Analysis and Comparison of Deep Learning Techniques for Image Classification using CNNs

Eric Zeng **Patrick Hickey** **Vipul Gharde** **Yating Fang**

Abstract

Deep learning techniques have been developed to address specific challenges related to image classification. However, selecting the optimal implementation of these techniques for a specific task can be challenging. To better understand CNN's, we investigate the effects that hyper-parameters have on a model. We primarily examine three aspects of CNN's: the layer architecture, optimizer, and data augmentation. In particular, different combinations of hyper-parameters are tested, and the accuracy of the models are compared. For convenience, the CIFAR-10[1] data set is used to train the model, and for efficiency, Ray Tune[2] is used to test different combinations of hyper-parameters.

Introduction

Deep learning techniques have been developed to address specific challenges related to image classification using CNN's. These techniques include increasing depth, using regularization to avoid over-fitting, using batch normalization to improve convergence, among others. However, selecting the optimal implementation of these techniques for a specific task can be challenging. Blindly trying different techniques and tuning hyper-parameters can be time-consuming without a solid understanding of the model building techniques.

We aim to investigate and compare the impact of selected deep learning techniques on model accuracy; analyze the effect of different combinations of technique on model performance; implement our own code to ensure full control over experimental setup. The data set that we used was CIFAR-10[1], which consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. This data set is relatively small, allowing us to freely explore different models and techniques without spending too much time on training. We will start with a simple CNN model and optimizer, as well as the original CIFAR-10 data set without modification. Then they will be systematically modified by applying various deep learning techniques. These techniques will be categorized into three main groups: model-related techniques, optimizer-related techniques, and pre-processing related techniques. We will identify existing patterns in our analysis and aim to answer different meaningful questions.

In order to make the process of playing with various deep learning techniques easier and more efficient, we applied a Python library that provides a framework for distributed hyper-parameter tuning and model selection called Ray Tune[2]. Ray Tune provides a framework for automating and parallelizing this process, allowing users to easily search over a large space of hyper-parameters and experiment with different optimization algorithms. The three widely-used search algorithms are grid search, random search, and Bayesian optimization Search. All of these tools allowed us to draw meaningful conclusions from the experimental results. These experimental results may provide some insights for practitioners and motivate further research to provide a comprehensive guideline to navigate the often challenging neural network hyper-parameter tuning.

Methods and Results

There still are many models or hyper-parameters within each group of the deep learning techniques, these hyper-parameters tend to be strongly correlated with each other. However, hyper-parameters across the main groups tend to have less dependence on each other. Due to the relative independence among the three main groups, we decided to experiment on each of them independently before integrating them together with the best version of each group. The purpose of this is to parallelize the process and distribute the computing resources. Afforded by our various conscious choices to reduce computational complexity, we ran a relatively large amount of experiments and identified some good general practises in CNN.

We chose a relatively simple baseline architecture in order to focus on the impacts of hyper-parameter adjustments. Our original architecture consisted of three convolutional layers, with 16, then 32, then 64 channels. We used batch normalization and max pooling after each one. The kernel sizes were 3x3, 3x3, and 6x6 with pad sizes of 2, 1, and 1. After these convolutional layer we had one fully connected layer with 10 nodes, each one signifying a separate category. We used stochastic gradient descent with Nesterov momentum as the optimizer. We used no data augmentation with this model. After optimizing the parameters, our final model's convolutional layers had channel sizes of 32, 64, and 128. Our kernel sizes were 3, 5, and 7 with padding sizes of 2, 3, and 3. The final model also had an additional hidden fully connected layer with 1024 nodes between the final convolutional layer and the 10 node layer. We maintained stochastic gradient descent with momentum, however learning rate and momentum parameters were modified. Our final model performed best using CIFAR-10 Auto-Augmentation. Our hyper-parameter adjustments and data modification were able to bring the over test accuracy from 65.1% to 74.3% while keeping the same overall architecture. The following subsections will explain in detail the experiments and results done on hyper-parameters in model architecture, optimization, as well as data augmentation.

	Pre-tuned	**With Best Hyper-parameters**	**Best Optimizer**	**Best Augmentation**	**Final Model**
Validation Accuracy	71.57%	75.28%	77.07%	78.23%	78.23%

Table 1: Progression of the model performance as each tuning experiment is being integrated into the final model

Tuning via Hyper-parameters in Architecture

Our initial model was never properly tuned, so we adjusted various hyper-parameters to optimize the performance while keeping the overall architecture consistent. The hyper-parameters that we adjusted were kernel size and pad size (for all three convolution layers), amount and size of fully connected layers, channel size, learning rate and optimizers, and data augmentation. Due to the fact that the number of options available scales exponentially with the amount of parameters, our approach was to fine tune them incrementally.

The first values we looked at were kernel size, pad size, and presence of an additional fully connected layer. We used a library called Ray Tune to test various combinations and ran each one for 10 epochs. The top performing combinations can be found in table 3 in Appendix. What we found was that in general, it was best to start with smaller kernel sizes and progress to larger ones. Additionally, pad size is ideally 2-3 pixels smaller than the kernel size. The presence of additional fully connected layers at the end of the architecture made a negligible difference. With these modifications we were able to bring our validation accuracy from 71.57% (baseline model) to 73.58% (improved model). Next, we took the best performing values discovered above, and adjusted the channel sizes (for all three convolutional layers) and the size of the fully connected layer at the end. The results can be seen in table 4 in Appendix. What we found was that increasing channel size increased performance across the board. In general, increasing the size of the fully connected hidden layer added some performance, but it was not nearly as significant as increasing the channel size. With these modifications we were able to bring our validation accuracy from 73.58% to 75.28%, as seen in table 1.

Tuning via Optimization

Our pre-tuned model used stochastic gradient descent with Nesterov momentum as the optimizer. However, optimization can have a significant effect on complex and nonlinear models such as Neural Network. So we try out four different optimizers: stochastic gradient descent, Adagrad, RMSProp, and Adam. For each of these optimizers, distinct learning rates and weight decays are examined, and results of the final performance are compared. For convenience, the learning rate and weight decay are tuned in magnitudes of 10. The learning rates tested include 1×10^{-3}, 1×10^{-2}, 1×10^{-1}, and 1. Likewise, the weight decays evaluated include 1×10^{-4}, 1×10^{-3}, 1×10^{-2}, and 1×10^{-1}. Moreover, we also experiment with momentum for the stochastic gradient descent and RMSProp optimizers. For momentum, we only look at three different values: 0, 0.5, and 0.9. We do not assess the performance of momentum over a large spectrum of values because we are mainly concerned with whether momentum has a noticeable effect.

We first analyze the best performing models, as shown in table 5 in Appendix. Noticeably, all four optimizers have related models among the top ten. This suggests that all optimizers perform well given the right set of hyper-parameters, and different optimizers favor different hyper-parameters. For the benchmark model, it seems that Adagrad achieves the best validation accuracy with the highest being 0.7327. But as a general trend, low learning rates and low weight decays perform the best. Momentum also consistently boosts the performance of stochastic gradient descent and RMSProp.

Conversely, models with a high learning rate perform poorly, evenly with relatively low weight decays. Hence the learning rate has a greater influence on the performance than the weight decay. Table 6 in Appendix shows the worst performing models, and the results are much worse in comparison to the best models. Among the top ten worst performing models, 7 of these models include stochastic gradient descent. The loss of all of these models are "NaN", thus it is possible that the loss is diverging. Hence, stochastic gradient descent is most negatively affected by high learning rates. We tuned the previous model (with the optimized hyper-parameters in architecture) in a similar way (shown in table 7), we were able to improve our accuracy from 75.28% to 77.05%.

Bayesian optimization has gained popularity in recent years due to its effectiveness in finding good hyper-parameters with relatively few evaluations of the objective function. It works by predicting the likelihood of other search points having minimal loss after trying a few, it then choose the most likely one to try next. As it tries more and more search points, it is able to more and more accurately predict the likelihood of the rest of the search points. This contrast with grid search with its purposefulness and ability to learn from past experience. We tried to apply some Bayesian optimization to find better learning rate and weight, it only improved a little, but if we compare the worst trials of grid search and Bayesian optimization, Bayesian optimization only tests trials that are more likely to lead to good results as observed in table 2, which is obviously more desirable. With this, we improved our model's validation accuracy further, to 77.07% before applying any data augmentation, as seen in table 1.

Grid Search Validation Accuracy	10.13%	10.36%	10.55%	10.58%	10.58%	10.58%
Bayesian Search Validation Accuracy	9.86%	10.03%	10.11%	66.13%	66.61%	74.05%

Table 2: Worst 5 performing combination found when tuning via optimization after 10 epochs. This is applied on the previous model optimized for all the hyper-parameters considered in architecture. This is to compare the efficiency of grid search and Bayesian search.

Tuning via Data Augmentation

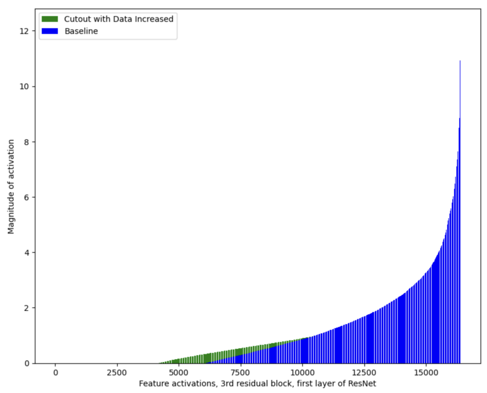

Figure 1: The magnitudes of all pixels (ordered) in the feature map after the third block of the first layer of ResNet

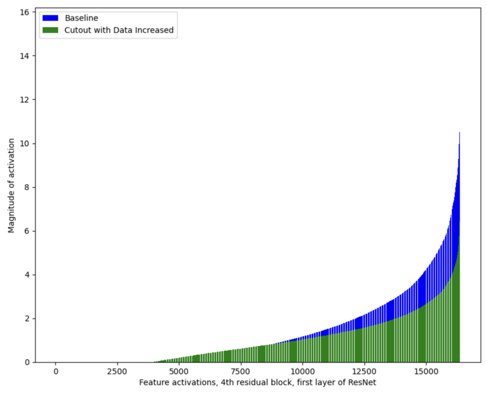

Figure 2: The magnitudes of all pixels (ordered) in the feature map after the fourth block of the first layer of ResNet

Currently our pre-tuned model merely used the unprocessed CIFAR-10 data set. However, CIFAR-10 is generally considered a small data set, which could potentially benefit from optimal data augmentation. Even though architecture is very crucial to the success of neural network, data augmentation can still play a very important role by increasing the diversity and quantity of the training data, leading to better model performance and greater generalization to new, unseen data. Unfortunately, data augmentation techniques used by Krizhevsky et al. for AlexNet back in 2012 remain largely the same, with only minor changes. The idea of optimal augmentation techniques calls for the use of machine learning. Auto-Augmentation was introduced in 2018 by Google, it uses a reinforcement learning algorithm to search for the best data augmentation policies. One exciting thing that Auto-Augmentation allows us to do is to extend the idea of transfer learning from weights of architecture to augmentation policies.

We selected four different data augmentation techniques for CIFAR-10 data set: the Auto-Augment policy previously optimized for CIFAR-10, the Auto-Augment policy previously optimized for ImageNet, mix-up, and cutout. Mix-up and cutout were chosen because they are some of the newer data augmentation techniques, introduced in 2018 and 2017 respectively. Mix-up takes the convex combination of two images to create a new image. It is particularly effective when data is limited, noisy, or biased; Cutout randomly masks out a rectangle region of the image and replace those pixel values with zeros. It works by forcing the model to rely on more diverse and invariant regions. In addition to experimenting with these data augmentation techniques using our pre-tuned model, we also experimented with those using a simple implementation of ResNet. The figures 1 and 2 plotted the magnitude of each pixel for a given feature map. The pixels are ordered according to their magnitudes to provide a more organized visualization. It is clear from the figures that cutout acts as an equalizer among all the pixels in a given feature map. Compared to baseline (which is no augmentation), feature maps with cutout implemented tend to give less extreme favouritism to the important pixels, and more attention to the previously ignored pixels. This shows that cutout works by forcing the model to rely on more diverse and invariant regions.

To compare different augmentation techniques, we applied all of them both on our pre-tuned model as well as a simple ResNet. Data augmentation aims to diversify a data set, with or without increasing the size of the data set. So for all the techniques besides mix-up, we also explored with directly using the transformed data set, rather than just augmenting the transformed data set to the original data set. Only one epoch was run to produce the validation accuracy of each combination, shown in table 8 in Appendix. We could conclude from the table that increasing data size is almost always better; mix-up can be comparable to Auto-Augmentation, showing its high potential; Auto-Augmentation produce the best results, even if it was optimized for a different data set like ImageNet. Our experiment helped to confirm that transfer learning for augmentation could be hugely beneficial. After applying the best data augmentation we found to the previously-tuned model (tuned for both the architecture hyper-parameters as well as the optimization), we increased our validation accuracy to the final accuracy of 78.23% as seen in table 1.

Conclusion

In order to run a relatively large amount of experiments, we made various conscious choices to reduce computational complexity: parallelize among main types of deep learning techniques; experiment incrementally with hyper-parameters in architecture; used Ray Tune to automate the process of tuning; considered Bayesian optimization to reduce time wasted on regions of search space that are not likely to lead to good results. We identified the following good general practises in CNN: moving from smaller to larger kernel sizes worked better for subsequent convolutional layers; increasing channel sizes almost always increased performance within our experiments, although the downside of it may arise such as over-fitting and expensive computing resources, more experiments would be needed to further explore that; Increasing the amount and size of fully connected layers had a minimal impact, this is probably because the features learned by the convolutional layers are often more important, while the fully connected layers learned to combine these features into a final prediction; optimization methods had a very large impact on performance, more than that optimal global learning rates vary widely with different methods. Optimization tends to have a bigger effect on a more complex loss function landscape and Neural Network has some of the most complex landscapes; Best augmentation technique tried was Auto-Augmentation optimized for CIFAR-10 and Auto-Augmentation optimized for ImageNet. In essence, Auto-Augmentation uses machine learning to find optimal data augmentation. Even if the augmentation was optimized for a different data set, it would still transfer well and perform better than normally applied augmentation on a new data set. These experimental results may provide some insights for practitioners and motivate further research to provide a comprehensive guideline to navigate the often challenging neural network hyper-parameter tuning.

References

1 https://www.cs.toronto.edu/ kriz/cifar.html

2 https://docs.ray.io/en/latest/tune/examples/tune-pytorch-cifar.html

Appendix

Validation Loss	**Validation Accuracy**	**Extra FC**	**Kernel Size 1**	**Kernel Size 2**	**Kernel Size 3**	**Pad Size 1**	**Pad Size 2**	**Pad Size 3**
0.7845	0.7358	N	3	3	5	1	1	3
0.8052	0.7303	Y	3	5	5	2	2	2
0.8222	0.7268	N	5	5	5	2	2	3
0.8145	0.7248	N	3	5	5	2	2	2
0.8219	0.7248	Y	3	5	7	2	2	3
0.8055	0.7246	N	3	5	7	1	2	2
0.8053	0.7237	Y	3	5	5	2	1	3
0.8069	0.7235	Y	5	5	7	2	1	3
0.8005	0.7233	N	5	5	5	1	2	3
0.7877	0.7233	Y	3	5	5	1	2	3

Table 3: Best 10 performing combinations found when tuning via hyper-parameters in architecture after 10 epochs. The hyper-parameters include kernel sizes, pad sizes, and the choice to add/remove the fully connected layer. This is applied directly on the pre-tuned model.

Validation Loss	**Validation Accuracy**	**Channel 1**	**Channel 2**	**Channel 3**	**FC Size**
0.8056	0.7528	32	64	128	1024
0.7781	0.7421	32	32	128	512
0.7636	0.7420	32	64	64	1024
0.7816	0.7415	32	64	64	256
0.8178	0.7398	32	64	128	256
0.7624	0.7376	32	32	128	256
0.8032	0.7365	16	64	64	512
0.7915	0.7338	32	64	128	512
0.7989	0.7328	32	64	64	512
0.8054	0.7313	32	32	128	1024

Table 4: Best 10 performing combinations found when tuning via hyper-parameters in architecture after 10 epochs. The hyper-parameters include the number of channels for each convolutional layer and the fully connected layer size. This is applied on the previous model that was optimized for kernel sizes and pad sizes.

Optimizer	Learning Rate	Weight Decay	Momentum	Validation Loss	Validation Accuracy
Adagrad	0.01	0.01	0.5	0.781422	0.7327
Adagrad	0.01	0.01	0	0.769208	0.7326
RMSprop	0.001	0.001	0.5	0.790606	0.7315
Adagrad	0.01	0.01	0.9	0.777699	0.7299
SGD	0.01	0.0001	0.9	0.808381	0.7279
RMSprop	0.001	0.001	0	0.815173	0.7269
SGD	0.1	0.001	0.5	0.835252	0.7266
Adagrad	0.01	0.001	0.5	840388	0.7258
Adam	0.001	0.001	0	0.894687	0.7245
SGD	0.01	0.0001	0	0.940317	0.7241

Table 5: Best 10 performing combination found when tuning via optimization after 10 epochs. This is applied on the initial pre-tuned model

Optimizer	Learning Rate	Weight Decay	Momentum	Validation Loss	Validation Accuracy
SGD	1.0	0.001	0.5	NaN	0.0971
SGD	1.0	0.01	0.9	NaN	0.0983
SGD	1.0	0.001	0	NaN	0.0990
SGD	1.0	0.01	0	NaN	0.0990
Adam	0.1	0.1	0	2.307736	0.0992
RMSprop	0.01	0.5	0	3338.569891	0.1007
SGD	1.0	0.001	0.9	NaN	0.1008
SGD	1.0	0.1	0.5	NaN	0.1015
Adam	1.0	0.1	0	2.36891	0.1016
SGD	1.0	0.1	0.9	NaN	0.1017

Table 6: Worst 10 performing combinations found when tuning via optimization after 10 epochs. This is applied on the initial pre-tuned model

Validation Loss	Validation Accuracy	Optimizer	Learning Rate	Momentum	Weight
0.7090	0.7705	SGD w/ Momentum	0.050	0.500	0.0010
0.7080	0.7697	SGD	0.050	0.500	0.0010
0.6761	0.7693	Adam	0.001	0.001	0.0010
0.7428	0.7651	Adam	0.001	0.500	0.0010
0.6814	0.7646	SGD	0.010	0.500	0.0100
0.8068	0.7628	SGD	0.050	0.001	0.0010
0.7206	0.7621	RMS	0.001	0.001	0.0010
0.7198	0.7611	SGD w/ Momentum	0.050	0.001	0.0010
0.8291	0.7608	SGD w/ Momentum	0.050	0.500	0.0001
0.7031	0.7601	SGD	0.050	0.500	0.0001

Table 7: Best 10 performing combination found when tuning via optimization after 10 epochs. This is applied on the previous model optimized for all the hyper-parameters considered in architecture

Pre-tuned Model	**Validation Accuracy (1 epoch)**	**ResNet**	**Validation Accuracy (1 epoch)**
CIFAR-10 Auto-Augmentation + increase data	63.35%	CIFAR-10 Auto-Augmentation + increase data	57.00%
ImageNet Auto-Augmentation + increase data	60.17%	ImageNet Auto-Augmentation + increase data	51.00%
Cutout + increase data	59.54%	Cutout + increase data	50.57%
CIFAR-10 Auto-Augmentation	56.66%	Mix-up	41.50%
Mix-up	56.33%	CIFAR-10 Auto-Augmentation	39.79%
Cutout	51.65%	ImageNet Auto-Augmentation	38.84%
ImageNet Auto-Augmentation	48.26%	Cutout	38.19%

Table 8: Validation accuracy of different augmentation techniques after 1 epoch for both pre-tuned model as well as ResNet. They are listed in descending order of accuracy, from top to bottom. This experiment is to just compare augmentation techniques, the validation/training split are slightly different from other experiments.