Image Classification on the CIFAR10 Dataset
Working Paper for CS4487 Machine Learning

BOHNSTEDT Timo , LÖHR Tim
City University of Hongkong, Department of Computer Science

***Abstract*—As part of our final group project, students shall implement a neural network to gain a better understanding of machine learning in practice. To achieve our goal, we try to get the best possible accuracy. To fully understand this dataset and the classification problem, we provided a detailed description of task and its issues. For having a broader understanding of what research contributes to this task of image recognition, we introduce a summary of the latest publications and developments around the image classification field. For our implementation we decided to use a VGG16 model architecture. We go into detail about our model architecture and explain all of its components and corresponding layers. In the end, we analyse the results of our training process and compare it with similar implementation and the state-of-the-art models.**

***Index Terms*—CIFAR10, machine learning, data science, image classification, VGG16, City University of Hongkong**

1. Problem Description

In our course Machine Learning at the City University of Hong Kong, Dr Kede taught us the fundamental mathematical knowledge to solve machine learning tasks. Furthermore, we improved our ability to use this knowledge while working on Jupyter Notebook tutorials, which were provided by Dr Kede and his assistant PhD students. To proof our learning progress in the theoretical and practical field, we are working on a group project. To solve an image classification task, we use the widely used dataset "CIFAR10" [1][2]. The original dataset consists of 60000 coloured images of objects from 10 classes, with 6000 images per category. There are 50, 000 training images and 10, 000 test images. To compare our work with other groups, we are using the following evaluation criteria:

$$Acc(f, D) = \frac{1}{m} \sum_{m}^{i=1} \mathbb{I}\left[y^{(i)} = f(x^{()i)})\right] \quad (1)$$

When working on machine learning tasks with the Python programming language, the three most commonly used frameworks are Keras, PyTorch and Tensorflow [3]. Keras itself is based on tensorflow and hence it is easier to use, but does not provide to full capacity of the entire tensorflow framework[4]. Recently Tensorflow 2.0 got released, so we decided in favour of that and chose Keras, because it impacts the performance of Keras itself too.

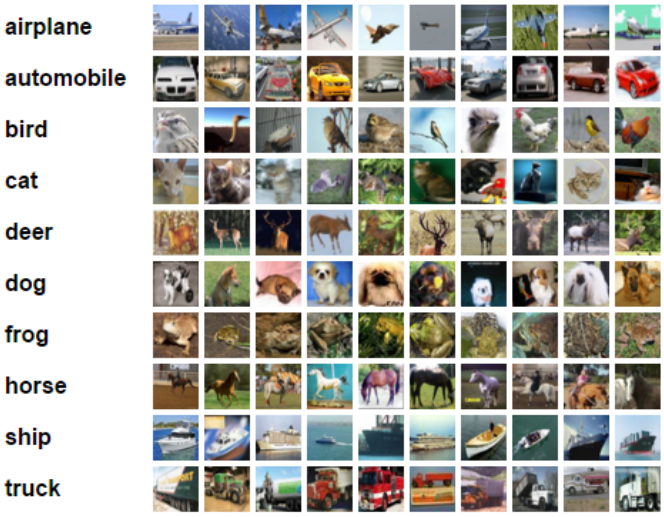

Fig. 1. CIFAR10 dataset

2. Literature Survey

As mentioned in our "Problem Desciption", for an image recognition task the CIFAR10 dataset is quite popular. The archive of the Cornell Univerity provides a huge amount of papers regarding the latest machine learning research. We found 336 publications as we searched for the term "CIFAR10" . There are two independent and well known websites [1] [2] which rank annually the most recent

[1] https://paperswithcode.com/sota/image-classification-on-cifar-10
[2] https://benchmarks.ai/cifar-10

RANK	METHOD	PERCENTAGE CORRECT	PERCENTAGE ERROR	EXTRA TRAINING DATA
1	GPIPE + transfer learning	99	1	✓
2	EfficientNet	98.9		✓
3	PyramidNet+ShakeDrop (Fast AA)	98.3		×
4	EnAET	98.01	1.99	×
5	Proxyless-G + c/o	97.92	2.08	×

Fig. 2. CIFAR10 Leaderboard Top 5, 2019.12.13

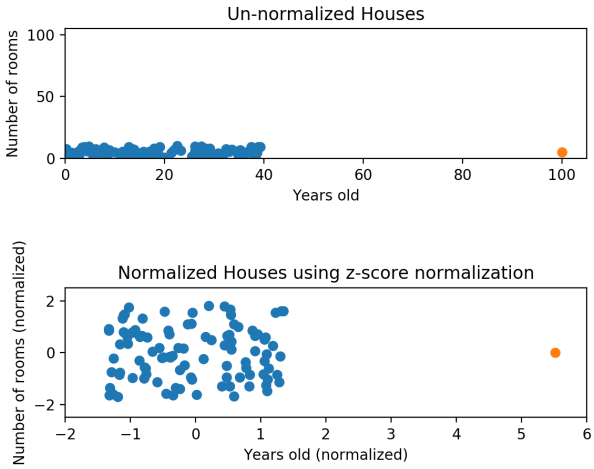

Fig. 3. Z-Scoring

performances for the image classifcation task on CIFAR10. The rankings are mostly referring to the overall test accuracy or the total loss [1]. For simplification, we are relating to the classification of the web page "paperswithcode.com" [3]. In the following we shortly introduce some of the most successful architectures which are at the top of the leaderboard [5] [6].

2.1 GPipe

In smaller experiments, the training process for a model is relatively fast, whereas the training process for a model with an upscaled complexity consumes dramatically more time. [7]. Researchers at Google implemented a machine learning library which offers parallelization for the training process of a machine learning model. The GPipe uses mini-batch gradient descent and pipeline parallelism. The goal is that k accelerators can calculate a defined part of the model k. If we can split our model into k parts, we can to scale the training process even higher [8]. This leads to an notable performance boost in machine learning tasks, such as image recognition[9]. The researchers could verify their work by using real-word datasets like the CIFAR10. They reached an accuracy of 99% [8] with this dataset.

2.2 EfficientNet

While we could use the GPipe for running our complicated and colossal model parallel, EfficientNet trys to get the same result in a more efficient way[10] . So, instead of using more resources, the team introduced the idea of a super-efficient method to reach even better or similar results for using fewer resources with already existing neural network architectures[10]. For example, they increased the accuracy of the ResNet up to 10% with almost not using any extra resources.

[3]http://arxiv.org/

2.3 PyramidNet

In many modern CNN architectures, the feature map dimensions are not increased until they are confronted with a downsampling layer. The key idea of the PyramidNet was introduced in the working paper *Deep Pyramidal Residual Networks* [11]. This architecture concentrats on the feature map dimensions, in particular it increases the dimensions gradually instead of increasing them sharply at each residual unit with a downsampling task. Additionally, this network architecture is a mixture of both plain and residual networks [11]. The team also introduced a *novel residual unit* which can improve the performance of an ResNet architecture even more dramatically. In conclusion, is the PyramidNet outperforming the normal ResNet and the DenseNet. [11].

3. Technical Details

3.1 Preprocessing

3.1.1 Data Augmentation: Data augmentation describes the method of creating new data points by transforming the original data. If we look at images, this can be proceeded for example by resizing, rotating, shifting, flipping and more. With this process we can artificially increase the amount of data, even though we have limited access to only 50000 training data. This leads to an improved performance for our VGG16 model. We use the *ImageDataGenerator class* built-in function from Keras itself to perform this transformation.

3.1.2 Normalization: The goal of normalization is to scale every datapoint in such a way that outliners cannot harm the rest of the data points. There are many possible ways to scale data e.g. MinMax-Scaling or Z-Score and all of them follow specific purposes. The MinMax-Scaling suffers from the so-called *outlier issue*. We decided to choose the Z-Scoring, because the Z-score normalization is a strategy of normalizing data that avoids this outlier issue [12].

The Z-scoring $\frac{value-\mu}{\sigma}$ transforms example data as shown in

the figure 3 [4].

3.1.3 Categorical Output: As the name of CIFAR10 already implies, this dataset contains ten different classes. The ten classes are ten different animal names. To predict one of those animals with the trained model, the model outputs ten different values. Each represents the probability of being one of the animals, respectively. We convert the arranged numbers from one to ten 50000, 1) into a binary output class matrix (50000, 10). Each row in this matrix values one for the desired animals. E.g. animal number three will be transformed into [0 0 1 0 0 0 0 0 0 0]. If we didn't convert our labels into this format, we could not resolve back the probability for each animal and would lose information to make further improvments.

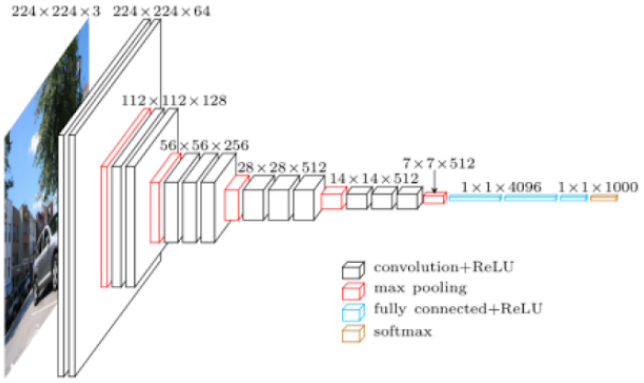

Fig. 4. VGG16 Model Architecture from https://arxiv.org/pdf/1409.1556.pdf

3.2 Model

To solve the image classification task, we are using a convolutional neuronal network (CNN). CNN's are a particular case of feed-forward neural networks [13]. On a very fundamental level, we can say that a feed-forward neural network - as the most machine learning models - is a function [14]. The function of a feed-forward neural network can simply be described as $y = f(x, \emptyset)$. CNN's and feed-forward neural networks are estimating parameter values. So for $y = f(x, \emptyset)$ we estimate the parameter $\emptyset$ [13]. Through those estimated parameter values we are receiving a function back with the smallest possible difference between the predicted output values and the defined output values. A function that measures the difference between the expected - and the defined output is a so-called loss function [13]. The problem description gives already a mathematical definition (8) from the accuracy loss function, which is used by us. CNN's and classical feed-forward neuronal networks differ in their basis of calculation. Feed-forward neural networks are using matrix multiplication, whereas CNN's are using convolutions. Convolutional layers, pooling layers, and fully-connected layers are specific layers for a CNN [15].

3.2.1 Model Architecture: We decided to use the VGG16 model architecture [16] (Figure 4).

In 2014, an implementation of the architecture won the ILSVR(Imagenet) competition. As we could already see in figure 2 that there are quite impressive results, we still decided to use this classic example. The complexity is rather small in comparison to other designs which are provided in the ranking. Note that - as mentioned in the "Problem Desciption", the goal of our project was not to get an outstanding result of the accuracy. The main goal was to understand the principals behind deep learning and to get a feeling of what the state of the art (in image recognition) is. So, because the VGG16 [16] is known as the best visualizable model architecture, we thought it would be the best choice to accomplish our goal in this way. The model architecture is also known for getting rid of a vast amount of hyper-parameters. Instead, it is focusing on tiny filter size and a small strate. The technical details

[4]https://www.codecademy.com/articles/normalization

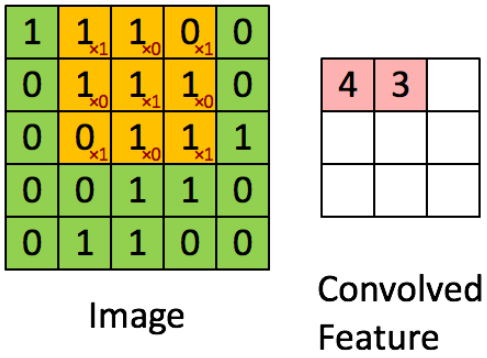

Fig. 5. Feature Map derived from http://deeplearning.stanford.edu

"Problem Desciption", of the different layers are providing more details about its implementation.

3.2.2 Convolutional Layers: The convolutional Layer performs the extracting of features from the input matrix into a feature map. For this procedure, we use matrix multiplication in the form of the dot product and a filter (feature detector, kernel) [13].

In picture 5 can be seen a calculation example of how convolution is applied to a matrix. We are iterating with the filter over the matrix, calculating the scalar product and writing the result into a feature map. An activation function like the rectified linear unit (ReLU) normalizes the feature map after we performed the convolution. Normalization guarantees a feature map which is not a linear transformation of its input value. If this is the case, we only have a linear problem which can be easily solved, but with a wrong results. In other words, the input values get just multiplied by coefficients. Note that there are plenty of reasons for activation functions in a neuronal network. We are giving a closer look at this topic at our section "Activation Functions". In Figure 6 you can see how our pictures had changed after applying the first convolutional layer, before and after doing normalization.

This represents the 64 first feature maps of a random picture

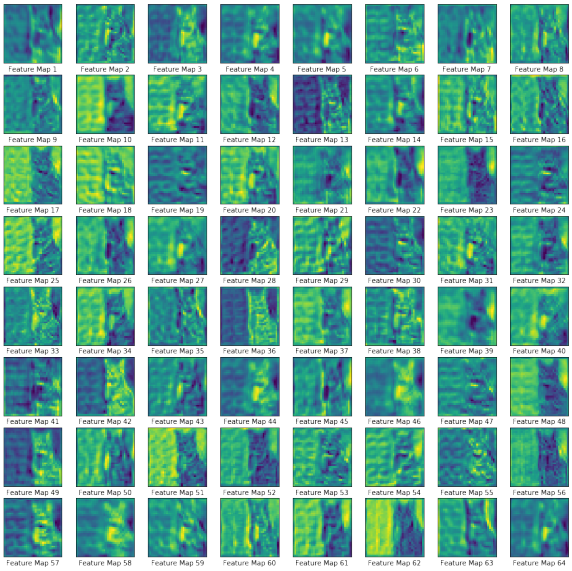

Fig. 6. Feature Map from CNN Layer 1

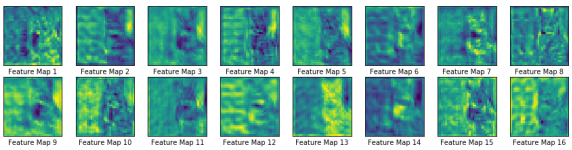

Fig. 7. Feature Map from CNN Layer 2

from the dataset. We can observe that the filters already focus on certain points on the cat's body. We further used this feature map as input for the second CNN layer. The output of the second feature map (figure 7) further sharpens the filters. For example we can see that feature map 2 until 5 focuses on the cat itself, whereas map 8 puts its focus on the cats ears.

3.2.3 Pooling Layers: It is common to add a pooling layer in-between the convolutional layers periodically. The function exists to avoid overfitting and step by step reduce the size of the input image by reducing the number of parameters and computations of the network [17]. We use three pooling layers with filters of size 3x3 applied with a stride of 0 and a padding set to *same*. This downsamples every depth slice in the input by 2 along both width and height, discarding 50% of the activations.

3.2.4 Fully-Connected Layers: The Dense layer is the last layer of our model. It is also called the *fully connected layer*. Neurons in the fully connected layer have full connections to all activations in the previous layer, as in the normal neural networks. Their activation functions will then be computed by a matrix multiplication which is followed by a bias offset. For the ten different animals we respectively have ten fully connected neurons in the dense layer.

3.2.5 Kernel Regularization: There are many different regularizations to prevent the neural network from overfitting. For our project we chose the *L2 regularization* because it is the most common form. It can be integrated by penalizing the squared magnitude of all parameters directly in the objective. So, for every weight w in the network, the term $\frac{1}{2}\lambda w^2$ gets added to the objective, where λ is the regularization strength. It's usual to see $\frac{1}{2}$ in the front, because the gradient of this term with respect to w is simply λw instead of $2\lambda w$. The L2 penalizes the peaky weight vectors and prefers the diffuse weight vectors [18]. During the gradient descent weight update, the L2 regularization has the meaning, that every weight is decayed linearly (w += -*lambda* * w) towards zero.

3.2.6 Batch Normalization: A technique developed by Ioffe and Szegedy [18] is called Batch Normalization properly initializes neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. Normalization is a simple differentiable operation. It allows to use higher learning rates at the beginning and is less vulnerable to a bad initialization. In other words, neural networks that implement batch normalization layers are significantly more robust. Additionally, batch normalization can be interpreted as a preprocessing step on every layer of the network. Batch normalization yields in general, a substantial improvement in the training process.

3.2.7 Activation Functions: There a couple of widely used activation functions like tanh, sigmoid function, ReLU or the ELU. For our model we decided to use the *ReLU* activation function. The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function $f(x) = max(0, x)$. In other words, the activation is simply thresholded at zero. There are plenty of pro's and con's for the ReLU:

- (+) Compared to tanh/sigmoid neurons that need to compute expensive operations like the exponentials, the ReLU can be implemented by directly thresholding a matrix of activations at zero.
- (+) It was found to notably accelerate the convergence of stochastic gradient descent (SGD) compared to the tanh/sigmoid functions. That is, because of its linear, non-saturating form.
- (-) Sadly, ReLU units can be weak during training and possibly "die". For example, a large gradient streaming through a ReLU neuron could cause the weights to update in a dead end, so that it will never activate again. If this happens, then the gradient streaming through the unit will forever be zero. The ReLU units can irreversibly die during training since they can get eliminated off the data manifold. With a good scheduling setting of the learning rate this is less likely to happen.

3.2.8 Loss function: Our compiled Keras model uses the *cross-entropy-loss* [19].

$$L_i = f_{y_i} + log \sum_j e^{f_j} \tag{2}$$

where we are using the notation f_j to mean the *j-th* element of the vector of class scores *f*. The full loss for the dataset

is the mean of f_i over all training examples together with a regularization term *R(W)*. The cross-entropy loss, or also called log loss, measures the performance of our classification model with the output as probability values between zero and one. The cross-entropy loss increases as the predicted probability diverges from real value labels.

3.2.9 Optimizer: There are many possible optimizer suitable for our task. The common ones are the RMSprob, Adam or the stochastic gradient descent (SGD) [20]. We decided to use the SGD to minimize the loss by computing the gradient with respect to a randomly selected batch from the training set. This method is more efficient than computing the gradient with respect to the whole training set before each update is performed.

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1-p_i) & if \quad i = j \\ -p_j p_i & if \quad i \neq j \end{cases} \tag{3}$$

$$\frac{\partial L}{\partial o_i} = p_i - y\_oh_i \tag{4}$$

For the derivation of the cross entropy loss, *y_oh* is the one-hot encoded representation of the class labels.

3.2.10 Softmax: The softmax normalizes the class probabilities to one and it has a probabilistic interpretation.

$$f_j(z) = \frac{e^{z_i}}{\sum_k e^{z_k}} \tag{5}$$

The exponential values can very quickly explode to an infinite large number, for example e_{1000}. To fix this issue, it takes a one-dimensional vector of arbitrary length (in z) and puts it into a vector of values between zero and one that sum together to one. The cross-entropy loss that includes the softmax function, hence to minimize the cross-entropy-loss between the estimated class probabilites.

3.2.11 Flatten: In between the convolutional layer (CNN) and the fully connected layer (Dense), there is a *Flatten layer*. The Flattening layer transforms a two-dimensional matrix of features into a one-dimensional vector that can be respectively streamed into the fully connected neural network classifier, which are our ten fully connected animal neurons.

4. Result Analysis

4.1 Settings

As mentioned in the model architecture, our implementation of the VGG16 network is still state of the art, but no longer among the very best of its field. Even more, we want to point out its difficulties and significant problems. As we look at our training process, we could see two things. First, the estimated time of our epochs is around 30 Minutes if we train it on our MacBook without Nvidia graphics card. So, if we want to train 100 epochs to gain the best possible result without overfitting our model, we would need 50 hours. This is obviously too long.
One possible way is to translate the *.ipynb* format into *.py* format, because we can then run the model as a background job on a tiny server. The tiny server has a similar training time, but can be trained in the background. The best solution for us was to use a server from the Amazon Web Services (AWS). We rented the *Deep Learning AMI (Amazon Linux 2) Version 26.0* [5]. On this server we can run the Jupyter Notebook in the background and use the portforward function from the SSH connection to show the user interface (UI) of the jupyter notebook on our Macbooks. In this way we can train the model as learned from the CS4487 lecture.
The AMI Server from Amazon gets shipped with multiple python environments which can be activated manually. We chose the *p36 Tensorflow Kernel* to activate the neccessary packages. Under this setting, our model needs approximately only 3 4 hours to be trained.

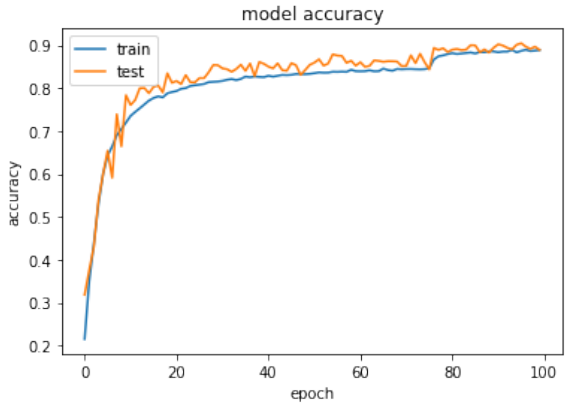

Fig. 8. Accuracy Plot

4.2 Model Analysis

Besides the difficulties we had, if we would consider improving our accuracy with the VGGNet, we could say that our model does a good job due to overfitting as you can see in Figure 8. This achievement is mostly because of our attempted to avoid overfitting while using a weight decay, kernel regularizer and batch normalization. The figure 8 furthermore shows, that no matter how many more epochs we train, the accuracy will not further increase. Other VGG networks reach similar results, no matter if it is VGG13, VGG16 or VGG19 [21]. All of the VGG networks have their limit at around 92 94% [22].

References

[1] P. Echersly, 2017-12-11. [Online]. Available: https://www.eff.org/ai/metrics
[2] 2019 2017. [Online]. Available: https://www.kaggle.com/benhamner/popular-datasets-over-time/code
[3] 03 2019. [Online]. Available: https://codete.com/blog/machine-learning-libraries-overview-top-10-libraries-and-frameworks-you-should-kno
[4] *Deep learning with applications using Python : chatbots and face, object, and speech recognition with tensorflow and keras*. New York, NY: Springer Science+Business Media, 2018.
[5] Benchmarks.Ai, "Classify 32x32 colour images into 10 categories," 2019. [Online]. Available: https://benchmarks.ai/cifar-10

[5] https://eu-west-1.console.aws.amazon.com/

[6] P. w. Code, "Image classification on cifar-10," 2019. [Online]. Available: https://paperswithcode.com/sota/image-classification-on-cifar-10
[7] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, "Complexity vs. performance: empirical analysis of machine learning as a service," 2017.
[8] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, 2018. [Online]. Available: http://arxiv.org/abs/1811.06965
[9] 03 2019. [Online]. Available: https://www.heise.de/developer/meldung/Google-veroeffentlicht-Pipeline-Parallelisierung-nutzende-ML-Bibliothek-4326656.html
[10] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019. [Online]. Available: http://arxiv.org/abs/1905.11946
[11] S. Lim, I. Kim, T. Kim, C. Kim, and S. Kim, "Fast autoaugment," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
[12] T. a. Dr.A.Santhakumaran, "Statistical normalization and back propagation for classification," 2011. [Online]. Available: https://pdfs.semanticscholar.org/4cbd/5fcd6081cfcd16e9111f1bcc17b6e283d439.pdf
[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.
[14] V. D. Visin and Francesco, "A guide to convolution arithmetic for deep learning," 2018. [Online]. Available: https://arxiv.org/pdf/1603.07285.pdf
[15] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, *Efficient BackProp*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_2
[16] K. S. . A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/pdf/1409.1556.pdf
[17] H. W. Gu and Xiaodong, "Max-pooling dropout for regularization of convolutional neural networks." [Online]. Available: https://arxiv.org/pdf/1512.01400.pdf
[18] S. I. Szegedy and Christian, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015. [Online]. Available: https://arxiv.org/pdf/1502.03167.pdf
[19] Z. Z. Sabuncu and M. R., "Generalized cross entropy loss for training deep neural networks with noisy labels." [Online]. Available: https://papers.nips.cc/paper/8094-generalized-cross-entropy-loss-for-training-deep-neural-networks-with-noisy-labels.pdf
[20] M. R. Gorti and S. Krishna, "Faster gradient descent via an adaptive learning rate," 2017. [Online]. Available: http://www.cs.toronto.edu/~mravox/p4.pdf
[21] B. Ayinde, T. Inanc, and J. Zurada, "On correlation of features extracted by deep neural networks," 01 2019. [Online]. Available: https://www.researchgate.net/publication/330751574_On_Correlation_of_Features_Extracted_by_Deep_Neural_Networks
[22] L. S. V. S. Benjamin Recht, Rebecca Roelofs, "Do cifar-10 classifiers generalize to cifar-10?" 04 2018. [Online]. Available: https://arxiv.org/pdf/1806.00451.pdf

LIST OF FIGURES