COMP 551 - Project 3 Report

Alexandre ASFAR 260771684
Michel Fadi MAJDALANI 260785279
Mohamed BOUAOUINA 260765511

***Abstract*—The task of this project is to develop models to classify image data from the CIFAR-10 dataset. The first model is an implementation of a multilayer perceptron from scratch. It reaches a test accuracy of 17.34% and consists of a single hidden layer with 8 units using the Leaky ReLU activation function. For our second model, we explored convolution, pooling, and padding operations as well as regularization techniques (L2 weight decay, dropout) to assemble a Convolutional Neural Network architecture that achieves an accuracy of 89.61%. We also present advanced optimization techniques (Batch Normalization, RMSprop gradient descent) for improving the timing performance of deep neural networks.**

***Index Terms*—MLP, CIFAR-10, CNN, ZCA whitening, data augmentation, activation function, batch normalization, dropout**

I. Introduction

DEEP neural networks are nowadays the preferred and superior machine learning approach to image recognition. Indeed, deep learning methods have even produced more accurate results than humans in a competition in 2012 on image classifications tasks [1]. In the following report, we compare the performance of basic multilayer perceptrons versus convolutional neural networks which are more widely used in the field of computer vision. All our experiments are run on the widely known and used CIFAR-10 dataset.

The labeled CIFAR-10 dataset is composed of 60,000 32x32 color images in 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). It was constructed by A. Krizhevsky by subsampling the 80 Million Tiny Images dataset collected by a joint effort from teams at MIT and NYU. For more details on how it was assembled, please refer to the manual written by the founder of the dataset himself [2].

II. Preprocessing

For the preprocessing of the dataset, we focused on two techniques: Zero Component Analysis (ZCA) and data augmentation.

A. ZCA

ZCA is an image whitening technique. The first step of ZCA is to center the data around the origin by first normalizing it and then applying a mean subtraction. In our implementation of ZCA, we used per-pixel mean subtraction, which consists of taking the mean of each pixel across all images and subtract the corresponding mean to the corresponding pixel. The goal of image whitening is to transform the data in a way that the covariance matrix is the identity matrix. In other words, we are rotating the data until the correlation is lost. This is done by first computing the covariance matrix of the data and finding its eigenvectors. We can then apply the matrix of eigenvectors to the data which will apply the rotation we are looking for. The final step is to rescale the data to obtain a correlation matrix equal to the identity matrix. To do that, we divide each dimension of our data by the square-root of its corresponding eigenvalue. For the details of our implementation, please refer to the *ZCA Preprocessing.ipynb* notebook submitted with this report.

B. Data Augmentation

Data augmentation is a technique that allows us to increase and diversify our data for the training model without actually gathering new data. This technique revolves around different simple transformations such as rotating, flipping, cropping or translating (moving the image along the X or Y-axis) an image. Other more advanced methods like conditional GANs or neural style transfer exist. These methods replicate an image in different environments (different times of the day or even different seasons). By increasing the diversity of our data, we can overcome a multitude of biases and thus considerably reduce overfitting.

In our experiments, we use the built-in *ImageDataGenerator* from Keras. For an example, please refer to the *AugmentedRegularizedCNN.ipynb* notebook

III. Multi-Layer Perceptron

A multi-layer perceptron (MLP) is a type of feed forward artificial neural network. MLPs are used to approximate a function f by learning a set of parameters θ [3]. MLPs consist of an input layer, hidden layers and an output layer. Each layer comprises of units, and the output value of a unit in a layer l is equal to the output value of an activation function h that takes for input the weighted sum of units in the l-1^{th} layer [3]. The value of the weights between two layers are learned by the model during training. Training consists of Gradient Descent where gradients are calculated using the back-propagation technique [3].

A. Implementation

In this project, we implemented a "vanilla" neural network from scratch, which is a neural network with one hidden layer.

In a compressed matrix form, the equation of a single hidden layer MLP is given by $\hat{Y} = g(h(XV)W)$, where $\hat{Y}$ is a $(N \times C)$ matrix representing the predicted probability of the n^{th} instance to belong to the c^{th} class, g represents the activation function at the output layer, h represents the activation function

at the hidden layer that is applied element-wise, *X* is a $(N \times D)$ matrix representing the input with *N* instances and *d* features, *V* is a $(D \times M)$ matrix where v_{dm} corresponds to the weights of the d^{th} feature to the m^{th} hidden unit, and *W* is a $(M \times C)$ matrix where w_{mc} corresponds to the weights of the m^{th} hidden unit to the c^{th} output unit.

For the classification task we had to perform on the CIFAR-10 dataset, *g* was chosen to be the softmax activation function. Hence, in the training phase, our model finds the weights that minimizes the softmax-cross-entropy loss function. We trained our MLP by implementing a Mini-batch gradient descent algorithm that calculates gradients using backpropagation.

In addition to the code we were given in the slides, the implementation of our model includes a bias vector in both the input and the hidden layers. The code can be found in *mlp.py*. In Fig 1, we show the train and test accuracy of our best MLP model. It consists of 8 hidden units in the hidden layer using a leaky ReLU activation function and achieves an accuracy of 17.34% on the test set.

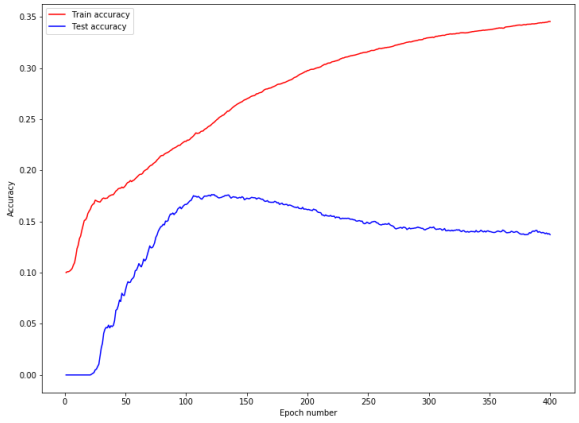

Fig. 1: Training and test accuracy of a MLP model with 8 units using Sigmoid activation function as a function of epoch numbers

B. Activation Functions

In addition to the required experiment, we investigated the use of different activation functions. Activation functions (AFs) are used in neural network to introduce non-linearity [3]. They give neural networks expressive power and provide them the ability to approximate non-linear functions [3]. A good choice of activation function contributes to better learning and generalization as the choice of AF impacts the flow of gradients in the network [4]. A famous related problem is the vanishing and exploding gradients which claims that gradient updates can be very small or very large, and as a result, the training never converges or diverges, respectively [4]. The implementation of all activation functions and their derivatives can be found in *activation_functions.py*.

We decided to create a super class called ActivationFunction that defines the behavior of any ActivationFunction. All AFs should be able to evaluate an output for the forward pass in the network and produce the derivative value for the backward pass. Hence, the two functions *evaluate* and *deriv* are defined in the superclass. This provides a flexible way to add any activation function we desire.

1) Logistic Function: The logistic Function (or sigmoid function) is historically the first activation function. It is given by $\sigma(x) = \frac{1}{1+e^{-x}}$. The logistic function suffers from the vanishing gradient problem [4] [5]. When the inputs become large, in either negative or positive way, the function saturates at 0 or 1, with a derivative extremely close to 0 [4]. Nonetheless, it has an easy derivative to compute which is given by $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

2) Hyperbolic Tangent Function: The hyperbolic tangent function is similar to the logistic function with the exception that it is centered at 0 and ranges from -1 to 1 [5]. It often performs better than the logistic function. Its main advantage is that it produces close to zero values for weighted sums that are approximately equal to 0 [5]. Moreover, it inherits the vanishing gradient problem present in the sigmoid function [4]. Given the similar nature of both the logistic function and the hyperbolic tangent function, we decided to implement the latter as a subclass of the former. It is expressed as $h(x) = 2\sigma(x) - 1$ and as a result has a derivative equal to $h'(x) = 2\sigma'(x)$.

3) ReLU Function: ReLU stands for Rectific Linear Unit and it defined as $h(x) = \max(0, x)$. ReLU has been shown to perform better than sigmoid or hyperbolic tangent because it does not saturate and is fast to compute [5]. Its derivative is equal to the step function

$$h'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Although, the derivative does not exist at 0, it can be set to 1. However, it suffers from the dying ReLU which happens when some neuron "die", meaning they stop outputting anything other than 0 [4].

4) Leaky ReLU Function: Leaky ReLU solves the dying ReLU problem as it introduces a small negative slope to ReLU for weighted sums of inputs less than 0 [4] [5]. It is given by

$$h(x) = \max(0, x) + \gamma \min(0, x)$$

and has the following derivative:

$$h'(x) = \begin{cases} \gamma & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

As a result, our implementation of ReLU is a subclass of Leaky ReLU where γ is set to 0.

In Fig 2, we show the results of training and test accuracy of models using several activation functions. By the universal approximation theorem, any continuous function can be approximated to any desired accuracy by a MLP with a single hidden layer [6]. Hence, as is expected from the theory, Fig 2 shows that the training accuracy increases as a function of number of epochs. After 500 epochs, models that used ReLU or Leaky ReLU reached an accuracy of over 40%. This suggests that the models are learning from data and can confirm the correctness

of our implementation. However, in machine learning, we are interested in the generalization error. From Fig 2, we can see that when the epoch number is greater than 50, the test accuracy decreases as the training accuracy increases. This trend suggests that the model is overfitting to the data in the training data and is not generalizing well. Traditional ways to address overfitting are performing regularization, simplifying the model and/or adding more data [4]. We will talk about regularization techniques for deep neural networks later in the Regularization section. Simplification of the model will be discussed further in the next experiment when analyzing results of Fig 3. Finally, according to Fig 2 and as is expected by theory, ReLU and Leaky ReLU perform better than sigmoid and hyperbolic tangent.

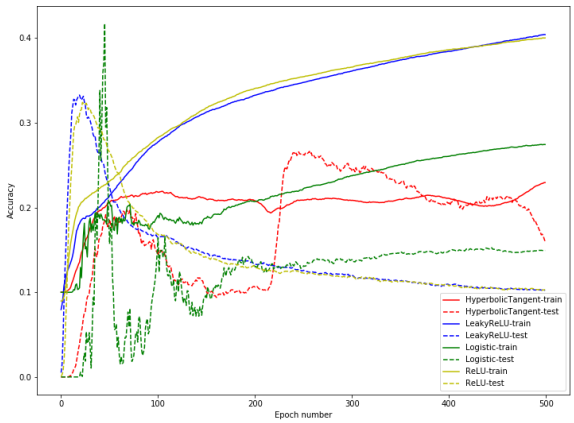

Fig. 2: Training and test accuracy of models using different activation functions with 20 hidden units as a function of epochs number

C. Width of a neural network

In addition to the required experiments, we analyzed the effects of several network widths on both the accuracy and training time. According to [6], the width of a neural network has a smaller impact on its expressive power when compared to its depth. Nonetheless, we were still interested in experimenting the performance of our single layer MLP when varying the number of units in its width.

Fig 3 shows the results of this experiment. Moreover, Fig 3 provides another justification to the fact that simplifying the model addresses overfitting. As can be observed in Fig 3, the simpler models perform better.

Furthermore, we were interested in reporting the effect of the width of a network to the average training time of an epoch shown in Table I. Unsurprisingly, as the number of units increases, the average training time increases.

For more precise values of our observations, refer to the text files of all our results.

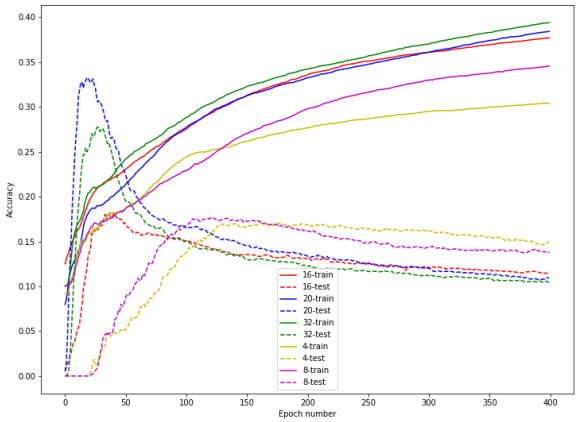

Fig. 3: Training and test accuracy of models with different number of units using Leaky ReLU activation function as a function of number of epochs

IV. CNN Model Introduction

Convolutional Neural Networks were popularized for machine vision problems with the introduction of the LeNet-5 architecture developed by Yann LeCun et al. in 1998 [7]. These networks mainly differ from traditional MLPs thanks to their convolutional layers that make use of the cross-correlation operation with filters (or kernels) on the input images. According to [4], their success in computer vision tasks is due to their structure that leverages the common hierarchical structure in real-world images. Indeed, CNNs tend to focus on smaller features in the early hidden layers before combining them into larger higher-level features in the latter ones.

In this section, we present our first attempt at building a CNN in PyTorch and evaluate its performers. This procedure was the stepping stone to our extended and improved architecture presented in the next section.

A. Baseline model

Our first implementation consisted of a simple architecture using two convolutional layers, a pooling layer, and three fully-connected ones that give a final output of 10 units corresponding to each class of the dataset.

In order to construct this in PyTorch, we extend the Module neural network class and configure our model by stacking the layers provided by the package. Notice that the parameters of each layer need to be carefully chosen such that the number of outputs in one layer match the number of inputs in the next layer.

After defining the model, we need to explicitly declare how a forward pass is performed as well as how the model is optimized. Firstly, we decide to choose the basic ReLU as the activation function for each hidden layer. Secondly, we pick the cross entropy loss function to evaluate the predictions. Implementing back propagation becomes trivial as the

Number of layers	4	8	16	32	64	128	256
Avg. training time (in s)	1.08746	1.03732	1.07667	1.32588	1.76454	2.45229	4.10004

TABLE I: Average training time of a MLP using different number of hidden layers units

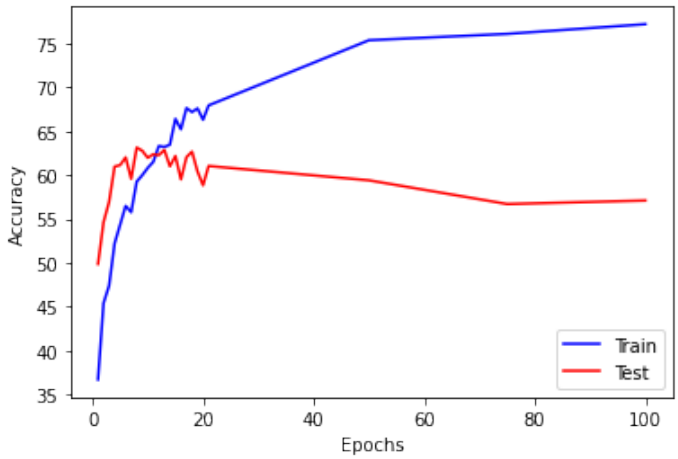

Fig. 4: Plot of training and validation accuracies for the baseline CNN model

CrossEntropyLoss function provided by PyTorch can easily compute the gradients and store them in the *.grad* fields of the *Variables* used for training. Finally, the gradients are used by the provided stochastic gradient descent optimizer to change the values of the weights.

For further details and comments about our implementation in PyTorch, please refer to the *BaselineCNN.ipynb* notebook.

B. Results discussion

In this subsection, we present the results of the evaluation of the baseline model implemented in PyTorch. For the purposes of training, we use Google Colab's built-in GPU to accelerate the processing of data and other computations.

We started first by running our model for a number of epochs and checking the accuracies on the training set as well as a held-out validation set. The results are plotted in Fig. 4. As it can be seen on the plot, the model starts to overfit to the training set after 11 epochs. Therefore, for the final evaluation of the model, we choose to stop training the model after 11 epochs. This yielded a final **accuracy of 62% on the test set**. Despite being somewhat of a low accuracy, this model is performing 6 times better than random (10%) for a 10-class classifications problem.

In order to get deeper insights into how the model is performing, we also explore its accuracy on each class of the problem. The results are presented in Table II. We notice that the model identifies well inanimate objects (truck 75%, car 71%) but fails to pick up the subtleties of animal morphologies (cat 37%, bird 47%). This motivates us to add more convolutions layers and filters to detect more features associated with specific animals.

Class name	Test accuracy
plane	64 %
car	71 %
bird	47 %
cat	37 %
deer	63 %
dog	58 %
frog	64 %
horse	61 %
ship	81 %
truck	75 %

TABLE II: Baseline CNN test accuracy on each class

V. Extending the CNN Model

After exploring the basics of a Convolutional Neural Network, we investigated stacking more layers, varying the units per layers, as well as adding regularization and optimization techniques to construct a final architecture with heightened overall performance. In this section, we present the said techniques and discuss the results of the final model implemented using Keras. Special thanks go to Mr. Abhijeet Kumar for his suggestions on the model architecture and Keras tutorials.

A. Regularization

In the final model, we implement two regularizations techniques: L2 weight decay and dropout. Both of these techniques aim to prevent overfitting by preventing the model from setting high weights to specific parameters.

L2 regularization consists of adding a penalty to the weight of each parameter of the convolutionals kernels. This limits the weight from growing uncontrollably as the penalty adds to the cost function when the weight increases in magnitude.

Dropout randomly deactivates a subset of units during each pass of the training and removes all of their (incoming and outgoing) connections to the remainder of the network. Therefore, the dropped unit does not participate in that training pass and its weights do not get updated.

B. Optimization and batch normalization

As we were working with limited computing resources and wanted to run through hundreds of epochs of data, it was critical for us to implement optimization techniques that improve the timing performance of our model while conserving high prediction accuracies. Therefore, instead of the simple SGD algorithm used in the previous model, we opted for the Root Mean Squared propagation (RMSprop) algorithm for our final model. This version of gradient descent aims to minimize the oscillations of SGD without stopping too early. It keeps a running average of the previous gradients and gives importance to the most recent ones when calculating new gradients.

We also implemented batch normalization after each convolutional layer which re-centers and re-scales the input for the next layer. Sergey Ioffe and Christian Szegedy presented this technique in 2015 and argued that it significantly reduces the problem of changing inputs at each layer in the training phase (internal covariate shift) [8]. More recently, authors have claimed that Batch Normalization rather helps with the smoothing of the cost function instead of addressing the covariate shift issue [9]. In addition, Batch Normalization has demonstrated regularizing effects in experimental work [8] which has also motivated its use in our model.

C. Other considerations

For the final model, we also experimented with other activation functions than the ReLU used for the previous model. Indeed, we chose the Exponential Linear Unit function for the hidden layers as defined here:

$$ELU(x) = \begin{cases} x & if x > 0 \\ \alpha(e^x - 1) & if x \leq 0 \end{cases}$$

The choice of this function was motivated by its known quality of addressing the "dying ReLU" problem by having very small but non-zero output for negative inputs. Additionally, we also substituted the ReLU by the softmax function in the output layer. Indeed, softmax is known to be more decisive when it comes to multi-class classification.

Finally, we also leveraged the "same" padding option in Keras that allows us to perform padded convolution without shrinking the output images' dimensions (same dimensions for input and output). Padding convolution allows to give more attention to the edges of the images (corners and size) and not lose information potentially placed there. The size of the padding p is determined by solving the following equation where n is the image size, f the kernel size, and p the number of padding pixels (assuming square images and kernels):

$$n + 2 \cdot p - f + 1 = n$$
$$\implies p = \frac{f-1}{2}$$

D. Results discussion

We implemented our final model in Keras by stacking five convolutional layers coupled with max pooling layers and assisted with regularization techniques and padding. For the details of the implementation, please refer to the *AugmentedRegularizedCNN.ipynb* notebook submitted with this report.

The model is fed with data augmented with the Keras *ImageDataGenerator* (cf. Preprocessing section) and trained for 300 epochs. We extract training and validation accuracy scores provided by Keras and plot them in Fig 5. Thanks to all our regularization techniques (L2, dropout, Batch Normalization), the model seems to resist overfitting. Indeed, as it can be seen on the plot, the validation accuracy does not drop even after a couple hundreds of epochs. In comparison, it took the previous CNN model only a dozen of epochs for the validation score to start dropping. After plotting the accuracy scores, we chose to train our model once again for 150 epochs before the final evaluation on the test set. This yielded a final **test accuracy of 89.610% with 0.422 loss**. This is our best result for the CIFAR-10 dataset.

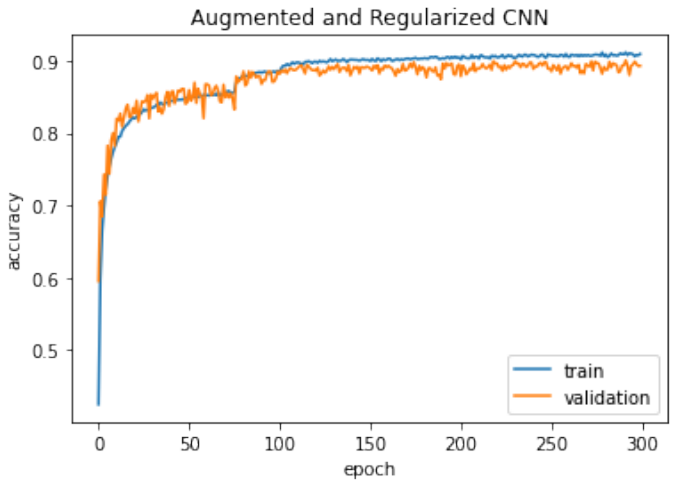

Fig. 5: Plot of training and validation accuracies for the final CNN model

Finally, beyond the high test accuracy reported by this model, we would also like to point out that it takes an average of 30 seconds to train over one epoch which means that the final model was trained in less than 1h15 in Google Colab. Unfortunately, we did not try to train on the Colab GPU which would have decreased significantly the training time especially with the optimizations techniques we implemented.

VI. Conclusion

This project was a valuable experience and familiarized us with the fundamentals of Deep Learning. We were introduced to key concepts in image preprocessing, such as whitening and data augmentation, and to techniques to optimize and regularize a neural network.

The objective of this paper was to compare the performance of our self-implemented single layer MLP to the performance of a CNN on the CIFAR-10 dataset. For the MLP, the Leaky ReLU activation function combined with a width of 8 units yielded the best test accuracy (17.34%). Our baseline CNN obtained a test accuracy of 62% on the test set, and our improved CNN resulted in 89.61%. Hence, we can conclude that CNNs are better image classifiers when compared to simple MLPs.

VII. Statement of Contributions

Alexander:

- Data preprocessing
- ZCA whitening implementation
- Report

Michel:

- MLP implementation
- Additional experiments with MLP
- Report

Mohamed:

- Data augmentation with Keras
- CNN implementation in PyTorch
- CNN implementation and extension in Keras
- Report

VIII. Review of Extra Experiments

In this section, we list the extra paths that we have explored beyond the project's basic requirements.

- Implementation from scratch of ZCA whitening
- Data augmentation in Keras
- Implementation and experimentation of different activation function in the MLP model
- Experimented with the impact of width in the MLP model
- Experimented with different architectures and activations functions for CNN
- Explored two gradient descent algorithms for CNN (SGD and RMSprop)
- Added regularization to the CNN models (L2, Dropout)
- Added an optimization technique for faster training (Batch Normalization)

References

[1] [5]D. Cireşan, U. Meier, J. Masci and J. Schmidhuber, "Multi-column deep neural network for traffic sign classification", *Neural Networks*, vol. 32, pp. 333-338, 2012. Available: 10.1016/j.neunet.2012.02.023.

[2] A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*. 2009.

[3] I. Goodfellow, Y. Bengio and A. Courville, "Deep Feedforward Networks" in *Deep Learning*, MIT Press, 2017, ch. 6, pp.164-223

[4] A. Geron, "Training Deep Neural Networks" in *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, 2nd edition, O'Reilly, 2019, ch. 11, pp.331-374

[5] Nwankpa et al., "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning", 2018. [Online]. Available: https://arxiv.org/pdf/1811.03378.pdf. [Accessed: 13- Apr- 2020]

[6] Z. Lu et al., "The Expressive Power of Neural Networks: A View from the Width", 2017. [Online]. Available: https://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf. [Accessed: 13- Apr- 2020]

[7] 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. Available: 10.1109/5.726791.

[8] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", 2015. Available: https://arxiv.org/abs/1502.03167. [Accessed 15 April 2020].

[9] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, "How Does Batch Normalization Help Optimization?", *NeurIPS 2018*, 2018. Available: https://arxiv.org/abs/1805.11604. [Accessed 15 April 2020].