Discovering Better Hyperparameters and Models: CNN Experiements on Tiny ImageNet

Kim Jasper, Mui
A12736589
kmui@ucsd.edu

Yang, Xinyi
A92048196
xiy078@ucsd.edu

Abstract

Optimizer is one of the most essential hyperparameters when training convolutional neural network (CNN) models. Certain optimizers are easier to use when training models, because they are easily adjustable and get less influence on changes of other variables such as numbers of layers, pooling methods and activation functions. In order to find the usefulness of the optimizers and find higher accuracy on tiny Imagenet dataset, we chose VGG-like model from keras to test on different optimizers, and use the highest accuracy attained as a standard to adjust other optimizers. For the models we built ourselves, we experimented on different optimizers, different layers of network, different pooling methods, and activation functions to find the best variables that gives us highest accuracy in the test dataset. In the tiny Imagenet dataset, we get the highest test accuracy of 0.39 with model1, which we built by ourselves and using Adam as the optimizer. Generally, Adam and Adamax are easier to use for training in practical, more layers of network tends to give higher accuracy, and ReLu is better than other activation functions in training the models.

1 Introduction

Over the past years, people are trying to identify the objects in their images. For example, police needs to identify the objects in the surveillance camera in order to find clues to solve the cases. Furthermore, people need to identify more specific category inside each genre. For example, when people are playing outside, they are curious about the kind of the flowers they see, rather than only knowing that the object is a flower. Consequently, this technique can further be used in human face detection, which means that after we collect much data on a certain person, the model can tell the emotion of this person based on the facial movement and expression.

This can be done in convolutional neural network (CNN). As the introduction of CNN emerges into the technical world, people attain higher accuracy on recognizing an object from training many image sets by developing powerful architectures and modifying the value of parameters on the existing architectures. In researches, the results showed that CNN with more layers and more complicated architecture has better performance. However, as the number of layers and the complexity of the architecture increases, we need longer time to train a model. This becomes very inefficient when doing experiment, and lots of time is wasted in order to find a desired model.

Training convolutional neural network not only takes a lot of time but is also hard to choose a suitable architecture with good parameters for the dataset. We want to speed up the training by helping researchers choose suitable architecture and good parameters. Although many other variables may also matter, we mainly focused on the one that is more significant, optimizer, and how to choose one that is easy to train and can still get higher accuracy on the dataset. For extension, we've also built our own model. Other than the optimizers, we also changed the values of other features like number of layers, kinds of pooling layers, and activation functions, which are easier to modify and get relatively comprehensive results; we experimented on how the changes on these features affected the accuracy of the test set.

If we want to analyze the category of the objects in images, we first need to identify the general genre of the object. In order to solve this problem, we used Tiny Imagenet dataset, which contains

200 different classes. We chose several different architectures: VGG-like model, VGG, and three self-built models. For the VGG-Like model, we mainly focused on discovering which optimizer is easier to train than others. In this model, we tried six different optimizers: Adadelta, Adam, Adagrad, RMSprop, SGD, and Adamax. For the self-built models, we did not only compare the effect of optimizers (Adam and Adadelta), but also experimented on other factors like numbers of layers, pooling layers, and activation functions.

2 Dataset Overview

We used Tiny ImageNet dataset from Stanford website. All images in the dataset have size of 64 pixels $\times$ 64 pixels. Note that for this dataset, although a 10000 test set is provided in the Tiny ImageNet dataset, we do not have the label to those images; thus we cannot use the test data provided.

2.1 Training Set

There are 200 distinct classes in the dataset, and it contains 80000 images in total (200 distinct classes $\times$ 400 images in each class).
The specific configuration of for each class in the training set is as follow:

- 500 images in this category. Each image is labeled by as WordNetId_PictureId, where Wordnet Id is the Id number for this category in the wordnet website and picture id is the index of picture in this class.

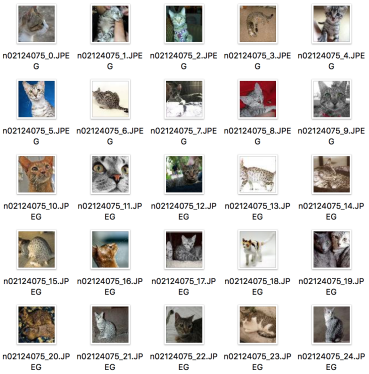

Figure 1: *Training set Images*

- A text file that contains the information for bounding boxes of each image in this class. Since all images have the same size, bounding boxes provide researchers with more information of where the object is in each image. This file contains the id of the image, x and y values of the upper-left corner, x and y values of the lower-right corner of the bounding box.

```
n02124075_0.JPEG        1       0       53      63
n02124075_1.JPEG        1       0       60      63
n02124075_2.JPEG        4       2       47      63
n02124075_3.JPEG        12      10      59      55
n02124075_4.JPEG        7       0       60      63
n02124075_5.JPEG        7       0       63      63
n02124075_6.JPEG        0       11      63      60
n02124075_7.JPEG        0       4       63      43
n02124075_8.JPEG        0       2       63      63
n02124075_9.JPEG        0       0       59      63
n02124075_10.JPEG       1       0       63      63
n02124075_11.JPEG       0       1       63      63
n02124075_12.JPEG       0       5       58      60
n02124075_13.JPEG       0       2       63      62
n02124075_14.JPEG       0       3       63      57
n02124075_15.JPEG       1       6       63      61
n02124075_16.JPEG       0       20      48      63
n02124075_17.JPEG       0       17      13      52
n02124075_18.JPEG       4       0       63      62
n02124075_19.JPEG       0       0       57      63
n02124075_20.JPEG       0       2       58      33
n02124075_21.JPEG       3       9       63      59
n02124075_22.JPEG       15      10      63      63
n02124075_23.JPEG       0       5       63      56
n02124075_24.JPEG       17      1       54      63
n02124075_25.JPEG       0       11      55      50
n02124075_26.JPEG       0       11      63      63
n02124075_27.JPEG       0       7       63      62
n02124075_28.JPEG       0       11      63      56
```

Figure 2: *Three Distinct Text Files*

2.2 Validation Set

We split the last 100 images from each of the training class to be put in the validation set. There are 200 distinct classes in the validation set, and it contains 20000 images in total (200 distinct classes $\times$ 100 images in each class). The structure and file name are the same as those in the training set. We used the validation set to find the best parameters we can get and calculate the final accuracy on the test set.

2.3 Test Set

There are 10000 images in total in the test dataset. The specific configuration for each class in the test set is as follow:

- 50 images in each category. Each image is labeled as "$val\_PictureId$", where PictureId is the index of the picture in the set. The id is from 0 to 9999.

Figure 3: *Original Validation set Images (used as test set in the experiment setting)*

- There is a text file that contains the labels and information for bounding boxes of each image in the dataset. This file contains the image id, true label, x value and y values of the upper-left corner, and x and y values of the lower-right corner of the bounding box.

```
val_0.JPEG      n03444034      0      32      44      62
val_1.JPEG      n04067472      52     55      57      59
val_2.JPEG      n04070727      4      0       60      55
val_3.JPEG      n02808440      3      3       63      63
val_4.JPEG      n02808440      9      27      63      48
val_5.JPEG      n04399382      7      0       59      63
val_6.JPEG      n04179913      0      0       63      56
val_7.JPEG      n02823428      5      0       57      63
val_8.JPEG      n04146614      0      31      60      60
val_9.JPEG      n02226429      0      3       63      57
val_10.JPEG     n04371430      37     38      44      45
val_11.JPEG     n07753592      0      1       63      47
val_12.JPEG     n02226429      5      12      57      53
val_13.JPEG     n03770439      21     33      36      43
val_14.JPEG     n02056570      0      19      63      47
val_15.JPEG     n02906734      1      0       48      22
val_16.JPEG     n02125311      7      16      60      55
val_17.JPEG     n04486054      0      9       63      63
val_18.JPEG     n04285008      0      11      63      57
val_19.JPEG     n03763968      18     6       53      63
val_20.JPEG     n03814639      16     22      33      36
val_21.JPEG     n03837869      40     7       51      63
val_22.JPEG     n01983481      27     28      61      63
val_23.JPEG     n01629819      0      1       52      48
val_24.JPEG     n04532670      0      19      63      31
val_25.JPEG     n04074963      40     0       58      63
val_26.JPEG     n04540053      36     0       44      6
val_27.JPEG     n04371430      2      5       63      54
val_28.JPEG     n02906734      37     24      52      32
```

Figure 4: *Original Validation set Text File (used as test set in the experiment setting)*

2.4 Challenges

- Size:
  Since the Tiny ImageNet dataset contains many images, the size of the whole folder is very large. We've tried to upload the whole dataset onto the server at first, but it did not work. After that, we decided to preprocess the data first by converting images into RGB matrices and save into a new file. However, even the preprocessed data is 3.6GB large. We then tried to compress the file and finally reached 1.6GB and successfully place the file onto the server.

- Categories:
  In the Tiny ImageNet dataset, there are 200 distinct classes in total. Compare to the MNIST dataset, which only contains 10 distinct classes, it is hard for our models to predict as accuracy as the prediction on the MNIST dataset.

- A good deep learning platform:
  We've tried to use Tensorflow for our first approaches. However, at this time, Tensorflow is hard to use and train. It not only takes a lot of time, but also error prone; it did not generate accurate results. We then changed to Keras. Compare to Tensorflow, Keras is more high-level. Fortunately, Keras creates less error and is easier to use in this situation.

3 Data Preprocess

In order to convert input images into data that can be processed by the models, we need to represent each images into matrix. For each image, we need to use one matrix to represent the R(ed), G(leen), B(lue) values for each pixels in the image. Since the images are 64 pixels $\times$ 64 pixels, we can convert the training set, validation set, and test set into 400 * 200 * [64 * [64 * [3]]], 100 * 200 * [64 * [64 * [3]]] and 10000 * [64 * [64 * [3]]] matrices. We also store the labels correspondingly with the new representation of the dataset created.

4 Models and Methods

We ran all of our models on the GPU, and here is the demo screenshot of the GPU usage:

```
Sat Dec 16 12:35:59 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:86:00.0 Off |                  N/A |
| 38%   66C    P2   236W / 250W |  10797MiB / 11172MiB |     96%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Figure 5: *GPU Usage*

4.1 VGG Like Model

Here is the board view of the VGG like model.

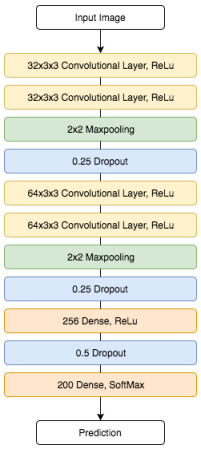

Figure 6: *VGG Like Model*

For the first model, we tried to implement this VGG-like model with the reference from Keras documentation. Since this is our first time using Keras, the purpose of this model is to make sure that the set up for Keras is correct, and we can also learn from this model and apply what we've learned to the model we built.

The network structure is shown above. For the two 2 $\times$ 2 Maxpooling, we used stride size of 2. To reduce more dropout, we also add L1 and L2 regularization to two dense layers. In the 256 dense layer, we used L1 and L2 regularization from Keras, and set both L1 and L2 to 1e-6; in the 200 dense layer, we set both L1 and L2 to 1e-5. We ran 100 epochs on all optimizers for this section.

In this model, we tried six distinct optimizers: Adadelta, Adagrad, Adam, Adamax, RMSprop

and SGD. For each experiment, we've tried to modify other parameters to get highest accuracy on each optimizer and see how easy to train the model using different optimizers. The chart below shows the accuracy of training and test set with different optimizers.
Below are the specific settings for each optimizer:

- Adadelta:
  lr (Learning rate): 0.5.
  rho: 0.95.
  epsilon (Fuzz Factor): 1e-8.
  decay (Learning rate decay over each update): 0.0.
  Epoch: 100.

- Adagrad:
  lr (Learning rate): 1e-3.
  epsilon (Fuzz Factor): 1e-8.
  decay (Learning rate decay over each update): 0.0.
  Epoch: 100.

- Adam:
  lr (Learning rate): 1e-4.
  $beta\_1$ = 0.9.
  $beta\_2$ = 0.999.
  epsilon (Fuzz Factor): 1e-8.
  decay (Learning rate decay over each update): 0.0.
  Epoch: 100.

- Adamax:
  lr (Learning rate): 2e-3.
  $beta\_1$ = 0.9.
  $beta\_2$ = 0.999.
  epsilon (Fuzz Factor): 1e-08.
  decay (Learning rate decay over each update): 0.0.
  Epoch: 100.

- RMSprop:
  lr (Learning rate): 1e-4.
  rho: 0.9.
  epsilon (Fuzz Factor): 1e-8.
  decay (Learning rate decay over each update): 0.0.
  Epoch: 100.

- SGD:
  lr (Learning rate): 1e-3.
  momentum: 0.0.
  decay: 0.0.
  nesterov (Whether to apply Nesterov momentum): False.

Table 1: *Accuracy on Training, Validation and Testing set with different Optimizers*

Optimizer	Training	Validation	Testing
Adadelta	0.0.3558	0.2706	0.2615
Adagrad	0.2148	0.2036	0.2048
Adam	0.4378	0.2489	0.2517
Adamax	0.4361	0.2803	0.2794
RMSprop	0.2417	0.2452	0.2482
SGD	0.0175	0.0249	0.0230

From the table above, we've found out that Adamax and Adam are easier to train than all other optimizers. During training, the accuracy of VGG-like model using Adamax increases in the early epoch and reaches highest accuracy on the validation set and test set.

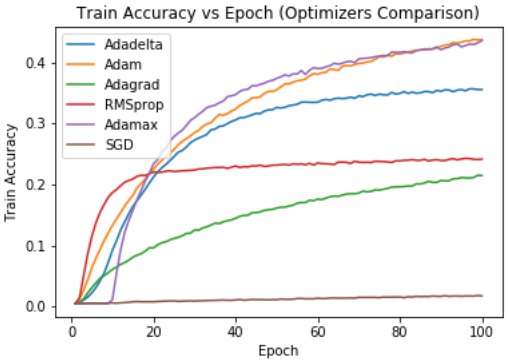

Figure 7: *Training Accuracy vs Epoch*

The graph above shows the training accuracy. The x axis is the epoch and y axis is the training accuracy. The accuracies for Adadelta, Adam and Adamax tend to increase for the training dataset, and this is easier to train than others. For other optimizers like RMSprop, Adagrad and SDG, SGD are the optimizers that are harder to train.

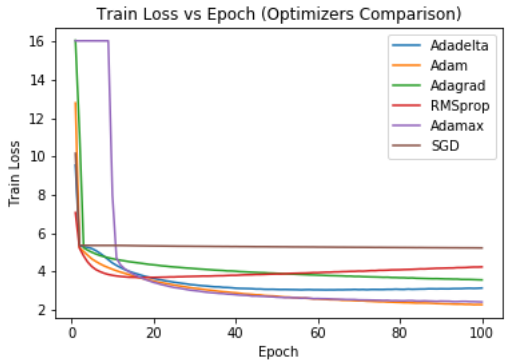

Figure 8: *Training Loss vs Epoch*

The graph above shows the training loss. The x axis is the epoch and y axis is the training loss. In those optimizers, we can confirm that Adamax and Adam perform relatively better than other optimizers and are easier to train, get higher accuracy, and lower loss on the training loss.

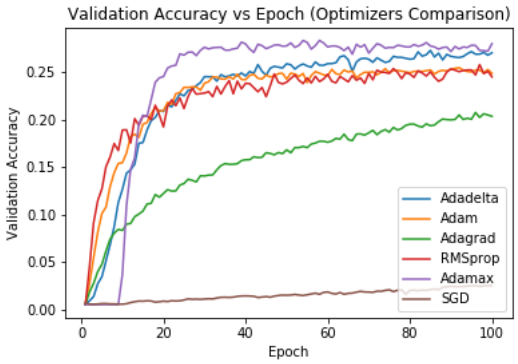

Figure 9: *Training Accuracy vs Epoch*

The graph above shows the validation accuracy. The x axis is the epoch and y axis is the validation accuracy. Adamax also performs better than other optimizers on validation accuracy.

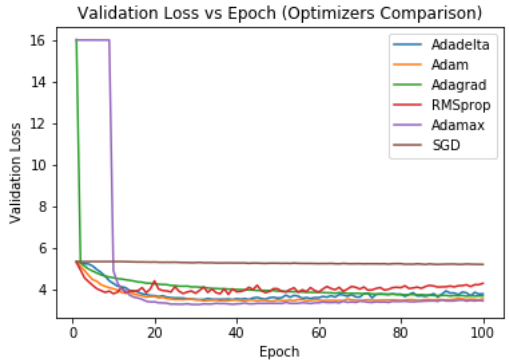

Figure 10: *Training Loss vs Epoch*

The graph above shows the validation loss. The x axis is the epoch and y axis is the validation loss. We get relatively the same result on the validation loss.

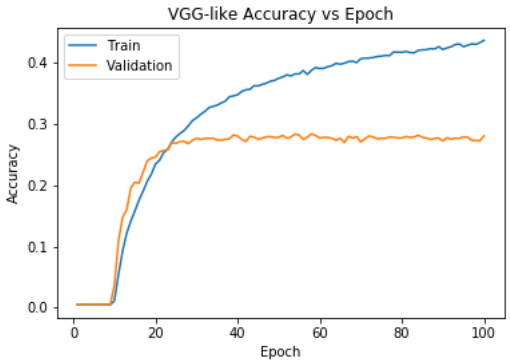

Figure 11: *VGG-Like Accuracy vs Epoch*

The graph above shows the training and validation accuracy of Adamax, which is the optimizer with the best accuracy. We plot this graph in order to see whether the model overfits the data. In the plot above, training accuracy keeps on increasing, but validation accuracy stops at 20s epoch. That is to say, the model begins to overfit starting from 20s epoch.

For this section, we found out that Adamax and Adam are easier to use for training and is able to get the highest accuracy on the test set.

4.2 Self-built Model 1

The broad view of this self-built model is shown in figure 11, first column.

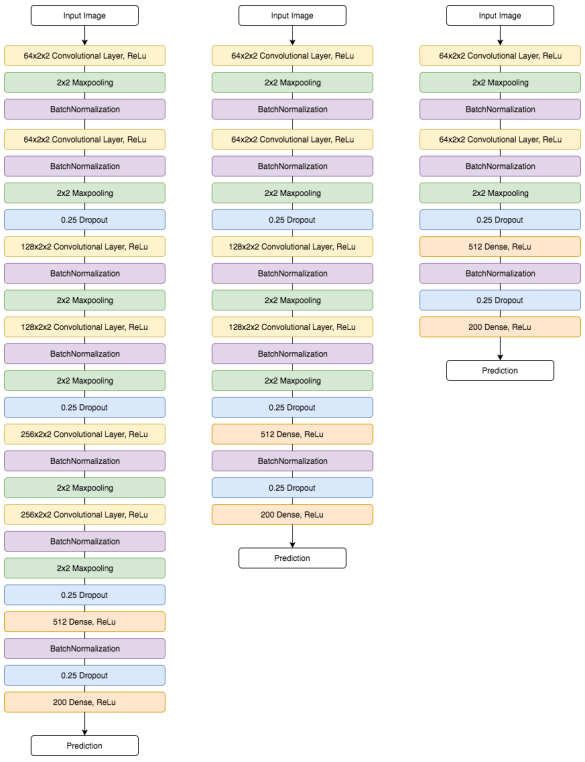

Figure 12: *Three Distinct Models*

We get the inspiration of this model from the VGG model. In this model, there are 8 layers: 6 convolutional layer and 2 dense layer. We also used batch normalization, 0.25 dropout and Maxpooling in this model. We used Adam optimizer with lr (Learning rate) = 1e-4, $beta\_1$ = 0.9, $beta\_2$ = 0.99, epsilon (Fuzz Factor): 1e-08, decay (Learning rate decay over each update): 0.0, Epoch: 100.

4.3 Self-built Model 2

The broad view of this self-built model is shown in figure 11, second column.

There are 6 layers in this model. Comparing to the

first model, the second model has two 256x2x2 convolutional layers less than the first model. We used Adam optimizer with lr(Learning rate) = 1e-4, $beta\_1$ = 0.9, $beta\_2$ = 0.99, epsilon(Fuzz Factor): 1e-08, decay(Learning rate decay over each update): 0.0, Epoch: 100.

4.4 Self-built Model 3

The broad view of this self-built model is shown in figure 11, third column.
There are 4 layers in this model. Comparing to the second model, the third model has two 128x2x2 convolutional layers less than the second model. We used Adam optimizer with lr (Learning rate) = 1e-4, $beta\_1$ = 0.9, $beta\_2$ = 0.99, epsilon (Fuzz Factor): 1e-08, decay (Learning rate decay over each update): 0.0, Epoch: 100.

4.5 Self-built Model Comparison

The accuracy we got from the three self-built model is shown below: We did the following comparisons between three models, in order to choose the better optimizers and get the higher accuracy.

Table 2: *Accuracy on Three Self-built Model*

Model	Training	Validation	Testing
model1	0.4457	0.3626	0.3606
model2	0.5271	0.3627	0.3595
model3	0.7484	0.1980	0.1965

- Comparison between Number of Layers:

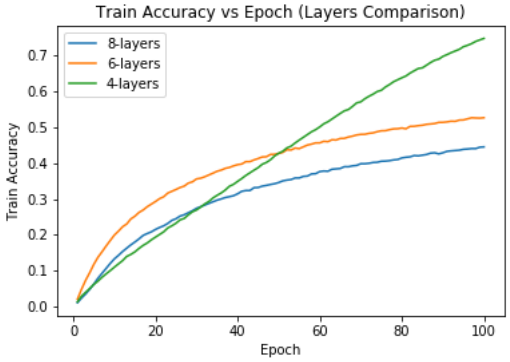

Figure 13: *Train accuracy on different layers vs Epoch*

The above graph is the accuracy on the training dataset on different numbers of layers. In this graph, we found out that the training accuracy for 4-layer model is higher than other two layers. However, in order to check whether the model overfits on the training dataset, we still need to find the accuracy and loss on the validation dataset.

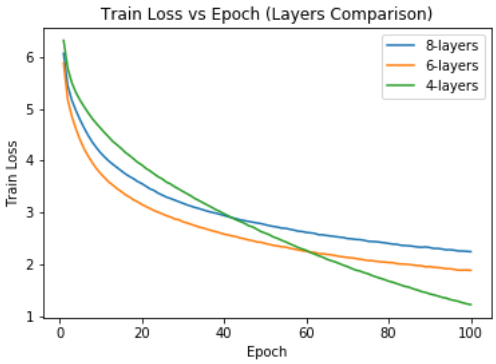

Figure 14: *Train Loss on different layers vs Epoch*

The above graph is the loss on the training dataset on different numbers of layers.

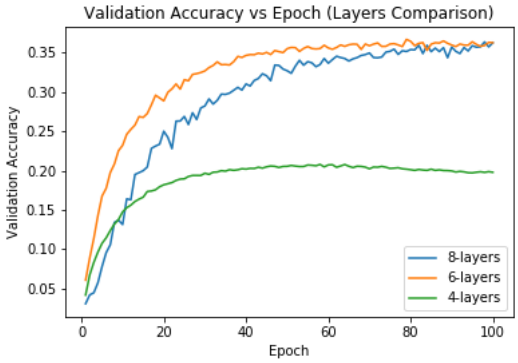

Figure 15: *Validation Accuracy on different layers vs Epoch*

The above graph is the accuracy on the validation dataset on different numbers of layers. In this graph, we found out that the validation accuracy for 6-layer and 8-layer model is higher than 4-layer model. From the information above, we found out that although the accuracy of training dataset for 4-layer network is the highest, the 4-layer model is overfitting.

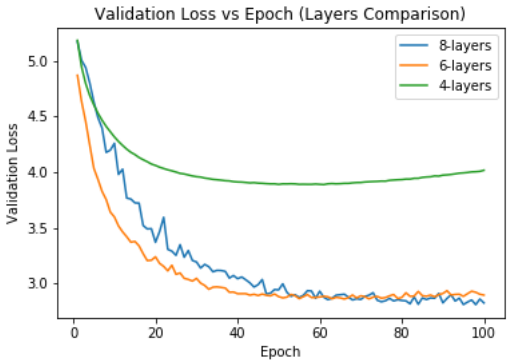

Figure 16: *Validation Loss on different layers vs Epoch*

The above graph is the loss on the validation dataset on different numbers of layers.

- Comparing Pooling Methods: Maxpooling and Averagepooling. From the table, we can

Table 3: *Accuracy on Two Distinct Pooling Methods*

Pooling Layers	Training	Validation	Testing
Maxpooling	0.4457	0.3626	0.3606
AveragePooling	0.4755	0.3968	0.3946

tell that average pooling gives higher accuracy in this model. The details are presented in the following graphs.

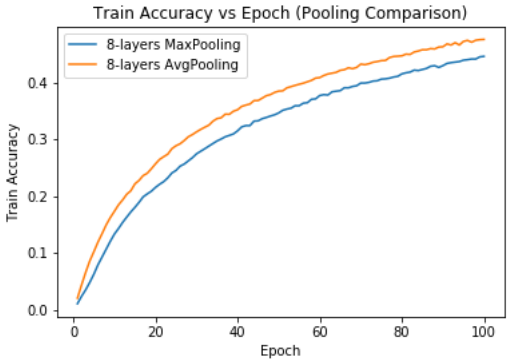

Figure 17: *Different Pooling Method Training Accuracy on 8-layer model vs Epoch*

The above graph is the accuracy of the 8-layer model using maxpooling and average pooling on the training set. From the graph, we found that average pooling gives higher accuracy. We still need the validation set to check whether average pooling is better than maxpooling in this situation.

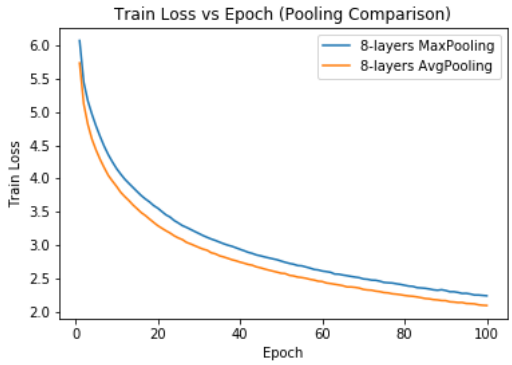

Figure 18: *Different Pooling Method Training Loss on 8-layer model vs Epoch*

The above graph is the loss on the 8-layer model using maxpooling and average pooling on the training set.

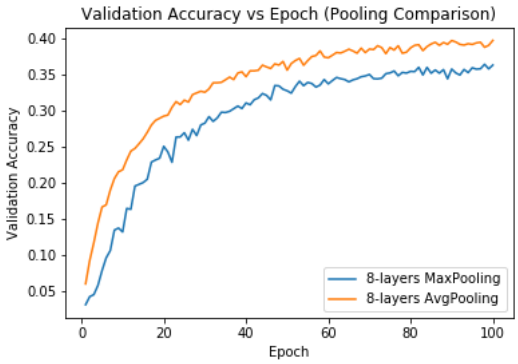

Figure 19: *Different Pooling Method Validation Accuracy on 8-layer model vs Epoch*

The above graph is the accuracy of the 8-layer model using maxpooling and average pooling on the validation set. From the graph and the information above, we found that in both training and validation set, average pooling attains higher accuracy. We now are sure that the model is not overfitting.

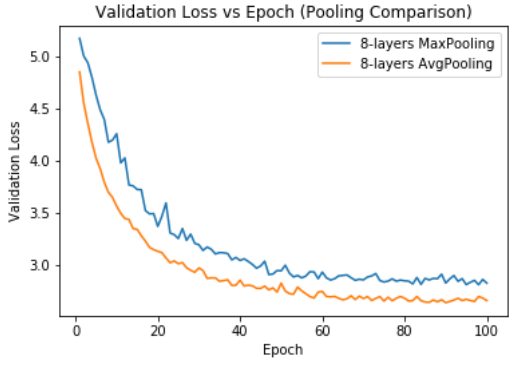

Figure 20: *Different Pooling Method Validation Loss on 8-layer model vs Epoch*

The above graph is the loss on the 8-layer model using maxpooling and average pooling on the validation set.

- Comparison between Activation Functions: ReLu, ELu and Sigmod function. The accuracies of different activation functions on the training and testing dataset are shown below: The graphs below shows the detail of the performance of each activation function.

Table 4: *Accuracy on Three Distinct Activation Functions*

Activations	Training	Validation	Testing
ReLu	0.4755	0.3968	0.3946
ELu	0.4453	0.3919	0.3904
Sigmoid	0.3651	0.3420	0.3355

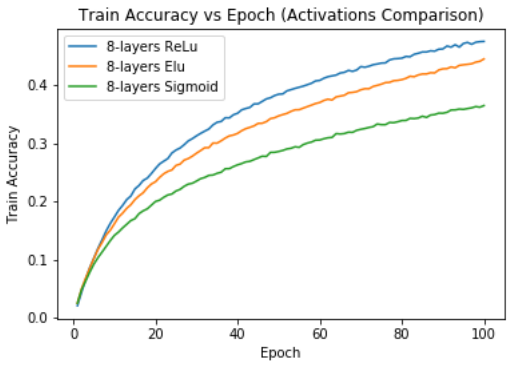

Figure 21: *Different Activation Function Training Accuracy on 8-layer model Average Pooling vs Epoch*

The above graph is the accuracy on the 8-layer model with average pooling using three different activation functions on the training set. From the graph, we found that ReLu is the best activation function to use, and Sigmod is the worst, which is the same as expected.

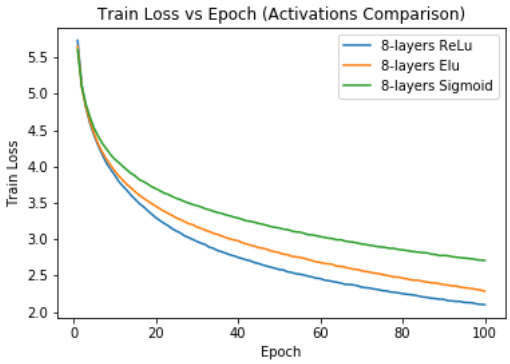

Figure 22: *Different Activation Function Training Loss on 8-layer model Average Pooling vs Epoch*

The above graph is the loss on the 8-layer model with average pooling using three different activation functions on the training set.

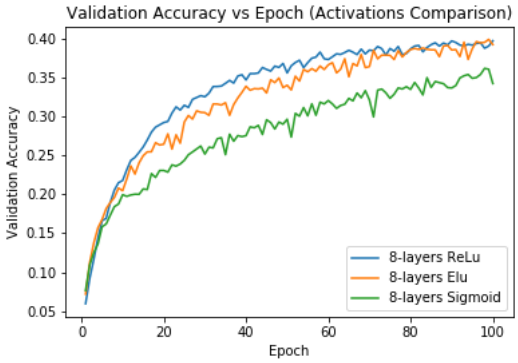

Figure 23: *Different Activation Function Validation Accuracy on 8-layer model Average Pooling vs Epoch*

The above graph is the accuracy on the 8-layer model with average pooling using three different activation functions on the validation set. From the graph, we found that ReLu is the best activation function on both training and validation set. We can now confirm that ReLu is a better activation function.

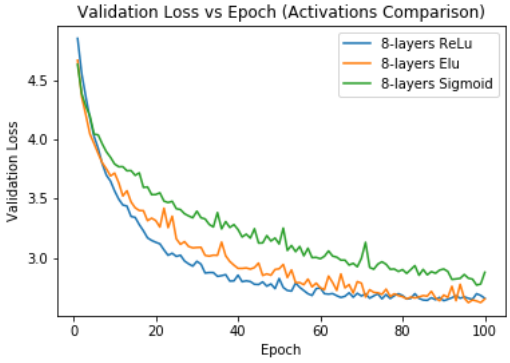

Figure 24: *Different Activation Function Validation Loss on 8-layer model Average Pooling vs Epoch*

The above graph is the accuracy on the 8-layer model with average pooling using three different activation functions on the validation set.

In the three models above, we found that the best model is the 8-layer model with average pooling and ReLu. The accuracy on the training and validation is as follow:

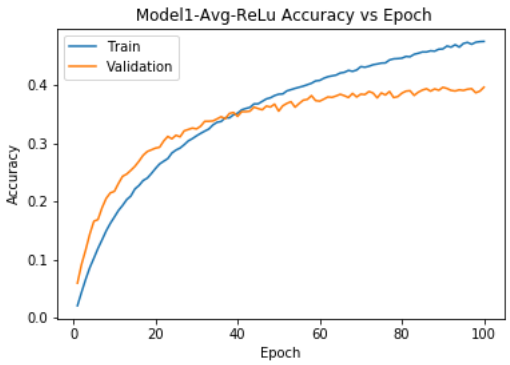

Figure 25: *Best model chosen*

4.6 VGG-16

The broad view of the VGG architecture is as follow:

Figure 26: *VGG-16*

VGG contains 16 layers in total. We implemented this architecture, and used the parameters listed below.

Optimizer: Adadelta.

Learning rate: 0.001.

Epoch: 100.

Here is the result we get:

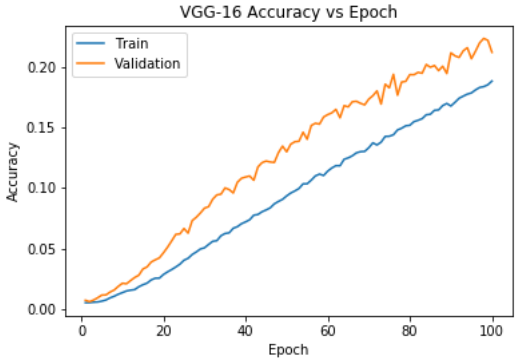

Figure 27: *VGG-16 Accuracy*

This is the accuracy of training and validation set of VGG. We ran 100 epoch on this model, and it took about two hours. From the trend of the training and validation set, we can tell that if we train with more epoch and for a longer time, we can attain better results.

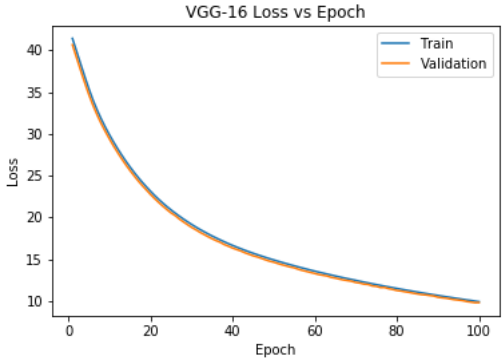

Figure 28: *VGG-16 Loss*

Tis is the loss of training and validation set of VGG.

The final accuracy on the training set is 0.1889, validation set is 0.2128, and on the test set is 0.2088.

5 Comparison Between Standardized Networks

5.1 VGG-16

We tried to implement VGG-16 on both Tensorflow and Keras. However, this model we

implemented gives us an accuracy of only around 20%. The reasons might be the follwing:

- It takes too long to train to get a good result, and we did not have enough time to adjust the parameters for the model, like the combination of the learning rate and optimizers. If we did not wait till the last epoch, we would have no idea what the final accuracy would be. From figure 27, we found that if we train the model with more epoch, the model can still attain good result judging from the trend.
- Lack of pre-trained model. Initially, we first tried to load pre-trained weights from the ImageNet VGG-16 into our model. That is to say, we did not load the dense layers into the model because we have different number of classes. At the very end, we did not acquire a good performance for using the pre-trained model. However, it may be that we did not set up our environment in the beginning.
- Since this is the first model we tried to implement in Keras, we spent lots of time making sure the model works. However, for the next time, it is important for us to start with a relatively simpler model and then implement harder ones.

We tried to look into other papers and found out that in other reports, people use momentum based SGD with momentum of 0.9 with a batch size of 256 and weight decay of 0.0001. The final accuracy reached on the testing dataset is 0.51.

5.2 ResNet

We tried to implement ResNet on both Tensorflow and Keras. However, this model contains too much filters and layers, and we are not able to successfully run it on the server.
We looked into other reports and found out that people normally use average pooling, momentum based SGD with momentum of 0.9 with a weight decay of 0.001 and a batch size of 128. The final accuracy is 0.59.

6 Conclusions

We tried four different models and changed several different variables like numbers of layers, pooling method and activation functions in order to approach the higher accuracy on the testing dataset. From the experiment, we found that Adam and Adamax are easier to train and can get higher accuracy than other optimizers in this situation. From the second experiment, we found that as the number of layers increases, the accuracy tends to increase; ReLu is indeed a better activation function compared to others, and as mentioned in lectures, sigmode did not provide good accuracy in most of the circumstances. Lastly, average pooling does perform better than max pooling in our model. It may not necessarily true for other cases as many models use max pooling.

7 Bonus Point

7.1 Novelty of the Project

From this report, we found out that it takes a lot of time to train the model, modify the parameters, and get the final result, so we hope that from this report, we can give other researchers some inspirations on how to choose a good parameter, in this report, an optimizer. Choosing an good optimizer can not only increases the accuracy of the model, but also help train the model faster, because some optimizers are not very sensitive with other variables in the model.
Normally in the reports and papers, people only tend to provide a solution they discovered, or make improvement on the previous project, like improving the accuracies of models. However, in our report, other than finding the model that gives the highest accuracy on the testing dataset, we also tried to help other researchers **decide how to choose a good parameter – an optimizer, which can reduces the time on training**.

7.2 New Approaches

In order to find the good optimizer to train and increase the accuracy, we also tried to combine two optimizers together. From the model section, the models with the highest accuracy is the 8-layer, average pooling, ReLu model we built ourselves. At first we tried 6 different optimizers on the model, and after that, we also tried to use adam and other optimizers one after another. However, the accuracy does not improve that much. Even the accuracy did not change by a large percentage, it is still a new way to try to combine optimizers together and discover a new way to train the model.
Instead of learning experiences only from other papers and implementing the standard network, we also implemented our own network and got the

relatively high accuracy. We also compare different parameters in order to find the better one with the model.

7.3 Models Attempted

In this project, we tried to implement our own model. This time, we did not only change the value of optimizers, but also number of layers, number of filters, learning rates, activation function and pooling methods. All pre-trained models are already uploaded to Google Drive for future uses. For the standardized CNN models, we have tried VGG, LeNet and Alexnet. However, we encounter some problems when building those models. We tried to train VGG with different parameters. Since VGG is the standardize network, we only tried to change many groups of optimizers and learning rates, but it still gives us lower accuracy than the CNN model we implemented. LeNet is easy to implement, but it seems that LeNet is very easy to converge and result in learning nothing from epoch to epoch. The final accuracy is as low as 0.5% of accuracy on the testing dataset, which is almost the same as the random guess on the 200-class dataset. Alexnet has too many filters, which takes a lot of resources on the server, so we are not able to get an result (Our only result is the "Disk Quota Exceeded" warning).

7.4 Time Spent

We started this project on Saturday before the final's week. However, training different types of standardized network took us too much time, even if we try to run on GPUs (sometimes the model ran as long as 5 hours depending on the complexity of the model we chose). We spent more than 10 hours a day on training the model, finding where the setting could be wrong and getting the result. After we finished our last final on Wednesday night, we tried to train the model more than 16 hours a day until Saturday in order to get comprehensive experiments and result then finalize the report.

7.5 Technical Difficulties

Although we started pretty early on this project, we are consistently demanding for GPUs to use, especially after Wednesday. However, when we are running our programs, sometimes the "delete pod" would show up in the command line very randomly, and we need to re-log back in to retrain our model. In order to solve this problem and save the data we already ran, we decided to save the models along with the parameters and histories so that when next time logging into the account, we can still trace back whatever we were working on without losing any important information and waste our time.

7.6 Video Presentation

We have made a video to give a brief overview of the experiments we have done for this project:
https://drive.google.com/file/d/1ochf0fFUco9xwM9RbHunU2kQ0wi89dAE/view?usp=sharing

References

[Anna Shcherbina2016] tanford CS231n Final Paper. 2016. *Tiny ImageNet Challenge.* Website: http://cs231n.stanford.edu/reports/2016/pdfs/401_Report.pdf

[Andrew Zhai] Stanford 231n Final Paper. 2016. *Going Deeper on the Tiny Imagenet Challenge* Website: http://cs231n.stanford.edu/reports/2016/pdfs/405_Report.pdf

[Jason Ting] Stanford 231n Final Paper. 2016. *Using Convolutional Neural Network for the Tiny ImageNet Challenge* Website: http://cs231n.stanford.edu/reports/2016/pdfs/425_Report.pdf

[CS231n Convolutional Neural Networks for Visual Recognition] Animation that represents performances of different optimizers and training speed. 2016.

[Keras Documentation] Keras: The Python Deep Learning Library. Compatible with python 2.7-3.6. Website: https://keras.io/

[Keras Documentation for Setup Purposes] Getting Started with the Keras Sequential model. Website: https://keras.io/getting-started/sequential-model-guide/

[VGG Pre-trained Model on Original Imagenet Dataset] The github code and explanation on pre-trained model for the large 1000 classes Imagenet dataset. Website: https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3

[GoogLeNet Github] The github code and explanation on GoogLeNet. Website: https://gist.github.com/joelouismarino/a2ede9ab3928f999575423b9887abd14

[Google Drive Models] This is shared folder containing all of the models that we have trained in this report. (Drive)