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, the basis function is 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 which provided the inspiration for artificial neural networks, the architecture used for machine learning neural networks can be more complicated and 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 relationship 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 area 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 a multitude of different types of layers. Each layer type has different connectivity and objectives allowing for a greater richness of information. In the case of images, convolutional layers are especially beneficial for examining only the relevance of spatially nearby pixels in the context of determining 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 convolutional layers and architectures. Not only

1

do we want to gain familiarity with the concept 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 the professional deep learning library to get faster results and be able to quickly build extensive architectures.

Thirdly, we hope to do some analysis of 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 professional software here as well.

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. In the approach section we will also describe the dataset, MNIST, as well as those networks we wish to replicate. Then we will describe our results starting with results from the simplest three-layer convolutional neural network and a similarly sized fully connected network to explore benefits and extract generalities about design and parameter choices to deepen our understanding and intuition behind the network. We will then introduce the professional libraries and the extended architectures see how close to the published results we can achieve on MNIST. Lastly we will do some further analysis on the networks and summarize our findings and successes.

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 of making the network out of convolution layers, where not all nodes are connected and 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 to learn the relationship of the output data from those selected features.

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

$$z_j = g(a_j) \qquad (2)$$

Where $g(x)$ is a nonlinear map of input to $(0,1)$, generally tanh or sigmoid, and a_j is an activation, which depends on the data, x_i, $i = \{1, ...N\}$, learned weights $w_{ji}^{(1)}$ and bias $w_0^{(1)}$.

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

$$a_k^{(2)} = \sum_{j=1}^{M} w_{jk}^{(2)} z_j + w_{k0}^{(2)} \qquad (3)$$

$$f_k = \tilde{g}(a_k^{(2)}) \qquad (4)$$

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

As the network grows in size, the computational time becomes more and more of a constraint. So for all studied neural networks we will use stochastic gradient descent as opposed to batch methods in order to find the optimal weights for each layer.

$$w^{(t+1)} = w^{(t)} + \eta_t \nabla_w J(w^{(t)}) \qquad (5)$$

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.

2.2.1 Backpropagation

To perform stochastic gradient descent on the neural net, the back-propagation algorithm is
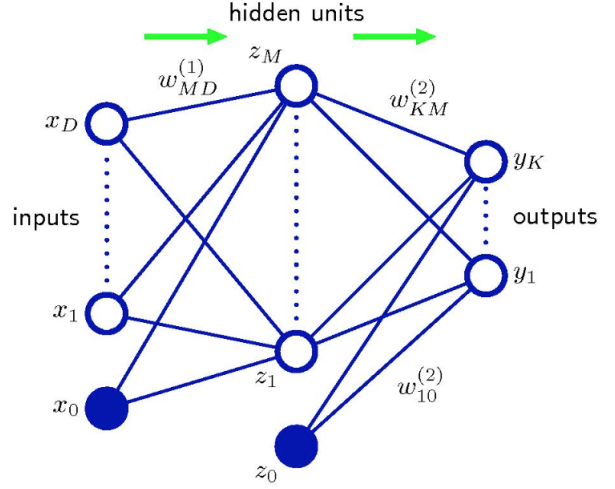

Figure 1: Visualization of a simple Neural Network with 1 hidden layer [5] .

used. The back-propagation uses the errors calculated in the forward propagation to calculate activations to calculate the necessary changed needed for the weights in the form of gradients.

2.3 Multi-layer Neural Network

For each fully connected layer forward propagation is straight forward, as you simply continue the formula for a single hidden layer and chain them together for more layers.

The error function $J(w)$ we will attempt to minimize is

$$J(w) = l(w) + \sum_{i=1}^{N} \lambda_i(\|w_i\|_F^2); \qquad (6)$$

where $l(w)$ is the negative loglikelihood function,

$$l(w) = \sum_{i=1}^{N} \sum_{k=1}^{K} [-y_k^i log(h_k(x_i, w)) - (1 - y_k^i) log(1 - h_k(x_i, w))] \qquad (7)$$

3

and

For backwards propagation, the error for each fully connected layer, l is computed from the error of its output layer, $l + 1$

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

and the following derivatives for weights and biases,

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

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

the weights are then adjusted in the gradient descent

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \qquad (11)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \qquad (12)$$

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.

$$\tilde{g}(x) = g(x) = \frac{1}{1 + e^{-x}} \qquad (13)$$

A stochastic gradient descent approach was used with a step size η shown below.

$$\eta = \frac{\alpha}{(\text{epoch} + \beta)^\gamma} \qquad (14)$$

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

To prevent over-fitting we used a regularization parameter λ.

2.4 General Architecture of a Convolutional Neural Network

As opposed to a simple 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 and tied weights followed by some form of pooling which results in translation invariant features. Compared to regular neural nets, CNNs have many fewer parameters than networks of similar size and thus they are easier to train.

2.4.1 Convolutional Layer

The idea of the CNN is to no longer have fully connected layers. In a CNN, the relationship between two nodes are described by a kernel, ie. the convolutional kernel. But instead of hard coding the filters, the filters used are learned by training.

By sub-sampling the image and having learned features over smaller segments of the larger image. The feature detector learning from the subsection can then be applied anywhere in the remaining image. Specifically, we can take the learned subsection features and convolve them with the larger image there by transforming 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 we learn low level features.

Generally, the process that occurs in a convolution layer can be described as for each image, x_n for $n = 1..N$ and for each filter $k = 1...K$ in the convolution layer, c

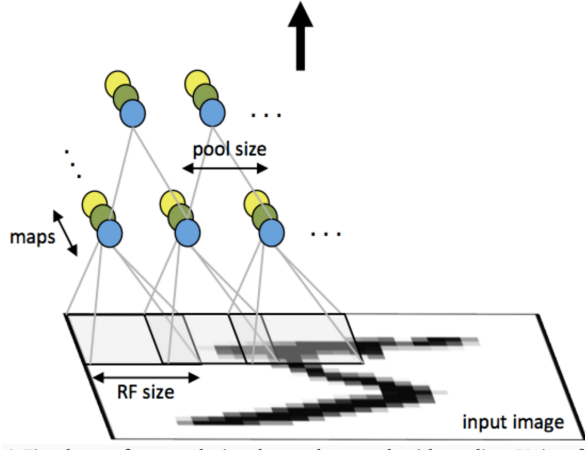$$a_{n,k}^{(c)} = x_n * w_k^{(c)} + b_k^{(c)} \qquad (15)$$

4

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].

$$f_{n,k} = \tilde{g}(a_{n,k}^{(c)}) \qquad (16)$$

Where $*$ is the convolution operator.

The number of feature maps directly controls capacity and so that depends 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 features, given a particular dataset.

Also CNNs introduce the idea of tied weights, which mean that certain connections between one layer to the next are defined to be the same. The tied weights dramatically reduces the number of parameters needed to describe and set up the 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 the fully connected neural network we utilized a softmax classifier, but this can be computationally challenging and expensive. Instead,

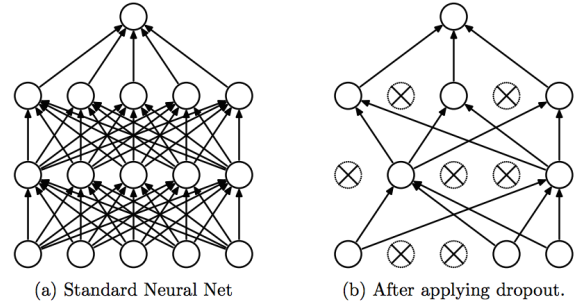(a) Standard Neural Net   (b) After applying dropout.

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

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 CNN 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 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

Yet another problem with deep nets is that they are 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,

$$g(x) = \max(0, x) \qquad (17)$$

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.

2.5 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

$$\delta_k^{(l)} = \text{upsample}((W_k^{(l)})^T \delta_k^{(l+1)}) \cdot f'(z_k^{(l)}) \quad (18)$$

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

$$\delta^{(l)} = \frac{1}{\text{p}^2} \text{kron}(\delta^{(l+1)}, \text{ones}(p)) \qquad (19)$$

where kron, is the Kroneckor Tensor Product.

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

$$\nabla_{W^{(l)}} J(W, b; x, y) = \sum_{i=1}^{m} (a_i^{(l)}) * \text{rot90}(\delta_k^{(l+1)}, 2)$$

$$(20)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b} \qquad (21)$$

2.6 Professional Libraries

Since deep learning has become increasingly popular, optimized professional libraries that allow students, researchers and hobbyists to easily build complex deep nets are becoming easily accessible. 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, 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].

2.7 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 on by many people including Yann LeCun [4], it provides a good system to check our results. Replication of the architecture of both the fully-connected and convolutional multi-layer neural nets provided in this paper is the first objective of our study. By

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

comparing our results for different layers and types of layers to published and documented results, we can make sure our implementation is working correctly before taking on a more complicated dataset.

2.7.1 LeNets

The CNN LeNet-1 architecture created by LeCun, 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 ar-

chitecture chosen, including filter size number and pooling dimension are shown in Fig. 5

We specifically chose to replicate the 2 and 3 layer fully connected neural nets as well as LeNet-1, -4 and -5 with no preprocessing or distortions.

3 Results

We start by examining the effects of multiple layers and convolutional network layers to improve performance on MNIST using networks that we made. We will try to quantify accuracy gains versus computation time for different type of networks of relatively small size. To test bigger networks and to replicate LeCun's results we will transition to utilizing TensorFlow to classify MNIST images.

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
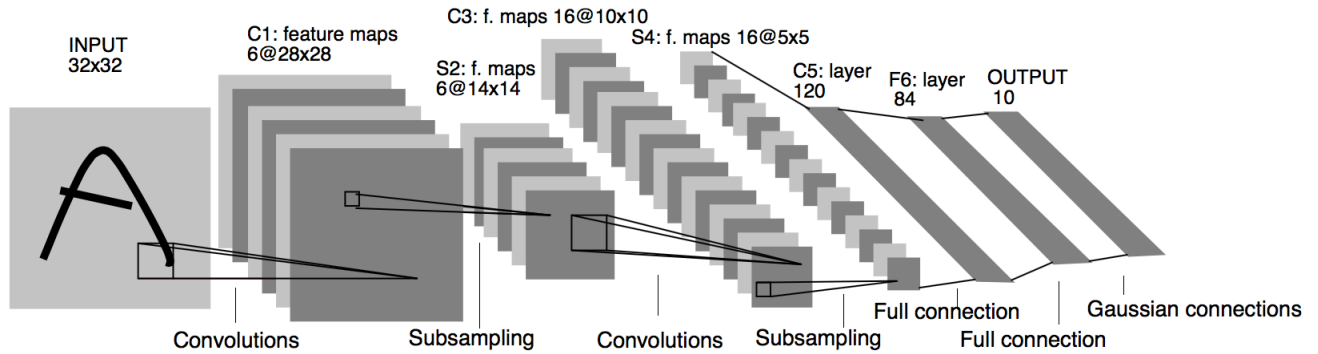
Figure 5: Architecture for LeNet5 [4]

significantly.

3.2 Simple CNN

The first objective was to build the simplest CNN, comprising of one convolutional layer, one pooling layer and one fully connected layer and implement it on the full MNIST dataset to ensure that it works and to test some simply features of performance and computation time.

Our process was to find a set of parameters that produced good results. We understand that it would be difficult to examine effects of individual parameters. Since there are a lot of different variables and the parameter space is large it would be to find true optima or fully generalizable statements about individual parameters with out investing more time than provided for this project. Since effects of one parameter around this local point could not be representative of the effect of that variable in general on a global scale or on other architectures.

The optimal combination for this simple three layer CNN was found to be with 20, 9 x 9 filters and 2 x 2 mean pooling, and was found to have only a 2.7 test error. However this was our best performance. With that same network and parameter we actually achieved errors varying from 4.6 to 2.7 percent error. Because of computation time we only ran through 3 epochs of training data but so we could expect even higher performance with more time and computational power.

For bigger CNNs we will turn to TensorFlow for faster performance.

3.3 Replication of NNs and LeNets Results

We hoped to replicate published results for testing error on MNIST for a variety of different network structures from LeCun, [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 (reminiscent of LeNet-1)

8

- 6 layer convolutional network (reminiscent of LeNet-4)

- 7 layer convolutional network (reminiscent of LeNet-5)

Although the LeNets come with specified filter numbers, sizes, and pooling sizes, there are still a lot of other parameters that were not given which make it non-trivial to replicate including, iterations, learning rates, batch size, dropout, pooling method, weight initializations bounds which are not included clearly in the published results. Because results are so sensitive to the whole combination of parameters, knowing the few Therefore we will attempt to replicate the results from these networks by keeping the same number and types of layers but we will not restrict ourselves to exactly the specifications of the LeNets. We don't believe the combinations of parameters we find to give the best results to be a global optima since we did not do a particularly thorough search over the parameter space.

For our 5 layer CNN we restricted ourselves to be comprised of a convolutional layer followed by pooling layer followed by another set of convolutional and pooling layers followed by a fully connections to the output layer. For the 6 layer, we simply add an additional fully connected layer and add another fully connected for the 7 layer CNN.

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
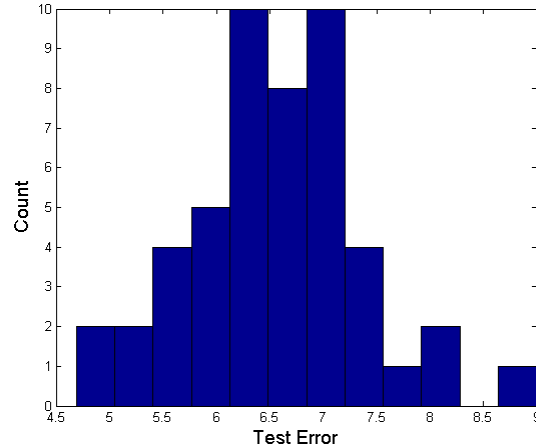

Figure 6: Histogram for Test Error for the 2 layer 1000 hidden node network

LeNet-5 [4]. However, since it takes greater than 4 minutes to run each of the multilayer CNNs for 1 epoch and we wanted to try 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.

9

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].

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

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. 7.

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

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(shown in next section/appendix?). We expect the number of training 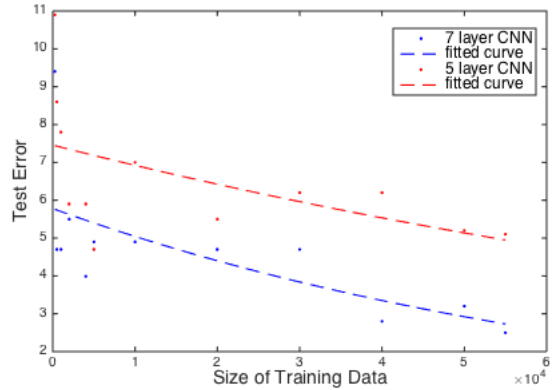data needed T is proportional to this number. We expect that the network does a increasingly better job at classifying until it hits achieves T and plateaus after this number.

In these plotted figures we have

3.6 Benefits of Multi-layer vs Single-layer

According to the Universal Approximation theorem, in a compact and continuous dataset the two-layer neural network can perform as well as any multi-layer neural network given enough hidden nodes. Since in our tests the two layer network does not perform as well as the three layer network it is an interesting question to ask how many nodes we would need. 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,

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

It is evident that the three-layer network outperforms the two-layer network with less parameters.

In regards to time complexity, it is evident that the bottleneck of the function is back-propagation. Since the complexity back-propagation is related to the number of parameters it is also evident that the three-layer method has a lower time complexity.

It is clear from the two factors above that multi-layer is more efficient and is no surprise that deep learning is increasing in popularity.

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. Le-Cun, L. D. Jackel and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. *International Conference on Artificial Neural Networks*, 53-60, 1995.