Summer of Science 2021

Mid-Summer Report

Neural Networks and Deep Learning

Swapnoneel Kayal
Roll No: 200100154
Mechanical Engineering
IIT Bombay

Mentor: Anuj Srivastava

June 12, 2021

Linear and Logistic Regression, Decision trees

Swapnoneel Kayal

18 May-24 May 2021

1 Linear Regression

1.1 Introduction and Basic Notations

Linear regression is a supervised machine learning regression algorithm which, given a training dataset along with the corresponding continuous valued labels, determines the straight line function which best fits the training data. Thus, it assumes a linear function for the task and optimizes the function's parameters to fit the training data.

NOTATION

m	the number of instances in the dataset
$\boldsymbol{x^{(i)}}$	the vector of all feature values (excluding the label) of the i^{th} instance in the dataset
$y^{(i)}$	the label (desired/expected output value) of the i^{th} instance in the dataset
$\boldsymbol{x_j}$	the vector of values from all data instances corresponding to the j^{th} feature in the dataset
$x_j^{(i)}$	the value of the j^{th} feature in the i^{th} instance of the dataset
$\boldsymbol{X}$	the entire dataset (excluding labels)
$\boldsymbol{y}$	the vector of all labels for the dataset
θ_j	the parameter of the regression model which corresponds to the j^{th} feature in the dataset
$\boldsymbol{\theta}$	the vector of all parameters of the linear regression model
$h_\theta\left(\boldsymbol{x^{(i)}}\right)$	the predicted label (output value) for the i^{th} training instance

---

1.2 Hypothesis Function

In linear regression tasks, the hypothesis function for an input data instance $\boldsymbol{x}$ having n features and parameter vector $\boldsymbol{\theta}$ is given by:

$$h_\theta(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$$

Here, x_0 is always equal to 1, and θ_0 is called the *bias term*. It can be thought of as the intercept parameter. Therefore, any linear regression task is effectively a problem of analyzing the given training dataset and figuring out the best possible parameter vector which achieves the maximum accuracy among all possible parameter vectors. This is where the cost function comes in.

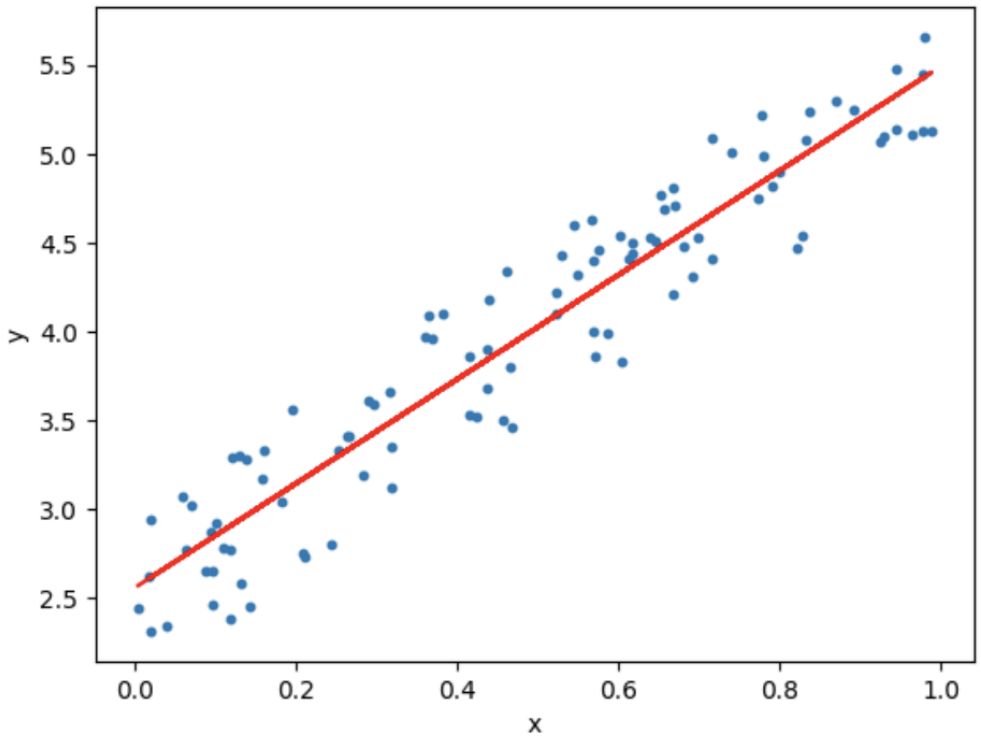


Figure 1: Linear regression

1.3 Cost Function

In a machine learning algorithm, the cost function is used to determine the most accurate set of parameters which fits the training dataset. It provides a measure of the inaccuracy of the machine learning model, and the best possible parameters are determined by minimizing the cost function, using various methods. In linear regression tasks, the most commonly used cost function is:

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \boldsymbol{x^{(i)}} - y^{(i)} \right)^2$$

The linear regression model attempts to optimize this cost function, i.e., it calculates the parameters that minimize it. Several methods are available for this optimization, such as batch gradient descent, stochastic gradient descent, mini-batch gradient descent, and the method of normal equations.

1.4 Gradient Descent

The method of gradient descent first assumes a totally random parameter vector, and then moves towards the optimal parameters step by step. During each such step, the gradient of the cost function at the current parameter vector is computed, and a multiple of the gradient is subtracted from the parameter vector, so as to always keep moving along the direction of fastest descent. Note that we have to be careful with this method, since gradient descent moves towards a local minimum, and may never reach the global minimum. The mathematical formulation for each step of gradient descent is as follows:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \alpha \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\boldsymbol{\theta}) \end{pmatrix} = \boldsymbol{\theta} - \frac{\alpha}{m} \begin{pmatrix} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) x_0^{(i)} \\ \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) x_1^{(i)} \\ \vdots \\ \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) x_n^{(i)} \end{pmatrix}$$

$$\implies \boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) \boldsymbol{x^{(i)}} = \boldsymbol{\theta} - \frac{\alpha}{m} \boldsymbol{X}^T \left(\boldsymbol{X\theta} - \boldsymbol{y} \right)$$

This formula is used for a method called batch gradient descent, where the entire dataset is considered in every single iteration, thus slowing down the training. To speed up the process, we can use stochastic gradient descent (one randomly selected instance in every iteration) or mini-batch gradient descent (a randomly selected subset of the dataset in every iteration).

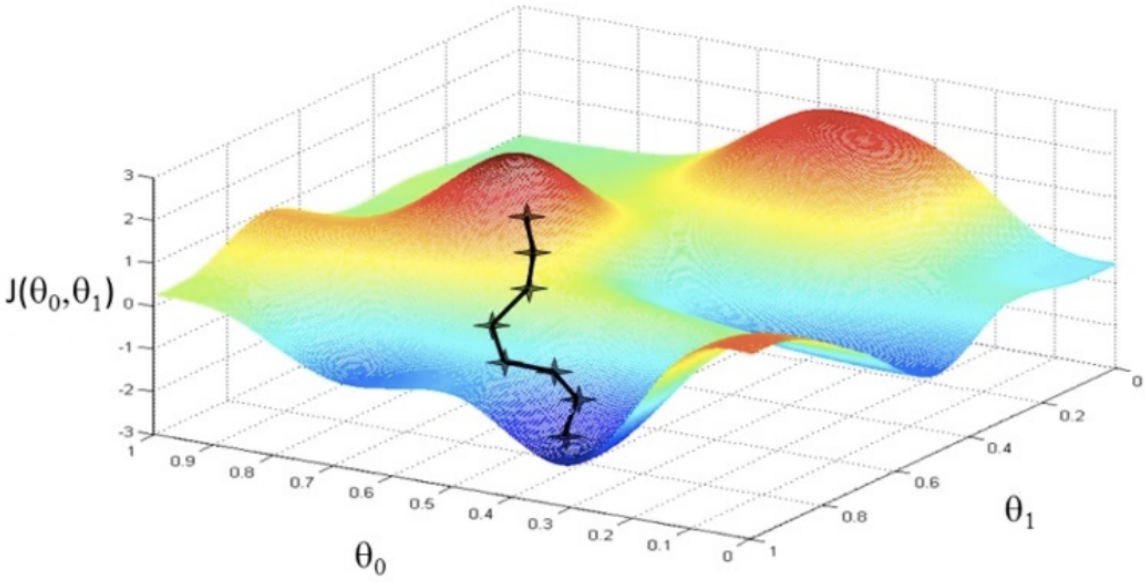


Figure 2: Movement towards lower cost function values using **gradient descent**. For the sake of making visualization possible, this figure considers a dataset consisting of only two features, the corresponding parameters being θ_0 and θ_1.

1.5 Normal Equations

The normal equations method gives us a direct formula to calculate the optimized vector of parameters:

$$\boldsymbol{\theta} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

Although this gives us a direct method to obtain the answer in just one step, there is one major demerit. The matrix $\boldsymbol{X}^T \boldsymbol{X}$ is a square matrix having order equal to the number of instances in the dataset, and this makes inverse calculation computationally very expensive for larger datasets. The complexity of matrix inversion algorithms for an $n \times n$ matrix range from nearly $O(n^{2.37})$ to $O(n^3)$, and this large complexity is the main drawback for this method. Thus, gradient descent is preferred for datasets with more than $10,000$ data instances, whereas the normal equations method may be used for smaller datasets.

1.6 Polynomial Regression

Although polynomial regression is different from linear regression, a polynomial regression task can be reduced to a linear regression task by simply adding more features, formed by multiplication of powers of the existing features (according to the degree of polynomial regression), and then applying ordinary linear regression with all the features. Some datasets are better fit by polynomial functions than linear functions, and thus, a higher degree polynomial is often a better solution for the problem, as compared to a linear best fit.

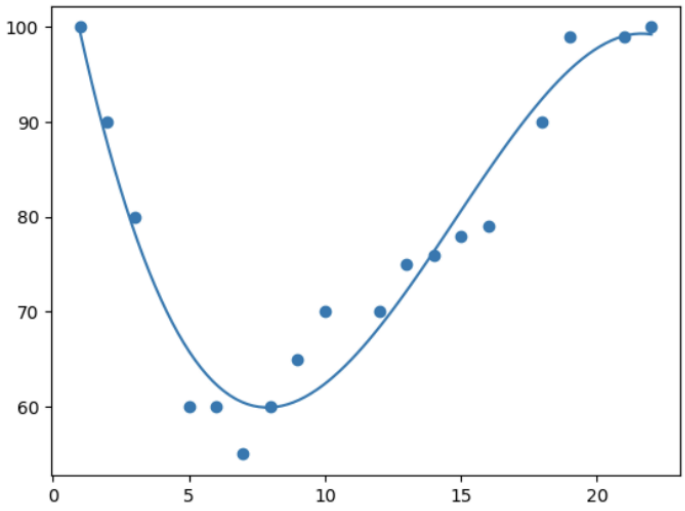


Figure 3: Polynomial regression

2 Logistic Regression

2.1 Introduction

Logistic regression is (quite confusingly, since it has the word *regression*) a classification algorithm, under supervised learning. Given a training dataset along with the corresponding discrete valued labels, it determines a function to fit the dataset. The working principles of this algorithm are somewhat similar to those of linear regression. As for class representation, in a binary classification task, there is one scalar label for each data instance, which contains either 0 or 1, whereas in multiclass classification, the label is expressed as a one-hot encoded vector, wherein the index numbering equal to the correct class number has value 1 and all other indices have value 0. We discuss binary classification first.

2.2 Hypothesis Function

The hypothesis function in this case forces the weighted sum of feature values into the interval $[0, 1]$. This number represents the predicted probability of that instance having a label of 1. The hypothesis function is usually written as:

$$h_\theta(\boldsymbol{x}) = g\left(\boldsymbol{\theta}^T \boldsymbol{x}\right)$$

$$\text{where} \quad g(z) = \frac{1}{1 + e^{-z}}$$

$$\therefore \quad h_\theta(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}}}$$

Using this probability vector, we predict the label as:

$$y = \begin{cases} 1 & h_\theta(\boldsymbol{x}) \geq 0.5 \\ 0 & h_\theta(\boldsymbol{x}) < 0.5 \end{cases}$$

The above function $g(z)$ is called the sigmoid function. Also, the threshold here is 0.5, but in some special cases, where one out of precision or recall (later) is more important than the other, the threshold value can be changed to obtain better results.

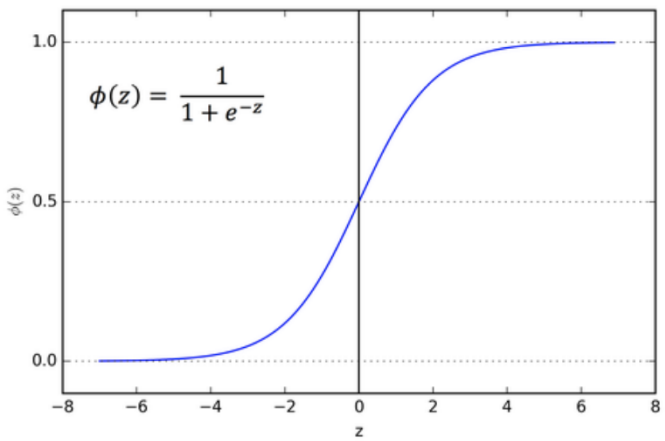


Figure 4: Sigmoid function

2.3 Decision Boundary

It is clear from the graph of the sigmoid function, that it is an increasing function, and it takes a value of 0.5 at $z = 0$. Thus, we predict label:

$$y = \begin{cases} 1 \text{ if } \boldsymbol{\theta}^T \boldsymbol{x} \geq 0 \\ 0 \text{ if } \boldsymbol{\theta}^T \boldsymbol{x} < 0 \end{cases}$$

This allows us to think of the classification in terms of a decision boundary. This boundary has the equation:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n = 0$$

The region where the above quantity is positive is the label 1 region, whereas the remaining region is the label 0 region.

2.4 Cost Function

In logistic regression, the cross-entropy cost function is used:

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

It is clear that the above function adds larger errors for larger deviations in the probability from the true labels. As in linear regression, the cost function is minimized using gradient descent (batch/stochastic/mini-batch). There is no normal equations method here.

2.5 Gradient Descent

As in linear regression, gradient descent is implemented by subtracting a multiple of the gradient of the cost function from the parameter vector at each step:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \alpha \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\boldsymbol{\theta}) \end{pmatrix} = \boldsymbol{\theta} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) \boldsymbol{x^{(i)}}$$

As can be seen, the gradient expression turns out to be somewhat similar to that in linear regression.

2.6 Multiclass Classification

For multiclass classification, the one-vs-rest algorithm is used. In this method, the probability that an instance belongs to class i is calculated for each class i, by creating several classifiers (number of classifiers equals the number of classes) and then training each classifier separately. For label prediction, the class label with the maximum probability is returned as the final prediction.

3 Decision trees

3.1 Introduction

A decision tree is a tree-like graph with nodes representing the place where we pick an attribute and ask a question; edges represent the answers the to the question; and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface.
Decision trees classify the examples by sorting them down the tree from the root to some leaf node, with the leaf node providing the classification to the example. Each node in the tree acts as a test case for some attribute, and each edge descending from that node corresponds to one of the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new nodes.
A general algorithm for a decision tree can be described as follows:

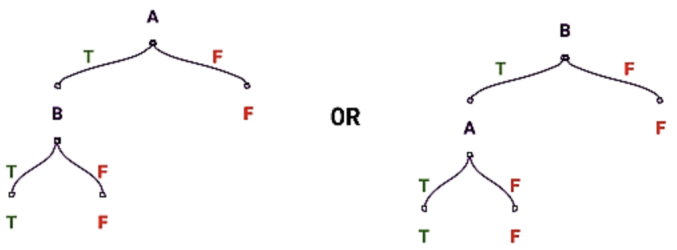
- Pick the best attribute/feature. The best attribute is one which best splits or separates the data.
- Ask the relevant question.
- Follow the answer path.
- Go to step 1 until you arrive to the answer.

The best split is one which separates two different labels into two sets.

3.2 Expressiveness of decision trees

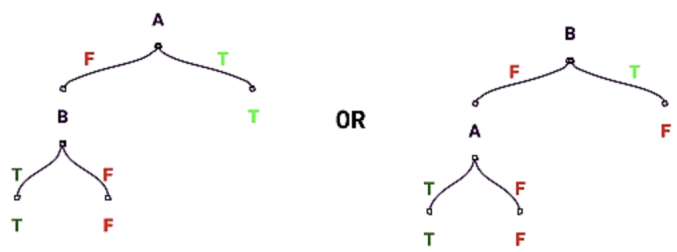
Decision trees can represent any boolean function of the input attributes. Let's use decision trees to perform the function of three boolean gates AND, OR and XOR.
Boolean Function: **AND**

A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T



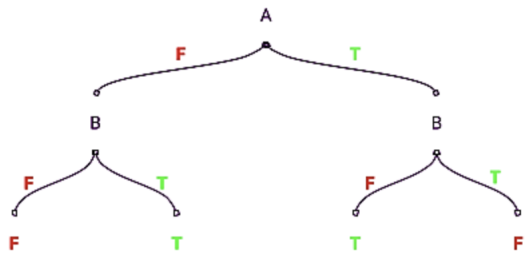
Boolean Function: **OR**

A	B	A OR B
F	F	F
F	T	T
T	F	T
T	T	T

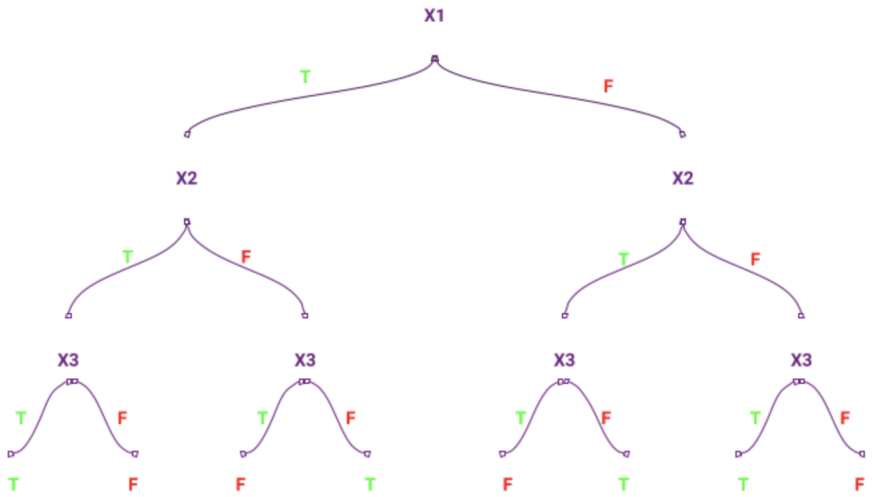


Boolean Function: **XOR**

A	B	A XOR B
F	F	F
F	T	T
T	F	T
T	T	F



Let's produce a decision tree performing XOR functionality using 3 attributes:



In the decision tree, shown above, for three attributes there are 7 nodes in the tree, i.e., for $n = 3$, number of nodes $= 2^3 - 1$. Similarly, if we have n attributes, there are 2^n nodes (approx.) in the decision tree. So, the tree requires exponential number of nodes in the worst case.
We can represent boolean operations using decision trees. But, what other kind of functions can we represent and if we search over the various possible decision trees to find the right one, how many decision trees do we have to worry about. Let's answer this question by finding out the possible number of decision trees we can generate given N different attributes (assuming the attributes are boolean). Since a truth table can be transformed into a decision tree, we will form a truth table of N attributes as input.

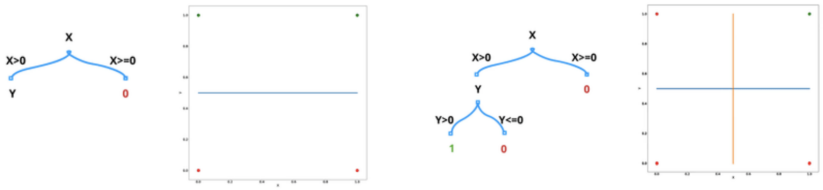
X1	X2	X3	....	XN	OUTPUT
T	T	T	...	T	
T	T	T	...	F	
...	...	...	...	...	
...	...	...	...	...	
...	...	...	...	...	
F	F	F	...	F	

The above truth table has 2^n rows (i.e. the number of nodes in the decision tree), which represents the possible combinations of the input attributes, and since each node can a hold a binary value, the number of ways to fill the values in the decision tree is 2^{2^n}. Thus, the space of decision trees, i.e, the hypothesis space of the decision tree is very expressive because there are a lot of different functions it can represent. But, it also means one needs to have a clever way to search the best tree among them.

3.3 Decision tree boundary

Decision trees divide the feature space into axis-parallel rectangles or hyperplanes. Let's demonstrate this with help of an example. Let's consider a simple AND operation on two variables(Refer to the table given below for clarity). Assume X and Y to be the coordinates on the x and y axes, respectively, and plot the possible values of X and Y (as seen the table below). The given figures below represent the formation of the decision boundary as each decision is taken. We can see that as each decision is made, the feature space gets divided into smaller rectangles and more data points get correctly classified.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1



SVM and PCA

Swapnoneel Kayal

25 May-31 May 2021

1 SVM : Support Vector Machines

The original SVM algorithm was invented by Vladimir N. Vapnik and the current standard incarnation (soft margin) was proposed by Corinna Cortes and Vapnik in 1993 and published in 1995.

1.1 What's SVM

A support vector machines (SVM) is a supervised learning model used for classification. It differs from logistic regression in the sense that it exploits the geometrical properties of the data; unlike the statistical approach of logistic regression, it tries to draw the best possible decision boundary for classification, and then uses this boundary for future predictions.

1.2 Hypothesis Function

The hypothesis function for an SVM is the same as that for logistic regression:

$$h_\theta(\boldsymbol{x}) = g\left(\boldsymbol{\theta}^T\boldsymbol{x}\right) = \frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x}}}$$

1.3 Cost Function

The cost function is modified for an SVM. Here, we want a decision boundary which can distinguish between the two classes quite well, and we wouldn't like to accept a boundary that barely distinguishes data points. For $y=1$, we want $\boldsymbol{\theta}^T\boldsymbol{x} \gg 0$, and for $y=0$, we want $\boldsymbol{\theta}^T\boldsymbol{x} \ll 0$.

For regularized logistic regression, we had the following cost function:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(\boldsymbol{x^{(i)}}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(\boldsymbol{x^{(i)}}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\left(-\log\frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}}}\right) + \left(1-y^{(i)}\right)\left(-\log\left(1-\frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}}}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \end{aligned}$$

For an SVM, we modify the cost function as follows:

$$J(\boldsymbol{\theta}) = C\sum_{i=1}^{m}\left[y^{(i)}cost_1\left(\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}\right) + \left(1-y^{(i)}\right)cost_0\left(\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}\right)\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

Here, C represents the inverse of the regularization parameter λ. The two component cost functions above are:

$$cost_1(z) = \max\left(0, 1 - \boldsymbol{\theta}^T \boldsymbol{x}\right)$$
$$cost_0(z) = \max\left(0, -1 + \boldsymbol{\theta}^T \boldsymbol{x}\right)$$

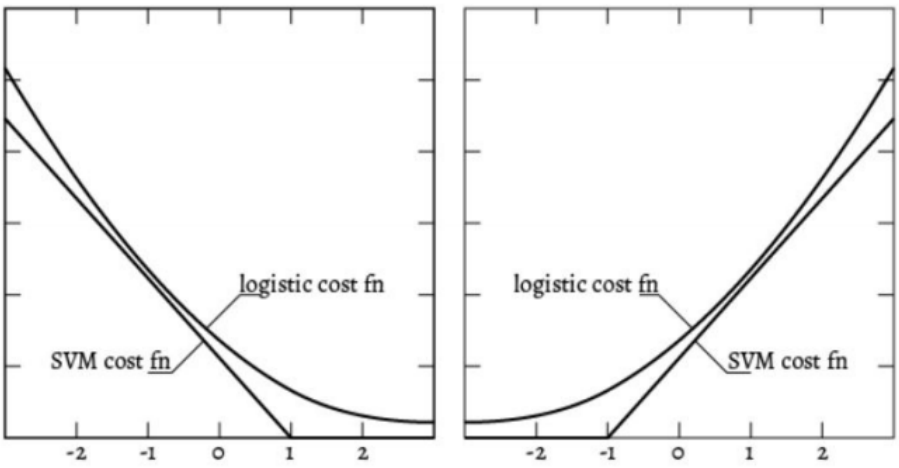


Figure 5.1: The two component cost functions of an SVM

1.4 Working of an SVM

If we ignore θ_0 and assume we are using a large value of C, the objective of an SVM can be expressed as follows:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \frac{1}{2}\|\boldsymbol{\theta}\| \\ \text{s.t.} \quad & \boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \geq 1 \qquad \text{if } y^{(i)} = 1 \\ \text{and} \quad & \boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \leq -1 \qquad \text{if } y^{(i)} = 0 \end{aligned}$$

Now, $\boldsymbol{\theta}^T \boldsymbol{x}^{(i)}$ is the scalar product of the vectors $\boldsymbol{\theta}$ and $\boldsymbol{x}^{(i)}$, and is thus also the projection of $\boldsymbol{x}^{(i)}$ along $\boldsymbol{\theta}$, multiplied by $\|\boldsymbol{\theta}\|$. Thus, the optimization of the above expression would mean maximizing the projection magnitudes of $\boldsymbol{x}^{(i)}$ along $\boldsymbol{\theta}$, so that $\|\boldsymbol{\theta}\|$ can be minimized. Thus, the SVM algorithm finds a hyperplane in the dataset which can clearly distinguish between data point classes, by maximizing the distance of the hyperplane from the data points of both classes. θ_0 appears just so that the optimal hyperplane is not constrained to pass through the origin. Another point to be noted is, the larger C is, the lesser is the model's regularization, and the more it tries to fit to the training dataset (i.e., to classify each training instance correctly). Thus, C must be chosen carefully to avoid overfitting.

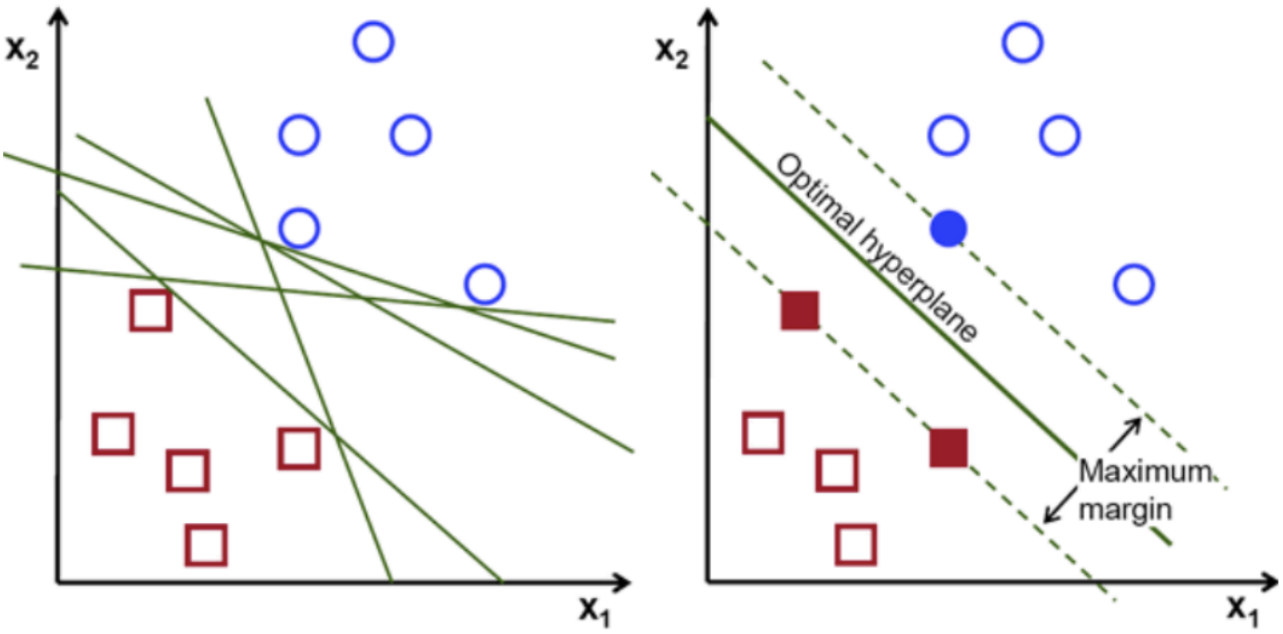


Figure 5.2: Hyperplane optimization by an SVM

1.5 SVMs with Kernels

Kernel functions are used with SVMs to make non-linear decision boundary formation very effective. A kernel function takes a data point and a landmark as input and returns a new data point. Thus, a dataset can be transformed into another using a kernel function. The most commonly used kernel is the Gaussian kernel, which we will denote by $sim\,(\boldsymbol{x}, \boldsymbol{y})$, since it is a measure of similarity between the two points $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$sim\,(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{2\sigma^2}\right)$$

Given a dataset $\boldsymbol{X}$, let $\boldsymbol{l^{(1)}} = \boldsymbol{x^{(1)}}$, $\boldsymbol{l^{(2)}} = \boldsymbol{x^{(2)}}$, $\ldots$, $\boldsymbol{l^{(m)}} = \boldsymbol{x^{(m)}}$ be landmarks.
For each training example $\left(\boldsymbol{x^{(i)}}, y^{(i)}\right)$, we calculate:

$$\begin{aligned}
f_0^{(i)} &= 1 \\
f_1^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(1)}}\right) \\
f_2^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(2)}}\right) \\
&\vdots \\
f_m^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(m)}}\right)
\end{aligned}$$

Thus, we obtain a new dataset $\boldsymbol{F}$ of size $m \times (m+1)$, and we use this to train the model.
Our new hypothesis predicts $y^{(i)} = 1$ if $\boldsymbol{\theta}^T \boldsymbol{f^{(i)}} \geq 0$, and $y^{(i)} = 0$ otherwise.
During training too, the cost function should be calculated using a new modified SVM cost function, replacing $\boldsymbol{\theta}^T \boldsymbol{x^{(i)}}$ with $\boldsymbol{\theta}^T \boldsymbol{f^{(i)}}$.

Note that:

- Large C and small σ can cause low bias and high variance.
- Small C and large σ can cause high bias and low variance.

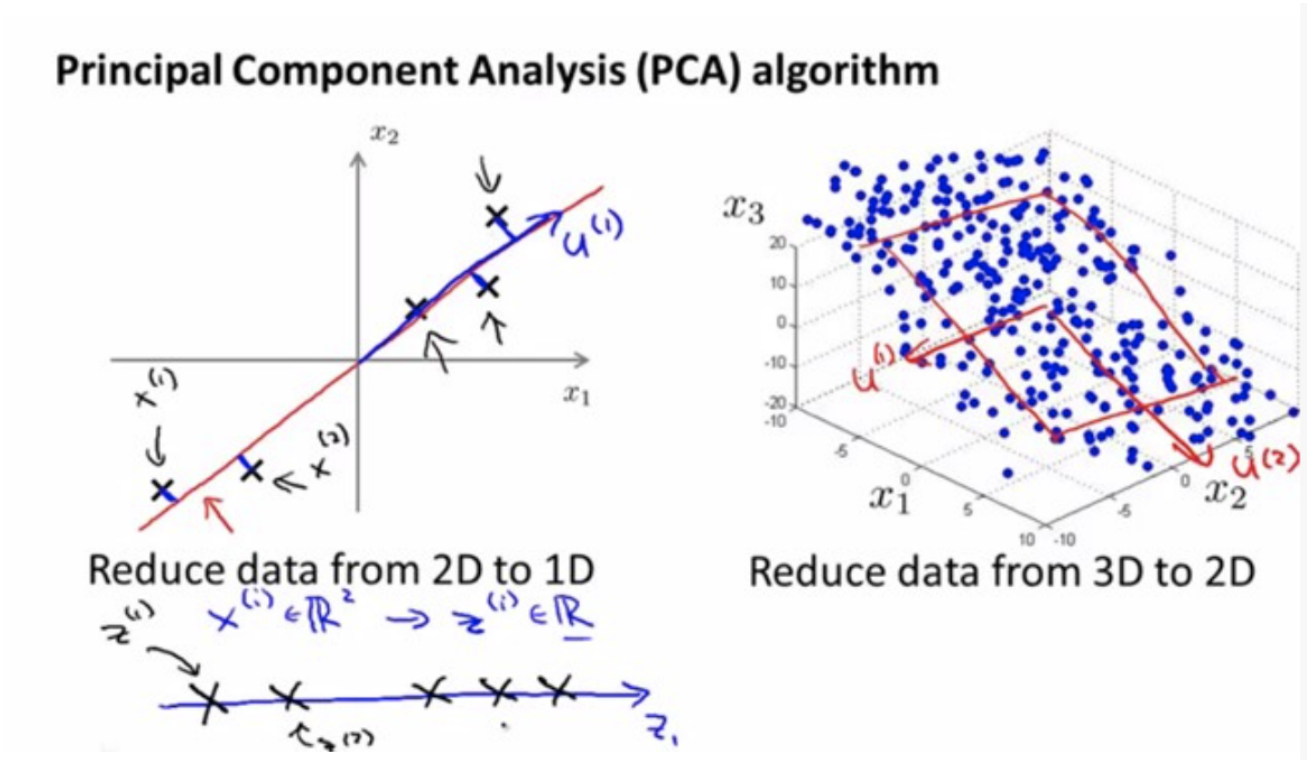
2 PCA : Principal Component Analysis

2.1 What's PCA

The main idea of principal component analysis (PCA) is to reduce the dimensionality of a data set consisting of many variables correlated with each other, either heavily or lightly, while retaining the variation present in the dataset, up to the maximum extent. The same is done by transforming the variables to a new set of variables, which are known as the principal components (or simply, the PCs) and are orthogonal, ordered such that the retention of variation present in the original variables decreases as we move down in the order. So, in this way, the 1st principal component retains maximum variation that was present in the original components. The principal components are the eigenvectors of a covariance matrix, and hence they are orthogonal.

Importantly, the dataset on which PCA technique is to be used must be scaled. The results are also sensitive to the relative scaling. As a layman, it is a method of summarizing data. Imagine some wine bottles on a dining table. Each wine is described by its attributes like colour, strength, age, etc. But redundancy will arise because many of them will measure related properties. So what PCA will do in this case is summarize each wine in the stock with less characteristics.

Intuitively, Principal Component Analysis can supply the user with a lower-dimensional picture, a projection or "shadow" of this object when viewed from its most informative viewpoint



2.2 Some Terminologies

- **Dimensionality:**
  It is the number of random variables in a dataset or simply the number of features, or rather more simply, the number of columns present in your dataset.

- **Correlation:**
  It shows how strongly two variables are related to each other. The value of the same ranges from -1 to +1. Positive indicates that when one variable increases, the other increases as well, while negative indicates the other decreases on increasing the former. And the modulus value of indicates the strength of relation.

- **Orthogonal:**
  Uncorrelated to each other, i.e., correlation between any pair of variables is 0.

- **Eigenvectors:**
  Consider a non-zero vector v. It is an eigenvector of a square matrix A, if Av is a scalar multiple of v. Then, v is the eigenvector and that scalar multiple is the eigenvalue associated with it.

- **Covariance Matrix:**
  This matrix consists of the covariances between the pairs of variables. The (i,j)th element is the covariance between i-th and j-th variable.

2.3 Properties of Principal Component

Technically, a principal component can be defined as a linear combination of optimally-weighted observed variables. The output of PCA are these principal components, the number of which is less than or equal to the number of original variables. Less, in case when we wish to discard or reduce the dimensions in our dataset. The PCs possess some useful properties which are listed below:

- The PCs are essentially the linear combinations of the original variables, the weights vector in this combination is actually the eigenvector found which in turn satisfies the principle of least squares.

- The PCs are orthogonal

- The variation present in the PCs decrease as we move from the 1st PC to the last one, hence the importance.

The least important PCs are also sometimes useful in regression, outlier detection, etc.

2.4 Implementing PCA on a 2-D Dataset

- **Step 1: Normalize the data**
  First step is to normalize the data that we have so that PCA works properly. This is done by subtracting the respective means from the numbers in the respective column. So if we have two dimensions X and Y, all X become x^- and all Y become y^-. This produces a dataset whose mean is zero.

- **Step 2: Calculate the covariance matrix**
  Since the dataset we took is 2-dimensional, this will result in a 2x2 Covariance matrix. Note that Var[X_1]=Cov[X_1,X_1] and Var[X_2]=Cov[X_2,X_2].

  $$\Rightarrow \text{Matrix(Covariance)} = \begin{bmatrix} Var[\text{X}_1] & Cov[\text{X}_1,X_2] \\ Cov[\text{X}_2,X_1] & Var[\text{X}_2] \end{bmatrix}$$

- **Step 3: Calculate the eigenvalues and eigenvectors**
  Next step is to calculate the eigenvalues and eigenvectors for the covariance matrix. The same is possible because it is a square matrix. λ is an eigenvalue for a matrix **A** if it is a solution of the characteristic equation: $\mathbf{det(\lambda I - A) = 0}$ where, **I** is the identity matrix of the same dimension as **A** which is a required condition for the matrix subtraction as well in this case and **'det'** is the determinant of the matrix. For each eigenvalue λ, a corresponding eigen-vector **v**, can be found by solving: $\mathbf{(\lambda I - A)v = 0}$

- **Step 4: Choosing components and forming a feature vector**
  We order the eigenvalues from largest to smallest so that it gives us the components in order or significance. Here comes the dimensionality reduction part. If we have a dataset with n variables, then we have the corresponding n eigenvalues and eigenvectors. It turns out that the eigenvector corresponding to the highest eigenvalue is the principal component of the dataset and it is our call as to how many eigenvalues we choose to proceed our analysis with. To reduce the dimensions, we choose the first p eigenvalues and ignore the rest. We do lose out some information in the process, but if the eigenvalues are small, we do not lose much.

  Next we form a feature vector which is a matrix of vectors, in our case, the eigenvectors. In fact, only those eigenvectors which we want to proceed with. Since we just have 2 dimensions in the running example, we can either choose the one corresponding to the greater eigenvalue or simply take both.

  $\Rightarrow$ Feature Vector = (eig_1,eig_2)

- **Step 5: Forming Principal Components:**
  This is the final step where we actually form the principal components using all the math we did till here. For the same, we take the transpose of the feature vector and left-multiply it with the transpose of scaled version of original dataset.

  $$\text{NewData} = FeatureVector^T \times ScaledData^T$$

  Here, *NewData* is the Matrix consisting of the principal components, *FeatureVector* is the matrix we formed using the eigenvectors we chose to keep and *ScaledData* is the scaled version of original. 'T' in the superscript denotes transpose of a matrix which is formed by interchanging the rows to columns and vice versa.

  If we go back to the theory of eigenvalues and eigenvectors, we see that, essentially, eigenvectors provide us with information about the patterns in the data. In particular, in the running example of 2-D set, if we plot the eigenvectors on the scatterplot of data, we find that the principal eigenvector (corresponding to the largest eigenvalue) actually fits well with the data. The other one, being perpendicular to it, does not carry much information and hence, we are at not much loss when deprecating it, hence reducing the dimension.

All the eigenvectors of a matrix are perpendicular to each other. So, in PCA, what we do is represent or transform the original dataset using these orthogonal (perpendicular) eigenvectors instead of representing on normal $\mathbf{x}$ and $\mathbf{y}$ axes. We have now classified our data points as a combination of contributions from both $\mathbf{x}$ and $\mathbf{y}$. The difference lies when we actually disregard one or many eigenvectors, hence, reducing the dimension of the dataset. Otherwise, in case, we take all the eigenvectors in account, we are just transforming the co-ordinates and hence, not serving the purpose.

2.5 Applications of Principal Component Analysis

PCA is predominantly used as a dimensionality reduction technique in domains like facial recognition, computer vision and image compression. It is also used for finding patterns in data of high dimension in the field of finance, data mining, bioinformatics, psychology, etc.

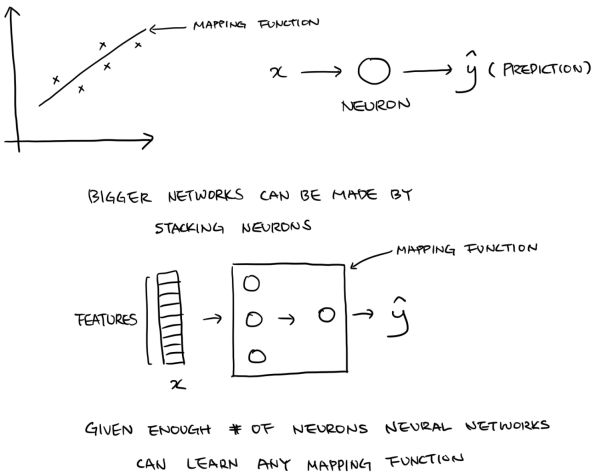
Neural Networks and Deep Neural Networks

Swapnoneel Kayal

1 June-7 June 2021

1 Neural Networks

Neural network is a stack of **neurons** that takes in some value and outputs some value. Given enough number of neurons, neural networks are incredibly good at **mapping x** to **y**. During **supervised learning**, we use this property to learn a **function** that maps **x** to $\hat{\mathbf{y}}$. To learn that function, we need data. There are two types of data: **structured** and **unstructured**. Structured data are well organized into arrays and often comes from a database. Unstructured data are things like audio, image, text, and etc. that does not come in typical structure.



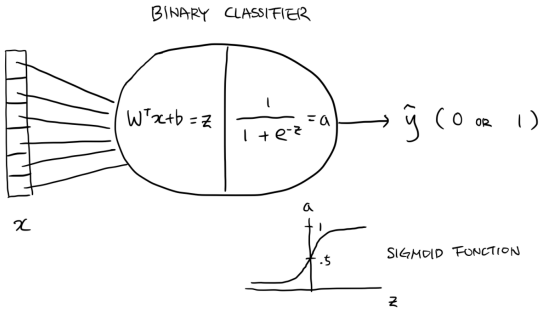
Binary classifier predicts the **class** of an input x by computing $\hat{\mathbf{y}}$ (**0** or **1** for two classes). We use a **neuron** that maps **x** to $\hat{\mathbf{y}}$ such that:

$$\hat{\mathbf{y}} = \boldsymbol{\sigma}\left(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}\right)$$

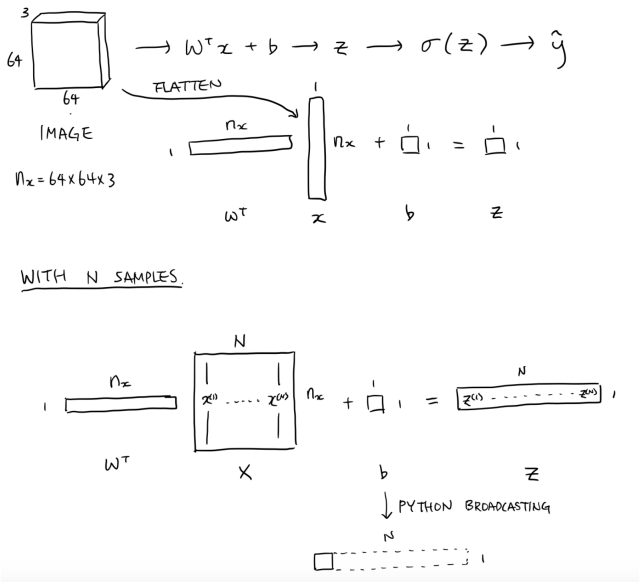
$$\text{where} \quad \boldsymbol{\sigma}(z) = \frac{1}{1+e^{-z}}$$

$$\therefore \quad \hat{\mathbf{y}} = \frac{1}{1+e^{-\left(\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{b}\right)}}$$

Sigmoid function outputs a value between 0 and 1, which happens to work beautifully for our purpose of binary classification.



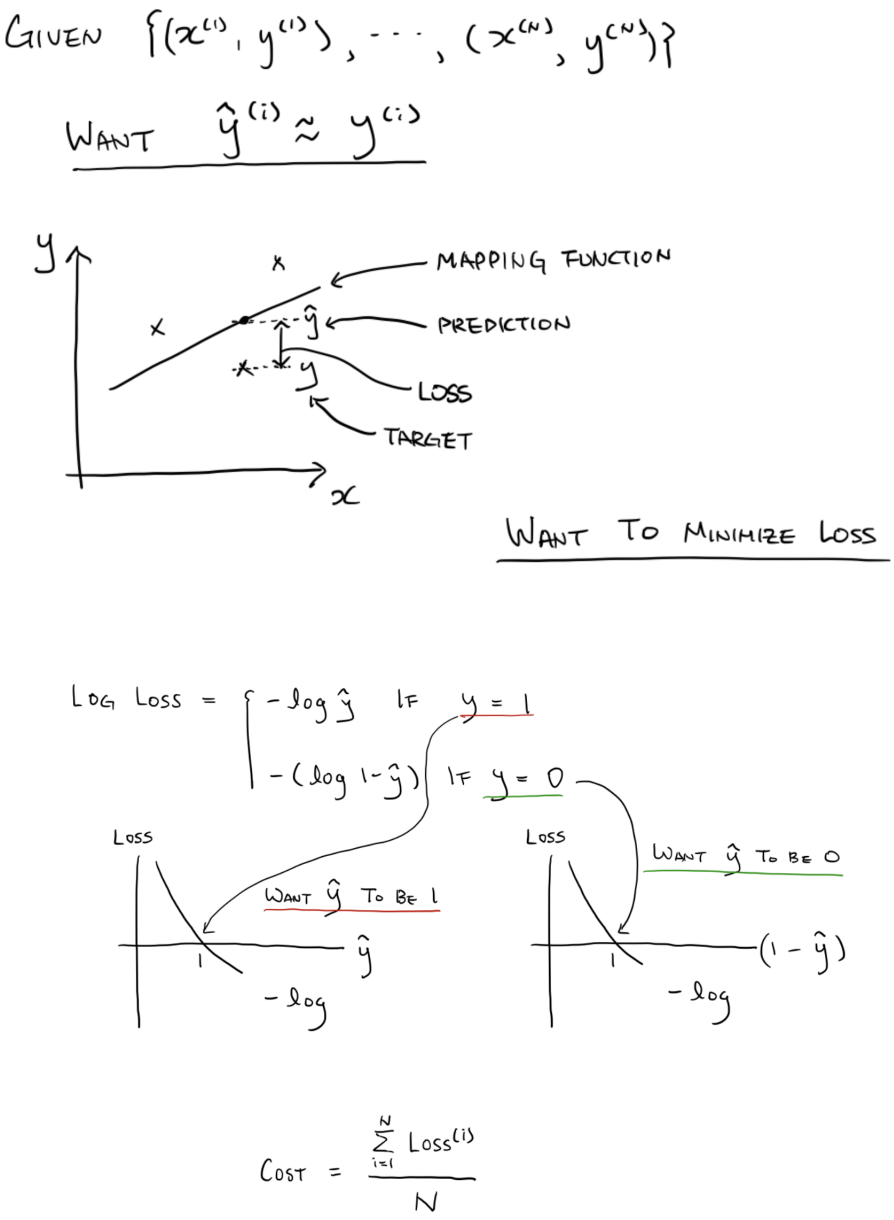
To use a 64x64x3 image as an input to our neuron, we need to flatten the image into a (64x64x3)x1 vector and to make $\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$ output a single value $\mathbf{z}$, we need $\mathbf{W}$ to be a (64x64x3)x1 vector: (dimension of input)x(dimension of output), and $\mathbf{b}$ to be a single value. With $\mathbf{N}$ number of images, we can make a matrix $\mathbf{X}$ of shape (64x64x3)x$\mathbf{N}$. $\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$ outputs $\mathbf{Z}$ of shape 1x$\mathbf{N}$ containing $\mathbf{z}$'s for every single sample, and by passing $\mathbf{Z}$ through a sigmoid function we get final $\hat{\mathbf{y}}$ of shape 1x$\mathbf{N}$ that contains predictions for every single sample. We do not have to explicitly create a $\mathbf{b}$ of 1x$\mathbf{N}$ with the same value copied $\mathbf{N}$ times, thanks to Python **broadcasting**.



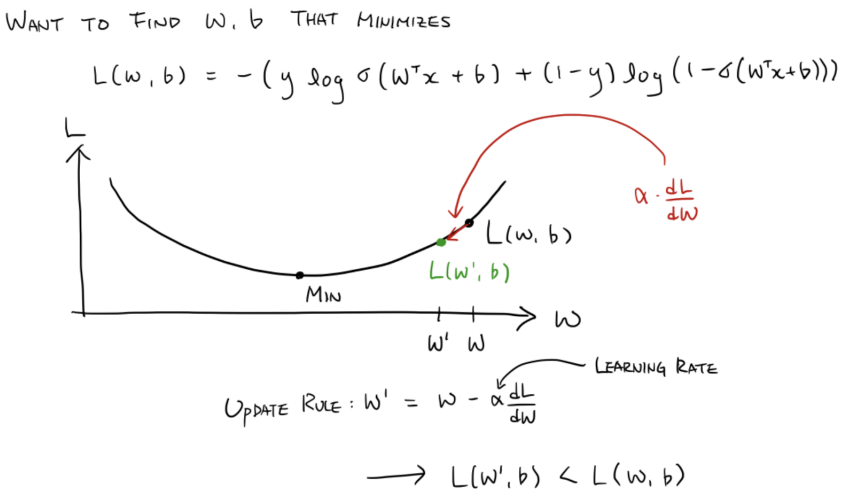
$\mathbf{y}$ is our given **target** or **label**. During **supervised learning**, our goal is to try to learn the exact mapping from $\mathbf{x}$ to $\mathbf{y}$. In other words, we want to figure out a mapping function that would give us **prediction** $\hat{\mathbf{y}}$ as close as possible to **target** $\mathbf{y}$. We use **loss function** to measure how close $\hat{\mathbf{y}}$ is to $\mathbf{y}$. We could simply use the distance or use something better like **logarithmic/cross-entropy loss**:

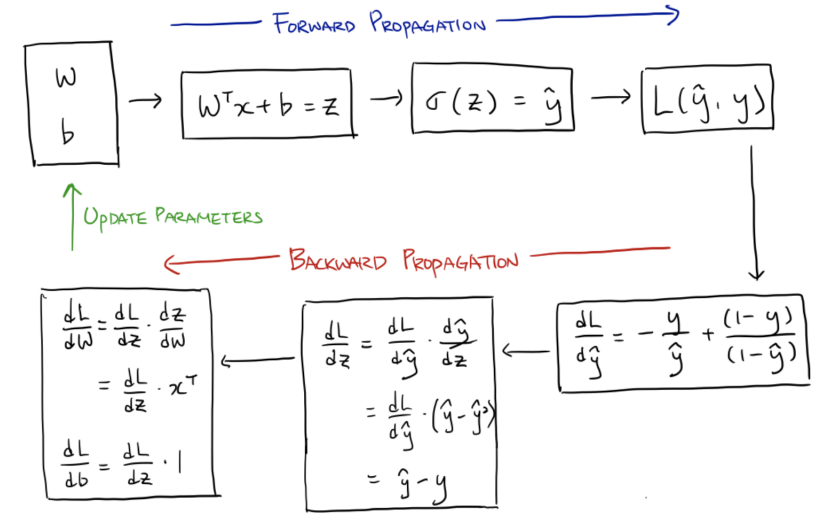
$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[\mathbf{y}^{(i)}\log\hat{\mathbf{y}} + \left(1-\mathbf{y}^{(i)}\right)\log\left(1-\hat{\mathbf{y}}\right)\right]$$

Now, we can simplify our goal to minimizing the loss function. When we have multiple $\hat{\mathbf{y}}$'s and $\mathbf{y}$'s from multiple $\mathbf{x}$'s, we take the average of **loss** to get **cost**.

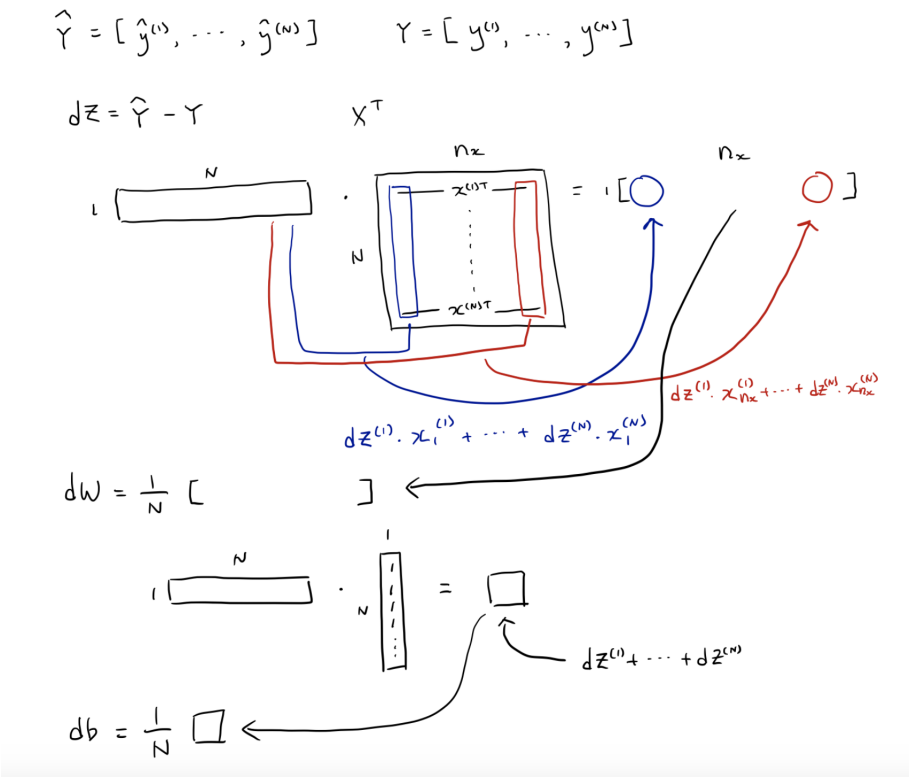


We use **gradient descent** to achieve our goal of finding the **W** and **b** that minimizes the loss function given by **L(W,b)** as given below.
To do this, we compute **L**, then derive **dL/dW** and **dL/db**, to use them for updating **W** and **b** so that they would give a better **L** when we compute it with the same **x** next time. The derivatives provide the directions, while **learning rate** denoted by α determines the magnitude of the changes we make. The **update rule** has been given below. Using the **chain rule**, we can easily derive **dL/dW** and **dL/db**. In deep learning, we call the chain of computations up to **L**, the **forward propagation**, and the following chain of computations to **dL/dW** and **dL/db**, the **backward propagation**. **Computation graphs** helps us visualize the chains. Finally, **logistic regression** is simply repeating the process of our forward propagation, backward propagation, and update parameters, to find **W** and **b** that maps **x** to $\hat{\mathbf{y}}$ with minimum **logarithmic loss** compared to **y**.

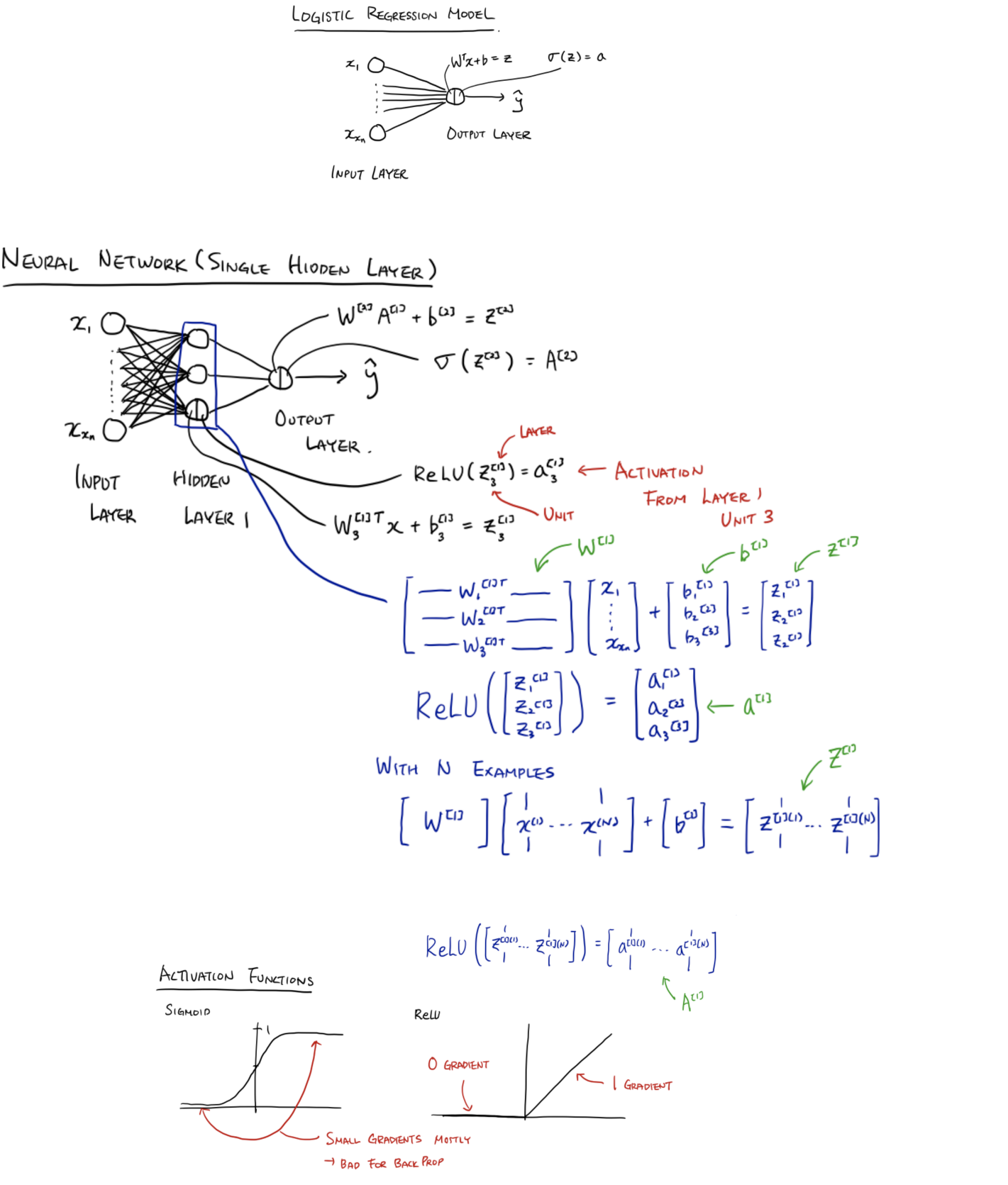




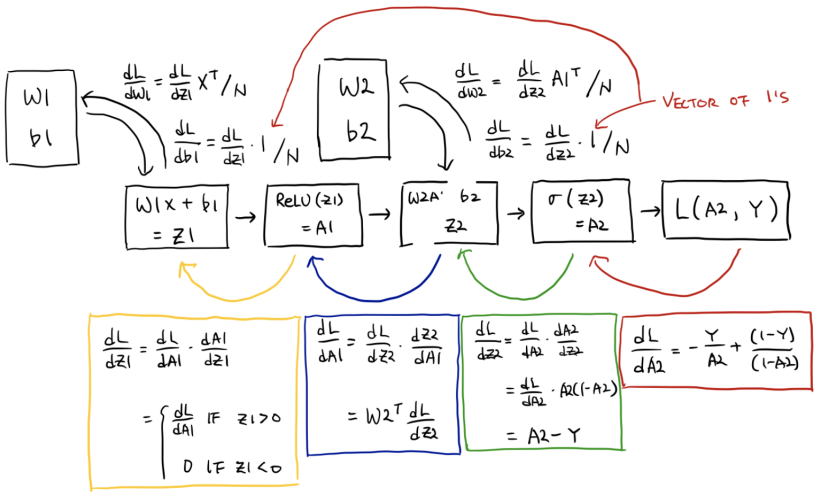
What happens to **dL/dW**(**dW** from here on) and **dL/db**(**db** from here on) when we have multiple x's? As mentioned before with **N** number of **x**'s, we create a **nx** by **N** matrix **X** that has **x**'s as columns. Similarly, its target **Y** is a 1x**N** row vector with **N** number of **y**'s stacked horizontally. Forward propagation of **X** computes $\hat{\mathbf{Y}}$ with the same shape as **Y** and their **cost** by taking an average over multiple losses. Likewise, we will have to take an average over multiple **dW**'s and **db**'s. With 1x**N** $\hat{\mathbf{Y}}$ and **Y**, **dZ** will be 1x**N** as well, carrying **dz**'s for N samples. Taking the dot product between 1x**N** **dZ** and **N**x**nx** **X.T** results in 1x**nx** row vector just like **W**. Even with just one sample, when dz is 1x1 and x.T is 1x**nx**, the dot product between the two outputs 1x**nx**. Then what's the difference? With **N** number of samples, each element in the row vector becomes a sum of **dW**'s from **N** samples, due to dot product. To take an average, all we have to do is dividing the 1x**nx** row vector by **N**. Unfortunately and obviously, there is no Python broadcasting for dot products. To mimic the operation with **dW** for **db**, I prefer to take a dot product between 1x**N** **dZ** and **N**x1 vector of ones, then divide the resulting number with **N** to take an average. But in the course, we take the sum of **dZ** over **N** samples and divide it by **N**.



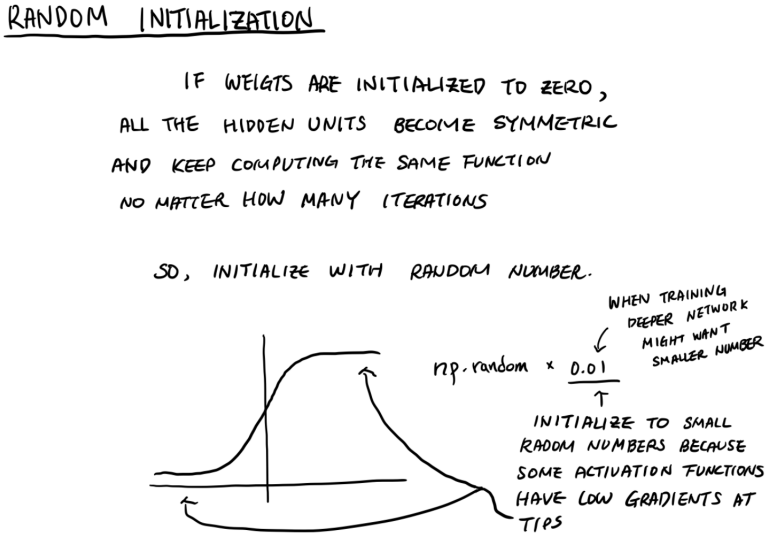
The beauty of the **forward** and **backward propagation** is that it works seamlessly even when we stack bunch of **neurons** to make a **neural network**. For **input layer** is just a placeholder for inputs and does not compute anything, we do not count it as **neurons**. So, in our **logistic regression model**, we only had a single **neuron** in **output layer**. To turn it into a **neural network binary classifier**, we add hidden layers in between the **input** and **output layers**. Each **hidden layer** have multiple number of **neurons**, and each **neuron**'s output is called an **activation**. **Activations** from a **hidden layer** become **inputs** for the next layer. The **activation** of the **neuron** in **output layer** is $\hat{\mathbf{y}}$. And it's **activation function** that wraps $W^T\mathbf{x} + \mathbf{b}$ is a **sigmoid function**. Likewise, every **neuron in hidden layers (hidden units)** computes **linear** operation $W^T\mathbf{x} + \mathbf{b}$ then pass it to a **nonlinear activation function** to output an **activation**. **ReLU function** is a better choice of **activation function** for **hidden units** because unlike in the **output layer**, we do not need to squeeze **activations** from **hidden units** to be in between **0** and **1**. Additionally, **ReLU(z)** = max(0, **z**); therefore, it does not suffer from having very small gradients at the ends, helping the **neural network** to learn much faster. **Activation functions** have to be **nonlinear** because composition of **linear** functions only result in another linear function, which is are not complex enough to solve complex problems. Repeatedly computing activations individually takes a lot of for-loops that makes things run very slow. So, we use **vectorization** to summarize for-loops into one matrix dot product. This is not a new concept for us because we already used it to make matrix **X** out of multiple **x**'s by just stacking them as columns. Likewise, we will stack multiple **W**'s for each units in the same layer as columns and transpose it to get the vectorized $W^{[1]}$. The shape of $W^{[1]}$ is **number of units in current layer** x **number of units from previous layer**. With vectorized $b^{[1]}$ and **X**, $W^{[1]}$X + $b^{[1]}$ will give **number of units** x **N** shaped $Z^{[1]}$. And, passing it through activation function will give $A^{[1]}$ with the same shape. $A^{[1]}$ is passed onto the next layer (output layer), which computes $W^{[2]}A^{[1]} + b^{[2]} = Z^{[2]} and A^{[2]} = \sigma(Z^{[2]})$.



For **backward propagation**, the derivative of $\mathbf{a = sigmoid(z)}$ is $\mathbf{a(1\text{-}a)}$, and the derivative of $\mathbf{a = ReLU(z)}$ is $\mathbf{0}$ **if** $\mathbf{z < 0}$ and $\mathbf{1}$ **if** $\mathbf{z \geq 0}$. Let's take a look at the **computation graph** of a **neural network** with a **single hidden layer**.

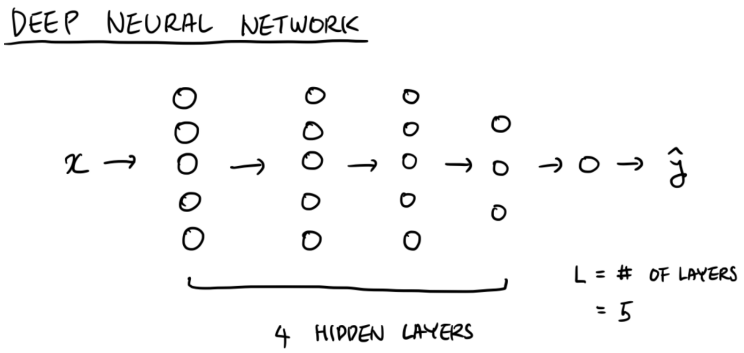


As mentioned, **gradient descent** is an iterative process that steps toward the point with minimum loss. To find **W**'s and **b**'s that give the minimum loss, we need to set the starting point. If we initialize the parameters (**W**'s and **b**'s) with zeros, all the **hidden units** will compute the same **activations** and during **backward propagation**, get the same **gradients**. Basically, all the **hidden units** will become symmetric and keep computing the same function no matter how may iterations. So, we initialize the parameters with random number. When using **sigmoid** as **activation functions**, it is important to keep the numbers small because gradients are nearly 0 at the ends.



By putting everything together, we can build a **binary classifier**. First, we **load data** and **initialize parameters**, then repeat **forward-backward propagation** and **parameter updates**. We have seen in detail what goes inside each neurons from a **logistic regression** model and a **single hidden layer neural network model**. As mentioned in the beginning, we can learn any x to y mapping function given enough number of neurons. Therefore, **deep networks** with multiple **hidden layers** can learn functions that the shallow ones cannot. The **hidden layer 2** will take $A^{[1]}$ in and output $A^{[2]}$, and pass it onto the **hidden layer 3**. Repeating the process

until the output layer is reached.The deeper the layer is the higher level features it learns. If we have a face recognizer, the first layer will figure out how to detect edges, the next layer will learn how to group those edges to detect eyes, ears, nose, and etc. And the later layer will learn how to group those informations to recognize different faces. Let's build a **deep neural network**.



Learning rate, number of epochs (number of iterations), number of hidden units, number of hidden layers, choice of activation functions, etc. are **hyper-parameters** that control the character of **parameters** W and b. We can build some insights but the only way to figure out good **hyper-parameters** is by experimenting.

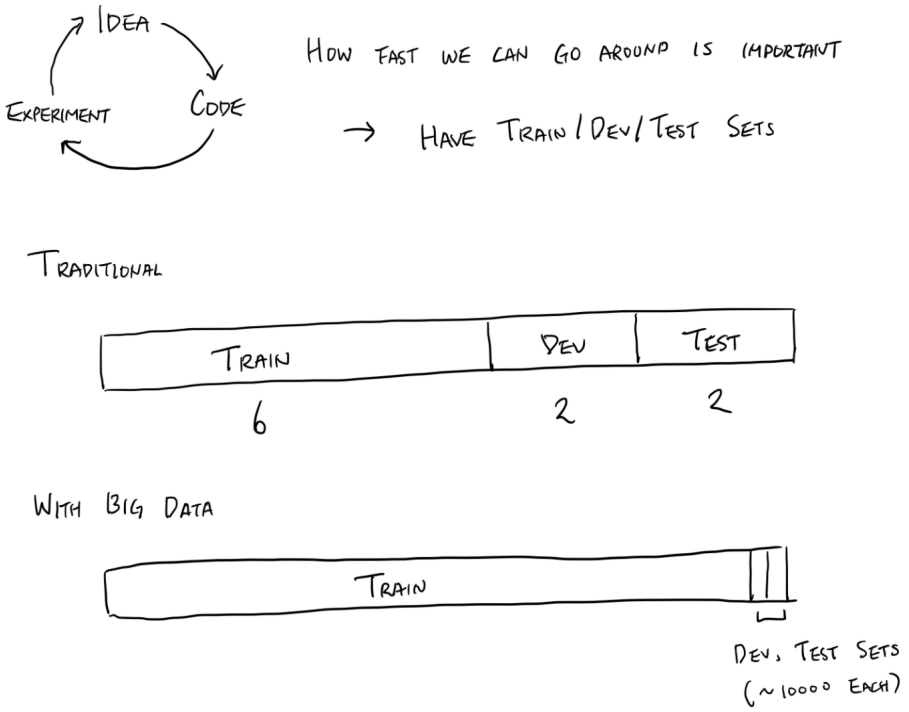
Optimization of Deep Neural Networks

Swapnoneel Kayal

8 June - 14 June 2021

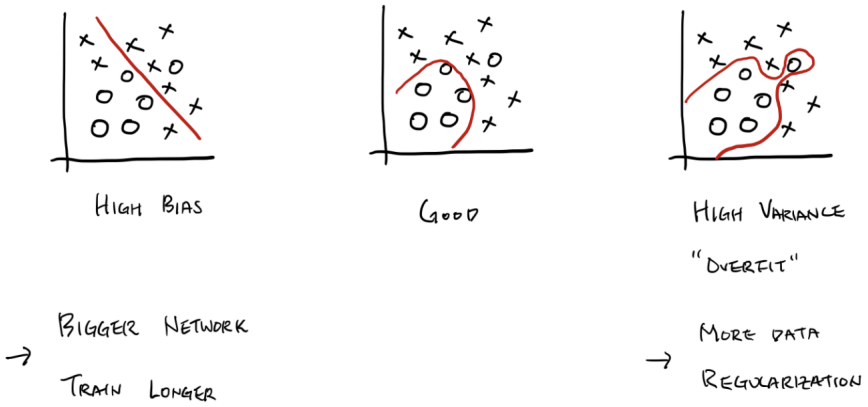
1 Train/Dev/Test Sets

Deep learning is a very iterative process looking for the right set of **hyper-parameters**. It is important to repeat the process of having an idea, coding, and experimenting. To do so, we need to set up dataset properly into **train**, **dev**, and **test** sets. **Train set** is used to train the model. **dev** and **test sets** are data that the model had never seen before and are used to analyze the model. Traditionally, we had the golden ratio of 6:2:2 = Train:Dev:Test. However with **big data**, that distribution will allocate too much data for **dev** and **test sets**. Considering that **deep learning** is very data hungry, it is wiser to use most of the data for **training** and just have around 10,000 samples each for **dev** and **test sets**. Also, it is okay to just have the **dev set**. If we have both **dev** and **test sets**, it is extremely important that they come from the same distribution. For example, one cannot be high-def images while the other is low-def images.



2 Bias/Variance

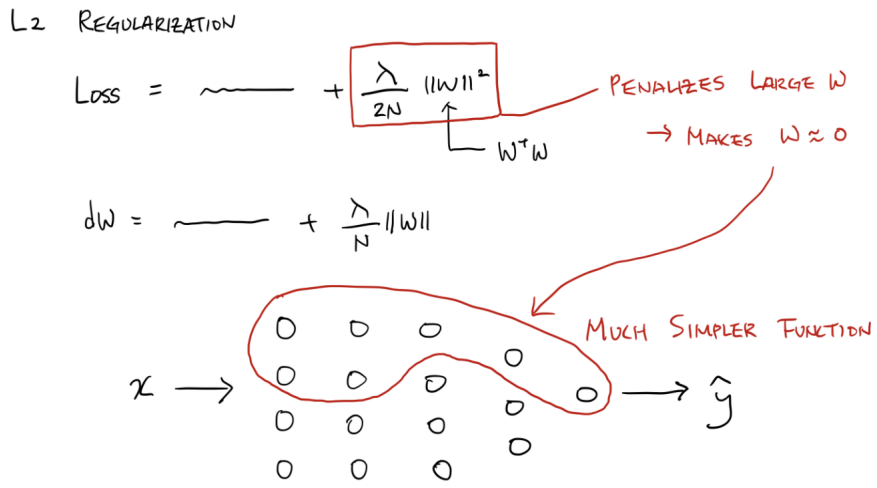
The reason we need separate **train** and **dev** set is to analyze the model's **bias** and **variance**. Model with **high bias** will be less flexible and fail to have enough complexity to classify properly. On the other hand, model with **high variance** will be too flexible and **overfit** to the **training data**, being specialized with the **train set** and not good with any data that it has not seen before. Therefore, when **dev error** is high while **train error** is low, we fix the **variance** problem and if the **train error** is high and **dev error** is about same, we fix the **bias** problem. If we have a higher **dev error** while **train error** is already high, we have both **bias** and **variance** problems to solve. Eventually, we want both **train error** and **dev error** to be low with very low gap in between them — low **bias** and low **variance**. When we have **high bias**, we need to make the network more flexible/complex or simply make it learn more. So for **bias** problem, we can build a bigger network, which almost never hurts, or train for longer. With **high variance**, we want to prevent **overfitting** by introducing more **training data** or **regularization** to the model.



TRN ERROR	1%	15%	15%	0.5%
DEV ERROR	15%	15%	30%	1%
	LOW BIAS HIGH VAR	HIGH B LOW V	HIGH B HIGH V	LOW B LOW V

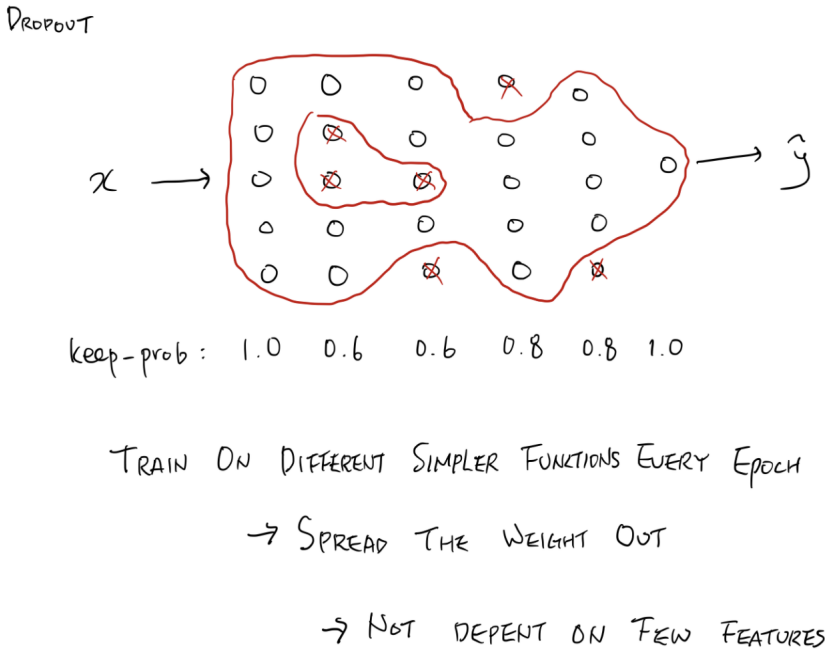
3 Regularization

There are several ways to do **regularization**. One of them, **weight decay** adds the sum of magnitudes of **weights** and **biases** to the **loss function**, so that it penalizes large **weights** (both negative and positive). **Weights** are more likely to be closer to 0 because our goal is to minimize the **loss function**. This allows us to learn much simpler mapping function. If you square the sum, it becomes the **L2 regularization** and if you don't, it becomes **L1 regularization**. $\lambda/2N$ is a term multiplied before the sum and λ is a new **hyper-parameter**. It is okay to omit **biases** because they are negligible.



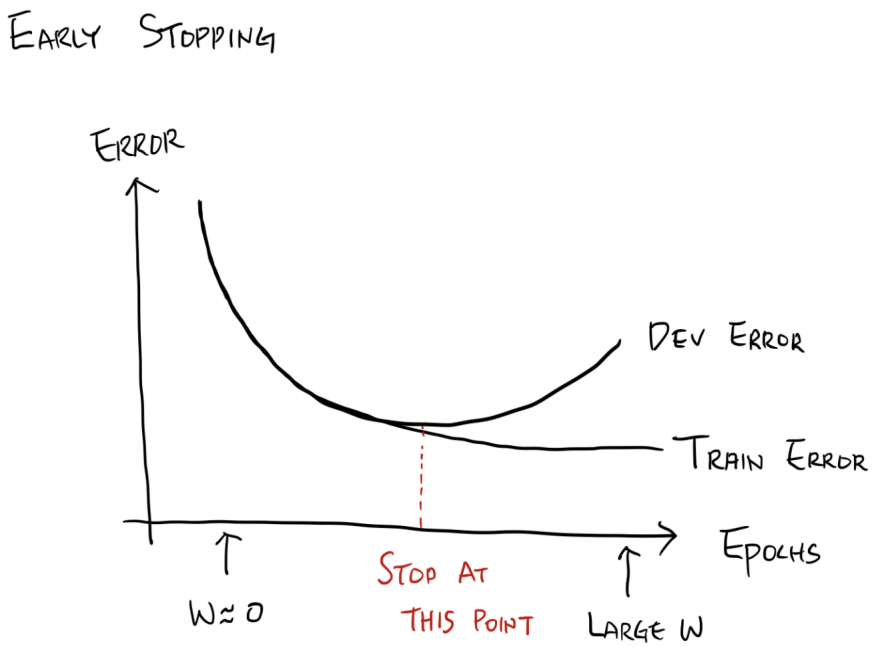
4 Dropout Regularization

Another **regularization** method is **dropout**. With certain chance, we shut some **neurons** off forcing the network to learn much simpler function. It is important to **dropout** different **neurons** every iteration. We might want to use different **dropout** chances for each layer because **earlier layers** have higher chance of overfitting compared to the **later layers**. So we would normally **dropout** more **neurons** in **earlier layers** by using lower **keep_prob**, which tells us how much **neurons** to keep. We usually do not use **dropout** for the **input layer** and definitely not during **test time**. The intuition behind **dropout** is in trying to learn simpler mapping function but also to spread out the **weights** so that we do not specialize in few features. One downside of **dropout** is that it kind of makes the **loss function** less well defined.



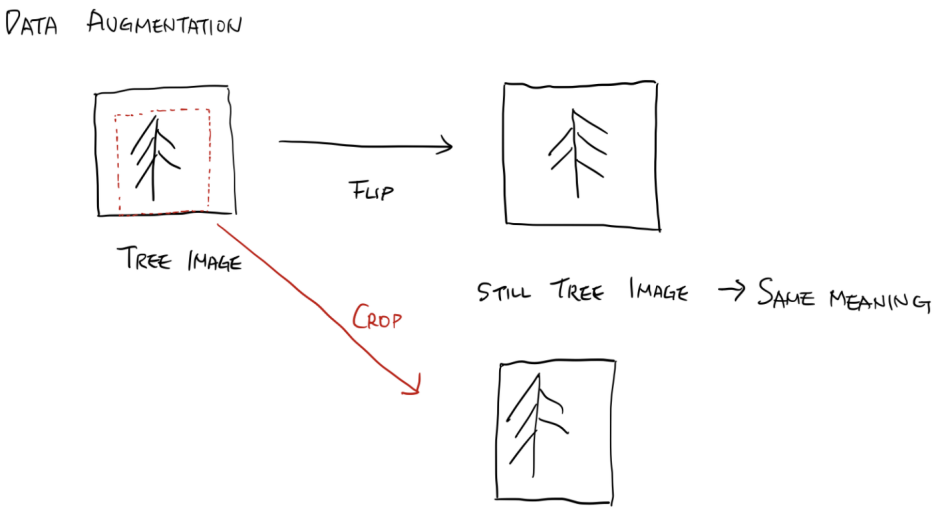
5 Early Stopping

Lastly, **early stopping** is something worth trying. If **weights** are initialized to small random numbers, they will start to grow too big and **overfit** as we train for longer time. So, we monitor both **train** and **dev error** during **train time** and end **training** when things seem to be just right. However, we mentioned that training for longer period affects **bias**, which means that shortening training time will affect bias as well. Therefore, **early stopping** is not an ideal option for **orthogonalization** which means to deal with **bias** and **variance** separately.



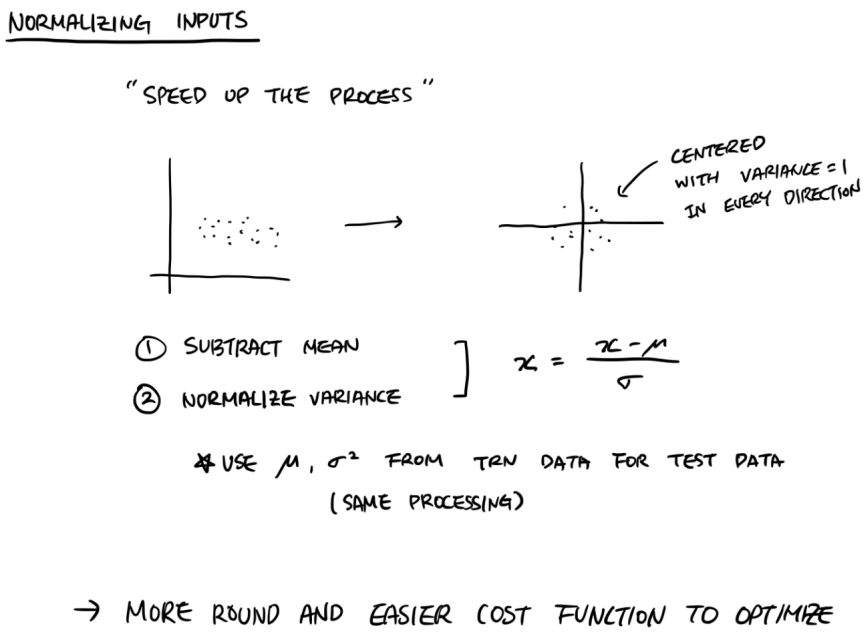
6 Data Augmentation

Coming up with more data could be very difficult and expensive. **Data augmentation** is a technique where we create new data by **flipping**, **random cropping**, and etc. to modify the original data just slightly enough to not change the meaning of it.



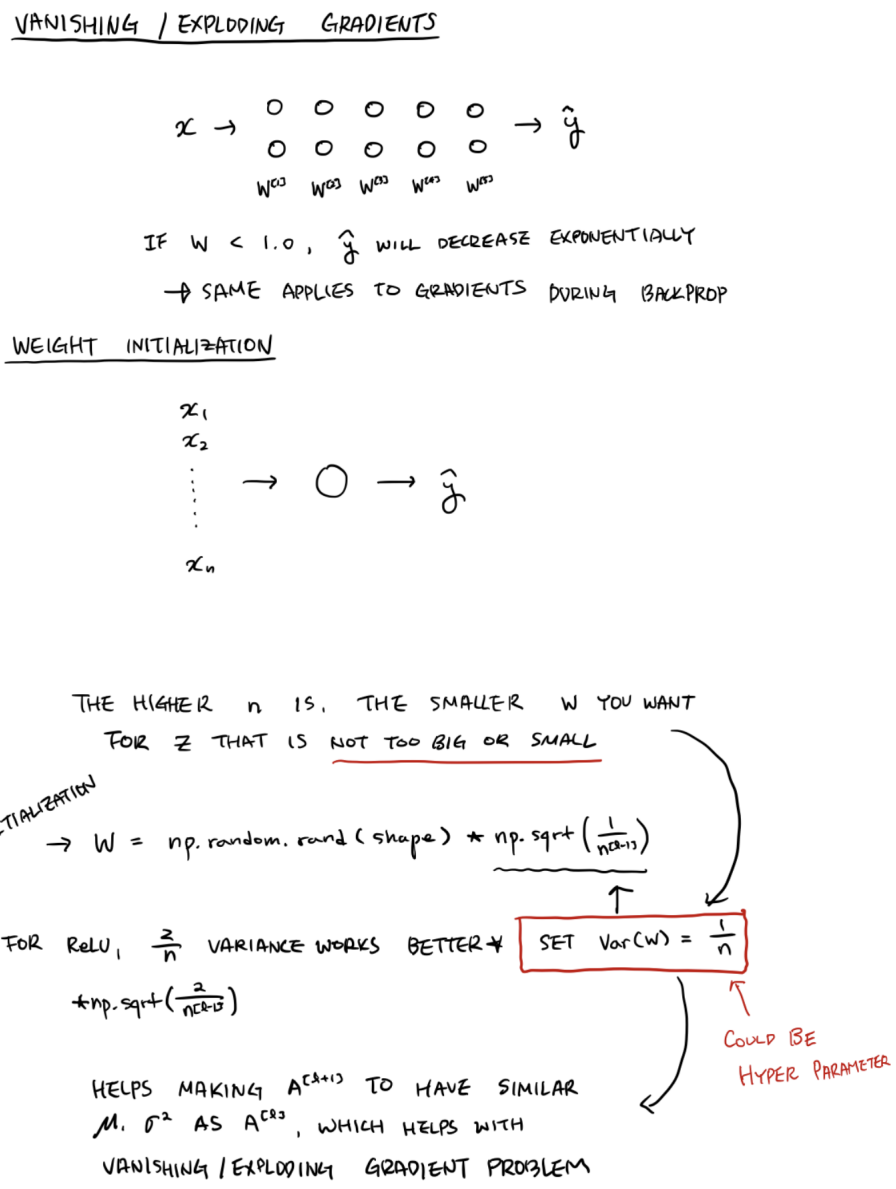
7 Normalizing Inputs

Normalizing inputs is always a good idea because it speeds up the process of learning. **Normalization** centers the inputs to (0, 0) with the **variance** of **1** in every direction. To achieve this, we first subtract the **mean (μ)** of **X** from **X** then divide it by the square root of its **variance** (σ^2). This allows to have a more round and easier **cost function** to optimize with. **Normalization** becomes more important when input features are in various scales. For example, with x_1 ranging from **1.0** to **-1.0** and x_2 ranging from **100** to **-100**, we really want to **normalize**.



8 Vanishing/Exploding Gradients And Weight Initialization

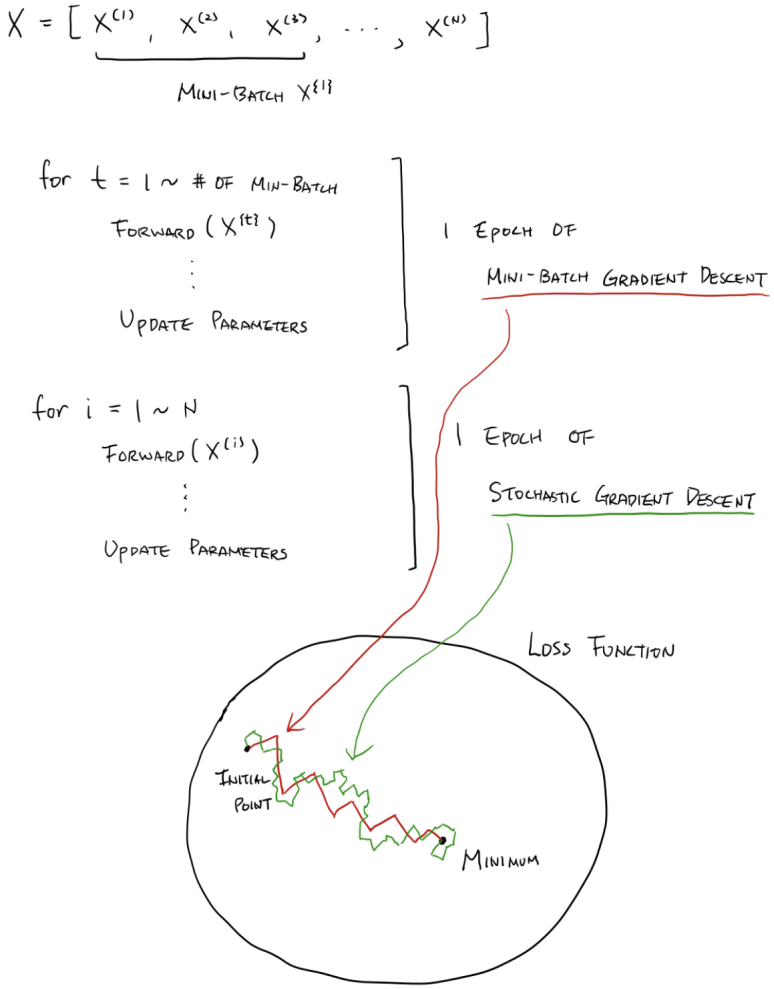
Vanishing gradient has been a problem for **deeper networks**. If all the **weights** are less than **0**, all the **gradients** flowing back during **back propagation** will be getting smaller and smaller as they pass through **layers**. One of the ways to alleviate this problem is initializing weights properly. As if we multiplied **0.01** to initialize **weights** to be small random numbers for **sigmoid functions**, we set the **variance** of **weights** to be **2 / (number of inputs)** for **ReLU activation**. This method is called the **He initialization**.



9 Optimization

There are several ways to **optimize** the model. Methods that we have already seen before are **stochastic gradient descent** and **mini-batch gradient descent**. **Stochastic gradient**

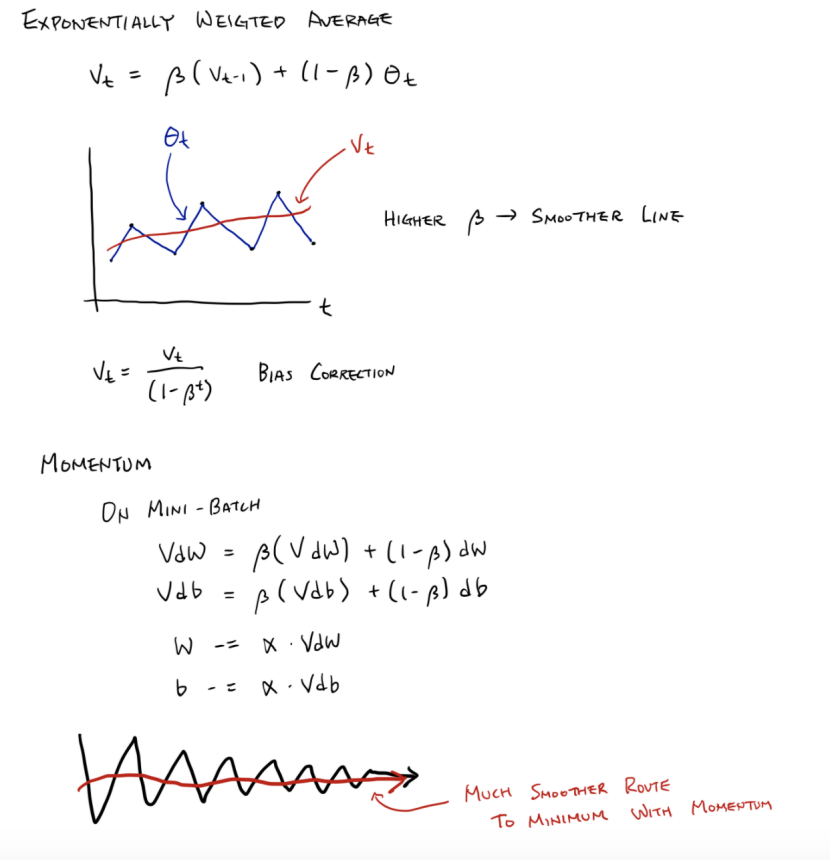
descent optimizes **parameters** by using **one sample** at a time and **mini-batch gradient descent** uses an assortment of **multiple samples** called a **mini-batch** at a time. We can consider **stochastic gradient descent** as **mini-batch gradient descent** with the **mini-batch size** of **1**. We call a cycle through training data, an **epoch**. Because of larger **mini-batch size**, **mini-batch gradient descent** has less noise than **stochastic gradient descent**. In other words, **mini-batch gradient descent** takes more direct route towards the **minimum** than **stochastic gradient descent**. **Mini-batch size** is another **hyper-parameter** that we have to figure out empirically. However, it is known that numbers like **64, 128, 256, and 512** that are computer memory sized work well for **mini-batch size**.



10 Exponentially Weighted Averages And Gradient Descent With Momentum

By using **exponentially weighted averages** of **gradients**, we could take even more direct route. **Exponentially weighted average** is computed by mixing **exponentially weighted average** till now (v_{t-1}) and **current value** (θ_t): $v_t = \beta v_{t-1} + (1\text{-}\beta)\theta_t$. β is another **hyper-parameter**, and 0.9 usually works well. You can notice from the formula that, v_t with low t will not be close to the original θ_t. We could use **bias correction** to fix such problem. After computing normal v_t, we divide it by $(1-\beta^t)$. Overall, **exponentially weighted average** will give smoother curve than the original values. Having oscillations during **gradient descent** made it hard for us use

large **learning rates**. To solve this problem, we can use **gradient descent with momentum** that uses **exponentially weighted averages** of **dW's** and **db's** (V_{dW} and V_{db}) to update the **parameters**. Things almost always work better with **momentum**.



11 RMS Prop and Adam Optimization

RMS Prop is an **optimization** method with the similar concept except that it takes **exponentially weighted average** on **gradient squared** (S_{dW} and S_{db}) instead of normal **gradient**. To update **parameters**, it uses $\mathrm{dW}/\sqrt{(\mathrm{S}_{dW})}$ and $\mathrm{db}/\sqrt{(\mathrm{S}_{db})}$. Another **optimization method** that uses **exponentially weighted average** is **Adam**. It computes **exponentially weighted average** for both **gradient** and **gradient squared**. So, we need two extra hyper-parameters β_1 and β_2. **0.9** works well for β_1, and **0.999** works well for β_2. With **Adam optimization**, we update **parameters** with $V_{dW}/\sqrt{(\mathrm{S}_{dW})}$ and $V_{db}/\sqrt{(\mathrm{S}_{db})}$.

RMS PROP

$$S_{dW} = \beta_2(S_{dW}) + (1-\beta_2)\, dW^2$$

$$S_{db} = \beta_2(S_{db}) + (1-\beta_2)\, db^2$$

$$W := \alpha \cdot \frac{dW}{\sqrt{S_{dW}}}$$

$$b := \alpha \cdot \frac{db}{\sqrt{S_{db}}}$$

ADAM

COMPUTE V_{dW}, S_{dW}, V_{db}, V_{db}

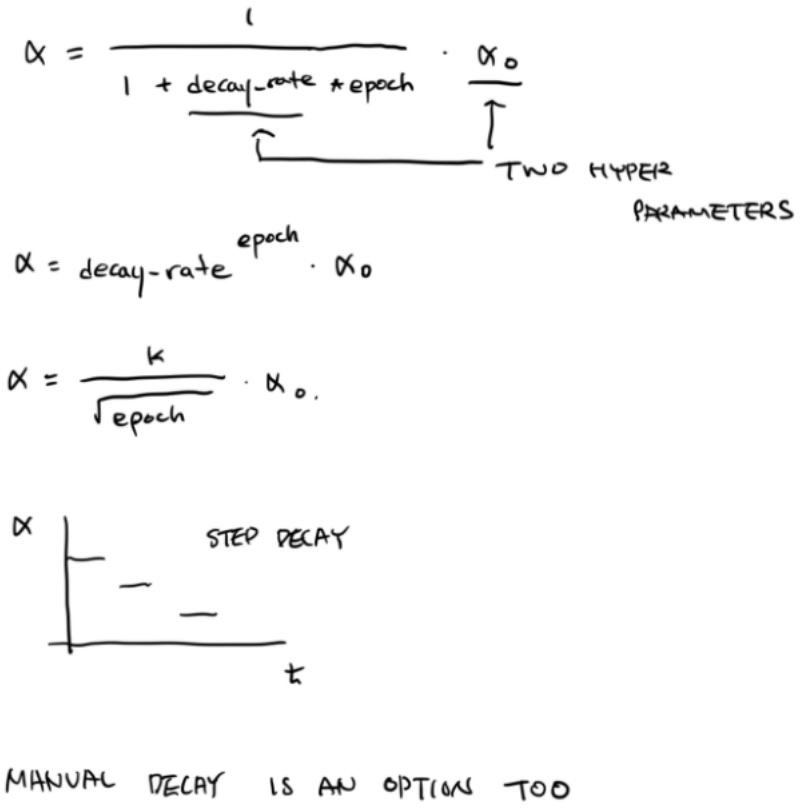
BIAS CORRECTION

$$W := \alpha \frac{V_{dW}}{\sqrt{S_{dW}}}$$

$$b := \alpha \frac{V_{db}}{\sqrt{S_{db}}}$$

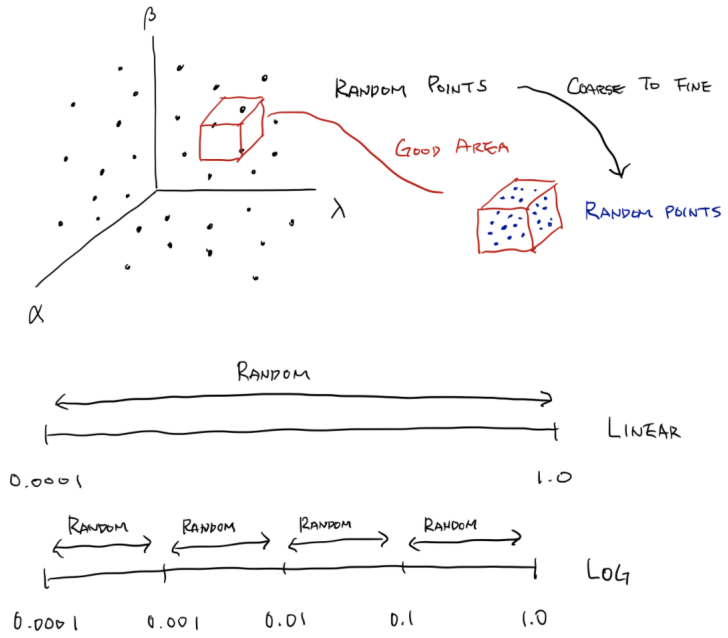
12 Learning Rate Decay

Last thing we have to know about **optimization** is **learning rate decay**. As its name suggests, we reduce **learning rate** as we get closer to the **minimum** in order to make converging to it easier. We can use several different functions to decay the original **learning rate**.



13 Tuning Process

We have discovered many new **hyper-parameters**. How do we tune **hyper-parameters**? First, we should try to find good **learning rate** because it is the most important **hyper-parameters**. Then, we should work on finding **the number of hidden units, mini-batch size**, and β's. How do we find good value for **hyper-parameters**? We should choose random points within certain ranges of **hyper-parameters**. As we try them out, we should find the good area with smaller ranges to sample the random points more densely and keep searching from coarse to fine. There are two search scales we can use: **linear scale** and **log scale**. **linear scale** samples things within the range evenly, while log scale samples things evenly on the orders of magnitude. For example, **linear scale** will sample 0.0001 to 1.0 evenly/linearly, but **log scale** will sample evenly within the orders of magnitude: from 0.0001 to 0.001, 0.001 to 0.01, and 0.01 to 0.1, and 0.1 to 1.0. In practice, even after finding the good **hyper-parameters** for our model, we should tune our **hyper-parameters** once in a while.



14 Batch Norm

Batch normalization is like having **input normalization** for every **layer**. So it normalizes every activation to **train** every **parameters** faster. But in practice, we actually normalize **Z** which has the same effect as normalizing **A**. We first normalize **Z** into Z_{norm}. Then, we compute the $Z_{new} = \gamma \cdot Z_{norm} + \beta$. γ and β are learnable **parameters** that basically allows **Z** to have other **mean** and **variance** that are more suitable for **activation functions**. Because Z_{new} is centered to 0 with **variance** 1, it is very likely for **Z** to hit the sweet spot of **activation functions**. **Batch norm** allows the model to be more robust to **covariate shift** when the distribution of input change. Moreover **Batch norm** has slight **regularization** effect because it kind of can cancel out large W's, adding noise. Its **regularization** effect gets weaker as **batch size** gets larger because there is less noise as **batch size** gets larger. If **batch norm** is not beneficial, the model can always learn γ to be the **variance of Z** and β to be the **mean of Z** to cancel out **batch norm** and make Z_{new} equal to **Z**. During **back propagation**, we need to compute dγ and dβ to update γ and β. Each layer has its own γ and β, and their shapes are **(number of units in the layer, 1)** because γ and β has to multiply and add to every single z's from the **layer**. Obviously, dγ and dβ will have the same size. During **test time**, we use **exponentially weighted average** of **means** and **variances** from **training** across different **mini-batches** and across different **layers**. This is as if we are learning general **mean** and **variance** during training.

15 Softmax Regression

We used **sigmoid activation function** in **output layer** for our **binary classifier**. What if we have more than two **classes**? We use **Softmax function** which is really good for picking the maximum probability out of many. As its name suggests, it is a soft version of **max function**. Instead of selecting the maximum, it distributes probability in between 0 and 1 so that the maximum gets the largest portion and the minimum gets the least portion, which makes **Softmax function** suitable for **multi-class classification**. Those probabilities add up to 1. For **loss**

function, we use **cross-entropy loss**. For **backprop, dL/dZ** of the output layer beautifully works out to be **A - Y**. For implementing **Softmax function**, there is a naive way that directly follows the mathematical function. Because exponentials can easily explode beyond float64 capacity, the naive version is numerically unstable. Therefore, we generally implement the stable version of **Softmax function** that basically normalizes the values before putting them through exponentials.

$$a = \text{SOFTMAX}(z) \longrightarrow a_i = \frac{e^{z_i}}{\Sigma e^{z}}$$

STABLE SOFTMAX

$$a_i = \frac{C e^{z_i}}{C \Sigma e^{z}} = \frac{e^{z_i + \log(C)}}{\Sigma e^{z + \log(C)}}$$

WE SET $\log(C) = -\max(z)$

$$\text{CROSS ENTROPY LOSS} = -\Sigma\, y \log(a)$$

Deep Learning and Neural Networks

Swapnoneel Kayal
Roll Number: 200100154
Mentor: Anuj Srivastava

June 2021

1 Materials to be used

- Deep learning Specialization on Coursera by Andrew Ng
- TensorFlow tutorials at :
  `https://www.youtube.com/watch?v=5Ym-dOS9ssA&list=PLhhyoLH6IjfxVOdVC1P1L5z5azs0XjMsb&index=1`
- Numpy tutorials at:
  `https://www.youtube.com/watch?v=8JfDAm9y_7s`
- Book : "Neural Networks and Deep Learning" by Michael Nielson
- Book : "Deep Learning" by Ian Goodfellow and Yoshua Bengio and Aaron Courville published by MIT Press
  `http://www.deeplearningbook.org`
- For computer vision:- Convolutional Neural Networks Course for Visual Recognition offered by Stanford.

2 Modified Plan Of Action

- **Week 1: 18 May - 24 May**
  Conventional Machine Learning tools I : Linear and Logistic Regression, Decision trees
- **Week 2: 25 May - 31 May**
  Conventional Machine Learning tools II : SVM and PCA
- **Week 3: 1 June - 7 June**
  Neural Networks and Deep Neural Networks
- **Week 4: 8 June - 14 June**
  Optimization of Deep Neural Networks
- **Week 5: 15 June - 21 June**
  Convolutional Neural Networks
- **Week 6: 22 June - 28 June**
  Recurrent Neural Networks and Building Sequence Models
- **Week 7: 29 June - 5 July**
  Implementing Convolutional Neural Network : Traffic Sign Classification
- **Week 8: 6 July - 12 July**
  Implementing Recurrent Neural Network : Language Translator

3 Progress Till Now

- In the first course of the Deep Learning Specialization, I was introduced to the foundational concept of neural networks and deep learning. By the end, I become familiar with the neural networks and was able to build, train, and apply fully connected deep neural networks. Not only that, I was also able to implement efficient vectorized neural networks and identify the key parameters in a neural network's architecture.
- In the second course of the Deep Learning Specialization, I was systematically introduced to the various processes that drive the performance of a deep neural network and are key in generating good results. By the end, I learnt the best practices to train and develop test sets and analyzed bias/variance for building deep learning applications and was able to use standard neural network techniques such as initialization, L2 and dropout regularization, hyperparameter tuning, batch normalization, and gradient checking. I also implemented and applied a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and checked for convergence in each case.
- Initially I went through the Numpy tutorials at : `https://www.youtube.com/watch?v=8JfDAm9y_7s` and then also completed the TensorFlow tutorials at : `https://www.youtube.com/watch?v=5Ym-dOS9ssA&list=PLhhyoLH6IjfxVOdVC1P1L5z5azs0XjMsb&index=1`

4 End Goal

Currently, I have the basic knowledge in the domain of Deep Learning and Neural Networks but still I feel till now I have been only scratching the surface. From this point onwards, I would be diving deep into the various different types of Neural Networks like Convolutional Neural Networks, Recurrent Neural Networks and would be implementing some of them as mini-projects just to get the feel of how powerful these can actually become.

5 Gratitude For My Mentor

I decided to keep a separate section where I would like to formally thank my mentor, Anuj Srivastava, who kept his patience, guided me in the right direction from the starting and helped me whenever I required him. He made sure I am confident with each and every concept and that I utilized the time well which I feel was only possible because of his expertise in the field of Neural Networks and Deep Learning. I also thank the MnP club for giving me such an opportunity.