Tiny ImageNet Image Classification

Alexei Bastidas
Stanford University
alexeib@stanford.edu

Abstract

In this work, I investigate how fine-tuning and adapting existing models, namely InceptionV3[7] and VGGNet[3], for benchmarking on the Tiny ImageNet Challenge. The top performing model was inspired by VGG architecture, leveraging Batch Normalization [9] and L-2 regularization to avoid over-fitting. Network performance was analyzed by evaluating validation loss and accuracy prior to running on test set. The work concludes with a discussion of how the modified dataset inspired modifications to the architectures.

1. Introduction

Image classification is a core task within Computer Vision that continues to be improved upon. The challenge in image classification is extracting quantifiable features from a 3 channel, for each color red green and blue, pixel matrix – in the past algorithms were written to manually detect edges and other shapes in efforts to attempt to extrapolate features that could be used for classification. However, these solutions were not scalable.

Starting in 2012, Convolutional Neural Networks (CNNs) began dominating the image classification space – and while deep learning with CNNs has yielded amazing results, we continue to look for more computationally efficient, more accurate, and more descriptive models to apply to this task. As more novel Convolutional Neural Network (CNN) architectures are designed, the current industry standard is for them to be benchmarked through the ImageNet Challenge[1].

In particular, in recent years there has been a shift from the traditional style of stacking convolutional layers, pooling, and activations such as the original AlexNet[2], to sophisticated architectures such that are not only deeper but also leverage unique topologies to achieve remarkable results such as VGGNet[3] and GoogleNet[4].

This project is a combination of experiments with varying architectures such as the afore mentioned to attempt winning this year's Tiny ImageNet Challenge – a smaller version of the ILSVRC with input images of 64x64 pixels, and only 200 possible classes. As in the original ImageNet challenge, the primary goal is to minimize classification error across a large range of distinct image classes.

Given the differences in data between the original ImageNet dataset and the modified Tiny ImageNet, I am drawing inspiration from top performing academic models, but re-implementing from scratch to explore varying architectures and network depth. By extension, no pre-trained weights are ever utilized.

Similarly, I explore varying means of regularization in order to reduce the overfitting problem – since the dataset is significantly smaller than the original ImageNet, a network is more likely to overfit it. In particular, I experiment with varying degrees of Dropout[10] as well as Batch Normalization[9].

I experiment with implementing Google's Inceptionv3 [7] layers as well as varying VGG inspired architectures to different depths, ultimately settling on a VGG style model to run on the test set. I discuss why this model was chosen and visualize different performance metrics used to compare the models and determine an optimal candidate.

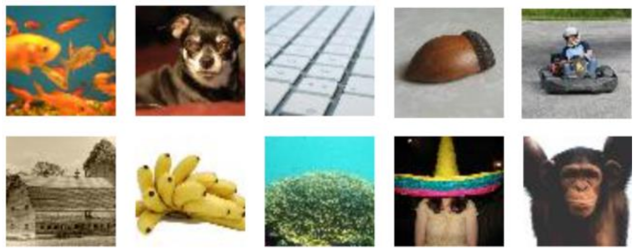
2. Dataset and Implementation

2.1. Dataset

The dataset used for my experiments is the Tiny ImageNet dataset. It contains a training set of 100,000 images, a validation set of 10,000 images, and a test set of also 10,000 images. These images are sourced from 200 different classes of objects. The images are downscaled from the original ImageNet's dataset size of 256x256 to 64x64.

2.2. Data Examples

Image examples from dataset are shown below:



2.3. Pre-Processing and Data Augmentation

Prior to training, the training, validation, and test data were zero centered by subtracting the mean image from the training data – this pre-processing step, as well as the data loading, was implemented by re-using code given by CS231N instructors during assignments[11].

Given the small size of the training dataset, live data augmentation was applied during training. In particular, images were randomly rotated by up to 60 degrees, zoomed in up to 1.2x magnification, and shifted vertically and horizontally.

In order to perform the data augmentation, Keras' Image Data Generator implementation was used[6].

3. Methods

To implement my models I am using version r1.2 of Google's TensorFlow[6] open source deep learning framework, stacked with the included higher level API, Keras[7]. Keras offers a functional API that allows for faster prototyping as well as creation of wide layers such as Inception with significantly less overhead than vanilla TensorFlow.

3.1. Objective Function

As is to be expected, the all the models trained leverage back-propagation to perform gradient updates. The updates were done by minimizing the cross-entropy loss as given by the Softmax function.

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\frac{1}{2}\lambda\sum_k\sum_l W_{k,l}^2}_{\text{regularization loss}} \qquad L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

[12]. This is of course the standard in the field, but it is worth noting that cross-entropy is preferable to other losses such as the SVM or hinge loss as cross-entropy provides a probabilistic interpretation.

Note that regularization and bias terms were added to each convolutional layer. In particular, L-2 regularization was utilized after early trials showed it outperformed L-1.

3.2. Weight Initialization

The means of initializing all weights for every layer of each model was the Glorot Uniform Initializer, also called the Xavier Uniform Initializer[13], as implemented by Keras. Namely, the weights were drawn from the following distribution, with n being the layer size. [13]

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

3.3. Optimization Algorithm

Per guideline presented by Justin Johnson in CS231N lecture 7, the primary optimization algorithm used was Adam[14], as implemented by Keras. Early trials tried using stochastic gradient descent, and Adam + Nesterov Momentum[8], all as implemented by Keras, but ultimately, empirical results showed Adam to be superior in my trials.

3.4. Network Regularization

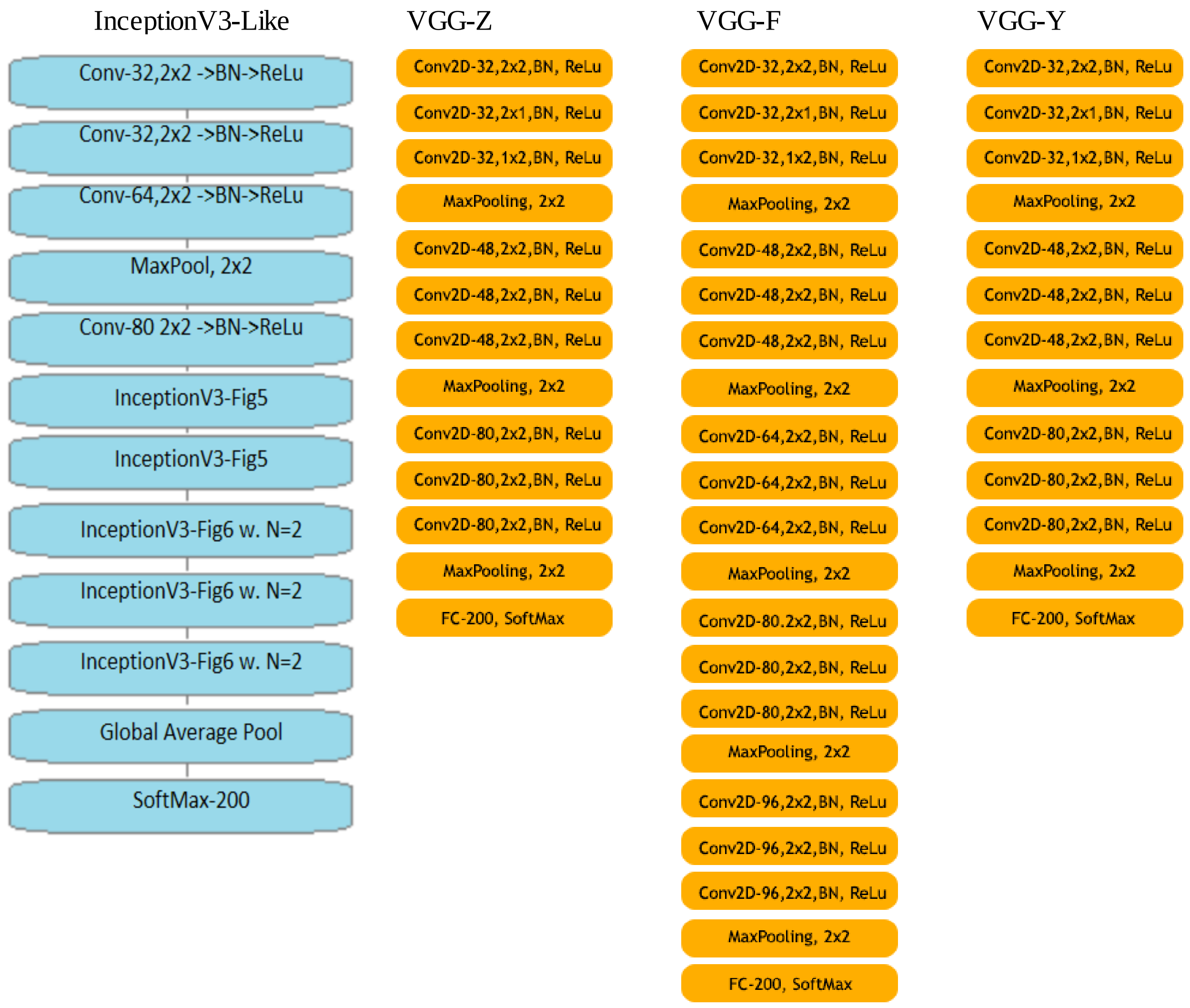
Early on, I experimented with incorporating Dropout [10] with varying rates, and at different points in the network – after each conv layer, before just the input, after pools, and after batch norm layers. However, ultimately I found Batch Normalization [9] to provide much better validation performance for the models that were trained. The idea behind Batch Normalization being that by adding multiple normalizing layers through the network, we can reduce the internal covariate shift from layer to layer within the network.

Of an interesting note, I found that, contrary to common wisdom, utilizing Dropout led to significantly faster training than Batch Normalization, at a cost in validation loss and accuracy – I found Dropout to overfit significantly more than Batch Normalization.

3.5. VGG Style Model Architectures

The VGG Style architecture features a structure

Final Architectures



wherein we stack activated convolutional layers along with a couple max pooling layers, to reduce the spatial dimension of the data, to varying depths prior to flattening the data and running it through a Softmax classifier. As mentioned previously, I also opted to include Batch Normalization throughout all my networks as a means of preventing overfitting.

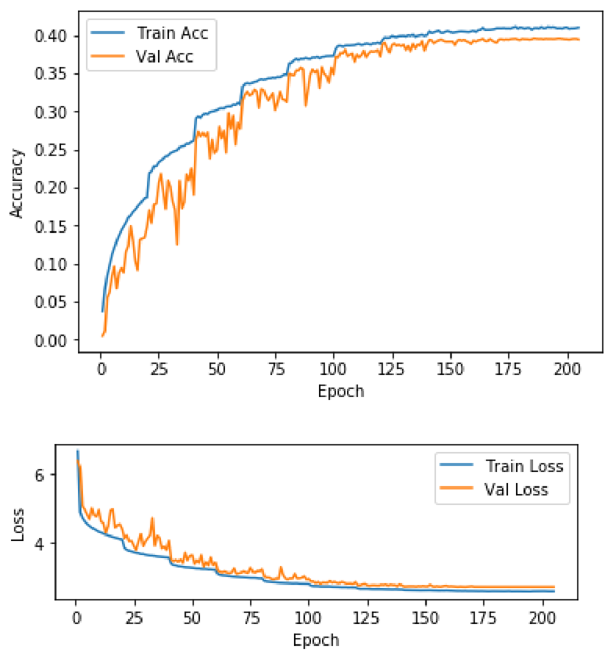
The original VGGNet [3] architectures call for filter sizes of 3x3 – being stacked to generate effective receptive fields of 5x5 and 7x7. Early trials with filter sizes of these dimensions led to relatively fast training, 200-300 seconds per epoch, however, the models would rather quickly, within 20 epochs, begin overfitting.

I made the hypothesis was that due to the reduced input size of the images in the Tiny ImageNet dataset, such large receptive fields were in fact looking at too large a slice of the image, and as such reduced the filter sizes to 1x1, 2x1,1x2, and 2x2 – stacking them at times to generate effective receptive fields of up to 3x3. Furthermore, I opted to remove the multiple final fully connected layers, and instead chose to use a single Global Average Pooling layer.

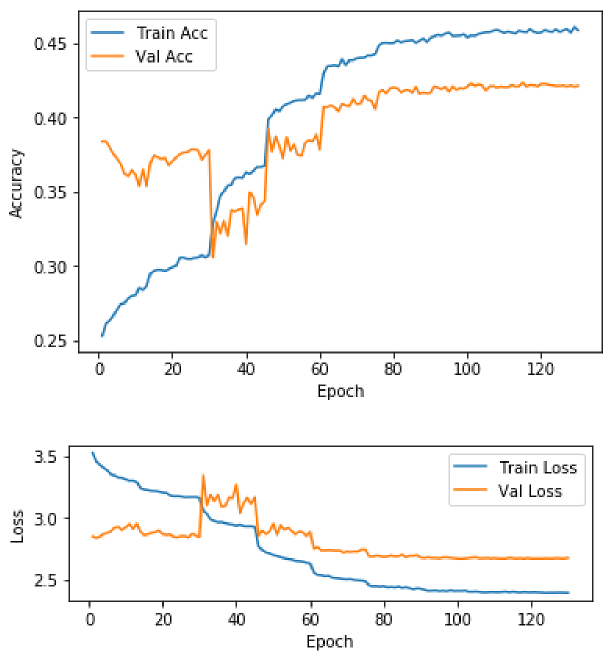
3.6. InceptionV3 Style Model Architectures

The proposed InceptionV3 architecture [7] provides three new wide Inception modules to stack within a CNN. The benefits of InceptionV3 are significantly fewer training parameters, new branching connections for gradients to flow through, and improved performance. The Inception modules are branching layers wherein the same input is ran through four different paths before ultimately being concatenated at the end. This adds a degree of regularization to the network as the inputs are ran, in parallel, through multiple layers – thereby reducing the likelihood of dead neurons affecting the overall

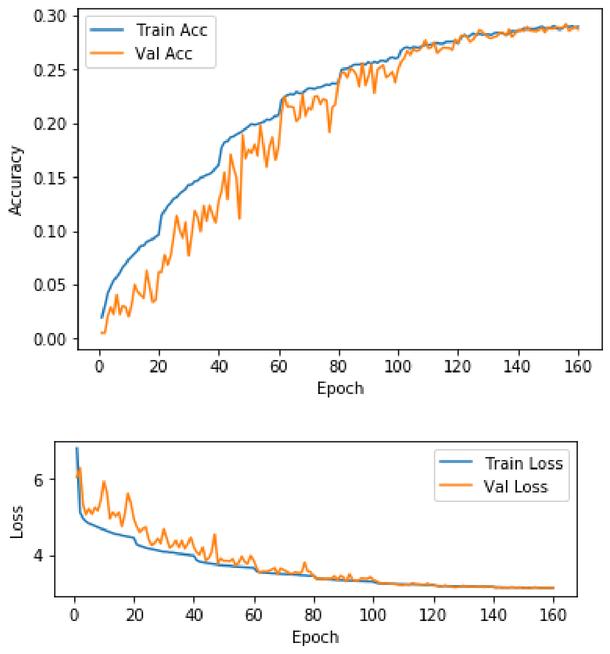
VGG-Z



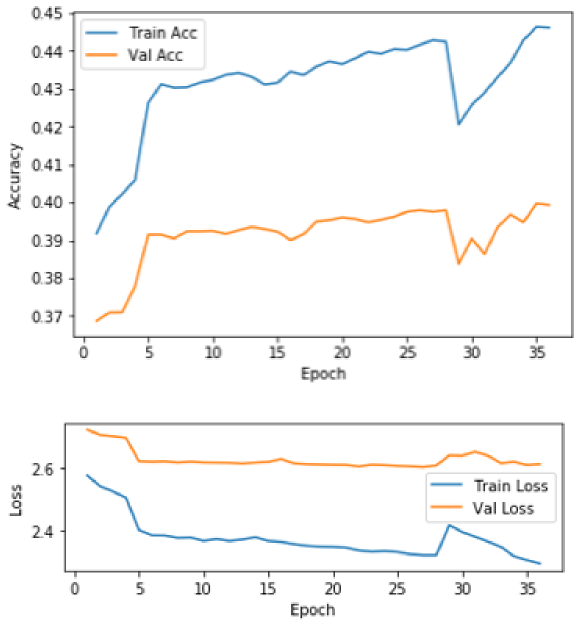
VGG-F



VGG-Y



InceptionV3



network performance.

As with the VGG architecture, I opted to make quite a few changes to the Inception architectures. First off I once again reduced the sizes of the filters to 1x1, 2x1,1x2, and 2x2. Similarly, the original paper calls for 3 of the first type of module, 5 of the second, and 3 of the final type. However, I found that such a deep network led to severe overfitting – often with very little generalization. In turn, I opted to reduce the size of the network, and indeed ended up altogether removing the last of the Inception modules from my networks.

As with the VGG networks, the driving hypothesis for the changes was the change in dimensions of the images, as well as the number of training examples available.

4. Experiments and Results

4.1. Proposed Architectures

Driven by the previously mentioned hypothesis, the final four network architectures are shown in the figure above. The three VGG networks were ensembled and final predictions on the test set were used leveraging the ensemble.

4.2. Results

The figure above shows training time loss and accuracy as well as validation loss and accuracy. The VGG models were all trained with an initial learning rate of 1e-3, with a learning rate drop by a factor of 0.5 every 20 epochs. The Inception model was trained with an initial learning rate of 5e-4 and plateau based learning rate reduction was used based on validation loss.

For VGG models, the higher learning rate was selected due to the abundance of Batch Normalization layers, as it is an added benefit of batch norm that we can increase the learning rate [9]. The InceptionV3 model shows training for the last 35 epochs of training – due to a pickling mishap the prior 50 epochs of logging data were lost. Similarly for VGG-F, the model was originally trained and then restarted at a new learning rate – hence the sharp dip at epoch 30.

As we can see, VGG-Z and VGG-Y minimized overfitting significantly – the training and validation curves are very similar, and indeed the gap between training and validation metrics is minimal. Conversely, the Inception module preserves a sizable 0.05 gap in accuracy between training and validation as well as a 0.3 gap in loss.

4.3. Assessment

My models' performance aligns with related work done in previous CS231n iterations, yet leaves much to be desired.

4. References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015

[2] A. Krizhevsky, S.Ilya, and G.E. Hinton. Imagenet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems(NIPS).1097-1105.2012

[3] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1–9, 2015

[5] M. Abadi, A. Agarwal, P.Barham, E. Brevdo, et. Al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Preliminary White Paper. 2015.

[6] Chollet, Francois and others. Keras. GitHub. https://github.com/fchollet/keras.2015

[7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna. Rethinking the Inception Architecture for Computer Vision.arXiv:1512.00567v3. 2015.

[8] T. Dozat. Incorporating Nesterov Momentum into Adam. CS229 Projects 2015. Cs229.stanford.edu/proj2015/054_report.pdf. 2015.

[9] S. Ioffe, C. Szegedy. Batch Normalization: Accelerating Deep Network Training by

Reducing Internal Covariate Shift. arXiv reprint: arXiv:1502.03167v3, 2015

[10] N. Sristava, G. Hinton, A. Krishevsky, I. Sutskever, R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15. 1929-1958. 2014.

[11] CS231n Staff. Load_tiny_imagenet(). Found in Utils.Load_Data.py.

[12] A. Karpathy. Softmax classifier: Cross-entropy loss. CS231n Course Notes. http://cs231n.github.io/linear-classify

[13] Glorot, X. Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010.

[14] Kingma, D. Lei-Ba, J. Adam: A Method for Stochastic Optimization. Conference paper at ICLR 2015. arXiv reprint: arXiv: 1412.6980v9. 2017