NEURAL NETWORKS AND DEEP LEARNING (JULY, 2021)

ATUL KUMAR VERMA
19B090004
MENTOR:PRAKASH PRASAD

FINAL REPORT

INTRODUCTION

We come into contact with products based on machine learning everyday in our life without even realizing it. From the famous google page rank algorithm to the applications in financial trading, machine learning is everywhere. Even the algorithm that email service providers use for identifying spam is based on machine learning. Machine learning is almost everywhere. So let's jump to the question of what is machine learning and what is a machine learning algorithm.
According to Arthur Samuel, a pioneer in the field of artificial intelligence and computer
gaming,
“Machine learning is a field of Computer Science that gives computers the ability to learn without being explicitly programmed”
Now, let’s get to how a Modern Computer Scientist thinks about a Machine Learning Algorithm:
“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."
Of course we are no computer scientists, so let’s try to put it in a more layman terms. We
can say that a machine learning algorithm is a computer program that allows a machine to do get better at doing a job by, if I may say, repeated practice without actually explicitly being programmed to that job. Of course here the practice is the training set we provide the algorithm with (The training set is a collection of relevant data related to the job/task we wish to accomplish). This may sound something like teaching a machine and this is what excites many people. The Big AI dream, though really far away from being realized, can one-day be achieved through Machine Learning techniques. Though we will not talk about AI here, I shall try to touch upon neural networks here, which were our first attempts at Trying to emulate the human brain.

MACHINE LEARNING LIBRARIES

A. NumPy

NumPy is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions. It is very useful for fundamental scientific computations in Machine Learning. It is particularly useful for linear algebra, Fourier transform, and random number capabilities. High-end libraries like TensorFlow uses NumPy internally for manipulation of Tensors.

```
# Python program using NumPy
# for some basic mathematical
# operations

import numpy as np

# Creating two arrays of rank 2
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])

# Creating two arrays of rank 1
v = np.array([9, 10])
w = np.array([11, 12])

# Inner product of vectors
print(np.dot(v, w), "\n")

# Matrix and Vector product
print(np.dot(x, v), "\n")

# Matrix and matrix product
print(np.dot(x, y))
```

Output:

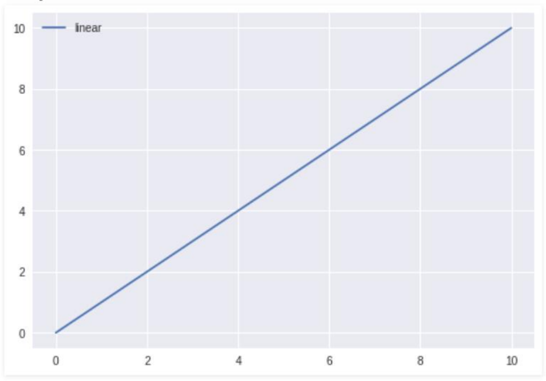
```
219 [29 67] [[19 22] [43
50]]
```

B. MATPLOTLIB

Matpoltlib is a very popular Python library for data visualization. Like Pandas, it is not directly related to Machine Learning. It particularly comes in handy when a programmer wants to visualize the patterns in the data. It is a 2D plotting library used for creating 2D graphs and plots. A module named pyplot makes it easy for programmers for plotting as it provides features to control line styles, font properties, formatting axes, etc. It provides various kinds of graphs and plots for data visualization, viz., histogram, error charts, bar chats, etc,

```
#  Python program using Matplotib
# for forming a linear plot

# importing the necessary packages and modules
import matplotlib.pyplot as plt
import numpy as np

# Prepare the data
x = np.linspace(0, 10, 100)

# Plot the data
plt.plot(x, x, label ='linear')

# Add a legend
plt.legend()

# Show the plot
plt.show()
```

Output

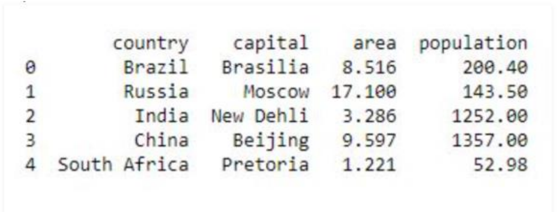


C. PANDAS

Pandas is a popular Python library for data analysis. It is not directly related to Machine Learning. As we know that the dataset must be prepared before training. In this case, Pandas comes handy as it was developed specifically for data extraction and preparation. It provides high-level data structures and wide variety tools for data analysis. It provides many inbuilt methods for groping, combining and filtering data.

```
# Python program using Pandas for
# arranging a given set of data
# into a  table

# importing pandas as pd
import pandas as pd

data = {"country": ["Brazil", "Russia", "India", "China", "South Africa"],
       "capital": ["Brasilia", "Moscow", "New Dehli", "Beijing", "Pretoria"],
       "area": [8.516, 17.10, 3.286, 9.597, 1.221],
       "population": [200.4, 143.5, 1252, 1357, 52.98] }

data_table = pd.DataFrame(data)
print(data_table)
```

Output

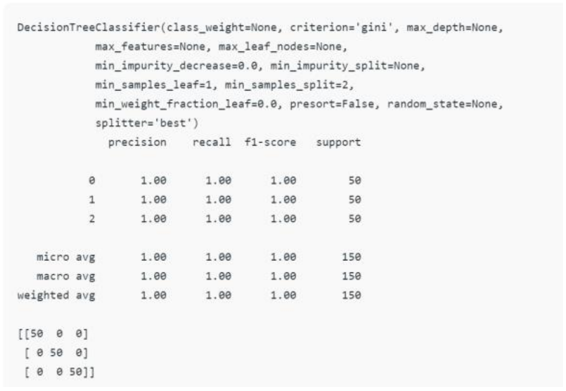


```
        country    capital    area  population
0        Brazil   Brasilia   8.516      200.40
1        Russia     Moscow  17.100      143.50
2         India  New Dehli   3.286     1252.00
3         China    Beijing   9.597     1357.00
4  South Africa   Pretoria   1.221       52.98
```

D. Scikit-learn

Skikit-learn is one of the most popular ML libraries for classical ML algorithms. It is built on top of two basic Python libraries, viz., NumPy and SciPy. Scikit-learn supports most of the supervised and unsupervised learning algorithms. Scikit-learn can also be used for data-mining and data-analysis, which makes it a great tool who is starting out with ML.

```
# Python script using Scikit-learn
# for Decision Tree Clasifier

# Sample Decision Tree Classifier
from sklearn import datasets
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier

# load the iris datasets
dataset = datasets.load_iris()

# fit a CART model to the data
model = DecisionTreeClassifier()
model.fit(dataset.data, dataset.target)
print(model)

# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)

# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```



```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        50
           2       1.00      1.00      1.00        50

   micro avg       1.00      1.00      1.00       150
   macro avg       1.00      1.00      1.00       150
weighted avg       1.00      1.00      1.00       150

[[50  0  0]
 [ 0 50  0]
 [ 0  0 50]]
```

1 Introduction

What is machine learning? We probably use it dozens of times a day without even knowing it. Each time us do a web search on Google or Bing, that works so well because their machine learning software has figured out how to rank what pages. When Facebook or Apple's photo application recognizes our friends in our pictures, that's also machine learning. Each time we read our email and a spam filter saves us from having to wade through tons of spam, again, that's because our computer has learned to distinguish spam from non-spam email. So, that's machinelearning.

One of the reasons I'm excited about this is the **AI**, or artificial intelligence problem.Building truly intelligent machines, we can do just about anything that you or I can do. Many scientists think the best way to make progress on this is through learning algorithms called **neural networks**, which mimic how the human brain works.

1. Defintion

The introduction given above gives us a broad view on machine learning. Even among machine learning practitioners, there isn't a well accepted definition of what is and what isn't machine learning. People tried to define it in different ways. Here aresome:

- **Arthur Samuel(1959):**

  Field of study that gives computers the ability to learn without being explicitly programmed.This is an older, informal definition.

- **Tom Mitchell(1998):**

  A computer program is said to learn from experience **E** with respect to some task **T** and some performance **P**, if its performance on **T**, as measured by **P**, improves with experience **E**.

2. Types

In general, any machine learning problem can be assigned to one of two broad classifications:

- **Supervised Learning**
- **Unsupervised Learning**

Supervised Learning

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

Supervised learning problems are categorized into **regression** and **classification** problems.

Regression : In a regression problem, we are trying to predict results within a continuous output, meani ng that we are trying to map input variables to some continuous function.

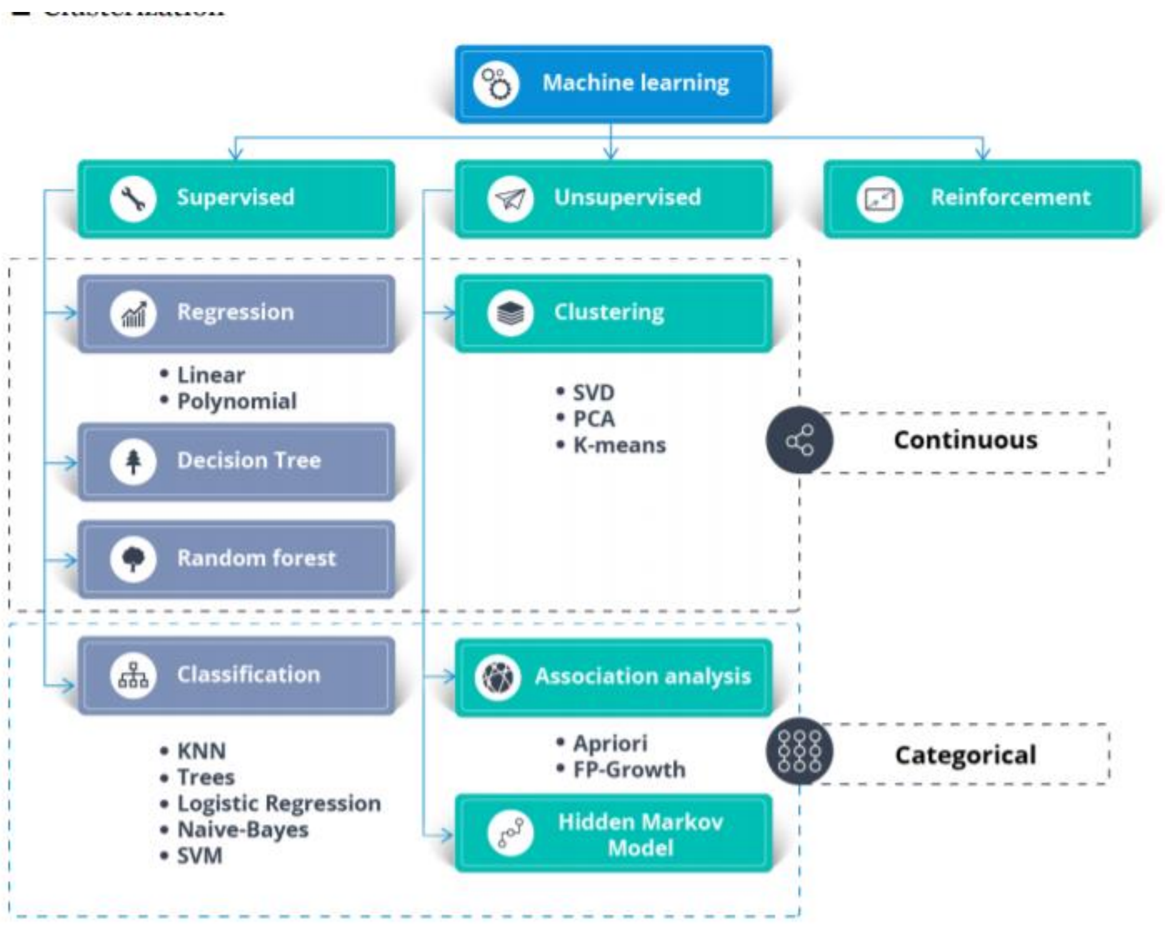
Classification : In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Unsupervised Learning

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

Clustering : Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering : The **Cocktail Party Algorithm**, allows you to find structure in a chaotic en- vironment(i.e identifying individual voices and music from a mesh of sounds at a <u>**cocktail party**</u>).



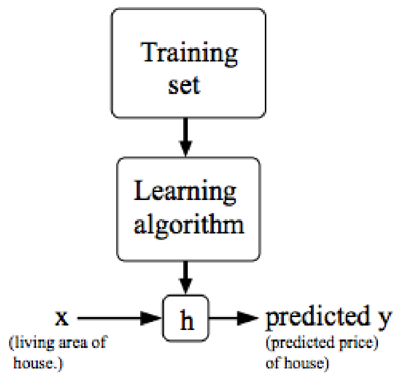
2 Model and Cost Function

1. Model Representation

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables, also called input

features, and $y^{(i)}$ to denote the "output" or target variable that we are trying to predict. A pair $x^{(i)}, y^{(i)}$ is called a training example, and the dataset that we will be using to learn - a list of m training examples is called a "training set". The superscript "(i)" has nothing to do with exponenti- ation. We will also use X to denote the space of input values, and Y to denote the space of output values.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : X \leftarrow Y$ so that $h(x)$ is a "good" predictor for the corresponding values of y. This function h is called a **hypothesis**.



2.2 Cost Function

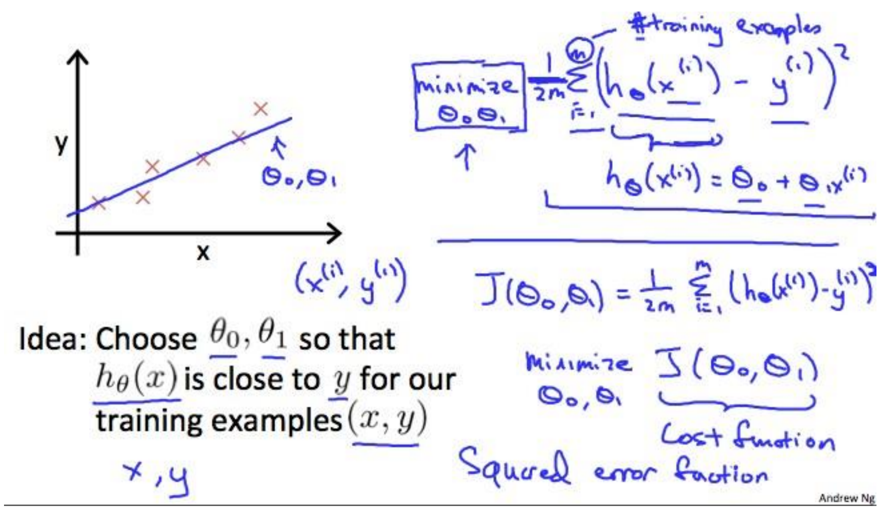
We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's

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

derivative term of the square function will cancel out the $\frac{1}{2}$ term.

To break it apart, it is $\frac{1}{2}\bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value. This function is otherwise called the "Squared error function". The mean is halved as a convenience for the computation of the gradient descent, as the

2



y
x
θ_0, θ_1
$(x^{(i)}, y^{(i)})$
minimize $\theta_0\theta_1$
$\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$
# training examples
$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$
$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$
minimize $J(\theta_0,\theta_1)$ θ_0,θ_1
Cost function
Squared error function
x, y
Andrew Ng

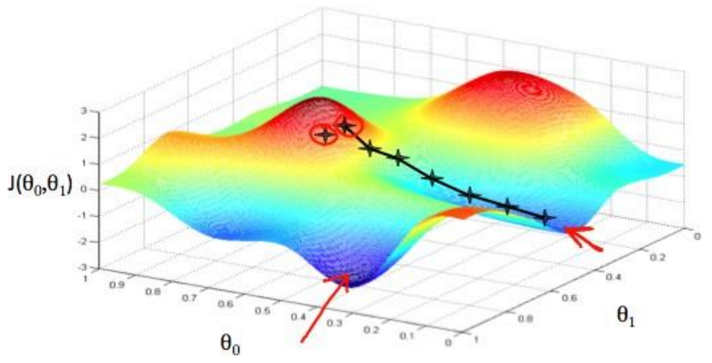
Idea: Choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y)

3 Parameter Learning

1. Gradient Descent

We have a 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.

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.



We are successful when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

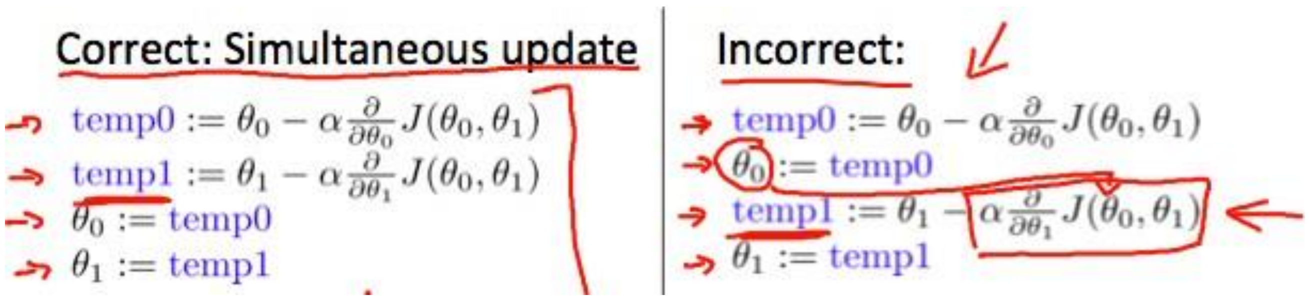
We do this by taking the derivative of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, whiich is called the **learning rate**.The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. Depending on where one starts on the graph, one could end up at different points.

The gradient descent algorithm is:
repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

where j = 0,1 represents the feature index number.

At each iteration j, one should simultaneously update the parameters $\theta_0,\theta_1,...\theta_n$. Updating a specific parameter prior to calculating another one on the j^{th} iteration would yield to a wrong imple- mentation.



Correct: Simultaneous update

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$

$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

$\theta_0 := \text{temp0}$

$\theta_1 := \text{temp1}$

Incorrect:

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$

$\theta_0 := \text{temp0}$

$\text{temp1} := \theta_1 - \boxed{\alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)}$

$\theta_1 := \text{temp1}$

3.2 Gradient Descent For Linear Regression

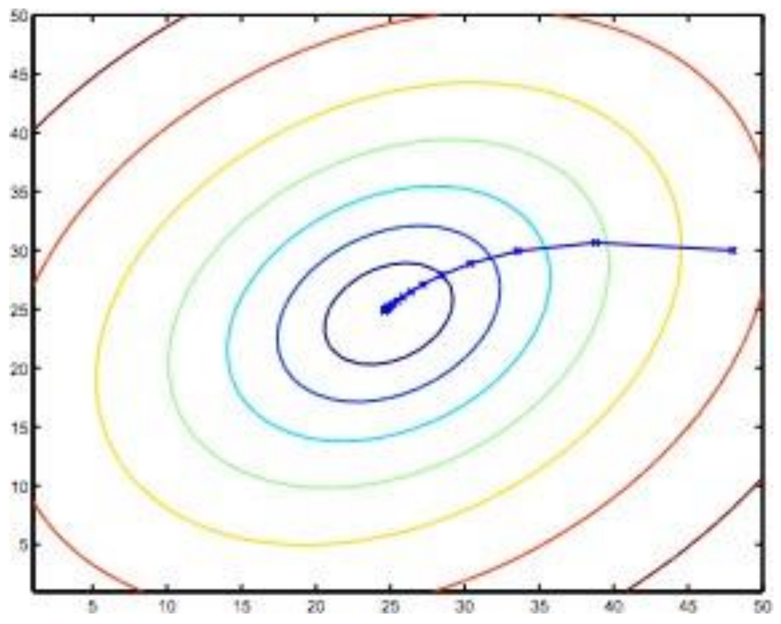
When specifically applied to the case of linear regression, gradient descent looks like

: repeat until convergence: $\{$

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \end{aligned}$$

$\}$

with usual notations.This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which was initialized at (48, 30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through as it converged to its minimum.

4 Multivariate Linear Regression

Linear regression with multiple variables is also known as “multivariate linear regression”.

1. Multiple Features

We follow the following notation :

$x_j^{(i)}$ = **value of feature j in the i^{th} trainingexample**

$x^{(i)}$ = **the input (features) of the i^{th} trainingexample**

m = **the number of training examples**

n = **the number offeatures**

The multivariate form of the hypothesis function accommodating these multiple features is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n$$

The above equation can be vectorized, for one training example, as follows :

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

4.2 Gradient Descent For Multiple Variables

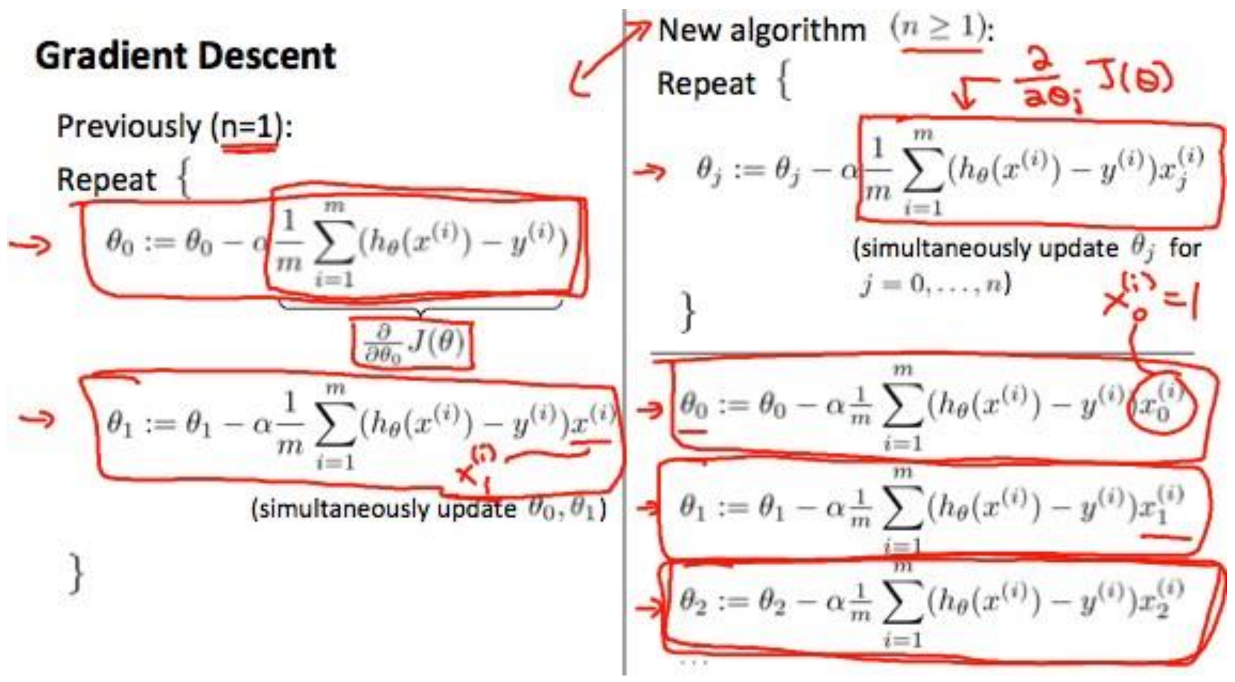
The gradient descent equation itself is generally the same form; we just have to repeat it for our ‘n’ features:

repeat until convergence: {

m

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \qquad \textbf{for } j := 0 \ldots n$$

}



Gradient Descent

Previously (n=1):

Repeat {

$$\rightarrow \boxed{\theta_0 := \theta_0 - \alpha \underbrace{\boxed{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})}}_{\boxed{\frac{\partial}{\partial \theta_0} J(\theta)}}}$$

$$\rightarrow \boxed{\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \underline{x^{(i)}}} \quad x_1^{(i)}$$

(simultaneously update θ_0, θ_1)

}

New algorithm $(n \geq 1)$:

Repeat {

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

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

(simultaneously update θ_j for $j = 0, \ldots, n$)

}

$x_0^{(i)} = 1$

$$\rightarrow \boxed{\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}}$$

$$\rightarrow \boxed{\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}}$$

$$\rightarrow \boxed{\theta_2 := \theta_2 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}}$$

...

4.3 Feature Scaling and Mean Normalization

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

Feature scaling 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. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero.

$$x_i := \frac{x_i - \mu_i}{s_i}$$

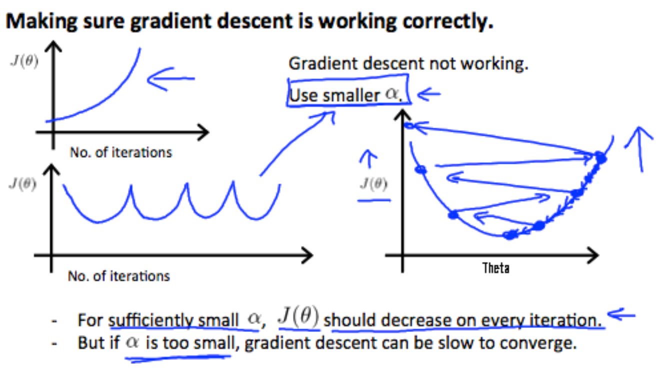
where μ_i is the 'average' of all the values for the feature (i) and s_i is the range of values or the standard deviation.

4.4 Learning Rate

Debuging gradient descent: Plot $J(\theta)$ vs number of iterations of gradient descent.If $J(\theta)$ ever increases, we have to decrease α.

Automatic convergence test: Declare convergence if $J(\theta)$ decreases by less than s in one iteration, where s is some small value such as 10^{-3}. However in practice it's difficult to choose this threshold value.

It has been proving that if learning rate α is sufficiently small, then $J(\theta)$ will decrease on every iteration.



To summarize :

- If α is too small : slow convergence
- If α is too large : may not decrease on every iteration and every iteration and thus may not converge.

4.5 Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking x_1x_2.

Polynomial Regression: We can **change the behaviour or curve** of our hypothesis function by making it a quadratic, cubic or square root function(or any other form) by creating additional features. Feature scaling becomes very important in this case.

5 Computing Parameters Analytically

This is another method to minimize our cost function $J(\theta)$. In the **Normal Equation** method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. The normal equation formula is given as follows:

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

There is no need to do feature scaling for normal equation.The following is a comparision of gradient descent and the normal equation:

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$, need to calculate inverse of $X^T X$
Works well when n is large	Slow if n is very large

Normal Equation Noninvertibility: If $X^T X$ is **noninvertible**, the common causes might be having:

- Redundant features, where two features are very closely related(i.e. they are linearly dependent)
- Too many features(e.g. $m \leq n$). In this case, delete some features or use "regularization"(will be detailed later).

6 Classification and Representation

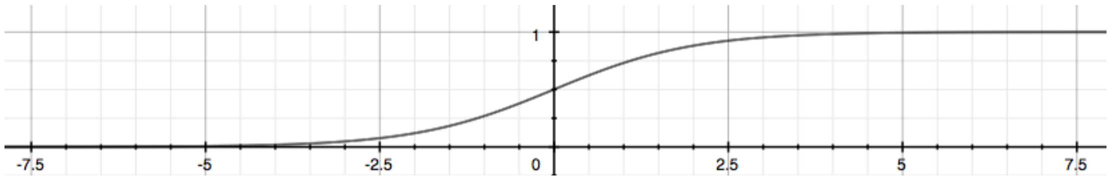
1. Classification

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. First we will focus on the **binary classification problem** in which y can take on only two values, 0 and 1. This will be generalized later to multi-class classification problem. Hence, $y \in \{0, 1\}$. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+". Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example.

2. Hypothesis Representation

Intuitively, it doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$. This is accomplished by plugging $\theta^T x$ into the Logistic Function. Our new form uses the "Sigmoid Function," also called the "Logistic Function":

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



$h_\theta(x)$ will give the **probability** that our output is 1.

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

3. Decision Boundary

The **decision boundary** is the line or the curve that seperates the area where $y = 0$ and where $y = 1$. It is created by our hypothesis function.

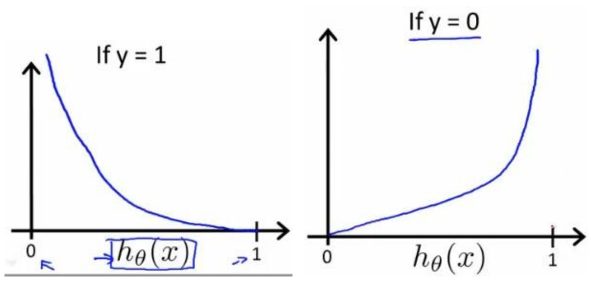
$$\theta^T x \geq 0 \rightarrow h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$\theta^T x < 0 \rightarrow h_\theta(x) < 0.5 \rightarrow y = 0$$

4. Cost Function and Gradient Descent

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function. Instead, our cost function for logistic regression looks like:

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

6



Writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression. A vectorized implementation is:

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m}\left(-y^T\log(h) - (1-y)^T\log(1-h)\right)$$

For gradient descent, a vectorized implementation is:

repeat {

$$\theta := \theta - \frac{\alpha}{m}X^T\left(g(X\theta) - \vec{y}\right)$$

}

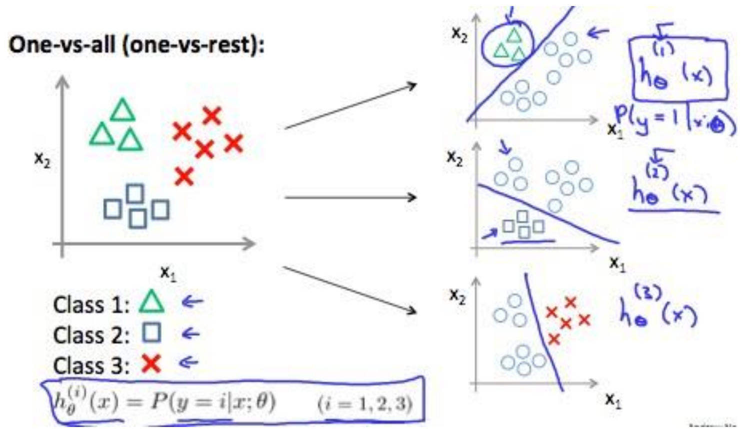
Notice that this algorithm is identical to the one we used in linear regression. But the hypothesis is different.

5. Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. Octave provides them in libraries. We first need to provide a function that evaluates both cost function and it's partial derivatives. If we pass this function to the advanced optimization algorithms, we get the optimum values of θ directly.

6. Multiclass Classification: One-vs-all

Let's say we have n classes. We can deal this by dividing our problem into n + 1 binary classification problems; in each one, we predict the probability of occurence of a particular class. 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.



7 Regularization

1. The Problem of Overfitting

Underfitting, or high bias, is when the form of our hypothesis function h 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.

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(will be detailed later).
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.

2. Regularized Linear Regression

To reduce the magnitude of parameters, we modify our cost function as follows:

Here, λ is called the **regularization parameter**. It determines hhow much the costs of our θ parameters are inflated. Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

Gradient Descent

We will modify our gradient descent function to seperate out θ_0 from the rest of the parameters because we don't want to penalize θ_0.

repeat $\{$

$$\begin{aligned}\theta_0 &:= \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha\left[\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j\right]\end{aligned}$$

Normal Equation

The formula for optimum θ will get modified as follows:

$$\theta = (X^T X + \lambda . L)^{-1} X^T y$$

$$\text{where } L = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension $(n + 1)$x$(n + 1)$. Recall that if $m < n$, then $X^T X$ is non-invertible. However, when we add the term $\lambda.L$, then $X^T X + \lambda.L$ becomes invertible.

8 Neural Networks

Neural networks are pretty old algorithms that was originally motivated by the goal of having machines that can mimic brain. This is generally used for non-linear hypotheses. A most common example where this is used is the problem of computer vision-object classification. These are computationally expensive.

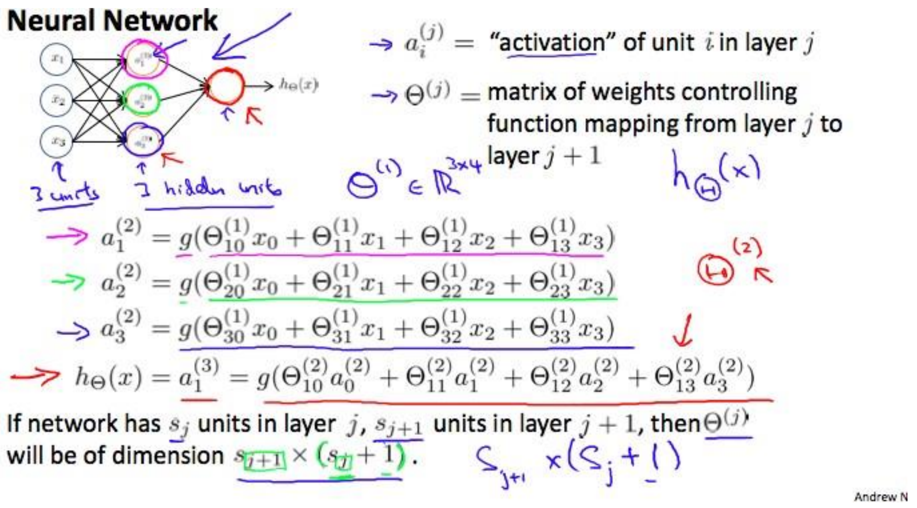
1. Model Representation

Neural networks are developed as simulating the networks of neurons in the brain. At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1 \dots x_n$, and the output is the result of our hypothesis function.In this model our x_0 input node is sometimes called the "bias unit", which is always equal to 1. We use the same logistic function as in classification,yet it is sometimes called a sigmoid **activation** function. Alsothe θ parameters are sometimes called "weights".

Our input nodes, also known as the **input layer**, go into a series of intermediate layers called the **hidden layers**, and finally the layer in which we get our output is called the **output layer**. This architecture is soomewhat similar to the computational graphs. The following notations are used:

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

If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension s_{j+1} x $(s_j + 1)$.



Neural Network

→ $a_i^{(j)} =$ "activation" of unit i in layer j

→ $\Theta^{(j)} =$ matrix of weights controlling function mapping from layer j to layer $j+1$

3 units   3 hidden units $\Theta^{(1)} \in \mathbb{R}^{3\times4}$ $h_\Theta(x)$

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

$\Theta^{(2)}$

→ If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. $s_{j+1} \times (s_j + 1)$

Andrew N

For a vectorized implementation, we will define a new variable $z_k^{(j)}$ that encompasses the parameters going into the sigmoid function.

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

Then we will have :

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

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This element will

be $a_0^{(j)}$ and will be equal to 1. Notice that in this last step, between layer j and layer $j+1$, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

To classify data into multiple classes, we let our hypothesis function return a vector of values, the size being wqual to the number of classes. The maximum value among the components is used to predict the class for a new test case.

8.2 Cost Function

The following convention will be followed:

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

$h_\theta(x)_k$ = hypothesis that results in the k^{th} output

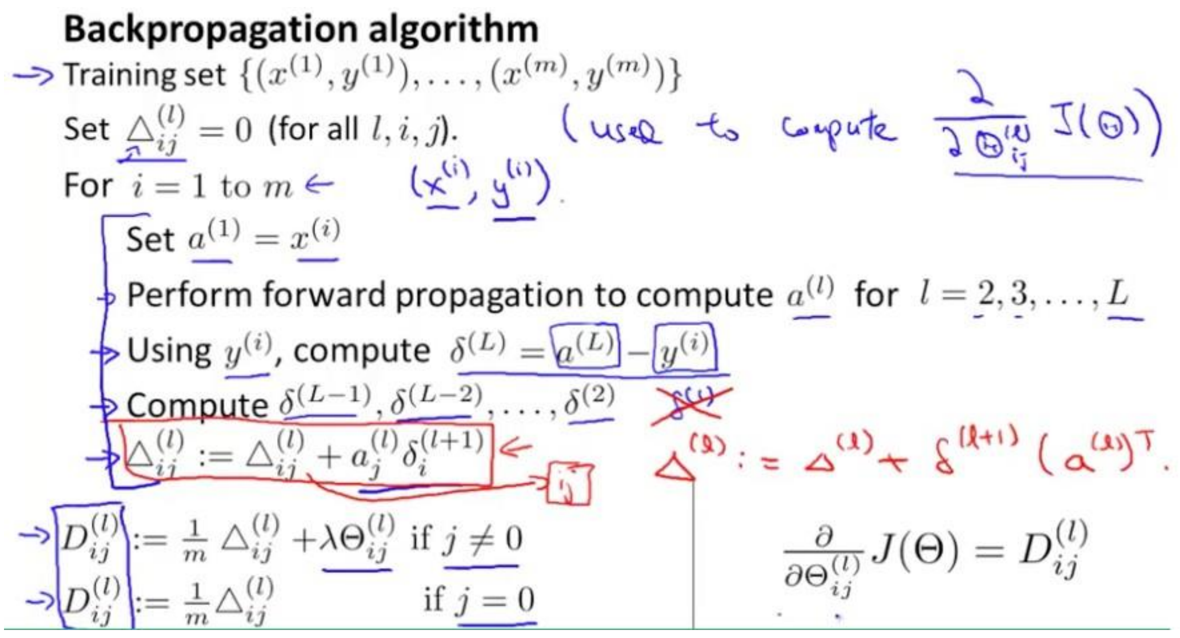
The cost function for neural networks will be a generalization of the one used for logistic regression. For neural networks, it will be slightly complicated:

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

The double sum simply adds up the logistic regression costs calculated for each cell in the output layer. The triple sum simply adds up the squares of all the individual Θs in the entire network. The i in the triple sum does **not** refer to training example i.

8.3 Backpropagation Algorithm

"Backpropagation" is a neural-network terminology for minimizing our cost function, just like the gradient descent in logistic and linear regression. We will compute the partial derivatives of $J(\Theta)$ as follows:



Backpropagation algorithm

```
→ Training set $\{(x^{(1)},y^{(1)}),\dots,(x^{(m)},y^{(m)})\}$
  Set $\triangle_{ij}^{(l)} = 0$ (for all $l,i,j$).   (use to compute $\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$)
  For $i = 1$ to $m$ ←   $(x^{(i)}, y^{(i)})$.
      Set $a^{(1)} = x^{(i)}$
      → Perform forward propagation to compute $a^{(l)}$ for $l = 2,3,\dots,L$
      → Using $y^{(i)}$, compute $\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}$
      → Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$   $\cancel{\delta^{(1)}}$
      → $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}}$ ←   $\boxed{ij}$   $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.
→ $\boxed{D_{ij}^{(l)}} := \frac{1}{m}\triangle_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ if $j \neq 0$
→ $\boxed{D_{ij}^{(l)}} := \frac{1}{m}\triangle_{ij}^{(l)}$   if $j = 0$
```

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)}$$

For each training example, after performing forward propagation, we compute the errors $\delta^{(l)}$ at each layer by using the concept of chain rule from calculus. So our error for the last layer is simply the difference between the actual output and the value of our hypothesis. The errors of the remaining layers can be calculated as follows :

$$\delta^{(l)} = \left((\Theta^{(l)})^T \delta^{(l+1)}\right) .* a^{(l)} .* (1 - a^{(l)})$$

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 the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.

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

As shown in the figure, the derivative of the non regularized part at each layer is stored in the Δ matrix. After considering regularization, the partial derivatives are stored in the vector D where :

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

We then use advanced optimization techniques provided by the software to compute the optimum values of the weights.

8.4 Gradient Checking

Gradient checking is used to check whether backpropagation is working as intended. We can approx- imate the derivative as follows :

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

where s is a small positive number($\sim 10^{-4}$).

By comparing the approximate gradient with the gradient computed using backprop, we can verify. **Once** verification is done, approximate gradient is not needed anymore and hence not computed. Backprop is used for learning.

Note: Always initializing the weights to zero doesn't work. This may lead to gradient killing. So we generally initialize the weights to a random value between $[-s, s]$. This s is unrelated to the one used in gradient checking.

5. Overall Procedure

1. Pick a suitable network architecture.
2. Randomly initialize weights.
3. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$.
4. Implement the cost function.
5. Implement the backpropagation to compute partial derivatives.
6. Use gradient checking to confirm that backprop works. Then disable gradient checking.
7. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in θ.

When we perform forward and back propagation, we loop on every training example.

9 Evaluating a Learning Algorithm

1. Model Selection and Train/Validation/Test Sets

Just because a learning algorithm fits a training set well, that doesn't mean it is a good hypothesis. It could over fit and result in poor predictions. The error of the hypothesis as measured on the training data set is always lower than the error on any other data set.
Any learning algorithm has some parameters called **hyper parameters**. These parameters are to be chosen such that the error will be minimum(e.g. regularization parameter λ, degree of the polynomial features, learning rate α, etc.). But choosing them based on the test set will lead to their biased values. So what is done in practice is that the given data set is broken into three sets:

- Training set : 60%
- Cross validation set : 20%
- Test set : 20%

We then calculate three separate error values for the three different sets using the following method :

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the hyper parameter with the least error using the cross validation set.
3. Estimate the generalization error using the test set.

This way, hyper parameters have not been trained using the test set.

10 Support Vector Machines

These are also called as large margin classifiers. An SVM is a slightly or computationally simpler version of logistic regression. An interesting feature of SVM is it is not sensitive tooutliers.

1. Optimization Objective

For logistic regression

If y = 1, we want $h_\theta(x) \approx 1$, $\theta^T x$ 0

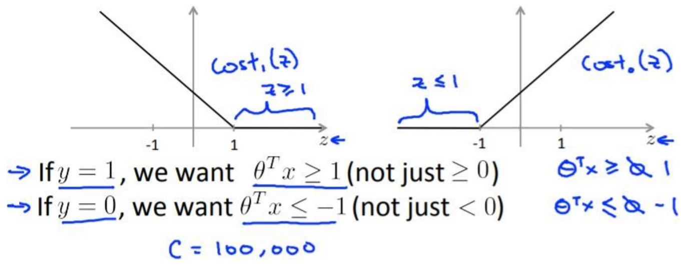
If y = 0, we want $h_\theta(x) \approx 0$, $\theta^T x$ 0 and

the optimization objective is :

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

But for SVM the optimization objective is the following :

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



→ If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0) $\theta^T x \geq \alpha$ 1

→ If $y = 0$, we want $\theta^T x \leq -1$ (not just < 0) $\theta^T x \leq \alpha$ -1

C = 100,000

10.2 Kernels

For non-linear decision boundaries, kernels are used. Kernels are nothing but similarity functions and are used to create new features from the training data instead of using the polynomial features that are used in convention, for logistic regression.
Given an input feature x, new features are computed depending on the proximity to landmarks $l^{(i)}$. These proximities are computed in general using the gaussian kernels or f_1 similarity function. It is defined as follows :

$$f_1 = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{\left\|x - l^{(i)}\right\|^2}{2\sigma^2}\right)$$

If $x \approx l^{(i)}$ then $f_1 \approx 1$

If x is far from $l^{(i)}$ then $f_1 \approx 0$

Generally all the training examples are chosen as landmarks and then new features are computed using gaussian kernels. Let the features be denoted by f . We also have a bias feature f_0 which is always equal to 1. So our $f \in R^{m+1}$ and also $\theta \in R^{m+1}$. So :

Hypothesis: Predict "y = 1" if $\theta^T f \geq 0$

Training:

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

Not all similarity functions make valid kernels. They need to satisfy a technical condition called **Mercer's Theorem.**

3. Bias and Variance

- Large C: Lower bias, high variance.
- Small C: Higher bias, low variance.
- Large σ^2: Features f_i vary more smoothly, hence higher bias and lower variance.
- Small σ^2: Features f_i vary less smoothly, hence lower bias and higher variance.

4. Logistic Regression vs SVMs

Let n be the number of features and m be the number of training examples.

- If $n \geq m$: Use logisti regression or SVM without a kernel("linear kernel").
- If $n < m$, 10$n \sim m$: Use SVM with Gaussian kernel.
- If n m: Create/add more features, then use logistic regression or SVM without a kernel. Neural

network is likely to work well for most of these settings, but may be slower to train.

11 Clustering

Clustering is an unsupervised learning algorithm. This algorithm is used to cluster and unlabelled data set. It is usually called K-means clustering which is nothing but clustering the data into K clusters.

1. K-means Algorithm

This algorithm is a method to automatically cluster similar data examples together. Concretely, we have a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \mathbb{R}^n$), and want to group the data into a few cohesive "clusters". The intuition behind K-means is an iterative procedure that starts by guessing the initial centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then recomputing the centroids based on the assignments.
The inner-loop of the algorithm repeatedly carries out two steps:

1. Assigning each training example $x^{(i)}$ to its closest centroid, and
2. Recomputing the mean of each centroid using the points assigned to it.

The K-means algorithm will always converge to some final set of means for the centroids. But the converged solution may not always be ideal and depends on the initial setting of the centroids. Therefore, in practice the K-means algorithm is usually run a few times with different random initializations. One way to choose between these different solutions from different random initializations is to choose the one with the lowest cost function value.
The cost function is the mean-squared distance between each training example and the centroid assigned to it. A general practice to choose the centroids is to randomly select K training examples.

12 Dimensionality Reduction

Dimensionality reduction is required to compress the data and to visualize it. In real life, the data will have lots and lots of features and there will be some dimensional ordering in the data.
Dimentionality reduction is formulated as the Principal Component Analysis problem.

1. Principal Component Analysis

The task of PCA is to reduce the data from n-dimensions to k-dimensions where $k < n$. So we need to find k vectors $u^{(1)}, u^{(2)}, \ldots, u^{(k)}$ in $\mathbb{R}^n$ onto which we can project our data so as to minimize the projection. It is implemented as follow:

1. **Data preprocessing:** Before applying PCA, we must do mean normalization. Feature scaling is not always necessary.
2. **Covariance matrix:** We then compute the covariance matrix Σ as follows:

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T$$

3. **Computing eigen vectors of** Σ**:** We then compute the eigen vectors of Σ by using singular value decomposition function.
4. **Getting the new features:** By using the first k eigen vectors, we can get k- dimensional features by doing the following:

$$z^{(i)} = U^T_{reduce} x^{(i)}$$

where U_{reduce} has the first k eigen vectors of Σ as its columns.

The approximate reconstruction from compressed representation is done as follows:

$$X_{approx} = U_{reduce} Z$$

Choosing the number of principal components: The value of k is chosen based on the ratio of the average squared projection error to the total variation in the data. The ratio is typically smaller than 0.01 so that 99% of variance is retained.

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \leq 0.01 \quad (1\%)$$

Also PCA **should not** be used to prevent overfitting. Regularization must be used for that purpose.

13 Deep Learning

1. Introduction

Deep Learning is a field which is a subset to machine learning. This field mainly focuses on neural network algorithms and their optimizations. The word “deep” suggests that a large number of hidden layers are used in the neural network architecture. Deep learning focuses on extracting patterns from the data without explicitly being programmed. This field came into light very recently and is still an area of active research.

2. Activation functions

We need non-linear activation functions to classify complex data distributions. So far in neural networks we have used only “‘sigmoid” activation function, i.e. for logistic regression. But this activation function has certain problems:

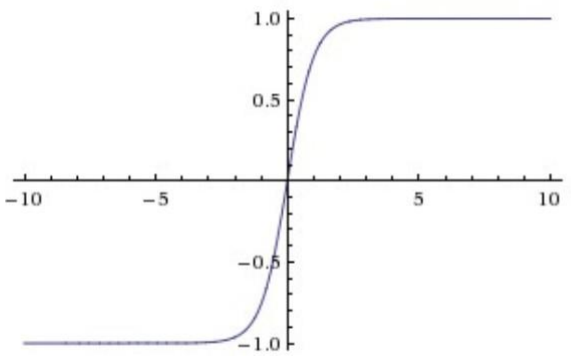
- **Sigmoids saturate and kill gradients**: A very undesirable property of the sigmoid neuron is that when the neuron’s activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate’s output for the whole objective. Therefore, if the local gradient is very small, it will effectively “kill” the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.

- **Sigmoid outputs are not zero-centered**: This is undesirable since neurons in later layers of processing in a neural network would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive then the gradient on the weights will during backpropagation become either all be positive, or all negative. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

We will look into some other activation functions.

Tanh: The tanh non-linearity squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds:

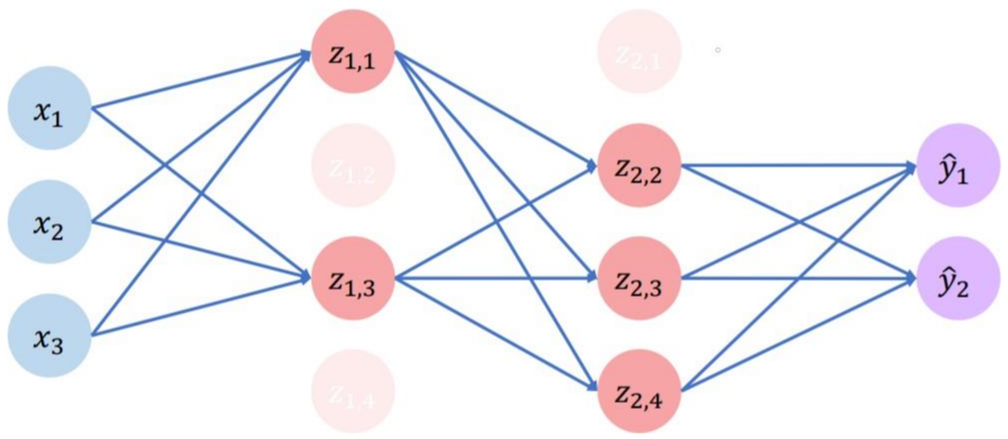
$$\tanh(x) = 2\sigma(2x) - 1$$



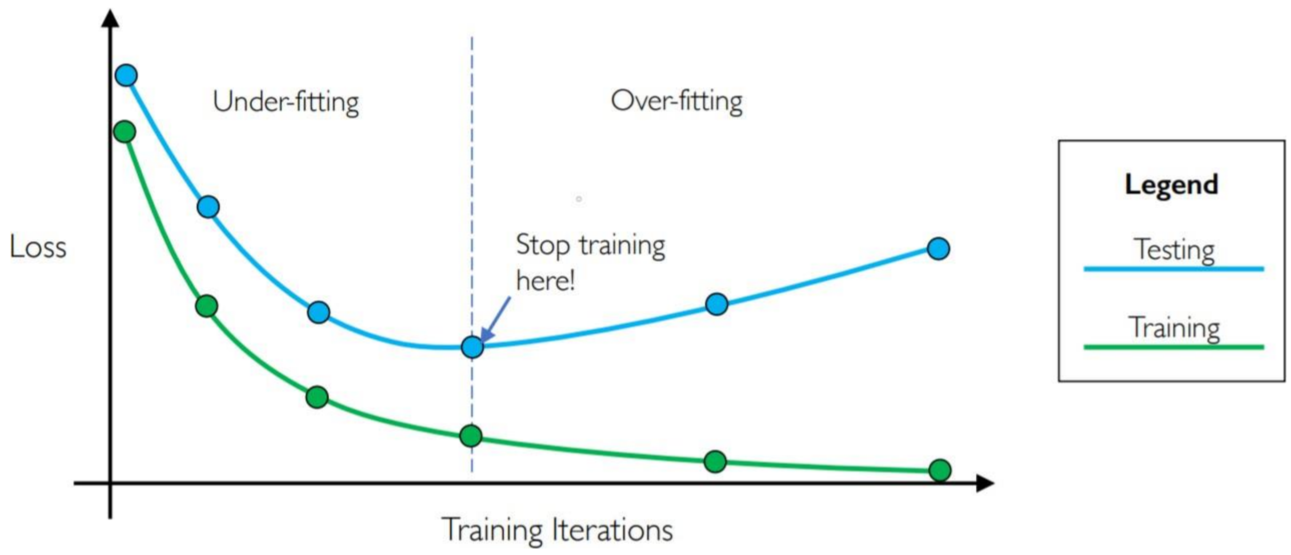
13.4 Regularization

This is a technique that constraints our optimization problem to discourage complex models. This is needed to improve the generalization of our model on unseen data. There are various ways to account this:

Dropout: During training randomly set some activations to zero during forward pass. Typically we drop 50% of activations in layer. This forces network to not rely on any one node.



Early Stopping: Stop training before we can overfit.

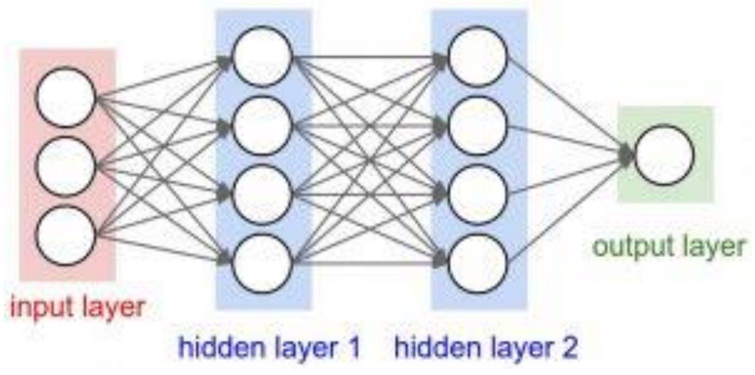


When it comes to Machine Learning, Artificial Neural Networks perform really well. Artificial Neural Networks are used in various classification task like image, audio, words. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use Recurrent Neural Networks more precisely an LSTM, similarly for image classification we use Convolution Neural Network. In

Regular neural networks are three types of layers:

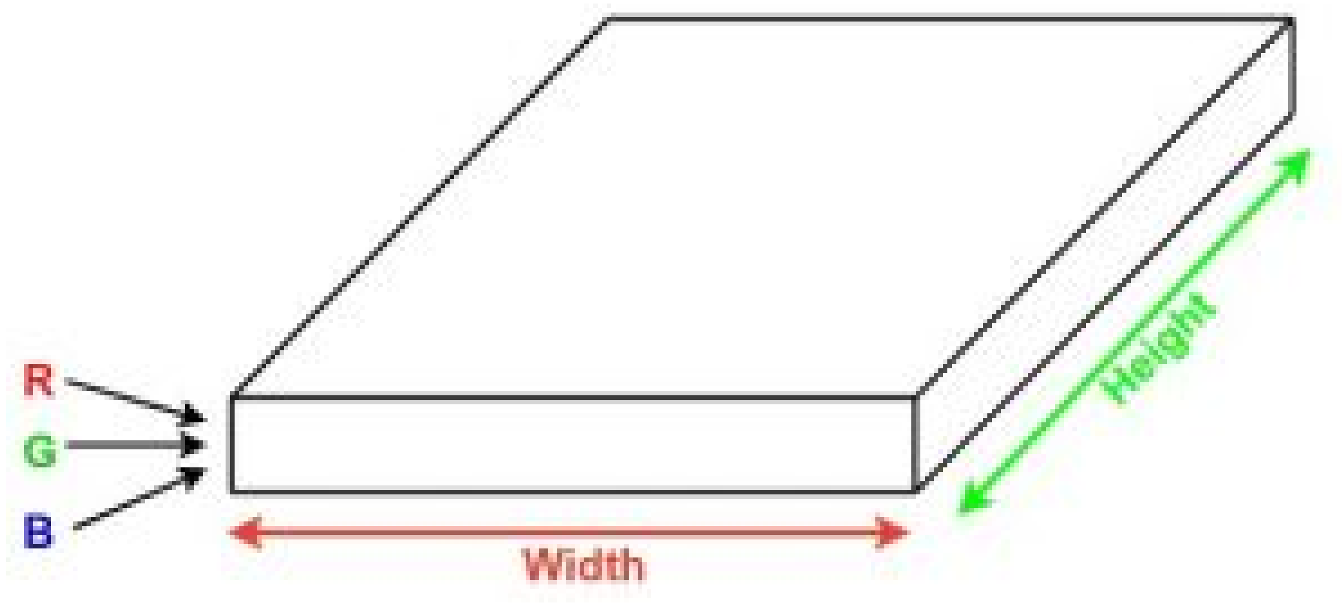
- **Input Layers:** It's the layer in which we give input to our model. The number of neurons in this layer is equal to total number of features in our data (number of pixels in case of an image).
- **Hidden Layer:** The input from Input layer is then feed into the hidden layer. There can be many hidden layers depending upon our model and data size. Each hidden layers can have different numbers of neurons which are generally greater than the number of features. The output from each layer is computed by matrix multiplication of output of the previous layer with learnable weights of that layer and then by addition of learnable biases followed by activation function which makes the network nonlinear.
- **Output Layer:** The output from the hidden layer is then fed into a logistic function like sigmoid or softmax which converts the output of each class into probability score of each class.

The data is then fed into the model and output from each layer is obtained this step is called feedforward, we then calculate the error using an error function, some common error functions are cross entropy, square loss error etc. After that, we backpropagate into the model by calculating the derivatives. This step is called Backpropagation which basically is used to minimize the loss.

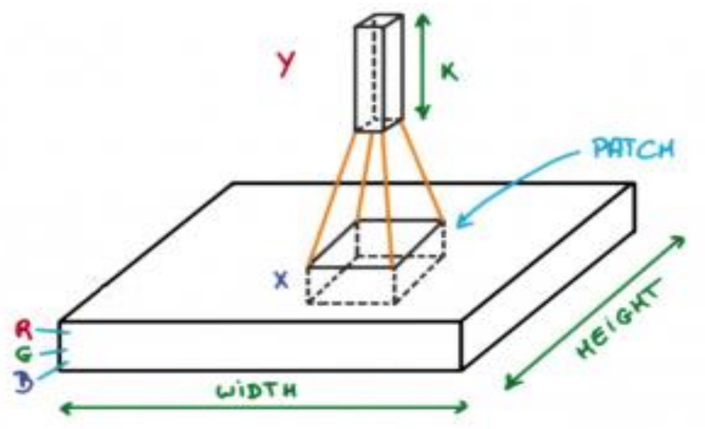


14. Convolutional Neural Networks

Convolution Neural Networks or covnets are neural networks that share their parameters. Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image) and height (as image generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network on it, with say, k outputs and represent th em vertically. Now slide that neural network across the whole image, as a result, we will get another image with different width , height, and depth. Instead of just R, G and B channels now we have more channels but lesser width and height. his operation i s called Convolution. If patch size is same as that of the image it will be a regular neural network. Because of this small pat ch, we have fewer weights.



Now let's talk about a bit of mathematics which is involved in the whole convolution process.

- Convolution layers consist of a set of learnable filters (patch in the above image). Every filter has small width and height and the same depth as that of input volume (3 if the input layer is image input).
- For example, if we have to run convolution on an image with dimension 34x34x3. Possible size of filters can be axax3, where 'a' can be 3, 5, 7, etc but small as compared to image dimension.
- During forward pass, we slide each filter across the whole input volume step by step where each step is called stride (which can have value 2 or 3 or even 4 for high dimensional images) and compute the dot product between the weights of filters and patch from input volume.
- As we slide our filters we'll get a 2-D output for each filter and we'll stack them together and as a result, we'll get output volume having a depth equal to the number of filters. The network will learn all the filters.

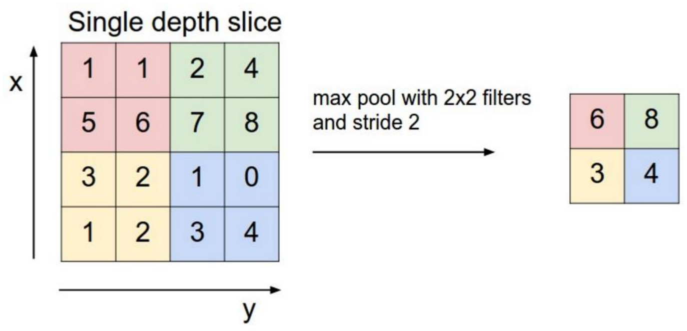
Layers used to build ConvNets

A covnets is a sequence of layers, and every layer transforms one volume to another through differentiable function.

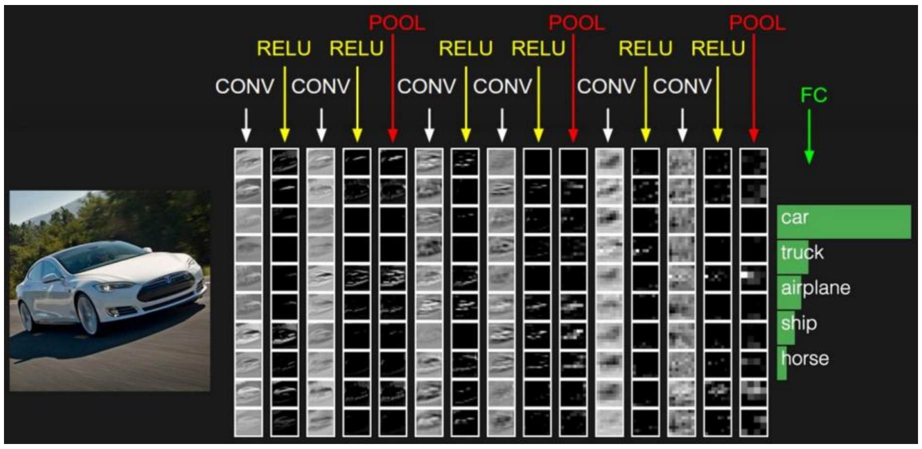
Types of layers:

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layer:** This layer holds the raw input of image with width 32, height 32 and depth 3.
- **Convolution Layer:** This layer computes the output volume by computing dot product between all filters and image patch. Suppose we use total 12 filters for this layer we'll get output volume of dimension 32 x 32 x 12.
- **Activation Function Layer:** This layer will apply element wise activation function to the output of convolution layer. Some common activation functions are RELU: max(0, x), Sigmoid: 1/(1+e^-x), Tanh, Leaky RELU, etc. The volume remains unchanged hence output volume will have dimension 32 x 32 x 12.
- **Pool Layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents from overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.

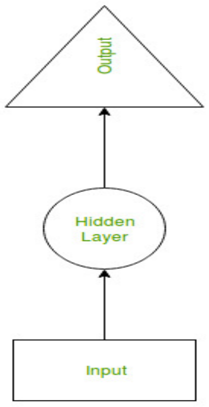


- **Fully-Connected Layer:** This layer is regular neural network layer which takes input from the previous layer and computes the class scores and outputs the 1-D array of size equal to the number of classes.



15. Recurrent Neural Network(RNN)

Recurrent Neural Network(RNN) are a type of Neural Network where the **output from previous step are fed as input to the current step**. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is **Hidden state**, which remembers some information about a sequence.

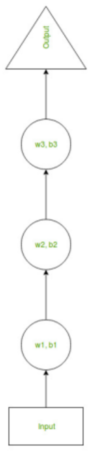


How RNN works

The working of a RNN can be understood with the help of below example:

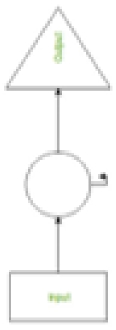
Example:

Suppose there is a deeper network with one input layer, three hidden layers and one output layer. Then like other neural networks, each hidden layer will have its own set of weights and biases, let's say, for hidden layer 1 the weights and biases are (w1, b1), (w2, b2) for second hidden layer and (w3, b3) for third hidden layer. This means that ea ese layers are independent of each other, i.e. they do not memorize the previous outputs.



Now the RNN will do the following:

- RNN converts the independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and memorizing each previous outputs by giving each output as input to the next hidden layer.
- Hence these three layers can be joined together such that the weights and bias of all the hidden layers is the same, into a single recurrent layer.



Formula for calculating current state: $h_t = f(h_{t-1}, x_t)$

```
ht -> current state
ht-1 -> previous state
xt -> input state
```

Formula for applying Activation function(tanh):

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

```
whh -> weight at recurrent neuron
wxh -> weight at input neuron
```

Formula for calculating output: $y_t = W_{hy}h_t$

```
Yt -> output
Why -> weight at output layer
```

Training through RNN

- A single time step of the input is provided to the network.
- Then calculate its current state using set of current input and the previous state.
- The current ht becomes ht-1 for the next time step.
- One can go as many time steps according to the problem and join the information from all the previous states.
- Once all the time steps are completed the final current state is used to calculate the output.
- The output is then compared to the actual output i.e the target output and the error is generated.
- The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

Advantages of Recurrent Neural Network:

- An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
- Recurrent neural network are even used with convolutional layers to extend the effective pixel neighborhood.

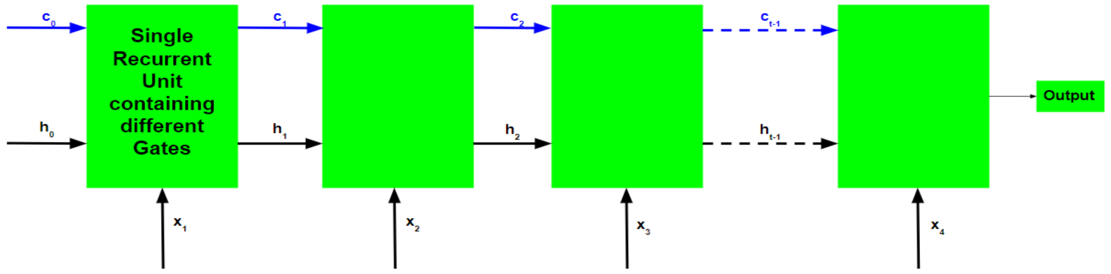
Disadvantages of Recurrent Neural Network:

- Gradient vanishing and exploding problems.
- Training an RNN is a very difficult task.
- It cannot process very long sequences if using tanh or Relu as an activation function.

16. Long Short Term Memory Networks

To solve the problem of Vanishing and Exploding Gradients in a deep Recurrent Neural Network, many variations were developed. One of the most famous of them is the **Long Short Term Memory Network**(LSTM). In concept, an LSTM recurrent unit tries to “remember” all the past knowledge that the network is seen so far and to “forget” irrelevant data. This is done by introducing different activation function layers called “gates” for different purposes. Each LSTM recurrent unit also maintains a vector called the **Internal Cell State** which conceptually describes the information that was chosen to be retained by the previous LSTM recurrent unit. A Long Short Term Memory Network consists of four different gates for different purposes as described below:-

1.Forget Gate(f): It determines to what extent to forget the previous data.
2.Input Gate(i): It determines the extent of information to be written onto the Internal Cell State.
3.Input Modulation Gate(g): It is often considered as a sub-part of the input gate and many literatures on LSTM’s do not even mention it and assume it inside the Input gate. It is used to modulate the information that the Input gate will write onto the Internal State Cell by adding non-linearity to the information and making the information **Zero-mean**. This is done to reduce the learning time as Zero-mean input has faster convergence. Although this gate’s actions are less important than the others and is often treated as a finesse-providing concept, it is good practice to include this gate into the structure of the LSTM unit.
4.Output Gate(o): It determines what output(next Hidden State) to generate from the current Internal Cell State.
The basic work-flow of a Long Short Term Memory Network is similar to the work-flow of a Recurrent Neural Network with only difference being that the Internal Cell State is also passed forward along with the Hidden State.



Working of an LSTM recurrent unit:

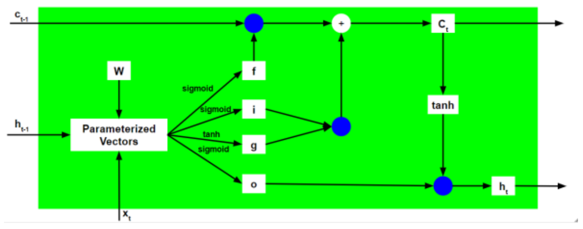
- Take input the current input, the previous hidden state and the previous internal cell state.
- Calculate the values of the four different gates by following the below steps:-
    - For each gate, calculate the parameterized vectors for the current input and the previous hidden state by element-wise multiplication with the concerned vector with the respective weights for each gate.
    - Apply the respective activation function for each gate element-wise on the parameterized vectors. Below given is the list of the gates with the activation function to be applied for the gate.

```
Input Gate            : Sigmoid Function
Forget Gate           : Sigmoid Function
Output Gate           : Sigmoid Function
Input Modulation Gate : Hyperbolic Tangent Function
```

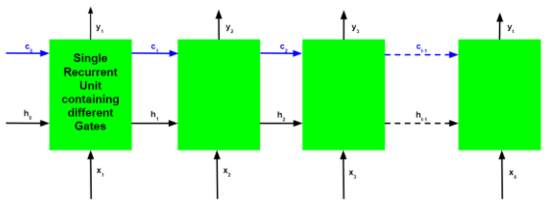
- Calculate the current internal cell state by first calculating the element-wise multiplication vector of the input gate and the input modulation gate, then calculate the element-wise multiplication vector of the forget gate and the previous internal cell state and then adding the two vectors. $c_t = i \odot g + f \odot c_{t-1}$

- Calculate the current hidden state by first taking the element-wise hyperbolic tangent of the current internal cell state vector and then performing element wise multiplication with the output gate. $h_t = o \odot tanh(c_t)$

The above stated working is illustrated as below:-



Note that the blue circles denote element-wise multiplication. The weight matrix W contains different weights for the current input vector and the previous hidden state for each gate.
Just like Recurrent Neural Networks, an LSTM network also generates an output at each time step and this output is used to train the network using gradient descent.



The only main difference between the Back-Propagation algorithms of Recurrent Neural Networks and Long Short Term Memory Networks is related to the mathematics of the algorithm.
Let be the predicted output at each time step and be the actual output at each time step. Then the error at each time step is given by:- $E_t = -y_t log(\overline{y}_t)$

The total error is thus given by the summation of errors at all time steps. $E = \sum_t E_t$

$$\Rightarrow E = \sum_t -y_t log(\overline{y}_t)$$

Similarly, the value can be calculated as the summation of the gradients at each time step.

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

Using the chain rule and using the fact that is a function of and which indeed is a function of , the following expression arises:-

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial \overline{y}_t}\frac{\partial \overline{y}_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}}\frac{\partial c_{t-1}}{\partial c_{t-2}}\dots\dots\frac{\partial c_0}{\partial W}$$

Thus the total error gradient is given by the following:-

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial \overline{y}_t}\frac{\partial \overline{y}_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}}\frac{\partial c_{t-1}}{\partial c_{t-2}}\dots\dots\frac{\partial c_0}{\partial W}$$

Note that the gradient equation involves a chain of for an LSTM Back-Propagation while the gradient equation involves a chain of dht for a basic Recurrent Neural Network.

PROJECT

USE CASE IMPLEMENTATION OF LSTM:

Stock Price Prediction using LSTM neural netwrok

GitHub Repo link for the project and report:
https://github.com/atul4411/Stock-Price-Prediction-using-Keras-and-Recurrent-Neural-Network

Acknowledgements

First and foremost, I would like to thank my mentor, Prakash Prasad, whose enthusiasm, patience and knowledge on this topic motivated me a lot. Getting to learn so much would not have been possible without you guiding the exact resources, trying to answer every little query and helping at every step. I would also like to thank the **Maths and Physics club**, IIT Bombay for this wonderful initiative. It helped me learn a lot about a topic I love and helped me spend the long summer vacations in a fruitful way.

THANK YOU!