Machine Learning Engineer Nanodegree

Capstone Project

Robert Crowe
Rev 2.0 23 November 2016

I. Definition

Project Overview

Learning and recognizing text in images is an application of deep learning with a wide range of uses. This project addresses a subset of text recognition by recognizing single digits from street addresses in outdoor settings using the Google Street View House Numbers dataset. This is important for accurate mapping, for example when used with the Google Street View capture system, by helping to verify and improve existing maps by more accurately connecting addresses with specific locations. The importance of this task for accurate mapping has led a Google team to also attempt the same task. It is also a tractable example of a real-world computer vision problem with practical applications.

The data for this project comes from the Google Street View House Numbers (SVHN) Dataset:

> *"SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images."*

Problem Statement

In order to attempt to match or possibly exceed the Google team's results, I will train a Convolutional Neural Network (CNN) based on the AlexNet (Krizhevsky, Sutskever, Hinton 2012) deep learning architecture using Tensorflow.

AlexNet was a breakthrough that established Convolutional Neural Networks (CNNs) as the leading architecture for image recognition by winning the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). ILSVRD is a competition where teams compete by creating computer vision models for tasks such as classification, localization, and detection. 2012 was the first year when a CNN (AlexNet) achieved a top 5 test error rate of 15.4%, which was a major improvement that convinced the computer vision community of the value of CNNs, and established CNNs as the leading architecture for computer vision. The Top-5 error rate is the fraction of test images for which the correct label is not among the five labels with the highest probability predicted by the model.

In this project I will train an AlexNet in Tensorflow on the SVHN Format 2 dataset, tuning the hyperparameters to achieve increased accuracy, and comparing the results to the Google architecture for performing the same task. Note that since I will be using the Format 2 dataset I will only be attempting part of the Google experimental work. Specifically, I will not be predicting sequences of digits or training the model to localize (find the numbers in a larger image) and segment (separate individual number bounding boxes). This was part of the experimental work that the Google team did, and their result was slightly better than the state of the art (97.53%, Goodfellow et al., 2013) with an accuracy of 97.84%.

However, it should be understood that the results of the Google team's work are very impressive, and it is likely that I will not be able to match or exceed their result. Therefore, a primary goal of this project is the journey itself – attempting to optimize the result from an AlexNet architecture, and developing an understanding of why or why not the results achieved were similar to those of the Google team. This process of attempting, evaluating, and learning from the attempt is a key part of technology development.

Metrics

I'm benchmarking against the Google team's result. Their model attempts to solve exactly the same problem, so the comparison is very relevant. They report their character-level accuracy, which is the accuracy that they achieve when recognizing individual digits, as 97.84%. I am also recognizing individual digits, rather than sequences of digits, so character-level accuracy is the relevant comparison to my work. Accuracy is the only metric that the Google team reports, so it is the primary metric that I will use also, so that I have a direct comparison and can benchmark my results against theirs. However since the dataset is somewhat imbalanced it is important to also examine the confusion matrix, to confirm whether or not we have a problem with high bias (as the saying goes, *"A broken clock is right twice a day"*). To be clear, character-

level accuracy is the accuracy of predictions for single characters, as opposed to sequences of characters.

The Google team also goes on to train and report on a much larger dataset which is not publically available, so I will not be making that comparison. Please note also that the Google team required 6 days with 10 DistBelief replicas to train their model, and I do not have the luxury of that level of resources.

This is a multi-class classification problem, where the model will examine an image and determine which of 10 classes (the digits 0-9) is contained in the image. The level of recognition accuracy is the primary evaluation metric. Accuracy is defined as the number of correct predictions made as a percentage of total predictions, using the highest probability class as the prediction (essentially, a Top-1 prediction).

II. Analysis

Data Exploration

The SVHN Format 2 dataset is in Matlab format, with each image 32x32x3 (RGB). The dataset is unbalanced, but hopefully reflects the real-world bias of street addresses, meaning that lower digits - especially 0, 1, 2, and 3 - are more common in street addresses than higher digits like 6, 7, and 8. The dataset is already split into test and training datasets, and the balance of the classes in both datasets is roughly equivalent. The histograms below display the class distributions of the datasets. There are 73,257 digits for training, and 26,032 digits for testing. 10% of the training dataset was held out for validation.

Note that if we viewed the problem as regression then the data would have a positive skew, but as a classification problem the order of the classes as numbers is not really valid. They are simply different classes, not really ordered.

The images in the dataset were cropped to bounding boxes and resized to 32x32. The bounding boxes were extended to form squares, rather than rectangles, and were extended rather than resized to a different aspect ratio to avoid introducing distortion. However, this means that parts of additional digits and/or other objects in the image were included along the margins of the image, which will introduce some distraction to the model. Although to the human eye more than one digit is clearly recognizable in many of the images, only one digit is identified in the label as the primary target.

Figure 1 shows some samples images from the dataset (source: Stanford University) The parts of additional digits and other objects along the margins of the images are clearly visible in many of the examples. Those additional features complicate the problem of learning and recognizing the digits by essentially distracting the model.

Figure 1- Sample images from the SVHN dataset

Image Pixel Statistics

Mean pixel intensity, training set: 0.45141735673
Pixel intensity standard deviation, training set: 0.199291199446
Pixel intensity variance, training set: 0.0397169813514

Mean pixel intensity, test set: 0.457966893911
Pixel intensity standard deviation, test set: 0.225005462766
Pixel intensity variance, test set: 0.0506274551153

Delta mean pixel intensity (train - test): -0.00654953718185
Delta pixel intensity standard deviation (train - test): -0.02571426332
Delta pixel intensity variance (train - test): -0.0109104737639

The training and testing sets of images are similar but not identical in the mean pixel intensity, the pixel intensity standard deviation, and pixel intensity variance. The test set has a somewhat higher mean, standard deviation, and variance. However, the differences do not appear to be significant.

Exploratory Visualization

The distribution of classes within the training and testing datasets are displayed in figures 2 and 3. The unbalanced nature of the datasets is clear, but the hope is that it reflects the real-world ground truth for house numbers in the wild. Fortunately, the balances of the training and testing datasets are roughly equivalent, since the data has already been split.

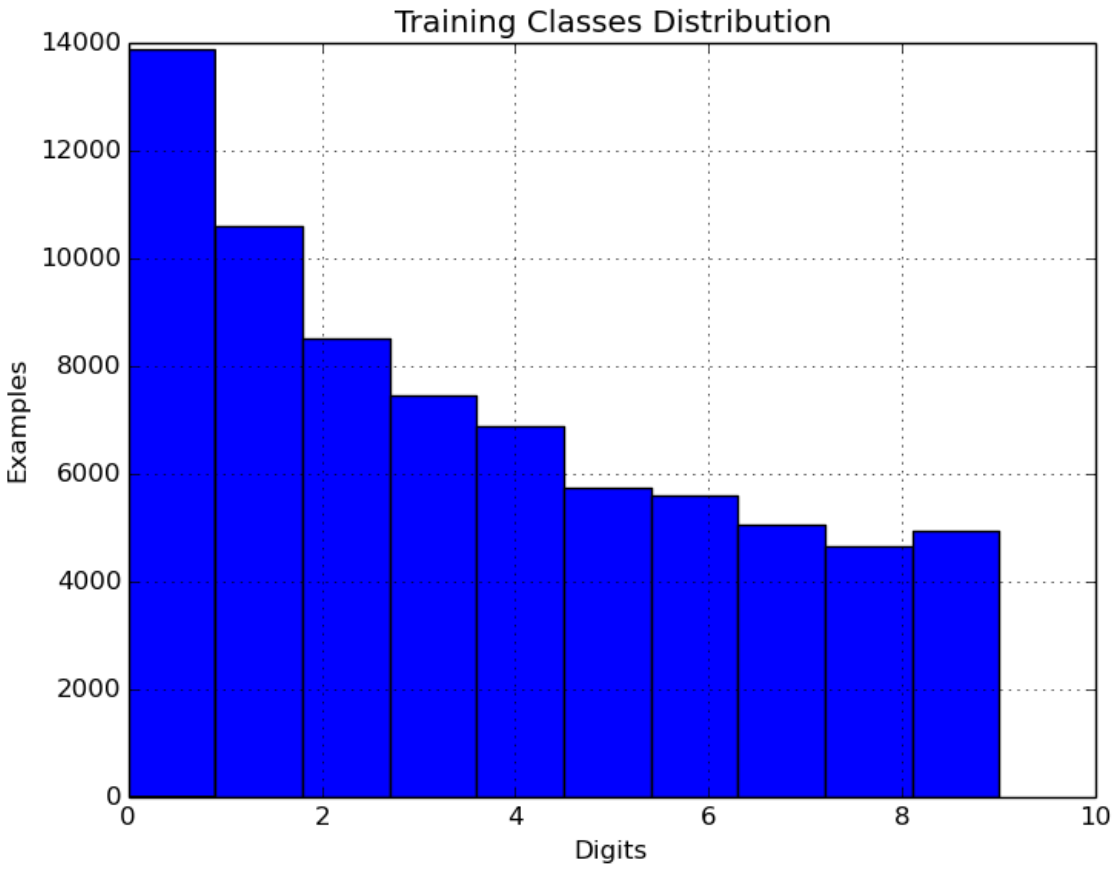

Figure 2 - Distribution of classes in training dataset

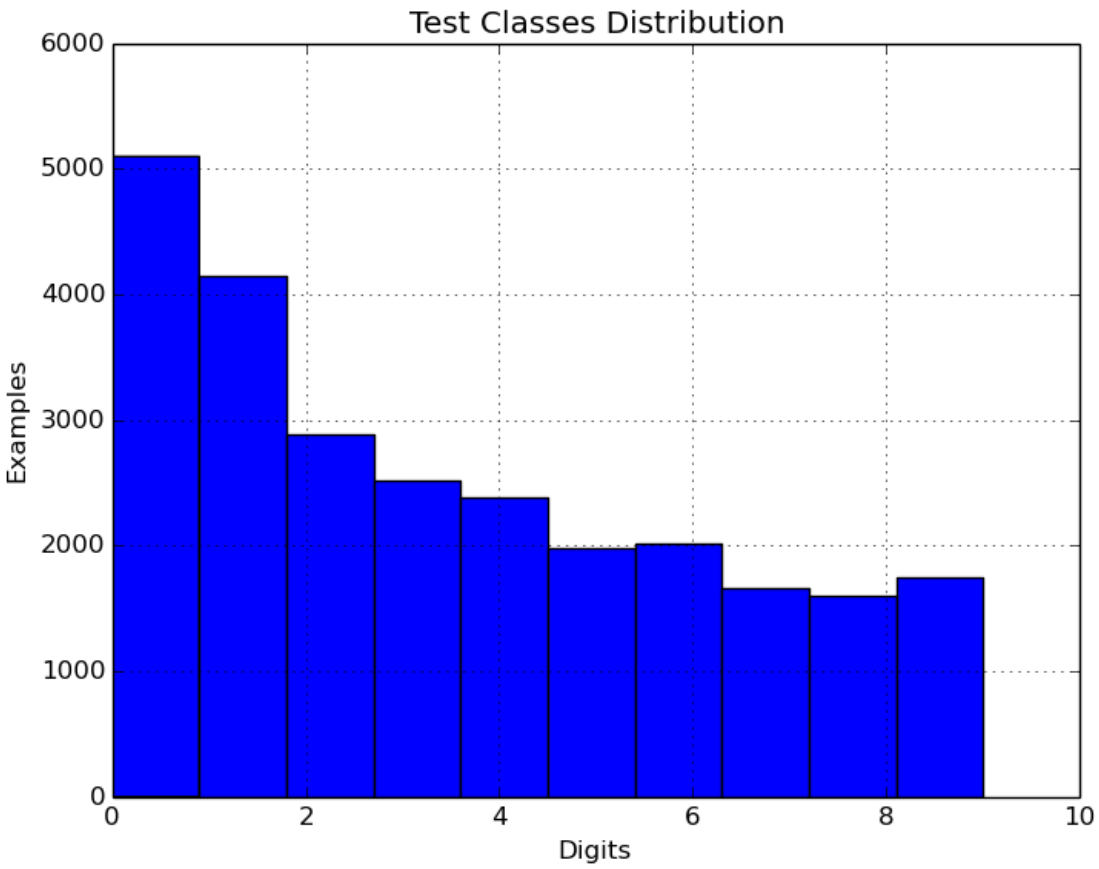

Figure 3 - Distribution of classes in test dataset

Algorithms and Techniques

Convolutional Neural Networks (CNNs) are currently the dominant architecture for creating and training models to recognize images. AlexNet (Krizhevsky, Sutskever, Hinton 2012) was one of the first implementations of a CNN to clearly show the ability of CNNs to achieve results that were beyond what had been achieved with other architectures by winning the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). Since 2012 the ILSVRC has been won by CNNs which incrementally improved upon AlexNet, finally exceeding human abilities in 2015.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural network architectures that take advantage of the 2-D (actually 3-D with color) nature of image data to both reduce the size of the solution space and recognize relevant features in image data. They are inspired by brain and vision research into how humans see and recognize images.

CNNs work by sliding 2-D filters across an image, moving it by the "stride" each time, and recognizing features in each iteration. These filters correspond to the "receptive fields" in the visual cortex of brains. As the layers progress from the input into the hidden layers, the semantic complexity of the recognized features increases. For example, the initial layers may simply locate a shape which might be an animal, while succeeding layers recognize that animal as a dog, and additional layers recognize that dog as a poodle.

CNNs typically have a group of layers that recognize features in the images, following by a group of traditional fully connected layers that perform classification based on those features.

The feature recognition portion of CNNs is typically constructed with three different kinds of layers:

- **Convolutional Layers** apply filters to 2-D sections of the image, sliding the filter across by the stride and learning weights for the convolutional operation.
- **Pooling Layers** reduce the 2-D size of the image by combining neighboring pixel values. The combining operation is typically a max(), but may also be an average().
- **Normalization Layers** work by inhibiting the activation of neurons in the neighborhood of neurons with larger activations, essentially increasing contrast around edges and reducing larger areas of high activation. This is inspired by the way that the brain performs "lateral inhibition", and is discussed in the AlexNet paper.

Layers are made up of individual neurons, each of which takes inputs from the previous layer, applies a function which includes weights and at least one bias value, and outputs the result to the next layer. The functions that neurons apply are known as "activations", and there are many different kinds. AlexNet uses 3 different activations:

- Tanh – The hyperbolic tangent function. Tanh is similar to a sigmoid function, but has a zero mean, which avoids the "vanishing gradient" problem during back-propagation while training the network
  $$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$
- ReLU – The rectified linear function. ReLU is less computationally complex, meaning that it trains faster, it encourages sparsity, and avoids the vanishing gradient problem
  $$f(x) = max(0, x)$$

- Softmax – The softmax or normalized exponential function. Softmax is used to output the probability distribution among a set of classes

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$ for j = 1 to K

Neural networks are trained using "backpropagation" with a cost function. During the forward pass the inputs are passed to the first layer, which applies its activations and passes the result to the next layer, continuing to the final layer where the result is compared to the ground truth value. The cost function is used to compute the error, which is then passed back up through the layers, and the gradient is computed at each step and used to adjust the weights and biases of the layer. This continues as each new example is fed into the network until the error value reaches a minimum. (I'm ignoring mini-batch training to keep things simple). The gradient is important to determine the direction and amount of adjustment to the weights on each pass, so when the gradient reaches an extremely small value training essentially stops (the vanishing gradient problem).

A full discussion of neural networks and CNNs is beyond the scope of this paper.

AlexNet Architecture

AlexNet is a specific CNN architecture that was designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, all at the University of Toronto. As shown in figure 4 it uses 5 convolutional layers for feature recognition, with max pooling and normalization between layers. It then uses three fully connected feed-forward layers for classification. It was designed for the ImageNet feature set, which has inputs of 224x224x3 and learns to classify into 1,000 classes.

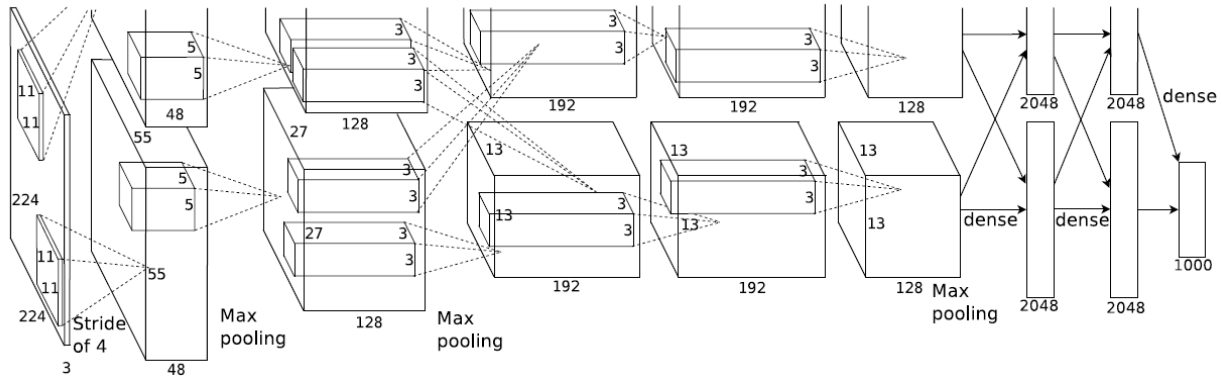

Figure 4 - AlexNet layers

Original AlexNet Layers

There is some controversy about the input layer for AlexNet:

> *"As a fun aside, if you read the actual paper it claims that the input images were 224x224, which is surely incorrect because (224 - 11)/4 + 1 is quite clearly not an integer. This has confused many people in the history of ConvNets and little is known about what happened."*

My assumption is that the input layer was actually 227x227x3.

- Input 227x227x3
- 2-D Convolutional, 96 filters, 11x11 each, stride 4, ReLU activation, output 55x55x96
- Max Pooling, 3x3, stride 2, output 27x27x96
- Normalization, output 27x27x96
- 2-D Convolutional, 256 filters, 5x5 each, stride 1, ReLU activation, output 23x23x256
- Max Pooling, 3x3, stride 2, output 11x11x256
- Normalization, output 11x11x256
- 2-D Convolutional, 384 filters, 3x3 each, stride 1, ReLU activation, output 9x9x384
- 2-D Convolutional, 384 filters, 3x3 each, stride 1, ReLU activation, output 7x7x384
- 2-D Convolutional, 256 filters, 3x3 each, stride 1, ReLU activation, output 5x5x256
- Max Pooling, 3x3, stride 2, output 2x2x256
- Normalization, output 2x2x256
- Fully connected, 4096 wide, Tanh activation
- Dropout, 0.5
- Fully connected, 4096 wide, Tanh activation
- Dropout, 0.5
- Softmax, 1000 wide

Visual Fields

The visual field of each layer is the 3-D size of the input to that layer. Each layer also outputs a visual field to the next layer. For a convolutional layer, the visual field that is output can be calculated from the input width W, the filter size F, the stride S, and any padding P:

$$\text{Convolutional Layer Output visual field size} = (W - F + 2P) / S + 1$$

For a pooling layer, the visual field that is output can be calculated from the input width W, the filter size F, and the stride S:

Pooling Layer Output visual field size = (W – F) / S + 1

Normalization layers output a visual field of the same size as their input.

Receptive Fields and Filters

Filters recognize features within an image. A filter cannot recognize a feature that is larger than the receptive field of that filter, and the visual field presented to a layer limits the maximum size of a filter. Filter sizes and visual field size therefore need to be carefully designed within the layers of a CNN in order to recognize as many relevant features as possible. Figure 5, taken from an unfinished paper by Cao, illustrates that problem well:

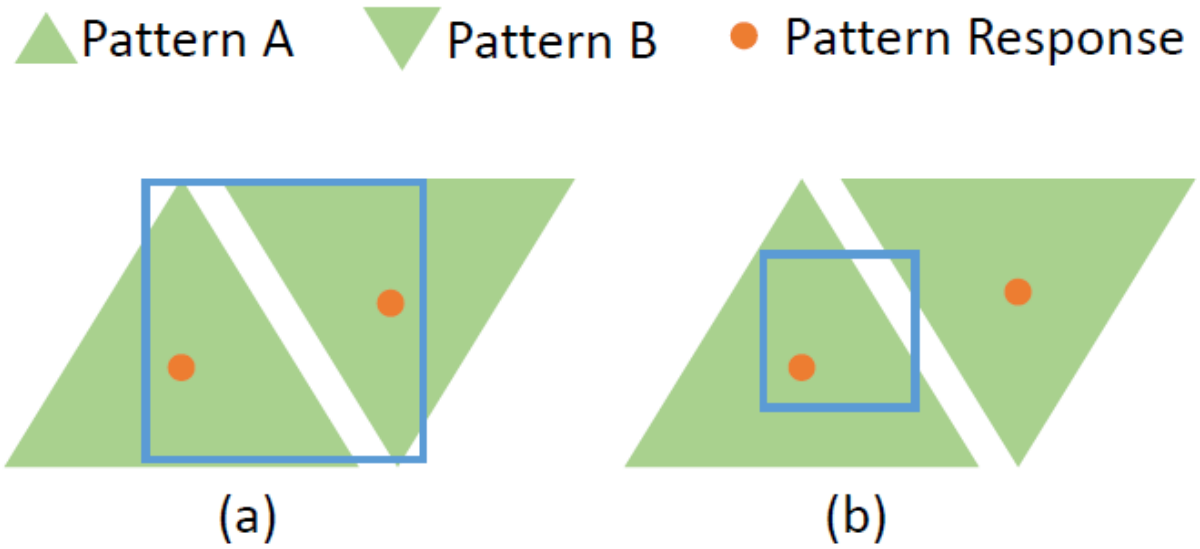

Figure 5 - An illustration of learning capacity. Example (a) has a better learning capacity than example (b). The blue rectangle shows the convolutional filter's size.

Benchmark

The Google team achieved an accuracy of 97.84% for per-digit recognition, which is the relevant comparison to my result.

This is a multi-class classification problem, where the model will examine an image and determine which of 10 classes (the digits 0-9) is contained in the image. The level of recognition accuracy is the primary evaluation metric. Accuracy is defined as the number of correct predictions made as a percentage of total predictions, using the highest probability class as the prediction (essentially, a Top-1 prediction).

The Google team also strongly advocates the use of confidence thresholding to limit the reliance on predictions that fall below an acceptable level of certainty:

> *"We therefore advocate evaluating this task based on the coverage at certain levels of accuracy, rather than evaluating only the total degree of accuracy of the system. To evaluate coverage, the system must return a confidence value, such as the probability of the most likely prediction being correct. Transcriptions below some confidence threshold can then be discarded. The coverage is defined to be the proportion of inputs that are not discarded. The coverage at a certain specific accuracy level is the coverage that results when the confidence threshold is chosen to achieve that desired accuracy level. For map-making purposes, we are primarily interested in coverage at 98% accuracy or better, since this roughly corresponds to human accuracy."* *(Google, 2014)*

The 97.84% result that the Google team reports is before confidence thresholding, so it can be directly compared to my result before confidence thresholding. While confidence thresholding can be used with either solution, the underlying accuracy will determine the coverage that is achieved. Adding confidence thresholding would be an incremental refinement before use in a production environment. I have not added confidence thresholding.

III. Methodology

Data Preprocessing

The SVHN dataset is only made available in MATLAB format, which I was able to read with loadmat() from scipy.io. The columns needed to be re-ordered for Tensorflow, since they are originally in H, W, Colors, N order, and need to be in N, H, W, Colors order, where H and W are height and width, and N is the number of examples.

Like the Google team I chose to subtract the mean from the images, but also like the Google team I did no whitening, local contrast normalization, etc. I felt that this was important to keeping the comparison even, rather than influencing the result through pre-processing.

Implementation

I chose to use Tensorflow as the platform for developing, training, and testing the models because of its open source nature, its powerful and scalable capabilities, and large community of developers. Tensorflow v10 was used on an AWS p2.8xlarge

instance, with Cuda 7.5 and cuDNN v.5. Training with 100 epochs required 6,345 seconds (~105 minutes). Working in Python, I used the TFlearn library on top of Tensorflow to simplify the coding effort, and found it to be well-suited to this project, along with Numpy and Scikit-Learn.

The original AlexNet was designed for input images of 227x227x3, but the SVHN dataset is 32x32x3, so a significant task was to reshape AlexNet for a smaller input size while trying to preserve the original AlexNet design, with similar changes in the visual fields between layers. This involved changing the filter sizes and strides, for both 2-D convolutional layers and Max Pool layers.

My intuitive view of the difference in complexity of the visual features in the ImageNet dataset that the original AlexNet was designed for, versus the SVHN dataset that I am using, is that ImageNet is significantly more complex. ImageNet has 1,000 object categories, whereas SVHN has only 10. This suggests that the number of filters required in 2-D convolutional layers to recognize important features should be significantly lower for SVHN, so I also made adjustments to the numbers of filters (aka "kernels", "kernel maps", or "feature maps"). This resulted in a significant decrease in resource requirements, with a small if any decrease in accuracy.

Figure 6 shows the result of the reshaping:

Layer	Original AlexNet	Reshaped AlexNet
Input	227x227x3	32x32x3
2-D Convolutional	96 filters, 11x11, stride 4	48 filters, 8x8, stride 3
Max Pool	3x3, stride 2	2x2, stride 1
Local response normalization	Original	Unchanged
2-D Convolutional	256 filters, 5x5, stride 1	126 filters, 2x2, stride 1
Max Pool	3x3, stride 2	2x2, stride 1
Local response normalization	Original	Unchanged
2-D Convolutional	384 filters, 3x3, stride 1	192 filters, 3x3, stride 1

2-D Convolutional	384 filters, 3x3, stride 1	192 filters, 2x2, stride 1
2-D Convolutional	256 filters, 3x3, stride 1	128 filters, 2x2, stride 1
Max Pool	3x3, stride 2	2x2, stride 1
Local response normalization	Original	Unchanged

Figure 6 - Original and reshaped layers

Figure 7 shows the resulting changes in the visual fields:

Layer	Original AlexNet Visual Field Output from Layer	Reshaped AlexNet Visual Field Output from Layer
Input	227x227	32x32
2-D Convolutional	55x55	9x9
Max Pool	27x27	8x8
Local response normalization	27x27	8x8
2-D Convolutional	23x23	7x7
Max Pool	11x11	6x6
Local response normalization	11x11	6x6
2-D Convolutional	9x9	4x4
2-D Convolutional	7x7	3x3
2-D Convolutional	5x5	2x2
Max Pool	2x2	1x1
Local response normalization	2x2	1x1

Figure 7 - Original and reshaped visual fields

For all convolutional and layers I kept the original ReLU activations. The feed forward classification network was similarly resized for the change in input size, and I kept the original AlexNet Tanh activations. Where the original AlexNet had an initial fan-out of 4 (from 1024 to 4096) I used a 1-1 ratio because of the reduced complexity of the dataset. Like the original AlexNet, dropout between the fully-connected layers is used to decrease overfitting.

The sizes of the visual fields at each layer were adjusted to adapt to the change in the input image dimensions, while attempting as much as possible to keep the layer-to-layer ratios as close as possible to the original AlexNet. This proved difficult because of the much smaller input size. For example, where AlexNet went from 55x55 to 27x27 (roughly a .75 reduction), I had to resize it to go from 32x32 to 9x9. This illustrates the difficulty of matching the layer-to-layer ratios with the much smaller input size.

Like AlexNet, the Momentum optimizer was used for training with categorical cross-entropy loss. The learning rate was set to 0.01. The batch size was set to 64, and training over 180 epochs seemed to reach an optimum accuracy.

The original AlexNet used weight decay of 0.0005 and weight initialization with a zero-mean Gaussian distribution with standard deviation of 0.01. In testing I found that the weight initialization used by AlexNet produced poor results, and replaced it with uniform scaling. (See discussion below under Results)

Refinement

The Tensorflow model was trained using standard back propagation, using a momentum optimizer and a cross-entropy loss function. The initial learning rate was set to 0.01

In "Practical Recommendations for Gradient-Based Training of Deep Architectures" (2012) Yoshua Bengio offers several practical recommendations for training and refinement of deep neural networks, including convolutional networks. This includes the various hyperparameters, weight initialization, regularization, normalization, grid search, and many other topics. I found the breadth of options and variations to be a bit daunting, and likely to be time consuming and resource intensive if all were pursued.

One important hyperparameter that Bengio emphasizes is the learning rate, so I chose to examine and try some variations to assess the impact of varying the learning rate. Looking at the validation curve with the learning rate at 0.01 in figure 8, I see the testing error rate apparently reaching a floor after about 180 epochs, with test accuracy of 93.56%:

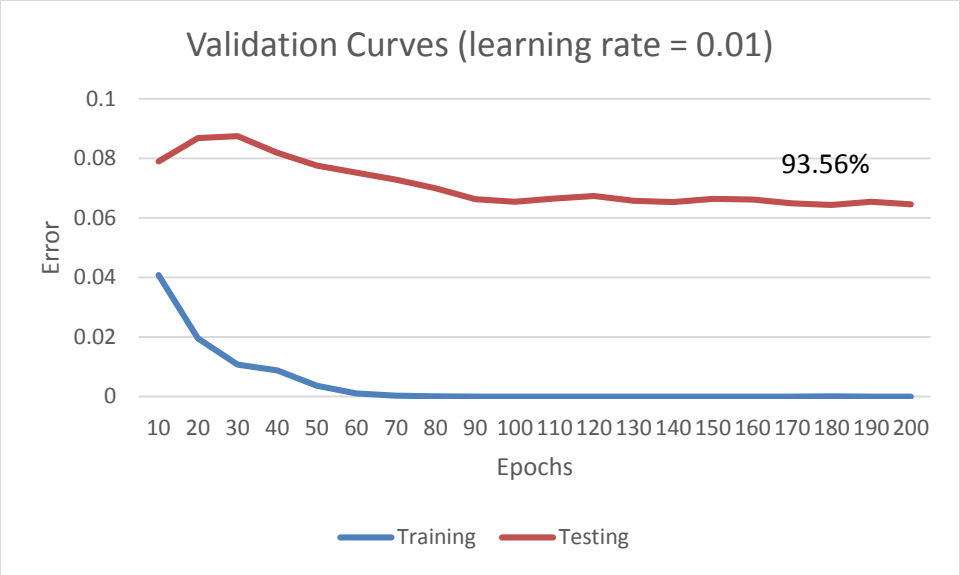

Figure 8 - Validation curves (learning rate = 0.01)

To test whether we are stuck in a local minima I tried setting the learning rate to 0.05, as shown in figure 9:

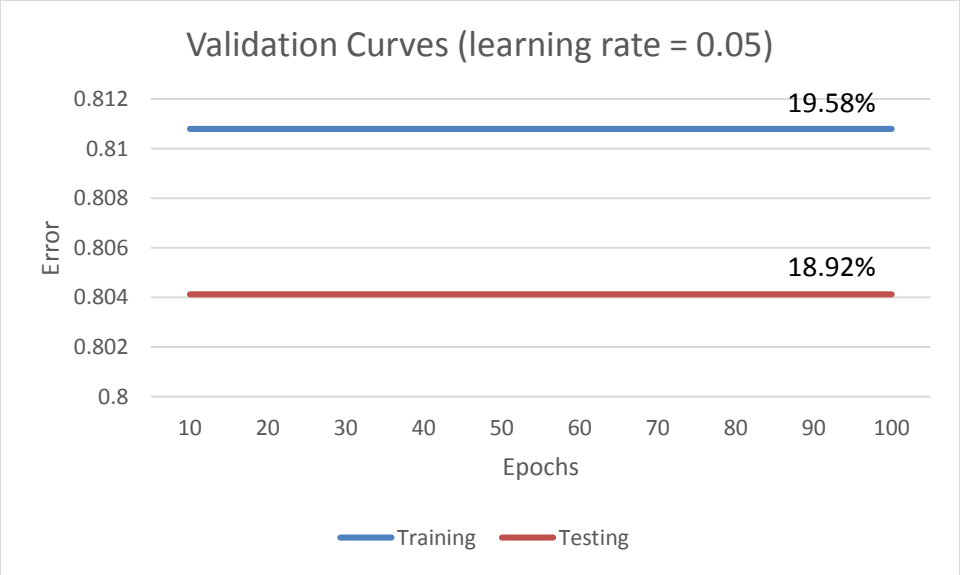

Figure 9 - Validation curves (learning rate = 0.05)

It became quickly apparent that the model was failing to optimize, which looks to me like a classic case of a learning rate that is too high and failing to descend into a lower minima. If we had been stuck in a local minima, setting the learning rate significantly higher should have had either no effect or a positive effect. Instead, it had a negative effect, indicating that we are not descending into a minima as far as we would with a

lower learning rate. So I then tried setting it to 0.005 (figure 10) to test if we can descend farther into a better minima:

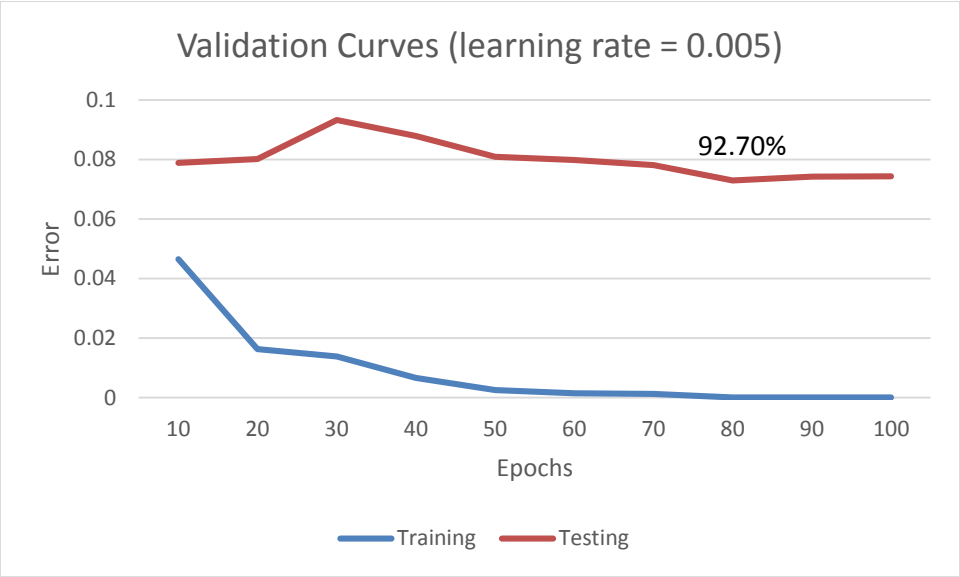

Figure 10 - Validation curves (learning rate = 0.005)

Lowering the learning rate produced slightly lower testing accuracy, to a peak of 92.70% after 80 epochs. That probably indicates that the learning rate is getting stuck in a slightly less optimal local minima, but to confirm I think that it makes sense to see if lowering it further will yield still better results, so I tried setting the learning rate to 0.001 (figure 11):

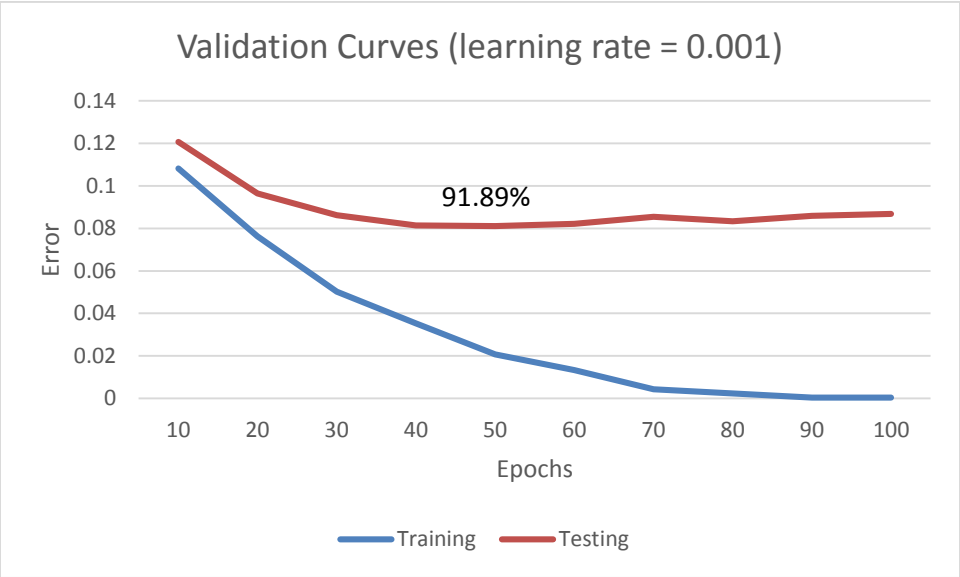

Figure 11 - Validation curves (learning rate = 0.001)

In this case the testing accuracy only reached 91.89% after 50 epochs. That would indicate that 0.001 is indeed too low.

IV. Results

Model Evaluation and Validation

The Google team achieved an accuracy of 97.84% for per-digit recognition, which is the relevant comparison to my result. The best accuracy that I was able to achieve with my reshaped AlexNet was 93.56%. Here is a summary of some of the intermediate variations and results:

- Full set of AlexNet filters (same number of filters for convolutional layers), 10 epochs, learning rate 0.01, accuracy = 88.82%, took 1403 seconds to train
- Adjusted filter set (reduced the number of filters as displayed in the table below), 0.001 weight decay, uniform scaling weight init, 10 epochs, learning rate 0.01, accuracy = 88.04%, took 633 seconds to train
- Adjusted filter set, 0.01 weight decay, uniform scaling weight init, 180 epochs, learning rate 0.01, accuracy = 93.56%, took ~10,800 seconds to train

The results of including the weight decay and weight initialization as in the original AlexNet were astounding. With weight decay of 0.0005 and weight initialization using a zero-mean Gaussian distribution with standard deviation 0.01, the result was:

- 100 epochs, accuracy 19.58%

With weight decay of 0.001 and weight initialization to a zero-mean Gaussian distribution with standard deviation 0.01, the result was also:

- 100 epochs, accuracy 19.58%

This is compared to the previous results, which used uniform scaling for weight initialization and weight decay of 0.001. Uniform scaling is initialization with random values from uniform distribution without scaling variance, so it does not explode or diminish by reaching the final layer.

Weight initialization clearly has a major impact on the training result, at least within 100 epochs of training. Looking at the validation curves, it seems that training for longer periods is unlikely to compensate for poor weight initialization. While initialization to a zero-mean Gaussian distribution with standard deviation 0.01 is clearly a poor match, weight initialization to appropriate values may significantly improve results.

Greedy layer-wise pre-training is an approach that I would like to try for weight initialization, since the results for many applications are often favorable, but it would require a larger dataset for the pre-training phase. The good news is that the pre-training dataset can be unlabeled, since pre-training is unsupervised.

I was unable to match or exceed the results of the Google team with my reshaped AlexNet implementation. Clearly their results are impressive, and the state of the art has improved since 2012 when AlexNet was designed.

Confusion Matrices – Unbalanced Classes

Since the SVHN dataset is somewhat unbalanced it is important to examine the rate of errors in different classes. For example, if 90% of the digits in the SVHN dataset were 5's, and the model always predicted 5, then we would be 90% accurate with a very poor model. This is a high-bias problem, which is illustrated in the old saying that "a broken clock is right twice a day". Simply always reporting the same result, like a broken clock does, can appear to be accurate when used with a dataset which is also biased to that fixed result.

To examine the error rates in each of the classes we can create a confusion matrix from the results, as shown in figure 12. The cells on the diagonal represent predictions that were correct, and the cells off the diagonal represent errors, with the rows and columns showing the true classes and the classes predicted by the model. Darker blue represents high values, with darker along the diagonal showing higher correct predictions, and darker off the diagonal showing higher error rates.

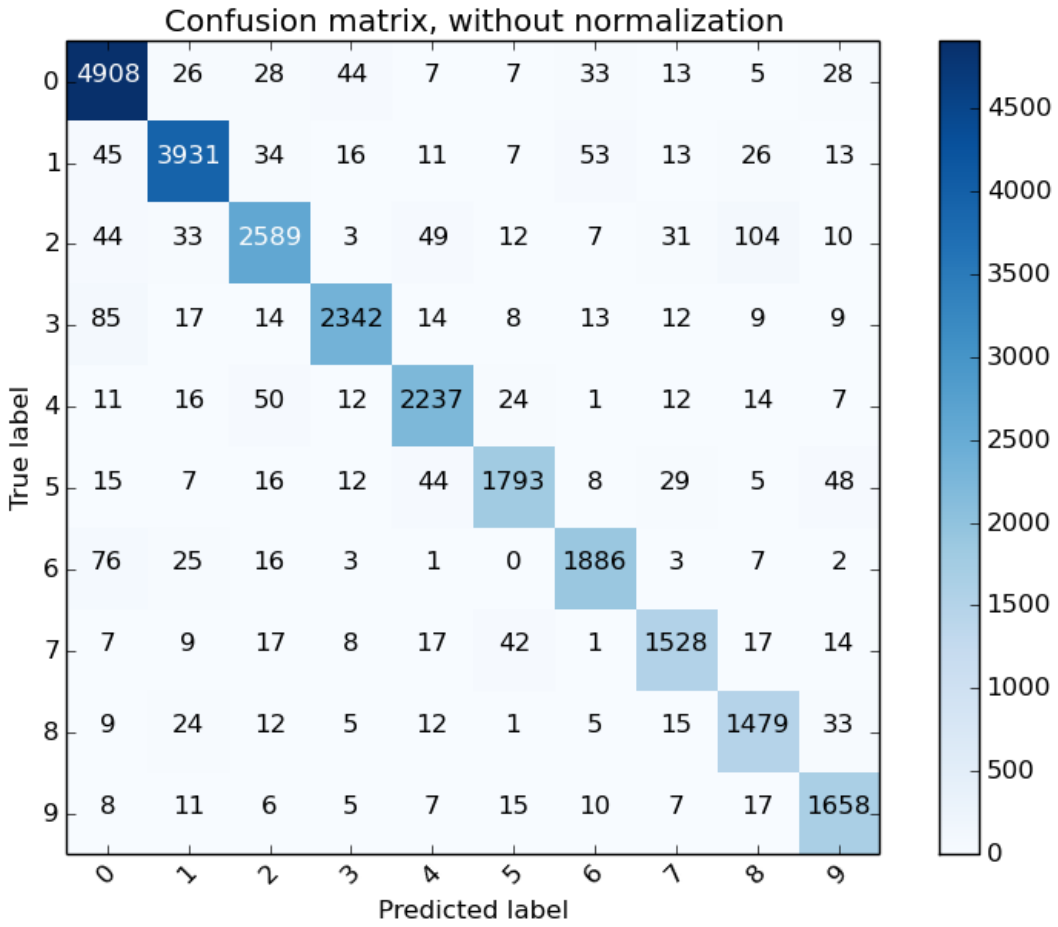

Figure 12 - Confusion matrix, no normalization

While figure 12 shows the raw numbers for each combination of classes, it is important to look at a normalized version of the confusion matrix so that we aren't confused by the class imbalance (Figure 13).

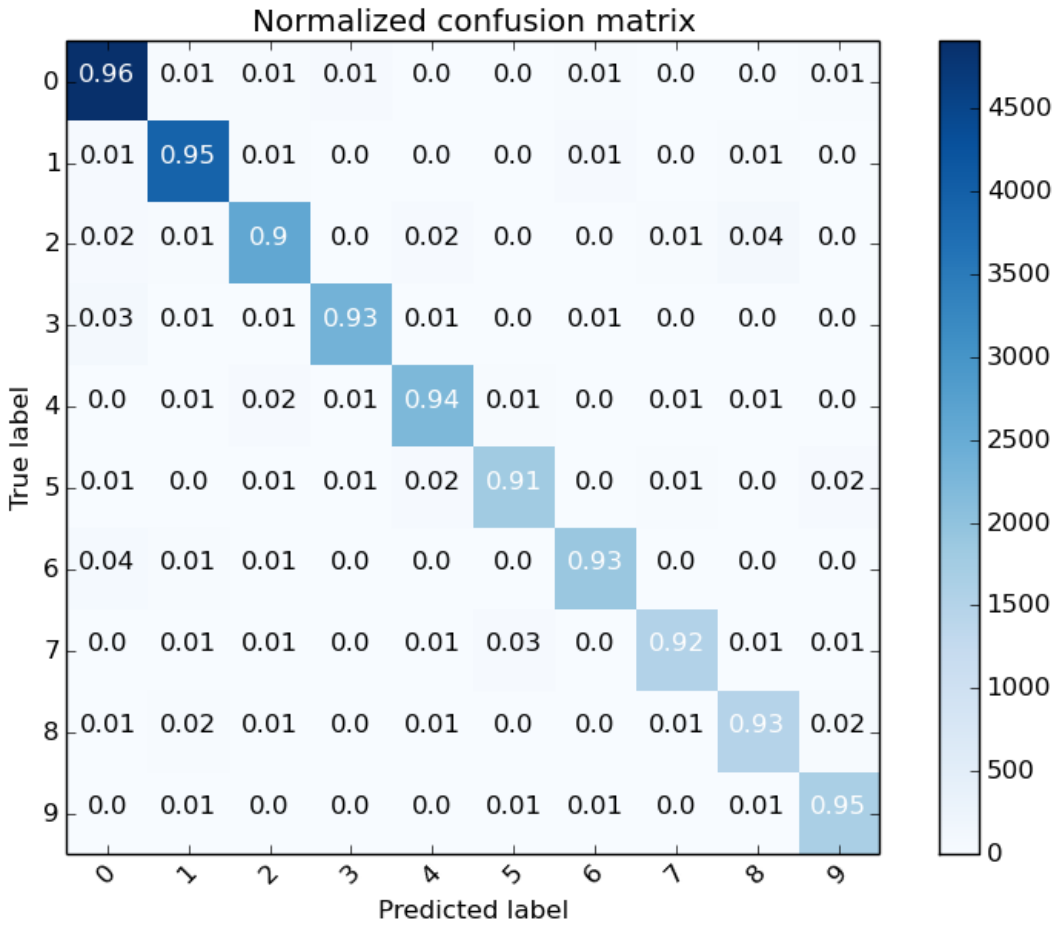

Figure 13 - Normalized confusion matrix

What do the confusion matrices tell us? First, since we see high rates along the diagonal they tell us that the model is classifying a high percentage of the examples correctly. They also show us a fairly even distribution among all of the classes, for both correct predictions and errors, which tells us that we are not too strongly biased. In the normalized matrix we seem to see a higher rate of correct predictions for bigger classes - classes which make up a higher percentage of the dataset - but since our lowest accuracy is for the digit "2" (3rd largest, see figure 2), and our second highest for digit "9" (9th largest), the trend towards higher accuracy for bigger classes does not hold. The shapes of different digits will naturally be easier or harder to recognize, which probably accounts for most of the difference in accuracy between classes.

Looking at errors between particular pairs of classes – mistaking a particular digit for another particular digit – we don't see any large rates of mistakes. The highest are for

mistaking a 6 for a 0 and mistaking a 2 for an 8, both of which have similar shapes, so those probably do not represent a training problem with the model.

Justification

The results of the Google team are 4.28% better than mine. While it's disappointing that my result is lower, if we follow Google's recommendation and apply the result with confidence thresholding it would still provide a reliable result for the vast majority of examples. Even the Google team recommends that their result should not be applied to 100% of examples, which is their reason for recommending confidence thresholding.

It's difficult to say exactly which aspects of the differences between my reshaped AlexNet and the Google design resulted in Google's higher accuracy. The Google team describes their architecture as:

> *"Our best architecture consists of eight convolutional hidden layers, one locally connected hidden layer, and two densely connected hidden layers. All connections are feedforward and go from one layer to the next (no skip connections). The first hidden layer contains maxout units (with three filters per unit) while the others contain rectifier units. The number of units at each spatial location in each layer is [48, 64, 128, 160] for the first four layers and 192 for all other locally connected layers. The fully connected layers contain 3,072 units each. Each convolutional layer includes max pooling and subtractive normalization. The max pooling window size is 2x2. The stride alternates between 2 and 1 at each layer, so that half of the layers don't reduce the spatial size of the representation. All convolutions use zero padding on the input to preserve representation size. The subtractive normalization operates on 3x3 windows and preserves representation size. All convolution kernels were of size 5x5. We trained with dropout applied to all hidden layers but not the input."* - *(Google 2013)*

It could be that the additional 3 convolutional layers increase the ability to recognize important features. It could also be the case that subtractive normalization is particularly effective for this data, or that maxout activation in the first hidden layer provides a significantly different output to the following layers.

It is often the case that optimizing deeper networks requires much trial and error, based on experience. As Bengio writes:

> *"Why would training deeper networks be more difficult? This is clearly still an open question. A plausible partial answer is that deeper networks are also more non-linear (since each layer composes more non-linearity on top of the previous ones), making gradient-based methods less efficient. It may also be that the number and structure*

of local minima both change qualitatively as we increase depth. Theoretical arguments support a potentially exponential gain in expressive power of deeper architectures and it would be plausible that with this added expressive power coming from the combinatorics of composed reuse of sub-functions could come a corresponding increase in the number (and possibly quality) of local minima. But the best ones could then also be more difficult to find." – [(Bengio 2012)]

Chances are that the Google team has more experience and more resources that I do to pursue iterative tuning of their model in order to improve their accuracy. Still, given my level of experience and resources, I feel pretty good about what I've been able to achieve.

V. Conclusion

Free-Form Visualization

Let's take a look again at some examples from the dataset and think about what these show us (figure 14):

Figure 14 - Sample images from SVHN dataset

This is computer vision in the real world, with data taken from the wild. Unlike a dataset such as MNIST there are significant distractions which complicate the task of recognizing objects like digits. This is a significant leap in complexity, but important

since we want to apply this technology to real world tasks and not simply lab studies. It's also the task that every human faces when we try to recognize objects in the real world using our eyes and brains. This is the reality for the task of recognizing digits, but also for a large range of other computer vision tasks including recognizing cars, road signs, household objects, people, etc. That makes this task more relevant and interesting, but also more challenging.

Of course humans also sometimes struggle to recognize characters. Figure 15 shows some examples of digits that my reshaped AlexNet recognized incorrectly:

 Predicted: 0 Actual: 1	 Predicted: 2 Actual: 4	 Predicted: 4 Actual: 9	 Predicted: 6 Actual: 8
 Predicted: 0 Actual: 4	 Predicted: 2 Actual: 6	 Predicted: 5 Actual: 1	 Predicted: 7 Actual: 0
 Predicted: 0 Actual: 5	 Predicted: 2 Actual: 9	 Predicted: 5 Actual: 2	 Predicted: 7 Actual: 1
 Predicted: 0 Actual: 6	 Predicted: 3 Actual: 4	 Predicted: 5 Actual: 3	 Predicted: 8 Actual: 3
 Predicted: 1 Actual: 2	 Predicted: 3 Actual: 7	 Predicted: 5 Actual: 8	 Predicted: 8 Actual: 6
 Predicted: 1 Actual: 3	 Predicted: 3 Actual: 9	 Predicted: 6 Actual: 0	 Predicted: 9 Actual: 3
 Predicted: 1 Actual: 4	 Predicted: 4 Actual: 1	 Predicted: 6 Actual: 3	 Predicted: 9 Actual: 4
 Predicted: 2 Actual: 1	 Predicted: 4 Actual: 2	 Predicted: 6 Actual: 5	 Predicted: 9 Actual: 8

Figure 15 - Examples of misclassed street numbers

Reflection

I wanted to get hands-on experience with a leading architecture for computer vision, and chose to work with AlexNet because of its place in history as a groundbreaking design. The SVHN dataset and the Google team's approach and result offered an ideal comparison to what I could do with AlexNet versus what they were able to do with greater resources and experience, and an architecture that was specifically designed for this dataset. I found the exercise and the comparison revealing in terms of trying to approach this kind of problem at this state-of-the-art level. While it is both daunting

and humbling to be comparing my work with the work of a highly experienced Google team, I enjoyed the challenge.

In the process I discovered and dealt with the problem of adapting a design for a dataset with one visual field to a different dataset with a very different visual field. During the reshaping process it felt that the design was being pushed to smaller filter sizes than I would normally prefer in order to try to preserve the character of the AlexNet design.

In the process of reshaping AlexNet I also realized that the number of features that need to be recognized for a dataset with 1,000 classes, like ImageNet, is significantly different than for a dataset with 10 classes, like SVHN. Decreasing the number of filters significantly decreased the resource requirements for training (633 seconds versus 1403 seconds, a 55% decrease), with only a small decrease in accuracy that could easily be random variation.

Building and using a Tensorflow GPU instance was a worthwhile exercise as well, but mostly in system integration. It was however interesting to experience the difference in time between training on my personal system (CPU only) and a fairly beefy GPU system.

It was fun to work with Tensorflow and the TFLearn framework, which significantly simplifies the Tensorflow development process. Understanding the operations, compute graph, and session structure of Tensorflow is essential, but using a higher level framework for most of the code is similar to the development processes for many other areas of software development. High level frameworks allow the developer to focus on the task at hand, and drill into key sections for more detailed control, but should be designed so that they do not conflict with or violate the underlying design concepts of the lower level frameworks that they sit on top of. TFLearn is one of the Tensorflow frameworks that appears to meet these requirements. Others include the tf.contrib.learn framework (formerly SKflow) and TensorLayer.

In the end, although I was not able to match the accuracy of the Google team's design, I feel pretty good about the level of accuracy that I was able to achieve. If we follow Google's recommendation and apply the result with confidence thresholding it would still provide a reliable result for the vast majority of examples, and could be used in production applications.

Improvement

There are a number of ways that this implementation could potentially be improved, starting with a different architecture. AlexNet was specifically designed for the

ImageNet dataset, a fact that I now have a new appreciation of, and is probably not the best architecture for the SVHN dataset.

Since the breakthrough work in 2006 that demonstrated the effectiveness of generative pre-training, using greedy layer-wise unsupervised approaches, I think an obvious candidate for an improvement would be adopting pre-training of the model on a large unlabeled dataset. This has been shown to often be very effective for reasons that are still being studied, but likely include shaping the decision surface to improve the process of gradient descent during the later supervised fine-tuning phase.

In addition, there are newer architectures that have shown impressive results on computer vision tasks, including Generative Adversarial Networks (GANs), Spatial Transformer Networks, and Deep Residual Networks. My guess is that these newer architectures may achieve better results than AlexNet on this task. It would be surprising if they didn't, since they are considered improvements of the state of the art.

References

See hyperlinks in text above. In addition, at the request of the Google team, the following reference is cited:

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*. (PDF)