COMP4107 Project Report

Christian Abbott 100 863 049

Basim Ramadhan 100 901 646

1 The Problem Being Solved

For our project we investigated and compared the performance of the state-of-the-art in CNN's for image recognition with the ImageNet dataset. We decided to use the Tiny ImageNet[1] dataset for our project because the entire ImageNet dataset proved to be too computationally expensive for us when we tried using it in the beginning. This reduced dataset contains 200 classes to recognize, as opposed to the 1000 classes in full ImageNet.

We wanted to learn about different researchers efforts in tackling the ImageNet dataset. We wanted to create our own classifier for recognizing images from the Tiny ImageNet dataset based on what those researchers did. The classifier models we implemented and compared are "reduced" versions of the originals. In other words, we tried to compress them and retain classification accuracy.

Our project involved lots of tuning and experimenting and failing and plenty of troubleshooting. We experimented with different hyperparameters, architectures, optimizers (provided by the TensorFlow library), etc.

2 Experimentation

2.1 General Info & Experimental Setup

We decided to train for top-1 accuracy on Tiny ImageNet and see if we can get some good accuracy. We wanted to eventually test for top-3 and top-5 accuracy values, but we were busy enough trying to create a good model for top-1 accuracy.

The Tiny ImageNet dataset contains $100,000$ images that belong to 200 classes. Each image is 64x64 and in color (three channel RGB).

We setup CUDA GPU acceleration on a GTX 970, which drastically reduced training time for us and gave us the flexibility to test different models extensively. However, even with the reduced-ImageNet that we used, we were wishing for more computation power. Adequately training a model from scratch was taking anywhere between 8 and 24 hours on the GTX 970.

While we didn't achieve top-tier accuracy on Tiny ImageNet, we did manage to do a whole lot of comparisons of various models using different architectures and optimizers, including AlexNet[2], YOLO, Fractional Max-pooling[3], VGG16[4] and SimpleNet[5].

[1] https://tiny-imagenet.herokuapp.com/

[2] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] https://arxiv.org/abs/1412.6071

[4] https://arxiv.org/abs/1409.1556

[5] https://arxiv.org/abs/1608.06037

2.2 Data Pipeline

We downloaded Tiny ImageNet and had to process the data so that we can train fast. The dataset came as 200 directories for the 200 classes containing their respectitve JPEG images. We had to write a program to read those image files and save them as a TFRecords format. Tensorflow has an abundant set of tools for batching and loading data from TFRecords, making it easier for us to work with the data and it sped up the training process dramatically.

We designed and implemented a multithreaded, concurrent batching tool that allowed us to load batches in various sizes extremely quickly from the TFRecords files, which contained our training and validation datasets.

We used a component of the TensorFlow library called TF Slim for our data pipeline and for some model definitions. TF Slim is specifically designed to make it easier and more efficient to deal with large-scale machine learning pipelines, and is the tool of choice for deploying large networks such as VGG16/19 and inception_v3 for distributed machine learning pipelines.

Our data pipeline involved the use of QueueRunners, String_input_producer, and shuffle_queue for concurrent batching. The added speed from concurrent processing ensured that our data loaders would not be a speed bottleneck in our training pipeline.

Our data pipeline was setup to allow us to add data augmentation. More specifically, we wanted to apply horizontal flips, vertical flips, and cropping at random to improve the classifier.

2.3 Hyperparameter Tuning

Successfully getting a network to learn this Tiny ImageNet dataset involved lots of hyperparameter tuning. At first, our models were failing to even begin the learning process due to the huge number of classes in this problem. After tuning, we peaked out at around 33% top-1 accuracy. Considering that we're building a classifier for a 200-class problem, this is a good start.

However after weeks of tuning we were still unable to isolate the cause of this tendency to stall at 33%. It appeared as if every model that we tested was underfitting the data.

We experimented with numerous models, regularizers, optimizers, cost functions, learning rates, batch sizes, etc., but we still could not achieve higher accuracy. We're still unsure about why this is happening. In hindsight, we should have used TensorBoard to investigate what's happening with our model.

2.4 Getting the Network to Learn

Our models wouldn't learn until we experimented with different optimizers, cost functions and learning rates. We ended up using learning decay, with an initial learning rate between 0.02 - 0.1, and a decay rate of 0.04 for every 5000 training steps.

We tested multiple models with 5 different optimizers: Nesterov Momentum, Gradient Descent, Adadelta, RMSProp and Adam. We had the highest accuracy with Nesterov Momentum. Meanwhile, we obtained the highest convergence speed with the Adam optimizer. Every other optimizer we tested fell somewhere in between these two extremes. The Adam optimizer was likely faster to learn because it implicitly tunes its own learning rate according to the parameters in the model. The Momentum optimizer was likely more accurate as it is able to more precisely navigate the valleys of the decision surface to find the optimal local minima.

2.5 Choosing a Model

Originally, we experimented with a model called TinyYOLO, which is a miniature network used for *realtime* object detection. We chose this network because it was small and fast, and had been known to be able to classify ImageNet images with reasonable accuracy. Unfortunately, the small size of this network made it extremely difficult to tune. Tuning and testing on a single GPU ended up taking a prohibitively long amount of time and did not yield ideal results. This experiment would have been more fruitful if we had more computational power and time at our disposal, as we would be able to automate the hyperparameter tuning process.

After experimenting with the TinyYOLO model, we decided to implement the VGG16 model. We successfully managed to obtain a reasonable degree of validation accuracy (roughly 21%). However, training and tuning VGG16 also took a huge amount of time and resources for similar reasons to our tinkering with TinyYOLO.

We decided to compromise and use the AlexNet model because it was deep enough to classify the dataset, but also shallow enough to be trained quickly. Upon implementing AlexNet, we managed to yield 24% accuracy within 4000 training iterations. This speed of training was much faster than we had seen before. At this point, we tested our own custom AlexNet variants: AlexNetLarge and AlexNetXL. Both models had more filters and extra layers of depth, however there was no noticeable increase in accuracy (but a significant increase in training time).

We also experimented with other architectures, such as those based upon SimpleNet and ChrisNet that made use of Ben Graham's Fractional Max Pooling operation. In theory, Fractional Max Pooling allowed us to add depth because it allowed us to downsize our images by a factor of less than 2 (as in max pooling), which meant that we could stack more convolutional and pooling layers before reaching a minimum image size (with max pooling, we can only downsample a maximum of 4 times before we reach an image size of 4x4, at which point we lose spatial information). However, even with the added depth in this network, we saw no gains in accuracy, which suggested that our models were not underfitting, as adding depth to a network should result in less underfitting.

2.6 Increasing the Accuracy of AlexNet

We we able to increase the accuracy of our AlexNet implementation by modifying its architecture and training loop in a number of ways:

1. We added batch normalization to each convolutional layer.
2. We added a layer of dropout between the final convolutional layer and the first fully connected layer.
3. Since we were using ReLU activations in our model, we switched our initializer from truncated-normal initialization to Xavier initialization.
4. We implemented learning rate decay (0.04/5k iterations).
5. We implemented L2-regularization of 0.0005 on each convolutional layer.

3 Our Best Model

Our best-performing model achieved 33% accuracy for top-1 classification.

3.1 Architecture

1. Input
2. Conv2D (96 filters, 5x5 kernel, stride=4)
3. ReLU
4. MaxPool2D (2x2 kernel)
5. Conv2D (256 filters, 3x3 kernel, stride=1)
6. ReLU
7. MaxPool2D (2x2 kernel)
8. Conv2D (256 filters, 3x3 kernel, stride=1)
9. ReLU
10. Conv2D (128 filters, 3x3 kernel, stride=1)
11. ReLU
12. Conv2D (64 filters, 3x3 kernel, stride=1)
13. ReLU
14. MaxPool2D (2x2 kernel)
15. Fully Connected (4096 neurons)
16. Dropout (0.5)
17. Fully Connected (4096 neurons)
18. Fully Connected (200 neurons)
19. Softmax

3.2 About the Model & Hyperparameters

- Implemented in vanilla TensorFlow (the same model in TF Slim had poorer results).
- The model is based on AlexNet, whose use we already discussed above.
- We used the Momentum optimizer with momentum = 0.9
- We used the Softmax cross-entropy loss function.
- We used the Xavier weight initializer.
- We used the ReLU activation function.
- Initial learning rate = 0.01
- Learning rate decay = 0.02 / 5000 training steps.
- L2-regularization = 0.0005 on each convolutional layer.
- Batch size = 128
- Batch normalization on each convolutional layer.

4 Conclusion & Future Work

During our experimentation, we enjoyed using TF Slim because it provides lots of functionality *"for free"*. However, we eventually discovered that using *"Vanilla"* TensorFlow seems to be more robust than TF Slim as it gives us more control over our models.

We found that the Momentum optimizer is by far the best one that we've tested so far. The Adam and Adadelta optimizers also showed promise, but we didn't experiment with them as much. We also found that adding layers is not a miracle antidote for improving classifier accuracy, since adding layers did not get rid of our plataeu at 30% accuracy.

For some reason, despite the fact that we tried numerous models, their accuracy always fell off around 30%. We still don't know understand why this occurs. Adding regularization did not change our results at all, which suggests that we weren't overfitting at all. It's still a mystery to us why we maxed out at around 30% top-1 accuracy. Since we used the Tiny version of ImageNet, we originally aspired to obtain over 60% top-1 accuracy. Despite this, we were satisfied with what we learned in our experimentation.

5 Figures

5.1 Baseline CNN Architecture

1. Conv2D (3x3 kernel)
2. MaxPool2D (2x2 kernel)
3. Fully Connected (1024 neurons)
4. Fully Connected (200 neurons)
5. Softmax

5.2 SimpleNet Architecture

1. Conv2D (2x2 kernel)
2. Conv2D (2x2 kernel)
3. Conv2D (1x1 kernel)
4. Conv2D (2x2 kernel)
5. MaxPool2D (2x2 kernel)
6. Conv2D (2x2 kernel)
7. Conv2D (2x2 kernel)
8. MaxPool2D (2x2 kernel)
9. Conv2D (2x2 kernel)
10. MaxPool2D (2x2 kernel)
11. Conv2D (2x2 kernel)
12. Conv2D (2x2 kernel)
13. MaxPool2D (2x2 kernel)
14. Conv2D (2x2 kernel)
15. Conv2D (1x1 kernel)
16. Conv2D (1x1 kernel)
17. MaxPool2D (2x2 kernel)
18. Conv2D (2x2 kernel)
19. MaxPool2D (2x2 kernel)
20. Conv2D (1x1 kernel)
21. Softmax

5.3 VGG16 Architecture

1. Conv2D (2x2 kernel)
2. Conv2D (2x1 kernel)
3. Conv2D (1x2 kernel)
4. MaxPool2D (2x2 kernel)
5. Conv2D (2x2 kernel)
6. Conv2D (2x2 kernel)
7. Conv2D (2x2 kernel)
8. MaxPool2D (2x2 kernel)
9. Conv2D (2x2 kernel)
10. Conv2D (2x1 kernel)
11. Conv2D (1x2 kernel)
12. MaxPool2D (2x2 kernel)
13. Fully Connected (200 neurons)
14. Softmax

5.4 Accuracy of our AlexNet Model Implementation

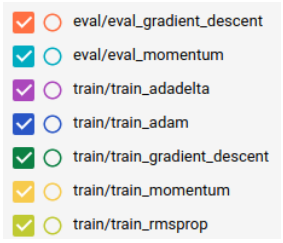

Figure 1: Legend for the Charts That Follow

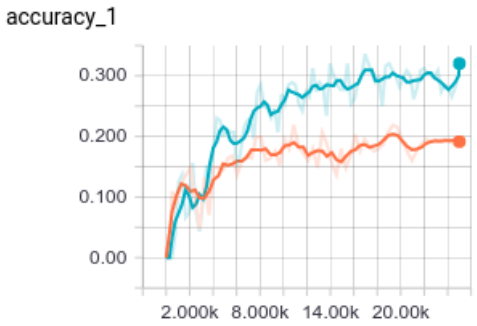

Figure 2: **AlexNet Accuracy:** Gradient Descent Optimizer vs. Momentum Optimizer

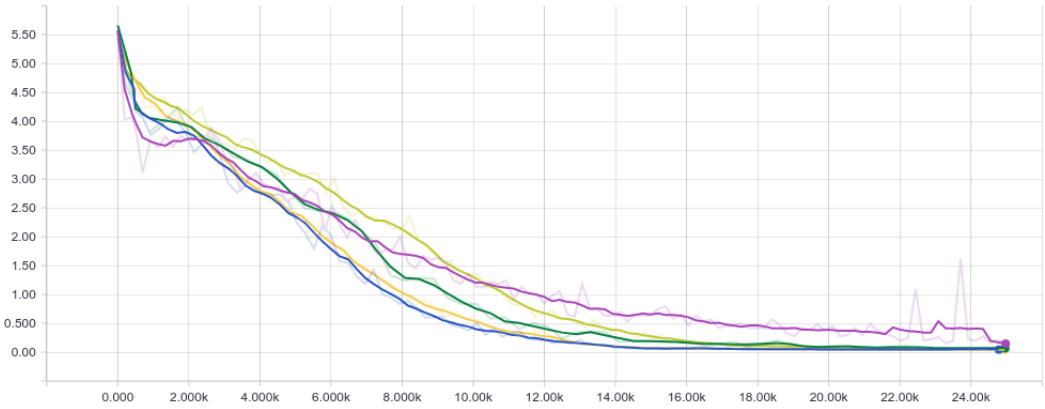

Figure 3: **AlexNet Loss:** Gradient Descent vs. Momentum vs. Adam vs. Adadelta vs. RMSProp Optimizers

5.5 Computational Graph

We used TensorBoard to visualize our final model. Unfortunately, the graph wasn't as neatly organized as we would have liked. We hope the following snippets from the computational graph are useful:

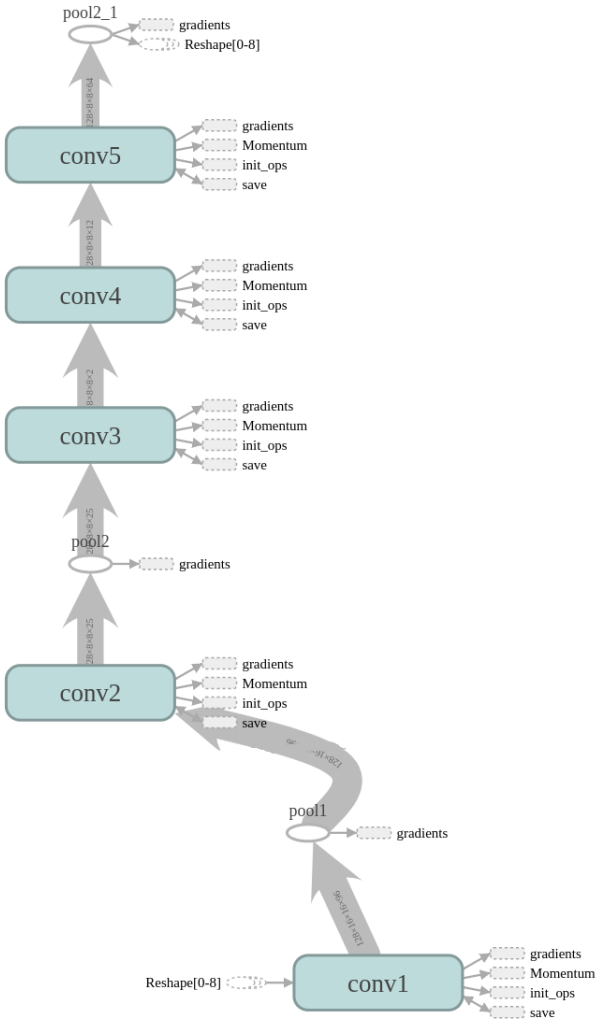

Figure 4: Part 1 of the Computational Graph: The Convolutional Layers

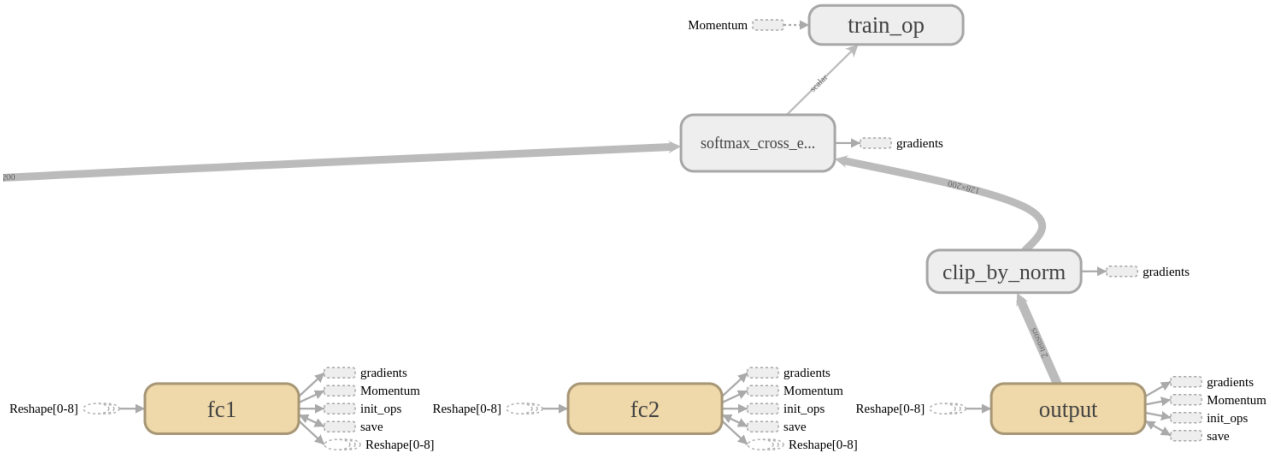

Figure 5: Part 2 of the Computational Graph: The Fully Connected Layers