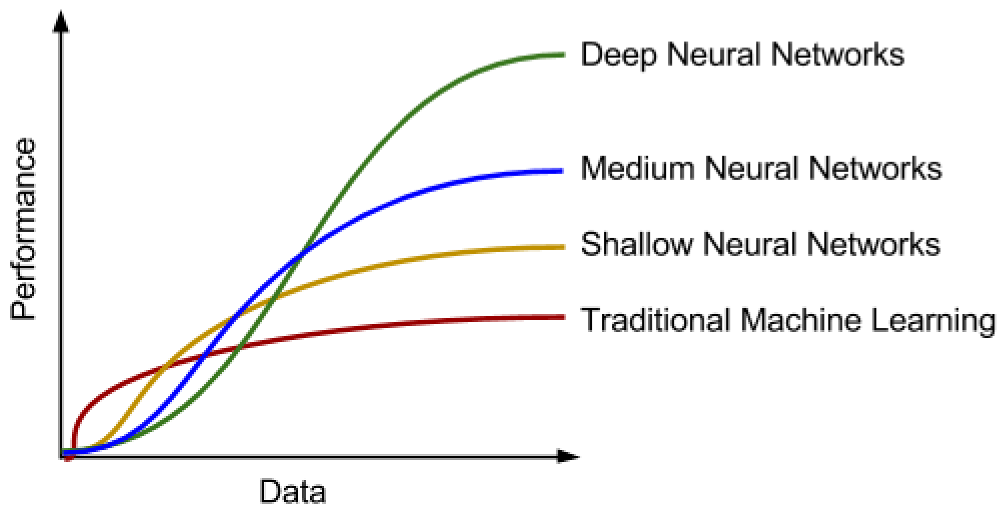
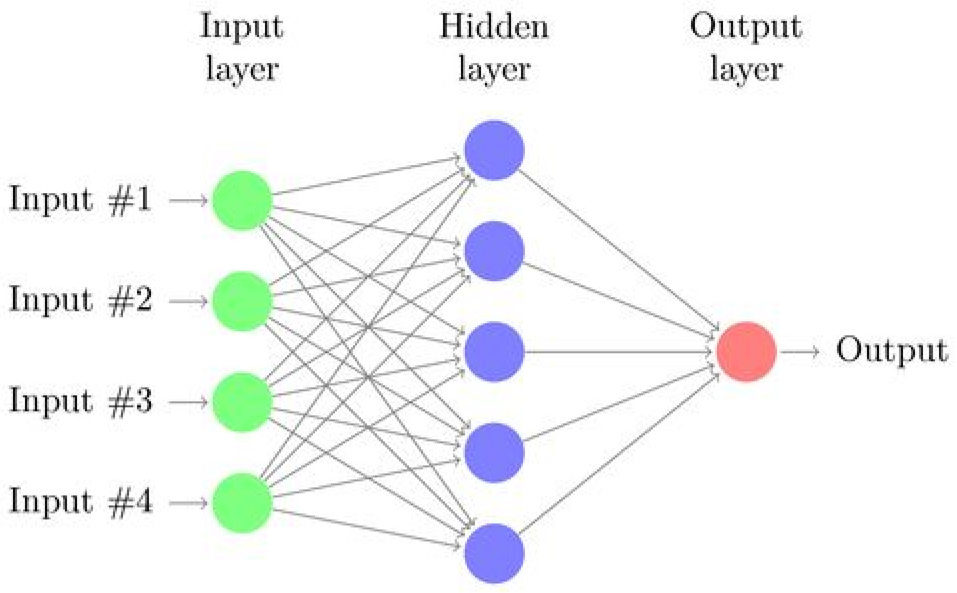
This project explores the effectiveness of different regression models in image classification. In the project, I train a multi-layered neural network, a logistic regression model, and a decision tree model on a dataset of handwritten digits.

Neural network was a technique invented as early as 1950s but it wasn't until as recent as early 2010s that neural network became widely known and used. One explanation for such phenomenon is that the world is digitized in 2010s due to the popularity of cell phones. Neural work is able to train on large amount data and performs significantly better than other learning techniques such as logistic regression and decision trees. In contrast, when there is not much data available, it is costly to train neural network and neural network usually doesn't perform better than other cheaper learning models. The idea can be captured by the following graph.



Understand Neural Network

Neural network is a series of layers of computation nodes, where each node performs a regression model. Usually, the regression model is logistic regression. Each layers of nodes obtain inputs from the previous layers and calculate the outputs using the regression models of each node of that layer. The first layer of the neural network would be the layer with raw input (X) and the final layer of the neural network would contain the prediction that this neural network makes ($\hat{Y}$). The structure of a neural network can be represented by the following computational graph.



Of course, plugging the inputs of each layer into its computation nodes is the easy part. We need to fit the data to the neural network in order to obtain the parameters such that the error of the model is lowest with respect to the data given. In order to estimate the error of the model, let's only focus on the output layer of the model since that's the output compute the predictions of the entire neural network $(\hat{Y})$. Here is a reasonable cost function that estimates the error that this model is making.

$cost = \sum_{i=1}^{m} -y^{(i)} log(\frac{1}{1+e^{-x^{(i)}\theta}}) - (1-y^{(i)}) log(1 - \frac{1}{1+e^{-x^{(i)}\theta}})$, where m is the number of examples.

As a classification problem, y of each example is either 1 or 0. Therefore,

$$If\ y = 0,\ then\ cost = \sum_{i=1}^{m} -log(1 - \frac{1}{1+e^{-x^{(i)}\theta}})$$

$$If\ y = 1,\ then\ cost = \sum_{i=1}^{m} -log(\frac{1}{1+e^{-x^{(i)}\theta}})$$

Further, $\frac{1}{1+e^{-x^{(i)}\theta}}$ is the output of a logistic regression model. If $y = 0,$ the larger $\frac{1}{1+e^{-x^{(i)}\theta}}$ is, the closer $1 - \frac{1}{1+e^{-x^{(i)}\theta}}$ is close to 0, the more negative $log(1 - \frac{1}{1+e^{-x^{(i)}\theta}})$ is, and therefore the larger $-log(1 - \frac{1}{1+e^{-x^{(i)}\theta}})$ is. Follow the same reasoning, if $y = 1,$ the smaller $\frac{1}{1+e^{-x^{(i)}\theta}}$ is, the closer $\frac{1}{1+e^{-x^{(i)}\theta}}$ is close to 0, the more negative $log(\frac{1}{1+e^{-x^{(i)}\theta}})$ is, and therefore the larger

$-\,log(\frac{1}{1+e^{-x^{(i)}\theta}})$ is. We can see that the cost increases as the deviation from the actual y increases. Now our task is to minimize the cost function with respect to the parameter θ, and hence, to find $\frac{\partial cost(\theta)}{\partial \theta_j}$ where θ_j is the every parameter in the output layer.

We can first calculate the cost with respect to one single example. Because derivative is a linear transformation, the derivative of the aggregate cost is simply the summation of the derivative of every example. Using chain rule, we get the following:

$$\frac{\partial(cost)}{\partial\theta_j} = \frac{\partial(cost)}{\partial(a)} \cdot \frac{\partial(a)}{\partial(z)} \cdot \frac{\partial(z)}{\partial\theta_j},\ where\ a = \frac{1}{1+e^{-z}},\ z = \theta \cdot x$$

$$\frac{\partial(cost)}{\partial a} =-(\frac{y}{a} - \frac{1-y}{1-a}),\ using\ derivative\ of\ natural\ log.$$

$$\frac{\partial a}{\partial z} = -\frac{-e^{-z}}{(1+e^{-z})^2},\ using\ quotient\ rule\ and\ chain\ rule,$$

$$\frac{\partial a}{\partial z} = \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}},$$

$$\frac{\partial a}{\partial z} = (1 - \frac{1}{1+e^{-z}}) \cdot \frac{1}{1+e^{-z}},$$

$$\frac{\partial a}{\partial z} = (1 - a) \cdot a,\ by\ substitution.$$

$$\frac{\partial(z)}{\partial(\theta_{j)}} = x_j.$$

$$\text{Hence, } \frac{\partial(cost)}{\partial\theta_j} =-(\frac{y}{a} - \frac{1-y}{1-a}) \cdot ((1 - a) \cdot a) \cdot x_j,$$

$$= \frac{(1-y)\,a - (1-a)\,y}{a(1-a)} \cdot (a(1-a)) \cdot x_{j,}$$

$$= (a - ay - y + ay) \cdot x_j\,,$$

$$= (a - y) \cdot x_j .$$

Aggregating the cost gradients across all examples, we get that $\frac{\partial cost}{\partial \theta_j} = \sum_{i=1}^{m} (\frac{1}{1+e^{-x^{(i)}\theta}} - y^{(i)}) x_j^{(i)}$ where $x_j^{(i)}$ is the j^{th} feature of i^{th} example. After obtaining the derivative of the cost function with respect to every single parameter in the output node, we can then either find θ at the minimum by setting the derivative to be 0 or we can use gradient descent to approach the minimum and update the parameters incrementally. In this project, I choose to use gradient descent because of the limited amount of computing power I have. Every iteration, I perform the follow computation and update the parameters:

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial cost}{\partial \theta_j}$$, for all θ_j in the final layer.

Using the math we have developed up till now, we can successfully update parameters for the output layer of a neural network. Next we need to figure out how to update parameters for the nodes in the hidden layers. To do this, we need to figure out the derivative of the cost function with respect to the output of each node in the hidden layer a, where $a = \frac{1}{1+e^{-x\theta}}$. Note that [i] indicates that that variable belongs to layer i. Input layer is layer 0, the hidden layer is layer 1, and the output layer is layer 2. Using chain rule, we can say that,

$$\frac{\partial(cost)}{\partial(a^{[1]})} = \frac{\partial(cost)}{\partial(a^{[2]})} \cdot \frac{\partial(a^{[2]})}{\partial(x^{[2]}\theta^{[2]})} \cdot \frac{\partial(x^{[2]}\theta^{[2]})}{\partial x^{[2]}}$$,(from the previous derivation we know

$$\frac{\partial(cost)}{\partial(a^{[2]})}, \frac{\partial(a^{[2]})}{\partial(x^{[2]}\theta^{[2]})})$$

$$\frac{\partial(cost)}{\partial(a^{[1]})} = \frac{\partial(cost)}{\partial(a^{[2]})} \cdot \frac{\partial(a^{[2]})}{\partial(x^{[2]}\theta^{[2]})} \cdot \theta^{[2]} .$$

Further,

$$\frac{\partial cost}{\partial \theta^{[1]}} = \frac{\partial(cost)}{\partial(a^{[1]})} \cdot \frac{\partial(a_{[1]})}{\partial(x^{[1]}\,\theta^{[1]})} \cdot \frac{\partial(x^{[1]}\,\theta^{[1]})}{\partial(\theta^{[1]})},$$

$$\frac{\partial cost}{\partial \theta^{[1]}} = \frac{\partial(cost)}{\partial(a^{[1]})} \cdot \frac{1}{1+e^{-x\theta^{[1]}}} \cdot (1 - \frac{1}{1+e^{-x\theta^{[1]}}}) \cdot x.$$

In essence, finding the gradients for parameters in the layer before the output layer is multiplying a series of derivatives. Using the same gradient descent technique, we can minimize cost with respect to the parameters in the hidden layer. Now, we can minimize the cost function with respect to every parameter in this neural network. If there are more layers than what the neural network has here, we can simply treat each layer before the layer for which we are computing the gradients of as the output layer and perform the same operation here from the layer of the network to the input layer of the network.

Developing neural network

Using the formula we developed above, we can develop a neural network of any size. First, we randomly initialize parameters θ. Second, we use the raw input to calculate output using the randomly initialized parameters. Third, once we obtained the output of the neural network, we can calculate the cost. Fourth, using the output of the neural network, we can calculate the gradients of the parameters θ and update the parameters θ. Then, we repeat the previous steps until we can reach a set of parameters with low cost.

For the purpose of training large amount of datasets, our algorithm needs to be very efficient in order save time and computing power. Instead of using for-loop to calculate the cost and the gradients, a vectorized implementation can save us a lot of time. Here is the vectorized implementation.

$$z^{[1]} = \theta^{[1]} \cdot x$$

$$a^{[1]} = \frac{1}{1 + e^{-z^{[1]}}}$$

$$z^{[2]} = \theta^{[2]} \cdot a^{[i]}$$

$$a^{[2]} = \frac{1}{1 + e^{-z^{[2]}}}$$

To calculate the gradients:

$$\frac{\partial(cost)}{\partial z^{[2]}} = a^{[2]} - y$$

$$\frac{\partial(cost)}{\partial \theta^{[2]}} = \frac{1}{m} \cdot \frac{\partial(cost)}{\partial z^{[2]}} \cdot (a^{[2]})^T$$

$$\frac{\partial(cost)}{\partial z^{[1]}} = \theta^{[2]} \cdot (\frac{\partial(cost)}{\partial z^{[2]}} * (1 - a^{[1]}) * a^{[1]}), \ * \ \textit{indicates element wise multiplication.}$$

$$\frac{\partial(cost)}{\partial \theta^{[1]}} = \frac{1}{m} \cdot \frac{\partial(cost)}{\partial z^{[1]}} \cdot (a^{[1]})^T$$

The output of a sigmoid function is the probability of the output. Because a neural network could have multiple output units when doing multi-class classification, we could potentially get a list of probabilities that corresponds to different outputs. To convert the list of probabilities to 0s and 1, we choose the highest probabilities and turn its value to 1 and set the rest of the values in the list to be 0s.

Train the MNIST dataset

About the dataset:

MNIST is a dataset of handwritten digits from 0 to 9. Each instance of dataset is a 28 × 28 picture of a handwritten digit. The following table is the training record for neural network of different sizes.

run id	run time	layer dimensions	number of training sets	each image size	number of iteration	learning rate	accuracy	accuracy
MNIST							mnist_png test set	origin train set
1		784,200,10	42000	28, 28, 1	3000	0.0075	9024 / 10000	37702 / 42000
2		784,2500,2000,1500,1000,500,10	42000	28, 28, 1	2316	0.0075	9415 / 10000	39549 / 42000
3		784, 5, 5, 7, 10	42000	28, 28, 1	3000	0.0075	2583 / 10000	10819 /42000
4		784, 1, 10	42000	28, 28, 1	3000	0.0075	2365 / 10000	9830/42000
5		784,	42000	28, 28, 1	3000	0.0075	1135 / 10000	4684/42000

		2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2, 10						
6		784,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5, 10	42000	28, 28, 1	3000	0.0075	1135 / 10000	4684/42000
7		784, 10	42000	28, 28, 1	3000	0.0075	8743 / 10000	36480/42000

Table 1

The table above reflect several patterns about the performance of neural network with respect to its layer dimension. The structure of the neural network matters a lot with respect to its performance. First, the performance of the neural network does not improve with respect to the depth of the neural work, or the number of layers of a neural network if the amount of neurons at each layer are small. We can see this effect when we compare run 6 and run 1. The neural network in run 6 has 50 hidden layers and each one has 5 neurons. In total, it has 250 neurons. The neural network in run 1 has 1 hidden layer with 200 neurons. Although the total amount of neurons are equivalent, the performance of two neural networks are different. On the test set, the neural network in run 1 correctly classifies 90% of the images whereas the neural network in run 6 only correctly classifies 10% of the images. The dense neural network clearly performs better than the thin and stretched network. More, we can say the neural network in run 6 has a disastrous performance. Consider that there are 10 classes in total, namely 0 through 9, the neural network in run 6 performs only slightly better than a random guesser who has no information other than the number of classes since a random guesser has a probability of 0.1 to guess the right class. The depth of the neural network is counterproductive when the number of neurons in each layer is small. This pattern is further consolidated by the performance of the neural network in run 5 which has 50 layers with 2 neurons at each layer. It performs as bad as the neural network in run 6.

The second pattern that is clear is that the bigger the neural network, the better the performance. The neural network in run 2 performs much better than any of the other neural networks. It has much more neurons than any other neural network. The neural network in run 1 performs much better than run 3, 4, and 7. It has also more neurons than all of them. The neural network in run 3 performs better than that in run 4. It as well has more neurons. Intuitively, the larger the neural network, the more complex it is, and hence, it can absorb more information from the inputs. However, this is not proven rigorously and it’s difficult to understand what each neurons are doing. Further, despite the overwhelming examples confirming this pattern, run 8 provides a counterexample. The neural network in run 8 has 0 hidden layer but performs much better than run 3 and 4. After examining the neural network closely, we conclude that the neural network in run 8 is essentially 10 logistic regressions running at the same. Since there is no hidden layers between the input layers and output layers, the input data is directly sent to the neuron in the output layers. Each neuron in the output layer performs a logistic regression calculation using the input data. For example, one neuron in the output layer has a logistic regression function that determines the input image represents the number 1 or not. Another neuron in the output layer has a logistic regression function that determines whether the input image represents the number 2 or not, so on and so forth. So the general picture is that the bigger the neural nets the better it forms but there are some neural networks that counter that intuition.

Another pattern arises when compare the run 4 and run 7. The neural network in run 4 has one more hidden neuron than the neural network in run 7 but it performs much worse. We can conjecture that perhaps because the information from the input are represented by one single neuron, much of the information are lost. The input of that hidden layer is 784 data points but the output of that hidden layer is one number between 0 and 1. The output layer then needs to predict what number the image represents from that one single number. The neural networks in run 3, 5, and 6 could potentially suffers from the same problem. Regardless the depth of the hidden layers, the number of neurons of the first layer is much less than the size of the input layers. The information of the input is compressed and some information is lost. Here is another run of neural network that more or less confirms this idea.

run id	run time	layer dimensions	number of training sets	each image size	number of iteration	learning rate	Accuracyon the test set	Accuracy on the training set
8		783, 1, 200, 10	42000	28, 28, 1	3000	0.0075	1135 / 10000	4684/42000

Table 2

We can directly compare the neural network in run 8 and run 1. Both have around the same amount of neurons but structures of the two neural network are only slightly different, analogous to the difference between the neural networks in run 7 and run 4. The neural network with one neuron connecting the input layer to the rest of the network performs significantly worse than the other. It does only slightly better than the a random guess. Again, some information could be lost because that one neuron connecting the input layer and the rest of the network.

The patterns demonstrated in table 1 needed to taken with a grain of salt. Notice that instead of finding the global minimum of the cost function with respect to the parameters by setting the its derivative to be 0, we find the minimum cost by applying gradient descent. The purpose of using gradient descent is to save computational power and debug. Finding the inverse of an invertible matrix runs in $O(n^3)$; therefore, solving a system of equations would be computationally heavy. Further, if anything goes wrong in the algorithms we set up, there is not much indication for bugs during the run. We simply have to wait until the calculations are finished to find out something is wrong. Gradient descent, however, allows us to display the cost at each iteration. If the cost of the neural network does not decrease, we would know something is wrong with our algorithms. The downside of gradient descent is that the neural network could stuck in local minimum. If the neural network reaches a local minimum, the derivative of the cost function with respect to the parameters will be very small. At each iteration, the parameters are barely changed, making it difficult for parameters to change so that the cost function can reach its global minimum. This problem could be addressed by randomly initialized the parameters and perform gradient descent multiple times. Each run of gradient descent would give us a different local minimum each time. However, due to the limit of computation power, we only perform each run once. Since it's probable that the neural network stucks at local minimum, it is possible that the performance at table one is not the best possible performance of its corresponding neural network.

Compare with Traditional Learning Algorithms

We learn two classification algorithms applicable to classification problems, which are logistic regression and decision tree. Since I don't know the specificity of constructing a decision tree, I have to rely on Scikit Learn, an existing numerical computation library in python, to build a decision tree model.

Here is a table containing the performance of logistic regression and decision tree.

Run id	Layer	Number of	Size of input	Number	Learning	Accuracy on	Accuracy on

	dimensions	training examples	image	of iterations	rate	test set	training set
7	784, 10	42000	28, 28, 1	3000	0.0075	8743 / 10000	36480 / 42000
9	decision_tree	42000	28, 28, 1			9061 / 10000	42000 / 42000

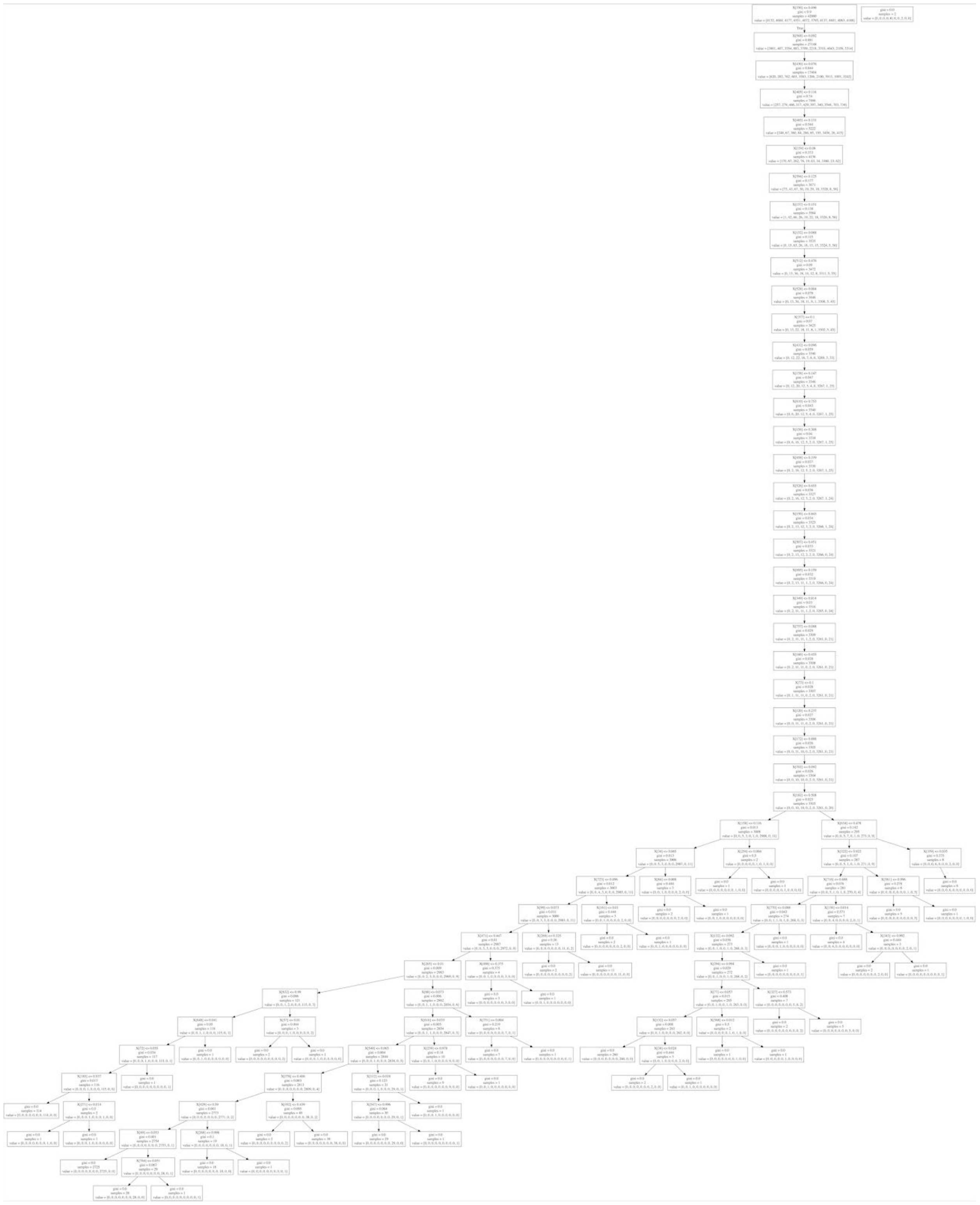
Table 3

The first thing that stands out in the table 2 is the incredible performance of decision tree on the training set. It correctly predicts every instance in the training set! This indicates that the decision tree model is complex enough to consider all the variation of data in the training set. However, this model does not generalize as well given a test set that it has never seen before. Though it correctly classifies 90% of the data in the test set, its performance is not as impressive as the big neural networks. This is an overfitting problem, where the cost of the model is extremely low on the training set but much higher on the testing set. The model is complex enough to fit the data in the training set perfectly but does not generalize well on other data. On the contrast, the large neural network in run 2 performs equally well on testing set and training set with higher performance. Nevertheless, decision tree model is the second best model that we have built. It performs better than all neural networks but the one in run 2.

The good performance of the trees come with its complexity. This decision tree has 6136 nodes and figure 1 is part of the decision tree.

Logistic regression has decent performance as well. Notice that the neural network in run 8 is essentially 10 logistic regressions running at the same. Since there is no hidden layers between the input layers and output layers, the input data is directly sent to the neuron in the output layers. Each neuron in the output layer performs a logistic regression calculation using the input data. For example, one neuron in the output layer has a logistic regression function that determines the input image represents the number 1 or not. Another neuron in the output layer has a logistic regression function that determines whether the input image represents the number 2 or not, so on and so forth. Logistic regression is the fourth best model and it classifies 87% of the instances correctly.

Figure 1



Conclusion

Does the graph shown in the beginning somewhat represent the performance of the algorithms? I would say this is the case. Among the models we produced, it's clear that the performance of a large deep neural networks performs the better than any other models by a descent margin(4%). When it comes to shallow network like the one in run 1, it performs more or less in the vicinity of the performance of traditional machine learning. Then again, it's plausible that neural networks stuck in local minimum and its performance is not the best it can do. Decision tree, on the other hand, reaches its utmost potential by correctly classifying every picture in the training set. Further, we find out that not only does the depth of the neural network matter, the width matters as well. Two neural networks with similar depth and around the same number of neurons can have drastically different performance. If the number of neurons in a layer is much smaller the number of input data points, neural network's performance would be horrible and we conjecture that small amount of neurons cause the neurons to lose some information about the input data.

Appendix

The repository of the codes:

https://github.com/PeterWilIiam/MNIST_classfier.git