Exploring Convolutional Neural Networks for Image Classification Problem

Liwen Dai
20552153, L7DAI@uwaterloo.ca

***Abstract*—In this project, I have built and trained neural networks to solve the image classification problem on the CIFAR-10 dataset. First I reimplemented a model provided by Tensorflow with two convolutional layers as a baseline. Next I reshaped the network by stacking multiple convolutional layers. I also attempted to make the network deeper by adding network-in-network layers and using fractional max pooling. My best result beat the baseline with 86% accuracy on the test set as well as a much faster speed of convergence.**

***Index Terms*—image classification, convolution, neural networks, deep learning, artificial intelligence**

I. Introduction

CIFAR-10 and CIFAR-100 are two famous datasets to benchmark image classification algorithms [1]. The CIFAR-10 dataset consists of 60000 32x32 images in 10 classes, with 5000 training images and 1000 test images each class. The CIFAR-100 datset is similar to CIFAR-10, but is has 100 classes containing 600 images each. An incomplete leaderboard on these two datasets has been collected by [2].

Neural networks, particularly, different variations of convolutional networks, have been proved the most effective methods for the task of image classfication. As in [2], they have occupied all the top results. But working real-world images of medium or high resolution requires very large neural networks, containing more than millions of parameters, which can be very slow to train [3]. For this reason many researchers choose to evaluate their methods on these tiny image data sets before scaling.

However, even with the complexity controlled by limiting the task on tiny images, the state-of-art results still rely on networks of considerable size [4]. I found it nearly impossible to replicate these models on my laptop, simply because the GPU would run out of memory or it would take too many days, even weeks to train. Considering this fact, the question has arised: what can we achieve if we perform deep learning tasks with limited computing resource? A very intuitive idea is to shrink a large neural network simply by reducing the number of parameters. Another idea is to limit the training to some acceptable time.

Based on the question above, I reimplemented a convolutional network provided by Tensorflow [5] as a baseline; then I built deeper networks using multiple techniques, including stacking convolutional layers, adding network-in-network layers and fractional max pooling.

After applying different combinations of techniques and parameter settings, my best model beat the baseline and achieved 86% test accuracy on the CIFAR-10 data set. Besides that, my model also had a much faster training speed to reach convergence.

I concluded that although networks with shrinked size cannot reach the state-of-art results on CIFAR-10 [2], there's still room for improvement. I also discovered that techniques such as dropout may not be as effective in small networks as they are in the larger counterparts. I expect these results and discoveries to be helpful when tackling more complicated problems with limited computing power.

This report is organized as following: section II has summarized related state-of-art works on the image classification task; section III introduced different techniques I applied to my network models; section IV stated the details of my implementations; section V listed my training and testing results; and in section VI I summarized my conclusions, thoughts, and future works.

II. Related Work

Convolutional neural network has been a very big jump for image classification and object recognition tasks, since introduced as Alexnet [3]. Alexnet won the ImageNet LSVRC-2010 completition with top-1 error rate of 37.5% and top-5 error rate of 17.0%, while the second best model had only reached 45.7% and 25.7% [3]. This neural network had 60 million parameters and 650,000 neurons, consisting of five convolutional layers, with max-pooling layers, dropout, followed by three fully-connected layers and a final softmax output layer [3].

In the recent few years, a lot of researchers have applied various techniques to improve the performance of convolutional networks, and many have benchmarked their models on the tiny image data sets, CIFAR-10 and CIFAR-100. The "Network in Network" structure was introduce to enhance the model discriminablility for local receptive fields, which achieved 91.2% and 64.3% accuracy on CIFAR-10 and CIFAR-100, respectively [6]. Spatially-sparse convolutional neural networks improved this accuracy to 93.72% and 75.7% [7]. Applying fractional max pooling pushed the result on CIFAR-10 further to 96.53% [8]. The deep residual network introduced by Mirosoft reached 93.57% on CIFAR-10 [9]; and its "wide" variance even reached 96.62% and 81.7%, with a much faster training speed [10].

It is worth mentioning that nowdays convolutional networks have grown deeper and deeper. Alexnet had 5 convolutional layers and was titled "Deep convolutional Neural Networks"

when published in 2012 [9]. The spatially sparse net with fractional max pooling benchmarked on CIFAR-100 with 14 convolutional layers [8]. The deep residual net was even implemented with 1000 layers on CIFAR-10 [9]. Luckily, going deeper is not always necessary: the wide but "shallow" residual nets reached the top accuracy with 28 convolutional layers [10]

III. Methodology

Convolutional Layers

A convolutional layer consists of a 4-D weight matrix, with size $s \times s \times M \times N$, where each square filter (or patch, kernel) has size $s \times s$. M is the number of input filters and N is the number of output filters, which are also called the number of input (output) channels. In our case, the first convolutional layer of a network always accepts the image as input, which has 3 channels (RGB). When stacking multiple convolutional layers, we can notice that the number of output filters of a convolutional layer is always equal to the number of input filters of the next convolutional layer. As a result, we can omit M, and denote a convolutional layer as conv$s \times s \times N$. Since I was working on tiny images, I picked small filter size, including $s = 1, 2, 3, 5$.

Next how to choose N, the number of fitlers at each layer? I used the idea of linearly increasing number of filters with respect to the layer depth, which was inspired by the architecture of spatially sparse neural networks [7]. Particularly, let $N = kl$, where k is some width factor, thus the l-th convolutional layer in a network would consist of kl filters.

Then how do we calucate the number of parameters in a network? Consider a network consisting of L convolutional layers, so we have $l = 1, 2, 3, ..., L$. We assume all convolutional layers use patch size $s \times s$. The total number of weights of convolutional layers would be:

$$s^2 \times 3 \times k + s^2 \times k \times 2k + s^2 \times 2k \times 3k + ... + s^2(L-1)k(Lk)$$
$$= O(k^2 L^3)$$

This shows the number of parameters, or the complexity, of a convolutional neural network, which is quadratic w.r.t the width factor k, and cubic w.r.t the depth factor L, providing a similar shape. We can notice that most parameters lie on the last few convolutional layers, while extra components, like fully connected layers, ususally do not increase this complexity. We use a similar notations as in [7] to denote such convolutional net barebones as $CN(k, L)$.

Loss Function

I picked softmax cross entropy between the network output and the true labels as the loss (cost) function [11]. It measures the probability error in discrete classification tasks where classes are mutually exclusive. I also used weight decay at each convolutional layer, by adding the L2 loss (sum of squares for the weights) of each layer to the total loss, in order to penalize weights that are too large [12].

Network-in-Network

Network-in-Network (NiN) is a special structure added after the convolutional layers, hence increasing the depth of the network yet without adding too much complexity [6]. An NiN layer is basically conv1x1, while the number of filters remains unchanged. The goal of adding these micro neural networks to more complex structures is to handle the variance of the local receptive fields, thus making the netwok more expressive. Researchers have found that adding NiN to conventional convolutional networks can significantly improve the accuracy on benchmarks [6]–[8]

Fractional Max Pooling

Fractional max pooling (FMP) is another technique to stack more convolutional layers [8]. Convolutional networks almost always include some spatial pooling, and often it is a 2x2 max-pooling [3]. Such max-pooling acts on the hidden convolutional layers; and each pooling has an inevitable side effect of discarding 75% of the data. 2x2 max-pooling also limits the depth of a network. Since CIFAR-10 has images of size only 32x32, the 2x2 max-pooling can be performed at most 5 times before the size reaches 1x1.

Methods to get around this problem include pooling from overlapping regions and stochastic pooling [8]. However both these techniques still reduce the size of the hidden layers by a factor of two, while intuitively we can discard data in a gentler way. FMP combined the idea of both these techniques: it pools stochastically, and preferably pools from overlapping regions. As a result, each pooling only reduces the size of image by a factor of α with $1 < \alpha < 2$; thus the average pooling region is only $\alpha \times \alpha$ [8]. Particularly, setting $\alpha = 1.4$ allows us to pool 8 times from a 32x32 image; and setting $\alpha = 1.2$ allows us to pool 12 times (due to rounding down).

Data Augmentation

Data augmentation has been used widely in neural networks to reduce overfitting [3]. However to reach the best results, heavy data augmentation may not be necessary [10]. I used data augmentation methods in the following order: padding the image to size 36x36; random cropping by size 3; random horizontal flip; and finally per-image standarization, so that each image is linearly scaled to have zero mean and unit norm.

Dropout

As in [3], dropout consists of setting to zero the output of each hidden neuron with probablity 0.5, which is a very effective way to reduce test errors. Some researchers believed that deeper layers tend to be more sensitive to dropout, while shallower layers tend to be relatively robust [7], [8]. A new idea came out: using linearly increasing dropout ratio w.r.t layer depth, from 0 at the first convolutional layer, to 0.5 at the last convolutional layer [7], [8]. I applied linearly increasing dropout and compared with the zero-dropout counterpart.

Leaky ReLUs

Rectified Linear Units(ReLUs) have been used widely in convolutional neural networks, since it can make the training several times faster than the equivalents with $tanh$ units [3]. However, ReLUs have a potential disadvantage during training, because the gradient is 0 whenever the unit is not active. This could cause the problem that a neuron never activates as the gradient descent algorithm will not ajust its weight if the neuron is not activated from the beginning. The vanishing gradients problem could also make the learning slow for ReLU networks with constant 0 gradients [13].

To solve the problem of "dead" ReLUs, leaky ReLUs was introduced [13]. The idea is to use the following activation function in instead of conventional ReLUs:

$$f(x) = \begin{cases} x, & \text{for } x >= 0 \\ \alpha x, & \text{for } x < 0 \end{cases}$$

As we can see, conventional ReLU is a special case of leaky ReLU by setting the constant factor $\alpha = 0$. Adding a small positive gradients for negative input ($\alpha = 0.01$) can address the issue of "dead" ReLUs by giving them a chance to recover [13]. It was even found that setting $\alpha = 1/3$ could speed up learning without harming the representation power of the network [7].

IV. Implementation

My source code can be found at Github https://github.com/lwdai/NN-for-image-recognition

Network Architecture

I implemented and trained four convolutional neural network models on the CIFAR-10 datset. The four models are: CNBase, CNDeep, CNNiN, CNFmp. Figure 1 shows the structure of CNBase and CNDeep. CNNiN and CNFmp were variations of CNDeep, as explained below.

CNBase: The CNBase model was my reimplementation of the tensorflow example [5], since the original code was too messy to be my starting point. It consisted of 2 convolutional layers, followed by 2 fully connected layers, and a final output layer. 2x2 max-pooling was applied after each convolutional layer. Conventional ReLUs were used without any leakiness. This model contained 1.76 million parameters.

CNDeep: Inspired by the design of spatially sparse convolutional networks [7], I implemented the CNDeep model by stacking 6 convolutional layers, which was essentially $CN(100, 6)$. In [7], [8] $k = 300$ was used as the width factor, but I shrinked it to 100 due to limited computing resource. I used only one fully connected layer instead of two as in CNBase, because I would expect enough representation power due to increased depth, as in [9], [10]. This model contained 2.96 million parameters. No pooling was applied after the final convolutional layer, because the image size would be 1x1 already. I applied linearly increasing dropout ratios to each convolutional layer, except for the first one, with the max dropout ratios of $0.0, 0.2$ and 0.5. For leaky ReLUs $\alpha = 0.01$ was used in the final version.

CNNiN: CNNiN was the "network-in-network" [6] version of $CNDeep$. The major change was that after each convolutional layer, an NiN layer, conv1x1 was applied with the same number of filters. Another change was that an extra fully connected layer with 500 neurons was added before FC250, due to the possible increase in its representative power. I used linearly increasing drop ratio with a max rate of 0.3. Leaky ReLUs with $\alpha = 0.3$ was used. Adding NiN made this model contain 12 convolutional layers, and 4.14 million parameters. We can see the number of parameters was controlled, w.r.t. the difference in depth from CNNDeep.

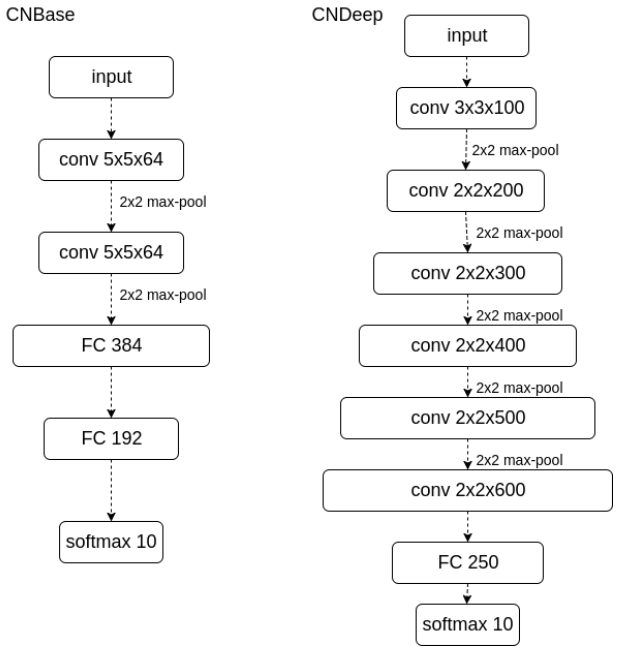

Fig. 1. The structure of two models: CNBase and CNDeep. "FC 384" represents a fully connected layer with 384 neurons etc.

CNFmp: Applying fractional max pooling [8] to CNDeep allowed stacking 13 convolutional layers, as in figure 2. Adding a final NiN layer made a total of 14 convolutional layers with a shape similar to $CN(32, 13)$. I shrinked the width factor k to 32 so that the total number of parameters was controlled to 3.35 million. I used leaky ReLU with $\alpha = 0.3$. Dropout rates increased linearly from 0.0 to 0.2.

Expriment Setup

I implemented all the convolutional network models using Python 2.7 and Tensorflow. All the training and evaluation were running on my laptop with GPU 3GB GTX 770M.

I picked the GradientDescent Optimizer. I also tried other optimizers including Adam and Momentum, but coundn't get them to work, possibly due to proper parameter settings.

I set the learning rate to start with 0.1, which decayed exponentially after the first 10,000 steps. Approximately every 30,000 steps the learning rate was multiplied by a factor of 0.1.

I adjusted the batch size to be 32 for training and 5 for evaluation. The max training step was set to 100,000. Higher

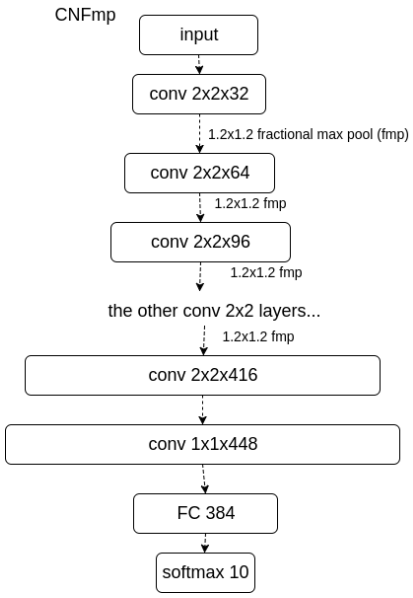

Fig. 2. The structure of CNFmp. It stacked 14 convolutional layers and 12 1.2x1.2 fractional max pooling operations.

batch size had led to crash due to running out of video memory.

In order to track the training state in real time, a checkpoint for the training model was saved every 2 minutes. A seperate process was running for evaluation on the test set, at the same time with the training process. The learner (training process) had no knowledge about the testing result.

V. Results

A short summary of my results can be found in table I. Surprisingly, CNDeep, with 0 dropout rate, reached the highest accuracy, 85.6%, as well as the fastest speed of convergence. As a comparison, human performance on CIFAR-10 was estimated to be 94% [14], while CNBase only reached an accuracy of 80.6%. Although it was claimed that the original impelmentation could also get 86%, that model was trained on Tesla K20m for 5 hours with a batch size of 128, which was nearly infeasible on my laptop with GTX 770M.

Figure 3 and 4 show the change of training error and testing error during the learning process. As in Figure 3, complex models would need longer time to train; however they don't necessarily outperform simpler models in the final accuracy. Particulary, neither CNNiN or CNFmp had outperformed CNDeep (0 dropout). CNFmp even ended with significantly degraded accuracy.

As in figure 3 I have observed the difference in the overfitting among different models. In CNBase overfitting was very serious at the end of training, since we can observe the big gap between test error and training error. Complex models, like CNNiN with dropout and CNFmp with fractional max pooling, had lighter overfitting during the training process, as we can observe the smaller gaps between test error and training error.

TABLE I
Benchmark Summary

Model Name	Details		
	#Parameters	*Test Accuracy %*	*Convergence Time*
CNBase	1.76M	80.6	$>$ 55K steps, 3 hours
CNDeep	2.96M	85.6	55K steps, 2 hours
CNNiN	4.14M	85.5	6K steps, 3 hours
CNFmp	3.35M	79.6	80K steps, 10 hours

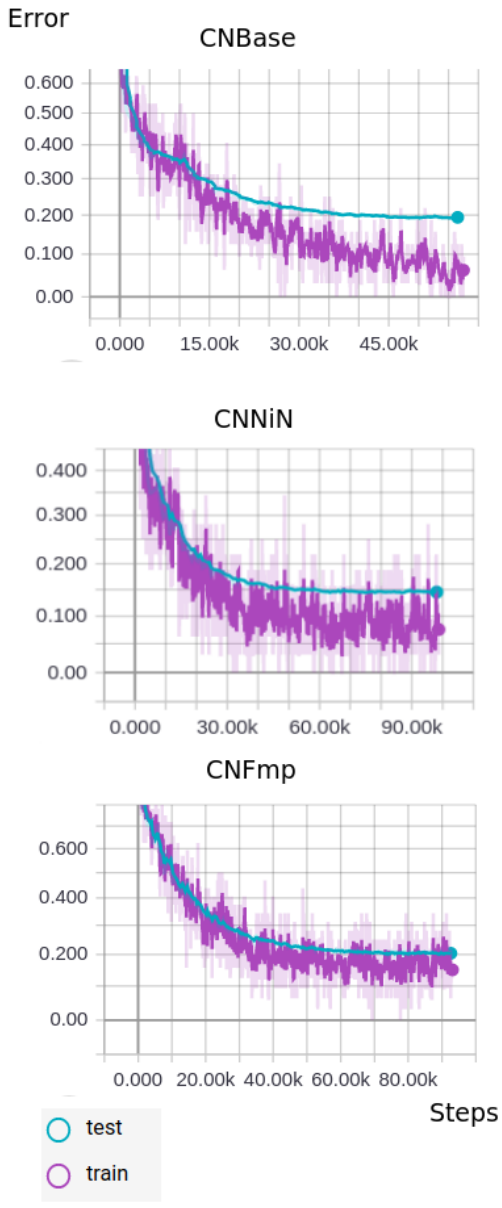

Fig. 3. Training results of CNBase, CNNiN and CNFmp. Training and test errors vs steps. The error axes were log scaled. Visualization was done by Tensorboard.

TABLE II
CNDEEP WITH DIFFERENT DROPOUT RATES

Max Drop Rate	*Test Accuracy %*	*Convergence Time*
0.0	85.6	55K steps, 2 hours
0.2	85.3	70K steps, 3 hours
0.5	83.2	100K steps, 4 hours

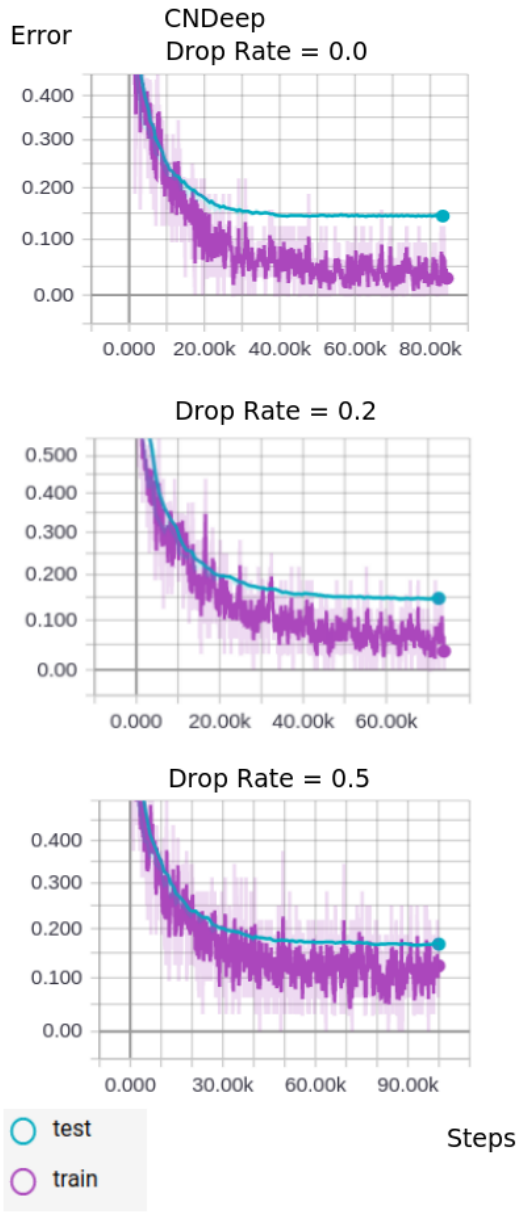

Fig. 4. Training results of CNDeep with differet max dropout rates. The error axes were log scaled. Visualization was done by Tensorboard.

As we know, techniques like dropout could reduce overfitting when training neural networks [3]. To verify this I applied different linearly increasing drop rates to the CNDeep model. The results were summarized in table II and figure 4. The results in figure 4 showed that increasing dropout ratio could reduce overfitting, although it would slow down the learning process. However, we coudl also easily reach another conclution: low overfitting doesn't necessarily mean better accuracy. As in table II, adding dropout ratio could lead to slightly degraded accuracy.

VI. DISCUSSION

What techniques are really useful?: With a simple stacked convolutional architecture, CNDeep had beaten the baseline model CNBase in both accuracy and training speed. However, after applying multiple state-of-art techniques, including dropout, network-in-network and fractional max pooling, none of them could improve the accuracy. Instead, they only slowed down my training speed.

I have two speculations for this result. First, these techniques are probably just not as effective for small networks as they are in the larger counterparts. Indeed, even to solve "tiny" problems like CIFAR-10 classification, we still need very large convolutional networks. Second, apart from the size of network, certain techniques are only effective when combined with certain parameter settings. For example, when applying Adam and Momentum optimizers to train my models, I observed nearly no learning at all, where the test error was always close to 0.1 for a long time. While this should have been caused by my lack of knowledge on how to set the parameters, finding a proper configuration still heavily rely on hands-on experience.

As a result, finding the best parameter settings still look like an open problem to me.

Small networks might not be expressive enough: From this opinion, anyone who is serious on deep learning should consider to work on a high performance computing platform – probably not your own laptop. Building a workstation with high-end GPUs and renting a cloud deep learning environment look like two nice options for personal interests. After all, we tend to believe that larger networks, as well as more computing power, can at least provide a higher potential.

Limits: Using an established deep learning library like Tensorflow made it easy to prototype network models, but it also limits the user to the pre-defined operations. For example, I found it impossible to apply randomly combined operations for image data augmentation. Another instance was that there's no implemention of spatially sparse convolutional networks in tensorflow, which could greatly enhance learning speed [7], [8]. After all, writing python program would not solve these problems, because the code should run on GPUs.

Future Work: I expect my models to also work on the more difficult CIFAR-100 data set, although I only tuned them for CIFAR-10. It is still possible that they will perform poorly on CIFAR-100. Moreover, it would be interesting the test a small

version of residual net [9], [10], which has been on the top of CIFAR-10 and CIFAR-100 leaderboard.

Attempting this image classification project has greatly boosted by interest and confidence in deep learning. Probably my next step will be implementing some models for the task of object detection (with bounding boxes).

REFERENCES

[1] Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." (2009).

[2] Rodrigo Benenson. "Classification datasets results." http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[3] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[4] Graham, Benjamin. "Fractional max-pooling." arXiv preprint arXiv:1412.6071 (2014).

[5] "Convolutional Neural Networks." https://www.tensorflow.org/tutorials/deep_cnn

[6] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).

[7] Graham, Benjamin. "Spatially-sparse convolutional neural networks." arXiv preprint arXiv:1409.6070 (2014).

[8] Graham, Benjamin. "Fractional max-pooling." arXiv preprint arXiv:1412.6071 (2014).

[9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[10] Zagoruyko, Sergey, and Nikos Komodakis. "Wide residual networks." arXiv preprint arXiv:1605.07146 (2016).

[11] De Boer, Pieter-Tjerk, et al. "A tutorial on the cross-entropy method." Annals of operations research 134.1 (2005): 19-67.

[12] Krogh, Anders, and John A. Hertz. "A simple weight decay can improve generalization." Advances in neural information processing systems. 1992.

[13] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." Proc. ICML. Vol. 30. No. 1. 2013.

[14] Andrej Karpathy blog, "Lessons learned from manually classifying CIFAR-10." 2011. http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/