6.867 Term Project: An Exploration of Deep Learning and Convolutional Neural Nets for Image Classification

Kathryn Evans, Andres Hasfura and Remy Mock

December 10, 2015

1 Introduction

Artificial Neural networks (ANN) are a useful machine learning framework. Their primary benefit is that instead of specifying the basis functions relating input to output, they are learned. This is especially useful for when the optimal basis function is complicated or non-intuitive, such as in the case of images.

Through this course, a simple singular layer artificial neural network was presented. However, much like the neural networks in the human brain visual system , the architectures used for neural networks can be more complex than a single hidden layer. The idea of utilizing many layers is known as deep learning. Increasing layers, allows for more complex input/output relationships and an ability to classify based on both information from low and high level features.

Deep learning, although not a recent idea, has recently exploded in popularity due to rise in labeled data and general purpose GPU programming and is revolutionizing very important subfields within artificial intelligence. Machine learning, machine vision, and natural language processing are examples of areas in which the use of deep learning has produced large jumps in performance on difficult test sets. Deep nets are now being used anywhere from pedestrian detection for autonomous vehicles [1], to facial expression recognition [2] to classifying whether or not a selfie is good [3].

Not only do deep nets contain more hidden layers but also a multitude of different types of layers. Each layer type has different connectivity and objective, allowing for a greater richness of information. In the case of images, convolutional layers are especially beneficial for examining spatially close pixels to determine features. Arrangements of convolutional layers as well as other types of layers leads to Convolutional Neural Networks (CNN).

In this project, we want to explore the benefits of deep nets as well as convolutional neural networks for image classification. Our first goal is implement our own multilayer neural networks and convolutional neural networks. From this we hope to gain insights on the benefit on image classification performance of firstly, additional fully connected hidden layers and secondly, more complicated layer types. Not only do we want to gain familiarity with the concepts and tools used in deep learning but we hope to benchmark our home made software on MNIST.

Our second goal is to attempt to replicate results from Yann LeCun on the classification of the MNIST data sets for more complicated frameworks like LeNet [4]. For this we will use a professional deep learning library to get faster results and be able to quickly build extensive architectures.

Thirdly, we hope to do some analysis on the results of multilayer architectures. The generalities we hope to explore include the following: How does the number of layers affect the amount of training data needed? How much variance is there in the output results due to random initialization/dropouts? How large of a dataset do we need to train on to be able to classify accurately? To quickly cycle through a number of different tests we will utilize TensorFlow, a professional software by Google.

This paper is organized as follows. First we will introduce fully connected and convolutional neural networks with an emphasis on types of layers and implementation. We will also describe the dataset which we use for testing our implementations, MNIST, as well as those networks we wish to replicate. Then we will describe our results for each of the three goals previously stated, starting with results from the implementation with software we built, then replication of LeCun's results using TensorFlow and lastly expanding to deeper analysis.

2 Methodology

In this project, we seek to implement a version of multi-layer neural network for supervised learning on images. We plan on trying our implementation on the MNIST dataset, so that we do not have to worry about data generation and labeling. Because we are specifically interested in applying our implementation to images, we also hope to see benefit in making the network out of convolution layers, where not all nodes are connected. We will compare this to a implementation with fully connected layers.

We will then compare our rudimentary approach to a professional library for deep learning, specifically the TensorFlow platform. Undoubtedly, the professional library will allow for better results and interesting conclusions. However, building and implementing a simplified version will give a better understanding into how the neural network works.

2.1 Division of Labor

For the project we split the work into three parts for each of the members. Kathryn Evans built and analyzed the CNN. Andres Hasfura led the implementation of TensorFlow and complex architectures for replication of results. Remy Mock built the fully connected multilayer neural nets. We worked collaboratively to do parameters searches and design experiments.

2.2 Neural Network Basics

In general, the idea of a neural network is to replace specific basis functions with learned basis functions. The basis functions are learned from the input data in the first layer of learning, and then utilized by a second layer which learns the relationship of the output data from those selected features.

$$a_j^{(1)} = \sum_{i=1}^{N} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{1}$$

$$z_j = g(a_j) \tag{2}$$

Where $g(x)$ is a nonlinear map of input to $(0, 1)$, generally tanh or sigmoid. The ac-

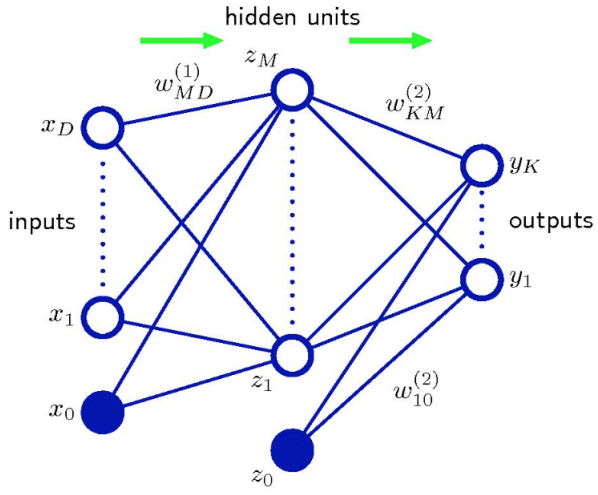


Figure 1: Visualization of a simple Neural Network with 1 hidden layer [5] .

tivation, a_j depends on the data, x_i, $i = \{1, ...N\}$, learned weights $w^{(1)}_{ji}$ and bias $w^{(1)}_0$.

The second layer learns in a similar fashion and can be described for K output classes and M features in the hidden layer as

$$a^{(2)}_k = \sum_{j=1}^{M} w^{(2)}_{jk} z_j + w^{(2)}_{k0} \tag{3}$$

$$f_k = \tilde{g}(a^{(2)}_k) \tag{4}$$

where $\tilde{g}(x)$ is the another nonlinear map, not necessarily the same as $g(x)$ and f_k is really our prediction $h_k(x, w)$.

2.2.1 Back-propagation

To find the optimum weights for the network, the back-propagation algorithm is used. Back-propagation uses the errors calculated in the forward propagation to calculate activations to calculate the necessary gradients with respect to the cost function J. The specific error function J we will attempt to minimize is

$$J(w) = l(w) + \sum_{i=1}^{N} \lambda_i (\|w_i\|^2_F); \tag{5}$$

Where λ is a regularization term to prevent overfitting and $l(w)$ is the negative loglikelihood function,

$$l(w) = \sum_{i=1}^{N} \sum_{k=1}^{K} [-y^i_k log(h_k(x_i, w)) - (1 - y^i_k) log(1 - h_k(x_i, w))] \tag{6}$$

Stochastic gradient descent is then used to update the weights with the following equation,

$$w^{(t+1)} = w^{(t)} + \eta_t \nabla_w J(w^{(t)}) \tag{7}$$

where η is the learning rate and the choice of that parameter varies as it is to be chosen to optimize results for each particular model on the data-set.

2.3 Multi-layer Neural Network

For each fully connected layer the forward propagation is straight forward, as you simply chain the formula for a single hidden layer together for more layers. For backwards propagation, the error for each fully connected layer, l is computed from the error of its output layer, $l + 1$

$$\delta^{(l)} = ((W^{(l)})^T \delta^{(l+1)}) \cdot f'(z^{(l)}) \tag{8}$$

Which create the the following derivatives for weights and biases,

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \tag{9}$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \tag{10}$$

the weights are then adjusted accordingly in the gradient descent.

2.3.1 Implementation

Using MATLAB, we have implemented the Multi-layer Neural Network. It is soft-coded so the hidden layers and number of hidden nodes per layer can easily be altered. The function created cells for the weights of each layers and initialized the weights randomly between -1 and 1.
A sigmoid function shown below is used as the activation function for all layers.

$$\tilde{g}(x) = g(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

A stochastic gradient descent approach was used with the error function shown in equation 5 and a step size η shown below.

$$\eta = \frac{\alpha}{(\text{epoch} + \beta)^{\gamma}} \tag{12}$$

where $\beta \leq 100$, $\gamma > .5$ and epoch is the number of times the descent method has gone through the training set.

2.4 General Architecture of a Convolutional Neural Network

As opposed to a fully connected neural network, a convolutional neural network is specifically designed to take advantage of the 2-D structure of an input image by considering only local connections. It also takes advantage of tied weights which reduces the number of decision parameters, as well as pooling which results in translation invariant features. All these advantages, make CNN generally easier to train.

2.4.1 Convolutional Layer

In a CNN, filters are learned to be used to identify key characteristics of a certain classification. These filters are convolved over subspaces of the original image dictated by the filter size, that learn features of these small segments of the larger image. This transforms the 2-D images into a 3-D space.

For a $M * N$ size image, using K filters that are $m * n$, after convolution, there are $K*(M-m+1)*(N-n+1)$ sub images. Using this method, we decrease the size of each image, but at the same time learn low level features. This is described for each image, x_n for $n = 1..N$ and for each filter $k = 1...K$ in the convolution layer, c as

$$a_{n,k}^{(c)} = x_n * w_k^{(c)} + b_k^{(c)} \tag{13}$$

$$f_{n,k} = \tilde{g}(a_{n,k}^{(c)}) \tag{14}$$

Where $*$ is the convolution operator.

The number of filters directly controls capacity and is chosen based on the number of available examples and the complexity of the task. The challenge is then to identify the correct filter size in order to obtain reasonable granularity of features, given a particular dataset.

Also CNNs introduce the idea of tied weights, which mean that certain connections affiliated with a particular filter between one layer to the next are defined to be the same. The use of tied weights dramatically reduces the number of parameters needed to define network compared to a fully connected network of similar size. This idea is shown in Fig. 2.

2.4.2 Pooling Layer

The features learned using convolution are next used for classification. Previously with

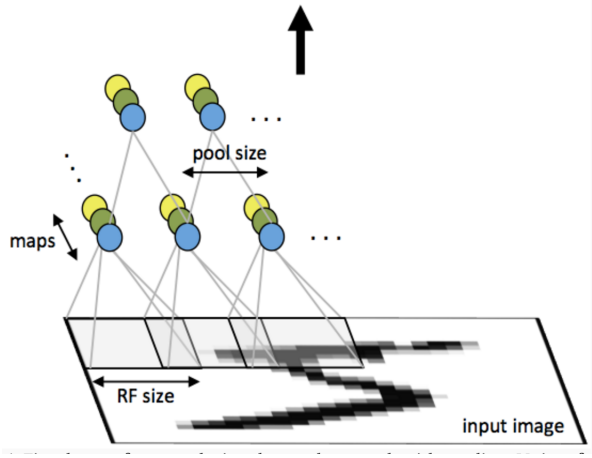


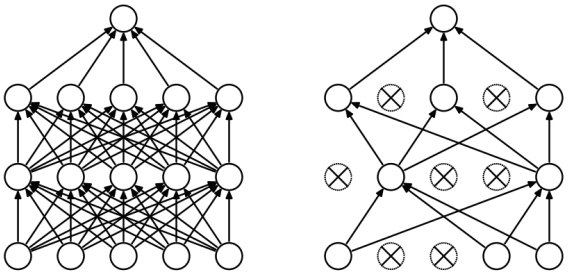
Figure 2: Layer of a convolutional neural network with pooling. Nodes of the same color have tied weights and units of different color represent different filter maps [6].

the fully connected neural network we utilized a softmax classifier, but this can be computationally challenging and expensive. Instead, pooling layers take advantage of a single parameter of these features, such as the mean, ('mean pooling') or max ('max pooling'), at various regions in the image. Not only does this reduce the dimensionality of the result but can also help reduce over-fitting, a problem that is of serious concern for networks with increasing numbers of hidden nodes.

The region to be pooled over is determined by the pooling dimensions. It again involves a convolution, this time between regions specified by the pooling dimensions and the results from the previous convolution layer.

2.4.3 Dropout

Another common trick used in CNNs is the inclusion of drop-out layers which help to decrease the likelihood of over-fitting. It achieves this by essentially thinning out the number of hidden nodes by randomly dropping units during training. A depiction of



(a) Standard Neural Net (b) After applying dropout.

Figure 3: Effect of dropout on a previously fully connected neural network. Dropped nodes are depicted as crossed out [7].

this effect is shown in Fig. 3. However, during test time all nodes are present.

The simplest way to implement drop-out is to keep each node with probability, p, and otherwise discard it. The key benefit is that dropout is able to outperform regularization in neural networks for reducing over-fitting [7]. Dropout also has the added benefit of being able to evaluate exponentially many different neural network architectures depending on which nodes are dropped.

2.4.4 Re-LU

As we already mentioned, a major flaw of deep nets is that they can be computationally expensive. One of the main tools to minimize this is a smart selection of the activation function, g.

Specifically using a rectified linear unit, (Re-LU) as opposed to more traditional sigmoids or tanh, can be beneficial in increasing the efficiency of a neural network [8]. The Re-LU function is simple,

$$g(x) = \max(0, x) \tag{15}$$

This function is not smooth but because the computation of g is simply a comparison,

it is generally much faster than smooth approximates. It also allows for sparse activation of the neural network, eliminating nodes with negative activations.

2.4.5 Backprop for CNN

For fully connected layers the derivative calculation remains the same. However, error propagations are more complicated through the pooling and convolutional layers.

For the pooling layer, the error must be up-sampled to propagate backwards

$$\delta_k^{(l)} = \text{upsample}((W_k^{(l)})^T \delta_k^{(l+1)}) \cdot f'(z_k^{(l)}) \quad (16)$$

For mean pooling, with pool dimension, p, this is equivalent to

$$\delta^{(l)} = \frac{1}{p^2} \text{kron}(\delta^{(l+1)}, \text{ones}(p)) \quad (17)$$

where kron, is the Kroneckor Tensor Product.

Thus for the convolution layer, the derivatives can be calculated as follows.

$$\nabla_{W^{(l)}} J(W, b; x, y) = \sum_{i=1}^{m} (a_i^{(l)}) * \delta_k^{(l+1)} \quad (18)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b} \quad (19)$$

2.5 Professional Libraries

Since deep learning has become increasingly popular, so too has the optimized professional libraries that allow students, researchers and hobbyists to easily build complex deep nets. . There are many of these libraries, but we chose to adopt TensorFlow, the open source software from Google [9]. We especially like TensorFlow because it is python based, has abundant tutorials, and is easy and intuitive to use. In addition, is uses an interface with Cuda for GPU processing to speed up results for image computations. [10].

Figure 4: Sample entries of the MNIST dataset [4]

2.6 Benchmarking with MNIST

The MNIST is a well-known dataset of handwritten digits, 0-9, comprising 60,000 training examples and 10,000 test examples [11] that can be used for image classification. This data set arose from the U.S. Postal Service zip code database in order to help the scanning and transport of package to the right area. The images are all centered in 28 x 28.

Since the data-set has already been thoroughly explored by many people including Yann LeCun [4], it is a good data set on which to measure our success. Replication of the architecture of both the fully-connected and convolutional multi-layer neural nets, (LeNet) provided in this paper is the one of the objective of our study.

2.6.1 LeNets

The LeNet framework was developed by LeCun beginning in the 1980's. They provide a few examples of more complex CNN architecture on which we will compare our performance. The CNN LeNet-1 architecture consists of 5 layers, which are denoted by C1, S2, C3, S4, and F5. The output of the first layer serves as input to the next layer. C1 and C3 are convolution layers, and S2 and S4 are sub-sampling layers and the fifth layer is fully connected. [12]

The CNN LeNet-4 architecture consists of 6 layers, which adds to LeNet-1 an additional hidden layer that is fully connected to its input and output layers [13].

LeNet-5 architecture expands on this even further by adding an additional layer, with gaussian connections. More details on the architecture chosen, including filter size number and pooling dimension are shown in Fig. 5

3 Results

We start by benchmarking our handmade multilayer and convolutional network on the MNIST dataset, and provide discussion about some of the challenges and difficulties behind building these deep nets. We then replicate the remaining deeper architectures presented by LeCun, but for these we use TensorFlow. We did not have access to GPU clusters, so we found it necessary to use TensorFlow's more computationally efficient implementations. Finally, we provide further analysis of the deep architectures, further illuminating characteristics of deep neural networks.

3.1 Multilayer NN

The initial goal was to alter the two-layer neural network done for homework 3 to have the functionality to soft-code the number of hidden layers and the number of nodes per hidden layers.

Once this was done, a grid search was done to find the optimal parameters. Due to the fact that all results documented by others have been on the three-layer model (using two hidden layers), we decided to focus on the three-layer model. The parameters optimized include the number of hidden nodes and regularization λ_1 ,λ_2 and λ_3 for layers 1, 2 and 3 respectively.

For the architecture with 300 hidden nodes and 100 hidden nodes, we obtained the best testing accuracy of 6.2% when λ_1 ,λ_2 and λ_3 were 10^{-3}, 10^{-8} and 10^{-5}. The testing error was extremely sensitive to the regularization parameters, and even changing one of the parameters slightly increased the testing error significantly.

3.2 Simple CNN

We began by building the most basic form of a convolutional neural network, which starts with a convolution layer, followed by one pooling layer, and finished with a fully connected layer to the output. This net was then trained and tested on the MNIST dataset to ensure proper implementation and extract performance and computation time metrics.

As we discovered with neural nets, the devil is in the details. It is very difficult to find the optimal parameters for CNNs because of the high dimensional optimization space and the lack of generalization between architectures. Finding optimal parameters for a particular net says very little about the best parameters for another architecture. The space is

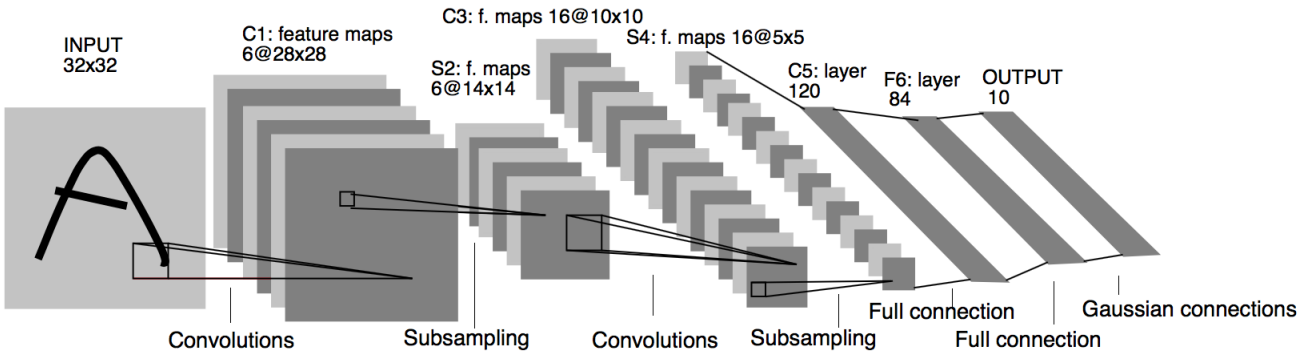


Figure 5: Architecture for LeNet5 [4]

too high dimensional for grid search and also highly non-convex, eliminating any guarantees for gradient descent based methods.

After much research and trail-and-error, we converged on a set of parameters which achieved 2.7% on the MNIST dataset. The optimal combination for our architecture was 20 filters, 9 x 9 convolution kernels, and 2 x 2 pooling dimensions. We were only able to reach 2.7 error after the third epoch, after only one pass through the training data the net still produced 4.6% error.

3.3 Replication of NNs and LeNets Results

We attempted to replicate the published results for testing error on MNIST for the network structures specified in LeCun's paper, [4]. Specifically we designed

- 2 layer fully connected with 1000 hidden nodes
- 3 layer fully connected with 300 then 100 hidden nodes
- 3 layer fully connected with 500, then 150 hidden nodes
- 5 layer convolutional network (based on LeNet-1)
- 6 layer convolutional network (based on LeNet-4)
- 7 layer convolutional network (based on LeNet-5)

Although the LeNets come with specified filter numbers, sizes, and pooling sizes, there remained many parameters that were not given which make it non-trivial to replicate. These parameters included number of iterations, learning rates, batch size, dropout, pooling method, weight initializations, and others, all of which are not included clearly in the published results. Because results are so sensitive to the entire combination of parameters, knowing the few provided was not sufficient to reproducing his results. Therefore, we maintained the same network architecture, but did not restrict ourselves to only using the parameters specified in LeCun's paper. Instead, we were able to closely replicate the results using our own combination of the system's parameters.

Our 5 layer CNN, which was modeled after LeCun's LeNet1, is constructed as follows.

NN Type	LeCun Error	Tensor Flow Error
2-layer NN, 1000 hidden units	4.5	5.8
3-layer NN, 300+100 hidden units	3.05	8.2
3-layer NN, 500+150 hidden units	2.95	5.6
5 layer CNN (LeNet-1)	1.7	3.1
6 layerCNN (LeNet-4)	1.1	1.6
7 layer CNN (LeNet-5)	0.95	2.3

Table 1: Comparison of Test Error results for multilayer ANN and CNN with published results [4].

Input images are fed through two convolution and pooling layer combinations in series, and finish with a fully connected layer. To implement our 6 layer CNN, modeled after LeCun's LeNet4, we simply added another fully connected layer to the tail, and similarly appended another fully connected layer to LeNet4 for our 7 layer LeNet5 emulation.

In general we fixed certain parameters to make our search simpler. For any convolutional layer we fixed filter size to be 5x5 pixels and pooling size to be 2x2 pixels. We found that the first convolution worked well with with 32 filters and second with 64. We left all fully connected layers with 1024 nodes.

Also we understand that LeCun cites convergence in testing error after 10-12 times the number of training data in iterations for LeNet-5 [4]. However, since it takes greater than 4 minutes to run each of the multilayer CNNs for 1 epoch and we were more interested in experimenting with a variety of different parameters, we capped the iterations at 100,000. Obviously these will not be as successful as runs that were carried out for an order of magnitude more iterations and therefore we do not expect to match Lecun's results perfectly. Our best results are shown in Table 1.

3.4 Variance of Results

In Table 1, we include only the maximum performance on the test data. However, this may not be a great metric since we found with our results, that when the data-set on the exact same network with exactly the same parameters with the randomized initial weight, we get a shockingly large range of testing errors. This is due to the weight initialization and drop outs that happen in a random manner. We ran each of the nets list twenty times to achieve the following results Table 2.

We reran two specific networks, the 2 layer fully connected and the 6 layer CNN, an additional 30 times and made histograms of the results so that the scatter could be visualized as well. These can be seen in Fig. 6 and Fig. 7 respectively.

The results show that the is low variance in the 2-layer fully connected neural network but high variance in both of the 3 layer fully connected networks. High variance could be indicative that many more iterations through the training data would be needed to converge to steady state weights and therefore a consistent test error. It could also be a sign that the cost space for that particular network is riddled with more local optima.

NN Type	Mean error	Variance
2-layer NN, 1000 hidden units	6.8	0.63
3-layer NN, 300+100 hidden units	11.4	3.9
3-layer NN, 500+150 hidden units	8.9	3.4
5 layer CNN (LeNet-1)	3.8	1.07
6 layerCNN (LeNet-4)	3.7	1.01
7 layer CNN (LeNet-5)	3.9	1.3

Table 2: Comparison of Mean Accuracy and Variance results for multilayer ANN and CNN

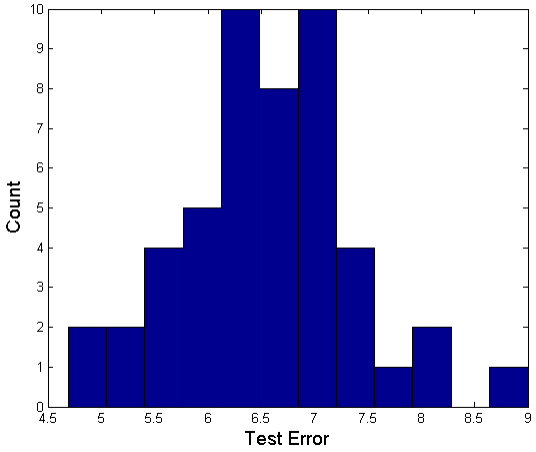


Figure 6: Histogram for Test Error for the 2 layer 1000 hidden node network

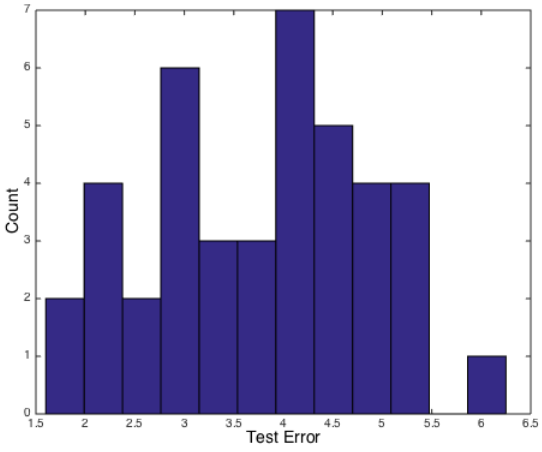


Figure 7: Histogram for Test Error for the 6 layer CNN

3.5 Effect of Training Data Size

One of the other main areas we wanted to investigate was how the relationship between amount of training data affects and testing error, changes for increasingly large networks. We decided to investigate this by running the 5 and 7 layer CNNs with the same parameters and iterations and gradually increase the percentage of the MNIST data we give as an input. These results are shown in Fig. 8.

With the same number of iterations but fewer examples, a simple way to guess the size of dataset needed comes from mathematics. We need at least the same number of equation and variables to solve a multi-variable equation. Since the number of unknown weights can be calculated(see appendix), we expect the number of training data needed T is proportional to this number. We expect that the network does a increasingly better job at classifying until it hits T and plateaus after this number.

In these plotted figures we see that as expected the networks ability to classify correctly increases as it is given more data-sets to train on. However the learning curve was not as steep as expected. Even with only 256 data-sets the network was able to per-

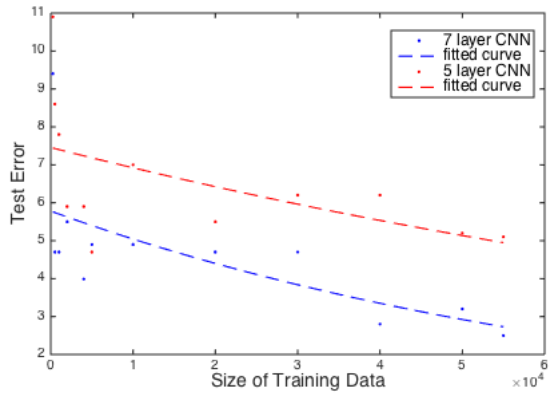


Figure 8: Effect of Training Data Size for 5 and 7 layer CNN

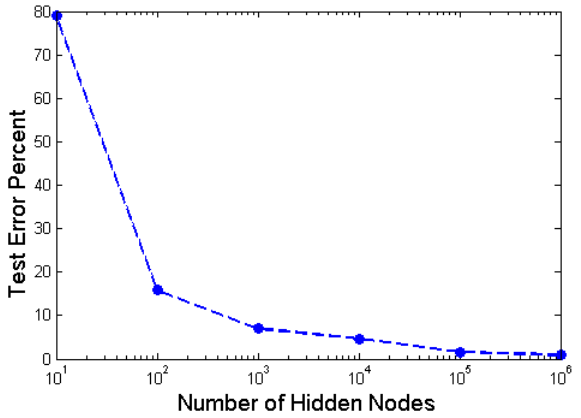


Figure 9: Effect of number of hidden nodes on a two layer model

form with only a 10.4% test error.

3.6 Comparison to a two-layer model

According to the Universal Approximation theorem, in a compact and continuous data-set the two-layer neural network can perform as well as any multi-layer neural network given enough hidden nodes. We decided to see if this theory can be applied to our data-set by varying the number of hidden nodes and measuring the test error on a two layer model.

Looking at the figure 8, it is evident that the networks performs better with increasing the amount of hidden nodes. However training takes increasingly longer since the network is linearly proportional to the number of hidden parameters which is proportionally to the number of hidden nodes (see appendix). Furthermore once the number of parameters is on the order of the number of data-sets we may see no additional benefit of increasing the number of nodes. This is due to the same reason discussed in the section before regarding the solving a multi-variable equation.

4 Conclusions

We implemented the multi-layer neural network and convolutional neural network and optimized them for varying parameters. The best test errors obtained on the networks we implemented were 6.2% and 2.7% for the multi-layer network and convolutional network respectively. We then replicated these networks using TensorFlow, compared our results with Lecun's results and did some further analysis to gain deeper insight. Using TensorFlow, we obtained our best result with a test error of merely 1.6% using the network architecture that corresponds with LeNet-4. Considering that our best results were from deep networks and not from two-layer model, it is evident to us that deep neural networks are superior to the simple two-layer model. Deep neural networks are a powerful tool and it is no surprise that deep learning is increasing in popularity.

References

[1] M. Szarvas, U. Sakai, and J Ogata. Real-time pedestrian detection using lidar and

convolutional neural networks. *Intelligent Vehicles Symposium, 2006 IEEE*, pages 213–218, 2006.

[2] Wei Li, Min Li, Zhong Su, and Zhigang Zhu. A deep-learning approach to facial expression recognition with candid images. *Machine Vision Applications (MVA), 2015 14th IAPR International Conference on*, pages 279–282, May 2015.

[3] Andrej Karpathy. What a deep neural network thinks about your selfie. *http://karpathy.github.io/2015/10/25/selfie/*, 2015.

[4] Y. Bengio Y. LeCun, L. Bottou and P. Haffne. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[5] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 1 edition, 2006.

[6] Stanford Unsupervised Feature Learning Deep Learning Tutorial. Convolutional neural netowrk. *http://ufldl.stanford.edu/tutorial /supervised/ConvolutionalNeuralNetwork/*.

[7] Krizhevsky Sutskever N. Srivastava, Hinton and Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, June 2014.

[8] G. Hinton A. Krizhevsky, I. Sutskever. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 2012.

[9] Google. Tensorflow. *https://github.com/tensorflow/tensorflow.git*, 2015.

[10] Google Brain Team. About tensor flow. *https://www.tensorflow.org/*, 2015.

[11] C.J.C. Burges Y. LeCunn, C. Cortes. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/index.html*.

[12] J.S. Denker D. Henderson R.E. Howard Y. Lecun, B. Boser. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[13] L. Bottou A. Brunot C. Cortes J. S. Denker H. Drucker I. Guyon U. A. Muller E. Sackinger P. Simard Y. LeCun, L. D. Jackel and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. *International Conference on Artificial Neural Networks*, 53-60, 1995.

5 Appendix

5.1 Calculation of the number of Parameters

Calculating the number of parameters of the two layer network with 1000 hidden nodes,

$$\text{params} = 784 * 1000 + 1000 * 10 = 794000$$

and in the case of the three layer model with 300 and 100 hidden nodes,

$$\text{params} = 784*300+300*100+100*10 = 266200$$

In regards to time complexity, it is evident that the bottleneck of the function is back-propagation. The back-propagation needs to

calculate the δ for each parameter. Therefore the time complexity of the training a network is related to the number of parameters.