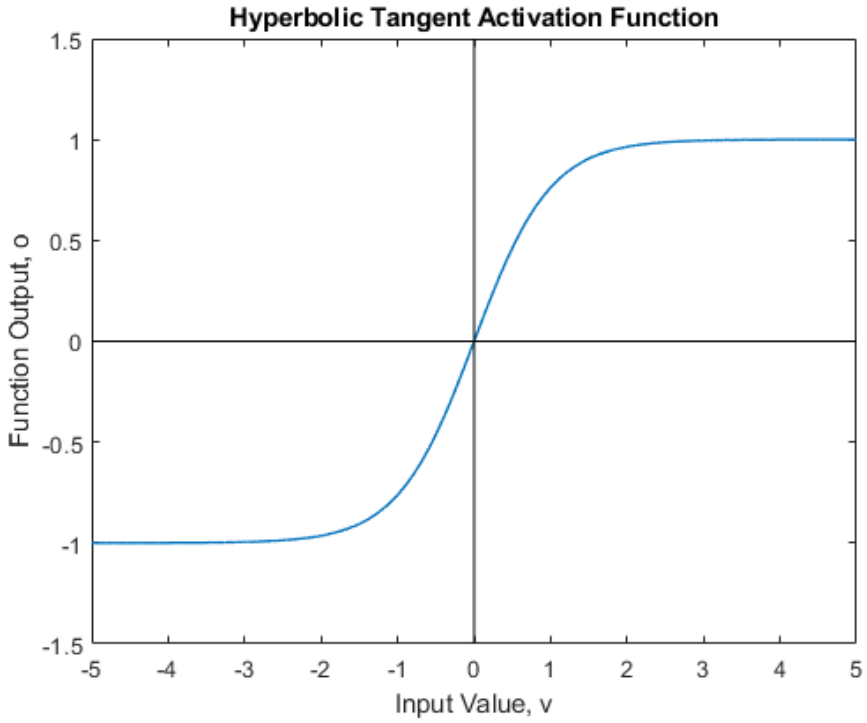

Figure 5.3: The $tanh(.)$ activation function is displayed.

$$tanh(v) = o = \frac{e^{v} - e^{-v}}{e^{-v} + e^{-v}} = (e^{v} - e^{-v})/(e^{-v} + e^{-v}) \tag{5.4}$$

Given the results obtained with different kinds of output activation candidates, the default Softmax function is chosen as the best output activation function available in the literature for the convolutional neural network which classifies the CIFAR-10 dataset. In the next part, internal activation functions will be investigated in detail, by replacing ReLU with other internal activation function candidates.

5.2 Internal Activation Functions

Internal activation functions correspond to the activation functions used in the neural network except the output layer. Throughout the history of neural networks, hyperbolic tangent and Sigmoid function are used as internal activation functions in early times. However, as the data complexity is increased with big data problems, the nonlinearity needed for solutions with feedforward and convolutional neural networks increased. Because of this reason, powerful activation functions like ReLU started to dominate the field.

In this section, the default internal activation function used by the convolutional neural network - Rectified Linear Unit (ReLU) - is replaced with other activation functions in the literature and a custom activation function candidate. Specifically Exponential Linear Unit (ELU), Softplus, Selective Exponential Linear Unit (SELU) and their partial combinations with ReLU are tried as solutions, in addition to custom activation function. After the replacements, results obtained by the changes are presented and by comparing the success rates, the best internal activation function is found.

Partial combinations are obtained for some of the activation functions used for experimenting on the convolutional neural network. These combinations included ReLU, ELU and Softplus activations namely. It is done by dividing the remaining

part of the neural network by into two: Entrance (Containing Convolutional and Max-Pool layers) and Middle (Feedforward) part. The specific details of combinations, such as which activation function is used in which part is given in the relevant subsection. The default activation function used in networks is ReLU. It has a simple graph and formula which are given below (v is input, o output value of neuron).

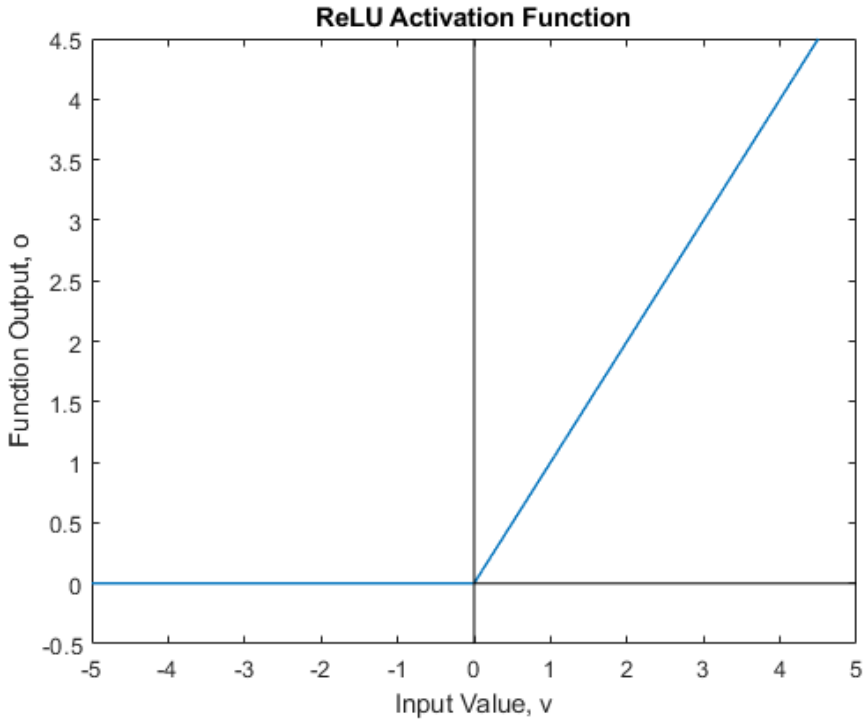

Figure 5.4: The ReLU activation function is displayed.

$$ReLU(v) = o = max\{0, v\} \tag{5.5}$$

As seen above, ReLU contains a non-differentiable point at origin where derivative isn't set. In our case, Keras library on Tensorflow in Python 3.6 sets a value:

$$\frac{dReLU(v)}{dv} = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{5.6}$$

As seen from the graph, although Sigmoid and hyperbolic tangent functions and their derivatives are more complex, their performance is not as good as the Rectified Linear Unit as internal activation function. Therefore, the comparison in this part did not include them. As a reminder from previous chapter, mean and maximum success rates obtained with only ReLU are given. Final version of SinAdaMax is used in these two tables, respectively for He-truncated Laplacian and He-truncated Gaussian initialization.

Table 5.1: Results from 10 trials: Used ReLU and He-truncated Laplacian.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.47%	82.00%	82.49%	82.56%	82.70%	**82.90**%
Maximum Success	82.90%	82.98%	83.31%	83.09%	83.99%	83.68%

Table 5.2: Results from 10 trials: Used ReLU and He-truncated Gaussian.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.07%	82.32%	82.55%	82.96%	83.20%	83.21%
Maximum Success	82.60%	82.71%	83.43%	83.44%	84.05%	84.15%

The first internal activation function candidate is the Softplus function. Its graph is given in the Figure 5.5 and its mathematical expression is given in (5.7):

$$softplus(v) = o = log(e^v + 1) \tag{5.7}$$

In the sequel, results are given for ReLU-Softplus combination obtained for He-truncated Gaussian initialization. Full Softplus activation function usage (except output layer) produces success rates around 70%, which is significantly low, therefore their experiments are excluded from this part. Results for combination are given in the Table 5.3, where original AdaMax is used:

In addition to that, a SinAdaMax candidate is used for this internal activation function combination, where both He-truncated Gaussian and He-truncated

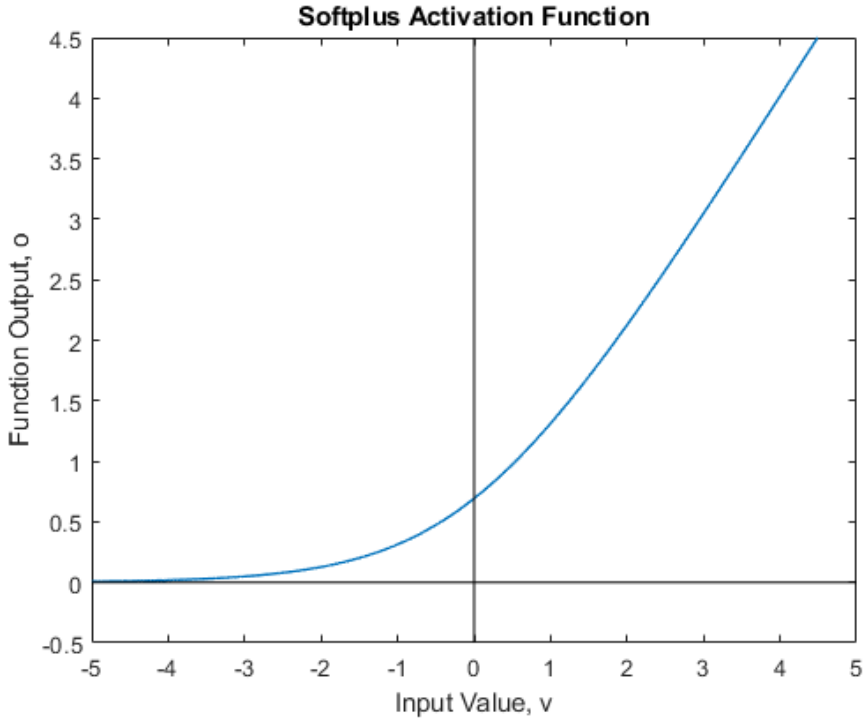

Figure 5.5: The Softplus activation function is displayed.

Table 5.3: Results from 10 trials: Used ReLU-Softplus & He-truncated Gaussian.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.99%	82.26%	82.61%	82.65%	82.81%	83.00%
Maximum Success	82.47%	83.00%	82.99%	83.16%	83.15%	83.41%

Laplacian initializations are experimented. The SinAdaMax candidate has 0.01 frequency constant, a slightly different initial learning rate (0.002) and sinusoid magnitude (0.001) with an absolute addition. Results are given in Tables 5.4-5.5.

These three tables contain the ReLU-Softplus combination's results for different optimizers and initializations, and compared to other results in the previous chapters and this chapter, they do not give better results. Thus, the Softplus activation function is discarded as a candidate for internal activation function of a better convolutional neural network in the final chapter, which will be designed to classify CIFAR-10 dataset.

Table 5.4: Results from 10 trials: Used ReLU-Softplus, He-truncated Gaussian and a SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.76%	82.37%	82.55%	82.57%	82.94%	82.84%
Maximum Success	82.23%	82.79%	83.27%	83.06%	83.58%	83.49%

Table 5.5: Results from 10 trials: Used ReLU-Softplus, He-truncated Laplacian and a SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.79%	82.01%	82.09%	82.42%	82.77%	82.55%
Maximum Success	82.39%	82.74%	82.67%	83.26%	83.47%	83.14%

The second internal activation function candidate is the Exponential Linear Unit (ELU) activation function [25]. Its graph is given in Figure 5.6, where the formula shown in the graph is given in (5.8). As seen the graph, left part ($v < 0$) does not correspond to $o = 0$, which differs from ReLU function. This allows the learning to continue even in this region. That is the main difference between ReLU and ELU activation functions which reflects to the success rates obtained by the experiments and shown in next tables of validation success rates.

$$ELU(v) = o = \begin{cases} v & v > 0 \\ e^v - 1 & v \leq 0 \end{cases} \tag{5.8}$$

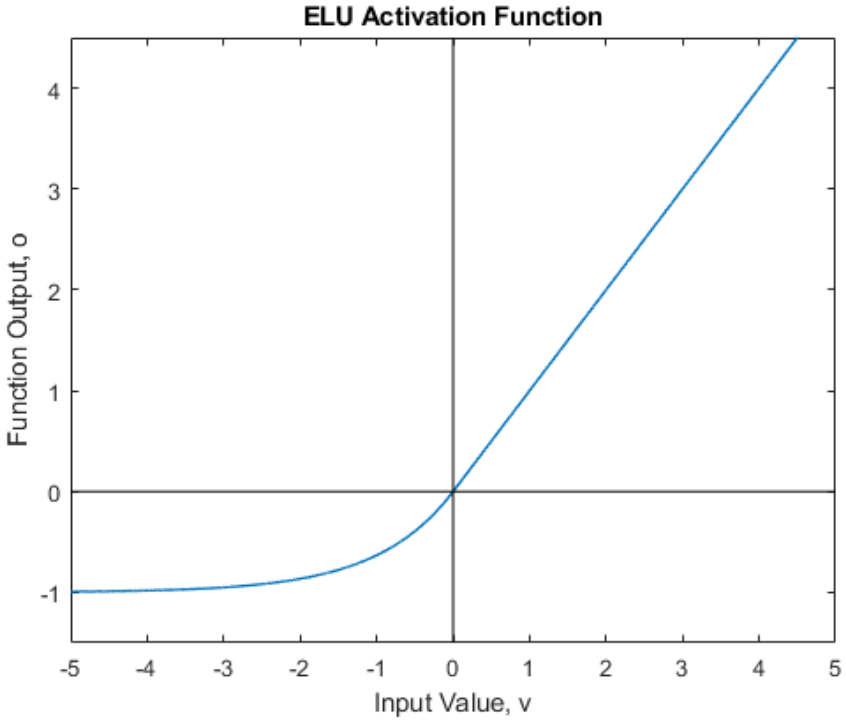

Figure 5.6: The Exponential Linear Unit (ELU) activation function is displayed.

This activation function is experimented in combination with ReLU activation function for the entrance and middle part of convolutional neural network, similar to ReLU-Softplus experiment. After that, ELU activation function is experimented as standalone. All of these results are obtained with different initialization methods and optimizers in place. Tables are given below, first ReLU-ELU combination results are shown:

Table 5.6: Results from 10 trials: Used ReLU-ELU, He-truncated Gaussian and same SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.17%	81.65%	82.08%	82.53%	82.80%	82.81%
Maximum Success	81.73%	82.18%	82.87%	83.01%	83.43%	83.42%

In this combination where convolutional layers had ReLU and feedforward layers had ELU internal activation functions, total success rates in both mean

and maximum are below the predecessor where only internal activation function was ReLU, and indeed lower than ReLU-Softplus combination. However, full ELU activation function usage gives different outcomes. The next results are generated by same candidate SinAdaMax optimizer, and He-truncated Gaussian or He-truncated Laplacian initialization is used. Tables contain the configuration details as well as success rates given.

Table 5.7: Results from 10 trials: Used ELU, He-truncated Gaussian and same SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.01%	82.65%	82.93%	83.18%	83.30%	83.16%
Maximum Success	82.44%	83.41%	83.83%	84.18%	83.95%	84.16%

Table 5.8: Results from 10 trials: Used ELU, He-truncated Laplacian and same SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.26%	81.96%	82.21%	82.61%	82.65%	82.49%
Maximum Success	82.54%	83.17%	83.18%	84.02%	83.65%	83.77%

As seen in above, using ELU as the only internal activation function resulted in significant increase of success rates at both mean and maximum ones, regardless of initialization technique used - and He-truncated Gaussian performs better than He-truncated Laplacian in this case and gives the current best results in thesis.

Another activation function tried as the internal activation function candidate is Selective Exponential Linear Unit (SELU). Indeed, it is a modified version of ELU where function is multiplied with a scale (s) value and have a different alpha (α) value which was simply 1 in the ELU function. The formula of SELU activation function [26] is given in Equation 5.9 and its graph is given in Figure 5.7, where alpha is chosen around 1.673 and scale value is chosen around 1.051 by the Keras library.

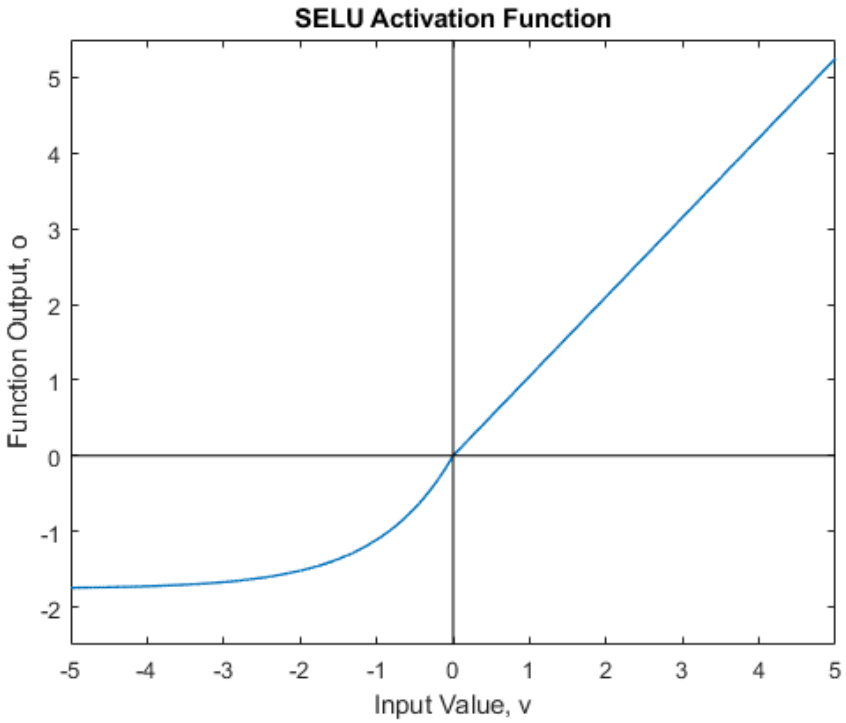

Figure 5.7: Selective Exponential Linear Unit (SELU) activation function.

$$SELU(v) = o = \begin{cases} sv & v > 0 \\ s\alpha(e^v - 1) & v \leq 0 \end{cases} \tag{5.9}$$

However, these modifications of the activation function resulted in computational complexity and increased the duration of training enormously, in addition to over-consumption of GPU sources available. The results taken with networks which contain only SELU as the internal activation function gave mean success rates around 81-82% and maximum success rates did not perform well, SELU is discarded as a candidate internal activation function due to these reasons.

Although modifying the activation function may yield in increasing the complexity, it can also increase the success - if the modification is made in a clever way. Here, a custom activation function, Dominantly Exponential Linear Unit (DELU), is developed step-by-step while optimizing the performance and limiting the complexity as much as possible. The graph of first candidate of DELU is

displayed in Figure 5.8 and formula is given in Figure 5.10 where v is the input and o is the output value of activation function. There are 3 sections in formula and graph, where 1.25643 is selected as the second transition point which satisfies $e^x - 1 = x$ equation, to ensure a smoother transition at the function and its derivative:

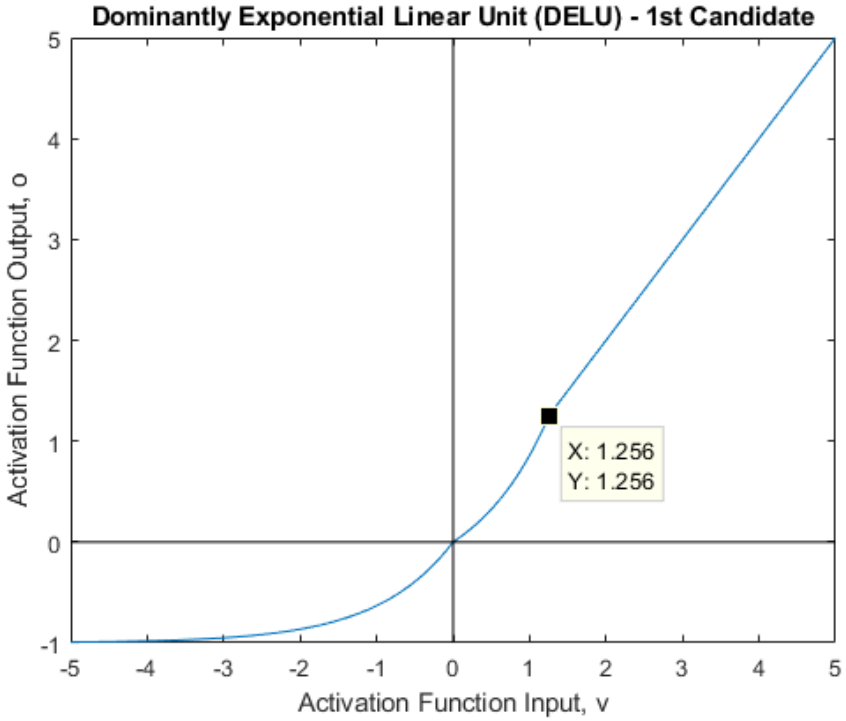

Figure 5.8: First attempt of Dominant Exponential Linear Unit (DELU).

$$DELU1(v) = o = \begin{cases} v & v > 1.25643 \\ (e^v - 1)/2 & 1.25643 \geq v > 0 \\ (e^v - 1) & v \leq 0 \end{cases} \tag{5.10}$$

However, this three-parted function increases the complexity and training time triples, in addition to instability of training with SinAdaMax candidate. Out of 10 trials with this DELU candidate as only internal activation function, just 1 trial ends 300 epochs with success rate more than 10% and over 80% (85.89%). To overcome this instability problem, second DELU candidate is proposed and this time results are taken with classical AdaMax optimizer, to figure out whether

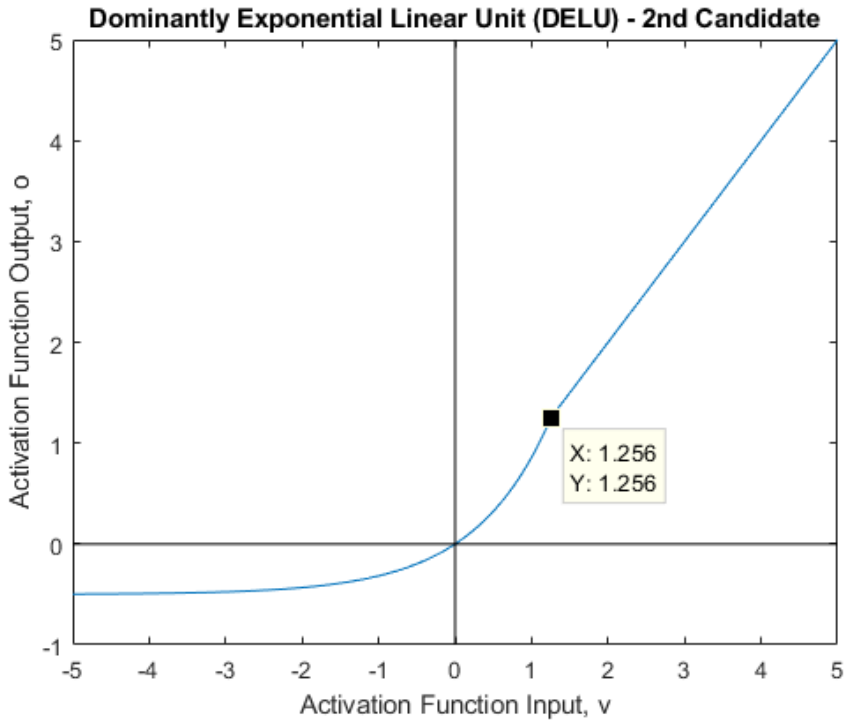

Figure 5.9: Second attempt of Dominant Exponential Linear Unit (DELU).

proposed activation function causes this problem. The function formula is more simple at this time, and there are only two function parts as seen in Figure 5.9. Again, v is input value and o is output value of activation function:

$$DELU2(v) = o = \begin{cases} v & v > 1.25643 \\ (e^v - 1)/2 & v \leq 1.25643 \end{cases} \tag{5.11}$$

As in Table 5.9, there is not a single trial whose success rate falls below 80%.

Table 5.9: Results from 10 trials: Used DELU2, He-truncated Gaussian and AdaMax.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.03%	83.91%	83.97%	84.33%	84.43%	84.59%
Maximum Success	84.15%	84.66%	84.89%	84.89%	85.09%	85.17%

Given the results of this DELU candidate which is clearly better than ReLU internal activation function with AdaMax optimizer, the candidate is also tried with a SinAdaMax candidate (see Chp. 4) which is same with first DELU candidate's part. Results contain some instable networks but apart from those, success percentages are the best amongst the thesis until this point. These instabilities are removed with the usage of regularizers which are discussed in detail at the next chapter. The results obtained by using second DELU candidate and SinAdaMax candidate are given in Table 5.10. He-truncated Gaussian initialization is again used in these networks.

Table 5.10: Results from 10 trials: Used DELU2 and a SinAdaMax candidate.

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.97%	83.51%	83.80%	84.23%	**84.71%**	84.33%
Maximum Success	83.83%	84.29%	84.39%	84.82%	**85.24%**	85.10%

However, there are 2 networks failing at 250 epochs and 3 networks failing at 300 epochs, out of 10 trials of convolutional neural networks. The mean and maximum success rates are slightly better than AdaMax version as expected, although the optimizer used is not final SinAdaMax but a SinAdaMax candidate (see Chp. 4). Besides, the DELU activation function candidate displays a strong performance compared to ReLU, ELU and others that are stable.

Although this pretty stable and good-performing internal activation function is found, the efforts for forming another one is continued. Instead of increasing exponential part at the beginning of positive part of x-axis, a decreasing exponential is proposed for this activation function candidate as seen in (5.12) where v is the input and o is the output value of the activation function. This time 1.59362 is selected as the second transition point for smoother transition. Its graph is given in Figure 5.10.

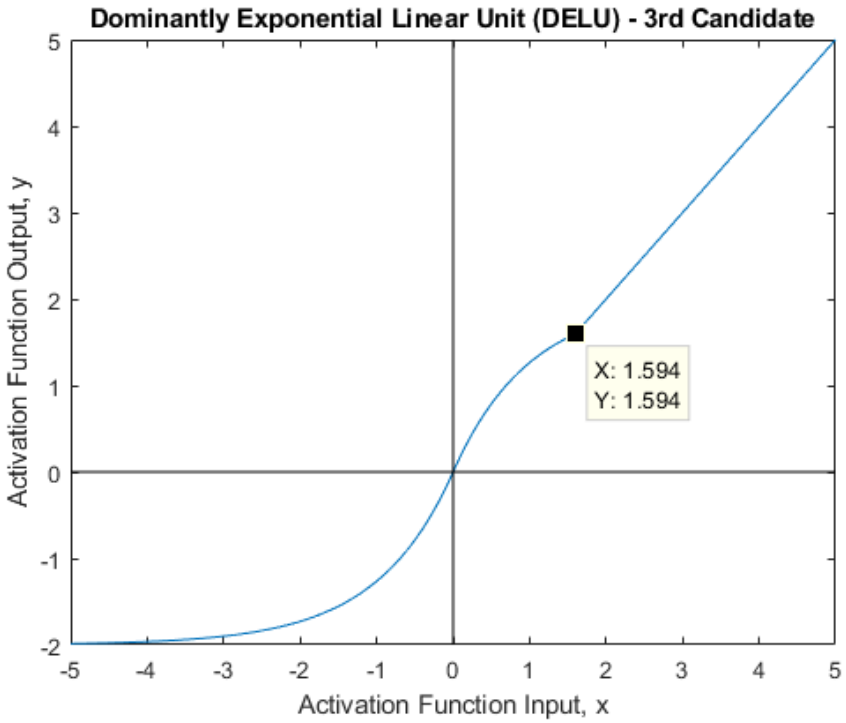

Figure 5.10: Third attempt of Dominant Exponential Linear Unit (DELU).

$$DELU3(v) = o = \begin{cases} v & v > 1.59362 \\ 2(1 - e^{-v}) & 1.59362 \geq v > 0 \\ 2(e^{v} - 1) & v \leq 0 \end{cases} \tag{5.12}$$

However, this DELU candidate performed so much worse than other candidates, and instability of the network is peaked such that just 4 trials of the convolutional neural network were enough to show that. Therefore, this activation function candidate is ignored. After these three attempts of custom activation functions, second candidate of DELU is set as the original internal activation function DELU which carries some clues from other activation function candidates but performs better than all of them. This is because of its shape in the neighborhood of zero - in which the function acts slower than classical ReLU or ELU, and this prevents early saturation of neurons while allowing the optimizer algorithm to align the neurons more efficiently to the positive or negative part. This results as better classification of input samples which is seen the success rate

tables as 1-2% increase of success percentages. Thus, the new activation function is partly similar to ReLU:

$$DELU(v) = o = \begin{cases} x & \text{if } x \geq 0 \end{cases} \tag{5.13}$$

And also partly similar to ELU (negative part, although multiplied with 0.5):

$$DELU(v) = o = \begin{cases} (e^v - 1)/2 & x < 0 \text{ (or otherwise).} \end{cases} \tag{5.14}$$

Therefore, final formula is given in (5.15) and resulting graph is given in Figure 5.11 where v is the input and o is the output value of the activation function:

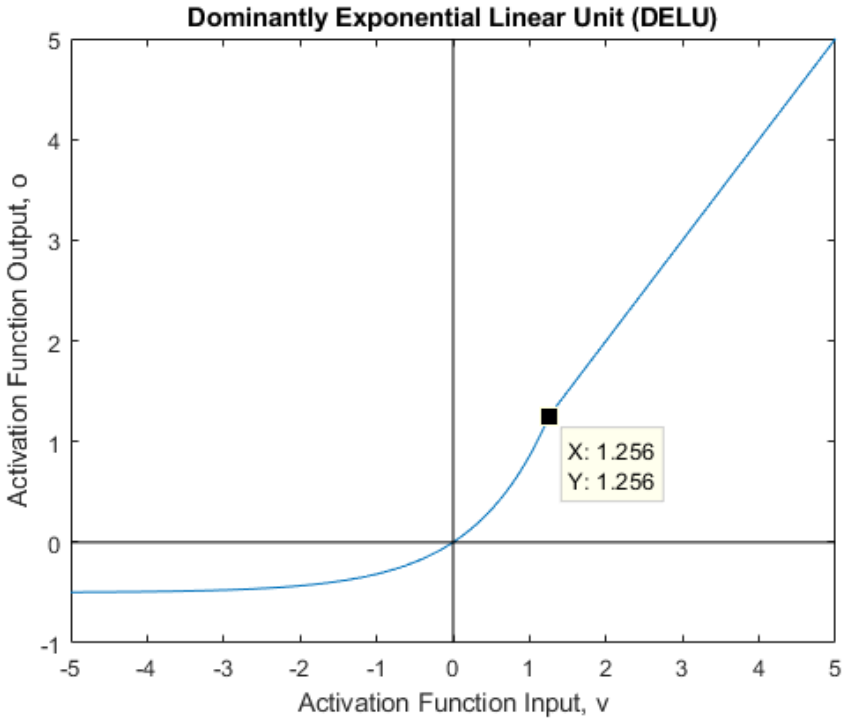

Figure 5.11: The Dominant Exponential Linear Unit (DELU).

$$DELU(v) = o = \begin{cases} v & v > 1.25643 \\ (e^v - 1)/2 & v \leq 1.25643 \end{cases} \tag{5.15}$$

In summary, this chapter covered activation functions used as internal and output neurons, and most of these activation functions (classics and better ones) are experimented with the convolutional neural network which classifies the CIFAR-10 dataset. The best output activation function turned out to be Softmax which was used in the previous chapters, and the best one amongst the literature (and in-built Keras library) was Exponential Linear Unit (ELU). After these results, a custom internal activation function is proposed and developed with the convolutional neural network with modifications which had different reasons. After its success over original ELU activation function, the second candidate proposed is set as the Dominantly Exponential Activation Function (DELU), and it is declared as the best activation function with a clear margin compared to other ones used from the existing literature. Thus, a new activation function is contributed to the literature with better results are taken with a standard convolutional neural network.

Chapter 6

Regularizers

Every neural network training method may not include regularizers, but the ones include them are performing better than the others in today's machine learning challenges. Regularizers are the assets which makes the training success rates fall, but validation and test success rates rise by avoiding overfitting in the artificial neural network, and instead forcing the network to learn more generally which boosts the validation and test success rates in this process. Regularizers, in simple, helping the neural networks to adjust themselves according to 'real' error surface rather than memorizing the training samples and limiting itself with error surface of training set. They are indeed an addition to loss function of the neural network formation and they change the original error surface in training, making it smoother. In this manner, regularizers prevent overfitting in this chapter.

From the beginning of neural networks, different regularizers are used. L1 regularization is used for penalizing the neural networks (and thus smoothing the class boundaries) which enables better learning frequently. After the millennium, usage of L1 - alongside the L2 - regularizers are widespread. This expansion happened because of the more data becoming available, in compliance with the increase of the processing power usable. With these factors, naturally overfitting of the neural networks became a significant problem. To overcome this situation, L1 and L2 regularizers, with different regularization constants, are used currently.

In this chapter, the regularizers used in the convolutional neural networks are separated into two different classes: Regularizers which are existent in the literature and custom regularizers. First part, named as "Classical Regularizers", contains well-known regularizers in use, such as L1 regularizer and L2 regularizer (which can be used in combination also). The second part, named as "Custom Regularizers" contains regularizers which are inspired from these, such as L1.5 regularizer, L3 regularizer etc. In addition to these many regularizers, each regularizer can have different regularization constants which means there is a range to scan to find the best regularizer constant for the convolutional neural network used to classify the CIFAR-10 dataset. While using more than one regularizers in the neural network structure (which is a regularizer combination), this tuning of regularization constants becomes more complex and consumes much more time. Briefly, the thesis will contain these two types of regularizers. Dropout, an optimization-based learning enhancement technique, is presented at next chapter as Keras library considers it as a layer rather than a regularization technique.

Neural networks in this chapter worked with following attributes (except different regularizers): SinAdaMax optimizer (see Chp. 4), Categorical Cross-Entropy loss (see [13]), He-truncated Gaussian initialization, DELU activation function except Softmax output layer (see Chp. 5 and [15]), Mini-Batch Learning (see [16]) with 128 samples in each batch, and training is completed in 300 epochs with 50 epoch intervals. Training is done with network at Figure 1.3 (Structure 1.1) using first 4 data batches of CIFAR-10 dataset, and validation is done with fifth data batch (each data batch has 10000 samples and shuffled randomly). From 10 trials with this settings, mean and maximum rates are obtained for comparison.

6.1 Classical Regularizers

In this part, L1 and L2 regularization techniques are explained and their usage in the training of the neural networks are discussed, with respective results. Afterwards, their combination is used as the neural network regularizer and better classification outcomes are observed for finely tuned regularization constants. After

that, another existent regularization technique named as "Max-Norm Regularization" from [27] is presented and used together with the L1 and L2 regularization techniques after a final tuning process to obtain the best results taken in this chapter. We'll start with L1 regularization.

L1 regularization is widely used not only in neural networks, but also in other machine learning algorithms such as Support Vector Machines etc. In the case of the convolutional neural network used in this thesis until this point, L1 regularization is applied to all convolutional and feedforward layers of the convolutional neural network. An exemplary application of this regularization on the error function (E) at the output layer (d is desired, o is actual output of the ith neuron) is shown below, where λ is the L1 regularization constant (simply if $\lambda = 0$, there is no L1 regularization):

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + \lambda\Sigma_{i=1}^{all}|w_i| \tag{6.1}$$

Although absolute value function is not directly differentiable, the derivative of this function is set as zero to avoid any improperness, opening the way of L1 regularization to be used regularly in the training of convolutional neural network. The addition of $\lambda\Sigma_{i=1}^{all}|w_i|$ for the regularization of neural network makes the class boundaries set by the neural network cornered, preventing the network to overfit if it is tuned properly. To make this tuning, several different values of the L1 regularization constants are tried. Results obtained with 0.000005, 0.00001, and 0.00002 L1 regularization constants are given in Tables 6.1-6.3, which are found by trial-and-error. Briefly, 0.000005 and its double, 0.00001, are good to begin.

Table 6.1: Results obtained from 10 trials: DELU, He-truncated Gaussian and L1 Constant: 0.000005

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.03%	84.12%	84.37%	85.00%	85.26%	85.41%
Maximum Success	84.07%	84.63%	85.49%	85.94%	86.43%	85.84%

Table 6.2: Results obtained from 10 trials: DELU, He-truncated Gaussian and L1 Constant: 0.00001

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.84%	83.65%	84.47%	84.84%	85.34%	85.36%
Maximum Success	83.69%	84.52%	85.44%	85.53%	85.98%	85.98%

Table 6.3: Results obtained from 10 trials: DELU, He-truncated Gaussian and L1 Constant: 0.00002

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	81.53%	83.47%	84.11%	84.86%	85.02%	85.06%
Maximum Success	83.26%	84.18%	85.09%	85.44%	86.58%	85.95%

Although tables show that L1 regularization gives better results, this does not mean that the investigative research on this issue needs to stop. The next regularization technique is L2 regularization, which will be used in combination at the next parts, but let's introduce the math behind the L2 regularization.

L2 regularization uses squares instead of absolute value function, and the summation is divided by 2 which makes it differ from L2 regularization.

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + (1/2)\lambda\Sigma_{i=1}^{all}(w_i)^2 \tag{6.2}$$

So long as squaring function is differentiable, the derivative of this function is not problematic in the training of convolutional neural network. The addition of $(1/2)\lambda\Sigma_{i=1}^{all}(w_i)^2$ for the regularization of neural network makes the class boundaries set by the neural network smoother in a different way than L1 regularization, preventing the network to overfit if it is tuned properly. To make this tuning, several different L2 regularization constants found by trial-and-error. The results obtained with 0.00005, 0.0001, and 0.0002 L1 regularization constants are given in Tables 6.4-6.6. Briefly, 0.00005 and its double, 0.0001, are good ones to begin.

Table 6.4: Results obtained from 10 trials: DELU, He-truncated Gaussian and L2 Constant: 0.00005

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.22%	84.38%	84.95%	85.26%	85.51%	85.30%
Maximum Success	84.22%	85.11%	85.87%	86.04%	86.69%	86.00%

Table 6.5: Results obtained from 10 trials: DELU, He-truncated Gaussian and L2 Constant: 0.0001

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.09%	83.86%	84.01%	84.78%	85.60%	85.26%
Maximum Success	84.10%	84.74%	85.60%	85.81%	86.05%	85.73%

Table 6.6: Results obtained from 10 trials: DELU, He-truncated Gaussian and L2 Constant: 0.0002

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.02%	84.20%	84.27%	84.46%	84.95%	85.12%
Maximum Success	83.68%	85.10%	84.95%	85.94%	85.58%	85.71%

Tables 6.4-6.6 show that L2 regularization gives better results than L1 regularization, but this does not bar the way in which they are used in combination at the training phase. However, the regularization constants need to be re-adjusted to prevent over-regularization and punish the network for real learning, rather than memorization. With this idea, we'll move to L1 and L2 regularization part.

L1 (λ_1) and L2 (λ_2) regularization is used given in (6.3) and they are chosen as given in Tables 6.7-6.9

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + \lambda_1\Sigma_{i=1}^{all}|w_i| + (1/2)\lambda_2\Sigma_{i=1}^{all}(w_i)^2 \tag{6.3}$$

Table 6.7: Results obtained from 10 trials: DELU, He-truncated Gaussian, λ_1: 0.00001, λ_2: 0.00005

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.75%	83.86%	84.18%	84.69%	85.19%	85.17%
Maximum Success	83.45%	84.65%	84.88%	85.64%	86.07%	85.95%

Table 6.8: Results obtained from 10 trials: DELU, He-truncated Gaussian, λ_1: 0.000005, λ_2: 0.000025

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.24%	84.41%	84.67%	84.77%	85.75%	85.48%
Maximum Success	84.01%	85.08%	85.25%	85.37%	**87.09%**	86.26%

Table 6.9: Results obtained from 10 trials: DELU, He-truncated Gaussian, λ_1: 0.0000025, λ_2: 0.0000125

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.18%	84.30%	84.56%	85.16%	85.29%	85.30%
Maximum Success	84.07%	84.65%	85.22%	85.74%	85.98%	86.15%

Compared to previous results, L1 and L2 regularization combined is better than both regularizers individually, especially couple of second regularization

constants, named as the 'L1 and L2 Regularization-Case 2' (λ_1: 0.000005, λ_2: 0.000025) and couple of third regularization constants, named as the 'L1 and L2 Regularization-Case 3' (λ_1: 0.0000025, λ_2: 0.0000125). The mean successes of these are already over the previous ones, but the maximum success rate (87.09%) is best in this chapter. However, mean success rates are more significant for success of neural network that classifies CIFAR-10 dataset. Next, we proceed to Max-Norm regularization for increasing these.

Max-Norm Regularization is described in [27]. It's used here to limit the input weight vectors only, and its details are given in the below algorithm generally. Max-Norm Regularization constant is abbreviated as Max-Norm Limit (MNL), and selected from a range between 0 and 5. However, both Case 2 and Case 3 of L1 and L2 regularizers are tried with these MNLs. Only the input layer of the networks were subject to Max-Norm regularization. Results are given in Tables 6.10-6.12, for each MNL value.

***Algorithm* 6.1 :** Max-Norm Regularization Algorithm.

Every input weight vector
for Each element of the weight vector W **do**
1. Calculate square of the weight vector element
2. Add this squared value to the summation

end for
Name the summation as 'Value'
if MNL is bigger than 'Value' **then**
MNL is set as 'Nominator'
else
'Value' is set as 'Nominator'
end if
Sum 'Value' with Keras' Epsilon (10^{-8}), call this 'Denominator'
Set the fraction by 'Nominator' and 'Denominator', call this 'Coefficient'
for Each element of the weight vector W **do**
1. Multiply weight vector element with 'Coefficient'

end for
Resulting weight vector is Max-Norm regularized new weight vector

Table 6.10: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 2, MNL: 0.25

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.04%	83.64%	84.36%	85.13%	85.15%	85.17%
Maximum Success	83.65%	84.33%	85.13%	85.82%	85.66%	85.88%

Table 6.11: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 2, MNL: 0.5

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.17%	83.91%	84.26%	85.04%	85.55%	85.72%
Maximum Success	83.79%	85.03%	85.08%	85.36%	86.10%	86.76%

Table 6.12: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 2, MNL: 1

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.39%	83.89%	84.54%	84.99%	85.48%	85.50%
Maximum Success	84.47%	84.94%	85.15%	86.07%	86.48%	86.45%

As Tables 6.10-6.12 show, the finest MNLs perform slightly below the networks without Max-Norm, which use Case 2 as L1 and L2 regularization constants. The expected success rate increase is not present due to possible over-regularization, and this will be discussed in the sequel.

Because of used small MNL values in this algorithm (0.25, 0.5, 1), we lowered L1 and L2 regularization constants and used Case 3, whose results are given below.

Table 6.13: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 3, MNL: 0.25

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.99%	83.85%	84.28%	84.75%	85.24%	85.14%
Maximum Success	83.73%	84.62%	84.86%	85.39%	85.80%	86.26%

Table 6.14: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 3, MNL: 0.5

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.51%	84.18%	84.72%	85.24%	85.64%	**85.78**%
Maximum Success	84.62%	85.33%	85.86%	85.98%	**86.42%**	86.41%

Table 6.15: Results from 10 trials: Used DELU, He-truncated Gaussian, Case 3, MNL: 1

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.41%	84.14%	84.45%	84.99%	85.33%	85.20%
Maximum Success	84.04%	84.61%	84.97%	85.89%	86.04%	85.81%

As seen from these Tables 6.13-6.15, lowering the L1 and L2 regularization constants increased the success rates obtained; especially for the mean success rates. Given the nature of Max-Norm regularization process as described previously, it is natural that lowering the strength of L1 and L2 regularization yields better results.

Until this point, different well-known regularizers such as L1 and L2 and other ones from the literature such as Max-Norm are used, independently and/or combined. They gave significant success rates such as 87.09% for the maximum and 85.78% at mean success rates. Although they are the best percentages obtained in this chapter, it is important to mention other custom regularizer attempts briefly.

6.2 Custom Regularizers

The first attempt is to merge the nature of both L1 and L2 regularizers, named as L1.5 regularizer. Its formula is given below, where the addition term with $\lambda_{1.5}$ differs as expected:

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + \lambda_{1.5}\Sigma_{i=1}^{all}(\sqrt{|w_i|})^3 \tag{6.4}$$

This regularizer did not perform well and its success rates fell around 80%.

The second attempt is to go beyond the L1 and L2 regularizers, named as L3 regularizer. Its formula is given below, where the addition term with λ_3 differs as expected:

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + \lambda_3\Sigma_{i=1}^{all}(|w_i|)^3 \tag{6.5}$$

This regularizer did not perform well and its success rates fell around 80% too, in addition to some failing networks whose success rates fell around 10% due to instability caused by this new regularization formula.

The third attempt is to use an idea different than L1 and L2 regularizers, named as LE regularizer. It uses exponential function instead of power or other functions, and the resulting formula is given below, where the addition term with $\lambda_e = 0.000000002$ differs with its smallness to ensure stability as needed:

$$E = (1/2)\Sigma_{i=1}^{all}(d_i - o_i)^2 + \lambda_e \Sigma_{i=1}^{all} e^{|w_i|} \tag{6.6}$$

This regularizer did not perform well - although it was better than other two custom regularizers - and some of the networks still failed, sometimes going down to 10% classification rates.

In summary, although a lot of ideas found and time is devoted for experiment, custom regularization attempts did not give a solid result better than the current combination of classical regularizers (L1, L2 and Max-Norm). It may be because of the complex nature of proposed regularizers which may need extra effort to tune, or simply their inefficiency. Thus, the custom regularizers part of this chapter is concluded with these results.

In conclusion of this chapter, the mean and maximum success rates obtained at the end of previous chapter are increased with addition of finely tuned classical regularizers, which yielded in a boost over 1% at the mean success rate and 1.75% at the maximum success rate. Also, custom regularizer from outside of the existing literature is experimented but did not give better results compared to these ones. In the next chapter, the network layers will be modified for obtaining better success rates, which starts with modifying the Dropout layers - which can also be considered as similar to regularizers.

Chapter 7

Network Layers

Every neural network contains layers. One or more, it is those layers which define the network itself. Without them, the neural network would not be existent. These layers contain different kinds such as feedforward, convolutional, max-pooling, batch normalization and Dropout (which can also be considered as a regularizer instead of a layer) which are used in this part, and the formation of some of these are modified in order to obtain better success rates. In this manner, a more successful neural network is expected which travels through the error surface better than their predecessors.

From the beginning of neural networks, different layers are used - similar to our work here. First neural networks contained only a single feedforward layer [4], and the convolutional layers were also introduced in early 1980's [5]. At these times, convolutional layers were inefficient to implement as they required much more processing power in conventional CPUs; however, after the millenium (especially around 2010's) multi-threaded CPUs and GPUs' usage in the neural network field paved the way of convolutional neural networks faster training. In addition to those, Max-Pooling layers decreased the computational complexity while keeping the efficiency of the convolutional neural networks. With the addition of Dropout and Batch Normalization (or Renormalization, which is a better version as of today), organizations of current neural networks are almost formed.

In this chapter, formations of different layers are investigated in different parts. In the first part, Dropout layers of the convolutional neural network which are used for enhancing the learning process are examined, with different alternatives proposed in terms of used probability values and working mechanism of this phenomenon. In the next part, default Max-Pool layer formation of the convolutional neural network is investigated and several candidates are proposed to train the network better. In the last part of the chapter, additional layers which are used for enhancing the learning process, such as Batch Normalization layers, are utilized for attempting to peak the classification success of the convolutional neural network which classifies the CIFAR-10 dataset.

The neural networks in this chapter worked with following attributes only (except different layer formations): Original SinAdaMax optimizer (see Chp. 4), Categorical Cross-Entropy loss (see [19]), He-truncated Gaussian initialization (see Chp. 2), DELU activation function (see Chp. 5) except Softmax output layer (see [15]), L1 and L2 Regularization with Case 2 or Case 3 and Max-Norm Regularization with 0.5 coefficient or no Max-Norm regularization (see Chp. 6), Mini-Batch Learning (see [16]) with 128 samples in each batch, and training is completed in 300 epochs with 50 epoch intervals. Training is done with network at Figure 1.3 (Structure 1.1, initially) using first 4 data batches of CIFAR-10 dataset, and validation is done with remaining fifth data batch (each data batch contains 10000 samples and shuffled randomly). From 10 trials with this configuration, mean and maximum success rates are obtained for comparison of success rates.

7.1 Dropout-based Layers

Dropout is a widely used technique which first appeared in 2014 in [27]. It proposed a different learning technique, by dropping some random neurons with all their connections during the training iteration, while all neurons are present at the test iterations. In this way, neurons are forced to share their specific recognition tasks and they train better, to compensate the lack of their neighbor neurons.

In this part, there are two different attempts for producing a more efficient network structure. One of them is tuning all of the existent Dropout layers with a probability value, p, which is varied from 0 to 0.625 with 0.125 spacing to maximize the success rate obtained. The other part contains the replacement of Dropout layers at the feedforward part with newly founded DropConnect layer, which yields problematic results. We will start with the first attempt.

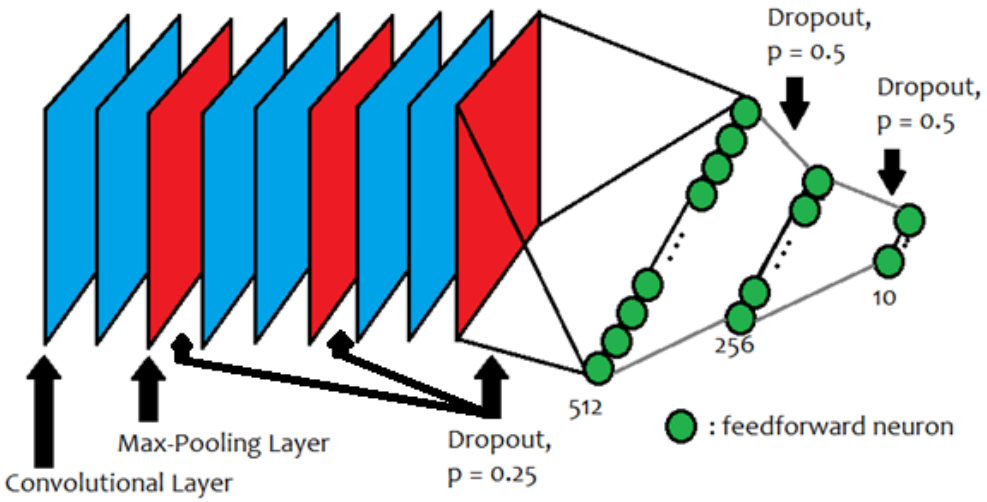

Figure 7.1: The convolutional neural network's general structure is displayed.

Until this point, all the convolutional neural network structures used for experiments had two different Dropout layers: The ones between the convolutional layers (with 0.25 p value) and the ones between the feedforward layers (with 0.5 p value). Their exact placement can be seen in the Figure 7.1. However, to figure out the best probability values for network, these probability values should be varied. These attempts are done with separate convolutional neural networks which have different regularizations applied: The first one contains only L1 & L2 regularization-Case 2 (L1 const.: 0.000025, L2 const.: 0.00005), the second one contains Max-Norm regularization in addition to L1 & L2 regularization-Case 2, and the third one contains L1 & L2 regularization-Case 3 (L1 const.: 0.0000125, L2 const.: 0.000025) while keeping the Max-Norm regularization with 0.5 limit coefficient. The finest probability value is decided with first two sets of neural networks, and this value is applied to last set of neural network for obtaining the best combination of finest Dropout value and regularizer set. The Tables 7.1-7.3,

which are given below, contain the results of best three probability values for Dropout: 0.25, 0.375 and 0.5. Other values (0, 0.125, 0.625) gave worse results which are not needed to discuss here.

Table 7.1: Results from 10 trials: L1 and L2 Regularization with Case 2, No Max-Norm, Dropout Probability: 0.25

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.39%	84.25%	84.79%	85.31%	85.81%	85.44%
Maximum Success	84.96%	85.33%	85.50%	85.93%	86.28%	86.14%

Table 7.2: Results from 10 trials: L1 and L2 Regularization with Case 2, No Max-Norm, Dropout Probability: 0.375

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.38%	84.50%	85.28%	85.54%	85.93%	85.80%
Maximum Success	84.59%	85.18%	86.10%	85.97%	86.42%	86.74%

Table 7.3: Results from 10 trials: L1 and L2 Regularization with Case 2, No Max-Norm, Dropout Probability: 0.5

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.65%	84.32%	85.07%	85.12%	85.77%	85.62%
Maximum Success	83.59%	85.27%	86.10%	85.79%	86.27%	86.16%

As seen from above tables, L1 and L2 regularization-Case 2 without Max-Norm regularization yields the best probability value as 0.375. Afterwards, we implemented these three probability values of L1 and L2 regularization-Case 2 with Max-Norm regularization (MNL: 0.5), with limit coefficient is 0.5, where 0.375 probability value proved itself as the best one again amongst the other values as seen in the Tables 7.4-7.6.

This value acquired by the experiments conducted with L1 and L2 regularization-Case 2 is carried to convolutional neural network with L1 and L2 regularization-Case 3, and their combined results are given in Table 7.7.

Table 7.4: Results from 10 trials: L1 and L2 Regularization with Case 2, Max-Norm with MNL: 0.5, Dropout Probability: 0.25

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.04%	84.41%	84.86%	84.96%	85.16%	85.07%
Maximum Success	83.92%	85.73%	85.74%	85.52%	86.41%	85.69%

Table 7.5: Results from 10 trials: L1 and L2 Regularization with Case 2, Max-Norm with MNL: 0.5, Dropout Probability: 0.375

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.27%	84.87%	85.16%	85.71%	86.03%	85.93%
Maximum Success	83.96%	86.15%	85.72%	86.73%	86.80%	86.54%

Table 7.6: Results from 10 trials: L1 and L2 Regularization with Case 2, Max-Norm with MNL: 0.5, Dropout Probability: 0.5

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	82.30%	83.69%	84.66%	85.05%	85.70%	85.89%
Maximum Success	82.88%	84.48%	85.72%	85.91%	86.24%	86.64%

Table 7.7: Results from 10 trials: L1 and L2 Regularization with Case 3, Max-Norm with MNL: 0.5, Dropout Probability: 0.375

Number of Epochs	**50**	**100**	**150**	**200**	**250**	**300**
Mean Success	83.60%	84.49%	85.17%	85.41%	85.90%	86.26%
Maximum Success	84.80%	85.03%	85.71%	86.40%	86.38%	86.97%

As seen from the above result, the mean and maximum success rates are the best obtained in this thesis until this chapter. The finest value, when set as global probability value amongst all the Dropout layers in the convolutional neural network, is 0.375 as the experiments suggest. However, one can propose different Dropout layer probabilities which varies according to order of layers, or its presence in the convolutional or feedforward part of the network as a future work, which was not considered in this thesis work so long as a significant increase of success is obtained (0.48% in mean and 0.55% in maximum success rate). The second attempt in this part is to replace Dropout layers with the DropConnect layers, at the feedforward part of the convolutional neural network.

In this attempt, the DropConnect implementation for Keras is obtained from the work [28], and it is applied only to feedforward layers. The DropConnect's working mechanism is explained in paper [29], where the main idea is same with the Dropout except one thing: This time some of the connections are randomly dropped out, instead of specific neurons with thier connections. In this manner, the efficiency of learning is aimed to be increased.

This mechanism is applied to the convolutional neural network which classifies the CIFAR-10 dataset, and the few results taken from the experiment showed the initial lack of success. Although same p value (0.375) was used for the experiment, success rates significantly dropped from 87% (while Pyramid Max-Pooling -can be seen in next section- was in place) to around 85% (sometimes the success falls around 10% - instability) Because of incompatibility of Keras -the inexistence of DropConnect application between convolutional layers- and time-consuming nature of this problem, DropConnect's integration is left as a future work, which

can be done in another research. Now, the next part of this chapter, Max-Pooling Layers will be discussed in detail with results.

7.2 Max-Pooling Layers

Max-Pooling (or Max-Pool) layers are widely used in convolutional neural networks, so there were three max-pooling layers we utilized in the convolutional neural network given in [9], which is modified and used throughout the thesis. All of them are 2x2 sized max-pooling layers, which concentrate on parts of input image (or interior output) and output the highest value as a 1x1 volume, decreasing the number of parameters and the complexity for the whole of the neural network.

Here, different max-pooling layers are experimented with different formations. Apart from default "2x2 - 2x2 - 2x2" formation illustrated in Figure 7.2 which is given below, two more candidates are applied to the network: Pyramid ("1x1 - 2x2 - 3x3") and reverse pyramid "3x3 - 2x2 - 1x1"). Note that Max-Pool layer with size 1x1 is indeed, equal to linear activation (an ineffective layer). Starting from input side of network and going through output side of network, those are the three max-pooling layer formation which are experimented with the convolutional neural network.

As seen in Figure 7.2, the default convolutional neural network structure has 3 Max-Pooling layers with 2x2 size, each of them are located after two consecutive convolutional layers. There are no Max-Pooling layers in the feedforward layer as usual. Those Max-Pooling layers significantly decrease the number of parameters needed for the network, in addition to lowered computational time, therefore their existence is necessary - one way or another.

Instead of this constant size 2x2 Max-Pooling layers, we decided to propose different sized Max-Pooling layers as described before. The first attempt was to implement Max-Pool layers with decreasing size (3x3, 2x2 and 1x1). Network

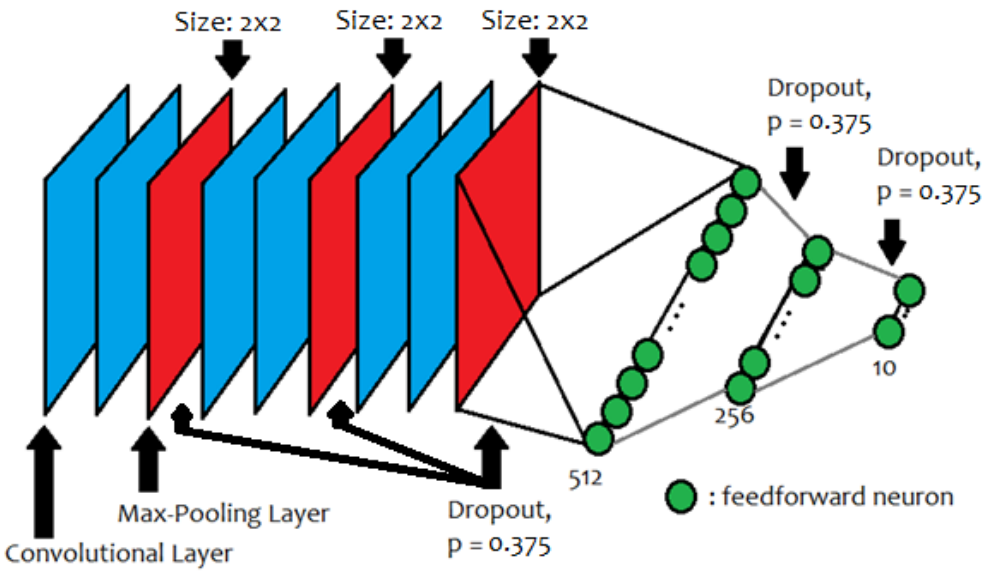

Figure 7.2: The convolutional neural network's general structure is displayed with details of Max-Pool layers.

structure built with this approach is given below at the same page, in Figure 7.3.

However, worse results are obtained with this new Max-Pool layer structure. Compared to result of default Max-Pool layer structure (Mean: 86.26%, Maximum: 86.97%, 10 trials), new structure has significantly lower success rates (Mean: 84.74%, Maximum: 84.93%, 20 trials). Although fewer number of trials are conducted, there is no need to do the remaining ones; because evidently this approach makes features of the input patterns (edges, circles etc.) ineffective (for example, merging nose with eye activation region). Therefore, experiments for this approach are abandoned and reverse approach is implemented.

In this approach, sizes of Max-Pool layer are reversed to capture the activations better, which are triggered by image regions (such as nose, eye etc.). Instead of pooling the activations for different image regions at the beginning, we increased the pooling at the end which is more meaningful for network to make the decision: Pool the feature activations and decide with the stronger activations. In this way, redacted image data is less decreased at the beginning (allows the network to learn more) and number of parameters are kept same with the previous approach.

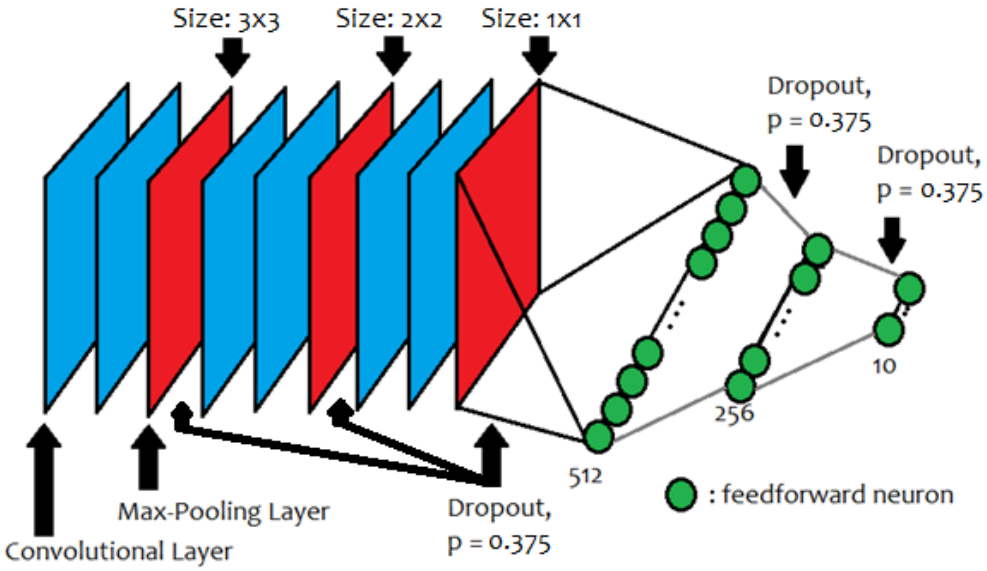

Figure 7.3: The convolutional neural network's general structure is displayed with details of Max-Pool layers, for new approach: Reverse pyramid (3x3 - 2x2 - 1x1).

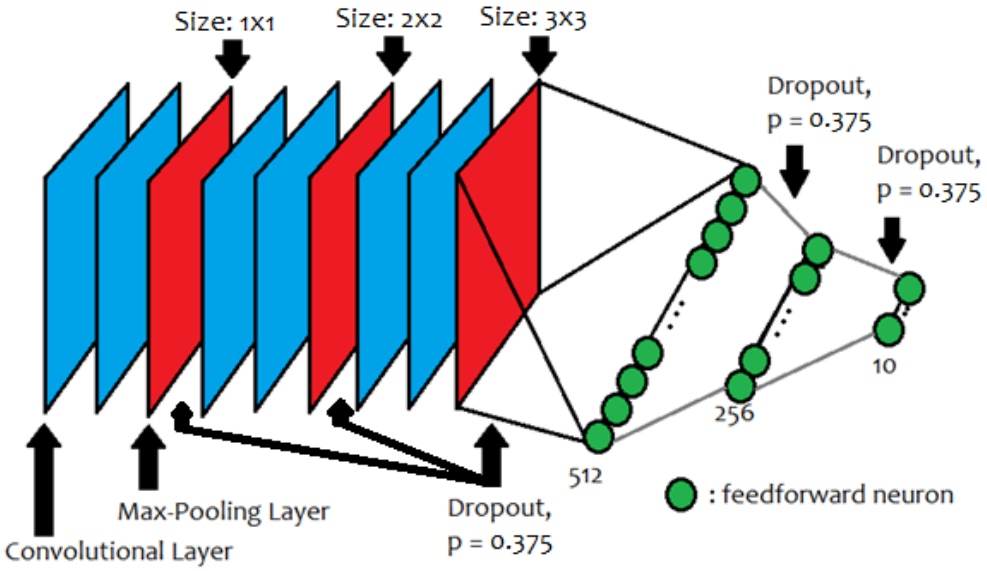

Figure 7.4: The convolutional neural network's general structure is displayed with details of Max-Pool layers, for last approach: Pyramid (1x1 - 2x2 - 3x3).

Although the number of parameters nearly doubled compared to default Max-Pool layer setting, increase of 1.25% for mean and 1.49% for maximum success rate seems to be worth it (Mean: **87.51%**, Maximum: **88.46%**, 10 trials).

In result, instead of using constant size Max-Pooling layers to lower the complexity of the network and make the training faster, implementing Max-Pool Layers with 'Pyramid Approach' (decision for its name is inspired from the pyramid shape; smaller sizes at the top and larger sizes at the bottom layers) gives better results in image classifying convolutional networks. In addition to CIFAR-10 networks, MNIST networks also gave better results with this approach - the details can be found in relevant chapter. As a future work, Average-Pooling layers can also be investigated and added to neural network, for being used with the Pyramid Approach. Now, we will move on the other types of layers which are used in neural networks.

7.3 Other Layers

The layers used in convolutional neural networks are not limited with the convolutional, max-pool or feedforward layers. There are also different layers which can be used to increase the test success rate, and an important one is the Batch Normalization layer which is applicable in Keras. Batch Normalization is proposed in work [30], and briefly it aims on normalizing the neuron activation to avoid saturation (as known as exploding) of neuron structures, by normalizing all activations that neuron made in a batch, by calculating mean and variance of those activation for normalization calculations. In this way, learning is aimed to be accelerated by activating formerly inactive neurons.

In our case, we tried a Batch Normalization layer instead of the ineffective 1x1 Max-Pool layer. Some results are obtained with this formation, for 8 trials instead of 10. That is because of the failing of neural network structure (2 of them at very beginning, 3 of them occured afterwards). The ones that can reach to the end of the training reach to 87.15% mean success rate, which is a less success rate; and

with the addition of those failures, it seems that Batch Normalization did not fit well to this neural network structure. Therefore, experiments with these layers are interrupted for trying other methods for the thesis work. Later, network with more Batch Normalization layers can also be experimented for trying to increase the success rate.

As a future work, Batch Renormalization [31] can be proposed. The only difference of Batch Renormalization from its precedent, Batch Normalization, is that new one works with all batches altogether instead of calculations restricted to each sample batch. Although it does not exist in the in-built Keras library, it can be implemented for trying with our convolutional neural network.

In result, we worked on different layers for the convolutional neural network. We modified the default Dropout probability values and tuned them to reach higher success rates, while tried to implement the newer version: DropConnect. After that section, we moved to the formation of Max-Pooling layers and tried to change them, where 'Pyramid Approach' emerged as a successful method to increase the success rates. Lastly, other adiditonal layers (Batch Normalization and Renormalization) are reviewed for a possible integration, however as the Batch Normalization implementation with one layer did not turn out well, this section and the chapter is concluded with a significant increase of mean success rate (1.73%) and maximum success rate (2.04%) compared to best results of the previous chapter.

Chapter 8

Data Preprocessing and Augmentation

Every neural network works with data. Training methods of the neural network do not have to contain Data Preprocessing or Data Augmentation but if they contain a finely chosen set of those techniques, their test success rates rise significantly. This increase is accompanied with a decrease of training success, similar to effect of regularization in neural networks. After this process applied to input data, the neural network structure nearly maximizes its learning capability.

From the beginning of artificial neural networks, data was present. However, intentional preprocessing and augmentation of data specifically for neural networks appeared in the papers after 1990s [32], [33]. Afterwards, usage of data preprocessing and data augmentation spread to the field, given the increase of test success and lack of data available. Although the data has increased enormously during this decade, data preprocessing and data augmentation still used widely today, because of its efficiency on the learning process of the neural networks.

In this chapter, data preprocessing and data augmentation parts are separately written although their similar effects on training and test success rates. In first section, applied data preprocessing techniques are presented alongside other

possible methods of preprocessing, while results obtained are discussed briefly. In second section, tuned data augmentation techniques are applied as different combinations, and results obtained are acquired with the Data Preprocessing techniques carried from first section.

The neural networks in this chapter worked with following attributes only (except different data preprocessing and augmentation methods): Original SinAdaMax optimizer (see Chp. 4), Categorical Cross-Entropy loss (see [19]), He-truncated Gaussian initialization (see Chp. 2), DELU activation function (see Chp. 5) except Softmax output layer (see [15]), L1 and L2 Regularization with Case 2 or Case 3 and Max-Norm Regularization with 0.5 coefficient or no Max-Norm regularization (see Chp. 6), Mini-Batch Learning (see [16]) with 128 samples in each batch, and training is completed in 300 epochs with 50 epoch intervals. Training is done with network at Figure 7.4 this time (Structure 1.1 is varied here: p's are set to 0.375 and Max-Pool layers are set with Pyramid Approach), using 5 data batches (except first 1000 samples, which are used for validation) of CIFAR-10 dataset, and test successes are botained with test data batch (each data batch contains 10000 samples). From 2 or 10 trials with this configuration, mean and maximum success rates are obtained each trial independently, to compare of validation success rates of each network in itself, and the one with least validation error (for instance, 150th epoch result for a 300 epoch training) sets its test success as the test success of the whole network.

8.1 Data Preprocessing

In this chapter, there are two different set of preprocessing techniques. One of them is the default preprocessing techniques recommended by the dataset creators, the other is our own applications and recommendations selected from the literature. No data augmentation is used in this section, like the previous parts of the thesis work. We'll start with the first part.

The CIFAR-10 dataset is used from the website [34]. In the .bin files

Krizhevsky posted, input images are not normalized but left as pixel values (0 to 255). First, this data (both training and test set) is normalized between 0 and 1. Second, a more symbolic one, is getting the desired output values as numbers from decimal numbers and converting them to one-hot encoding type. After this standard preprocessing for each of the data samples, the training starts.

In addition to that default preprocessing, there are different ones which have an effect on the training process. As an example, centering of the data onto 0.5 (by normalizing the samples between 0 and 1) can be modified. In this part, we will try different center values and observe their effects.

The first idea is to normalize the image samples between -0.5 and 0.5. This corresponds to centering the data onto zero, and increases the number of neurons at the left-half-plane region (see Figure 5.11), which is expected to make the training slower. Indeed, it causes such an outcome: The training success of the convolutional neural network drops to 93%, from 99%. The test success, in this two-trial experiment, barely exceeds 88% and this means that not a significant increase of success. To figure out the finest value of centering, we decided to scan through [0-0.5] interval.

The next value to use was 0.25, which means that the input images are normalized between -0.25 and 0.75. This value yielded higher test success and higher training success in 300 epochs (with 88.11% mean success rate) compared to previous one, although training success is still lower than networks without the data preprocessing. This phenomenon bears similarity with regularizers indeed: Lower training success and higher test success is their typical behavior. Although those numbers obtained from two trials are enough for us to continue, we tried another value as centering position.

The last value to use was 0.375, which means that the input images are normalized between -0.125 and 0.875. Indeed, it is closer to the original 0.5 centering value but still has some values below zero, which yields in training success around 96% - higher than the 0.25 case. The mean of the test success rates obtained in

two trials results is 87.61%, which is higher than network without data preprocessing but less than previous result. However, this value will be the one that we'll use in the next section as the centering value. At the next section, data augmentation also causes training success rates to fall - just like data preprocessing technique and regularization phenomenon. Because of this situation, to avoid over-regularization of the neural network, 0.375 is selected as a centering value with the consideration of incoming similar regularization behavior of the data augmentation methods.

Apart from this centering methods, there are also different data preprocessing methods. One of them is making an RGB to HSV transformation for raw images and modifying the Saturation and Value components, the other is applying PCA Whitening (which is a method borrowed from linear algebra) for enhancing the learning process [33]. As a future work, these techniques can also be applied and respective results can be compared for discussion. For now, we will continue with the Data Augmentation section.

8.2 Data Augmentation

In this section, all of the techniques are applied for convolutional neural networks with data preprocessing applied as centering to 0.375 (or in other words, normalizing the data between -0.125 and 0.875). Data Augmentation methods are applied in different combinations, where some the methods are tuned throughout the process. In result, a fine combination is selected which results in a significant increase of test success only in 300 epochs.

Networks with the specific data preprocessing we used reach around 88% mean success rate as shown in previous section, without data augmentation. To increase this success rate, we applied four different sets of data augmentation methods. Although there are 20 augmentation methods available in Keras documentary, our time and capability did not allow us to try all of them one by one. Instead, a good set of methods are picked up from the Keras documentary (which are and

also inspired from sources) and tried with 2 trials to obtain mean success rates roughly. An example data augmentation output is given for a different dataset in the Keras documentary [35], in Figure 8.1. From the ones that are applied in these example images, we used only a small subset of techniques with different ranges: Rotation, height and width shift, zooming and horizontal flipping.

Figure 8.1: Augmented images with rotation, height and width shift, zooming, shearing, horizontal flipping and nearest fill mode options.

The details of the first data augmentation application for each sample is as follows: A random rotation with degrees up to 90, a random height shift range between -0.1 and 0.1, a random width shift range between -0.1 and 0.1, and randomly horizontal flipping of the image. This data augmentation yields 82% test success with only difference with only data preprocessing network was data augmentation. To compensate this failure, we decided to tune the data augmentation by lowering the rotation range limit to half.

The details of the second data augmentation application for each sample is as follows: A random rotation with degrees up to 45, a random height shift range between -0.1 and 0.1, a random width shift range between -0.1 and 0.1, and randomly horizontal flipping of the image. This data augmentation yields 87% test success. So long as the problem is detected as large rotation range allowed for data augmentation process, we decided to tune the data augmentation by lowering the rotation range furthermore.

The details of the third data augmentation application for each sample is as

follows: A random rotation with degrees up to 5, a random height shift range between -0.1 and 0.1, a random width shift range between -0.1 and 0.1, and randomly horizontal flipping of the image. This data augmentation yields more than 90% test success, which is best of all results obtained in this thesis work until this point. Indeed, this was the most successful network obtained in this chapter, therefore all 10 trials are conducted for the discussion of the next chapter: **90.40%** as the mean success rate and **91.03%** as the maximum success rate. The jump of mean and maximum success rates by a factor of 2-3% is highly significant and makes the CIFAR-10 dataset exceed the 90% test success threshold (for maximum success rates, 91% test success threshold). In this manner, our advanced methods proved themselves for this neural network structure (5-6% test success rate increase compared to original network results [9]), but we will try one more combination for the sake of completeness of this section.

The details of the fourth data augmentation application for each sample is as follows: A random rotation with degrees up to 5, a random height shift range between -0.1 and 0.1, a random width shift range between -0.1 and 0.1, a zoom range from 0.8 to 1.2, and randomly horizontal flipping of the image. This data augmentation yields 89% test success, which is pretty lower than previous data augmentation application. Therefore, the third data augmentation application is chosen as the combination that will prevail to next section.

In conclusion, different data preprocessing and data augmentation techniques from the literature are experimented. In first part, a convenient amount of center shift is chosen (form 0.5 to 0.375) to avoid over-regularization of neural network while being used with third data augmentation application, and 90% threshold for mean success rates are exceeded by our convolutional neural network. In next chapter, different learning conditions are exercised for increasing this test success more.

Chapter 9

Learning Conditions

Every neural network have a specific learning condition (or training condition) in the training phase. It is defined with the size of the mini-batch (or batch), number of epochs etc. The success of the convolutional neural network heavily depends on the selection of those conditions, where poor choices can lead to failing neural networks. After those are set finely, the potential of the neural network structure can show itself completely.

From the beginning of artificial neural networks, training are conducted with different conditions. Some were based on one sample at a time mechanism, some were based on using the whole data at a time which is named as batch learning [22]. Each technique were accompanied by different iteration and/or epoch numbers for reaching the optimal success rates. Apart from sample amount, existence or nonexistence of some techniques from some point of training can also be considered as the adjustment of learning conditions.

In this chapter, we considered three different learning conditions. The first section contains the varying of the training batch sizes during the training phase, which yielded interesting and significant results. The next section is 'Segmented Learning', which is a practice inspired from real-life with some results obtained. The last section, named as Training Epochs and Duration, which contains results

for training with different amounts of epochs and information about durations of the trials.

The neural networks in this chapter worked with following attributes only (except learning conditions): Data centered on 0.375 as Data Preprocessing, Third Data Augmentation Application, Original SinAdaMax optimizer (see Chp. 4), Categorical Cross-Entropy loss (see [19]), He-truncated Gaussian or Laplacian initialization (see Chp. 2), DELU activation function (see Chp. 5) except Softmax output layer (see [15]), L1 and L2 Regularization with Case 2 or Case 3 and Max-Norm Regularization with 0.5 coefficient or no Max-Norm regularization (see Chp. 6), Mini-Batch Learning (see [16]) with 64, 128 or 256 samples in each batch (varies in chapter), and training is completed in 300 or 600 epochs with 50 or 100 epoch intervals. Training is done with network at Figure 7.4 this time (Structure 1.1 is varied here: p's are set to 0.375 and Max-Pool layers are set with Pyramid Approach), using 5 data batches (except first 1000 samples, which are used for validation) of CIFAR-10 dataset, and test successes are obtained with test data batch (each data batch contains 10000 samples). From varying number of (mostly 10) trials with this configuration, mean and maximum success rates are obtained each trial independently, to compare of validation success rates of each network in itself, and the one with least validation error (for instance, 150th epoch result for a 300 epoch training) sets its test success as the success of the whole network.

9.1 Training Batch Size

For every training to be proper, a good size of mini-batch (interchangeably batch) should be selected. It is necessary for better learning and feasible training durations, where this two are mostly competing goals in which we try to optimize both. In this section, brief results of training with three different batch sizes (64, 128 - default - and 256) are comparably discussed first, then specific batch size formations (constant size - default -, expanding size and shrinking size) are discussed in detail.

The default mini-batch size, 128, was used until this point of the thesis work. To find the optimal batch size, we scanned the interval basically, by trying double (256) and half (64) values of the batch sizes. In this way, we obtained results which are used for the next part of the section. The mini-batch size 64 naturally doubles the training duration which is a heavy cost, however it seems to increase the test success rates slightly. In contrast, mini-batch size 256 lowers the training duration significantly (about the half of the default one) while test success slightly fall this time. But still, all test successes are around 88-89% - and exceptionally for this part, results are taken without Data Preprocessing and Data Augmentation so that success rates can seem appropriate.

To compensate on this training duration problem, a hybrid approach is proposed. By dividing the training phase into 3 equal parts with 100 epochs in each part, different batch sizes are applied. The default batch size formation, for each 50 epochs, was simply 128-128-128-128-128-128 which is named as 'Constant Size Batches'. The other two formations are named as 'Shrinking Batches' and 'Expanding Batches', which are respectively 256-256-128-128-64-64 and 64-64-128-128-256-256 for again each 50 epochs. In the Table 9.1, their results can be observed for training with both data preprocessing and data augmentation this time. Compared to previous results, one of the 10 trials for 'Shrinking Batches' has failed in first 50 epochs, so its result is excluded from the fold, so long as this batch size formation will not be used from this point of the chapter.

Table 9.1: Results from 10 trials (He-truncated Gaussian): 'Shrinking Batches', 'Constant Size Batches' and 'Expanding Batches' are used for 300 epochs training.

Formation of Batch Sizes	**Shrinking**	**Constant Size**	**Expanding**
Mean Success	89.94%	90.40%	**90.82%**
Maximum Success	90.85%	91.03%	**91.36%**

As seen from the table, the best success rates are obtained by 'Expanding Batches' formation, where 0.42% mean success rate and 0.33% maximum success rate increase are present which is significant. This may result from sensitivity of the neural network at the end phases of training, where larger batch size decreases sensitivity to independent samples. One can also claim that fine tuning the neural

network at the beginning may result in a better training situation, but all those claims should be verified by experiments on another well-known dataset so long as they are not mathematically proven or completely straightforward for the field of neural networks. Despite this, we decided to move on with 'Expanding Batches' formation to the next section.

To conclude this section, let us also look at the results obtained with He-truncated Laplacian initialization. So long as we're coming to the end of thesis work, it is beneficial to review the behavior of this initialization and see whether it can pass the 90% threshold for CIFAR-10 dataset, to be an alternative initialization method. The results are given in Table 9.2 below.

Table 9.2: Results from 10 trials (He-truncated Laplacian): 'Constant Size Batches' and 'Expanding Batches' are used for training with 300 epochs.

Formation of Batch Sizes	**Constant Size**	**Expanding**
Mean Success	89.93%	90.35%
Maximum Success	90.85%	91.12%

As seen from this table, test success rates with 'Expanding Batches' for He-truncated Laplacian initialization passes 90% threshold, however it is lower than He-truncated Gaussian results with a margin less than 0.5% percentage (much lower for maximum success rates). This means that He-truncated Laplacian can still be considered as an alternative initialization method at the end, but not a primary one compared He-truncated Gaussian initialization. Now, we can proceed to next section.

9.2 Segmented Learning

In this section, a different learning practice is proposed. Instead of starting to train the neural network with augmented samples of standard data augmentation, it is proposed that the first half of the training will only contain the data

preprocessing applied samples and the remaining part can contain the data augmentation applied samples in addition to data preprocessing. Thus, the duration of training is lowered while number of epochs were kept same. This practice indeed originated from a simple analogy: We considered the neural network as a student who prepares for the exam (test data). Instead of starting with medium to hard questions (data augmented samples), student starts with the easy questions (non-augmented samples) for the beginning half of the training (first 150 epochs), then switches to harder questions (for remaining 150 epochs) for an easier and better training to the exam. With this understanding, several results are obtained and compared to previous results.

Table 9.3: Results from 10 trials (He-truncated Gaussian): Default learning and segmented learning are used for training with 300 epochs with expanding batches.

Learning Type	**Default Learning**	**Segmented Learning**
Mean Success	**90.82%**	90.38%
Maximum Success	**91.36%**	91.13%

As seen from this table, test success rates with segmented learning configuration falls around 0.5% for mean success rate and less for maximum success rate. This means that as a future work, segmented learning settings can be altered to change the outcome and the opportunities can be explored, and there may be another combination which can retain the advantage of less training time needed while at least keeping the success rates same with default learning type. For now, we will proceed to last section of the chapter.

9.3 Training Epochs/Duration

In this section, number of training epochs (interchangeably training duration, for networks with same configuration) are modified to figure out their depth of effect on the test success rates. The default 300 epochs training is compared with 150 epochs and 600 epochs training results, which corresponds to half and double of

the default training duration roughly. From those results, a basic conclusion is reached.

Here, three different training durations are given. Namely 150 epochs, 300 epochs and 600 epochs results are compared with each other, where results of 150 epochs are taken from results of networks trained with 300 epochs. In the Table 9.4, results are given for each training duration except 600 epochs, which takes too much time and only 2 trials are successful for now.

Table 9.4: Results from 2 or 10 trials (He-truncated Gaussian): Same configuration is used training with except total number of epochs.

Number of Epochs	**150 (10 trials)**	**300 (10 trials)**	**600 (2 trials)**
Mean Success	89.59%	**90.82%**	**91.56%**
Maximum Success	90.46%	**91.36%**	**91.87%**

As seen from the table, the most successful results are obtained with the longest training duration 600 epochs, although only 2 trials are conducted with this number of epochs. However, the important takeaway from this table is that although training duration is doubled twice, the increase of mean success rate is lowered to 60% of the previous increase, which shows a slow but important saturation. And although this increase, we did not set the 600 epochs training as the best network configuration of thesis work, because other 2 trials conducted with this duration have fallen around 10% percent due to lack of available RAM memory in the GPU of workstation computer. Therefore, we will note down the mean and maximum percentages of both 300 epochs (in which such situation did not happen) and 600 epochs to show the final results obtained for CIFAR-10 dataset in our thesis work.

Chapter 10

Other Datasets and Results

Every neural network is capable of working with a different dataset. Not only image data, but also words (more generally semantic data), sound or video data can be classified, clustered or processed in another way; in our case the convolutional neural network can work with other image data. Specifically MNIST dataset, which is also classifiable by some feedforward neural network structures, is classified by our convolutional neural network (with a minor difference at input layer). Also, the methods used and developed in this thesis work can be employed in other well-known neural network structures to demonstrate their better performance and higher learning capability.

From the beginning of neural networks, different datasets are used. One of the first experiments are conducted with streaming bits of the telephone lines [4], and after that many different data are processed with neural networks. An easy but well-known one, the MNIST image dataset is constituted in work [36], and its popularity continued after the millennium increasingly. It contains grayscale images of the 10 digits, where 60000 samples are given for the training of the network and 10000 samples are given for the test purpose. Today, it is used as a toy dataset to test the neural network itself, and figure out any mistakes in the configuration if there is one - before dealing with more complex datasets. Another well-known but far more complex image dataset is ImageNet datasets which expands each

year, contains more than million image samples [37]. ImageNet datasets are not only used for classification tasks like MNIST and CIFAR-10 dataset, but also for the localization etc. tasks for other networks too. In this thesis work, possible implementations of the advanced methods to the ImageNet networks are discussed only.

In this chapter, there are two separate sections. In first section, convolutional neural network for CIFAR-10 dataset is modified for classifying the MNIST dataset by changing its input layer. After this change, the remaining neural network structure is first used as given in the internet source [9] to obtain a base success rate, then our advanced methods are applied with different validation set sizes to demonstrate the increase of the classification success and compare with the best results on the literature. In second section, as a future work, different types of implementations for advanced methods to increase the success rates are discussed.

10.1 MNIST Dataset and Results

In this section, MNIST dataset is classified with 4 different network applications. Generally, the convolutional neural network given at Figure 1.3 and Structure 1.1 are used with one significant exception: Input layer size is dropped to 28x28x1 from 32x32x3, so long as MNIST samples are 28x28 pixels and have only one color channel (grayscale). This significantly lowers the number of parameters needed for the convolutional neural network and naturally the training duration for the same number of epochs, although the number of samples are more than CIFAR-10 dataset by 10000. In addition to this modification, the 'default' neural network structure worked with following attributes only: Gaussian initialization with 0.05 standard deviation, Adam optimizer (a learning algorithm that uses backpropagation to modify network parameters, see [12]), Categorical Cross-Entropy loss (a mathematical formula that calculates loss values from desired and actual output values to be used in backpropagation, see example at [13]), ReLU activation function (a function that is used as seen in Chapter 1, for ReLU specifically see [14])

except Softmax output layer (see [15]), Mini-Batch Learning (a technique calculates losses per each batch and modifies the network after the batch, see [16]) with 128 samples in each batch, and training is completed in 300 epochs. Training is done with network at Figure 1.3 this time (Max-Pool layers with constant 2x2 size), using all 6 data batches (zero samples are used for validation) of MNIST dataset, and test successes are obtained with test data batch (each data batch contains 10000 samples). From 2 trials with this configuration, mean, minimum and maximum success rates are obtained each trial independently. Here, no validation samples are present, therefore final network after the 300th epoch is used for testing phase.

After this 'default' neural network configuration, the advanced methods are applied to the convolutional neural network with different validation set sizes. Apart from the original neural network structure given in the internet source [9], neural networks in this section worked with following attributes only (except different validation set sizes): No Data Preprocessing, No Data Augmentation or MNIST Data Augmentation Application (see below paragraph), Original SinAdaMax optimizer (see Chp. 4), Categorical Cross-Entropy loss (see [19]), He-truncated Gaussian initialization if not explicitly said He-truncated Laplacian initialization (see Chp. 2), DELU activation function (see Chp. 5) except Softmax output layer (see [15]), L1 and L2 Regularization with Case 2 or Case 3 and Max-Norm Regularization with 0.5 coefficient or no Max-Norm regularization (see Chp. 6), Mini-Batch Learning (see [16]) with 128 samples in each batch (varies in this chapter), and training is completed in 300 epochs with 50 epoch intervals. Training is done with network at Figure 7.4 (Structure 1.1 with same input modification and p values set to 0.375) this time (Max-Pool layers with Pyramid Approach), using 6 data batches (except first 6000, 1000 or zero samples, which are used for validation) of MNIST dataset, and test successes are obtained with test data batch (each data batch contains 10000 samples). From 2 or 10 trials with this configuration, mean, minimum and maximum success rates are obtained each trial independently, to compare of validation success rates of each network in itself, and the one with least validation error (for instance, 150th epoch result for a 300 epoch training) sets its test success as the test success of the whole network.

If no validation samples are present, final network after the 300th epoch is used for testing phase.

A difference in the above attributes needs furthermore explanation. MNIST Data Augmentation slightly differed from the final (fourth) Data Augmentation of the Chapter 8 used for CIFAR-10 networks. So long as it is not logical to horizontally flip digits as a augmentation method, it is removed from the data augmentation techniques used in the experiment. Instead, rotation range is doubled and made 10 degrees to compensate this exclusion, compared to previous data augmentation method. Remaining methods are left as the same. This data augmentation is only used for the final convolutional neural network configuration given in this section. It is also important to remark that data preprocessing is not used for MNIST dataset networks in this section. Now, we'll move the the results taken by the 'default' neural network configuration, named as Default MNIST Network (shortly Default). So long as deviation is so small for the training of this dataset (can be seen from maximum and minimum results in tables), only the final convolutional neural network structure has 10 trials conducted - others have 2 trials for each configuration.

Table 10.1: Results from 2 trials: 'Default' neural network configuration trained with 300 epochs. No validation sample is used.

Network Types	**Default**
Mean Success	98.53%
Minimum Success	98.52%
Maximum Success	98.54%

As seen in the Table 10.1, this network - without any data augmentation and advanced methods - reached around 98.5% mean success rate after 300 epochs. To increase this success rate, we'll employ advanced methods and improve the learning of the neural network. The results of the improved neural network are given below, in Table 10.2. First attempt of MNIST Network used 6000 validation samples and trained in 240 epochs, a lower number. To remind, both of the networks did not used data augmentation and data preprocessing.

Table 10.2: Results from 2 trials: 'Default' neural network configuration is compared with first attempt of improved neural network, named as MNIST Network Attempt 1 (called Attempt 1, shortly).

Network Types	**Default**	**Attempt 1**
Mean Success	98.53%	99.58%
Minimum Success	98.52%	99.55%
Maximum Success	98.54%	99.61%

As seen in the Table 10.2, although less number of epochs are set for this training, it exceeded the previous results by a margin of 1%, which means 71% fall in the test error. It shows the success of the advanced methods used in the convolutional neural network, and superiority of the improved neural network over the default one. To try different things, the size of the validation set is lowered (to 1000 samples) with a goal to increase the learning in shorter time (only 150 epochs) and reach higher success rates (MNIST Network Attempt 2, shortly Attempt 2). Accumulated results are given below, in Table 10.3 for comparison.

Table 10.3: Results from 2 trials: 'Default' neural network configuration is compared with attempts of improved neural networks.

Network Types	**Default**	**Attempt 1**	**Attempt 2**
Mean Success	98.53%	99.58%	99.57%
Minimum Success	98.52%	99.55%	99.55%
Maximum Success	98.54%	99.61%	99.58%

As seen in the Table 10.3, this time mean success rates are slightly lowered due to lowering of validation set size, which makes the neural network go to testing phase relatively immature while it is in an early stage although obtained validation success rates close to 100%. This makes the test results of the less trained (< 100 epochs) neural networks to emerge as the last test success of the neural network structure. To get rid of this problem, we decided to remove the validation set and move with the test success of the neural network after 300th training epoch (MNIST Network Attempt 3, shortly Attempt 3). The mean success rate of Attempt 3 trials is indeed exactly 99.565%, but it is rounded.

Table 10.4: Results from 2 trials: 'Default' neural network configuration is compared with attempts of improved neural networks.

Network Types	**Default**	**Attempt 1**	**Attempt 2**	**Attempt 3**
Mean Success	98.53%	99.58%	99.57%	99.54%
Minimum Success	98.52%	99.55%	99.55%	99.67%
Maximum Success	98.54%	99.61%	99.58%	99.41%

As seen in the Table 10.4, this time although mean success rate fell a little more, highest maximum success rate is obtained by a significant margin (around 0.1%). With the combination of all training samples without validation set and data augmentation techniques designed for MNIST dataset, this success rates will be boosted furthermore. Named as Final MNIST Network Attempt (shortly Final), results obtained for 10 trials (each of them are trained for 300 epochs) are shown in Table 10.5 which is given below. In this table, mean success rate of the new results is indeed exactly 99.695%, but it is rounded for convenience.

Table 10.5: Results from 2 trials (except **Final**, which is 10 trials): 'Default' neural network configuration is compared with attempts of improved neural networks.

Network Types	**Default**	**Attempt 1**	**Attempt 2**	**Attempt 3**	**Final**
Mean Success	98.53%	99.58%	99.57%	99.54%	**99.70%**
Minimum Success	98.52%	99.55%	99.55%	99.67%	**99.64%**
Maximum Success	98.54%	99.61%	99.58%	99.41%	**99.75%**

As seen in the Table 10.5, the final success rates indicate that the neural network reached to a success rate which challenges the best ones in the literature, both in mean and maximum success. The best of neural network classifiers for this dataset, as shown in this work by the year 2016 [38], has an error rate 0.21% [29] which shows the strength of our improved convolutional neural network (0.305%). Apart from this results, there are other important papers which has significant contributions to the literature and falls behind our success rates as seen in [38]. In summary, a simple network with less than 2 million parameters reached to this success rates with training durations around 24 hours.

Table 10.6: Results from 10 trials: Final neural network configuration is used for obtaining the results of different initialization techniques.

Network Types	**He-truncated Gaussian**	**He-truncated Laplacian**
Mean Success	**99.70%**	99.67%
Minimum Success	**99.64%**	99.53%
Maximum Success	**99.75%**	**99.75%**

As seen from the Table 10.6, He-truncated Laplacian has a close mean success rate to He-truncated Gaussian and their maximum success rates for 10 different trials are equal, which shows that He-truncated Laplacian is - slightly - less successful than He-truncated Gaussian initialization, but it is still an alternative initialization method worth of trying.

10.2 Possible Implementations to ImageNet Networks

In this section, we will discuss possible implementations of our advanced methods to the networks which work on ImageNet datasets [37]. There are two different kind of networks used for this purpose, first ones are the networks with pre-trained weights and they can be fine-tuned to learn better, for example VGG16 and VGG19 networks in the Keras' in-built library can be used. In these networks, only a few of the advanced methods are available due to existence of pre-trained weights, which include SinAdaMax optimizer and Expanding Mini-Batches training recommendation. The other type of networks are keeping the same neural network structure, but starting with random weights instead. In this case, we can apply nearly all the advanced methods we proposed in this thesis.

As the future work for this chapter, if possible, these advanced methods can also be employed to increase the success rates of these networks by enhancing their classification performances - although the possibility that this act may make their training durations longer than before (especially the usage of DELU).

Chapter 11

Conclusion and Future Works

As conclusion, this thesis consisted of works conducted on each part of the neural network structure, by taking a convolutional neural network as an example and modifying its parts. While doing that, alternative methods are scanned from the literature and in addition to that, new techniques are proposed by this thesis - many of which are successful - as contributions to the existing literature in 4 of 8 thesis chapters. In result, success rates on different datasets are increased as desired, and met the goals of the thesis work which are set beforehand.

To review the contributions of this thesis, our advanced methods, to the literature briefly, we will start from the Laplacian initializations. As an alternative to the existent initialization methods such as Uniform and Gaussian, this initialization technique is developed and experimented for both untruncated ad He-truncated cases. Untruncated Laplacian initialization performed better than Uniform initialization method and came after Untruncated Gaussian initialization as second best method for CIFAR-10 network. He-truncated Laplacian exceeded success rates of untruncated initializations and Xavier-truncated initializations, came after the He-truncated Gaussian and Uniform initialization as the third best parameter initialization method for mean and second best one for the maximum success rates, even without varying the Truncation Constant from 12 to 13.5 where its success rate peaks. Therefore, it became an alternative technique

for addition to the literature, while being subject to further investigation.

Our next contribution was the SinAdaMax optimizer (learning) algorithm. Inspired from the Adam and AdaMax optimizers, this adaptive learning method made use of absolute sinusoids to increase the constant learning rate during the training epochs, enhancing the learning process. Compared to its predecessors, it increased the success rates for CIFAR-10 networks more and continued its success for MNIST networks too. In result, it became a possible contribution to the literature.

Another addition of our thesis to the literature is the Dominantly Linear Exponential Unit (DELU). Based on Exponential Linear Unit (ELU) which is derived from Rectified Linear Unit (ReLU), DELU takes advantage from its slow transition region in the neighborhood of origin, preventing neurons to decide immaturely and settle after a long time, which allows the neural network to perform much better in comparison to sharp increase regions seen in ReLU and ELU. Although that increases the computational complexity, its boost of the success rates are more promising for an addition to the literature both for CIFAR-10 and MNIST networks as demonstrated.

Our last contribution in this thesis work is the 'Pyramid Approach' for designing Max-Pooling layers in a neural network. Instead of constant size Max-Pooling layers which are frequently used throughout the literature, Max-Pool layers with expanding size (2-3-4, 1-2-3 etc.) are performing better, and far more effective than reverse pyramid-shaped layer formations (4-3-2, 3-2-1 etc.) as shown in both CIFAR-10 and MNIST networks in Chapter 7, 9 and 10 results. Thus, Max-Pool layers formatted with 'Pyramid Approach' understanding can be shown as an improvement which can be added to the existing literature.

To finish this part, we'll note down the success rates obtained with the usage of our methods. For the CIFAR-10 dataset, convolutional neural network given in Figure 7.4 with final configurations given in Chapter 9 (He-truncated Gaussian) reached to 90.82% mean and 91.36% maximum success rate for 10 trials with 300 training epochs, and - less reliably - 91.56% mean and 91.87% maximum success

rate for 2 trials with 600 training epochs. For the MNIST dataset, convolutional neural network given in Figure 7.4 with final configurations given in Chapter 10 (He-truncated Gaussian), which are specific to MNIST dataset, obtained 99.70% mean (rounded from 99.695%), 99.75% maximum (also obtained by 10 trials of He-truncated Laplacian initialization) and 99.64% minimum success rate for 10 trials with 300 training epochs.

The possible future works for this thesis can be trying the non-experimented methods. In addition to ones mentioned at the end of the chapter, those can include Laplacian initialization with Xavier-truncation and using different truncation constants, trying different loss functions which modifies itself during the training, applying batch-based optimizer algorithms such as Quasi-Newton methods, experimenting on DELU activations with saturation limits, working more on proposed custom regularizers in the Chapter 6, fully employing batch normalization or renormalization, replacing all Dropouts with stable implementation of DropConnect or other Dropout-based methods (such as k-lowest Dropout [7]), changing the learning conditions furthermore and coming up with different sets of data preprocessing and augmentation methods. Given the time and available resources, this thesis managed to reach its goals in this uneasy situation, however as seen from those future works it can possible to reach higher success rates after this presentation and thesis work.

Bibliography

[1] B. Çatalbaş, B. Çatalbaş, and Ö. Morgül, "Human activity recognition with different artificial neural network based classifiers," in *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, May 2017.

[2] G. K. Dziugaite, D. M. Roy, and Z. Ghahramani, "Training generative neural networks via maximum mean discrepancy optimization," *arXiv preprint arXiv:1505.03906*, 2015.

[3] B. Çatalbaş, "Recurrent neural network learning with an application to the control of legged locomotion," Master's thesis, Bilkent University, 2015.

[4] C. Clabaugh, D. Myszewski, and J. Pang, "Neural networks - history." https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html.

[5] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, pp. 267–285, Springer, 1982.

[6] T. Chen and H. Chen, "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems," *IEEE Transactions on Neural Networks*, vol. 6, pp. 911–917, July 1995.

[7] M. B. Ali, "Use of dropouts and sparsity for regularization of autoencoders in deep neural networks," Master's thesis, Bilkent University, 2015.

[8] A. J. Papagelis and D. S. Kim, "Back propagation tutorial." https://www.cse.unsw.edu.au/ cs9417ml/MLP2/.

[9] P. Kaur, "Convolutional neural networks (cnn) for cifar-10 dataset." http://parneetk.github.io/blog/cnn-cifar10/, 2017.

[10] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Proceedings of the thirteenth international conference on artificial intelligence and statistics, DIRO, Université de Montréal, Montréal, Québec, Canada*, pp. 249 – 256, 2010.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[13] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the rprop algorithm," in *IEEE International Conference on Neural Networks*, pp. 586–591 vol.1, March 1993.

[14] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[15] R. A. Dunne and N. A. Campbell, "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function," in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, vol. 181, p. 185, 1997.

[16] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Coursera Lecture slides*, p. 14, 2012.

[17] B. Çatalbaş, B. Çatalbaş, and Ö. Morgül, "A new initialization method for artificial neural networks: Laplacian," *2018 26th Signal Processing and Communications Applications Conference (SIU), Izmir*, pp. 1 – 4, 2018.

[18] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, pp. 296–298, Dec 1990.

[19] I. Changhau, "Loss functions in neural networks — isaac changhau." https://bit.ly/2PEQOQQ, 2017.

[20] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[21] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[22] N. M. Nawi, M. R. Ransing, and R. S. Ransing, "An improved learning algorithm based on the broyden-fletcher-goldfarb-shanno (bfgs) method for back propagation neural networks," in *Intelligent Systems Design and Applications, 2006. ISDA'06. Sixth International Conference on*, vol. 1, pp. 152–157, IEEE, 2006.

[23] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[24] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, 2013.

[25] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[26] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in Neural Information Processing Systems*, pp. 971–980, 2017.

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, pp. 1929 – 1958, 2014.

[28] A. Santilli, "Kerasdropconnect." https://github.com/andry9454/KerasDropconnect, 2018.

[29] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, pp. 1058–1066, 2013.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[31] S. Ioffe, "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models," in *Advances in Neural Information Processing Systems*, pp. 1945–1953, 2017.

[32] M. Strickert, *Self-organizing neural networks for sequence processing*. PhD thesis, Verlag nicht ermittelbar, 2004.

[33] A. Cichocki, J. Karhunen, W. Kasprzak, and R. Vigario, "Neural networks for blind separation with unknown number of sources," *Neurocomputing*, vol. 24, no. 1-3, pp. 55–93, 1999.

[34] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., 2009.

[35] F. Chollet, "Building powerful image classification models using very little data." https://bit.ly/2i4ZraH, 2016.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[37] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[38] R. Benenson, "Classification datasets results." https://bit.ly/2PEQOQQ, 2016.