● Part I

About the Technology Used

Supervised Learning

For the purpose of our project we used supervised learning. It is a learning method that can infer a function from labeled training data. The training data consist of a set of training examples. Each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way.

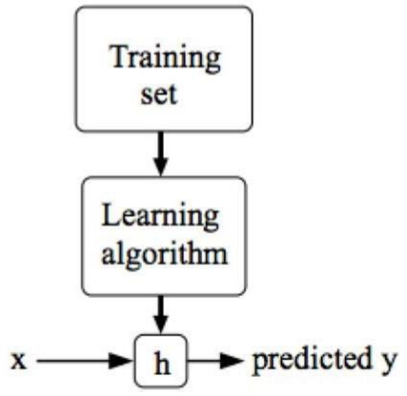


Figure 1: To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function h : X → Y so that h(x) is a "good" predictor for the corresponding value of y. Here, x represents input and y represents output.

Supervised Learning can be broadly classified into regression and classification problems:

In a regression problem, we try to predict the value of a continuous-valued function, given a number of input data.

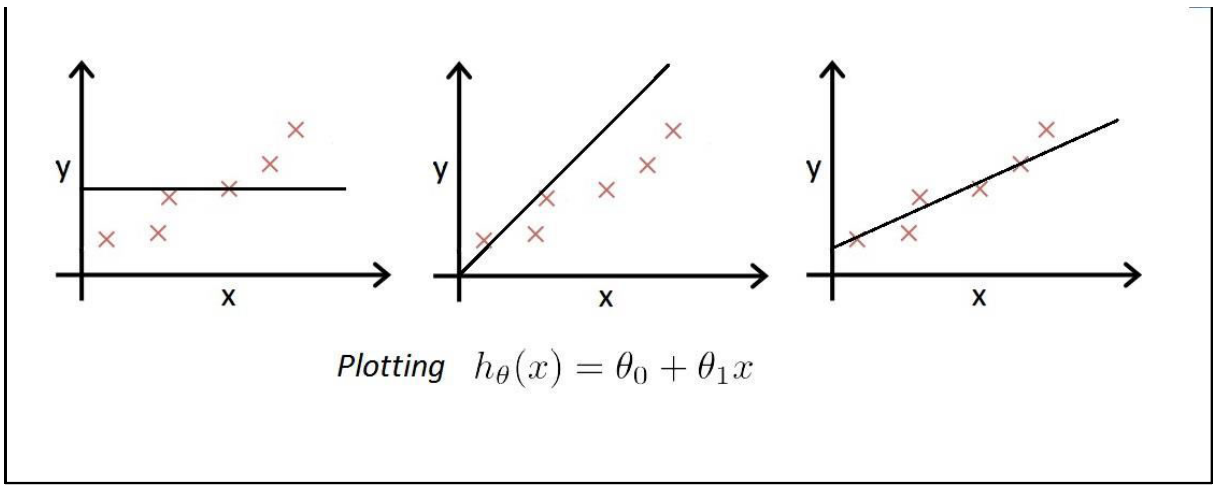


Figure 2: In a regression problem, we try to find the best value of $\theta_0$and $\theta_1$where $\theta_i$is the parameter vector. Choose θ_0,θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y) .

Whereas in classification, we try to find the correct class label for the given input. So, in Figure 1, if the 'predicted y' is continuous, we call it a regression problem. And if the 'predicted y' takes discrete values only, we call it a classification problem.

Classification Problem

In our project, we are only concerned with the classification problem. Here, y can take on only a small number of discrete values.

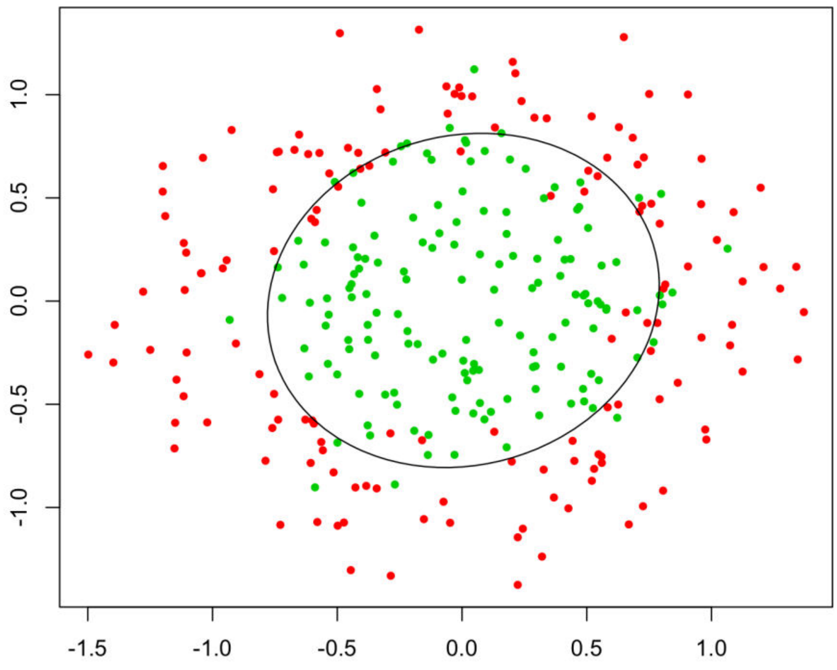


Figure 3: An example of a classification problem. Here, we are trying to separate the green points from the red points.

Our hypotheses$h_\theta(x)$need to satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the Logistic Function, also known as the Sigmoid Function.

Here,

$h_\theta(x) = g(z)$ $\quad$ $z = \theta^T x$ $\quad$ $g(z) = \frac{1}{1+e^{-Z}}$

i.e, $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

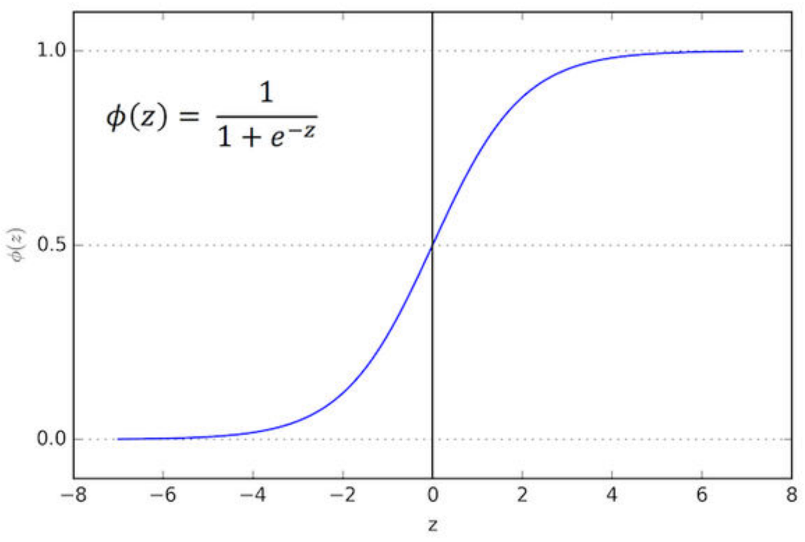


Figure 4: The sigmoid function Φ(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

$h_\theta(x)$ will give us the probability that our output is 1. For example, $h_\theta(x)$=0.7 gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

$$h_\theta(x) = P(y=1|x;\theta) = 1 - P(y=0|x;\theta)$$
$$P(y=0|x;\theta) + P(y=1|x;\theta) = 1$$

Cost Function

We can measure the accuracy of our hypothesis function by using a cost function. The cost function J(θ) for logistic regression is:

$$\mathrm{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

$$\begin{aligned} J(\theta) &= \tfrac{1}{m} \sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)}) \\ &= -\tfrac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))] \end{aligned}$$

To break it apart, we are calculating the cost for parameter vector θ , where, $h_\theta\left(x^{(i)}\right)$ is the predicted value for i^{th} training example and $y^{(i)}$is the corresponding actual value.

Using these equations:

- If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.
- If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

Our main goal here is to minimise the cost function. To achieve that goal, we have to find $min_\theta J(\theta)$.

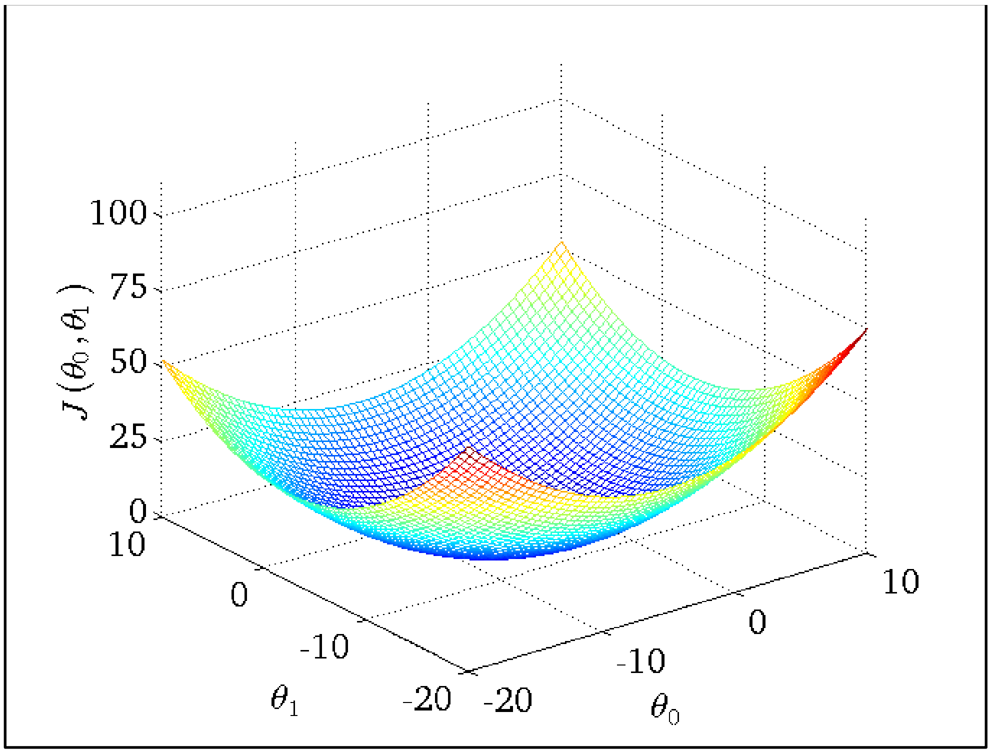


Figure 5: In this above graph we plot $\theta_0$vs $\theta_1$vs J(θ_0,θ_1) to find out global minimum of J(θ).

Learning Algorithm

For the purpose of this project, we choose gradient descent to find the minimum value of cost function J(θ).

Gradient Descent

______So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in. In gradient descent algorithm we initialise parameter vector $\theta$with some random values and run the algorithm for few iterations, until the value of θconverges to the global minimum. Also we have to check that the value of θdecreases in every iteration.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the learning rate.

The gradient descent algorithm is:

Repeat until convergence:

$\theta_j := \theta_j - \alpha \frac{\delta}{\delta\theta_j} J(\theta)$

For each input feature j, one should simultaneously update the parameter vector θ.

On a side note, we should adjust our parameter 'α' to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong. While small value of α takes long time to converge, a large value may never converge at all, oscillating on both sides of the minima.

Solving the derivative $\frac{\delta}{\delta\theta_j} J(\theta)$,

$$\begin{aligned}&Repeat\ \{\\ &\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\\ &\}\end{aligned}$$

where, m is the total number of training examples.

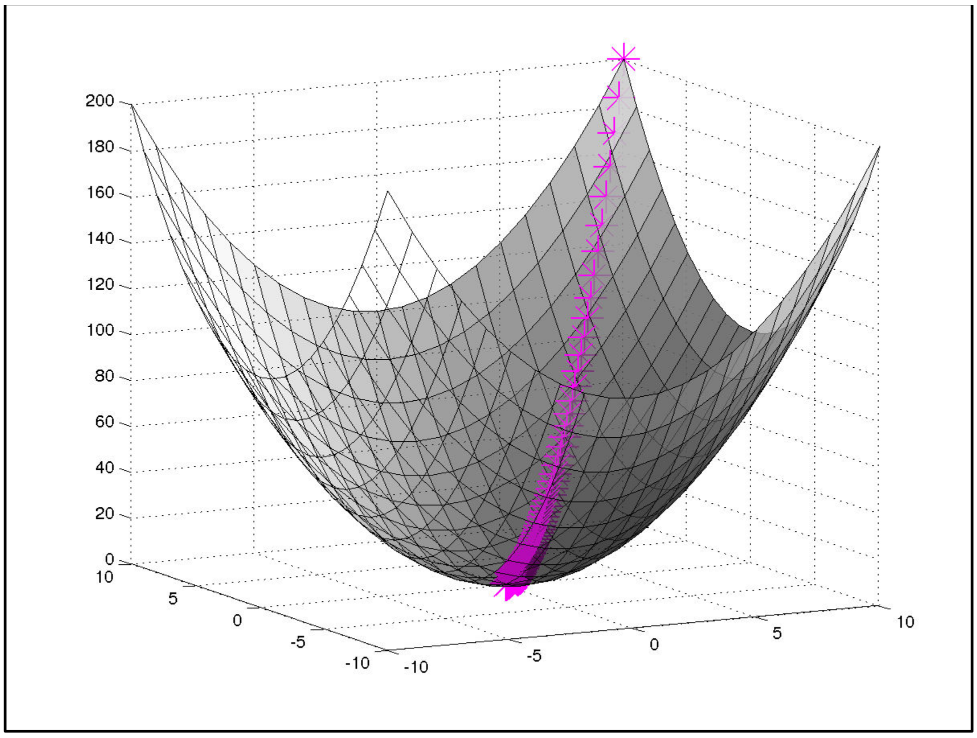


Figure 6: In GD, we initialise parameter vector with some random values and run the algorithm for few iterations, until the value of converges to the global minimum.

Multiclass Classification: One-vs-all

As our problem is to classify a single digit (0-9) for a given input, we approach with generalized logistic regression to multiclass problems.

Since y = {0,1...n}, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

$$
\begin{aligned}
&y \in \{0, 1 \dots n\} \\
&h_\theta^{(0)}(x) = P(y = 0 | x; \theta) \\
&h_\theta^{(1)}(x) = P(y = 1 | x; \theta) \\
&\cdots \\
&h_\theta^{(n)}(x) = P(y = n | x; \theta) \\
&\mathrm{prediction} = \max_i (h_\theta^{(i)}(x))
\end{aligned}
$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

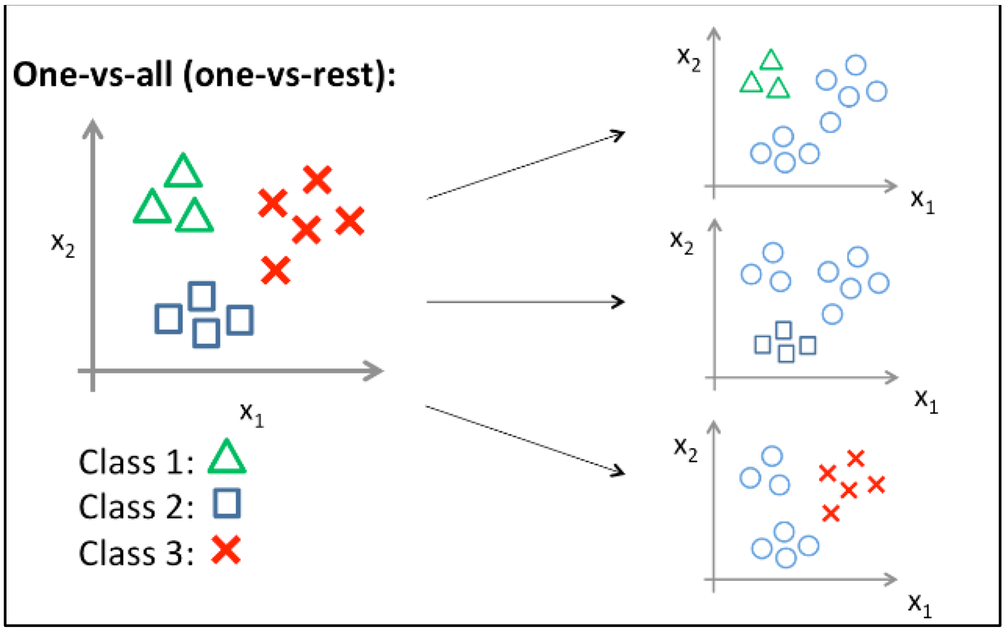


Figure 7: This figure shows how one could classify 3 classes: $h_\theta^{(i)}$ = P(y = i|x; θ) (x = 1,2,3)

To summarize:

- Train a logistic regression classifier h_θ(x) for each class￼to predict the probability that ￼ ￼y = i￼ ￼
- To make a prediction on a new x, pick the class ￼that maximizes h_θ(x).

For the purpose of our project instead of using normal gradient descent algorithm, we used an optimised version of the algorithm - fmincg(). This function is more sophisticated, and provides us with faster ways to optimize θ that can be used instead of gradient descent. It works on a continuous differentiable multivariate function.We just need to provide it the following two functions for a given input value θ:

1. Cost Function J(θ).
2. Partial derivative of the cost function $\frac{\delta}{\delta \ \ \theta_j}J(\theta)$.

Then, with multiple iterations, "fmincg()" can quickly reach the global minimum of the cost function.

This function works faster and more efficiently with larger dataset.

The Problem of Overfitting

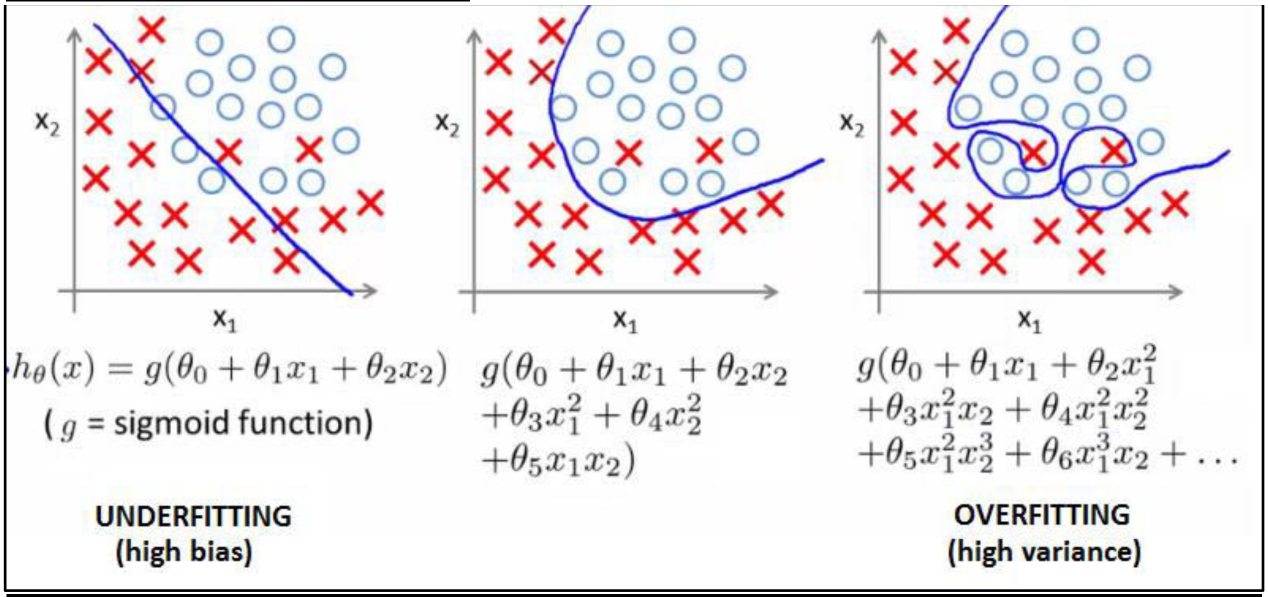

x₂
x₁
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$
(g = sigmoid function)
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots$
UNDERFITTING
(high bias)
OVERFITTING
(high variance)

Figure 8: The problem of overfitting (or underfitting) is caused by the wrong choice of the hypothesis function and it can either result in a function too simple or a very complicated function which does not generalize to predict new data.

Underfitting, or high bias, is when the form of our hypothesis function $h_\theta(x)$ maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features.

At the other extreme, overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

Between these two extremes, lies the hypothesis function which is just right, and provides the best fit to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1) Reduce the number of features:
- Manually select which features to keep.
- Use a model selection algorithm.

2) Regularization
- Keep all the features, but reduce the magnitude of parameters θ_j.
- Regularization works well when we have a lot of slightly useful features.

Regularization

Recall that our cost function for logistic regression was:

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

We can regularize this equation by adding a term to the end:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

The λ is the regularization parameter. It determines how much the costs of our θ parameters are inflated. By multiplying the parameter vector θ with regularization parameter ⍰ we penalize all the features except θ_0.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If λ is chosen to be too large, it may smooth out the function too much and cause underfitting.

Thus, when computing the equation, we should continuously update the two following equations:

$$
\begin{aligned}
&\text{Repeat } \{ \\
&\quad \theta_0 := \theta_0 - \alpha \ \frac{1}{m} \ \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\
&\quad \theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \ \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad\qquad j \in \{1, 2...n\} \\
&\}
\end{aligned}
$$

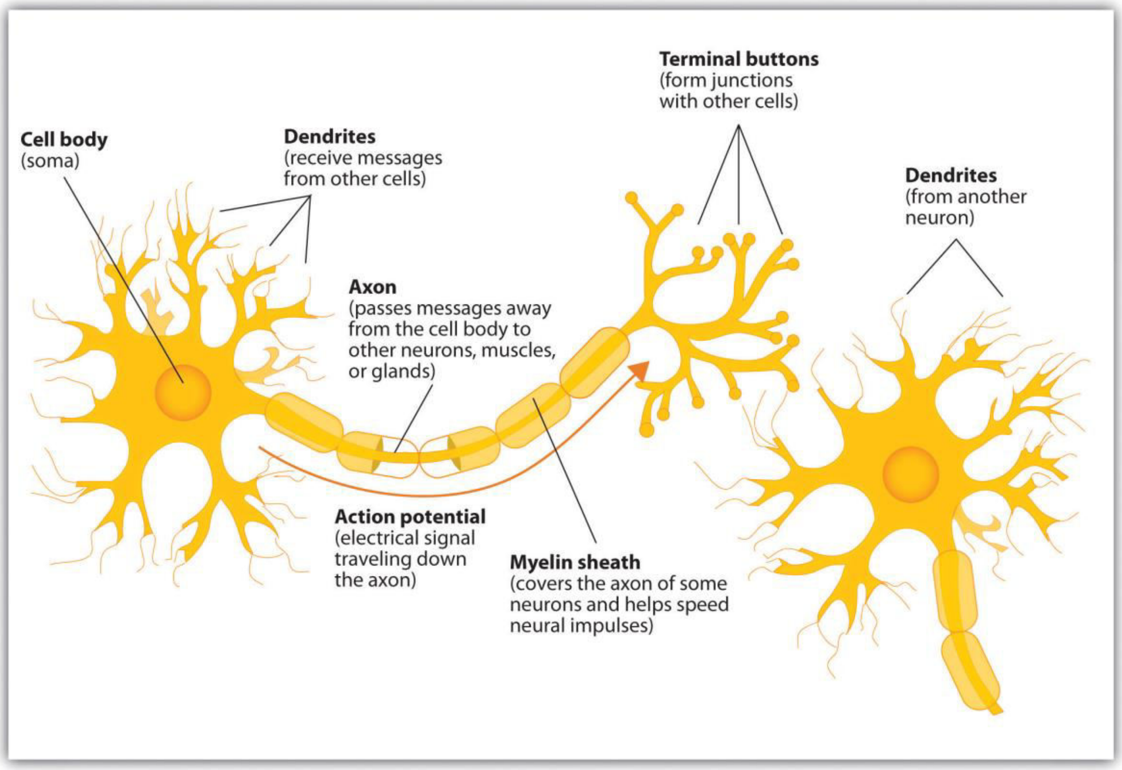
Now we can apply gradient descent on □(θ) to get the desired result.

Our goal is to minimise cost function □(θ) with using all □ number of features of parameter vector θ. With the help of validation set, we can set the best fitted value of □ to achieve this goal.

● Part II

Understanding Neural Networks

Biological Neuron



Terminal buttons (form junctions with other cells)
Cell body (soma)
Dendrites (receive messages from other cells)
Dendrites (from another neuron)
Axon (passes messages away from the cell body to other neurons, muscles, or glands)
Action potential (electrical signal traveling down the axon)
Myelin sheath (covers the axon of some neurons and helps speed neural impulses)

Figure 9: A biological neuron gets the input signals along the dendrites and based on them, sends an output signal along the axon.

Biological neural networks in our brain have inspired the design of artificial neural networks (ANN). It originated when we tried to find algorithms to mimic the brain. It was very widely used in 80s and early 90s, but the popularity diminished in late 90s due to the lack of enough computing power. It has seen a recent resurgence with highly improved computing performance and state-of-the-art techniques for many applications.

Neuron model: Logistic Unit

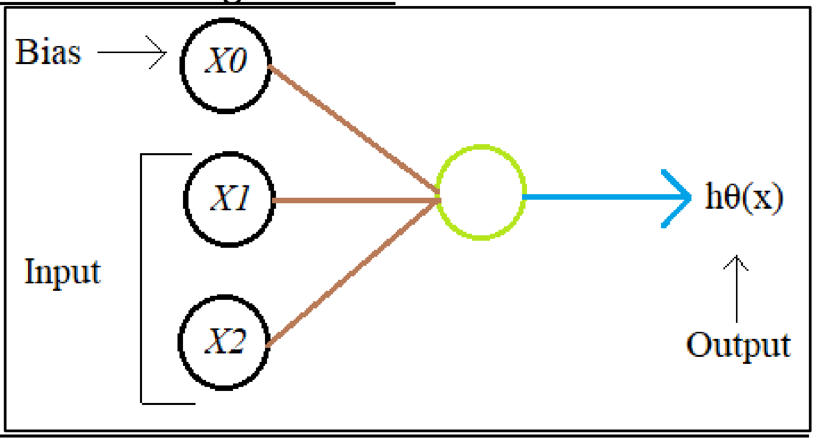


Figure 10: An artificial neuron analogous to the biological neuron - the yellow circle is the body of the neuron taking input from the black circles and pass the output to the next (level)neuron.

An artificial neuron is a mathematical function conceived as a model of biological neuron. Artificial neurons are elementary units in an artificial neural network. In an artificial neuron, the input features are $x_1 \cdots x_n$, and the output is the result of our hypothesis function. In the above model, the x_0 input node is called the "bias unit." It is always equal to 1. For this project, we used the transfer function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, it is called the sigmoid (logistic) activation function. Connecting each input to the neuron, there are "weights" (here represented by the "theta" parameters).

It can also be represented as:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow [\quad] \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

Building A Neural Network

A single neuron is a type of linear classifier, i.e. a classification algorithm that can separate two set of input points which are linearly separable. But if the set of points is not linearly separable (ie. we need a non-linear decision boundary), then we require a neural network containing at least two layers of neurons.

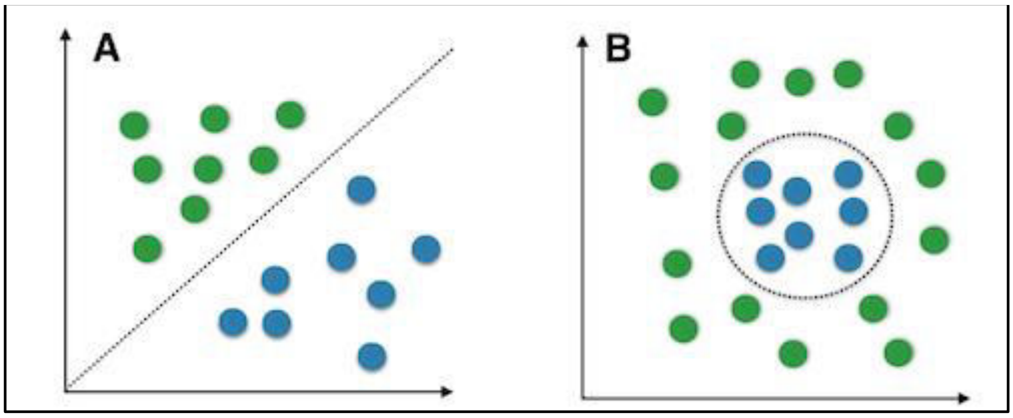


Figure 11: A: A single neuron can only form a linear decision boundary separating two set of points. **B**: If the set of points is not linearly separable, we need to use a neural network consisting of at least two layers of neurons, to form a non-linear decision boundary.

In a neural network, there are multiple neurons (also called nodes), arranged into layers. Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer". We can have intermediate layers of nodes between the input and output layers called the "hidden layers." This final hypothesis function can be any complex function which can fit into any complex distribution of points.

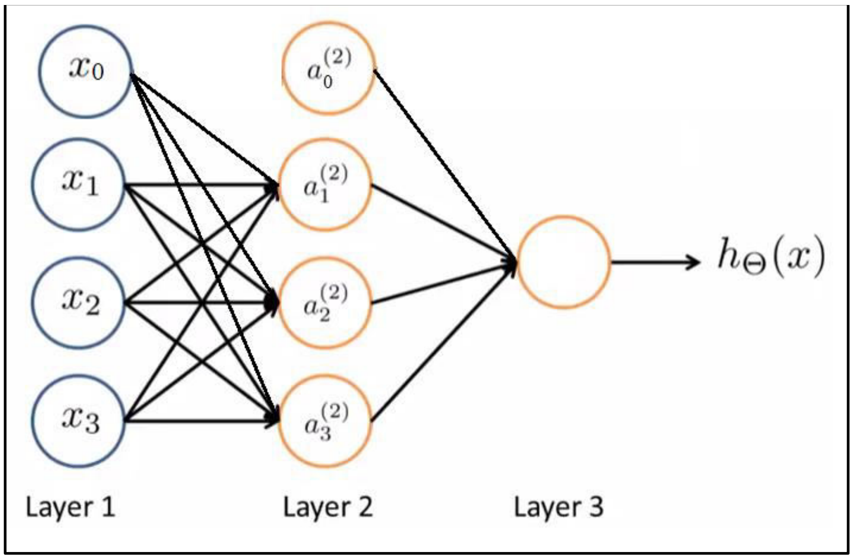


Figure 12: A two-layer neural network consist 1 input layer, 1 hidden layer and 1 output layer.

Feedforward Propagation

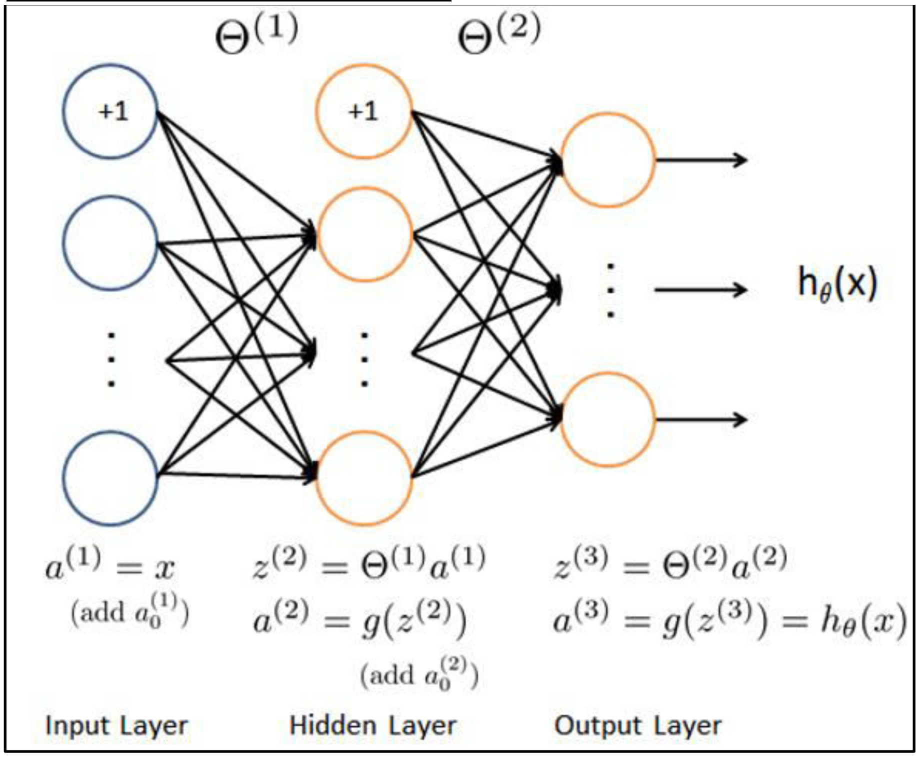


Figure 13: In forward propagation, the activation values of the neurons are calculated using the inputs and the weights of the connections; and propagated forward to their next neurons.

In this example, we label these intermediate or "hidden" layer nodes $a_0^2 \cdots a_n^2$ and call them "activation units."

$$a_i^{(j)} = \text{"activation" of unit } i \text{ in layer } j$$
$$\Theta^{(j)} = \text{matrix of weights controlling function mapping from layer } j \text{ to layer } j+1$$

If we had one hidden layer, it would look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\theta(x)$$

The values for each of the "activation" nodes is obtained as follows:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned}$$

We compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix$\Theta^{(2)}$ containing the weights for our second layer of nodes. Each layer gets its own matrix of weights, $\Theta^{(j)}$.

<u>Vectorized Implementation</u>

In this section we'll do a vectorized implementation of the above functions. Doing this allows us to more elegantly produce interesting and more complex non-linear hypotheses. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

$a_1^{(2)}$ = g($z_1^{(2)}$)

$a_2^{(2)}$ = g($z_2^{(2)}$)

$a_3^{(2)}$ = g($z_3^{(2)}$)

In other words, for layer j=2 and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \cdots + \Theta_{k,n}^{(1)} x_n$$

The vector representation of x and $z^j$is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \quad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \cdots \\ z_n^{(j)} \end{bmatrix}$$

Setting x =$a^{(1)}$, we can rewrite the equation as:

$z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$

Now we can get a vector of our activation nodes for layer j as follows:

$a^{(j)} = g(z^{(j)})$

Here the function g is applied element-wise to the vector $z^{(j)}$.

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

$z^{(j+1)} = \Theta^{(j)} a^{(j)}$

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got. This last theta matrix $\Theta^{(j)}$ will have only one row which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with:

$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})$

Multiple output units: One-vs-all

In this section we will discuss about Multiple output units: One-vs-all classification

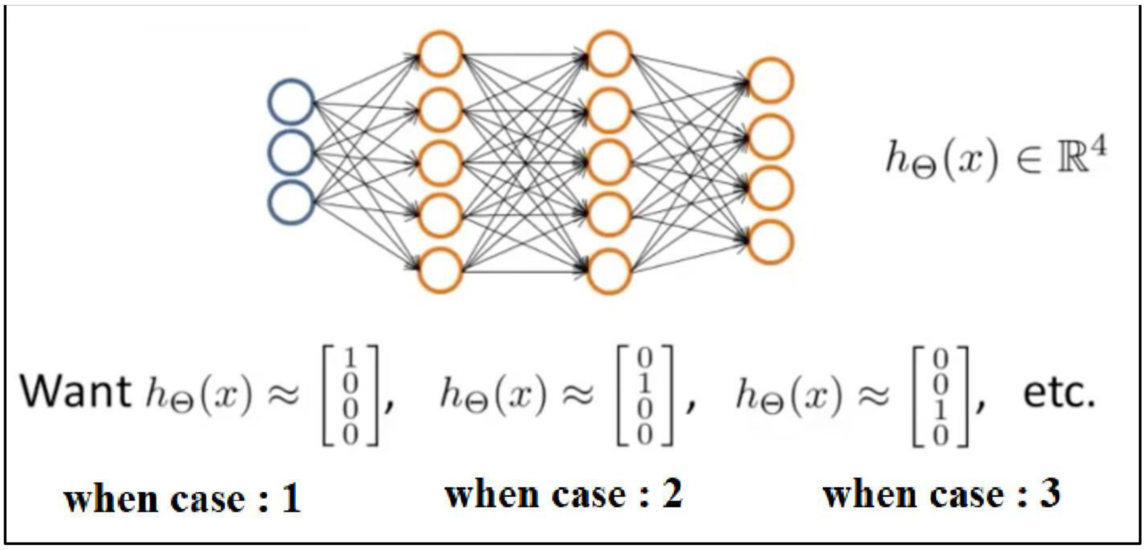

$h_\Theta(x) \in \mathbb{R}^4$
Want $h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.
when case : 1
when case : 2
when case : 3

Figure 14: To classify data into multiple classes, we let our hypothesis function return a vector of values. Here we wanted to classify our data into one of four categories.

A training set is in the form of $(x| \;\; |(1), y^{(1)})$, $(x| \;\; |(2), y^{(2)})$,...., $(x| \;\; |(m), y^{(m)})$.

In the above figure, we have $h_\Theta(x) \in R^4$, so the ANN has to identify 4 possible outcomes. So the output vector from the neural network can be:

$$y^{(i)} \text{ one of } \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Each $y^{(i)}$ represents a different case. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like:

$$\begin{bmatrix}x_0\\x_1\\x_2\\\cdots\\x_n\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(2)}\\a_1^{(2)}\\a_2^{(2)}\\\cdots\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(3)}\\a_1^{(3)}\\a_2^{(3)}\\\cdots\end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix}h_\Theta(x)_1\\h_\Theta(x)_2\\h_\Theta(x)_3\\h_\Theta(x)_4\end{bmatrix}$$

In our project, the ANN has to decide among multiple choices (10 to be specific). To address this problem, the output layer will have 10 neurons: the first output neuron will try to classify whether it is case 1 or not; and so will the rest of the neurons try to classify their corresponding cases. To get a more precise output, each neuron will give the probability for each case.

Back-propagation

Cost Function:

Let's first define a few variables that we will need to use:

L = total number of layers in the network

s_l = number of units (not counting bias unit) in layer l

K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$as being a hypothesis that results in the k^{th} output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression.

Recall that the cost function for regularized logistic regression was:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For neural networks, it is going to be slightly more complicated:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

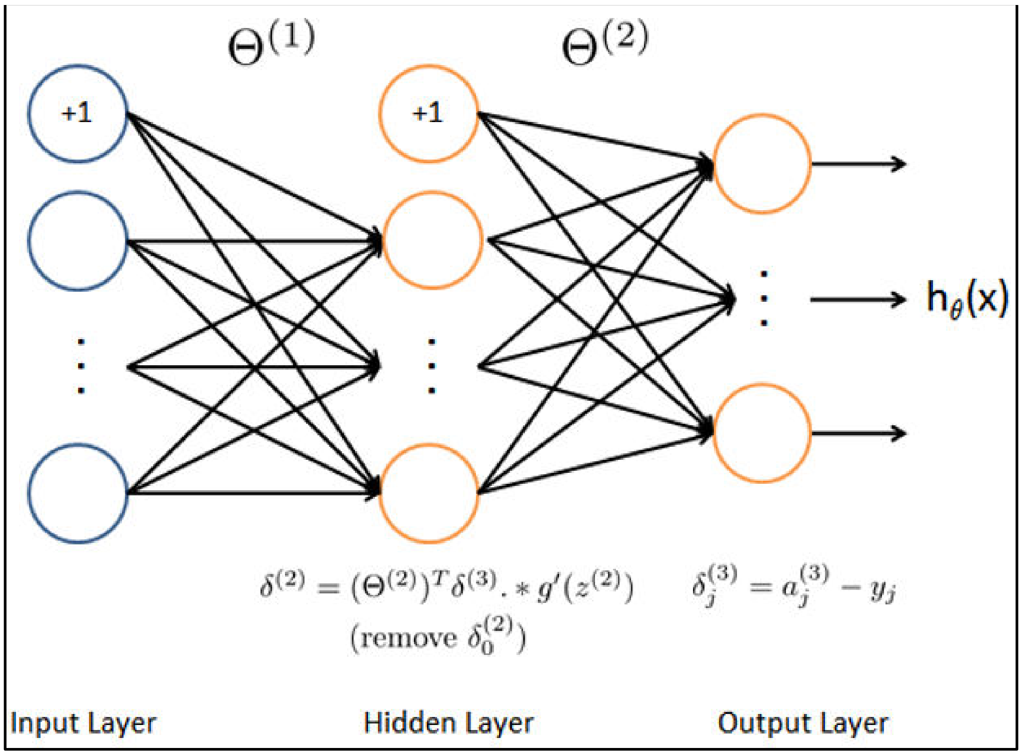
Algorithm:

"Back-propagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute: min_{Θ}J(Θ)

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the

partial derivative of J(Θ):

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$



$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$
Input Layer
Hidden Layer
Output Layer

Figure 15: In backpropagation, the error values are propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output.

Given training set {($x^{(1)}$, $y^{(1)}$) ... ($x^{(m)}$, $y^{(m)}$)}

Set $\Delta_{i,j}^{(l)} := 0$ for all (l,i,j), (hence we end up having a matrix full of zeros)

For training example t =1 to m:
1. Set $a^{(1)} := x^{(t)}$
2. Perform forward propagation to compute $a^{(l)}$ for l=2,3,…,L
3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

4. Compute

$\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$. The g-prime derivative terms can also be written out as:

$g'(z^{(l)}) = a^{(l)}\ .*\ (1 - a^{(l)})$

5. $\Delta^{(l)}_{i,j} := \Delta^{(l)}_{i,j} + a^{(l)}_j \delta^{(l+1)}_i$

or with vectorization,

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

Hence we update our new Δ matrix.

- $D^{(l)}_{i,j} := \frac{1}{m}\left(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j}\right)$, if j≠0.
- $D^{(l)}_{i,j} := \frac{1}{m}\Delta^{(l)}_{i,j}$ If j=0

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get:

$\frac{\partial}{\partial\Theta^{(l)}_{ij}} J(\Theta) = D^{(l)}_{ij}$

Putting It Together

First, we have to pick a network architecture

Choose the layout of our neural network, including how many hidden units in each layer and how many layers in total we want to have.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If we have more than 1 hidden layer, then it is recommended that we have the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights.
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$.
3. Implement the cost function.
4. Implement back-propagation to compute partial derivatives.
5. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta

● Part III

Working With Data

Data Source - MNIST

The first thing we'll need is a data set to learn from - a training data set. We'll use the MNIST data set, which contains 60,000 scanned images of handwritten digits, together with their correct classifications. The name 'MNIST' comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology.

Here's a few images from MNIST:

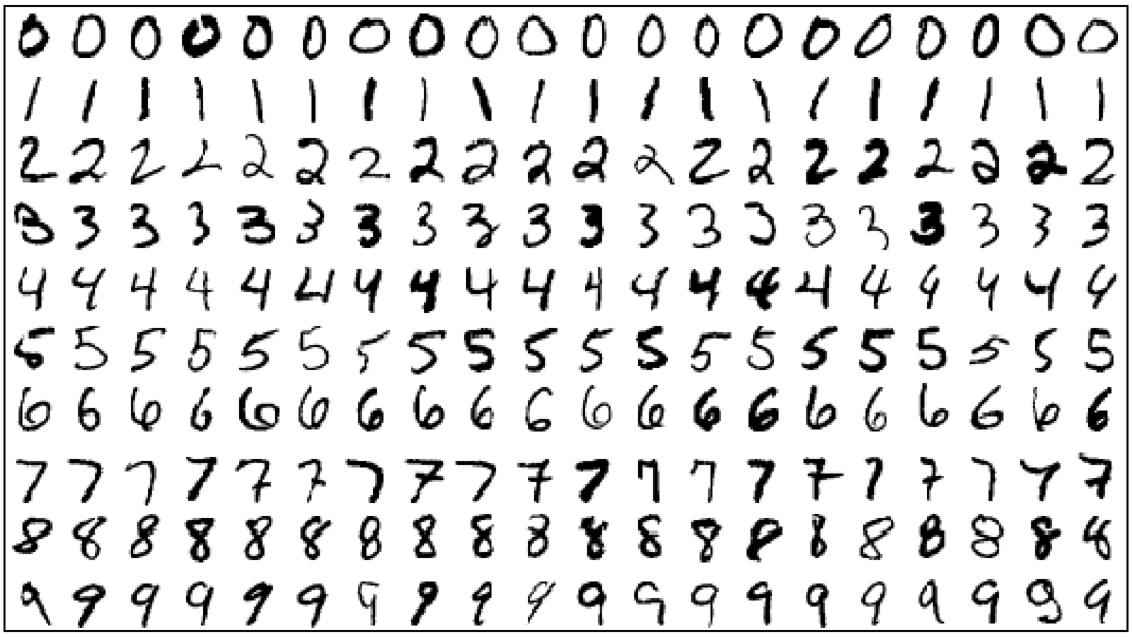


Figure 16: Few images from the MNIST dataset, 20 images of each digit. Note: The original images were white-on-black. They have been inverted to show here.

Each of these image is an 8-bit grayscale image and 28 by 28 pixels in size.

The dataset can be broadly divided into two parts:

1. The first part contains 60,000 images to be used as training data; about 6K training examples of each digit from 0 to 9. The images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and remaining half were high school students.
2. The second part of the MNIST data set is 10,000 images to be used as test data; about 1K testing examples of each digit from 0 to 9. We'll use the test data to evaluate how well our neural network has learned to recognize digits. To make this a good test of performance, the test data was taken from a different set of 250 people than the original training data (a group split between Census Bureau employees and high school students).

Inputs and outputs of NN

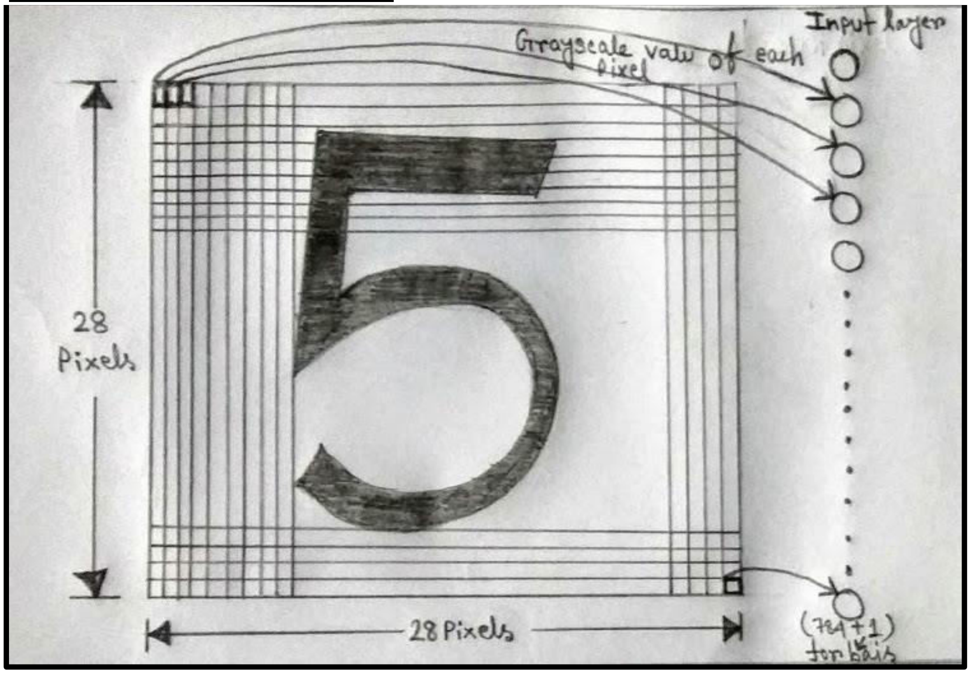


Figure 17: Each image is 28 by 28 pixels (= a total of 784 pixels) which is fed into the neural network. So the input layer of the NN consists of 784 (one for each pixel) + 1 (for bias) neurons.

There are 60,000 training examples in MNIST dataset, where each training example is a 28 pixel by 28 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 28 by 28 grid of pixels is "unrolled" into a 784-dimensional vector. Each of these training examples becomes a single row in our data matrix X. This gives us a 60,000 by 784 matrix X where every row is a training example for a handwritten digit image:

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

The second part of the training set is a 60,000-dimensional vector y that contains labels for the training set.

Feature Scaling

We can speed up gradient descent (or other optimization algorithm) by having each of our input values in roughly the same range. This is because θ (the weights) will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. One way to achieve this is feature scaling (it is also known as data normalization). It involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1.

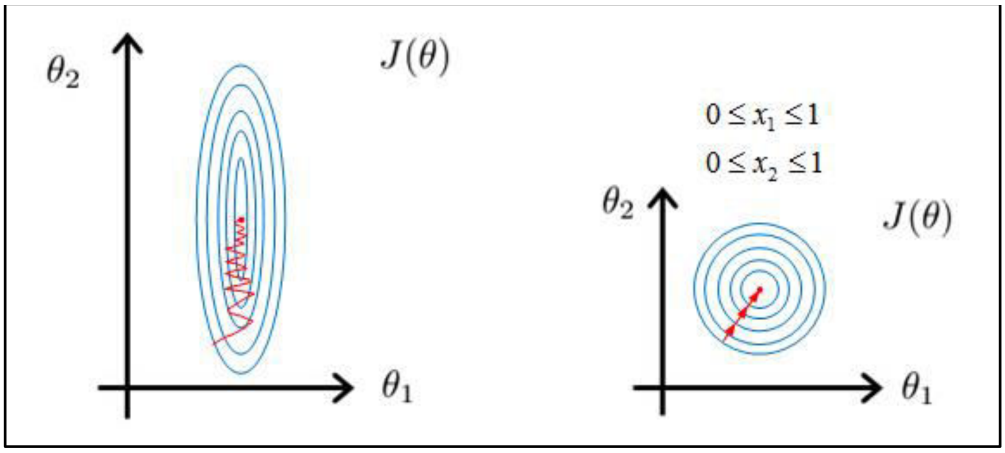


Figure 18: The gradients (the path of gradient is drawn in red) could take a long time and go back and forth to find the optimal solution. Instead if we scaled our feature, the contour of the cost function might look like circles; then the gradient can take a much more straight path and achieve the optimal point much faster.

In the MNIST dataset, all the pixels were represented by their grayscale values (0-255). So, we divided each pixel value by 255 to get the results within 0-1. This dramatically improves the number of iterations required for training.

Training

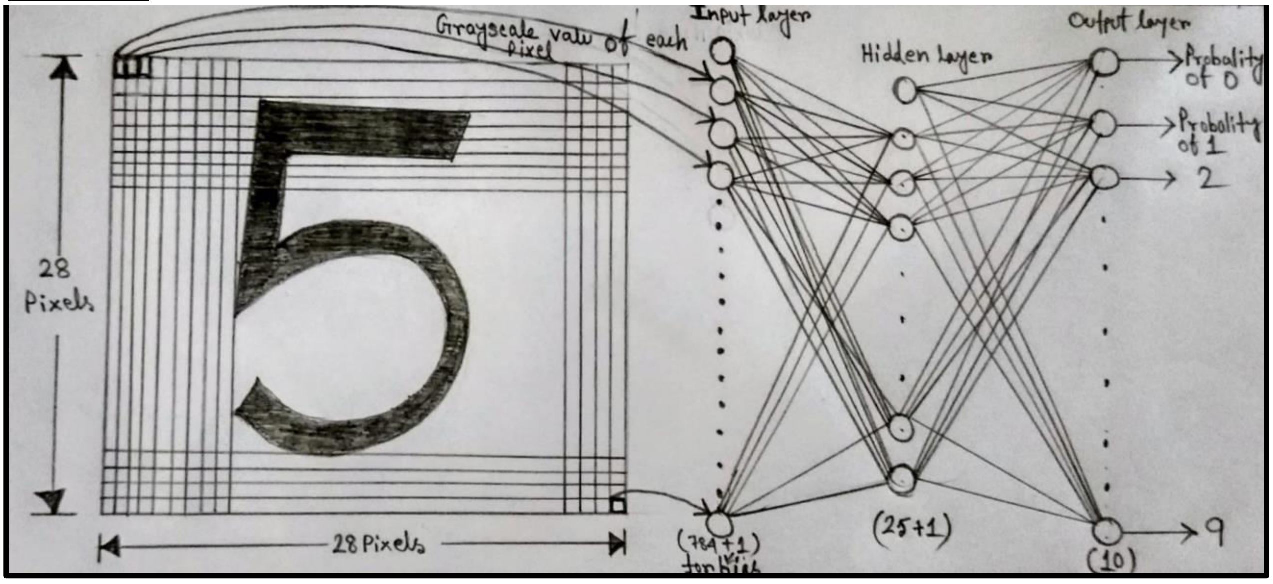


Figure 19: Our neural network with 3 layers – an input layer, a hidden layer and an output layer.

We will implement a neural network to recognize handwritten digits using the MNIST training set. The neural network will be able take the MNIST images for training and form complex non-linear hypotheses to fit the data; and achieve good accuracy in recognizing the test set images.

We will implement a neural network with 3 layers – an input layer, a hidden layer and an output layer:

1. The input layer consists of (784 + 1) input neurons, one for taking in each pixel of an image. The one extra neuron is for the bias.
2. The output layer consists of 10 neurons, one each for predicting the probability (values between 0 and 1) of the image being one of the ten digits (from 0 to 9). The prediction from the neural network will be the digit that has the highest probability.
3. The hidden layer consists of 25 (+ 1, for bias) neurons. This number of hidden units is good for achieving decent accuracy in recognizing test set images, while giving good performance on average-specced personal computers. Later on, we will carry out model selection using validation set to choose the number of hidden neurons, in addition to choosing the other hyper-parameters of the neural network.

Feedforward Propagation

We will implement feedforward propagation for the neural network.

The grayscale values of the pixels of each image is fed to the input layer of the neural network. Then, the outputs of all the input neurons is fed as inputs to the hidden neurons. And, the outputs of the hidden neurons is fed as inputs to the neurons of the output layer. Finally, the outputs from the output neurons is the probability of the image being a particular digit.

Mathematically speaking, the NN computes the output $h_\theta\big(x(i)\big)$for every training example i and returns the associated predictions. To classify data into multiple classes (from 0 to 9), we let our hypothesis function return a 10-element vector of decimal values between 0 and 1. Similar to the one-vs-all classification strategy, the prediction from the neural network will be the label that has the largest output.

Random Initialization of Weights

When training neural networks, it is important to randomly initialize the parameters (the weights) for symmetry breaking. One effective strategy for random initialization is to randomly select values for the weights uniformly in the range [-ϵ, ϵ].

One effective strategy for choosing ϵ is to base it on the number of units in the network. A good choice of ϵ is $\epsilon = \sqrt{6} /(\sqrt{L_{in}+L_{out}})$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(l)}$.

This range of values ensures that the parameters are kept small and makes the learning more efficient.

Regularized Cost Function

We used the cross-entropy error function as the cost function, with L2 regularization on the weights to prevent the problem of overfitting. We will choose the best value of the regularization parameter λ using the validation set.

Note that we should not be regularizing the terms that correspond to the bias.

Back-Propagation

We will implement the backpropagation algorithm to train the NN and minimize the cost. The intuition behind the back-propagation algorithm is as follows:

1. Given a training example $(x^{(t)},y^{(t)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$.
2. Then, for each node j in layer l, we would compute an "error term" $\delta_j^{(l)}$ that measures how much that node was "responsible" for any errors in our output.
3. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer).
4. For the hidden units, we will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer (l + 1).

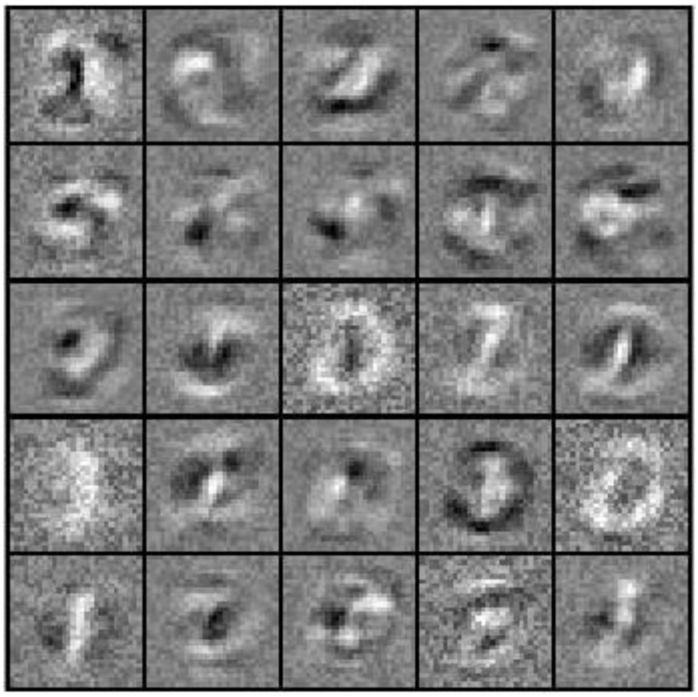


Figure 20: Visualizing 25 hidden neurons after training the NN with the MNIST dataset.

Model Selection using Validation Set

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result our predictions on the test set would be poor. The error of our hypothesis as measured on the data set with which we trained the parameters will be lower than the error on any other data set.

Given many models with different hyperparameters, we can use a systematic approach to identify the 'best' function. So, we will define a dataset to "test" the model in the training phase (i.e., the validation dataset), in order to limit problems like overfitting and others. For our project, we followed the holdout method of cross-validation. In the holdout method, we randomly assign data points to two sets d_0 and d_1, usually called the training set and the validation set, respectively. The size of each of the sets is arbitrary although typically the validation set is smaller than the training set. We then train on d_0 and validate the model on d_1. In typical cross-validation, multiple runs are aggregated together; in contrast, the holdout method, in isolation, involves a single run.

We divided the training set of the MNIST dataset containing 60,000 images into two parts: 70% training set (for training the different models), and the remaining 30% is the validation set (for testing the different models). Then there is the test set containing 10,000 images of the MNIST database for testing the best chosen model on unseen data.

The steps we followed to choose the best suited value of the hyper-parameters are:

1. Optimize the parameters in Θ using the training set for each value of the hyper-parameters.
2. Find the value of the hyper-parameter with the least error (misclassification error) using the validation set.
3. Use this value of the hyper-parameters to form the final model and check the error of this model using the test set (unseen data). This way, the hyper-parameters have not been trained using the test set.

The two hyper-parameters we are concerned with, in our neural network model are the regularization parameter λ and the number of neurons in the hidden layer:

Regularization parameter λ

We checked with these values of the regularization parameter λ:

[0 0.001 0.003 0.01 0.03 0.1 0.3 1 3 10]

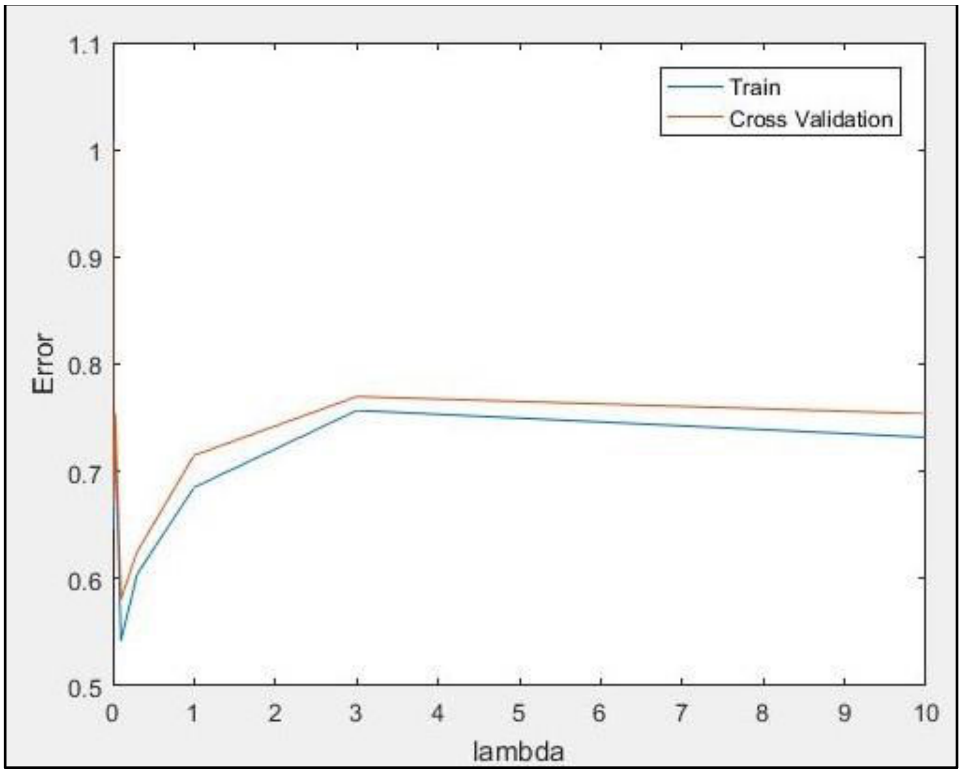
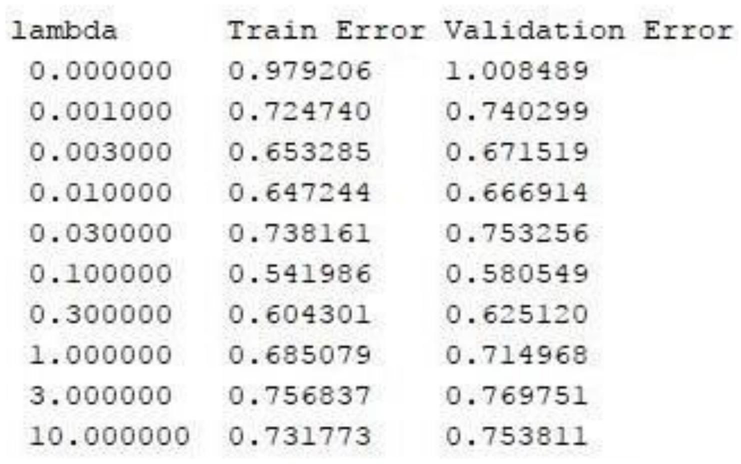


Figure 21: Checking the different models of the neural network using the validation set, we get the best accuracy at λ = 0.1.

The training and the validation error we got with different values of the regularization parameter (lambda) are:



```
lambda      Train Error Validation Error
 0.000000   0.979206    1.008489
 0.001000   0.724740    0.740299
 0.003000   0.653285    0.671519
 0.010000   0.647244    0.666914
 0.030000   0.738161    0.753256
 0.100000   0.541986    0.580549
 0.300000   0.604301    0.625120
 1.000000   0.685079    0.714968
 3.000000   0.756837    0.769751
 10.000000  0.731773    0.753811
```

So, we chose the value of the regularization parameter to be 0.1 for our final model.

Number of hidden neurons

Deciding the number of neurons in the hidden layer is a very important part of deciding our overall neural network architecture. Though the hidden layer does not directly interact with the external environment, it has a tremendous influence on the final output.

Using too few neurons in the hidden layer will result in underfitting. Underfitting occurs when there are too few neurons in the hidden layer to adequately detect the signals in a complicated data set. Too many neurons in the hidden layer may result in overfitting.

We applied the validation set to select the number of hidden neurons.

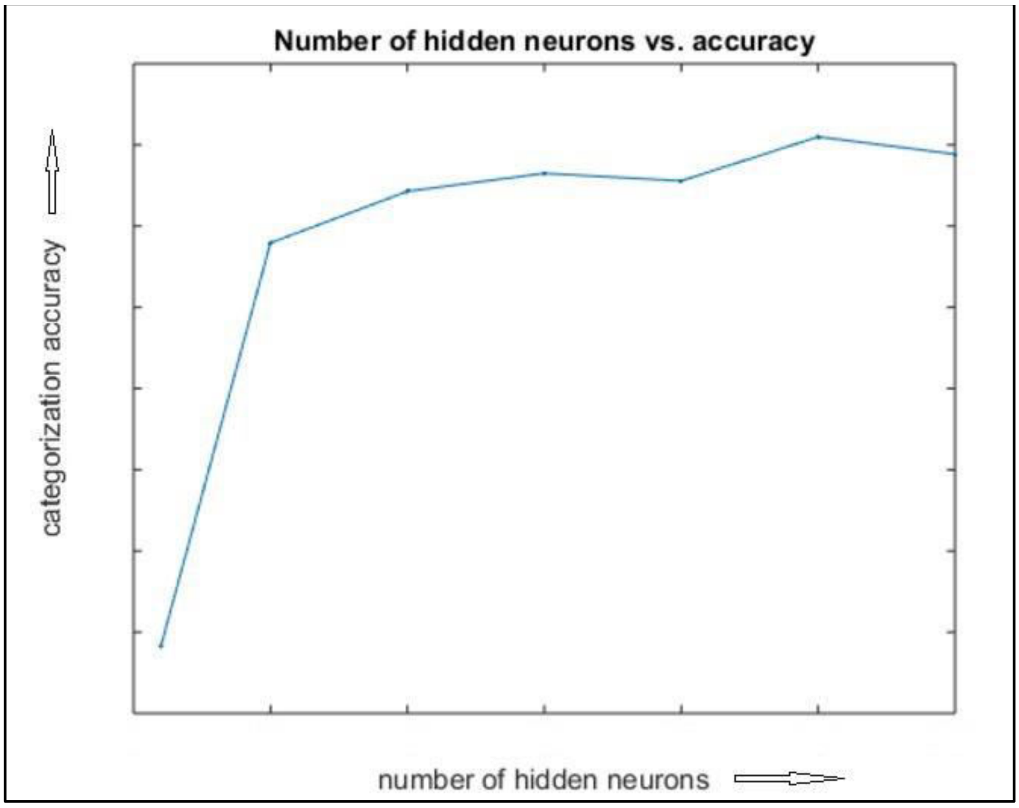


Figure 22: We gain more accuracy if we increase the number of hidden neurons, but then the accuracy decreases at some point due to overfitting the training set.

As you can see, we gain more accuracy if we increase the number of hidden neurons, but then the accuracy decreases at some point (results may differ a bit due to random initialization of weights). As we increase the number of neurons, our model will be able to capture more features, but if we capture too many features, then we end up overfitting our model to the training data and it won't do well with unseen data.

We found that we can achieve high accuracy on the validation set with 200 hidden neurons.

Note:

1. There is another hyper-parameter, the learning rate (alpha) in the gradient descent learning algorithm. We can achieve good performance at alpha = 0.5, but we are using fmincg(), an optimized learning algorithm for working with large datasets like the MNIST dataset. This algorithm doesn't need an explicit learning rate.
2. We didn't use K-Fold Cross-Validation. One of the main reasons for using cross-validation instead of using the conventional validation (holdout

method) is that there is not enough data available to partition it into separate training and test sets. It can then help avoid overfitting the function to the validation set. But we are using MNIST dataset, where enough data is already available, with separate data for training and testing.

Testing

We used the validation set and found that the best value of regularization parameter is 0.1. We also found that with increasing the number of neurons in the hidden layer, we can achieve higher accuracies on the test data.

We used the cross-entropy loss as the cost function; backpropagation for updating the weights of each neuron; and fmincg() algorithm as the cost function optimizer.

In each iteration (epoch), we trained using all the 60,000 images (batch-size) of the training set. And we used the misclassification error (ie., percentage of images correctly recognized by our NN of all the images in the set) to calculate the training and test set accuracy.

We tried with different combinations of number of hidden units and the number of iterations and recorded the accuracies on the dataset.

Number of hidden neurons	**Number of iterations (epochs)**	**Training Set Accuracy (%)**	**Test Set Accuracy (%)**
25	30	89.9533	90.18
50	30	91.54	91.57
100	30	90.29	90.94
200	100	96.261667	96.07
200	1000	100	**98.29**

We achieved maximum accuracy of 98.29% (error = 1.71%) on the test set of the MNIST dataset.

Note:

1. Due to random initialization of the weights of the neural network, we might get slightly different values while training and testing again.

2. If we train using the first 1000 images of the MNIST Training Set for 1000 iterations, we can achieve MNIST Test Set accuracy of 87.37%. This is really interesting, as we can achieve accuracies close to 90% using just a small part of MNIST Training Set.
Trained using Gradient Descent (for 1000 iterations every time using the first 1000 images of MNIST Training Set) with learning rate = 0.5, regularisation parameter = 0.1 and no. of hidden units = 25.

3. **We haven't used any ready-made toolkit for achieving these results.**

● Part IV

Applying our Neural Network on External Image

Noise Removal

- Noise present in the images can be removed before or after segmentation. But because it is more practical to do it for all the digits at once, we apply noise removal on the original picture before segmentation.
- First of all, we remove what we call 'Salt and Pepper noise' using a Median Filter.
- Next type of noise we remove is called Gaussian Noise using Linear Filter.
- Then, we use apply thresholding to convert grayscale images into binary images.

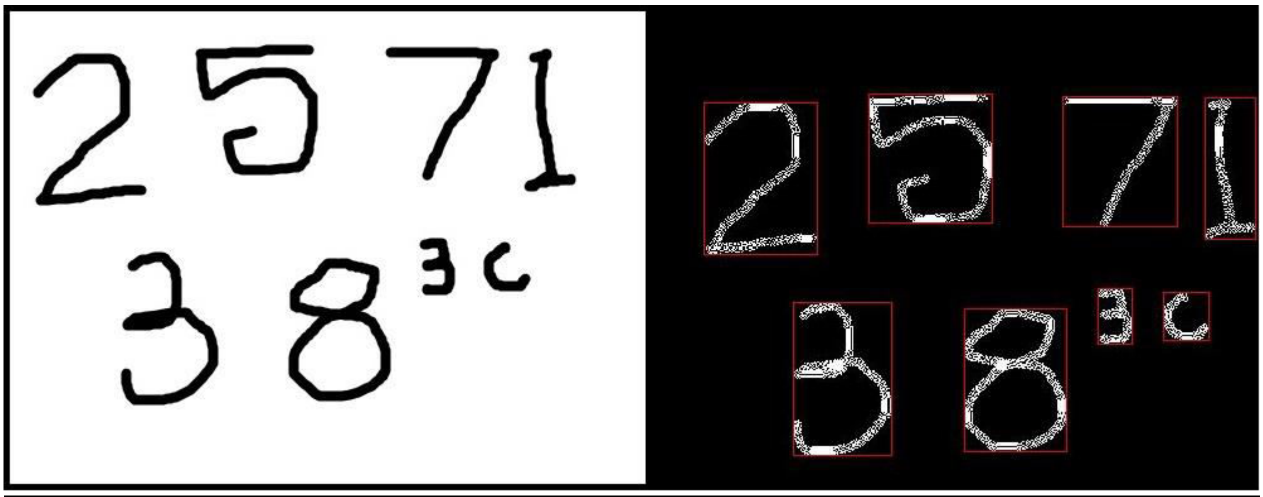


Figure 23: Input image with multiple digits before and after segmentation

Segmentation

- What we do is, take an image containing more than one digits.
- We detect the individual objects and use 'regionprops' to find the bounding boxes around each such object.
- The next step is to return a structure which contains elements, which are the digits within the bounding boxes.
- The bounding box property contains four elements, the x and y coordinates of the starting point of the box around each object; and the height and the width of the box.
- If we want to see the segmented digits within the image we can give a colour to the box around the objects.
- Next, we store each individual object into a cell array, so that we can extract each digit whenever we need.

Normalization

- In our project for digit recognition, we apply normalization in the sense that the image to be fed to the neural network, be reduced to the size that our neural network will accept.
- Hence, we normalize our image to 28x28 pixels. We have tried our best to maintain the aspect ratio of the image or else the accuracy for our result could have been lesser.

Feeding To Our Neural Network

- The image is stored in form of a matrix with grayscale values of the pixels between 0 to 255. The grayscale values are divided by 255 to get the fractional values between 0 and 1. This feature scaling is performed so that we can fit the function in fewer iterations while training.
- The network has 784 input neurons, hence we reshape the matrix from a 28x28 one, to a 1x784 one for easier input feeding.

Acknowledgement

Effective noise removal from external images is a very complex process and requires in-depth knowledge in this domain. And without good noise removal, it is impossible to achieve good success rate in detecting digits from external images.

As we didn’t prioritise on processing of external images over getting good results on test data set, the results on external images is not good and is inconsistent.

The segmentation algorithm can also be vastly improved to identify individual objects from all types of images, without false positives.