Machine Learning Nano Degree Capstone Project

Mike Cassell, mcasse01@gmail.com, November 5th 2016

I. Definition

Project Overview

Our project will be to try to reach the top 10% in Kaggle's hosted competition for the CIFAR 10 dataset. Kaggle is a website which hosts Machine Learning challenges for people to try and create the best solution for a given dataset and objective in open competitions. The competition is to build and train a machine learning model that when shown a new image, can accurately predict the object in the picture. The model needs to conceptualize and predict images from one of ten classes of objects.

Computer vision for recognizing objects is an important field which is used by companies like Google and Facebook for image classification (figuring out who is in a given Hangout photo or what is pictured in a Facebook post). Other uses include automating quality control or even classification of the quality of cucumbers to save manual labor[1].

Object recognition can be generalized to many other real world use cases including identifying and counting vehicles[2] or detecting roads[3] in satellite imagery, identifying or counting people in a public setting or even automatically classifying the contents of YouTube videos[4]. They could also be used for quality control applications or counting objects on a shelf in a retail environment.

The CIFAR 10 dataset used in this competition is a collection of 60,000 images from a larger set of small images called the 80 Million Tiny Image dataset[5]. The dataset was compiled by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton at the University of Toronto.

The CIFAR dataset has been labeled and depicts images from ten different classes of things. The goal of the competition is to use the 50,000 labeled training images to train a model to be able to identify new images from those ten classes correctly. The remaining 10,000 images in the test set have been hidden within a set of 300,000 images to prevent cheating through hashing and labeling by the Kaggle team. The additional photos are just junk images and are ignored during scoring. Finally the 10,000 test images have been modified in non-material ways to ensure no one can complete a hash lookup.

The images are low resolution and in some cases are even difficult for a human to see the object easily. Overall it is estimated that a human operator would have an overall error rate of about 6% (or alternatively a 94% accuracy rate)[6].

[1] Google, Kaz Sato. 2016. https://cloud.google.com/blog/big-data/2016/08/how-a-japanese-cucumber-farmer-is-using-deep-learning-and-tensorflow
[2] T. Nathan Mundhenk(B), Goran Konjevod, Wesam A. Sakla, and Kofi Boakye. A Large Contextual Dataset for Classification, Detection and Counting of Cars with Deep Learning. 2016
[3] Volodymyr Mnih and Geoffrey E. Hinton, Learning to Detect Roads in High-Resolution Aerial Images
[4] Large-scale Video Classification with Convolutional Neural Networks. Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, Li Fei-Fei. 2014
[5] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images
[6] Torch Blog, Sergey Zagoruyko. 2015. http://torch.ch/blog/2015/07/30/cifar.html

Problem Statement

The approach for this project is to design and train a convolutional neural network to model the CIFAR images and be able to predict new images correctly at least 89.93% of the time (this was the cutoff for being in the top 10% of the Kaggle leaderboard.) This will be approached through the development of a basic convolutional network which will be subsequently refined and adapted to attempt to meet the benchmark goal.

Approaches that will be attempted to improve the model include augmenting the dataset, changing the architecture and applying additional operations that have proven successful in ML literature.

Metrics

The model will be evaluated with two metrics: Accuracy and the Categorical Cross Entropy (CCE) loss. Accuracy is simply the number of correct predictions divided by the total number of predicted values and will serve as the method to compare performance to the Kaggle competition since that is the metric reported on the leaderboard. Accuracy is a good metric for this use case since the success of the model is judged as a binary decision on if the model correctly predicted the image.

This metric can fall victim to the accuracy paradox with unbalanced datasets where one class dominates the sample. The issue arises when the model can simply predict all values to be the dominating class and end up with a higher accuracy than the model would otherwise predict. As discussed below, this dataset is balanced and so this should not be an issue in this case.

CCE is the cost function for the model and estimates the variance between the predicted probability of each class for a given image compared to the actual (where there is one true value and 9 false with a probability of zero.) This is only used as the cost function in back-propogation and as a measure of the models training. This is helpful as this type of model generally predicts the probability of an image belonging to each class and the highest probability is taken as the truth instead of looking solely at a binary correct decision that doesn't provide any information about the certainty of a prediction. CCE is only effective when the categories are mutually exclusive which this dataset is. If the images could contain both a car and a dog, a different measure would be needed.

The formula for calculating CCE is the negative sum of the product of the actual values and the log of their predicted values:

In pseudo code[7]:

CCE = - sum(ya[i] * log(yp[i]), I=0 ,,, N-1)

Where ya is the actual (one hot encoded) classes, yp is the predicted class probability distribution, and N is the number of classes.

Mathematically[8]:

$$H(p, q) = -\sum_{x} p(x) \log q(x).$$

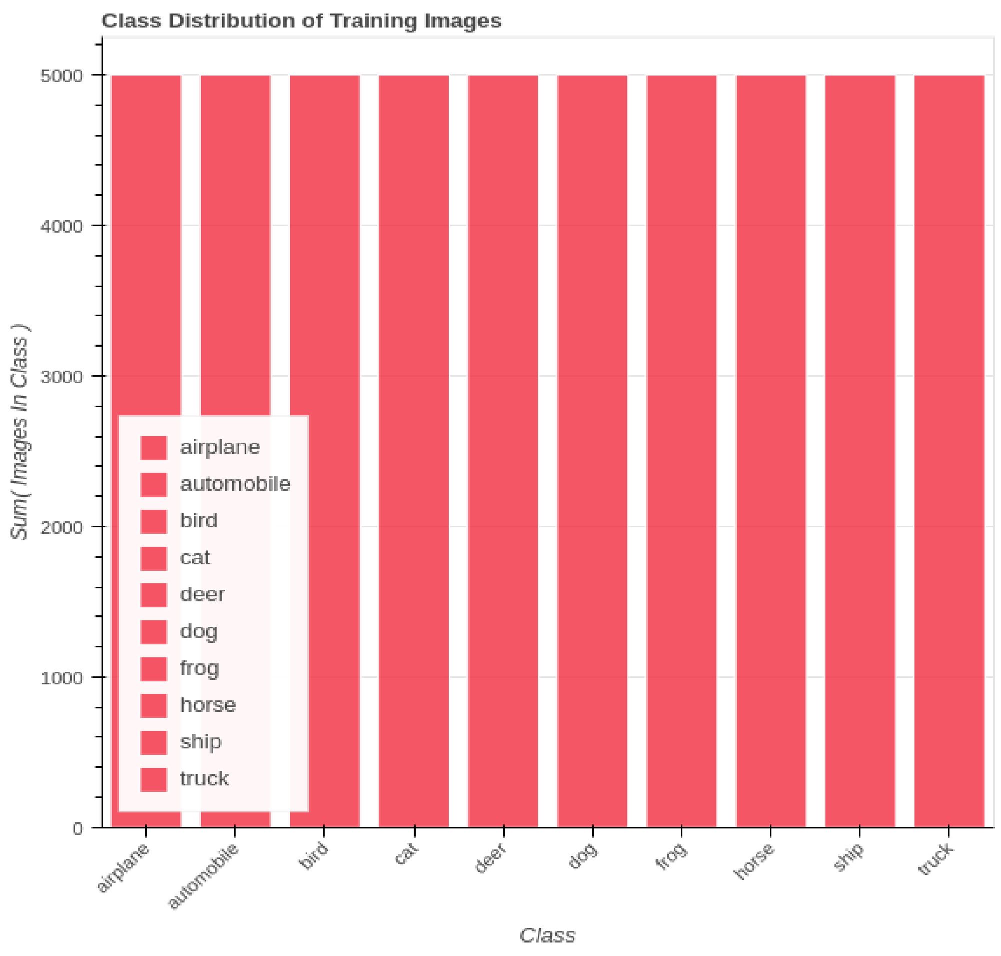
[7] https://www.reddit.com/r/MachineLearning/comments/39bo7k/can_softmax_be_used_with_cross_entropy/
[8] https://en.wikipedia.org/wiki/Cross_entropy

II. Analysis

Data Exploration

As mentioned in the introduction, the dataset consists of 50,000 labeled images and 10,000 unlabeled test images (hidden within 300,000 total testing images). The images are all of one class only (no multiple class images) and there are none that do not have any objects. The distribution of the images is even at 5,000 images from each class represented in the training set as illustrated below in figure 1.

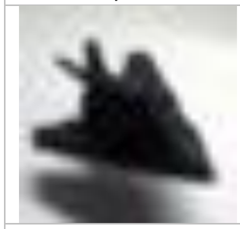
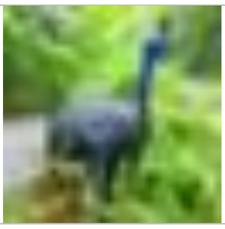
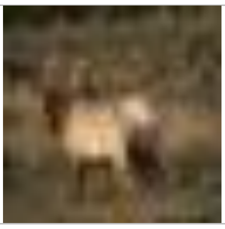
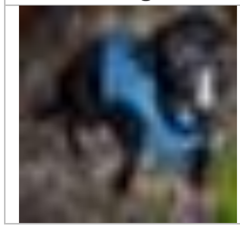
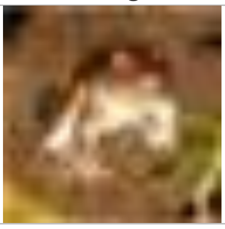
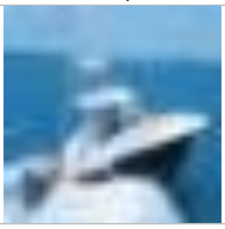
Figure 1: Distribution of the Training Classes



Each image is stored as an individual PNG file and they are uniformly 32x32 pixels with 3 color channels (RGB). There are no size or layer level outliers although a number of photos appear to be in black and white or are desaturated in color but these are still stored as RGB images. All of the images are loaded and converted to integer based arrays of 32x32x3 prior to normalization and cropping.

Since all of the images were consistent in dimensions, there were only what were considered contextual outliers – cases where the object represented in the image that could be argued to belong in between classes (large automobile classed trucks). The entire set could not be inspected by hand to determine the quantity of these cases but the ideal model will be able to account for these and classify them correctly.

Figure 2: Visualization of the Classes

Airplane	Automobile	Bird	Cat	Deer
				
Dog	Frog	Horse	Ship	Truck
				

Algorithms and Techniques

Neural Networks and Fully Connected Layers

Neural networks are a form of mathematical algorithm that was originally inspired by the working of neurons in living systems[i]. While the analogy isn't perfect, a neuron in a neural network is simply a function that given a set of inputs, will conditionally be activated to produce an output value which replicates a simple neuron in a brain. An artificial neural network (NN) is a group of these neurons that have been arranged in layers to produce an output approximating a more complicated mathematical function than a single neuron could express. These neurons apply a simple linear function to the inputs, possibly add a bias term and then typically use an activation function to determine if they will 'fire' to the next layer. The deeper a network (in terms of these stacked layers), the more complex a system can be modeled and by using different activation functions, they can even learn to approximate advanced non-linear functions.

The learning or training phase is simply establishing the weights and biases to be used by the model. This is done by starting to expose example inputs to the model and calculating a cost or loss function to determine how accurate the resulting output was. By using the cost value, a corrective term can be pushed back through the network using calculus, adjusting the biases and weights to reduce the cost (a process called back-propagation). When done slowly over a number of these training iterations with many diverse examples, the model begins to 'learn' how to correctly handle the inputs to produce the desired output. The basic layer in a basic NN is called a Fully Connected layer as each of the previous layer’s outputs are exposed to each of the next layer’s neurons so that every neuron has full visibility of the prior layer.

Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a form of Neural Networks that have effectively been the state of the art in image recognition over the last few years. A CNN may have a similar overall structure to a regular NN in that there are layers of neurons that are stacked and trained by first passing labeled examples through the network and then adjusting weights through back-propagating a cost function through the network.

In a convolutional layer, the neurons are arrayed in a 3 dimensional shape instead of a linear arrangement: typically a small number of pixels or features from the input are focused on at any given moment in the window (commonly 5x5 or 3x3 windows are used for the x and y dimensions) but with a number of additional depth layers added in the calculation weights (the starting image will likely have just three for red, green and blue to which additional layers are stacked, all of the depth weights collectively are referred to as the filters of the layer).

The network is slid over the image and the dot product between the weights in the filter and the focused area are computed. This will normally result in the output image being slightly smaller in the x and y dimensions but with new layers of depth due to the sliding window being smaller than the actual image. These products are then exposed to an activation function to determine if it will 'fire' similarly to a normal network neuron responding to its inputs.

As the model learns, these convolutional filters are exposed to the entire image as they are slid and each layer should begin to learn general features within the window instead of specific input pixel meaning (at a low level, edges are a good example of something that might be activated in a given filter layer). As the model passes through additional convolutional layers, these features become more complex: the first layer might learn basic features like corners or edges, the next layer might learn lines, followed by a shape layer and finally a layer that can recognize a house. While this is dramatically oversimplifying, it's a good illustrative model.

Activation Functions: Relu, Leak Relu and Softmax

In this model we made use of three forms of activation functions to determine if the neurons 'fired' to pass on a signal in testing and two in the final model. The first and most common in the model is named Relu or Rectified Linear Unit. Relu is a very simple function which simply compares the input to zero and takes the maximum[9]. If the input to the function is negative, the neuron does not fire. We also tested but did not use Leaky Relus which follow the same logic but instead of zero, pass on a small negative value (normally the input multiplied by a small alpha value).

Finally, Softmax is the final activation function used in the network which acts to take the fed in values separated by class and to squash them into an array where all the values are between zero and one and sum to one. This output represents the probability distribution amongst those classes and allows for easily distinguishing the most likely class represented by the model. The mathematical formula for the function is illustrated below[10] where K is the number of classes and Z is the vector:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

[9] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[10] https://en.wikipedia.org/wiki/Softmax_function

Max Pooling

Between the layers it's common to use a pooling operation. Pooling operations are an effective way to reduce the x and y dimensions of the models data as it flows allowing for the previously activated filters to be summarized in a given two dimensional areas which essentially scales in the image for the next filter. If an image is 32x32 pixels and is pooled by a factor of two, the resulting image would be summarized to 16x16 pixels.

While some information is lost in the summarizing of the data, when the next convolutional layer is applied, a 3x3 filter would now effectively be able to see an area that was originally 6x6 in the previous layer's activations. This allows the model to learn larger patterns as it aggregates the activations in the prior layers in space. As the model gets smaller in the x and y dimensions, if everything works as intended, more and more of the underlying information is translated into the appropriate depth layers making a final summary much more efficient since the model is emphasizing learning general features instead of specific pixel values.

Other pooling operations are also commonly used with Average pooling being the most common, simply replacing the max function with a mean. All discard some of the information in the image in order to reduce the size.

Dropout

Dropout is a method to avoid overfitting a model during training. A simple explanation is that between two layers of a network, a dropout operation will remove a set proportion of the output of the first layer before it is presented as an input on the next. The network accounts for the lost data points so inference can be run without removing data in non-training passes. By randomly removing features from the data being presented, the model should avoid overfitting since any given feature may not be present during training which should prevent the model from overfitting to that feature.

ADAM Optimizer

Instead of normal stochastic gradient descent, we opted to use the adam optimizer in this model. The adam model differs from SGD in a few key ways: adam computes a moving average of the gradient and the gradient squared which are the first and second order moment estimates. These are decayed over time and used to alter the gradient descent step size used in updating the weights in the model. As the model gets closer to converging the step size decreases further controlling the effect of the learning rate better than in regular SGD or through exponential decay[11].

Benchmark

There are a few relevant benchmarks for this experiment: The top 10% of Kaggle scores was at 89.93% accuracy and this represents our primary benchmark. Since this is a well-known popular data set for the study of machine learning architectures, we can also refer to published results using more complex models, conveniently curated on github by Rodrigo Benenson[12]. The current state of the art is 96.53% by Graham in their paper titled Fractional Max Pooling[13].

[11] http://colinraffel.com/wiki/stochastic_optimization_techniques
[12] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130 retrieved 10/30/16
[13]Benjamin Graham. 2015. Fractional Max-Pooling, 2 ttps://arxiv.org/abs/1412.6071

III. Methodology

Data Preprocessing

The base model we started with used no pre-processing other than normalizing the images and were then simply batched and fed into the network as 32x32x3 integer arrays. Part of the experimentation in making the model involved testing several augmentation methods discussed here. We tested with normalizing the images further with mean centering but found the results actually lowered accuracy in testing and so aside from ZCA whitening, no other manipulation of the input pixel values was completed in the final model. The augmentation schemes were tested additively:

1. Randomly flipping images vertically
2. Randomly flipping images left to right
3. Randomly cropping the images to a 28x28 square

These were done to effectively generate additional training samples for the model: by flipping the images both vertically and horizontally, we can essentially quadruple the size of the training data. By then cropping randomly for 28x28 squares from within the original 32x32, by sacrificing a small amount of edge data, we can even further increase the overall size of the training set.

All of these operations were done using TFLearn's built in augmentation pipeline. To accommodate the 28x28 samples, we opted to add a center cropping for all input images so that the sizing remained consistent. While it was also possible to add 0 padding around the randomly cropped images to maintain their original size, since in training this would essentially always be null data, we opted to keep the convolutional filters slightly smaller for computational efficiency.

The random cropping was done through TFLearn's built in random crop function. Since the operation acts on the data feed (vs. generating additional sample images) the classes remain balanced. Since the method uses Pyhton's built in random function to generate the cropping patterns and the shuffling of the batches, the cropping is not consistent between passes and classes. Due to the high final number of passes (256 epochs) and the smaller number of possible cropping patterns (4x4, 16 possible crops per image) the balance should be consistent over the life of the training run.

ZCA Whitening

The one special case of pre-processing that proved useful was ZCA whitening. Whitening is a process where the covariance is removed from features in a model leaving the resulting features more statistically independent. The most common form of whitening is in Principal Component Analysis, which tends to provide a smaller set of features that are statistically independent allowing for a better model to be built taking only the *k* best features (and generally produces fewer and better output features for models to learn.) PCA does this through a process of rotating the features through orthogonal transformations resulting in features that are linearly uncorrelated[14].

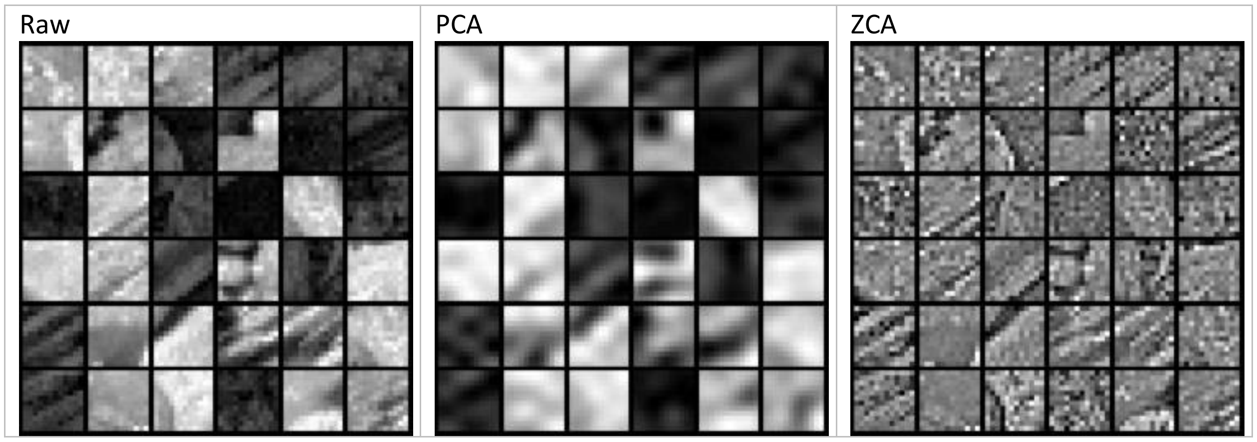
In CNNs this covariance can be thought of as how individual pixels tend to be very highly correlated with their neighbors (even the edge of a house or a tree branch might span many pixels). PCA whitening tends to cause issues with CNNs in two ways: typically, PCA reduces the number of dimensions (and can

[14] https://en.wikipedia.org/wiki/Principal_component_analysis

be used as a form of feature selection by taking only some of the best statistically independent features) and due to the rotational nature, may shift features locations within the 2d space of an image.

ZCA works in a very similar manner but minimizes the rotation and typically retains the same number of resulting features. By maintaining this spatial information, the convolutions can better learn spatial patterns in the output while still maintaining the benefits of whitening (through reducing covariance in neighboring pixels) which in practical terms can help to highlight edges or other transitions in the images. This is very well illustrated in the below images from the Stanford Unsupervised Feature Learning/Deep Learning exercise page[15]:

Figure Three: Comparing Whitening Methods



The PCA processed images tend to have less detail visible in the resulting image while the ZCA tends to highlight regional differences, enhancing circles and lines.

Implementation

The project was completed using TensorFlow and an abstraction layer named TFLearn in Python. A pipeline was constructed to read in the images in batches, set aside 10% of the images as a validation set, preprocess the images and for the training steps, apply the augmentation outlined above. The base network consisted of 2 convolutional layers composed of a convolutional element followed by a RELU activation, repeated and then pooled. After this, we applied a 512 neuron fully connected network (again with RELU) followed by a 10 neuron network with Softmax outputs resulting in the final prediction.

Refinement

The baseline results of the network were encouraging (test accuracy was 80.69%) but there were a number of potential improvements tested. Due to computing limitations, we were unable to test every modification on its own (in a grid search fashion). For consistency the batch size was maintained at 64 images per batch throughout all the experiments not including FMP. Due to an interaction between

[15] http://deeplearning.stanford.edu/wiki/index.php/Exercise:PCA_and_Whitening

TensorFlow and TFLearn with FMP, the batch size was adjusted to 100 to ensure that the batches could be evenly divided into the total sample (leaving no shorter end batch.)

The following is a summary of the most important changes that were tested (additively):

Figure 4: Model Development Iterations

Model	Description	Epochs	Val Acc.	Test Acc.
1	Vanilla CNN	64	82.60%	80.690%
2	Second fully connected layer	64	77.64%	77.070%
3	Vertical and horizontal flipping augmentation	64	71.37%	75.000%
4	Random cropping (32x32 to 28x28)	128	70.41%	74.420%
5	Post-activation function batch normalization	128	84.71%	81.750%
6	Pre-activation function batch normalization	128	87.40%	83.250%
7	ZCA Whitening	128	89.00%	83.375%
8	Leaky Relu in place of Relu	128	85.28%	81.750%
9	Fractional Max Pooling	128	81.32%	80.750%
10	Longer training run	256	84.56%	85.500%
11	Used multi-eval for 10 vote ensemble polling	256	84.56%	88.100%

1. Batch Normalization added prior to max pool operations
2. Batch Normalization added prior to each RELU step

The basic model building blocks were a convolutional layer followed by a relu activation, another convolutional layer followed by another relu followed by a max pooling with a kernel and stride of 2 and a dropout layer with 30% of the data discarded. The final dropout layer in the convolutions leading to the fully connected layers was increased to 50% and an additional dropout layer with the higher rate was added between the fully connected layers.

The models consistently used a filter size of 3x3 with a stride of one for all of the convolutional layers and the depths followed a uniform pattern of 32, 64, 128 (and 160 in the final models with a fourth CNN layer). The models were uniformly finished with a 512 neuron fully connected layer with a relu activation followed by a 10 neuron softmax layer.

Unless otherwise noted, we used the default TFLearn configuration for layers: the convolution layers uniformly used uniform scaling for the initialization of weights and zero initialization of the biases. The weights were decayed at a rate of 0.001 and all of the layers were set to a same padding to preserve the size of the data in the x and y dimensions. We did not make use of regularization.

Additional epochs may have resulted in further gains for some of the earlier models but due to computational constraints, we weren't able to run the models past when the training loss appeared to stop decreasing. There were dozens of other iterations that either did not produce interesting results or performed very badly which were not documented in this report but were still made note of in the iteration of future models. The initial FMP run appeared to still have a noticeably decreasing loss so despite it having a lower overall training accuracy, it was kept as a useful feature.

The two major changes tested in the network were Batch Normalization and Fractional Max Pooling:

Batch Normalization

Ioffe and Szegedy[16] asserted that one of the major issues in training deep neural networks was that as the parameters change in each layer, the output space being passed through to the next layer was also modified in such a way that it's distribution is changed. They referred to this as internal covariant shifting. The next layer must then 'learn' the it's activation parameters with with respect to it's modified input distribution. This forces models to have a lower learning rate (to keep changes small) slowing down training.

Their breakthrough was to build normalization directly into the architecture of the model and to normalize each batch as it passes through the network. In their research this allowed for much higher learning rate, dramatically reducing training time, partially eliminated the need for dropout and relieves some of the need for fine tuning parameter initialization.

Through the addition of batch normalization, the model increased in output accuracy by 7.3% in the same number of training epochs with a drastically increased learning rate which greatly assisted in the iterating of additional model architectures.

Fractional Max Pooling

As discussed above max pooling is a common way of reducing an images input space as it moves through a convolutional neural network, essentially allowing the subsequent set of filters to view a scaled out view of the image (since what was previously 4 pixels distance is now aggregated to 2.) The proposed issue with max pooling is that in each transition information is lost (with a typical stride 2 MP, 3 of the 4 pixels are discarded or 75% of the information.)

As discussed by Graham and implemented in TensorFlow RC11, fractional max pooling is a method of pooling at a floating point number between integers and is implemented in part by randomly altering the stride of the pooling operation randomly as it operates over an input so that the output is approximately scaled by that decimal. An input that would normally be scaled from a dimension of 20x20 to 10x10 can instead be scaled to 14x14 preserving additional information. More importantly, since the operation randomizes the stride throughout the training and testing, the same image can be translated from layer to layer differently preventing overfitting (similar to how drop out can prevent overfitting by randomizing the passed on features.)

It was also noted that unlike with dropout, the action continues during testing (and must since the output size must be maintained), by showing the same image several times and summarizing the output an ensemble model can be produced with a positive impact on the final score since the model has the opportunity to infer against multiple versions of the image with differences in the preserved information being passed through.

[16] Sergey Ioffe, Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe

Finally since the output is not as scaled down, the final iteration of our model added a fourth convolutional layer to ensure any spatial relationships represented in the data were captured (since the model had not been as compressed with the fractional pooling added.) Through the replacement of max pooling with fractional max pooling, we observed a decrease in performance compared to the prior iteration but by deepening the network by one layer to preserve the size of the input presented to the fully connected layer we saw a gain of 4.25% accuracy in testing. Further ensembling the model with ten passes increased it by an additional 2.6% providing the highest score of testing by a margin.

Iterative Development Processes

Some of the notes from the development iteration process can be found below. The specific parameters for the models can be found in Appendix B.

The basic network described in the beginning of this section.

1. The basic network with an additional fully connected layer added after the convolutions: this showed a decrease in accuracy and did not appear to be increasing (ruling out additional training.) This change was discarded.
2. The basic network from 1 with the addition of vertical and horizontal flipping (randomly implemented.) This decreased the accuracy but appeared to still be improving (the loss function was decreasing) and so was left in.
1. The training steps were doubled to give the model additional training time. Random cropping was added into the pre-processing pipeline which again caused a small decrease in the accuracy but it was unclear that the loss function had stopped decreasing. Given the importance of augmentation in preventing overfitting, it was decided to keep both augmentation steps.
2. We added batch normalization in place of dropout in the convolutional layers and removing dropout from the fully connected layers as discussed above. The training rate was also increased by a factor of ten to great results (beating the base model maintaining the augmentation).
3. The batch normalization layers were moved to be before the activation functions as suggested in our research[17] which provided further gains in accuracy
4. ZCA whitening was added as a part of the pre-processing pipeline to attempt to assist the model in learning the underlying features.
5. The RELU functions were replaced with Leaky RELU functions to attempt to allow the model to express negative values to subsequent layers but accuracy decreased against both the validation and test sets. The implementation appeared to be tied to the CPU causing a dramatic slowdown and so this was not included in future models.
6. Fractional Max Pooling was added in place of the regular MP operations. The initial results were negative.
7. Since FMP causes the model to increase the size of the data flowing through the model (by not reducing the sizes as much) it was speculated that a fourth convolutional layer was needed to reduce the size (and proportion of the data still encoded spatially) before it is presented to the fully connected layer. This caused a significant gain in both training and validation accuracies.

https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md

8. We implemented passing the test set through the network multiple times (10) and then polling the results for the most consistent prediction as suggested in the FMP paper. This resulted in the highest accuracies in the development and became the final model.

IV. Results

Model Evaluation and Validation

The final model made use of most of the modifications tested throughout the design phase, incorporating batch normalization, fractional max pooling and the augmentation steps to effectively expand our training dataset. With the addition of fractional max pooling (and taking advantage of using this as a built in ensemble method) and a fourth convolutional layer, the model appeared to reach an optimal point between complexity, computational cost and accuracy.

To evaluate the models robustness, we found 10 images from Google Image Search representing the 10 classes in the project. These were not in the test or training sets (and were not even from the original data source for CIFAR-10 or originally scaled to the same size or proportions) and tested them against the model resulting in 9 of 10 being correctly classified which indicates that the model has generalized well to new images.

We also tested the model against the original CIFAR-10 dataset's test set. While Kaggle may have mixed the training and test sets, the final score was at 84.62 percent which is slightly lower than the Kaggle results. The confusion matrix and discussion about the mistakes of the models predictions can be viewed in the Free Form Visualization section below. The divergence from the Kaggle competition data may relate to the method of organization of the data or possibly encoding differences between the PNG format files and the downloaded dictionary of files.

The model appears to be robust and accurate in predicting the output class both with the original Kaggle test set and novel images from the internet.

Justification

The model would have placed 32nd on the public leaderboard and 33rd on the private which indicates the model would not have been in the top ten percent of final submissions. While it's disappointing to have not hit the benchmark, the performance is still excellent for what is otherwise a very simple model with few layers. In terms of practical value, the model correctly predicted the class of images 88 times in 100 which is only 2 fewer than the benchmark goal in real image recognition terms which given the small margin of difference, is still an exciting result.

V. Conclusion

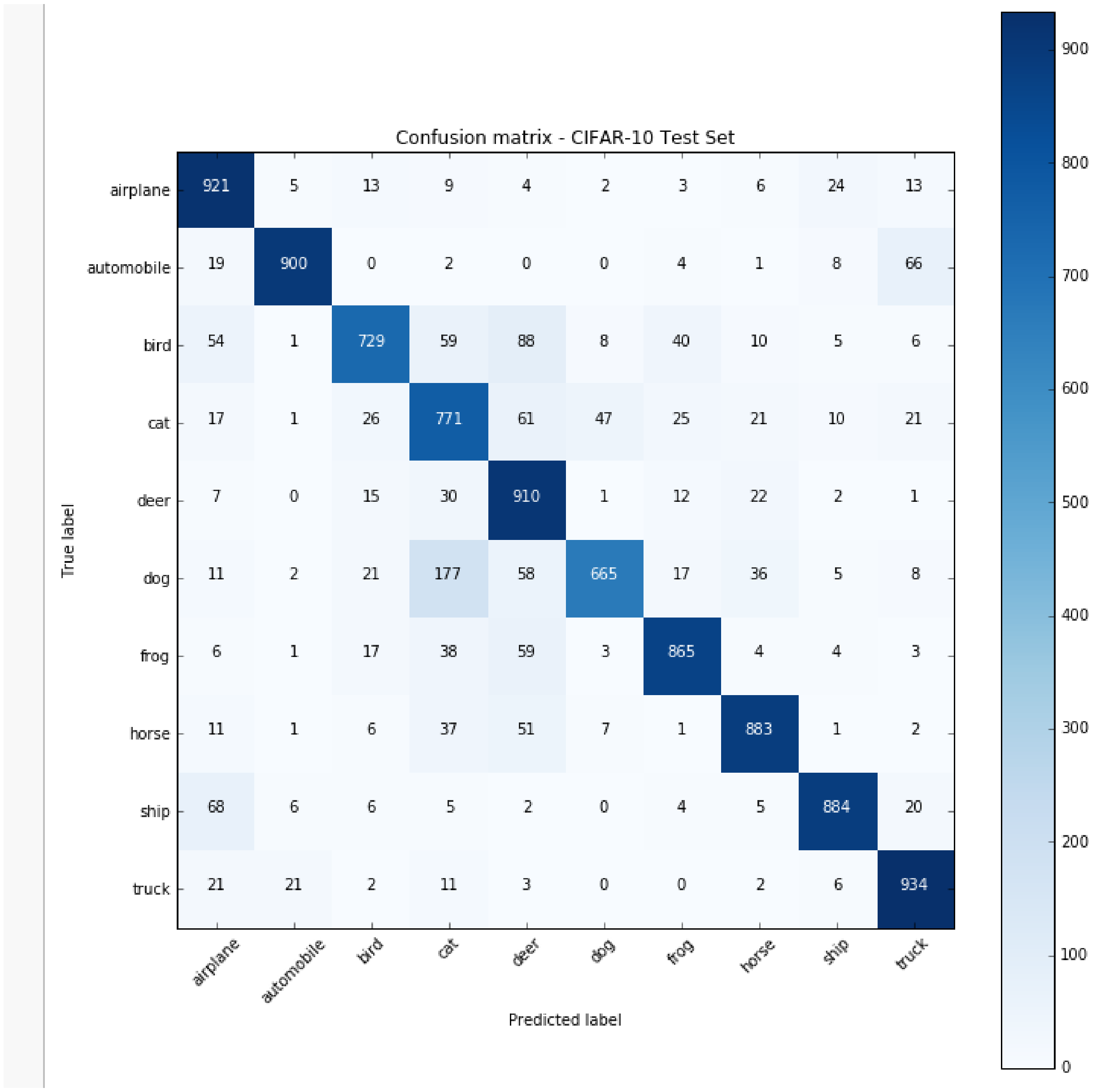
Free-Form Visualization

This is the confusion matrix illustrating where the model was strong vs. weak in it's prediction accuracy against the CIFAR-10 test set as supplied by the original authors. Some interesting observations are that the Bird and Ship classes seem to have a higher degree of cross-confusion with airplanes which might imply that the model struggles with images with a lot of blue (the sky and water). What is especially interesting is that most of these errors are asymmetrical in their frequency – airplanes were very accurately identified (over 92%) and the highest error rates were with ships (2.4%) but the inverse of ships coded as airplanes was much higher (6.8%).

Another interesting error trend can be seen between automobiles and trucks which makes sense intuitively since they might be represented by similar concepts (wheels, headlights and a general shape) and we can expect similar background features (roads on the bottom portion and sky above) in some cases.

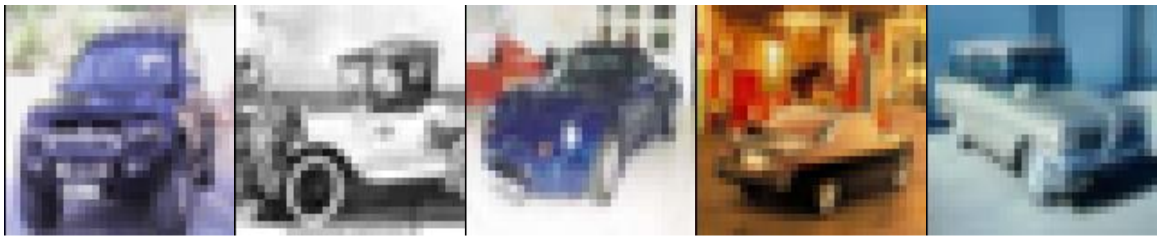
Finally there seems to be a lot of difficulty for the model to have generalized small animals as many dogs were mistaken for cats, birds were frequently predicted to be cats, deer or frogs and the frogs were frequently predicted to be cats and deer. This might point to the model having accurately discovered some features (eyes, mouths or a general facial layout) but having overfit on backgrounds possibly.

A Confusion Matrix Visualizing the Correct and Predicted Classes



True label \ Predicted label	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	921	5	13	9	4	2	3	6	24	13
automobile	19	900	0	2	0	0	4	1	8	66
bird	54	1	729	59	88	8	40	10	5	6
cat	17	1	26	771	61	47	25	21	10	21
deer	7	0	15	30	910	1	12	22	2	1
dog	11	2	21	177	58	665	17	36	5	8
frog	6	1	17	38	59	3	865	4	4	3
horse	11	1	6	37	51	7	1	883	1	2
ship	68	6	6	5	2	0	4	5	884	20
truck	21	21	2	11	3	0	0	2	6	934

Some example images which were Automobiles confused as Trucks



In the first two cases, these vehicles appear to actually be light trucks (which were included with automobiles) but they share similar features to larger trucks in the author's opinion. The remaining images are clearly cars although the third does have a nearly vertical windshield. It's unclear why the third and fourth images could have been mistaken (by a person) but the simple model devised for this project would not be able to replicate human level vision and cognitive capabilities.

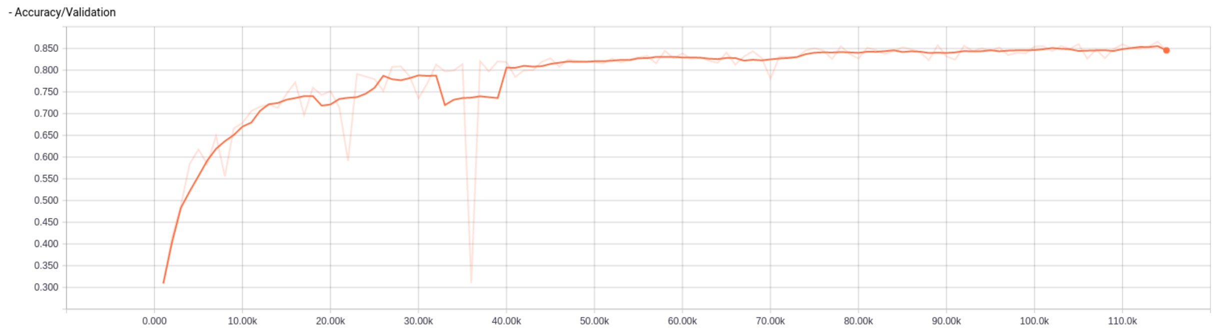
Reflection

The process for approaching this project was to start with a very simple convolutional network and to make changes to it that have been shown to improve recognition performance. The original models were built directly in TensorFlow but due to the complexity in making changes to the networks (altering the graphs, resizing convolutional filters with every layer change etc.) we elected to move to TFLearn to act as an easily modifiable framework. While the flexibility of TensorFlow itself is amazing for understanding the model and optimizing, the tradeoff between speed of iterating vs. depth of modification was too steep (considering early runs took between 8-12 hours and the final run was 18, any time lost between iterations was a significant penalty which was extremely frustrating to the development process).

Another point of frustration in coding the model was in TFLearns handling of batches and implementing Tensorflow's FMP operation. Once FMP was included in the model, the model crashed consistently at the end of the first training pass consistently. After debugging it appeared that the current TF implementation was not handling the partial end batch (with 45000 training samples and the initial batch size of 64, there was a final batch of 8 images) cleanly. Luckily the model did not exceed the available GPU memory and in the interest of speeding up training, the batch size was increased to 100 at that point to eliminate the error and speed up training slightly without discarding samples.

While the final model works well, additional fine-tuning would likely result in better performance gains, when analyzing the final training validation accuracy, it's apparent that the high training rate may have been unstable at some points but thanks to the batch normalization the loss did not explode and the model recovered. Training at a lower rate may prove beneficial as one potential solution that we were not able to approach due to the computational cost.

Figure 5: Training Loss



The testing and modification process was extremely interesting: reading relevant papers and trying to put concepts into production such as batch normalization or fractional max pooling was very exciting. While we weren't able to directly implement either, TFLearn had implemented a batch normalization process over TensorFlow and only recently did TensorFlow itself have MFP added as an operation. Getting to understand and implement these (especially with the resulting improvements in performance) was really gratifying.

The final network takes in a normalized image, crops it to the appropriate size if not already done. It then uses four convolutional layers consisting of two CONV-BN-RELU arrangements followed by a fractional max pooling layer. By then conducting inference operations on the image being evaluated multiple times and taking a vote for the most frequently predicted class, we then get the final prediction.

As the image information passes through these layers, the convolutional layers work to essentially compress data from the two visible dimensions into higher level layers of depth that begin to represent concepts contained in the image. Through the batch normalization, the data is kept consistent (as an input space) for the activation layer so the layer can focus on learning the actual features contained in the data during training instead of having to adjust to the prior layers changes as well.

Once this is done, the activation layer works to infer the features through a non-linear function to build the transformed input to the next layer. Once this is done twice for each phase of the model, we use fractional max pooling to reduce the size of the learned feature space in a way that introduces randomness to the both the learning and inference phases which helped to reduce overfitting in training and provides an easy polling ensemble ability in inference which improves the accuracy of the model by several percentage points. The resulting output is then passed through a simple fully connected layer and then using SoftMax, we produce the output result. The entire final model's structure as visualized in TensorBoard can be viewed in Appendix A.

The final model works very well and even with its limited tuning and relatively shallow depth, it does a great job illustrating how well convolutional networks can be at image classification. While we had originally anticipated (very naively) that an even simpler model would be able to achieve 90% accuracy against the test set, the final model meets its primary objective of accurately classifying images the majority of the time.

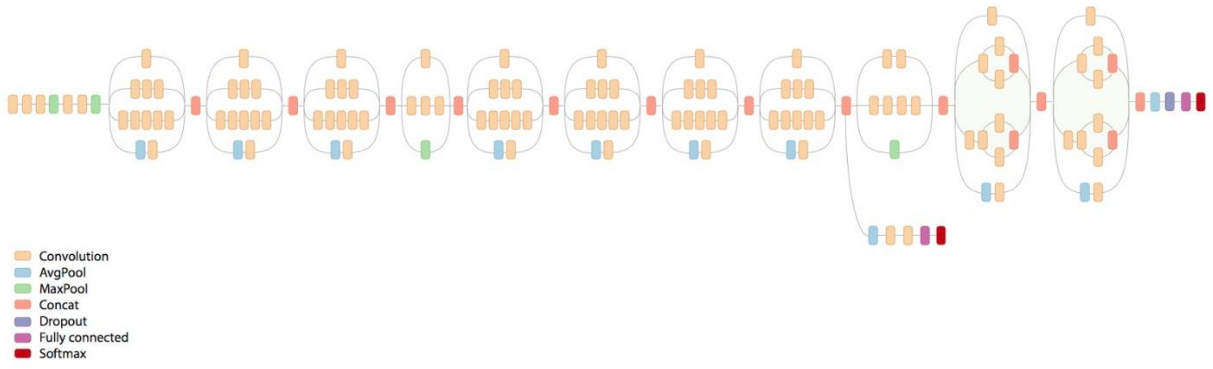
Improvements

The biggest improvement to the model would likely be making it deeper and further hyper parameter tuning (depth of filters at each layer etc.) Unfortunately due to computational constraints, we were not able to grid search these in a meaningful way. This was compounded by the current implementation of Batch Normalization on our system being bottlenecked by the CPU which slowed training times considerably on larger models. In the future as better hardware becomes available, I'd like to revisit the project and continue to refine the model.

There were also several other base architectures which would warrant additional investigation including all convolutional networks (where there is no fully connected layers, instead maintaining convolutions throughout, possibly even using larger convolutional strides in place of pooling operations.) These offer the advantage of not only classifying objects but since spatial data remains throughout, can even be used in the localizing objects within the image to a much easier degree than fully connected models.

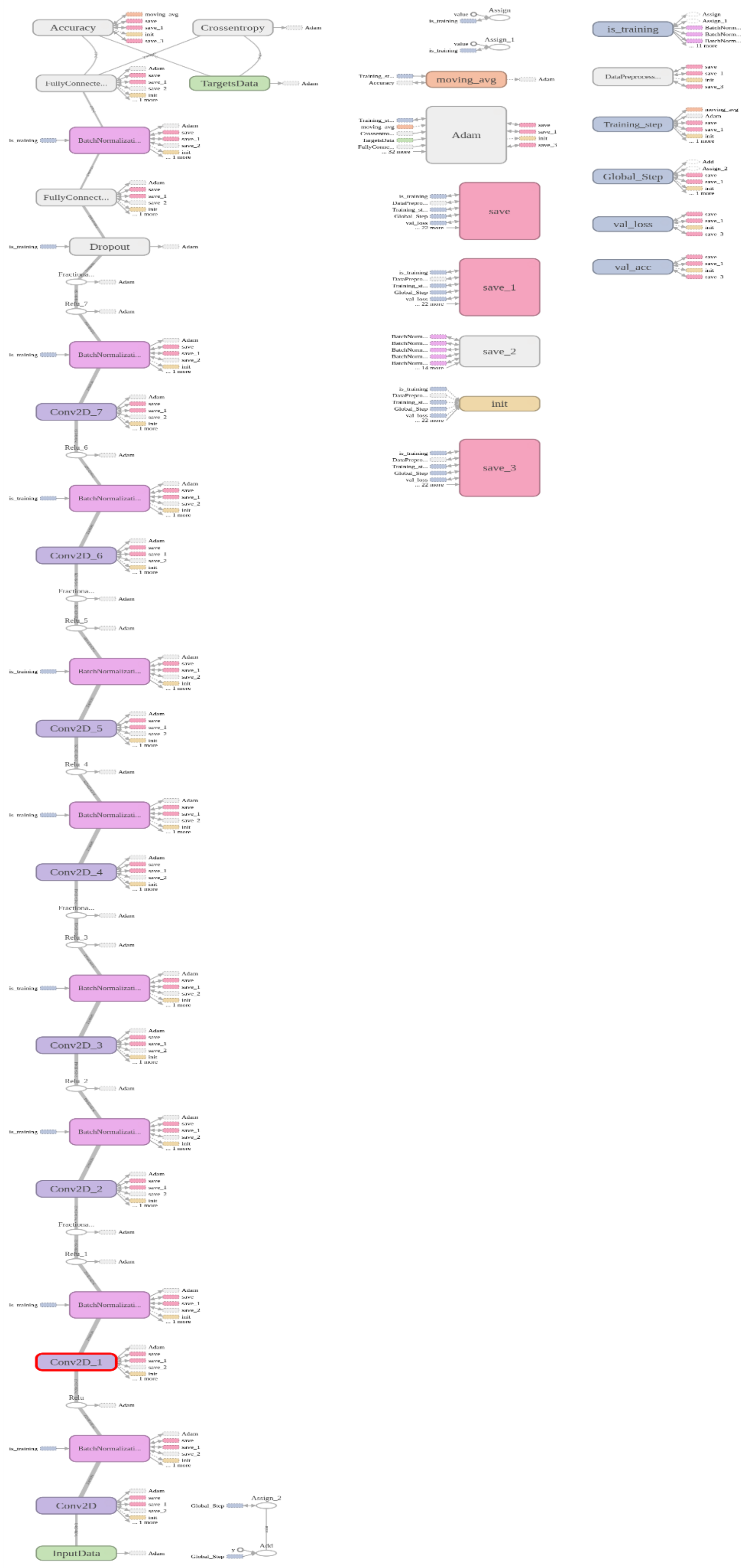
Lastly, larger and deeper models such as Google's Inception V3[18] (pictured below) would likely also offer a better starting point for improved accuracy but can't be efficiently trained on the limited hardware available.

Figure 6: Inception V3 Network



[18] 2016, Google, https://research.googleblog.com/2016/03/train-your-own-image-classifier-with.html

Appendix A: The Final Model



Appendix B

Model	Learning Rate	Conv. Dropout	FC Dropout	Batch Size	Conv. Filter	Conv. Stride	Pooling Kernel	Pooling Stride	Epochs	CNN Groups	FC	Activation
1	0.001	30.00%	50.00%	64	3x3	1	2	2	64	32, 64, 128	512, 10	RELU
2	0.001	30.00%	50.00%	64	3x3	1	2	2	64	32, 64, 128	1024, 512, 10	RELU
3	0.001	30.00%	50.00%	64	3x3	1	2	2	64	32, 64, 128	512, 10	RELU
4	0.001	30.00%	50.00%	64	3x3	1	2	2	128	32, 64, 128	512, 10	RELU
5	0.01			64	3x3	1	2	2	128	32, 64, 128	512, 10	RELU
6	0.05			64	3x3	1	2	2	128	32, 64, 128	512, 10	RELU
7	0.05			64	3x3	1	2	2	128	32, 64, 128	512, 10	RELU
8	0.05			64	3x3	1	2	2	128	32, 64, 128	512, 10	RELU
9	0.05			64	3x3	1	2	2	128	32, 64, 128	512, 10	Leaky RELU
10	0.05		50.00%	100	3x3	1	10/7	10/7	256	32, 64, 128, 160	512, 10	RELU
11	0.05		50.00%	100	3x3	1	10/7	10/7	256	32, 64, 128, 160	512, 10	RELU