Summer Report of Reading project on Machine Learning

By

Vishal Singh

16MS007

Under the supervision of

Dr. Siddhartha Lal

Indian Institute of Science Education and Research, Kolkata

Mohanpur, West Bengal 741246

Contents

1 What is Machine Learning

Machine Learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data, known as ”training data”, in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning algorithms are used in a wide variety of applications, such as email filtering and computer vision, where it is difficult or infeasible to develop a conventional algorithm for effectively performing the task. The types of machine learning depend on a variety of parameters but the most common types are:

- **Supervised learning:** In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

  Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

- **Unsupervised Learning:** Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results

2 Linear Regression with One variable

Linear regression with one variable, also known as univariate linear regression, is used when we want to predict a single output value y from a single input value x. We're doing supervised learning here, so that means we already have an idea about what the input/output cause and effect should be.
The Hypothesis Function Our hypothesis function has the general form

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \tag{1}$$

We give to $h_\theta(x)$ values for θ_0 and θ_1 to give our estimated output $\hat{y}$. We try to create a function $h_\theta(x)$ that is trying to map our input data to our output data.
Cost Function: We measure the accuracy of our hypothesis function by using a cost function. This takes an average of all the results of the hypothesis with inputs from x's compared to the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 \tag{2}$$

This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term

3 Gradient Descent

Almost every problem in Machine learning boils down to minimising the cost function described above for a variety of problems. One of the most powerful and widely used method for performing this minimization is Gradient Descent. The basic idea is to iteratively adjust the parameters in the direction where the gradient of the cost function is large and negative.
For gradient descent in linear regression, we have the algorithm of repeating the following equations till convergence.

$$\begin{cases} \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i) \\ \theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_\theta(x_i) - y_i)x_i) \end{cases} \tag{3}$$

Here m is the size of the training set,θ_0 is a constant that will be changing simultaneously with θ_i and x_i, y_i are values of the given training set. We start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

4 Multivariate Linear Regression

Multivariate linear regression allows us to add more features. This is similar to univariate linear regression in the sense that the hypothesis function is still a straight line but there are more than one variables. Let's introduce some notation before we dive deeper,
x_j^i is the value of feature j in the i^{th} training example
x^i is the column vector of all the feature inputs of the i^{th} training example
m is the number of training examples
$n = |x(i)|$ is the number of features.
The new hypothesis function now becomes
$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...\theta_n x_n$ The cost function is now,

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 \tag{4}$$

And gradient descent equation itself is of the same form; we just have to repeat it for our 'n' features. Repeat until convergence,

$$\begin{cases} \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ \theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\ \theta_2 := \theta_2 - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\ \quad ... \end{cases} \tag{5}$$

This can be written in a compact form as,

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad \text{for j=0...n} \tag{6}$$

Now the obvious next step is **Polynomial Regression**. Here our hypothesis function need not be linear (a straight line) if that does not fit the data well. We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function, or any other form. We can also combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$.

5 Logistic regression

Linear regression focuses on learning from datasets for which there is a "continuous" output. We try to learn the coefficients of a polynomial to predict the response of a continuous variable y_i on unseen data based on its independent variables x_i . However, a wide variety of problems, such as classification, are concerned with outcomes taking the form of discrete variables (i.e. categories). For example, we may want to detect if there is a cat or a dog in an image. Here, Logistic Regression comes to our rescue. The logistic model is used to model the probability of a certain class or event existing such as pass/fail, win/lose, alive/dead or healthy/sick. This can be extended to model several classes of events such as determining whether an image contains a cat, dog, lion, etc... Each object being detected in the image would be assigned a probability between 0 and 1 and the sum adding to one.

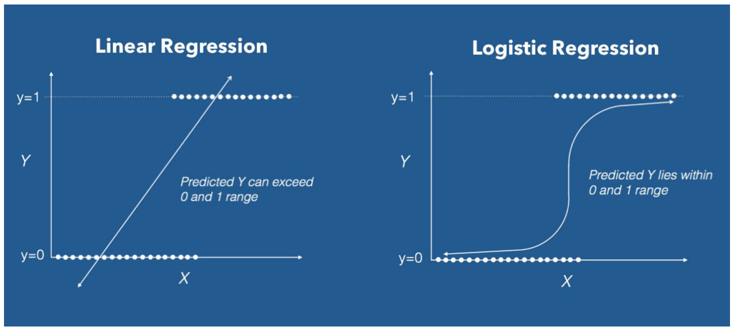


Figure 1: Linear Regression VS Logistic Regression Graph

Lets discuss the simpler case where we have binary classification . Instead of our output vector y being a continuous range of values, it will only be 0 or 1. $y \in \{0, 1\}$, where 0 is usually taken as the "negative class" and 1 as the "positive class", but you are free to assign any representation to it. Our hypothesis should satisfy : $0 \leq h_\theta(x) \leq 1$. Our new form uses the "Sigmoid Function," also called the "Logistic Function":$h_\theta(x) = g(\theta^T x)$, $z = (\theta^T x), g(z) = \frac{1}{1+e^{-z}}$.
h_θ gives us the probability that our output is 1. Now we need a **Decision Boundary**. In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1 \tag{7}$$
$$h_\theta(x) < 0.5 \rightarrow y = 0 \tag{8}$$

The decision boundary is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function. Now we can have a simplified **cost function** for logistic regression as, $\text{Cost}(h_\theta(x), y) = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$. We can see that when when y is equal to 1, then the second term$(1-y)log(1-h_\theta(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-ylog(h_\theta(x))$ will be zero and will not affect the result. The entire cost function can be written as,

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))] \tag{9}$$

For **Gradient Descent**, we have to evaluate,

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{10}$$

Upon working out the derivative part, we get

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{11}$$

which we have to repeat. One thing of note here is that this algorithm is identical to the one we used in linear regression.

6 Bias vs Variance

The bias-variance tradeoff summarizes the fundamental tension in machine learning between the complexity of a model and the amount of training data needed to fit it. Since data is often limited, in practice it is frequently useful to use a less complex model with higher bias, a model whose asymptotic performance is worse than another model – because it is easier to train and less sensitive to sampling noise arising from having a finite-sized training dataset (i.e. smaller variance).

Bias are the simplifying assumptions made by a model to make the target function easier to learn. Generally, linear algorithms have a high bias making them fast to learn and easier to understand but generally less flexible. In turn, they have lower predictive performance on complex problems that fail to meet the simplifying assumptions of the algorithms bias.

Low Bias: Suggests less assumptions about the form of the target function.

High-Bias: Suggests more assumptions about the form of the target function.

High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).

Variance is the amount that the estimate of the target function will change if different training data was used.

The target function is estimated from the training data by a machine learning algorithm, so we should expect the algorithm to have some variance. Ideally, it should not change too much from one training dataset to the next, meaning that the algorithm is good at picking out the hidden underlying mapping between the inputs and the output variables.

Machine learning algorithms that have a high variance are strongly influenced by the specifics of the training data. This means that the specifics of the training have influences the number and types of parameters used to

characterize the mapping function.
Low Variance: Suggests small changes to the estimate of the target function with changes to the training dataset.
High Variance: Suggests large changes to the estimate of the target function with changes to the training dataset.
High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

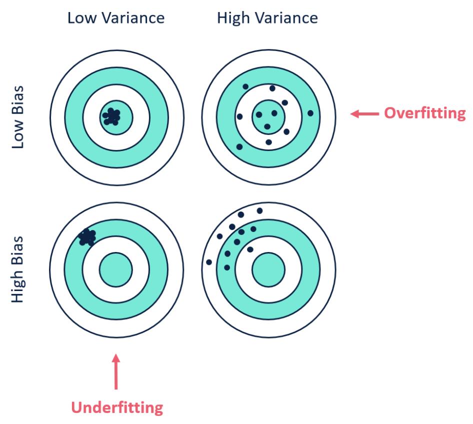


Figure 2: Illustrating Bias and Variance through hitting points on a target. Each point on the target represents a different iteration of a model, fit for the same problem with different training data sets.

Models with low variance tend to be less complex with a simple underlying structure. They also tend to be more robust (stable) to different training data (i.e., consistent, but inaccurate). Models that fall in this category generally include parametric algorithms, such as regression models. Depending on the data, algorithms with low variance may not be complex or flexible enough to learn the true pattern of a data set, resulting in underfitting.

Models with low bias algorithms tend to be more complex, with a more flexible underlying structure. The higher level of flexibility in the models can allow for more complex relationships between data but can also cause overfitting because the model is free to memorize the training data, instead of generalizing a pattern found in the data. Models with low bias also tend to be less stable between training data sets. Non-parametric models typically have low bias and high variability.

There is no escaping the relationship between bias and variance in machine learning.

- Increasing the bias will decrease the variance.
- Increasing the variance will decrease the bias.

The goal in relation to bias and variance is to find the balance between the two that minimizes overall (total) error.

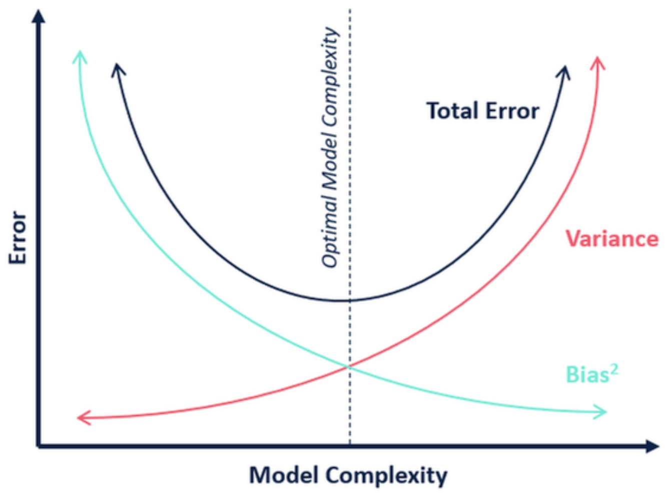


Figure 3: Bias vs Variance Tradeoff.

7 Artificial Neural Networks

Artificial neural networks (ANN) are computing systems that are inspired by, but not identical to, biological neural networks that constitute animal brains. Such systems ”learn” to perform tasks by considering examples, generally without being programmed with task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as ”cat” or ”no cat” and using the results to identify cats in other images. They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it.

In ANN implementations, the ”signal” at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. Neurons have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), after traversing the layers multiple times. An ANN has the following components:

- **Neurons :** ANNs have artificial neurons, which receive input, combine the input with their internal state (activation) and an optional threshold using an activation function, and produce output using an output function. The initial inputs are external data, such as images and documents. The ultimate outputs accomplish the task, such as recognizing an object in an image. The important characteristic of the activation function is that it provides a smooth transition as input values change, i.e. a small change in input produces a small change in output

- **Connections and weights :** The network consists of connections, each connection providing the output of one neuron as an input to another neuron. Each connection is assigned a weight that represents its relative importance. A given neuron can have multiple input and output connections.

- **Propagation function :** The propagation function computes the input to a neuron from the outputs of its predecessor neurons and their connections as a weighted sum.

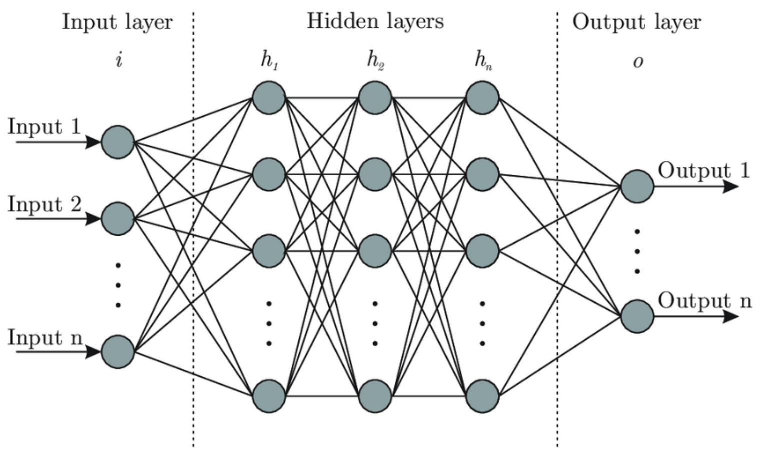


Figure 4: ANN Schematic.

An ANN over time learns and adapts to get better at the task it is supposed to do. Learning involves adjusting the weights (and optional thresholds) of the network to improve the accuracy of the result. This is done by minimizing the observed errors. Learning is complete when examining additional observations does not usefully reduce the error rate. For accomplishing this task we have,

- **Learning Rate:** The learning rate defines the size of the corrective steps that the model takes to adjust for errors in each observation. A high learning rate shortens the training time, but with lower ultimate accuracy, while a lower learning rate takes longer, but with the potential for greater accuracy

- **Cost Function :** A cost function is what we try to minimize to make the network better at the relevant task. This depends on what specifically the neural ntwork is designed to do.

- **Backpropagation :** Backpropagation is a method to adjust the connection weights to compensate for each error found during learning. The error amount is effectively divided among the connections. Technically, backprop calculates the gradient of the cost function associated with a given state with respect to the weights. The weight updates can be done via many methods, for example, stochastic gradient descent. The Backpropagation Algorithm is the backbone of how an artificial neural network works and warrants a deeper discussion.

8 The Backpropagation Algorithm

At its core, backpropagation is simply the ordinary chain rule for partial differentiation, and can be summarized using four equations. In order to see this, we must first establish some useful notation. We will assume that there are L layers in our network with l = 1, . . . , L indexing the layer. Denote by ω_{jk}^l the weight for the connection from the k-th neuron in layer l-1 to the j-th neuron in layer l. We denote the bias of this neuron by b_j^l. By construction, in a feed-forward neural network the activation $a_j^l$of the j-th neuron in the l-th layer can be related to the activities of the neurons in the layer l-1 by the equation,

$$a_j^l = \sigma\left(\sum_k \omega_{jk}^l a_k^{l-1} + b_j^l\right) = \sigma(z_j^l) \tag{12}$$

where we have defined the linear weighted sum $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$

By definition, the cost function E depends directly on the activities of the output layer a_j^L. It of course also indirectly depends on all the activities of neurons in lower layers in the neural network through iteration of eq (12). We define the error Δ_j^L of the j-th neuron in the L-th layer as the change in cost function with respect to the weighted input z_j^l

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} \tag{13}$$

We can analogously define the error of neuron j in layer l,Δ_j^l, as the change in the cost function w.r.t. the weighted input z_j^l:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma^{'}(z_j^L) \tag{14}$$

where in the last line we have used the fact that $\partial b_j^l / \partial z_j^l = 1$ This is the second of the four backpropagation equations.

Since the error depends on neurons in layer l only through the activation of neurons in the subsequent layer l + 1, we can use the chain rule to write,

$$\Delta_j^l = \frac{\partial E}{\partial z_j^L} = \sum_k \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{15}$$

$$= \sum_k \Delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{16}$$

$$= \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^L) \tag{17}$$

This is the third backpropagation equation. The final equation can be derived by differentiating of the cost function with respect to the weight w_{jk}^l as,

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \tag{18}$$

Together, Eqs. (13), (14), (17) and (18) define the four backpropagation equations relating the gradients of the activations of various neurons a_j^l the weighted inputs,$z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$ and the errors, Δ_j^l. These equations can be combined into a simple, computationally efficient algorithm to calculate the gradient with respect to all parameters.

The Backpropagation Algorithm

1. **Activation at input layer:** calculate the activations a_j^l of all the neurons in the input layer.

2. **Feedforward:** starting with the first layer, exploit the feed-forward architecture through Eq.(12) to compute z^l and $a^l$for each subsequent layer.

3. **Error at top layer:** calculate the error of the top layer using Eq.(13). This requires to know the expression for the derivative of both the cost function $E(w) = E(a^L)$ and the activation function $\sigma(z)$

4. **"Backpropagate" the error:** use Eq.(17) to propagate the error backwards and calculate Δ_j^l for all layers.

5. **Calculate gradient:** Finally use Eqs. (14) and (18) to calculate $\frac{\partial E}{\partial b_j^l}$ and $\frac{\partial E}{\partial w_{jk}^l}$

Its evident now where the name backpropagation comes from. The algorithm consists of a forward pass from the bottom layer to the top layer where one calculates the weighted inputs and activations of all the neurons. One then backpropagates the error starting with the top layer down to the input layer and uses these errors to calculate the desired gradients. This description makes clear the incredible utility and computational efficiency of the backpropagation algorithm. We can calculate all the derivatives using a single "forward" and "backward" pass of the neural network. This computational efficiency is crucial since we must calculate the gradient with respect to all parameters of the neural net at each step of gradient descent.

To summarize, in Backpropagation, we calculate the error , check if the error is minimized or not and update the parameters. Repeat these three steps until the error becomes minimum.

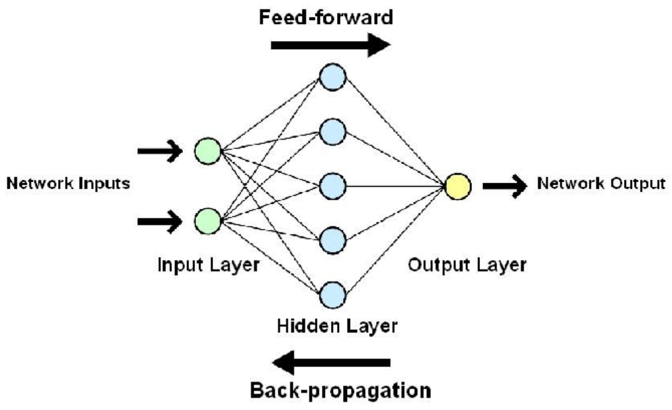


Figure 5: Schematic explaining Backpropagation of errors working in tandem with feed forward to reduce cost function of neural network.

9 Coding a neural network

The code will have three parts

- initialisation - to set the number of input, hidden and output nodes
- train - refine the weights after being given a training set example to learn from

- query - give an answer from the output nodes after being given an input

The skeleton code will look something like this.

```
# neural network class definition
class neuralNetwork:
 # initialise the neural network
 def __init__():
 pass
 # train the neural network
 def train():
 pass
 # query the neural network
 def query():
 pass
```

We will make a simple 3 layer neural network here, input layer, 1 hidden layer, and the output layer.

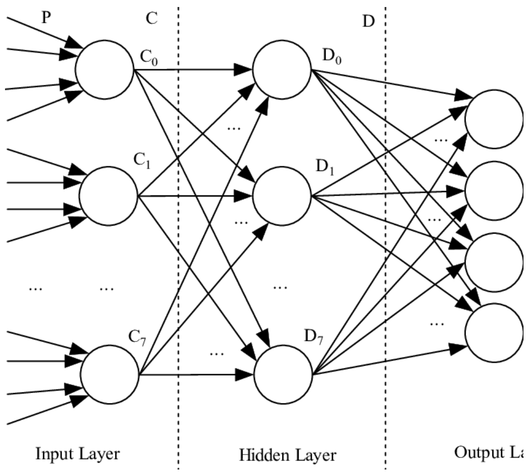


Figure 6: The neural network we are trying to make.

9.1 Initialisation

We need to set the number of input, hidden and output layer nodes and the learning rate. That defines the shape and size of the neural network. We set them using parameters when a neural network object is created to allow us to make neural networks of varying specifications according to our requirements.

```
 # initialise the neural network
def __init__(self, inputnodes, hiddennodes, outputnodes,
   learningrate ):
 # set number of nodes in each input, hidden, output layer
 self.inodes = inputnodes
 self.hnodes = hiddennodes
 self.onodes = outputnodes
 # learning rate
 self.lr = learningrate
 pass
```

9.1.1 Weights

The most efficient way of expressing weights is using matrix. So we create two matrices.

- W_{input_hidden}- matrix for the weights for links between the input and hidden layers, of size hidden_nodes by input_nodes
- W_{hidden_output}- matrix for the links between the hidden and output layers, of size output_nodes by hidden_nodes.

```
# link weight matrices, wih and who
# weights inside the arrays are w_i_j, where link is from
   node
i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = (numpy.random.rand(self.hnodes, self.inodes) -
   0.5)
self.who = (numpy.random.rand(self.onodes, self.hnodes) -
   0.5)
```

9.2 Query

The query() function takes the input to a neural network and returns the network's output. We to pass the input signals from the input layer of nodes, through the hidden layer and out of the final output layer.Now the matrix of weights for the link between the input and hidden layers can be combined with the matrix of inputs to give the signals into the hidden layer nodes.

$$X_{hidden} = W_{input\_hidden} \cdot I \tag{19}$$

To get the signals emerging from the hidden node, we simply apply the sigmoid squashing function to each of these emerging signals.

$$O_{hidden} = sigmoid(X_{hidden}) \tag{20}$$

So to summarise, we have.

```python
# calculate signals into hidden layer
hidden_inputs = numpy.dot(self.wih, inputs)
# calculate the signals emerging from hidden layer
hidden_outputs = self.activation_function(hidden_inputs)
# calculate signals into final output layer
final_inputs = numpy.dot(self.who, hidden_outputs)
# calculate the signals emerging from final output layer
final_outputs = self.activation_function(final_inputs)
```

9.3 Training

In the train() function we update the weights of the neural network in an attempt to get a set of weights that gives the correct output from a given input.

```python
# update the weights for the links between the hidden and
    output
layers
self.who += self.lr * numpy.dot((output_errors *
    final_outputs *
(1.0 - final_outputs)), numpy.transpose(hidden_outputs))
# update the weights for the links between the input and
    hidden
layers
self.wih += self.lr * numpy.dot((hidden_errors *
    hidden_outputs *
(1.0 - hidden_outputs)), numpy.transpose(inputs))
```

9.4 Final Code

After putting all the pieces together, we end up with ,

```python
import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
# neural network class definition
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.inodes, -0.5), (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.hnodes, -0.5), (self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass


    # train the neural network
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and output layers
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass


    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs
```

A test run of the code could be done for example using the following parameters

```
# number of input, hidden and output nodes
input_nodes = 3
hidden_nodes = 3
output_nodes = 3

# learning rate is 0.3
learning_rate = 0.3

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
    learning_rate)
```

9.5 Testing the Neural Network on MNIST Data Set

The previous code shows us how to make a neural network. Now we can test it and see how it works on the MNIST Data Set. The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The previous subsection was titled ”Final Code”. That was a lie. Kind of. If we are going to test the code on an actual dataset, we need to add some more things.
First we need to add some lines for reading data from the dataset,

```
data_file = open("mnist_train_100.csv", 'r') #enter the name
    of file to be used
data_list = data_file.readlines()
data_file.close()
```

Then we need to convert the dataset into a suitable format which can then be used by our neural network for processing.

```
all_values = data_list[0].split(',')
image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys',
interpolation='None')
scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 *
    0.99) +0.01
```

Now, if we are going to test our neural network, we need to have a way of providing it with a score at the end. What use is a test without a result? So for that, we add the following code snippet,

```
# test the neural network
# scorecard for how well the network performs, initially empty
scorecard = []
# go through all the records in the test data set
for record in test_data_list:
 # split the record by the ',' commas
 all_values = record.split(',')
 # correct answer is first value
 correct_label = int(all_values[0])
 print(correct_label, "correct label")
 # scale and shift the inputs
 inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
 # query the network
 outputs = n.query(inputs)
 # the index of the highest value corresponds to the label
 label = numpy.argmax(outputs)
 print(label, "network's answer")
 # append correct or incorrect to list
 if (label == correct_label):
 # network's answer matches correct answer, add 1 to
scorecard
 scorecard.append(1)
 else:
 # network's answer doesn't match correct answer, add 0 to
scorecard
 scorecard.append(0)
 pass
 pass
# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() /
scorecard_array.size)
```

Another useful addition to the code would be the option of doing multiple runs. That can be achieved using the following lines of code.

```
# train the neural network
# epochs is the number of times the training data set is used for
training
epochs = 2
for e in range(epochs):
 # go through all records in the training data set
 for record in training_data_list:
 # split the record by the ',' commas
 all_values = record.split(',')
 # scale and shift the inputs
 inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
0.01
 # create the target output values (all 0.01, except the
desired label which is 0.99)
 targets = numpy.zeros(output_nodes) + 0.01
 # all_values[0] is the target label for this record
 targets[int(all_values[0])] = 0.99
 n.train(inputs, targets)
 pass
 pass
```

Now with all these snippets added, we finally get the code which we can use to test the neural network. The Real Final Code,

```
# python notebook for Make Your Own Neural Network
# code for a 3-layer neural network, and code for learning
   the MNIST dataset
# (c) Tariq Rashid, 2016
# license is GPLv2
import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
- 169 -
# library for plotting arrays
import matplotlib.pyplot
# ensure the plots are inside this notebook, not an external
   window
%matplotlib inline
# neural network class definition
class neuralNetwork:
 # initialise the neural network
 def __init__(self, inputnodes, hiddennodes, outputnodes,
learningrate):
 # set number of nodes in each input, hidden, output layer
 self.inodes = inputnodes
 self.hnodes = hiddennodes
 self.onodes = outputnodes
 # link weight matrices, wih and who
 # weights inside the arrays are w_i_j, where link is from
node i to node j in the next layer
 # w11 w21
 # w12 w22 etc
 self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
 self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))
 # learning rate
 self.lr = learningrate
 # activation function is the sigmoid function
 self.activation_function = lambda x: scipy.special.expit(x)
 pass
 # train the neural network
 def train(self, inputs_list, targets_list):
 # convert inputs list to 2d array
 inputs = numpy.array(inputs_list, ndmin=2).T
 targets = numpy.array(targets_list, ndmin=2).T
 # calculate signals into hidden layer
 hidden_inputs = numpy.dot(self.wih, inputs)
 # calculate the signals emerging from hidden layer
- 170 -
 hidden_outputs = self.activation_function(hidden_inputs)
 # calculate signals into final output layer
 final_inputs = numpy.dot(self.who, hidden_outputs)
 # calculate the signals emerging from final output layer
 final_outputs = self.activation_function(final_inputs)
 # output layer error is the (target - actual)
 output_errors = targets - final_outputs
 # hidden layer error is the output_errors, split by weights,
recombined at hidden nodes
 hidden_errors = numpy.dot(self.who.T, output_errors)
 # update the weights for the links between the hidden and
output layers
 self.who += self.lr * numpy.dot((output_errors *
final_outputs * (1.0 - final_outputs)),
numpy.transpose(hidden_outputs))
 # update the weights for the links between the input and
hidden layers
 self.wih += self.lr * numpy.dot((hidden_errors *
hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(
    inputs))
 pass
 # query the neural network
 def query(self, inputs_list):
 # convert inputs list to 2d array
 inputs = numpy.array(inputs_list, ndmin=2).T
 # calculate signals into hidden layer
 hidden_inputs = numpy.dot(self.wih, inputs)
 # calculate the signals emerging from hidden layer
 hidden_outputs = self.activation_function(hidden_inputs)
 # calculate signals into final output layer
 final_inputs = numpy.dot(self.who, hidden_outputs)
 # calculate the signals emerging from final output layer
 final_outputs = self.activation_function(final_inputs)
 return final_outputs
- 171 -
# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 200
output_nodes = 10
# learning rate
learning_rate = 0.1
# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)
# load the mnist training data CSV file into a list
training_data_file = open("mnist_dataset/mnist_train.csv", 'r
    ')
training_data_list = training_data_file.readlines()
training_data_file.close()
# train the neural network
# epochs is the number of times the training data set is used
     for
training
epochs = 5
for e in range(epochs):
 # go through all records in the training data set
 for record in training_data_list:
 # split the record by the ',' commas
 all_values = record.split(',')
 # scale and shift the inputs
 inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
0.01
 # create the target output values (all 0.01, except the
desired label which is 0.99)
 targets = numpy.zeros(output_nodes) + 0.01
 # all_values[0] is the target label for this record
 targets[int(all_values[0])] = 0.99
 n.train(inputs, targets)
 pass
 pass
# load the mnist test data CSV file into a list
test_data_file = open("mnist_dataset/mnist_test.csv", 'r')
```

```
- 172 -
test_data_list = test_data_file.readlines()
test_data_file.close()
# test the neural network
# scorecard for how well the network performs, initially
   empty
scorecard = []
# go through all the records in the test data set
for record in test_data_list:
 # split the record by the ',' commas
 all_values = record.split(',')
 # correct answer is first value
 correct_label = int(all_values[0])
 # scale and shift the inputs
 inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
   0.01
 # query the network
 outputs = n.query(inputs)
 # the index of the highest value corresponds to the label
 label = numpy.argmax(outputs)
 # append correct or incorrect to list
 if (label == correct_label):
 # network's answer matches correct answer, add 1 to
scorecard
 scorecard.append(1)
 else:
 # network's answer doesn't match correct answer, add 0 to
scorecard
 scorecard.append(0)
 pass
 pass
# calculate the performance score, the fraction of correct
   answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() /
scorecard_array.size)
```

Now we test the neural network and rate its performance while varying three relevant parameters, learning rate, number of epochs (runs), and number of hidden nodes. The results are as follows.

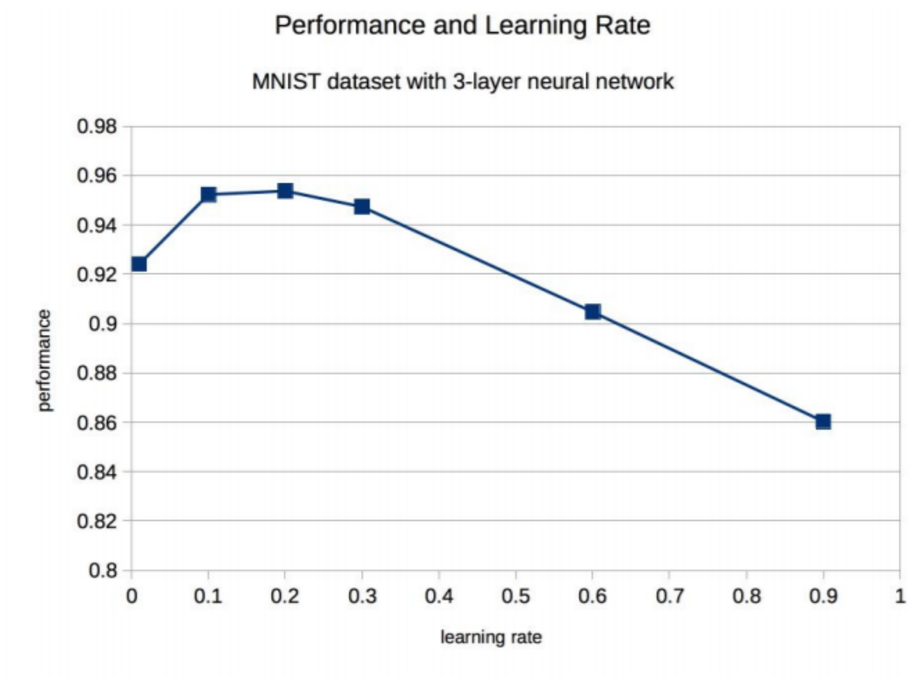
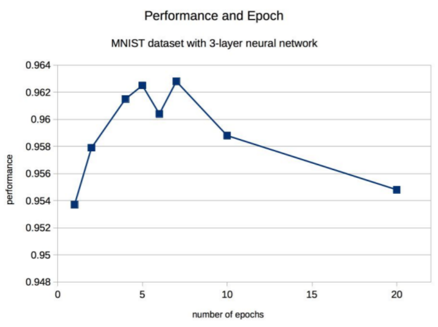
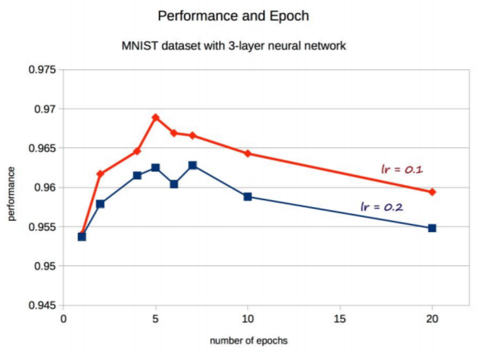


Figure 7: Varying performance with Learning rate



(a) Singular Learning Rate.



(b) Different Learning Rates.

Figure 8: Varying performance with number of epochs.

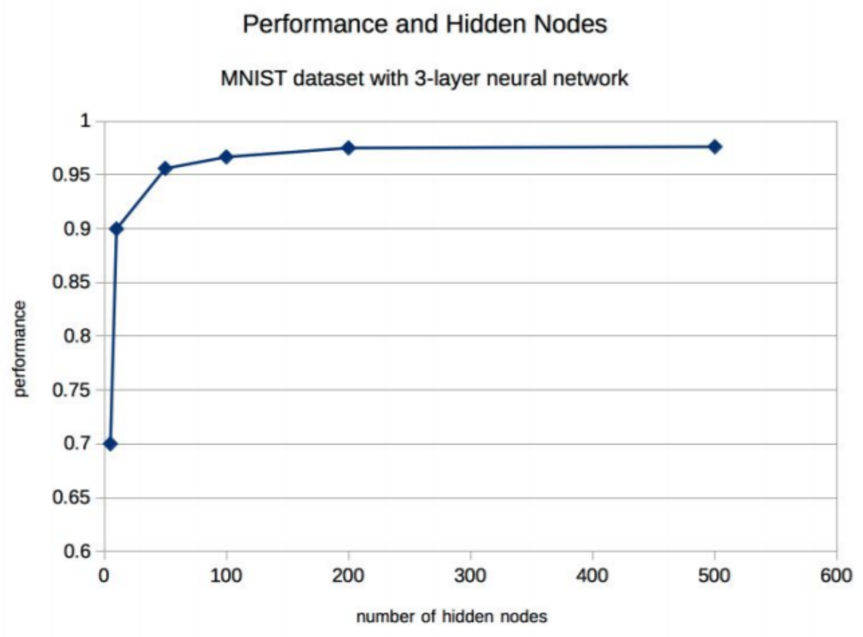


Figure 9: Varying performance with number of Hidden Nodes

Acknowledgement

I would like to thank my professor, Dr. Siddhartha Lal, who was extremely friendly and encouraging. This project was a great introduction to machine learning and neural networks.

Thanks to the IISER Kolkata administration for letting me stay in the hostel despite the severe lack of rooms. And the SMC for ensuring the availability of delicious food throughout the summer.

A special thanks is also due for my friends Aditi and Abhinav, for providing me with the much needed encouragement and motivation to keep reading during this reading project, and more importantly making me panic an appropriate amount for completing this report.

References

[1] Mehta, Pankaj et al. "A High-Bias, Low-Variance Introduction to Machine Learning for Physicists." Physics Reports 810 (2019): 1–124. Crossref. Web.

[2] Rashid, Tariq. *Make Your Own Neural Network.*

[3] Coursera Course
`https://www.coursera.org/learn/machine-learning/`

[4] `https://towardsdatascience.com/`

[5] `https://www.edureka.co/blog/backpropagation/`

[6] `https://en.wikipedia.org/`