Programming Assignment 1

Kobee Raveendran

February 1, 2020

1 Parameter Selection

Many of the parameters I've chosen to use for each of the custom networks are simply based on typical values used in popular networks, or on previous networks that I've used in the past with good performances. Much of it could be simplified to a combination of trial and error, and somewhat standardized values used in the literature. Yet other parameters are chosen for their computational or memory efficiency in my training setup. I trained on a graphics card with only 8 GB of VRAM, so I was heavily bottlenecked by memory when training the larger models.

1.1 Convolution Kernels

For number of kernels per convolution layer, I used solely powers of 2, starting with 64 kernels, then increasing by factors of 2 until a mid-point in the network, and finally descending back down to 64. I used this inverted autoencoder-like structure for two reasons. Primarily, this was to avoid having extremely large numbers of parameters caused by monotonically-increasing kernel sizes, which allowed me to expedite training. Additionally, I found that for the models with more convolution layers, steadily increasing the number of kernels posed a problem for me due to heavier resource consumption (primarily memory-wise; once tested on 6- and 9-layer models, I consistently received CUDA memory allocation errors). Additionally, past 3 layers, I did not see much of a difference in performance accuracy-wise between the two configurations, so I opted to save on memory.

Additionally, keeping with convention, I opted to make each kernel be 3x3 in size. Keeping kernel sizes small allows the network to avoid the computational costs of larger kernel sizes since there are less weights to learn, but trades this for more layers, which necessitate extra memory used by the extra layers. However, this is not necessarily detrimental, as using more layers lets the network learn more complex features. Additionally, using odd-number dimensions allows each central pixel to be surrounded by the previous layer's output pixels, giving the network good information on neighboring pixels while also avoiding the awkward case of dealing with a non-existent central pixel in an even-numbered kernel size.

1.2 Stride, Padding and Activation Functions

For all convolution layers, I opted to keep the stride at the default of 1. Using a stride greater than 1 leads to downsampling of the input, which became a problem for me when using layers with more kernels (tensor dimension reduction errors). So instead, using a stride of 1 only results in depth-wise transformation instead of width- and height-wise. These are later handled instead by the pooling layers, which use a stride of 2 to downsample the input by a factor of 2 width- and height-wise (only 1/4 of the activations in the 2x2 grid are chosen, in my case the maximum).

As mentioned earlier, I wanted to avoid dimension problems as models got deeper. Part of the source of the problem was the reduction of dimensions caused by the pooling layers. To remedy this, I used zero-padding after pooling was done to keep the tensors around the same size as they flowed through the network.

Staying consistent with convention, I used the ReLU activation function for all layers excluding the output. ReLU is 0 when the activation is below 0 and equal to the input when above 0. ReLU is used instead of other activation functions for several reasons. Purely linear activation functions prevent the model from learning complex features, so they are out of the question for a task such as object classification, especially with this magnitude of classes and variability. Sigmoid and hyperbolic tangent, the two most popular activation functions, have a problem in that the input, regardless of how large it becomes, is always constrained between 0 and 1 (or -1 and 1 for tanh). This means that the two functions are only majorly influenced by inputs near the middle of their ranges, and very large or very small inputs have little effect. Later changes in weights will change activations very little, which can hamper training.

With the final output layers of each model, I opted for a softmax activation since the layer would need to output a probability corresponding to a given class. The maximum probability from this output tensor would be considered the model's prediction for a given training or test sample.

1.3 Loss Function and Optimizer

Since the task at hand is object classification into only **one** of n categories, I used the categorical cross-entropy loss. This loss function mandated a one-hot encoded vector for class labels (as opposed to sparse categorical cross entropy).

The optimizer I used for the custom models was the Adam optimizer, primarily because it is commonly-used and robust, as well as several other factors. It is highly performant on large data sets, is not memory-intensive, and is inexpensive computationally, all factors that are important to my training setup. Although SGD would be better overall for performance in that it can converge to flatter optima more than adaptive methods like Adam do, I opted for Adam due to its computation benefits. While SGD tends to generalize better than Adam due to the aforementioned optima it can find, Adam converges faster. Though there are better, newer optimizers like AdaBound that can do a good

job of both (convergence speed and generalization), it is not currently officially implemented into many frameworks, including Tensorflow and the Keras API.

2 Network Architectures

NOTE: I HAVE PROVIDED MORE DETAILED MODEL ARCHITECTURES THAN DESCRIBED IN THIS SECTION BELOW, IN THE APPENDIX (SECTION 4). INCLUDED ARE INPUT AND OUTPUT SHAPES, NUMBERS OF PARAMETERS, AND THE FULL SEQUENCE OF LAYERS USED.

2.1 Custom Networks

For the three custom networks implemented, I used repetitions of a single module of layers, only changing the number of modules and the kernel sizes of each (described in section 1). Each module consisted of a convolution layer (with 3x3 kernels), followed by a max pooling layer (with receptive field of size 2x2), followed by batch normalization and zero padding layers. Batch normalization was primarily used because of the increase in top-1 classification accuracy observed after including it. These modules are chained together, and the final module's output is fed to a series of fully-connected layers. The custom models all had two fully-connected layers, the final of which had an output size of 200 since there are 200 classes in the Tiny Imagenet dataset. This output would be a vector with dimensions (200, 1), where each element is the predicted probability of the corresponding class. The final predicted class would then be the index corresponding to the element with the maximum value.

2.2 Finetuned model: VGG16

VGG16's architecture is somewhat similar in that it also consists of repeated convolution modules. However, the structure of each module differs greatly. Modules appearing early in the model consist of two consecutive convolution layers followed by a max pooling layer. Later on in the model, the modules consist of three consecutive convolution layers instead. Finally, the output of the last module is flattened and fed to two fully-connected layers with 4,096 output units, and a final fully-connected layer for predictions containing 1,000 output units.

However, when finetuning VGG16, I excluded the original fully-connected layers since the model must now predict among 10 classes (in SVHN), not 1,000. Instead, I appended a global average pooling layer, a fully-connected layer with 512 output units, a dropout layer, and the final fully-connected layer with 10 output units for predictions. Additionally, since the SVHN dataset's images have different dimensions than Imagenet's, the bottom layers for input also had to be changed. Since adjusting to an input size of (64, 64, 3) was too small (I ran into problems with pooling layers downsampling so much that no more downsampling was possible), I was forced to resize the SVHN images to (128,

128, 3), then using this as the input to the model. The original accepted an input size of (224, 224, 3), and though I could have resized to this instead, I wanted to avoid bloating the images more than necessary to pass through the model.

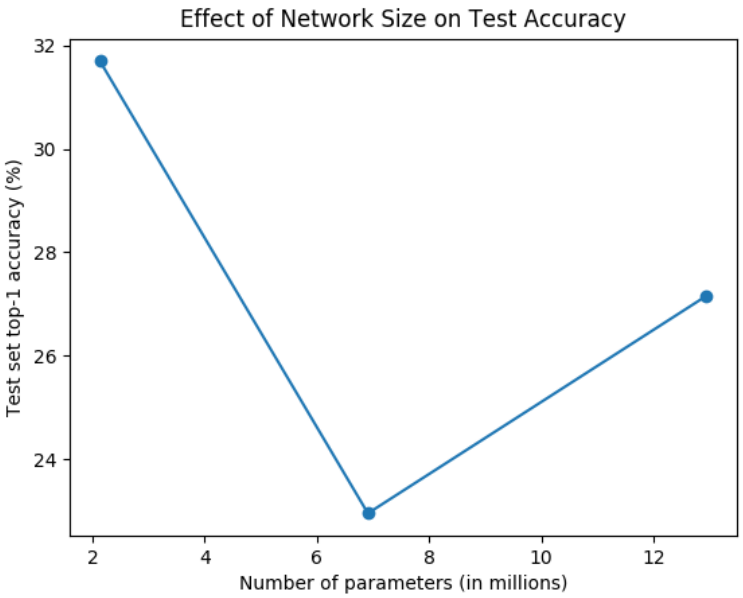
3 Experiments

Several factors were studied to gauge their effects on model classification accuracy and training time. Those factors included model size (number of convolution layers) and size of the data set. All experiments were carried out on a GeForce RTX 2070 8GB graphics card, and results collected are the averages of experiment outcomes over 3 trials.

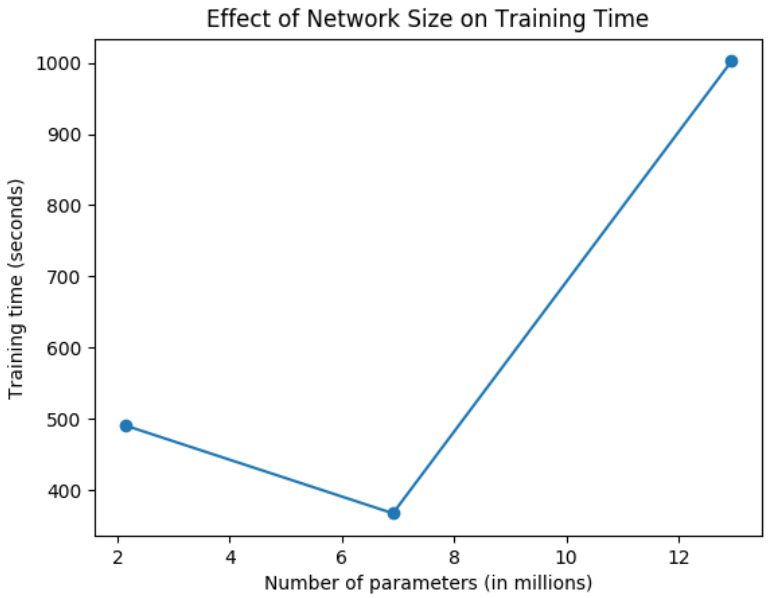
3.1 Effects of Model Size on Accuracy and Training Time

My custom models had three variations of model size (measured by number of convolution blocks present). The sizes were 3, 6, and 9 convolution blocks. However, the number of convolution blocks did not always correspond to the number of parameters each had. In my 6 conv-block model, there were only approximately 2 million parameters, which is less than both the 3 conv-block (7 million parameters) and 9 conv-block (12 million parameters) models. I believe this is entirely due to an architectural choice I had to make since this model had an even number of convolution blocks. The layers in the middle (conv_3 and conv_4) had equal numbers of kernels, thus breaking the trend of my otherwise inverted-autoencoder-like structures in the odd-numbered variants. I surprisingly found that VGG16 follows a similar pattern with its convolution blocks, though within a single block (multiple conv layers are chained, each with the same number of kernels). As a result, VGG16 only has about 14 million parameters despite being nearly twice as large as my conv9 model, which has 12 million parameters.

So, with this in mind, the plots below may seem confusing at first, but only because the ordering of the models is swapped (conv3 and conv6 are swapped, since conv6 has less parameters). Conv6, despite having less trainable parameters, outperforms the other two models, probably because of its unique balance of number of layers (allowing it to learn varied features with different levels of complexity), while also not having so many parameters as to heavily overfit as much as the other models did. Of course, upon reaching 9 conv blocks, test set accuracy dips again because the model has become too complex and heavily overfits. See the plot below for the full view.

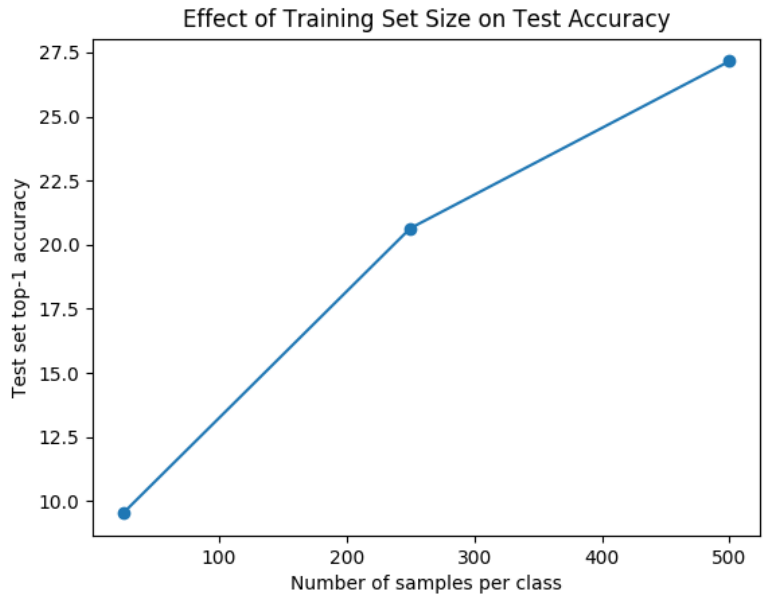


With regard to training time, a more expected result shows (when considering number of *layers*). It is important to note this because, as can be seen in the plots below, the number of parameters alone does not directly correlate to training time. While conv6 had three times less trainable parameters than conv3, it had twice as many layers, and thus took a similar factor more time to train. This trend continues with conv9.

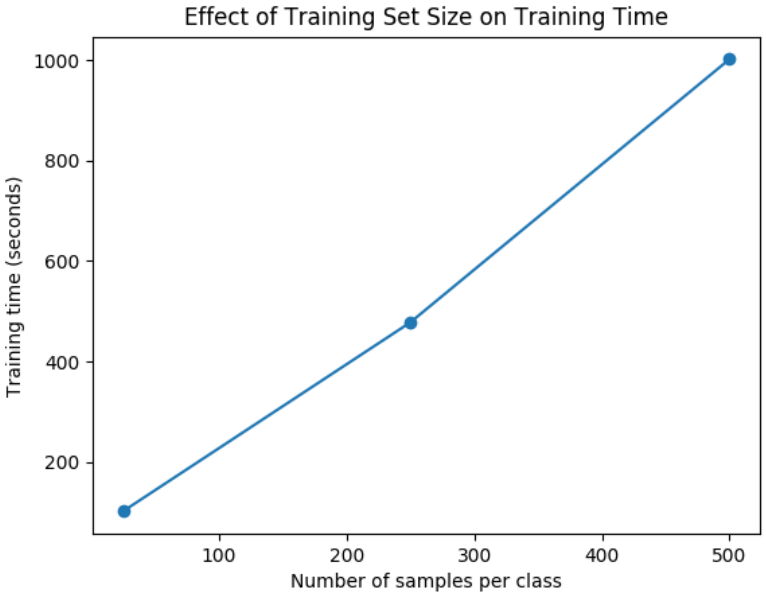


3.2 Effects of Training Data Size on Accuracy and Training Time

The degree to which training data availability affected model classification performance was surprising. Excluding as little as 50 data samples per class from the training set had a very noticeable impact on test set accuracy. In my tests, however, to show a clearer relationship, I chose values of `samples per class` to be 50 (the minimum), 250, and 500 (the maximum). As was expected, training on very little data led to terrible validation and test set accuracy. As models get larger, they demand more training data, so it is only expected for my largest model, conv9, to perform so terribly with little data. With little data, larger networks cannot determine meaningful patterns and features throughout the layers, so they fail to generalize well. Because of this, with more data, performance does increase with respect to accuracy. See the plot below for the trends.

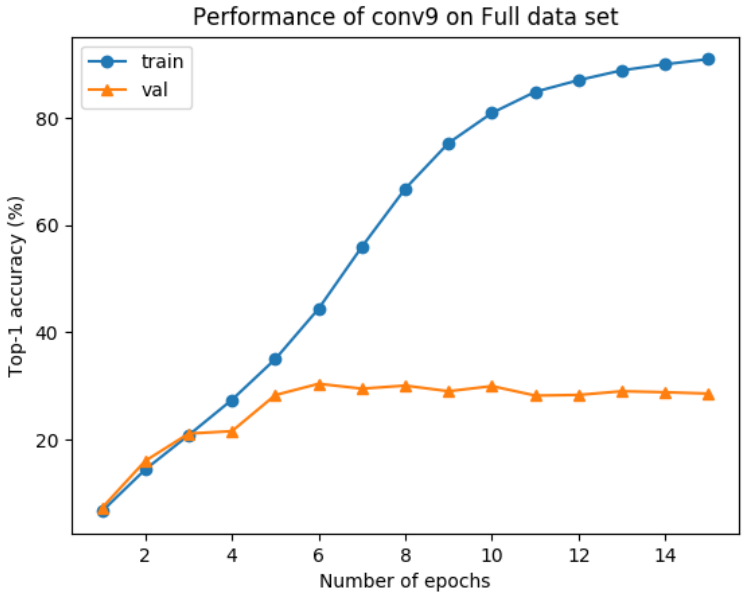


While increasing training set size has been now shown to lead to better classification accuracy, it also quite linearly increases training time. In fact, just as 500 samples is ten times greater than 50 samples in magnitude, conv9 trained ten times slower on 500 samples than on 50. See the exact trend in the plot below.

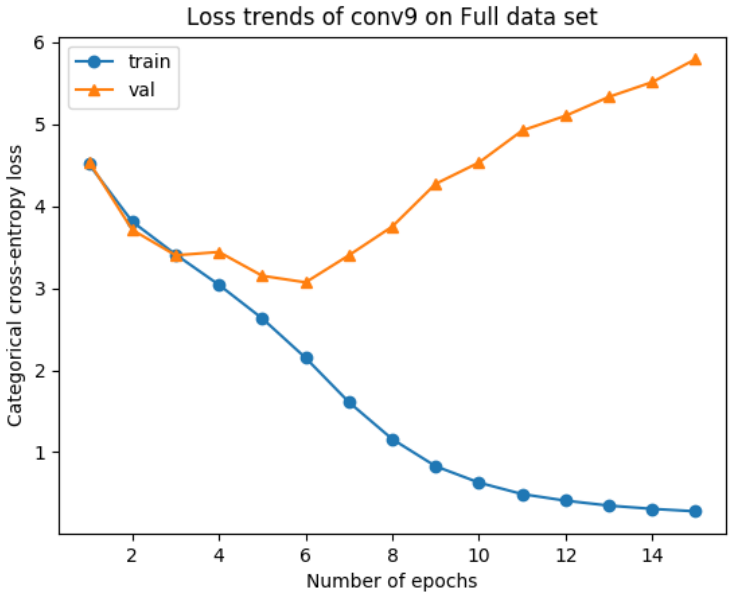


3.3 Conv9 Performance Analysis

Initially when writing the code for my models, I expected conv9 to perform the best simply because I believed more layers would lead to better capability to understand features of all levels of complexity. I did not, however, anticipate conv9 to have worse performance than smaller networks. Once I trained the model though, I saw why this occurred. Examine the below plots to better understand what happened with a larger network.



As can be seen, when trained for few epochs (approximately less than 3), the model keeps up its validation accuracy with the training set accuracy, which is surprisingly good. However, after being trained for more epochs, it rapidly diverges from the training set accuracy and it is clear that by the end of training, or even at 5 epochs, the model has rampant overfitting, and can no longer generalize to unseen data (even on validation data, which it is tested on once per epoch). This suspicion is confirmed when comparing the training and validation losses of conv9 below. It is apparent that the loss is steadily decreasing, on pace with the training set, until a certain epoch, after which training loss continues to monotonically decrease while validation loss quickly increases.



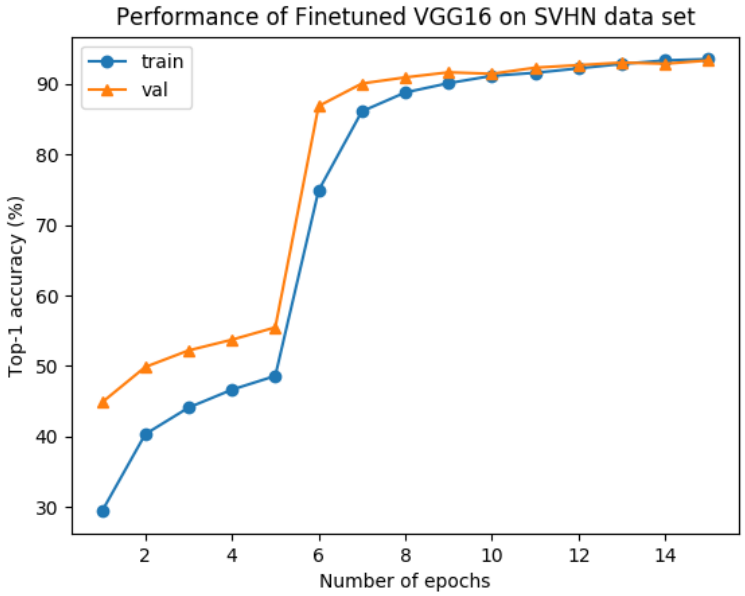
3.4 Finetuned VGG16 Performance Analysis

VGG16 performed much better than I expected, given that it was applie to a completely different data set with even more difference in classes; this showcased the extreme power of transfer learning and finetuning which I was unaware of before. The pretrained VGG16 model I used had weights from being trained on the Imagenet dataset, which consists of 1,000 classes and contains over 1 million images total. Meanwhile, the SVHN data set is composed of approximately 100,000 training set images of cropped numbers seen in the real world. It is trivially noted that the two data sets vary wildly in both size and class properties. So intially, while I knew VGG would perform better than my custom models even on a different data set while using Imagenet weights, I could not have expected it to consistently reach 94% top-1 accuracy, only 5% away from SVHN's state-of-the-art models.

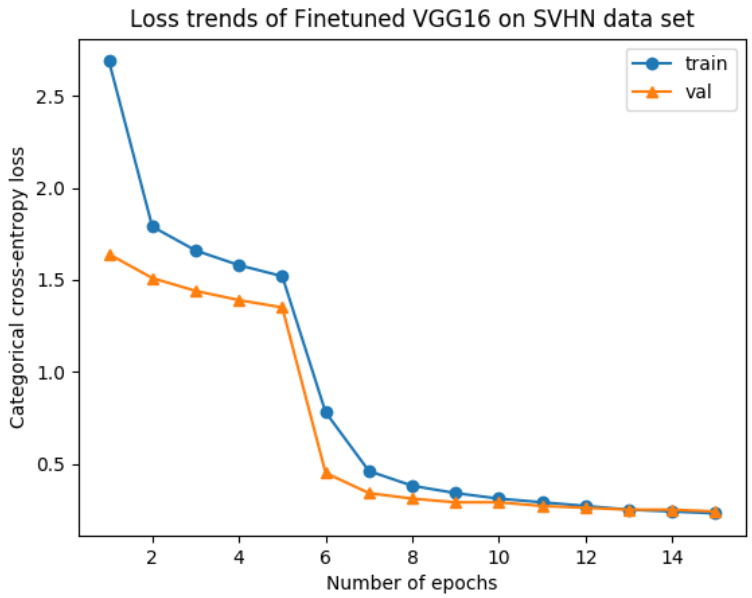
The performance boost came from the methods of finetuning. First, the upper (end) layers of the original VGG16 model are removed and replaced with my custom fully-connected layers (with global average pooling), in order to adjust for the change in output dimensions (due to the need of predicting one of 10 classes instead of 1,000). Additionally, in order for the network to even accept the new input images, which were of different sizes than what VGG expected for Imagenet (32x32 rather than the 224x224 that VGG accepted), some resizing was done to grow the images to 128x128, while changing the input layer to VGG to accept 128x128. Changing the input to 64x64 made me encounter dimension subtraction problems as the inputs to each layer became too small, which is why

I chose to adjust to the middle ground.

Once this was set up, ***only*** the new top layers of the network were trained (by freezing VGG's layers) in order to re-initialize them (they were initially randomly initialized) to adjust to the weights of VGG. This was done for 5 epochs, after which the VGG layers were unfrozen. Now, I chose to freeze all VGG conv blocks excluding the last two layers. This was done because I would still need the network to learn the low level features (such as basic edges) in the earlier layers, which thus did not need tweaking. However, the final few layers would need to be adjusted because the high-level features of numbers and the high-level features of the 1,000 objects differ greatly. So, freezing all but the last few layers, I re-trained the model for an additional 10 epochs. This served as the final model upon which predictions were made. Below are the exceptional results.



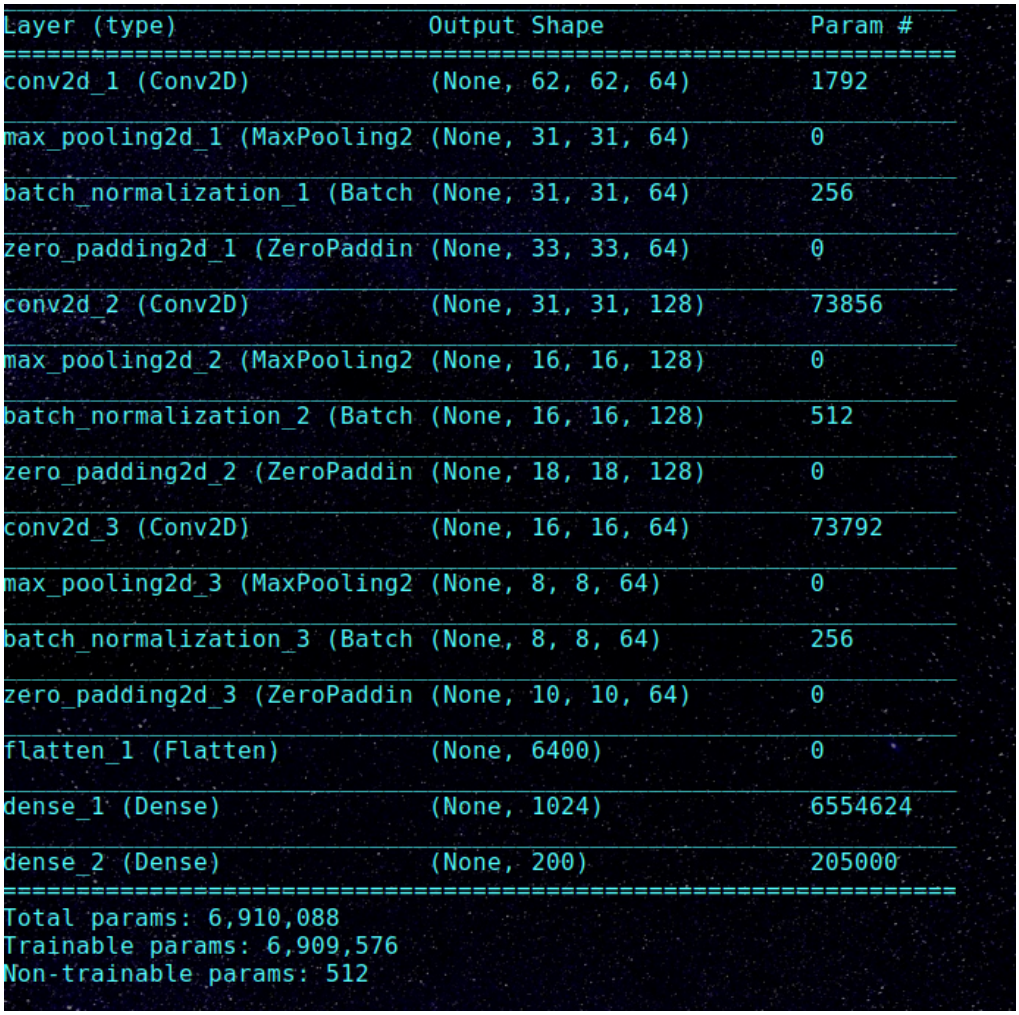
As can be seen above, surprinsgly VGG did *better* when classifying on the validation set than the training set, even early in training. This is something I rarely see, so it deservedly surprised me. Additionally, unlike my custom models trained on Tiny Imagenet, VGG16 monotonically improves its validation accuracy, only reaching an asymptote when nearing 100% top-1 accuracy.



To confirm our earlier observation, we even see here that validation loss is constantly less than training loss. I am still doubtful of why this happens, as usually validation loss and accuracy are meant to *catch up* to the training loss and accuracy, but I am nevertheless impressed. Additionally, I feel it should be noted that in both plots there is a stark jump a third of the way through training. This is because I have included the losses and accuracy measurements from both portions of training; when re-initializing the output layers and when training the newly-joined model. I believe that, since the output layers were then adjusted to the pre-existing weights, there wa s large jump when training the model in full because those output layers had "prior experience," and as such we do not see a low accuracy early in training for the second phase. I believe it would be analogous to implanting in a human baby the knowledge of the alphabet and basic words; it would then be able to learn to read and write at a much earlier age than normal children. And this seems to be half of what finetuning does, with the other half being transfer learning, in which we transfer between problem domains using a single model.

4 Appendix

4.1 conv3 Architecture



Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 62, 62, 64)	1792
max_pooling2d_1 (MaxPooling2	(None, 31, 31, 64)	0
batch_normalization_1 (Batch	(None, 31, 31, 64)	256
zero_padding2d_1 (ZeroPaddin	(None, 33, 33, 64)	0
conv2d_2 (Conv2D)	(None, 31, 31, 128)	73856
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 128)	0
batch_normalization_2 (Batch	(None, 16, 16, 128)	512
zero_padding2d_2 (ZeroPaddin	(None, 18, 18, 128)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	73792
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 64)	0
batch_normalization_3 (Batch	(None, 8, 8, 64)	256
zero_padding2d_3 (ZeroPaddin	(None, 10, 10, 64)	0
flatten_1 (Flatten)	(None, 6400)	0
dense_1 (Dense)	(None, 1024)	6554624
dense_2 (Dense)	(None, 200)	205000

Total params: 6,910,088
Trainable params: 6,909,576
Non-trainable params: 512

4.2 conv6 Architecture

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 64)        1792
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 64)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 31, 31, 64)        256
_________________________________________________________________
zero_padding2d_1 (ZeroPaddin (None, 33, 33, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 31, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
zero_padding2d_2 (ZeroPaddin (None, 18, 18, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 256)       295168
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 256)         0
_________________________________________________________________
batch_normalization_3 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
zero_padding2d_3 (ZeroPaddin (None, 10, 10, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 256)         590080
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 256)         0
_________________________________________________________________
batch_normalization_4 (Batch (None, 4, 4, 256)         1024
_________________________________________________________________
zero_padding2d_4 (ZeroPaddin (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 128)         295040
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 2, 2, 128)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 2, 2, 128)         512
_________________________________________________________________
zero_padding2d_5 (ZeroPaddin (None, 4, 4, 128)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 2, 2, 64)          73792
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 1, 1, 64)          0
_________________________________________________________________
batch_normalization_6 (Batch (None, 1, 1, 64)          256
_________________________________________________________________
zero_padding2d_6 (ZeroPaddin (None, 3, 3, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              590848
_________________________________________________________________
dense_2 (Dense)              (None, 200)               205000
=================================================================
Total params: 2,129,160
Trainable params: 2,127,368
Non-trainable params: 1,792
_________________________________________________________________
```

4.3 conv9 Architecture

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 62, 62, 64)	1792
max_pooling2d_1 (MaxPooling2	(None, 31, 31, 64)	0
batch_normalization_1 (Batch	(None, 31, 31, 64)	256
zero_padding2d_1 (ZeroPaddin	(None, 33, 33, 64)	0
conv2d_2 (Conv2D)	(None, 31, 31, 128)	73856
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 128)	0
batch_normalization_2 (Batch	(None, 16, 16, 128)	512
zero_padding2d_2 (ZeroPaddin	(None, 18, 18, 128)	0
conv2d_3 (Conv2D)	(None, 16, 16, 256)	295168
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 256)	0
batch_normalization_3 (Batch	(None, 8, 8, 256)	1024
zero_padding2d_3 (ZeroPaddin	(None, 10, 10, 256)	0
conv2d_4 (Conv2D)	(None, 8, 8, 512)	1180160
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 512)	0
batch_normalization_4 (Batch	(None, 4, 4, 512)	2048
zero_padding2d_4 (ZeroPaddin	(None, 6, 6, 512)	0
conv2d_5 (Conv2D)	(None, 4, 4, 1024)	4719616
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 1024)	0
batch_normalization_5 (Batch	(None, 2, 2, 1024)	4096
zero_padding2d_5 (ZeroPaddin	(None, 4, 4, 1024)	0
conv2d_6 (Conv2D)	(None, 2, 2, 512)	4719104
max_pooling2d_6 (MaxPooling2	(None, 1, 1, 512)	0
batch_normalization_6 (Batch	(None, 1, 1, 512)	2048
zero_padding2d_6 (ZeroPaddin	(None, 3, 3, 512)	0
conv2d_7 (Conv2D)	(None, 1, 1, 256)	1179904
max_pooling2d_7 (MaxPooling2	(None, 1, 1, 256)	0
batch_normalization_7 (Batch	(None, 1, 1, 256)	1024
zero_padding2d_7 (ZeroPaddin	(None, 3, 3, 256)	0
conv2d_8 (Conv2D)	(None, 1, 1, 128)	295040
max_pooling2d_8 (MaxPooling2	(None, 1, 1, 128)	0
batch_normalization_8 (Batch	(None, 1, 1, 128)	512
zero_padding2d_8 (ZeroPaddin	(None, 3, 3, 128)	0
conv2d_9 (Conv2D)	(None, 1, 1, 64)	73792
max_pooling2d_9 (MaxPooling2	(None, 1, 1, 64)	0
batch_normalization_9 (Batch	(None, 1, 1, 64)	256
zero_padding2d_9 (ZeroPaddin	(None, 3, 3, 64)	0
flatten_1 (Flatten)	(None, 576)	0
dense_1 (Dense)	(None, 512)	295424
dense_2 (Dense)	(None, 200)	102600

Total params: 12,948,232
Trainable params: 12,942,344
Non-trainable params: 5,888