I. Univariate, Bivariate and Multivariate Analysis

Univariate Analysis: We just pick up one feature and try to see / classify those points w.r.t output. In the y-axis, we would have nothing as there is just one feature and the points would appear in a line parallel to the x-axis.
For example: we choose weight feature - then we get the clusters in line as slim, fit and obese just with the help of weight.
But, these easily classifiable points will not always be there; there would be many overlaps.
To tackle that, we would go with bivariate and multivariate analysis.

Bivariate Analysis: For example, we take height and weight features. Based on these points are classified. We see certain clusters with some overlaps. This overlap will also help us understand which machine learning algorithm we can use to classify. For example if there is lesser overlap, we can use logistic regression because in it we are using sigmoid function and we would get higher errors. So, in case of higher overlaps, we would choose non-linear algorithms such as decision tree, KNN, random forest etc.

Multivariate analysis: What if we have multiple features. So, in order to analyze such data, we use multivariate analysis. Seaborn pairplot helps us to visualize such data. For example, features: age, height, weight. This pairplot can lead to correlation. Whenever we see age and height, can we find out if age is increasing, if height is increasing. This would show positive correlation. If we are not able to find positive or negative correlation, it might have zero correlation. We can use Pearson correlation (value ranges from -1 to +1). Should I use some non-linear classifying plane/line to classify? - all these questions would be answered by a pairplot.

II. Histograms

Histograms help us to visualize the number of points within a particular category in univariate analysis as those points are not quite evident in a univariate analysis. Default bin count is 10. Y-axis shows the count of the number of values in a particular range. We can use the Matplotlib hist() function or seaborn histogram function. It would be respect to one feature. This figure would look like a curve (if bell, then the distribution might be normal / gaussian). The Bell curve is called the Probability density function (PDF). Then , we would be converting to PDF function - at this point of time what percentage of distribution is within that particular range.

III. Z-Score statistics

In a Gaussian or normal curve, when we go to the right of the mean, we get 1 standard deviation, then 2 standard deviation. On the left of the mean, we see -1 standard deviation, -2 standard deviation and so on. Within the first standard deviation of a Gaussian distribution, we have around 68% of our information, and within the second standard deviation we have around 95% information. And, if we want to convert this entire information to our standard normal distribution where mean is 0 and standard deviation is 1, we use Z-Score.

Z-Score = (x(i) - mean)/ s.d.

If I have {1,2,3,4,5} in our distribution, then mean = 3, s.d. = 1
Z-score = 3 - 3 / 1 = 0 for x(i) = 3
And so on.

If I want to find out 1.5 s.d. Away from the mean, then we use standard normal distribution and for getting that we use Z-score and Z-score table.

Example (Student data):
Mean = 75, s.d. 10, P(x > 60) = ?
Now, when we convert it to standard normal distribution, we see that our mean 75 gets converted to mean = 0; 65 gets converted to -1 and 60 gets converted to -.15. And, we need to find the region under the standard normal curve where s.d. > -1.5

For this we would use the Z-score table, which would give us the left hand side value of area under the curve for s.d. < -1.5.

Now, our curve has 3 regions, Z-score basically gives region 3 where s.d. < -1.5. Region 1 is for the region between s.d >= -1.5 and s.d. <= 0. Region 2 is the s.d. > 0.

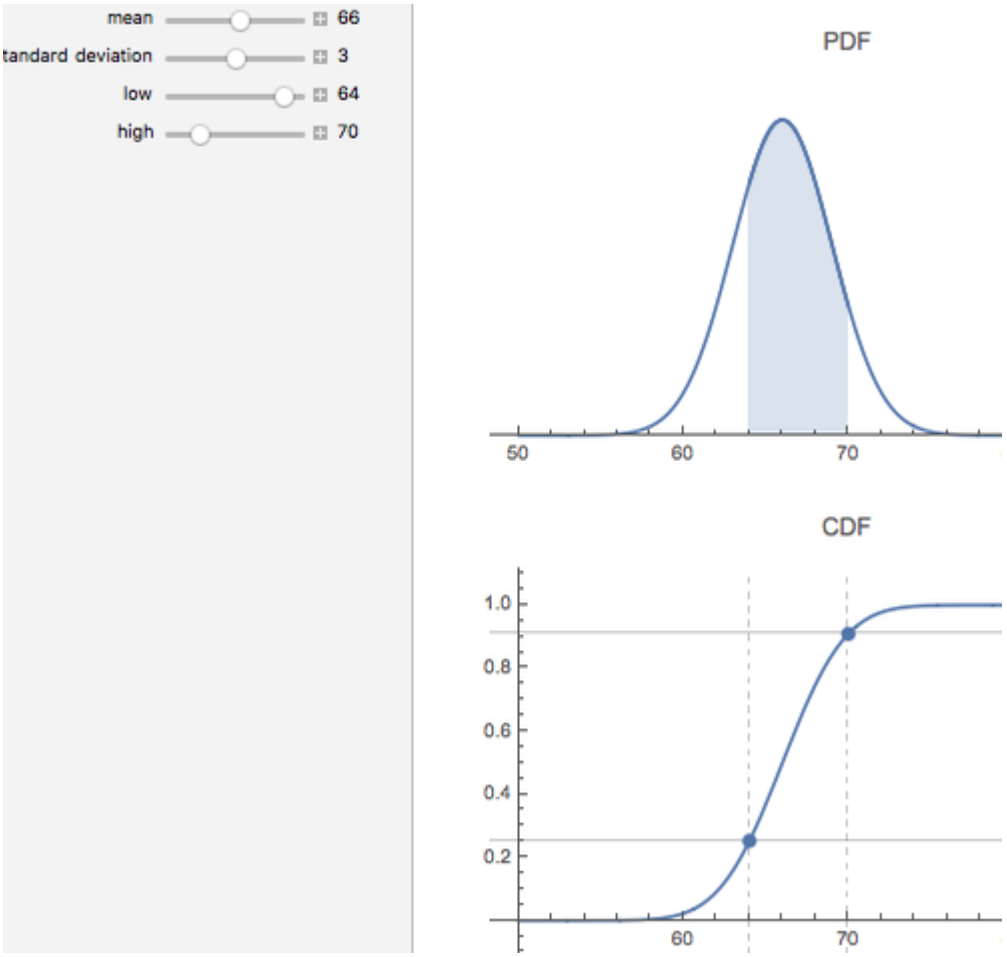
Region 3 value from the Z-score table is 0.0668 (6.68%). Now, we know that for a standard normal distribution, the curve is symmetrical around the mean, so region 2 is 50% of the overall value under the curve. Then, region 1 is (50 - 6.668)% i.e. 43.37% approximately. Then, the value of Region 1 and Region 2 combined would be around 94% approx.

IV. Probability Density Function:

Plotting all the points in the form of histograms shows on y-axis how many points are within a particular bin / range. If I want to convert it into a PDF or smoothen the histogram, it resembles a bell curve. As soon as we draw a bell curve, the count value gets replaced by % of the distribution on the y-axis.

It basically says that within this range, what is the % of the distribution present in the particular region. If that particular value is 0.2, which means 20% of the points distribution are in that particular region.

Cumulative density function (CDF) is different from PDF. In this , we basically add the percentage distribution for these points and the curve looks different from PDF.



Suppose, we take a point x(j) and we get 90% on the y-axis corresponding to it. This indicates that 90% of the distribution is less than ….x(j)…kgs. That also indicates that 10% of the distribution is greater than x(j) kgs.

V. Ridge and Lasso Regression

Regularization hypertuning techniques.
Sum of residuals in linear regression: $\sum(y' - y)^2$

Suppose there are two parameters: Experience and Salary. We try to create a best fit line using linear regression.

Suppose my training data has only two points which are far and fall almost on a line. We create a line and then calculate the sum of residuals to reduce the cost function. Now, in this case, for the training dataset, the points lie exactly on the line we made, the residuals would be zero. But, we wish to create a generalized model. Now, the test data might show high errors due to the huge distance from this line. So, this case is of overfitting.

For my training dataset, we have got a low bias / low error result. But, for test data, we see that the model is less generalized and shows high errors.

Now, we can use Ridge and Lasso regression to convert this high variance condition to a low variance condition.

How does Ridge and Lasso regression solve these problems?

Usually, in a linear regression model, Sum of residuals is given by $\sum(y' - y)^2$. But, in ridge regression, we add one more parameter. Cost function for ridge regression = $\sum(y' - y)^2 + \lambda*(slope)^2$.
So, if we see a steep curve in our linear regression model, we can deduce that it might lead to overfitting as with a unit increase in the experience, there is a quite high increase in the salary. In Ridge regression, we add $\lambda*(slope)^2$ to this model where we already see that the slope is quite high. We can assign 0 to any positive value to λ. Let us take it as 1 for now. Let us assume that the slope is 1.3 at present. If we do the calculation $\lambda*(slope)^2 = 1 *(1.3)(1.3) = 1.69$. Now, we will see the same value for other lines possible and see if they can reduce it.

Suppose residuals for this line might be a smaller value and slope might also be a smaller value as steepness has gone, thus, $\lambda*(slope)^2$ would also be a smaller number. Hence, cost reduces on the whole. So, we can use this line as the new best fit line.

We are basically penalizing features which have higher slopes to make the best fit line less steeper. If there is just one feature slope would be m. If we have 2 features, $\lambda*(slope)^2 = \lambda*(m1^2 + m2^2)$.

This will lead to less variance and better generalization. There would be slightly higher bias for the training dataset now but the test accuracy improves.

Cost function for Lasso regression = $\sum(y' - y)^2 + \lambda*|slope|$. It not only helps in regularization, but also helps in feature selection.

Suppose, we have $y = m1x1 + m2x2 + m3x3 + m4x4 + c1$. Now, as per Lasso regression, $|slope| = |m1 + m2 + m3 + m4|$. Wherever, the slope is very less, those features would get removed as they are not important / significant in making the prediction. This leads to feature selection as well. For Ridge regression, on the other hand, the $(slope)^2$ shrinks but never reduces to zero.

Curse of Dimensionality:

Dimensions, features, attributes are all terms used for features.

Suppose, for M1, I am taking 2 features; for M2 , I am taking 5 features;
For M3, I am taking 10 features; for M4 100 features;
For M5, I am taking 200 features; for M6 1000 features and for M7, 10000 features.

In M1, We will calculate Price of the house w.r.t Size of the house and no. of bedrooms.

In M2, additional features are state, bed size and area schools.
This model will definitely give better accuracy than the previous one as this provides better information to predict.

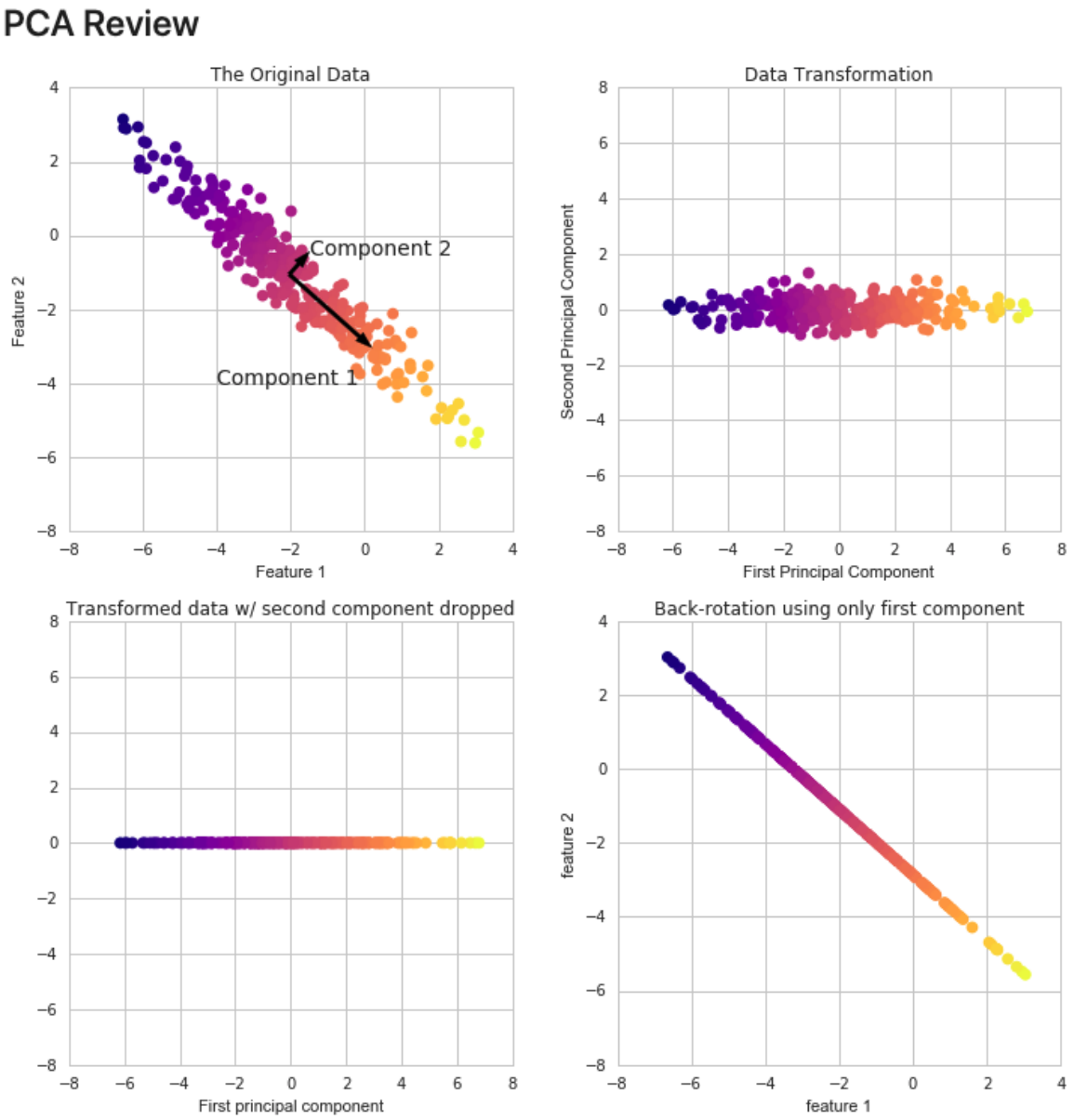
Now, in M3, we are taking 10 independent features. This model also gives greater accuracy than M2 and M1.

As we increase the number of features, our accuracy and performance is also increasing. But after a certain threshold value, our model will not give increased accuracy even by increasing the number of independent features / dimensions.

This is called the curse of dimensionality. With the increase of dimensions, it is not necessary that the accuracy improves after a certain threshold value. It is possible that this accuracy might decrease after a certain threshold. Until the threshold is achieved, the model learns new information from the independent datasets and features. With the exponential increase in the number of features, the model gets confused and the accuracy drops.

Then how to select the correct number of features / dimensions?
We use Chi square test, correlation coefficient etc. to determine this number of features.

Dimensionality Reduction - Principal Component Analysis



Suppose, I have features in a 2-dimensional space and we need to convert these features into 1-dimensional space.

Step 1: Consider the best vector space (1-d line) and project all the points to it. This line is called Principal Component 1 (PCA1).
Step 2: The next line (PCA2) would be drawn orthogonal to the PCA1 line. Now, when I try to project all the points to this line PCA2, there is a lot of information / variance that is lost during this process. So, we try to reduce this information loss. The reason we consider these two orthogonal lines is that they cause lesser information loss.

VI. Hypothesis Testing

P Values, T Test, Anova Test, Z Test, Type I and Type II Error

What exactly is hypothesis testing?

Any data in statistics is useful only if you analyze it and deduce conclusions or inferences. In hypothesis testing, we actually evaluate two or more exclusive statements on a population using a sample of data.

- ***Statement 1***: The person is guilty
- ***Statement 2***: The person is innocent

Steps of hypothesis testing:

a. Make an initial assumption (Ho - null hypothesis - The person is innocent)
b. Collect data / evidences
c. Gather evidence to reject or not reject the null hypothesis

Alternate hypothesis, H1 - Opposite of null hypothesis. If we are able to prove Ho, then we consider it as true else due to lack of evidence, we consider H1 as true.

Suppose if a null hypothesis is actually true (defendant is innocent) but I do not have enough evidence. Then Ho gets rejected and H1 would be considered as true (the defendant will be treated as guilty). If we look at the confusion matrix as follows:

	Ho	**H1**
Do not reject	Ok	***Type 2 Error***: we took H1 as true as per evidence, but actually Ho was true
Reject	***Type I Error***: Due to the lack of evidence, I have to reject Ho even though it might be true.	Ok

Suppose. Ho: the market is going to crash and H1: the market is not going to crash.
I collected various pieces of evidence and found that H1 is true but actually Ho was true : the market crashed. This is type II error.

If $P < 0.05$ (significance value), then we reject Ho and consider H1.

VII. P Value

Based upon the number of touches in the center of the spacebar key, we can make a normal bell curve. It is also called a 2-tailed test.

Suppose for a particular point, P value is 0.01. This means that if we repeat this experiment 100 times, then the number of times we will be touching the space key in this area will be 1. Suppose the P value at a point is 0.8, then if we repeat this experiment 100 times, then the number of times we will be touching the space key in this area will be 80 times.

P-value is the probability for the null hypothesis to be true.
Null hypothesis: It treats everything same or equal

Consider I have a fair coin. If I am performing a specific toss experiment 100 times, based on fair nature at least 50 times head or tail should be coming up.
Null hypothesis, Ho - the coin is fair
H1: the coin is not fair
Now, when we perform an experiment of toss (100 times). Out of that we achieve: 67 times head, 33 times tail. 50 would be the mean value of the bell curve and 67 would be on its right. It would be good to get this value nearer to the mean. We see that the P-value is falling in the tail region , away from the mean. SO, we need to reject the null hypothesis and we cannot say that it is treating all the same. The coin is not fair as Ho is rejected.

5% of data is usually divided in two parts under the tail region and the rest 95% is there within the two tails.

Now, suppose, P value is 0.05 (significance). If my experiment comes with a P value and it lies in the tail region, then we will reject Ho definitely because it is likely not possible to be true.

If suppose , we get 55% head in 100 tosses, we notice that P value does not come in the extreme tail regions, so we accept the null hypothesis and say that it is really a fair coin.

VIII. T Test, Chi Square test, ANOVA Test

Given the age, gender, weight and height data of males and females.

Null Hypothesis, Ho: There is no difference.
Alternate Hypothesis, H1: There is a difference between male and female proportion.
Test: {Considering Ho is true, where is the likelihood that H1 is true?}

a. ***One sample proportion test:*** (if we are considering just one-categorical feature such as gender). P - value needs to be selected before we start this test.
   If P value is less than 0.05, then we reject Ho. There is actually a difference between male and female ratio / proportion.
   P<= 0.05 → significance value alpha

b. ***Chi Square Test***: (if we are considering two categorical features; gender given the age group).

c. **T test**: (If we are considering One Continuous numerical variable e.g. height - mean height)

d. ***T test or Correlation Test***: (If we are considering Two continuous numerical variables e.g. weight and height). According to Pearson correlation, the value ranges from -1 to 1. The correlation near to 0 signifies that there is no relationship.

e. ***ANOVA Test***: If we are considering one categorical and one numerical variable and a categorical variable with more than two categories. If it has two categories only, then we apply the T test.

IX. Metrics in Classification Model Performance

a. Confusion matrix
b. FPR (False Positive Rate - Type I Error)
c. FNR (False Negative Rate - Type II Error)
d. Recall (TPR i.e. True Positive Rate, Sensitivity)
e. Precision (+ive pred val)
f. Accuracy
g. F Beta
h. Cohen Kappa
i. ROC Curve, AUC Score
j. PR Curve

Any classification problem can be solved either through Class Labels (A, B = 0.5) or Probabilities (We have to select this threshold probability very carefully).

Suppose in a problem statement of binary classification we have 1000 records, out of which 500 are Yes and 500 are No. OR 600 Yes and 400 No, then we can say that it is a balanced dataset. Even 700 Yes, 300 No → balanced dataset. My machine learning will not get biased based on the output (as more or less the dataset is balanced).

But if this ratio is 80: 20 (Yes: No), some of the machine learning algorithms will get biased based on the output.

If we have a balanced dataset, we use the metrics such as Accuracy. If we have an imbalance dataset, we use Precision, recall or F Beta score.

Confusion is a 2X2 matrix where the top values are the actual values and on the left hand side are my predicted values.

Actual values →	**1**	**0**
Predicted values → 1	True Positives	Type I Error (FPR): False Positives

0	Type II Error (FNR): False Negatives	True Negatives

Type I Error - FPR = FP / FP + TN
Type II Error - FNR = FN / FN + TN
Accuracy = TP + TN / TP + FP + TN + FN

What if my data is imbalanced? Suppose we have 900 Yes and 100 No. My model might predict everything belonging to the Yes category based upon output ratio. It basically is blindly suggesting that it belongs to this particular category. For this we use precision and recall.

True Positivity Rate / Sensitivity / Recall: TP/(TP + FN) - Out of the total actual positive values, how many did we predict correctly

Positive prediction value / Precision: TP/(TP + FP) - Out of the total predicted positive results, how many results were actually positive.

Use cases of Recall and Precision:

Spam detection - Precision (the mail is not a spam but it has been predicted as a spam - the customer is going to miss that email)

Cancer or not - Recall (the person is told that he does not have cancer but he actually has cancer)

Whenever, your false positive is much more important, then use precision.
Whenever, your false negative is much more important, then use recall.

F-Beta: Sometimes in an imbalanced dataset , both false positives and false negatives are important, then we use F-Beta score.

F Beta = $(1 + \beta^2)$ Precision * Recall / β^2 (Precision + Recall)

If β value is 1, then it becomes F1 - score
If β value is 0.5, then it becomes F0.5 - score
If β value is 2, then it becomes F2 - score

If β value is 1, then F1 Score: 2*Precision * Recall / (Precision + Recall): Harmonic mean

When do we select Beta as 1?
When FP and FN are equally important, then we select Beta as 1.

When FP is having more impact than FN, at that time we select Beta as 0.5. (Range: 0-1)
When FN is more impactful, at that time we select Beta as 2. (Range: 1-9)

Now, let us explore performance metrics where we have to decide a probability threshold:

Use case: Disease data in healthcare

For our model, we would decide some threshold value, [0, 0.2, 0.4, 0.6, 0.8, 1]
Suppose it is 0, then anything which has a predicted value greater than 0 will have a value of class label as 1. Then we calculate TPR and FPR. Here, TPR = 1, FPR = 1, Then we plot TPR versus FPR for the ROC Curve.

Now, when we connect all the points on the ROC curve, the area under this curve is called AUC score. The greater the area under the curve, the better the model is. This area should be more than the diagonal line connected in the graph. If it is less than that, it is a dumb model.

If we plot this curve and show it to a domain expert, he would tell us whether they require higher TPR, we can select one particular value of threshold where FPR is zero for higher TPR. If the domain expert says that we do not care about FPR but we just need the highest TPR, we can select the top most point in the curve.

X. Logistic Regression

Logistic regression is basically used for Binary classification. But why is it called regression? On X-axis if we take weights and on Y-axis we classify them as Obese or not obese. Can we solve this problem with the help of linear regression?

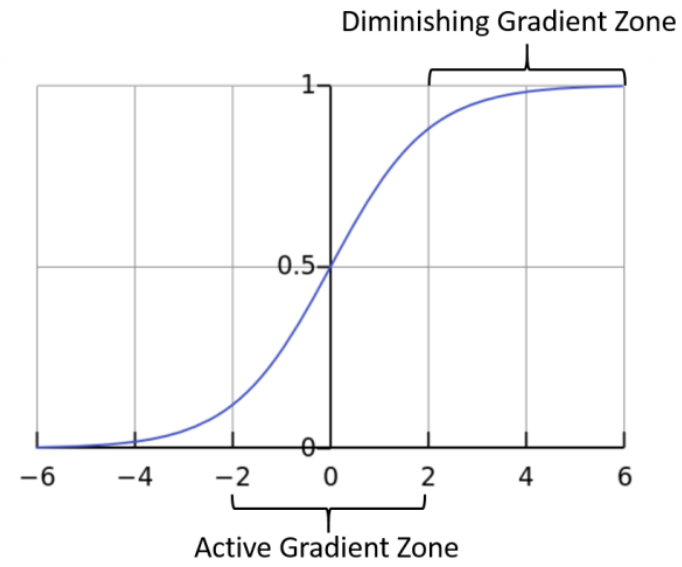
Now, linear regression states that the distance between predicted values and actual values should be minimal and predicted values are given by: $y' = mx + c$ or $h(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5$; here, β_0 is the y-intercept; β_1, β_2 and β_3 are changes in y with unit changes in x_1 , x_2 and x_3 respectively.

Now, suppose we consider the weight in our example of weights → 75; at this point, the output would be exactly in the middle (0 and 1) = 0.5. $h(x) >= 0.5$ {obese} and $h(x) < 0.5$ {Not obese}. With the help of this line, we are able to classify the points (weights).

Just by considering a straight line, we can solve the classification problem, then why do we need Logistic regression? To answer this, suppose, we just have one new outlier in our data, this changes our best fit line in linear regression and classification for all the points as the straight line gets deviated.

Because of this drawback of the need of creating a best fit line each time a datapoint is added, we use logistic regression for classification. Another drawback of linear regression, is the best fit line might give outputs greater than 1 or lesser than 0 as well. This problem is solved by using sigmoid function in logistic regression as it squashes the values greater than 1 or lesser than 0.

$$A = \frac{1}{1+e^{-x}}$$



Logistic Regression:

It is usually applied to those problems where the two classification points can be classified using linear regression. We need to find the exact coefficients of the best fit line but we do not use linear regression. Assumptions we make are:

- For positive points are denoted as + 1 and negative points are denoted as -1.
- 'Y-intercept - b' value is considered as 0. Y = w'x. The distance between any point x and the best fit plane is given as w'x + b / ||w||. If we consider w as a unit vector, ||w|| becomes 1 and b has been assumed as 0. Then we can say that the distance of a point x and a plane is w'x. Above the best fit plane, if we have a point, then its distance from the plane comes out as positive whereas below it comes as negative.
- Now, the distance of point x1 above the plane would be positive and would be w'x1.
- y(w'x) > 0 (Case 1)
- Now, consider y = w'x2 below the best fit line for x2 point. For it y would be negative. Then we can say: y(w'x) > 0 (Case 2) for this point as well.
- Case 3: Consider an outlier above the best fit line. Y = -1 for the outlier and w'x > 0. Thus, y(w'x) < 0 for the outlier. This point is incorrectly classified.

Our cost function should be maximized: max∑yw'x for i = 1 to n . Here, y and x are given and the only thing we need to find is the parameter - w_i.

What if we have outliers in logistic regression?

For a positive point lying in the negative region i.e. below the best fit line, we get a negative cost: yw'x Then, the best fit line deviates to make the cost positive. How do we rectify it?

For tacking it, we modify the cost function by applying sigmoid function on the earlier cost function.

Max∑f (yw'x) for i = 1 to n and f() is the sigmoid function. This sigmoid function makes sure that the entire value is transformed between 0 and 1. By doing that, it is removing the effect of outlier.

How can we solve a multiclass classification problem using Logistic Regression?

Logistic Regression (One vs. Rest): First M1 model is created segregating one category from the rest of the categories. Next, iteration, model M2 is created, treating one category and the rest as another. Similarly, in the next iteration, another model M3 is created.

Suppose, the three categories we have are O1, O2 and O3. In the first iteration, O1 might be + 1 and O2, O3 might be -1. Similarly, in the next iteration, O1, O2 might be + 1 and O3 might be -1. This way we combine values of O1 to create a model M1; values of O2 to create a model O2 and values of O3 to create a model M3. If I give a new test data, then the whole feature will go to M1 model, then M1 model will give some probability. Then, M2 will give some probability when a feature is given to it. And, then M3 gives another probability. Now, the model that gives the highest probability, say M3 , then that point belongs to the O3 category. While coding, there is a multi-class parameter which needs to be set as ‘ovr’ to implement multiclass Logistic Regression.

XI. Cross-validation Types

When we do a train-test-split with 30% split, 70% of the random dataset is selected. There is a random state variable. There would be certain data points which were important but were not taken. This might lead to reduced accuracy. Suppose, we change the random state and again do train-test-split, our accuracy fluctuates. This way our accuracy changes as the random state changes. This shows fluctuating output.

To handle this problem, we use cross validation. Different types of cross-validation:

a. ***Leave One Out CV (LOOCV)***: Suppose I have 1000 records, the LOOCV rule is that from all these records, I will take one dataset at a time as my test and remaining all would be my training data points. Based on this model will be trained on all the training data and would be tested on that one dataset. This way the next experiment is repeated with different test samples. I have to perform many iterations to make the model ready for generalization. It will lead to low bias. For the training dataset and one test sample, we would get quite good results but very poor results on unseen data points.
b. ***K-Fold CV***: Suppose my dataset is 1000 records, we select a K value. This K basically means K experiments. For each experiment, it will decide the test data. For example we can take a K value of 5 and a split of 20% for train and test data. Find out the mean accuracy of all K experiments. Sometimes, one can communicate the minimum and maximum accuracy of the model as well to the stakeholders. One of the drawbacks of K Fold accuracy is that 205 split might lead to an imbalance dataset wherein the test data might have only one type of points and training data might have another.
c. ***Stratified K-Fold CV***: Suppose, K = 5, it is made sure that every time test data is selected, the number of instances of each class is taken in a proper way so that the data does not get imbalanced. Please note that if the entire dataset is imbalanced, we need to handle it first at the time of feature engineering.
d. ***Time series CV***: Suppose we have data for Day 1to Day 7 and we need to predict some results for Day 6 and 7. We need to rely on the previous days to predict for the

desired days. For Day 6, suppose my inputs would be Day 1 to Day 5, for Day 7, my inputs would be Day 2 to Day 6. For Day 8, the inputs would be Day 3 to Day 7. That is how time series CV works.

XII. Naive Bayes Classifier

Theoretical Understanding:

Independent Events - Suppose, if i am tossing one coin, then the probability of heads and tails are ½ and ½ respectively. Now, if we toss these coins again. The probability of heads or tails in the second toss is not dependent on the outcome of the first toss. Thus, this is an example of independent events.

Dependent Events - Suppose, in a bag I have marbles (3R 2B). Suppose, we pick up 1 blue marble, the probability of getting a blue marble is ⅗. Now in the next event if i want to pick up one marble out of the remaining 4 marbles. Now the probability of the second marble being blue is dependent on the outcome of the first marble. Thus, this is an example of dependent events.

Conditional Probability - P(A|B) = P (A∩B)/P(B) Suppose, in event A, I pick up a blue marble, the probability would be ⅗. Now, suppose, in event B, I take out second blue marble given A event is already performed i.e. P(B|A) = ¼ as there is just one blue marble remaining. When we are considering P(B|A), if we multiply P(A) and P(B|A), we get 1/10. This is equal to P (A∩B). This is actually our conditional probability formula only.

Bayes theorem: P(A|B) = P (A∩B)/P(B) and P(B) = P (B∩A)/P(A). Now, P (A∩B) = P (B∩A).

Thus, P(A|B) * P(B) = P(B|A) * P(A)

This implies that P(A|B) = (P(B|A) * P(A)) / P(B)

Here, P(A|B) is the posterior probability. P(B|A) is the likelihood. P(A) is the prior probability. P(B) is the marginal probability.

How is the Bayes theorem used in the Naive Bayes ALgorithm?

Suppose I have x = {x1, x2, x3, x4,.....xn} features and output as {y}. Let us consider that it is a classification problem.

P(y|x1,x2,x3,x4…..xn) = P(x1|y)P(x2|y)P(x3|y).....P(xn|y) * P(y) / P(x1) P(x2) P(x3)....P(xn)

P(y|x1,x2,x3,x4…..xn) = P(y) $\prod$P(xi/y) / P(x1) P(x2) P(x3)....P(xn)

for i ranging from 1 to n

P(y|x1,x2,x3,x4…..xn) $\propto$ P(y) $\prod$P(xi/y) for i ranging from 1 to n

y = ARGMAX(P(y) $\prod$P(xi/y)) for i ranging from 1 to n

Practical Example:

Outlook feature:

	Yes	No	P(Y)	P(N)
Sunny	2	3	2/9	3/5
Overcast	4	0	4/9	0
Rainy	3	2	3/9	2/5
Total	9	5	100%	100%

Temperature feature:

	Yes	No	P(Y)	P(N)
Hot	2	2	2/9	2/5
Mild	4	2	4/9	2/5
Cold	3	1	3/9	1/5
Total	9	5	100%	100%

Play Outcome:

		P(Y) & P(N)
Yes	9	9/14
No	5	5/14
Total	14	

Now, suppose for today the day is sunny and hot:

P(Yes | Today) = P(Sunny | Yes) * P(Hot | Yes) * P(Yes) / P(Today)

P(Yes | Today) = 2/9 * 2/9 *9/14 = 2/63 = 0.031

P(No | Today) = ⅗ * ⅖ * 5/14 = 3/35 = 0.08571

P(Yes) = 0.031 / 0.031 + 0.08571 (normalizing it) = 0.27 approx.

P(No) = 1 - 0.27 = 0.73

I.e. the output would be No as it has higher probability

Another practical example:

NLP: *Judge whether the statement review is good or bad*

f1	f2	f3	f4	o/p
The	food	Delicious	Bad	
1	1	1	0	1
1	1	0	1	0
0	1	0	1	0
0	1	1	0	1
0	0	0	1	0

Sentence 1: The food is delicious

Sentence 2: The food is bad

Sentence 3: Food is bad

After using Stop words, stemming, Bag of words, TFIDF → we utilize the final features.

P(y = Yes | Sentences) = P(y = Yes | x1, x2, x3,....xn) $\propto$ P(y) ΠP(xi/y) for i ranging from 1 to n

X1 is nothing but the words in the sentences after pre-processing.

P(y = Yes) = ⅖ (from o/p column)

P(x1/y = Yes) = 1/2 (where The is present and o/p is 1)

P(x2/y = yes) = 2/4 (where food is present and o/p is 1)

P(x3/y = yes) = 2/2 (where delicious is present and o/p is 1)

P(y = Yes | Sentences) = P(y = Yes) * ΠP(xi/y) for i ranging from 1 to n = ⅖ * ½ * 2/4 * 2/2 = 1/10 = 0.01

Similarly P(y = No | Sentences) = ⅗ * ½ * ⅔ * 3/3 = ⅕ = 0.2

Thus, normalizing , P(y = Yes) = 0.01 / 0/01 + 0.02 = 0.33

P(No) ~ 0.67 , we get that finally the output is No i.e 0 for the max inputs

Where does this fail?

Suppose in a new sentence: The food is tasty. Tasty is a word which has not been seen before. By default, it would be treated as a negative scenario. Also, it might fail when there is an imbalanced dataset. We would see how to rectify this in the NLP section.

1. What Are the Basic Assumptions?

Features Are Independent

2. Advantages

1. Work Very well with many number of features

   Explanation: Because of the class independence assumption, naive Bayes classifiers can quickly learn to use high dimensional features with limited training data compared to more sophisticated methods. This can be useful in situations where the dataset is small compared to the number of features, such as images or texts. Naive Bayes implicitly treats all features as being independent of one another, and therefore the sorts of curse-of-dimensionality problems which typically rear their head when dealing with high-dimensional data do not apply.

   If your data has k dimensions, then a fully general ML algorithm which attempts to learn all possible correlations between these features has to deal with 2^k possible feature interactions, and therefore needs on the order of 2^k many data points to be performant. However because Naive Bayes assumes independence between features, it only needs on the order of k many data points, exponentially fewer.

   However this comes at the cost of only being able to capture much simpler mappings between the input variables and the output class, and as such Naive Bayes could never compete with something like a large neural network trained on a large dataset when it comes to tasks like image recognition, although it might perform better on *very* small datasets.

2. Works Well with Large training Dataset
3. It converges faster when we are training the model

   Explanation: Unlike other machine learning models, naive bayes require little to no training. When trying to make a prediction that involves multiple features, we simply use the math by making the *naive* assumption that the features are independent.

4. It also performs well with categorical features

   Explanation: For a Naive Bayes classifier, categorical values are the easiest to deal with. All you are really after is P(Feature | Class). This should be easy for the days of the week. Compute P(Monday | Class=Yes) and so on.

3. Disadvantages

1. Correlated features affects performance

4. Whether Feature Scaling is required?

No. *In fact, any Algorithm which is NOT distance based, is not affected by Feature Scaling.* As Naive Bayes algorithm is based on probability not on distance, so it doesn't require feature scaling.

5. Impact of Missing Values?

Naive Bayes can handle missing data. Attributes are handled separately by the algorithm at both model construction time and prediction time. As such, if a data instance has a missing value for an attribute, it can be ignored while preparing the model, and ignored when a probability is calculated for a class value tutorial : https://www.youtube.com/watch?v=EqjyLfpv5oA

6. Impact of outliers?

It is usually robust to outliers.

One potential issue with outliers is that unseen observations can lead to 0 probabilities. And we know in naive bayes we multiply probab of words lying in that particular class and results zero. For example,

Bernoulli Naive Bayes applied to word features will always produce 0 probabilities when it encounters a word that wasn't seen in the training data. Outliers in this sense can be a problem.

However, all these and similar issues of Naive Bayes have well-known solutions (like Laplace smoothing, i.e. adding an artificial count for every word) and are routinely implemented. In Gaussian Naive Bayes, outliers will affect the shape of the Gaussian distribution and have the usual effects on the mean etc. So depending on your use case, it still makes sense to remove outliers.

Different Problem statement you can solve using Naive Bayes

1. Sentiment Analysis
2. Spam classification
3. twitter sentiment analysis
4. document categorization

XIII. Linear Regression Algorithm

Theoretical Concepts:

y=mx + c; m = slope, c = intercept

We need to find the best fit line. If my x is 0, then y is c i.e. Y-intercept. Within a unit change in X-axis, the change in the y-value is called slope or m. m = (y2 - y1) / (x2 - x1)

The summation of squared error should be minimized by the requisite values of m and c obtained. Here, cost function = distance b/w the best fit point and the actual point should be minimum. Thus, cost function = $1/2n \sum(y' - y)^2$ where i ranges from 1 to m. Here, m is the number of points.

Predicted points: y' and actual points: y. But, if we just use the above equation, we might get several best fit lines, and choosing just one might be time consuming.

So, what can we do?

Example: y = x; y' = mx + c; here, we can assume the best fit line passes through origin and c = 0; thus y' = mx

Now, we can substitute x = 1, and m = 1; then y' = 1;

For x = 2, y' = 2 and y = 2 and so on. This is actually my best fit line when my slope is 1. After getting this equation, we calculate our cost function and try to reduce it.

Here, cost function, $J(m) = 1/2n((1-1)^2 + (2-2)^2 + (3-3)^2) = 0$

With respect to every m value, we can plot our cost function. For m = 1, J(m) = 0.

For m = 0.5, x = 1; y = 0.5,

For x= 2, y = 1,

For x = 3, y = 1.5

Then our cost function, $J(m) = 1/2n((1-0.5)^2 + (2-1)^2 + (3-1.5)^2) = 0.58$. Thus, a curvature of J(m) versus m can be plotted and we can see the gradient descent. How do we arrive at the global minimum?

Based on some m value, we get some initial J(m) value. In order to move downwards, we should use the convergence theorem.

Convergence theorem: $m = m - \alpha \partial J / \partial m$ where α is the learning rate

If the slope ($\partial J / \partial m$) is negative, the curve points downwards. When we get a negative slope, then m = m + (+ive smaller value). This step would be very very small and it would move slowly towards global minimum. If we take a larger alpha, then the jumps might be bigger and it might not converge after several iterations and keep oscillating.

At the global minima, the slope will be zero. This would be the slope of the best fit line. Until then, we would keep following the convergence theorem.

If I have multiple independent features, then each of the features will try to reach a global minimum.

For multiple linear regression, $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5$; here, β_0 is the y-intercept; β_1, β_2 and β_3 are changes in y with unit changes in x_1 , x_2 and x_3 respectively.

Multicollinearity: In regression, "multicollinearity" refers to predictors that are correlated with other predictors. Multicollinearity occurs when your model includes multiple factors that are correlated not just to your response variable, but also to each other. In other words, it results when you have factors that are a bit redundant. To handle such a multicollinearity situation, one solution is to remove highly correlated feature (check for P value in model summary and remove the one for which it is higher).

R Square and Adjusted R Square:

$R^2 = 1 - SSres/SStot$

Here, $SSres$ is the sum of squares of residuals or errors = $\sum(y' - y)^2$

$SStot$ is the sum of average totals = $\sum(y' - y_{mean})^2$

The closer the value of R^2 to 1, the better the model is.

It will be less than zero only when the best fit line is worse than the average value.

As we go on adding new independent features, our R^2 value usually increases. This is because the model then tries to apply some coefficient value such that our $SSres$ value decreases. Then, resultantly, our R^2 value increases. We should also note that this value will never decrease when we keep on adding independent features to it. But, there might be a scenario that the independent feature being added might not be correlated to the target output. Even though R^2 value might show an increase in such a case, but it is basically not penalizing the newly added features which do not have any correlation with the target. For this reason, we use adjusted R square.

$$Adjusted\ R^2 = 1 - \frac{(1 - R^2)(N - 1)}{N - p - 1}$$

Where

R^2 Sample R-Squared

N Total Sample Size

p Number of independent variable

From the above formula, we can note that as the value of p increases when independent features which are not correlated to the target variable are added, N - p - 1 value decreases, leading to an overall decrease in the Adjusted R square value as a higher term is subtracted from 1 to achieve it. In case, the added features have correlation with the target variable, then R^2 value would be higher such that the rest of the multiplication factor will become overwhelmed and would not make much of a difference to the overall R^2 value.

Differences between R^2 value and Adjusted R^2 value:

Every time an independent variable is added to a model, R^2 value increases, even if the independent variable is insignificant. It never declines, whereas adjusted R^2 value increases only when the independent variable is significant and affects the dependent variable.

Adjusted R^2 value is always less than or equal to R^2 value.

Removing the highly correlated feature would work when we have quite fewer numbers of features and a smaller dataset. We also lose certain information when we remove some data. In case, we have a large dataset, we would go for Ridge or Lasso correlation.

1. What Are the Basic Assumptions?(favorite)

There are four assumptions associated with a linear regression model:

1. ***Linearity***: The relationship between X and the mean of Y is linear.
2. ***Homoscedasticity***: The variance of residual is the same for any value of X.
3. ***Independence***: Observations are independent of each other.
4. ***Normality***: For any fixed value of X, Y is normally distributed.

2. Advantages

1. Linear regression performs exceptionally well for linearly separable data
2. Easy to implement and train the model
3. It can handle overfitting using dimensionality reduction techniques and cross validation and regularization

3. Disadvantages

1. Sometimes Lot of Feature Engineering Is required
2. If the independent features are correlated it may affect performance
3. It is often quite prone to noise and overfitting

4. Whether Feature Scaling is required?

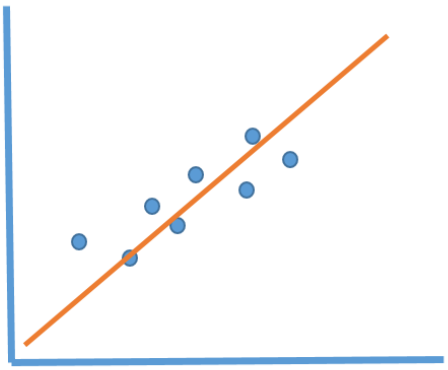
Yes (Whenever we talk about gradient descent, we need to do feature scaling because that will help us to reach the optimum solution / global minima very quickly.)

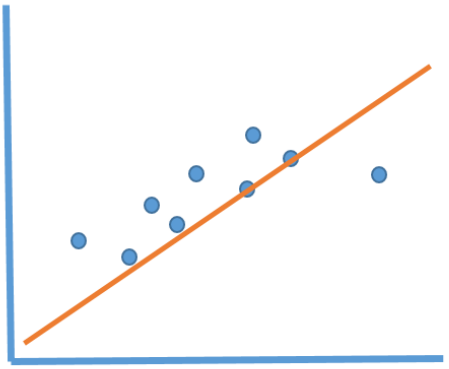
5. Impact of Missing Values?

It is sensitive to missing values (feature engineering is used to handle it)

6. Impact of outliers?

Linear regression needs the relationship between the independent and dependent variables to be linear. It is also important to check for outliers since linear regression is sensitive to outlier effects. Just to reduce the mean squared error (MSE), the line is changing in the below figure with the appearance of an outlier. To handle it, ridge and lasso regression are used.





Types of Problems it can solve(Supervised)

1. Regression

Overfitting And Underfitting

We will use polynomial linear regression to explain bias and variance tradeoff as well as overfitting and underfitting. Please note that if the degree of the polynomial is one, then the curve is a straight line. The sum of mean square error (cost function) is higher in that scenario when compared to a polynomial curve with degree greater than one. When error is very high for a training dataset, then the particular scenario is called *underfitting*. Suppose, we use a quite higher order polynomial such that almost all the training data points are fitted quite well by the model, this scenario is called *overfitting*. This is due to the fact that the accuracy would go down for the test data even though it is very high for the training data.

Our main aim is to achieve an optimum solution such that accuracy is high for both training and test data. That model will give us low bias and low variance.

In an underfitting scenario, we basically have high bias (error of the training data) and high variance (generalizability - error on the test data). For overfitting scenario, we have low bias but high variance.

Different Problem statement you can solve using Linear Regression

1. Advance House Price Prediction
2. Flight Price Prediction

XIV. Support vector Machines (SVM)

Theoretical Concepts:

The main aim of SVM is to create a hyperplane and two separating margin lines such that there are two more hyperplanes in parallel to the particular hyperplane and these hyperplanes pass through the nearest neighboring point in the two clusters.

This whole distance between the r=two parallel hyperplanes - d+ive and d-ive is called margin. We aim to get a generalized model so that it is easily applicable on unseen points. This margin and hyperplane helps in such generalization.

Now, we can create multiple hyperplanes separating the two clusters apart from this one hyperplane. Then, why choose this hyperplane? Our main aim is to select the hyperplane which gives us the maximum marginal distance. This linear hyperplane is for linearly separable points.

What if the points are non-linearly separable?
A linearly separable hyperplane cannot be made for such a dataset. For tackling this we use kernels which convert 2 -dimension to a higher dimension. Then, between the higher dimensional dataset, we can construct linearly separable hyperplanes

What is a support vector?
The nearest positive point and nearest negative point that passes through the parallel marginal hyperplanes are called support vectors. It may have multiple support vectors.

SVM Math Intuition:

Let us consider a simple example of logistic regression where we have two points (4,4) and (-4,0) in a 2-D plane. We wish to separate them using a linear hyperplane. Now, suppose the slope of such a linear plane is -1. The equation of this line is given as
$w'x + b = 0 \, where \, ' \, denotes \, the \, transpose.$
We also know this equation: y = mx + c; m = -1 (assumption made).
c = b = 0 as the line has been assumed to be passing through origin.Thus, y = $w'x$
$y = [-1\ 0]'[-4\ 0] = 4 \, (this \, value \, is \, always \, going \, to \, be \, positive)$

Anytime, we calculate y for points below the linear separator, y is going to be positive. And, when we calculate it for points above the linear separator, y is going to be negative. Now, we can take this value as + 1 and -1 for two clusters respectively.

In SVM, this hyperplane equation can also be given similar to logistic regression:
$w'x + b = 0.$
I am going to find the nearest points for the hyperplane in the two clusters. Suppose, it's at a distance -1 and + 1 respectively as assumed above.
So, the equations can be $w'x + b = -1 \, and \, w'x + b = 1$ respectively.

Now, here b will not be zero as the hyperplane is not passing through zero. Now, we basically wish to compute the distance between two points x1 and x2 lying on these two parallel hyperplanes. We can write the equations as follows:
$w'x1 + b = -1 \, and \, w'x2 + b = 1$. So, we can subtract the two equations:
$w'(x2 - x1) = 2.$
To remove w' , we are going to divide by ||w|| on both sides. So, 2/||w|| would be our optimization function and we need to maximize this.
We need to update (w,b) to maximize the optimization function such that y = 1 when $w'x + b$ >= 1 and y = -1 when $w'x + b$ <= -1.

We can also write it as: y*$(w'x + b)$ >= 1

If it is not greater than equal to 1, then it means that it is a misclassification.

One point to note is that in a real world scenario, we will not have such separable data points. There would be a lot of overlap.

We have to change (w*, b*) such that min(||w||/2) + c∑ζi where i ranges from 1 to n. 'c' represents how many errors the model can consider; **ζ** is the value of the error. This value 'c' is called regularization and is obtained by hyperparameter tuning. This particular SVM is called linear SVM and the margin is a ***hard margin***.
(w*, b*) = min(||w||/2) + c∑ζi where i ranges from 1 to n

SVM Kernels:

1. Polynomial Kernels
2. RBF Kernels
3. Sigmoid Kernels

In case of soft margin, there would be certain consideration for error. Now, non-linearly separable data can be handled using the SVM kernel which will convert a lower dimensional non-linearly separable dataset into a linearly separable higher dimensional dataset. This transformation is done by SVM Kernel from lower dimension to higher dimension. It does it using some mathematical formulas.

Let us consider one-dimensional points for example. —-------*****-----****-------******------
To classify these points, in order to use SVM, we first need to draw a hyperplane but they are 1-dimensional. So, we would use a kernel to make this dataset 2-dimensional. Let us suppose, we are using a Polynomial kernel (y = f(x) = x²). So, these points get projected as a parabola. Then, we can draw a hyperplane to separate these points.

Polynomial Kernel: Let us consider we have elliptical data points which are non-linearly separable in x1 and x2 plane. We will use a polynomial kernel to make the data points linearly separable in a higher dimension.

The formula for such a polynomial kernel would be: f(x1, x2) = (x1'.x2 + 1)^d where d is the higher order dimension.

The unique elements in x1 and x2 multiplication would be x1² and x2² whereas the repetitive parts would be x1x2. So, we see that our 2-dimensional points are becoming higher-dimensional - x1, x2, x1² , x2², x1x2 (axes: x1, x2 and x1x2) such that we can obtain linearly separable data points and hyperplane.

1. What Are the Basic Assumptions?
There are no such assumptions

2. Advantages

1. SVM is more effective in high dimensional spaces. : the reason that SVMs work well with high-dimensional data is that **they are automatically regularized**, and regularization is a way to prevent overfitting with high-dimensional data.
2. SVM is relatively memory efficient. (It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.)
3. SVM's are very good when we have no idea on the data.
4. Works well with even unstructured and semi structured data like text, Images and trees.
5. The kernel trick is the real strength of SVM. With an appropriate kernel function, we can solve any complex problem.
6. SVM models have generalization in practice, the risk of overfitting is less in SVM.

3. Disadvantages

1. More Training Time is required for larger dataset
2. It is difficult to choose a good kernel function https://www.youtube.com/watch?v=mTyT-oHoivA
3. The SVM hyperparameters are Cost -C and gamma. It is not that easy to fine-tune these hyper-parameters. It is hard to visualize their impact

4. Whether Feature Scaling is required?

Yes - Feature scaling is crucial for some machine learning algorithms, which **consider distances between observations because the distance between two observations differs for non-scaled and scaled cases**. As we've already stated, the decision boundary maximizes the distance to the nearest data points from different classes.

5. Impact of Missing Values?

Although SVMs are an attractive option when constructing a classifier, SVMs do not easily accommodate missing covariate information. Similar to other prediction and classification methods, in-attention to missing data when constructing an SVM can impact the accuracy and utility of the resulting classifier.

6. Impact of outliers?

It is usually sensitive to outliers. The penalty on misclassification is defined by a convex loss called the hinge loss, and the unboundedness of the convex loss causes sensitivity to outliers.

Types of Problems it can solve(Supervised)

1. Classification
2. Regression

Overfitting And Underfitting

In SVM, to avoid overfitting, we choose a Soft Margin, instead of a Hard one i.e. we let some data points enter our margin intentionally (but we still penalize it) so that our classifier don't overfit on our training sample

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Different Problem statement you can solve using SVM

1. We can use SVM with every ANN use cases
2. Intrusion Detection
3. Handwriting Recognition

Practical Implementation

1. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
2. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

Performance Metrics

Classification

1. Confusion Matrix
2. Precision,Recall, F1 score

Regression

1. R2,Adjusted R2
2. MSE,RMSE,MAE

Which all algorithms are impacted by an imbalance dataset?

Machine Learning Algorithms such as Logistic Regression, KNN,SVM (basically which includes Gradient Descent and Euclidean Distance Computation) are affected by Imbalanced Datasets.

XV. Decision Tree Classifier And Regressor

Interview Questions:

1. Decision Tree
2. Entropy, Information Gain, Gini Impurity
3. Decision Tree Working For Categorical and Numerical Features
4. What are the scenarios where Decision Tree works well
5. Decision Tree ***Low Bias And High Variance***- Overfitting
6. Hyperparameter Techniques
7. Library used for constructing decision tree
8. Impact of Outliers Of Decision Tree
9. Impact of missing values on Decision Tree
10. Does Decision Tree require Feature Scaling

Theoretical Understanding:

1. ***Entropy***: f1 , f2, f3 etc. features; o/p is yes or no

   ID3 Algorithm → which node to select for split up

Here, is when entropy comes into picture to select the right attribute for splitting purposes.

Entropy helps us to measure the purity of the split.

Suppose we split f1. Initially, we had 9 yes and 5 No's. After the split, at f2 we had 3 yes, 2 No and at f3 we had 6 yes / 3 No. Ideally, after the split we should have all the Yes on one side and all the No on another.

Again the split happens, this time we get 3 Yes, 0 No ; 2 No and 0 Yes on one side. We get 6 Yes, 0 No; 3 No, 0 Yes on the other side after the split. In order to get the correct leaf nodes fast, we need to select the right parameters / features of the dataset for split.

We have to go and calculate the purity split at each split.

Entropy: $H(S) = -P_{+}\log_2(P_{+}) - P_{-}\log_2(P_{-})$

P_{+}/P_{-} : % of +ive cases / % of -ive cases

S: subset of training sample

Entropy for above example:

$H(S) = -3/5\log_2(⅗) -2/5\log_2(2/5)$

H(S) = 0.78 bits APPROX.

Note: If we have a completely impure subset (3 Yes, 3 No), then its entropy is 1 bit. And, for a pure split (4 yes, 0 no) , entropy is 0 bits.

So, we would check for entropy at f2 and f3 for above example and select the one for which entropy is better. But, this calculation is just for one node. We should also consider other attributes which are needed for reaching pure leaf nodes. For that, we use Information gain. Basically, it calculates total entropy value from top to the leaf node to decide which path is better. Entropy ranges between 0 and 1.

2. ***Information Gain***: In short, we can say that the average of all the entropy is based on a specific split. Suppose, based on f2 we wish to split into f1 and f3; or based on f1 into f2 and f3. We need to decide which split would be better to reach leaf nodes earlier. For that we need information gain.

   $Gain(S,A) = H(S) - \sum|S_v| / |S| (H(S_v))$ where v belongs to val

   Here, S_v is the subset after the split, S is the total subset, H(S) is the entropy; $H(S_v)$ is the entropy after the splitting for the subset

   Suppose, f1 (9Y, 5N) is splitting into f2 and f3. For f2, we get (6Y, 2N) and for f3 (3Y, 3N).

   Here, H(f1) = H(S) i.e. entropy before the split = 0.94

   $H(S_v)$ is the entropy after the split

   H(f2) = 0.81 ; H(f3) = 1

Thus, gain = 0.91 - (8/14 * 0.81) - (6/14 * 1) = 0.049

Instead, I can also go from f2 splitting into f1 and f3. So, we will check which one gives a higher information gain and we will select that particular way of splitting.

3. ***Gini Impurity***: Both entropy and gini impurity do the same task which is to calculate the purity of the split. But, in most cases Gini impurity is preferred. Let us see why?

   This is computationally more efficient and takes a lesser amount of time. For entropy, logarithmic calculations take some amount of time whereas in GI , we do not have any logarithmic calculations.

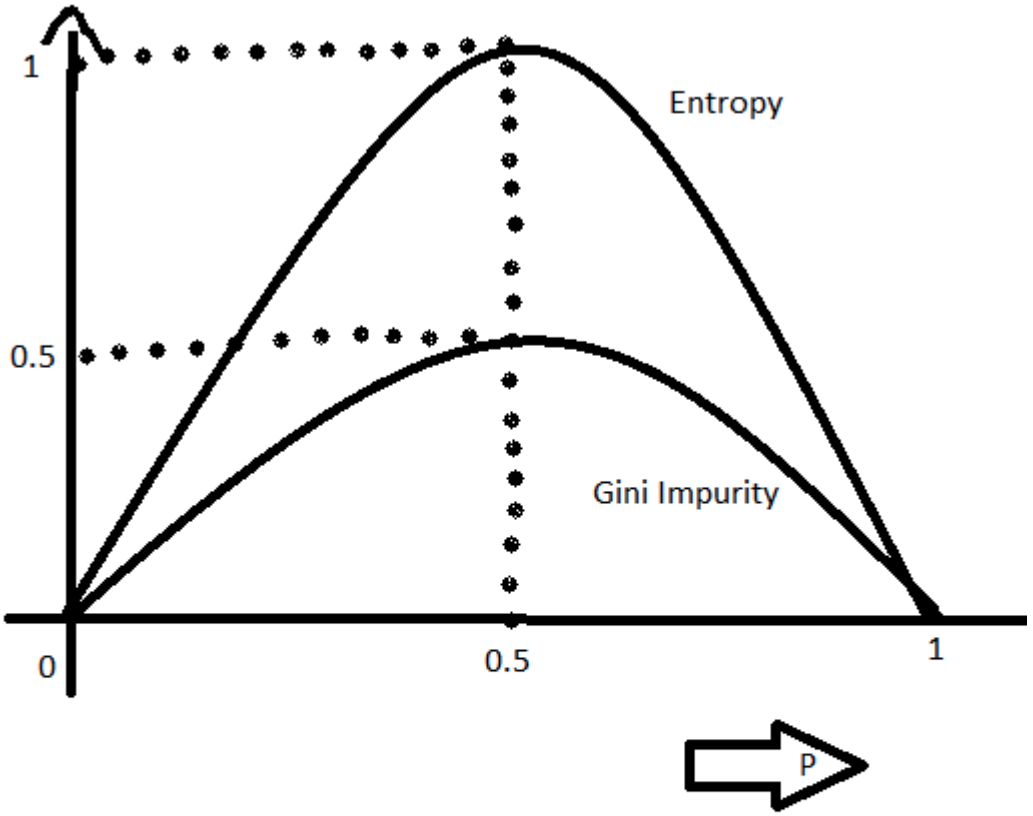
   F1 (6Y/3N) → f2 (3Y/3N) and f3 (3Y/0N)

   Entropy → 0.5 → 1 and 0 respectively

   Now, Gini Impurity, GI = 1 - $\sum(P^2)$ where i ranges from 1 to n

   Where P^2 is the summation of squares of P_+ and P_-

   GI = 1 - [(3/6)^2 + (3/6)^2] = 1 - [0.25 + 0.25] = 0.5 , whereas entropy for f2 was 1. So, we are getting less GI when compared to entropy.



$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

4. If your feature is a numerical variable, the first step decision tree algorithm does is sorting all the values. It will try to consider some threshold values and is going to check for each and every record for a particular feature. If that xi is less than or equal the threshold, it will create a branch for it

   In the given example, f1 (4yes, 4 no) , threshold is 2.3; and it is split; on one side there is just 1 Yes. On the other side (>2.3) we have 3Yes / 4No. The next threshold would be considered the next value, say 3.6. Again the split will happen for <= 3.6 (2Yes / 0No) and > 3.6 (1Yes/4 No). For this entropy and information gain is calculated and then decided based on which has the better value and that split is taken.

   There is a disadvantage here. Suppose, we have millions of records. The time complexity would keep on increasing with so many samples. Decision trees with numerical features will take more time for training. The same disadvantage is true for ensemble methods.

1. What Are the Basic Assumptions?

There are no such assumptions

2. Advantages

Advantages of Decision Tree

1. ***Clear Visualization*:** The algorithm is simple to understand, interpret and visualize as the idea is mostly used in our daily lives. Output of a Decision Tree can be easily interpreted by humans.
2. ***Simple and easy to understand***: Decision Tree looks like simple if-else statements which are very easy to understand.
3. Decision Tree can be used for ***both classification and regression problems.***
4. The Decision Tree can handle ***both continuous and categorical variables***.
5. ***No feature scaling required***: No feature scaling (standardization and normalization) required in case of Decision Tree as it ***uses rule based approach instead of distance calculation.***
6. ***Handles non-linear parameters*** efficiently: Non linear parameters don't affect the performance of a Decision Tree unlike curve based algorithms. So, if there is high non-linearity between the independent variables, Decision Trees may outperform as compared to other curve based algorithms.
7. Decision Tree can ***automatically handle missing values***.
8. The Decision Tree is usually ***robust to outliers*** and can handle them automatically.
9. ***Less Training Period***: Training period is less as compared to Random Forest because it generates only one tree unlike forest of trees in the Random Forest.

3. Disadvantages

Disadvantages of Decision Tree

1. ***Overfitting***: This is the main problem of the Decision Tree. It generally leads to overfitting of the data which ultimately leads to ***wrong predictions***. In order to fit the data (even noisy data), it keeps generating new nodes and ultimately the ***tree becomes too complex to interpret.*** In this way, it ***loses its generalization capabilities***. It performs very well on the trained data but starts making a lot of mistakes on the unseen data.
2. ***High variance***: As mentioned in point 1, Decision Tree generally leads to the overfitting of data. Due to the overfitting, there are very high chances of ***high variance*** in the output which leads to many errors in the final estimation and shows high inaccuracy in the results. In order to achieve zero bias (overfitting), it leads to high variance.
3. ***Unstable***: Adding a new data point can lead to re-generation of the overall tree and all nodes need to be recalculated and recreated.
4. ***Not suitable for large datasets***: If data size is large, then one single tree may grow complex and lead to overfitting. So in this case, we should use Random Forest instead of a single Decision Tree.

4. Whether Feature Scaling is required?

No

6. Impact of outliers?

It is ***not sensitive to outliers***.Since, extreme values or outliers, never cause much reduction in RSS, they are never involved in split. Hence, tree based methods are insensitive to outliers.

Types of Problems it can solve(Supervised)

1. Classification

2. Regression

Overfitting And Underfitting

How to avoid overfitting

Practical Implementation

1. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
2. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

Performance Metrics

Classification

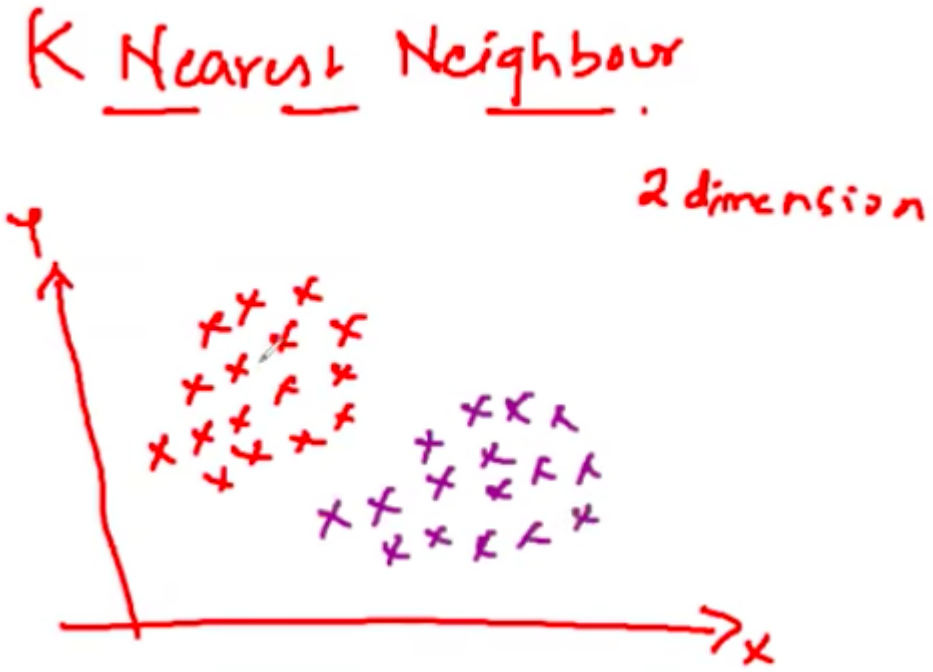
1. Confusion Matrix
2. Precision,Recall, F1 score

Regression

1. R2,Adjusted R2
2. MSE,RMSE,MAE

XVI. K Nearest Neighbor (KNN) Classification

Suppose I have a two-dimensional dataset as shown below:

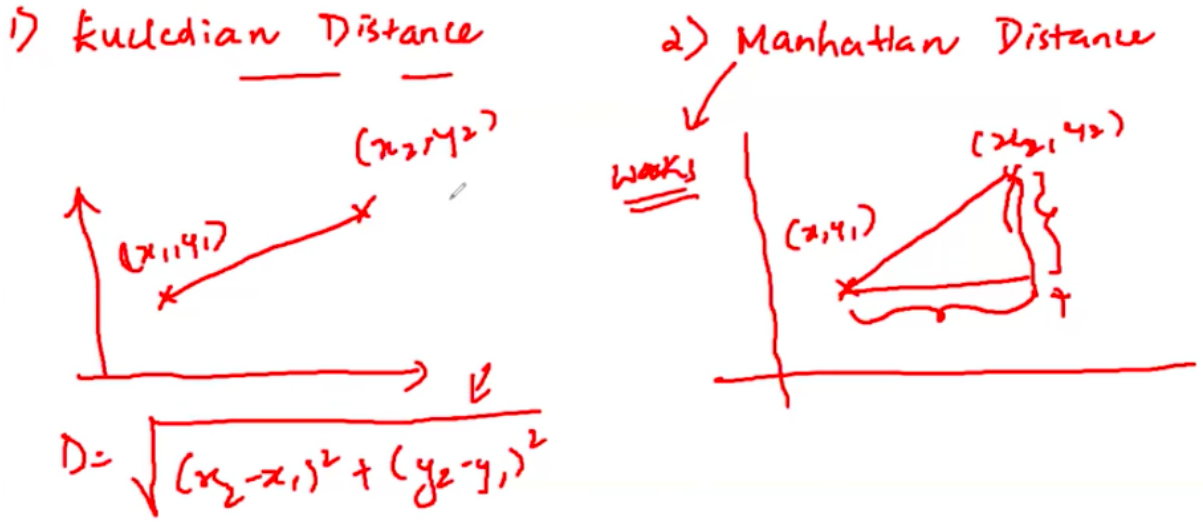


If a new point comes, then how do we determine whether this point belongs to Category 1 or Category 2. KNN helps us to determine the answer to this question.

In the first step, we select the K value which means how many nearest neighbors would be considered. After K is decided, the nearest K points from the new point are

identified for the two categories. So, we basically calculate this distance of the K nearest neighbors. The third step would be to identify how many nearest neighbors are of Category 1 and how many of Category 2 in the total K nearest neighbors chosen. If there is a maximum number of Category 1 neighbors, then that point belongs to Category 1 or else it would belong to Category 2. Once the model is able to classify points in this manner, the model is ready.

Now, how is this distance calculated?
Euclidean Distance: This formula says the distance between two points (x1 1 , y1 1) and (x2 2 , y2 2) is $\mathbf{d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}}$.
Manhattan Distance: Manhattan distance is calculated as **the sum of the absolute differences between the two vectors**. The Manhattan distance is defined by $\mathbf{Dm(x,y)=\sum i=1D|xi-yi| = |x2 - x1| + |y2 - y1|}$, which is its L1-norm.



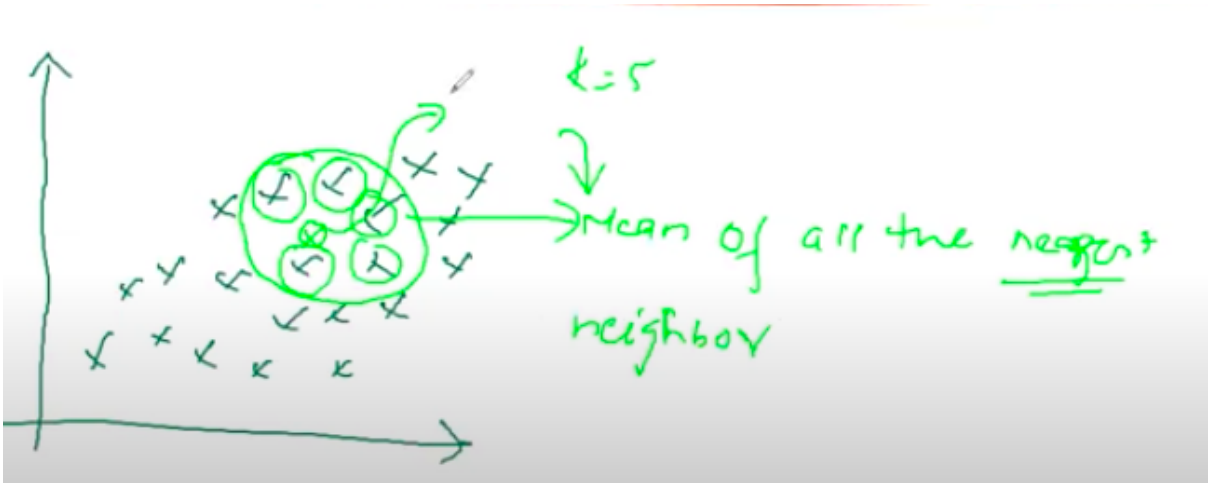
Suppose we have a highly imbalanced dataset with 900 Yes and 100 No, will the KNN classification model be biased?
Now, suppose a new point comes over in this dataset. Then, will the KNN be biased?

It will definitely get impacted. Suppose we take K as 150 and take 150 nearest neighbors. At that time, the maximum points present are of the majority category and this leads to bias inherent in the output.

What if we have outliers? Will the result of the classification get impacted?
The kNN may get biased or impacted because the bias cluster of neighbors might get selected in the K nearest neighbors.

If we get an equal number of nearest neighbors on both sides, we should then select the value of k as odd.

Regression use case of KNN: We take the mean value of the nearest neighbors to predict the regression value of a particular new input.



XVII. Ensemble Techniques

Two techniques:

Bagging (Bootstrap aggregation)

One of the examples is RandomForest.

Boosting

Examples: ADABOOST, Gradient Boosting, XgBoost

Bagging Technique:

Combining multiple models - for each and every model, we will provide a sample of records D' such that the dataset D' is always less than the overall dataset D. The dataset is resampled again and provided to the other model M2. This is called Row sampling with replacement. The models will get trained for the respective datasets simultaneously. For M1, another D'' data goes for testing and a certain output is generated. For a binary classifier model, the output would be of the form 0 and 1. Then, the output of all the models is applied with a voting classifier - majority of the output is taken as the final output.

If the majority vote is for 1 , then the output is 1 else it is 0. The step where row sampling with replacement is done is called bootstrap. The step where voting is done is called aggregation. Thus, the technique is called bootstrap aggregation.

Now, in Random forests, the models M1, M2 etc. get replaced with decision trees.

Random Forests:

In Random Forest , the models M1, M2, M3 etc. are decision trees. So, we also do row sampling with replacement and we also do feature sampling with replacement along with it. For aggregating, we use majority vote.
A decision tree model has low bias and high variance - it basically is prone to overfitting. The training error is very less but the test error might be high. In Random Forests, we are using multiple decision trees, and each might have high variance, but when we combine the result of these decision trees , it leads to low variance for the Random Forest model.

Suppose we have 1000 records in the beginning, and suppose at a later stage we change 200 records, will the result change for the Random Forest?
This data change will not make much impact on the model output because the data gets split in various decision trees using row sampling with replacement.

Instead of classification, for a regression problem, the random forest takes the average or median of the outputs from the various decision trees.

What are the hyper parameters used for Random Forest?
Number of decision trees to be used for a specific model

Boosting Techniques:
Base Learner models created sequentially to work on the datasets incorrectly classified by each previous base learner.
It will go on unless we provide the number of base learners to be used.

ADABoost (Boosting Technique):

Suppose for a dataset, we have 7 records, 3 features and one output column. We also assign a sample weight in ADABOOST. Initially, all the records are basically assigned the same weights. In Step 2, we create our first base learner with the help of decision trees in ADABoost. It is created with the help of only one depth unlike random forests. These decision trees are called Stomps. When we consider f1 , we create one stomp. For f2, we create another stomp and for f3, we consider another stomp. From these stomps, we need to select the first sequential base learning model based on which model has least Gini impurity or entropy.

Suppose the first option model classifies 4 correct and 1 incorrect records. Total error is calculated by summing up all sample weights which are misclassified. Thus, just one record is misclassified, then total error would be 1/7.

Now, performance of stomp is given by the following formula:
½ loge ((1 - total error)/total error).

Thus, performance of stomp = ½ loge(1 - 1/7 / 1/7) = 1/2loge6 = 0.896

Now, we need to update this weight for the records. We are going to increase the weights of incorrectly classified records whereas decrease the weights for others.

Now, new sample weight for incorrectly classified record = old weight * e^(performance)

New sample weight for incorrectly classified = 1/ 7 * (e^0.896) = 0.349

new sample weight for correctly classified record = old weight * e^(-performance)

New sample weight for correctly classified = 1/ 7 * (e^-0.896) = 0.05

Summation of all the weight values is done and then the weights are normalized by dividing by this sum. This will lead to a total sum value of weights equalizing to 1.

Now, for the second sequential model, based on the normalized weights, buckets are made. Our algorithm will run 8 different iterations to select the record randomly. Based on this new dataset, a new stomp model is prepared based on which model has least entropy or gini impurity.

Finally, after passing through all the stomps, we would get quite less error than at the initial stage.

Then, the majority vote would happen as it happens in random forests. Basically, we are combining weak learners to make a strong learner.

Gradient Boosting (Boosting Technique):

Suppose, we have two independent features: Experience and Degree; dependent feature is salary.

Step 1: Devise a base model to compute the average salary of all the outputs. We can take it as y', let's say we got it as 75.

Step 2: Compute residual errors, pseudo residuals:
Residual formula for example is (actual - prediction). There are various loss functions. This is just an example.

Step 3: Construct a decision tree sequentially. Inputs would be experience and degree. Output would be residual errors, R1.

Step 4: If we pass the independent features again from the decision tree, we get R2 values.

This leads to overfitting after calculating y' + R2 (i.e. predicted output + residuals). To prevent this, we will be adding a learning rate , α multiplied by R2. This results in a very higher salary as residuals get reduced by multiplying with alpha.

Thus, in the next step we add another decision tree and so on.

$F(x) = h_o(x) + \alpha_1 h_1(x) + \alpha_2 h_2(x) + \ldots\ldots + \alpha_n h_n(x)$

The main aim is to reduce this residual error. Here, alpha ranges from 0 to 1.

This is called a sequential tree or a boosting tree.

Mathematical intuition and Pseudo algorithm:

Wikipedia Link: https://en.wikipedia.org/wiki/Gradient_boosting

Inputs:

- $\{x_i, y_i\}$,
- Loss function (differentiable) - log loss, hinge loss etc. $L(y, FC_x)$,
- Number of Trees

Pseudo algorithm:

1. Initialize the model with constant value

$$F_0(x) = \arg\min_{\gamma} \sum_{i-1}^{n} L(y_i, \gamma)$$

Gamma is nothing but the predicted value. The loss function would basically be 1/2 $\sum(y' - y)^2$ where i ranges from 1 to n and y' is the predicted value.

We need to find a y' value such that the loss is minimized. Take the first order derivative of the loss function with respect to y' and find y'

2. For m = 1 to M:
M is the number of trees.

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

Fit a base learner (or weak learner, e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training $\{(x_i, r_{im})\}$

Compute multiplier γ_m by solving the following one-dimensional optimization problem

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Output $F_M(x)$.

Extreme Gradient - XgBoost (Boosting Technique):

In XgBoost, we always go for binary splits even if we have more than two classes.

Suppose, we have features, such as Salary, creditScore (Bad, Good, Normal) and the output is Loan approval(1/0).

Salary	Credit Score	Approval	Res	
<=50K	B	0	-0.5	
<=50K	G	1	0.5	
<=50K	G	1	0.5	
>50K	B	0	-0.5	
>50K	G	1	0.5	
>50K	N	1	0.5	
<=50K	N	0	-0.5	

We will start constructing the tree for base model:

[-0.5,0.5,0.5,-0.5,0.5,0.5,-0.5]
Salary → <50K and >50K i.e. [-0.5,0.5,0.5,-0.5] and [-0.5,0.5,0.5] respectively.

Step 1: Construct a base model

Step 2: Calculate the similarity score:
∑(Residuals)^2 / (∑Probability(1 - Probability) + **λ**)

Thus, similarity score for first split in our example: [-0.5 + 0.5 + 0.5 - 0.5]^2 /[0.5(1-0.5) + 0.5 (1 - 0.5) + 0.5 (1 - 0.5) + 0.5 (1 - 0.5)] = 0

Similarity score for second split: (-0.5 + 0.5 + 0.5)^2 / /[0.5(1-0.5) + 0.5 (1 - 0.5) + 0.5 (1 - 0.5)] = 0.25/0.75 = 0.33

Similarity weight for the initial salary (root) = 0.25 / 1.75 = 1/7 = 0.142

Step 3: Compute the gain

Gain = 0 + 0.33 - 0.14 = 0.21

Now, we will go for a credit split (binary): Bad vs. Good & Normal: [-0.5] vs.[0.5, 0.5, -0.5] for the first salary split in the previous step. Their similarity scores would come out to be 1 and 0.33 respectively.

The gain would be 1 + 0.33 - 0 = 1.33

Similarly, we would calculate the gain for the right side split by considering the categories as Bad and Good vs. Normal. The split which gives better information gain is what we would choose. The similarity scores in this case would be 0.33 and 1 respectively. So, we are getting the same gain. Then we can go ahead with either of them.

To decide whether we need to do further split or not - we use post pruning. It is basically calculated by a cover value = Probability(1 - Probability). We can find the value and cut the tree based on this or branch it.

Whenever I get new data, I need to find the base model output using log(odds) = log(P/1-P). For Probability 0.5, log(odds) = 1. The learning rate is 0.01. Then, applying sigmoid function σ(0 + 0.01*1) = σ(0.1) = 1 / 1 + e^ -0.1 = 0.6

Then, my new probability will get computed. Based on this new probability, new residuals are calculated. We will compute the decision trees by taking the residual independent features and take $\sigma(0 + \alpha(T_1) + \alpha(T_2) + \alpha(T_3) + \ldots\ldots \alpha(T_n)$. Probability would be good when residuals would be very less.

We can select this alpha value as a hyper parameter.

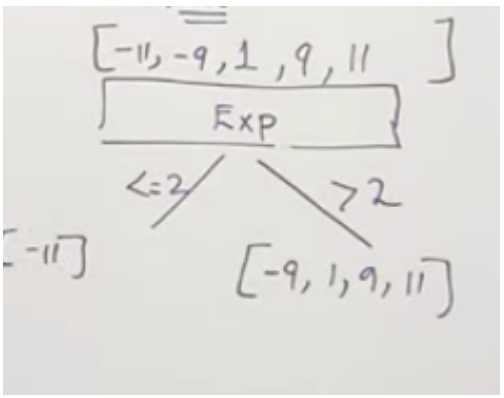
XgBoost Regression - In-depth Intuition

Experience	**Gap**	**Salary**	**Res1(w.r.t. 51K)**
2	Yes	40K	-11K
2.5	Yes	42K	-9K
3	No	52K	1K
4	No	60K	9K
4.5	Yes	62K	11K

Let us try to create a Base Model by taking the average of all the salaries → 51K.

Based on this average salary, we would find Residual, Res1.

Now, we would take Experience, Gap and Res1 and try to find the root node on which we would split.



We calculate similarity weight.

Now, if we recall that in XgBoost Classifier, the similarity weight was calculated by the following formula: ∑(Residuals)^2 / (∑Probability(1 - Probability))

Here, for XgBoost Regressor, this formula would change, and the similarity weight would be calculated by the following formula: ∑(Residuals)^2 / (No. of Residuals + **λ**)

To check if the split around 2 years experience was a good split or not, we calculate the similarity weight followed by Information gain and then decide.

Thus, similarity weight for the left side of current split using the above formula = 121 / 1 + 1) if we take **λ** = 1;

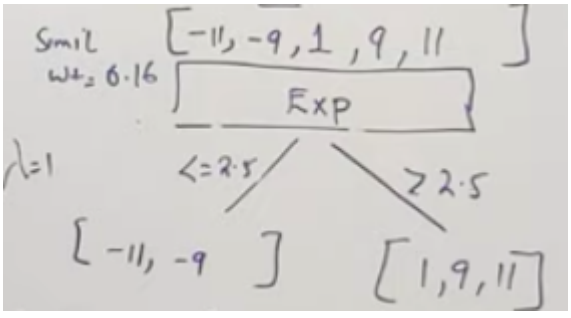
Thus, the similarity weight here for the left side of current split around 2 years experience comes out to be 60.5

Similarity weight for the right side of the split is calculated the same way: $(-9 + 1 + 9 + 11)^2/4 + 1 = 144/5 = 28.8$

Now, the similarity weight of the root: $(-11 + 1 - 9 + 9 + 11)^2/5 + 1 = 0.16$

Thus, our **information gain for first split** = 60.5 + 28.8 - 0.16 = **89.14**

Now, I will go with the next split. Let us split around 2.5 years of experience instead of 2 years as done earlier.



The left side would include [-11, -9] and the right side would include [1, 9, 11].

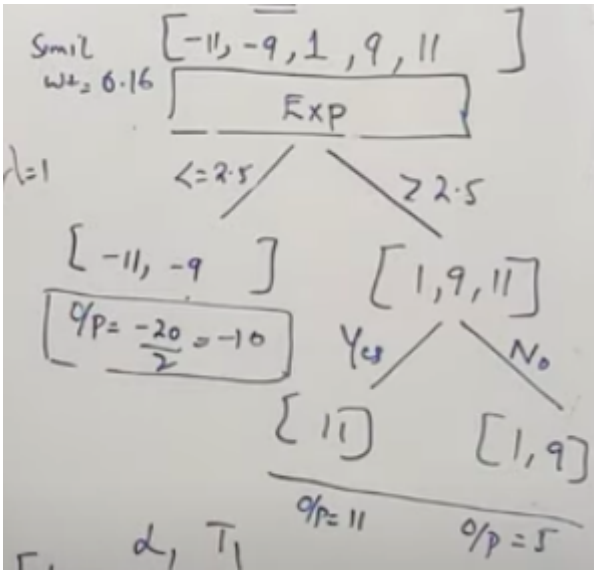
In this case, the similarity score for the left side would come out to be 133.33 and the similarity weight for the right side would be 110.25.

Thus, our **information gain for this split** = 133.33 + 110.25 - 0.16 = **143.42**

We deduce that this gain is better than the earlier split, thus we would go with this split rather than the first one. Similarly, we check for other possible splits and choose the one with the best information gain.

The second split after deciding the first split would be around Gap (Yes, No) and would try to compare which has the highest information gain.

Finally, the output of each node would be the average value.



After this base model, we pass the records through this base model,

Let us consider our learning parameter, alpha = 0.5. Suppose the first record is being passed through this base model, the result would be: 51 + (0.5)[-10] = 46K

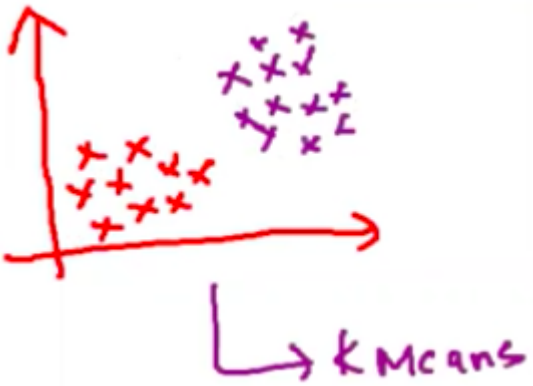
Like this, we consider multiple decision trees, output = base salary + $\alpha(T_1)$ + $\alpha(T_2)$ + $\alpha(T_3)$ +..... $\alpha(T_n)$

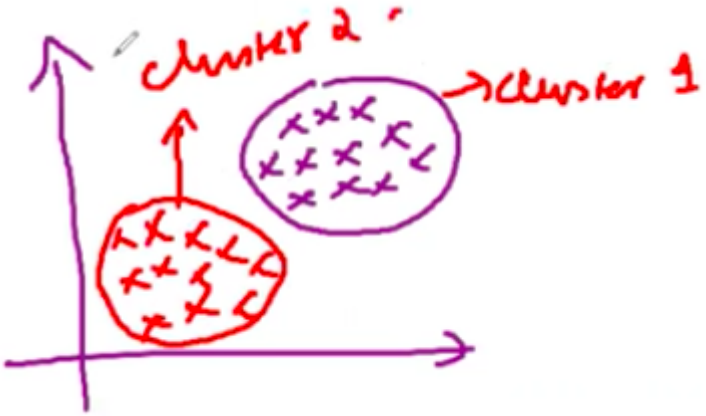
There is also one more parameter, γ. For example, we can take γ as 150.5. Now, we subtract information gain after each split from γ and see if the value is negative or positive. If this value is negative, we can prune the tree at this point whereas we can continue if the value is positive. This γ is a hyperparameter that is used in post pruning.

XVIII. K Means Clustering (Unsupervised technique)

Note: Elbow method is used to find K

Math Intuition:

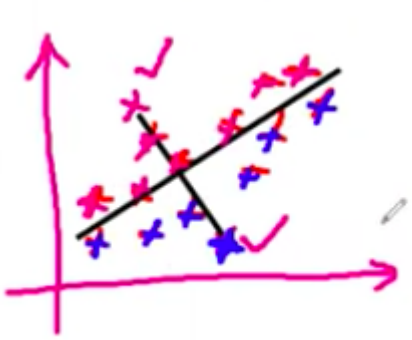




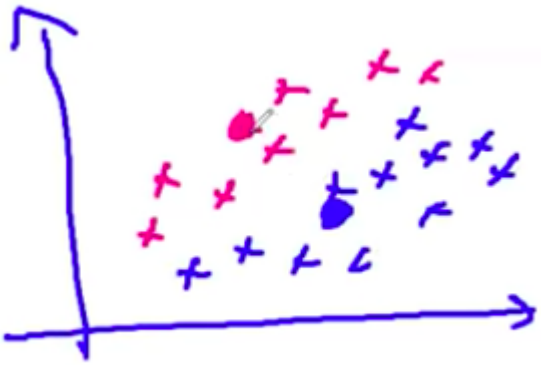
How does this algorithm work?
What metrics should we use?
Which distance do we consider - Euclidean or Manhattan?
What is this elbow method?

This K value is basically centroids - which means those many clusters.

Step 1: Let us suppose K = 2
Step 2: Initialize K centroids randomly in the plane
Step 3: We will use Euclidean distance and find points which are nearer to these centroids respectively. We select the groups and find the mean value for each group.



Step 4: Move the centroids taken initially to these mean positions.



Step 5: Again, we find out which all points are nearer to the new centroids and redefine the groups and calculate their means. Update the centroids and continue so on.

This will continue unless we get two fixed constant groups with less changes. No movement of points happens from one group to another.

But, how do we select this K value initially?

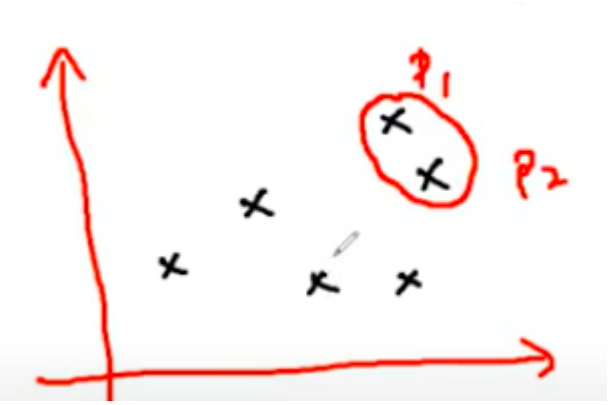
Using Elbow method: We run a loop from K = 1 to say some n value (suppose 20)

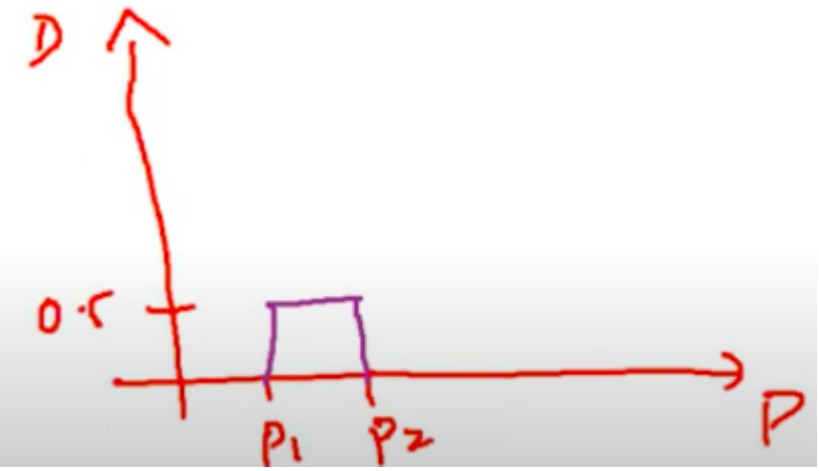
Whole process of K means it is run for K= 1. Then, we find out within cluster sum of squares, Wcss = $\sum(c_i + x_i)^2$ where i ranges from 1 to n. For K = 1, this Wcss would be quite high.

Then, we plot Wcss vs. K. Initially, with K = 1, Wcss would be quite high. As we increase K = 2, this Wcss value decreases because we have two centroids now. At a certain point, this Wcss value becomes almost constant. We have to select this last K value that had an abrupt decrease. After selecting a particular optimal K value, we implement the K-Means clustering algorithm using the above steps.

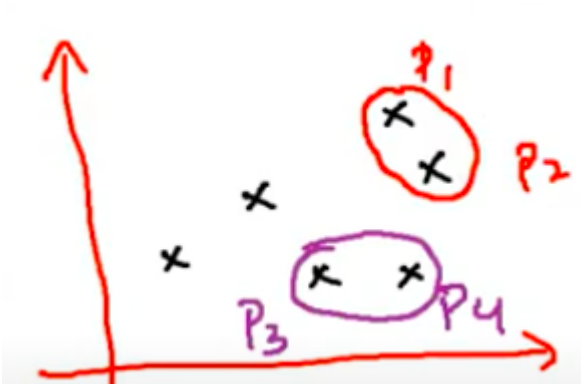
XIX. Hierarchical Clustering (Unsupervised technique)

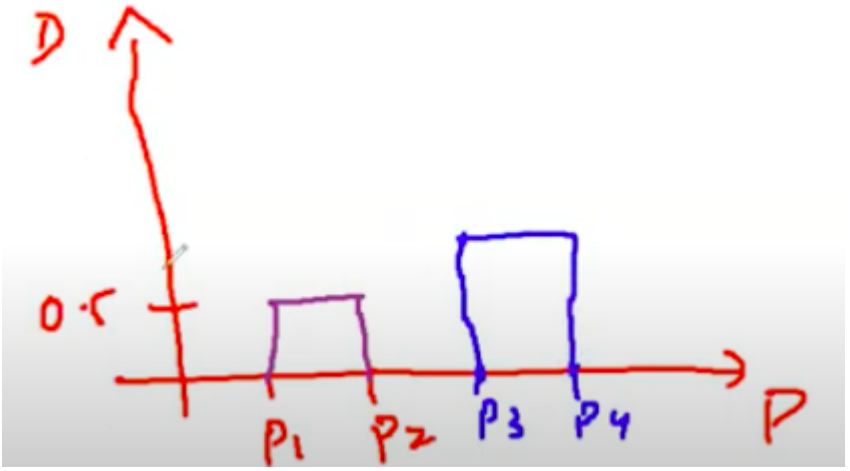
Initially, we would have some points. Each and every point is specified as a distinct cluster initially. Suppose, two points appear to be nearer and another three points appear nearer than to the rest of the points. We would combine these points in a dendogram and find the average value.



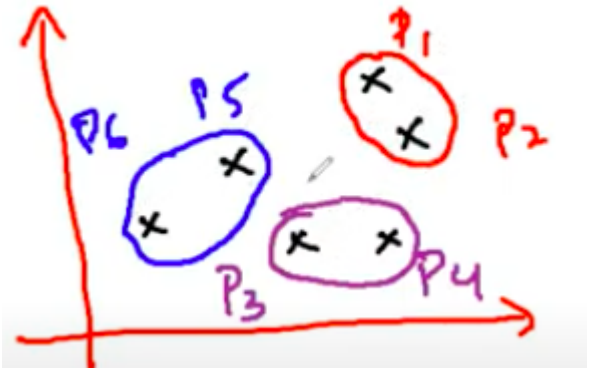


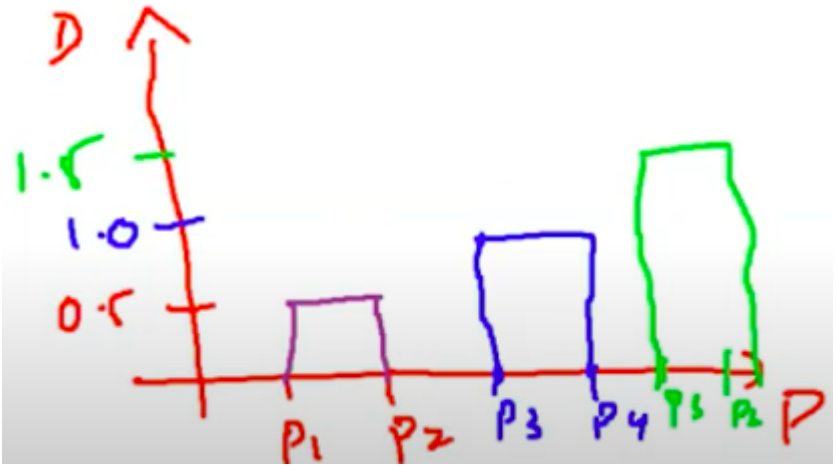
Then, we find the next two or more closer points and define another dendogram.



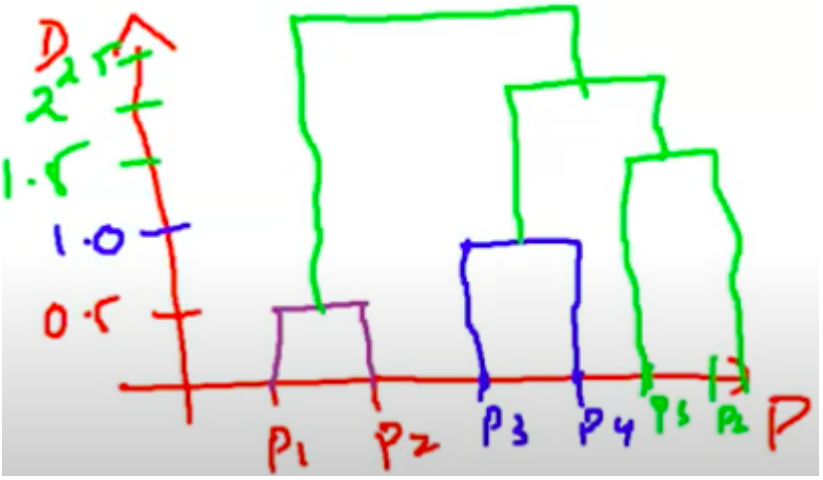


Then, we proceed to find the next nearest points and define another dendogram.



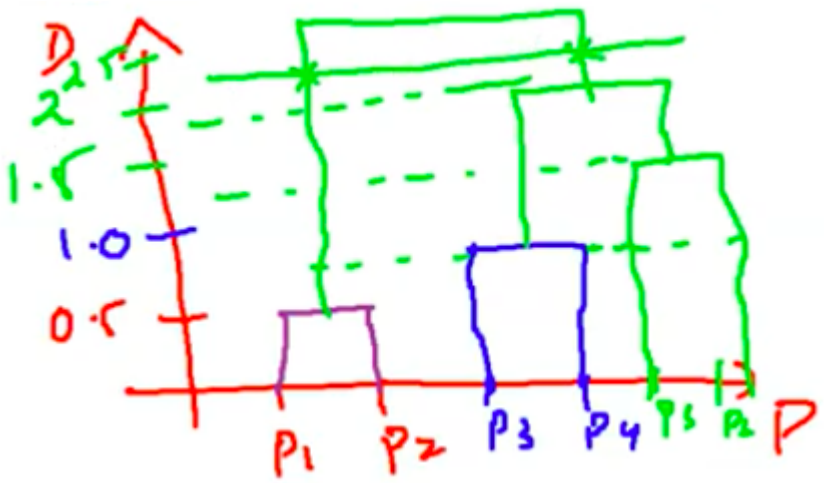


Then, we try to find out which two clusters are closer to each other. Then, we combine those two clusters. And finally we combine the entire groups to form a single cluster.

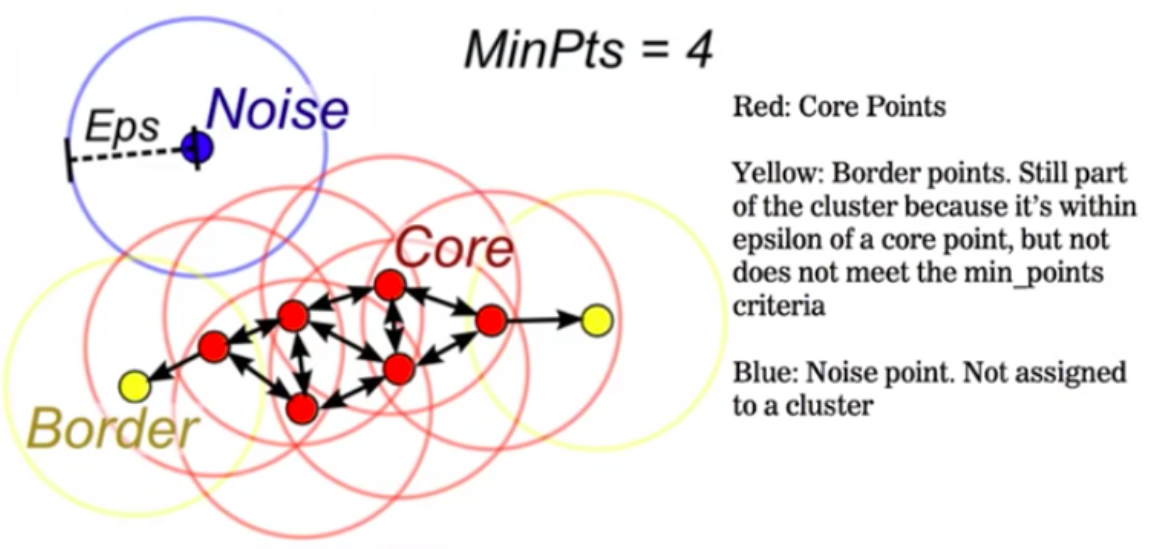


Our main aim is to find out basically what should be the exact number of clusters we should use. This distance is measured by the Euclidean distance formula. To find out the exact number of clusters we should form, we should select the longest vertical line such that no horizontal line passes through it.

Now, once we find this line, we draw a horizontal line and find how many intersecting points are there. Those many clusters need to be formed.



XX. DBScan algorithm (Density-Based Spatial Clustering of Applications with Noise) {Unsupervised Technique}



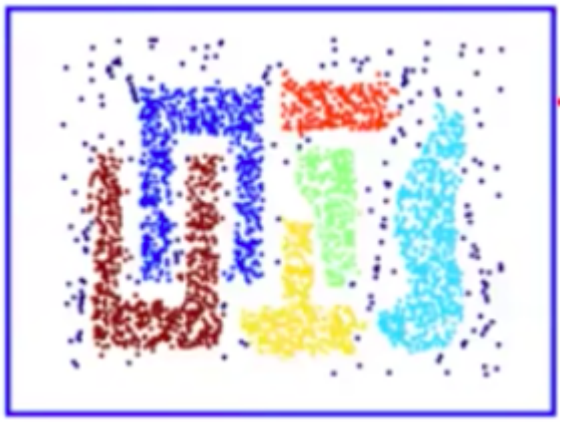
Five components:

1. Epsilon
2. Minimum points
3. Core points
4. Border points
5. Noise points

We initially consider some Epsilon and minimum points values. Epsilon value indicates the radius to create a circle. Within this circle, I have to consider at least certain points given by minPoints. The point within this circle is Core point. Suppose, a specific point does not satisfy the condition of minimum points, even though it's within epsilon of a core point, then this is called a border point. Suppose, I have another point which does not satisfy epsilon and minimum number of points condition, then this point becomes a noise point or an outlier. DBScan treats these noise points very nicely and does not treat them as a separate cluster.

Example of K-Means clustering with noisy data:

Example of DBScan clustering with noisy data:



Advantages of DBScan:

- Is good at separating clusters of high density versus clusters of low density within a given dataset
- Good with handling outliers

Disadvantages of DBScan:

- Not works well with clusters of varying densities. It struggles with clusters of similar densities.
- Struggles with high dimensionality data.

Library used: sklearn.cluster.DBSCAN

XXI. Validation Techniques for Clustering algorithms

1. ***Silhouette (Clustering)*** :
   The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to

its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.

The silhouette can be calculated with any distance metric, such as the Euclidean distance or the Manhattan distance.

Three steps are involved:
Step 1: We will take one datapoint and will calculate the distance within the cluster. We will calculate this distance for each point with respect to the other points in the same cluster. This average of distance is denoted by a_i and is given by the following formula:

For data point $i \in C_I$ (data point i in the cluster C_I),

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j)$$

Where, a(i) is the mean distance in cluster C_I for data point i. $|C_I|$ is the number of points belonging to cluster I and d(i,j) is the distance of point i from j. We have taken $|C_I|$ - 1 in the denominator because we are not considering the distance of point i from itself.

2. We then define the mean dissimilarity of point i from a cluster C_J as the mean of the distance from i to all the points in C_J (J is not equal to I).

For each data point $i \in C_I$, we now define

$$b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

3.

We now define a *silhouette* (value) of one data point i

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \text{ if } |C_I| > 1$$

and

$$s(i) = 0, \text{ if } |C_I| = 1$$

Which can be also written as:

$$s(i) = \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases}$$

From the above definition it is clear that

$$-1 \leq s(i) \leq 1$$

If a(i) >> b(i), then clustering is not done properly. If a(i) << b(i), then clustering is done appropriately.

A high value of s(i) indicates that the points are correctly matched to the right clusters and poorly matched to the neighboring clusters.

XXII. Pandas Profiling Library

If you are new to data science and EDA, you can do reverse engineering and utilize pandas profiling library to get the steps you need to take for EDA.

Vaex Library
Used for reading quite huge datasets

Pandas Visual Analysis Library
Used for visual analysis

Sweetviz Library
One statement EDA

D-Tale Library
If you are an expert in EDA, then you can improve your EDA using this library.

XXIII. Flask Framework

Introduction: Web application framework which is written in Python. Used to create end-to-end projects. Armin Ronacher developed this framework and he leads an international group of Python enthusiasts called Pocco. This framework is based on the WSGI and Jinja2 template engine.

WSGI - Web server gateway interface. It is a standard or a protocol. Any application is hosted in a web server such as Apache or IIS. Now, the Flask framework also requests for web server access (http requests). From this web server, a standard or a protocol is required for communication. This kind of protocol is WSGI. Any kind of callable functions, API can be there.

Jinja2 template engine is a popular template engine for python. Web templating system combines a web template along with a certain data source wherein this data source would render dynamic pages.

For example, my machine learning application has a goal to classify images into cat or dog. When the person uploads it, the system goes and interacts with the pickle file or the H5 file of the model. Then it gives us the response. Here, our data source is our machine learning model. When it comes back, here the information is getting

displayed. You are basically trying to combine a data source (ML mode) with the web template.

Both the projects (WSGI and Jinja2) template engine are called as Pocco projects

Why do we write this - if __name__=='__main__':

Decorator - which has two things in it - rule, options

```
@app.route
```

XXIV. Django Framework

Introduction: A high level Python web framework which is very easy to code, with a lot of configurations that are easy to set up.
It is very pragmatic and loosely coupled. It is open source.
Uses MVC (Model View controller) architecture.
Allows us to build better web apps.
Ridiculously fast, secure and scalable.

When should we use Django and when should we use Flask?
When your application is extremely large with various modules and functionalities → use Django. You will be able to write a clean code and loosely coupled code that can be integrated with various modules created by diverse developers.

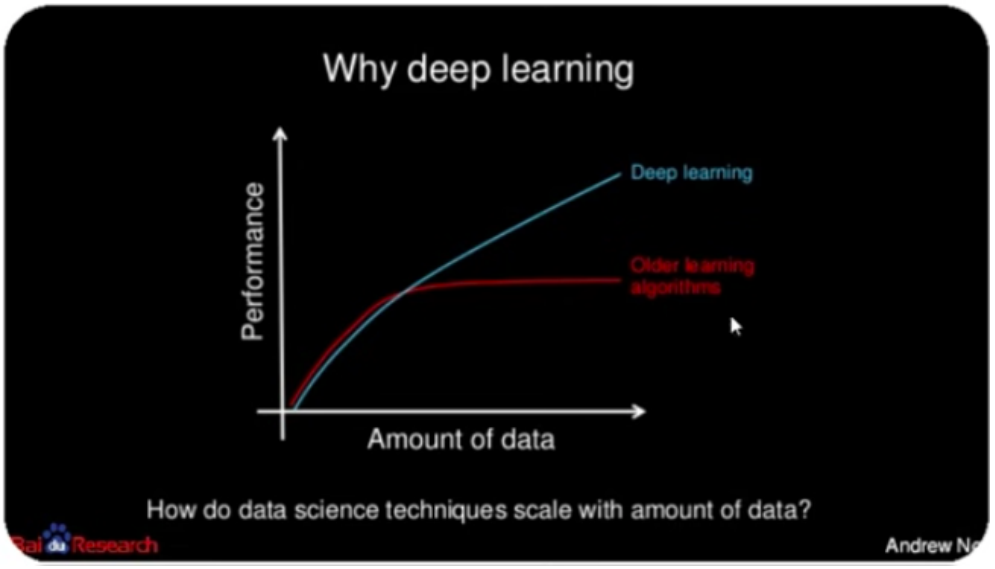
For full stack development, Django framework is the best to go ahead.

Flask works well when AI models need to be integrated with a web application / mobile application, then Flask can help create API's to integrate AI module functionalities with those applications as an end point.
For django, you do not connect for one or two functionalities but the entire huge module.

XXV. Deep Learning

Why is it becoming so popular?
In the recent trend from 2013 onwards, people searching about Deep learning and researching about it has increased a lot. What led to such a trend?

- There was an exponential growth in data as people started using smart phones, social media platforms, youTube.
- Interesting use cases: recommendation systems, face detection are possible because of data itself.



The performance of the model (machine learning) decreased at a point as the data increased exponentially whereas the performance of deep learning models performed really well.

- Technology upgradation also made cheap and good hardware readily available for example NVidia is bringing up so many new GPU's in the market.
- Cloud availability at a very cheap rate which can be used in data centers. We are able to train an image classification model for 10000 images in 20-22 minutes for so many epochs.
- Life cycle of a machine learning project comprises so many steps: feature extraction and selection, training the model and performance evaluation. But, in deep learning, this feature extraction and training the model are basically included in a single step.
- Deep learning projects can really solve complex problem statements because of this feature extraction and model training happening in one single task.

Complete RoadMap one should follow to learn deep learning:

Libraries: PyTorch, Keras, TensorFlow, nlp, HuggingFace, ktrain

Concepts of Perceptron and neuron brought by Geoffery Hinton.

Basic Concepts: Introduction to Neural Network, Loss function, Optimum minima, Gradient Descent, SGD, Adagrad, RMSprop, Adam

RoadMap: Deployment projects, Artificial Neural Network → Deployment projects, RCNN, Masked RCNN, SSD, YOLO, Object Detection, Transfer Learning Eg. VggI6, Alexnet, Convolutional Neural Network, Computer vision → NLP, HuggingFace, Ktrain, BERT, Transformers, Encoders And Decoders, Attention Models, Word Embeddings, Word2Vec, LSTM, GRU, Bidirectional LSTM, Recurrent Neural Network

With the help of RNN, we will be able to do forecasting as it handles sequences of data.

Follow: Keras Blog to explore Model API's

Introduction:
What is the difference between AI, ML, DL, DS?

Artificial Intelligence (AI) : If we consider the entire universe of applications, we are creating an AI application. Some of the examples: Netflix, autonomous vehicles, amazon insights, YouTube. Netflix basically is a software application which performs on user interaction and has recommendation systems. In an AI application, basically not everything is an AI module, but a part of it is. Creating an AI application which works without any human interaction.

Machine Learning(ML) is basically a subset of AI. For ML the most important thing is data and it provides stats tools to analyze, preprocess, predict and forecast. Different types of machine learning techniques:

- ***Supervised ML*** - We have two features: height and weight (Labeled data). It will be able to train with these labeled features, and when we give it height, it gives weight as the output based on the learning. Here, height would be an independent feature and weight would be a dependent feature. Supervised ML basically has 2 distinguished techniques: ***Regression*** (output will be a continuous value) and ***Classification*** (output will be a discrete categorical value - fixed number of categories). Classification can be ***binary classification*** (for example: Gender - male, female) or ***multiclass classification*** (for example: Gender - male, female, none) or ***multilabel classification*** (for example: Recommendation systems - Movies Categories 1,2 and 3. A movie can also be an action, thriller and romance movie at the same time.). ***Forecasting*** is also a regression problem. It will always be a continuous value. We usually have such a problem statement in time series analysis.
- ***Unsupervised ML*** - We don't know basically what would be the output. Clustering, segmentation, dimensionality reduction are some of the tasks we do in an unsupervised ML model. Suppose, we have a costly product being launched and some of the customer data (salary, spending score). We wish to send an email to all the customers to tell them about this specific product. Which customers should we actually send the email to? From this entire dataset, we cluster them into multiple groups / clusters based on similarities on their salaries and spending score. Then, we can categorize them later on based on the clusters obtained. For example, ones with the highest salaries can be categorized as richest people.
- ***Semi-supervised ML*** - Initially, we would have some kind of labeled data and we would train our model based on this data. Later on the model based on its interaction with the users, the model learns. For example, we feed some initial information while signing up on Netflix and the model trains. Later on, based on our watch history and downloads, the model learns and recommends. ***Reinforcement learning*** is an example of this.

Deep Learning (DL) - Subset of Machine learning. Deep learning is mostly about neural networks. It is used to mimic the human brain. Make the machine learn the way humans do. For example, chess. In deep learning, we also have supervised, semi supervised and unsupervised learning.

Topics of Deep Learning to focus upon:

- ☐ Artificial Neural Network (ANN) - works on all kinds of tabular data {f1, f2, f3….}
- ☐ Convolutional Neural Network (CNN) - works on images or videos (multiple frames of images with sound) as input data - Highlight generation of cricket matches (combination of video and sounds generated from 3-hour matches)
    - Image classification (CNN, Transfer learning Techniques)
    - Object Detection (RCNN, Fast RCNN, Faster RCNN, Masked RCNN, SSD, YOLO, Detectron)
    - Object segmentation (Used in self-driving cars)
    - Tracking
    - GAN
- ☐ Recurrent Neural Network (RNN) - works with inputs: text, time series and sequential data
    - RNN
    - LSTM RNN
    - Bidirectional LSTM RNN
    - Encoder, decoder
    - Transformers
    - BERT
    - GPT1, GPT2, GPT3

NLP → some text data → machine will not be able to directly understand it → convert it into numerical vectors → Bag of words, TF IDF, word2vec, embeddings

Google is also doing research in sarcasm - Hey! You are very intelligent! (sarcasm)

Data science (DS): Part of everything - ML, DL and AI. One cannot say that one is not good in NLP or DL or ML.

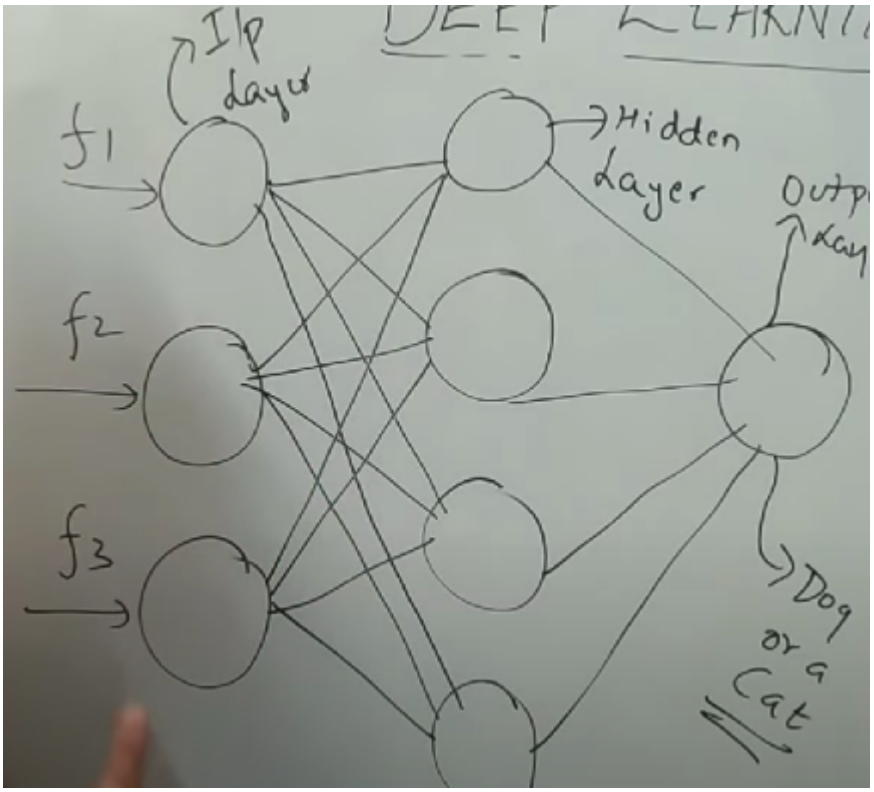
Predict tea crop production for the next six months? → It is a supervised learning problem. The features would be like max rainfall, average rainfall, max temperature, average temperature. What should be our approach? First we should make clusters (groups based on climate conditions) using clustering algorithms → and for each group we can develop a regression model to predict.

Introduction to Neural Network:
In the 1950's and 60's, researchers thought can we make them learn the same way as we do?" Then, they came up with a neural network and perceptron. But, there were some problems with it. Then, later on Geoffery Hinton came up with back propagation.

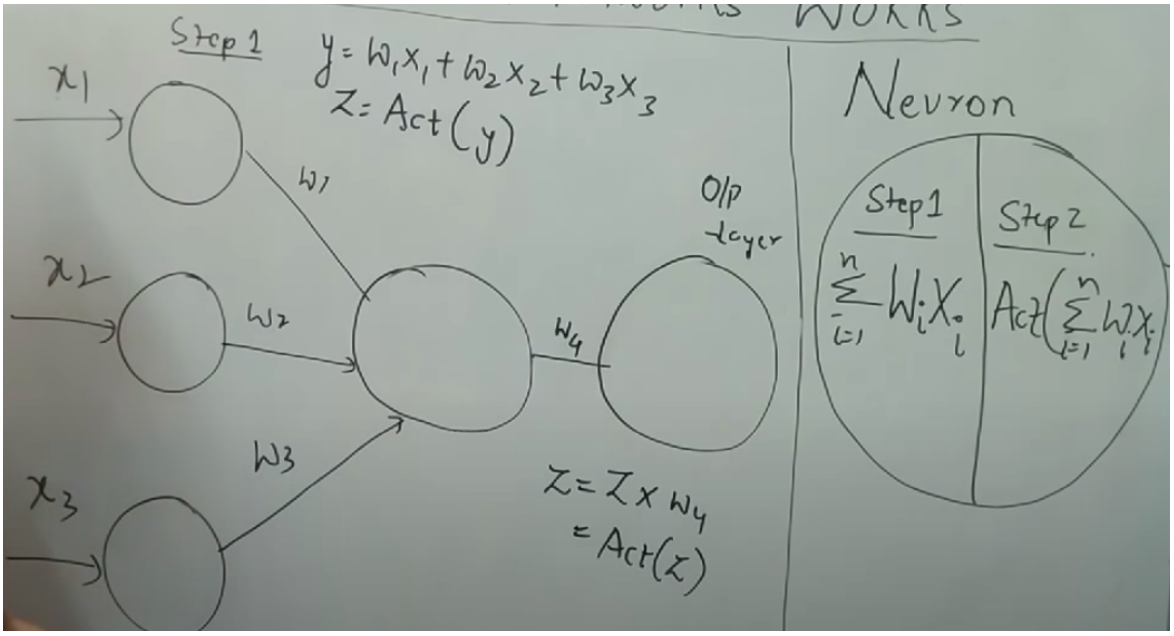
When a child is born, she is not able to distinguish directly between a cat and a dog. Then, the child's parents help the child identify a cat and a dog with the help of certain features that distinguish them. The same thing can be done for a neural network.

The first layer of a neuron network is the input layer with the features (nodes) present in it. The hidden layer can have any number of layers and any number of neurons. Then, we

finally get the output in the output layer (number of outputs decides the number of neurons in the output layer).



How does a neural network work?



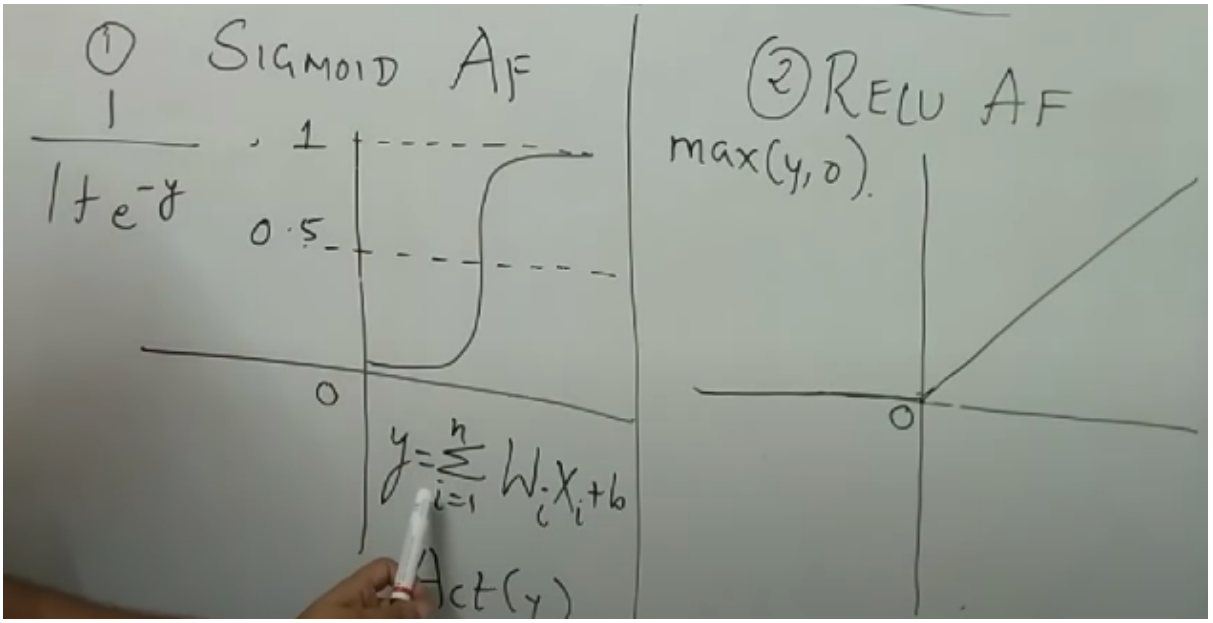
I have three features x1, x2, x3 that pass through the neural network as shown above in the classification problem. Two types of operations happen in the hidden neuron, as shown in the right side of the figure above: Step 1: $\Sigma W_i X_i$, Step 2: Activation function is applied.

For example, sigmoid is an activation function, equation is 1/(1 + e^(-y)). If this value is less than 0.5, then the neuron does not get activated whereas if it is more than 0.5, the neuron gets activated and passes the information to the brain.

There are various kinds of activation functions. We will be discussing them in detail in the below section.

The neurons to trigger are decided by updating the weights and the weights are updated by training the model

Various kinds of activation functions:



Sigmoid activation function transforms your 'y' between 0 and 1. Its equation is 1/(1 + e^(-y)). If this value is less than 0.5, then the neuron does not get activated whereas if it is more than 0.5, the neuron gets activated. This activation function is also used in logistic regression. Here, y = $\Sigma W_i X_i + b_i$ for i from 1 to n. When the output is 0, the neuron is not activated whereas it is activated when the output is 1.

RELU Activation function: Its equation is max(y,0) where in y = $\Sigma W_i X_i + b_i$ for i from 1 to n. The output ranges between 0 and 1.

When we do regression problems, we try to use RELU. But, when we do classification problems, we tend to use RELU in the hidden layers whereas Sigmoid functions in the output layer. Because the output could be 2 categories, 3 categories and so on and that is what sigmoid function does.

Neural Network Training: y = [w1x1 + w2x2 + w3x3] + b1 ; z = Act(y)

Play : 2 hrs
Study : 4 hrs

Sleep : 8 hrs
O/p : 1 (PASS)

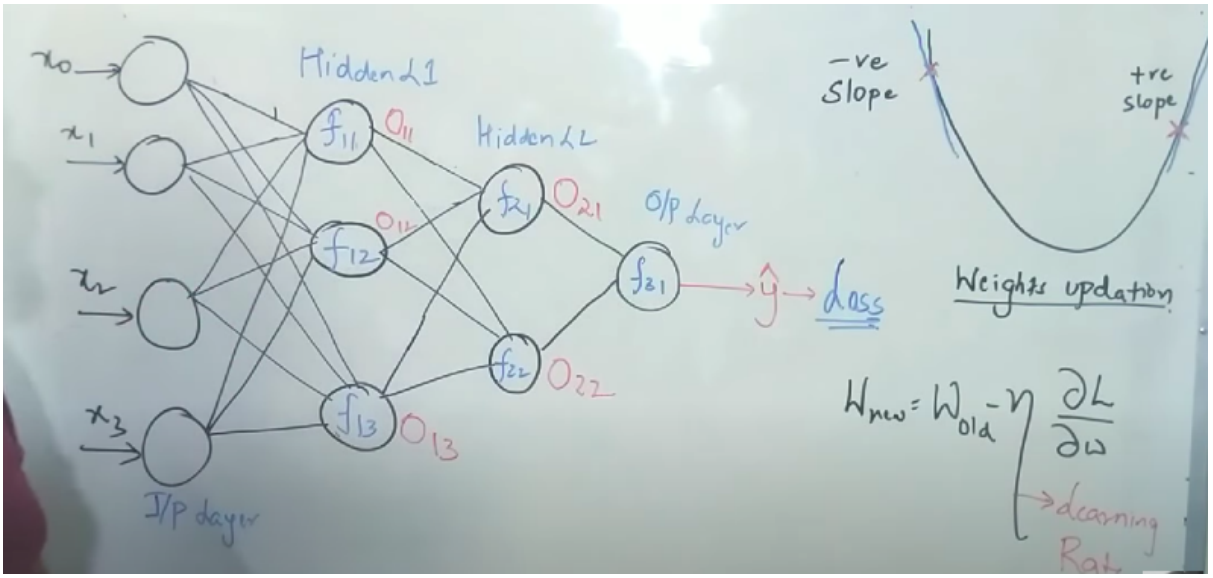
Activation function may be sigmoid. “Y’ “ is the predicted output. Suppose Y’ is predicted as 0. Now, by training data we know that y = 1 (actual output). Then, we would calculate the loss function, Loss = (y - Y’)^2 = (1 - 0)^2 = 1. So, we should try to adjust the weights w1, w2, and w3 along with bias to predict the correct output to reduce the loss function value to a minimum. During the training phase, we basically update the weights such that the loss value decreases. It can basically be done by using an optimizer, such as Gradient descent, SGD and many more. We basically do back propagation to update the weights. During back propagation, weights are updated as follows:

$w4_{new} = w4_{old} - \eta dL/dw4$ where L is the loss function; η is the learning rate.

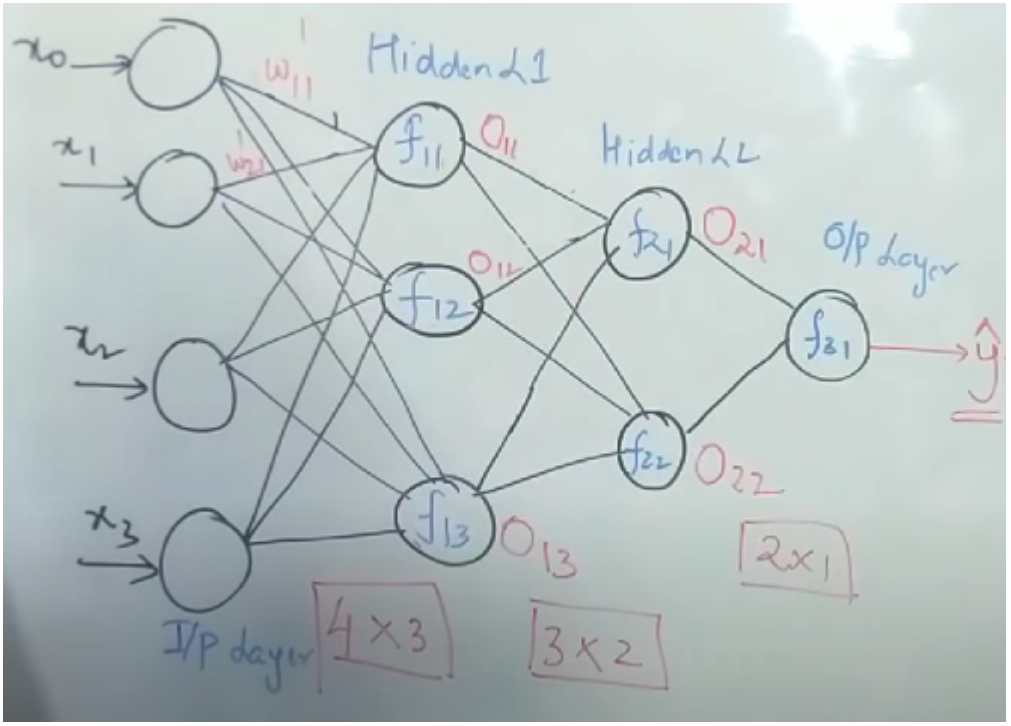
The learning rate is a very small value as it helps us to reach a global minimum. Once w4 is calculated, we update w1, w2 and w3 as well.

$w3_{new} = w3_{old} - \eta dL/dw3$. Once all the weights are updated, the process is repeated again, unless the loss is minimized. Since we had a single entry value, we defined a loss function. But if we have multiple records, we calculate the cost function.

Multilayered Neural Network Training -



We Are Using Two Hidden Layers And One Output Layer. We have 4 features - x1, x2, x3 and x4 in the input layer. The four features get passed to the hidden layer wherein certain weights get applied to it. Thus, the matrix would be of the shape 4 X 3 as we have 3 neurons in the first hidden layer. The weights would be w11’, w21’ and so on. Then, in the next layer, there would be a 3X2 matrix of weights. Finally, in the output layer, it would be a 2X1 matrix. Finally after multiplying weights with xi and then applying bias and activation function, from the output layer we get our predicted output y’.



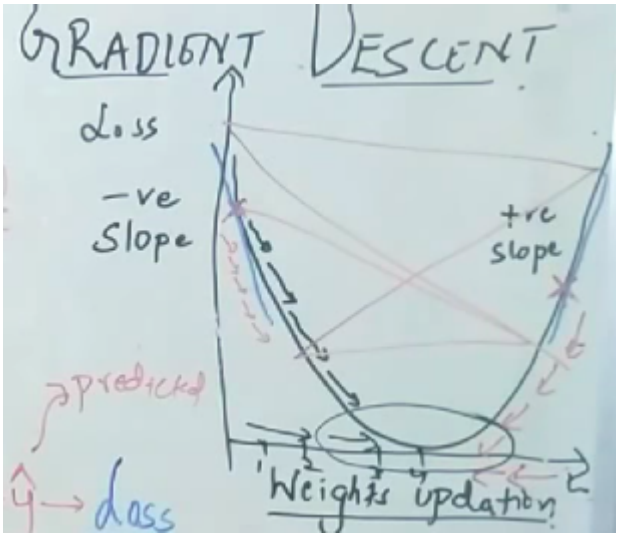
Loss function is calculated as (y - y')^2. My aim is to reduce this loss value. To reduce this loss we use an optimizer. One of such optimizers is Gradient descent. It updates the weights such as the cost/loss minimizes.

Weights are updated in each and every layer by backpropagation.

$w_{new} = w_{old} - \eta dL/dw$ where η is the learning rate.

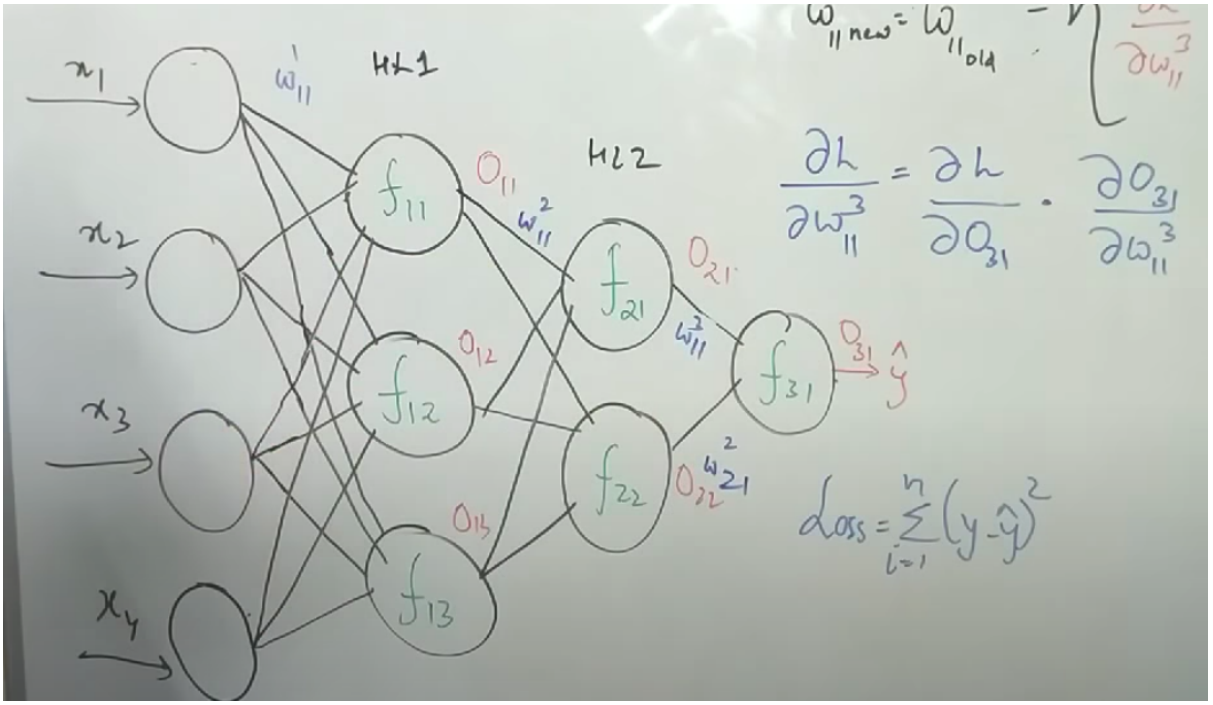
Derivative basically means the slope as shown in the above gradient descent graph. Suppose, the initial weights are initialized somewhere on the curve with negative slope. We check the right hand side of the tangent line drawn from that point if it is pointing downwards to see if it has a negative slope. It needs to reach the global minima for reducing the cost function / loss value. If this loss value is not decreasing, it means the point is oscillating.

As the slope is negative, w_{new} would be more than w_{old}. And, gradually we achieve a global minimum. But, if we take a very large learning rate, then the value keeps oscillating and might not reach the global minimum. Suppose the slope is positive, then w_{new} would be lesser than w_{old}. And, thus it will reach the global minimum gradually.



This entire movement from w_{old} to w_{new} is one epoch. And, after some epochs the loss value should decrease otherwise there is some problem.

Backpropagation Chain Rule:



For the first layer, w_{11}^1 where suffix denotes the neuron 1 in input layer connected to the neuron 1 in first hidden layer and superscript denotes the first layer. The function in each hidden layer performs two steps as discussed earlier (multiplying inputs and weights, applying bias and activation function to the whole).

We basically have to reduce this loss value by using an optimizer that updates the weights using back propagation.

$w_{11}^{3}{}_{new} = w_{11}^{3}{}_{old} - \eta dL/dw_{11}^{3}$
$dL/dw_{11}^{3} = dL/do_{31}.\ do_{31}/dw_{11}^{3}$

This is basically the chain rule in back propagation.

$w_{21}^{3}{}_{new} = w_{21}^{3}{}_{old} - \eta dL/dw_{21}^{3}$
$dL/w_{21}^{3} = dL/do_{31}.\ do_{31}/dw_{21}^{3}$

Similarly, $w_{11}^{2}{}_{new} = w_{11}^{2}{}_{old} - \eta dL/dw_{11}^{2}$, here $dL/dw_{11}^{2} = (dL/do_{31}).(do_{31}/do_{21}).\ (do_{21}/dw_{11}^{2})$

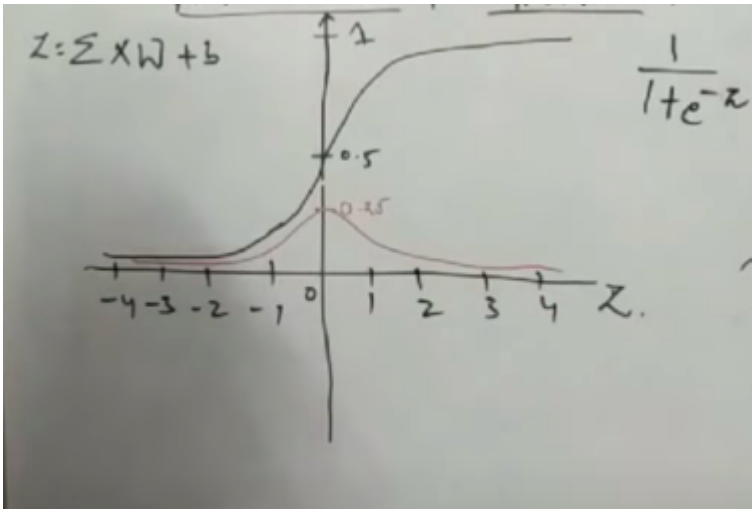
But, from the same node we are also impacting the other nodes in the next layer. To account for its impact, we would take additional derivative as shown below:

$dL/dw_{11}^{2} = (dL/do_{31}).(do_{31}/do_{21}).\ (do_{21}/dw_{11}^{2}) + (dL/do_{31}).(do_{31}/do_{22}).\ (do_{22}/dw_{12}^{2})$

These steps will go on unless we reach the global minimum.

Vanishing gradient problem: Earlier in 1960's , only sigmoid activation function was used which led to vanishing gradient problem. Due to this, deep neural networks were not possible. Let us see what is the vanishing gradient problem. Sigmoid activation function basically transforms the o/p of a neuron between 0 and 1. It then passes the o/p to the next layer neuron. Then, finally to reduce the loss function, we reduce the back propagation chain rule. The weight updation formula: $w_{11}^{1}{}_{new} = w_{11}^{1}{}_{old} - \eta dL/dw_{11}^{1}{}_{old}$.
$dL/dw_{11}^{1} = do_{21}/do_{11}.do_{11}/dw_{11}^{1}$

Since, we are using sigmoid activation function, the derivative of sigmoid would be specifically between 0 and 0.25.

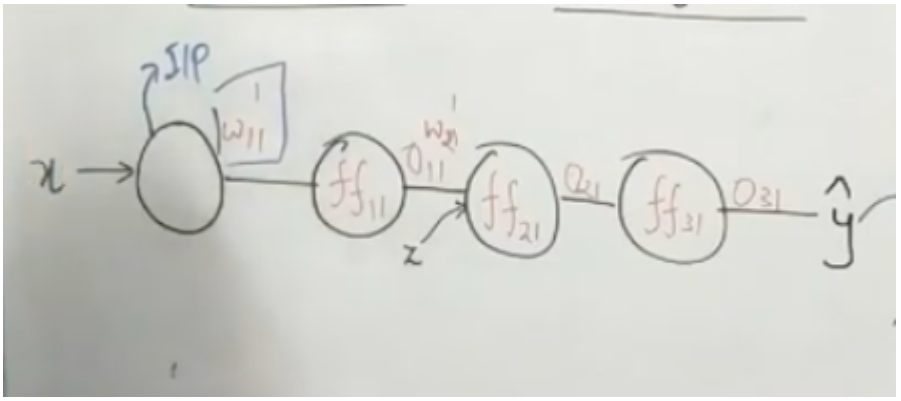


$0 <= \sigma(z) <= 0.25$. Thus, in our chain rule, all the derivative values range from 0 to 0.25. Suppose, one of the derivative values is around 0.2 and the other value is around 0.02. As we go backwards, this derivative value reduces. When we multiply these values, we get 0.04 approximately. Then, we apply this value in weight updation. Let us presume that our learning rate is 1, then $w_{11}^{1}{}_{new} = w_{11}^{1}{}_{old} - 0.04$. When we compute this we get 2.46 if our old

weight is 2.5. We see that as our number of epochs increases, this derivative value decreases gradually to an exponential value. So, there would be very less weight update. As the number of layers increases, this derivative value becomes quite small. This is why sigmoid function was not used in each layer. It leads to a vanishing gradient problem.

If you use tanh, you would face the same problem as its derivative ranges between 0 and 1.

Exploding gradient problem:



$w_{11}^{1}{}_{new} = w_{11}^{1}{}_{old} - \eta dL/dw_{11}^{1}{}_{old}$.
$dL/dw_{11}^{1} = do_{31}/do_{21}.do_{21}/do_{11}.do_{11}/dw_{11}^{1}$

$o_{21} = \phi(z)$ where $z = w_{21}.o_{11} + b_2$

The main reason the exploding gradient problem occurs is because of weights.
Since, we are using sigmoid activation function, the derivative of sigmoid would be specifically between 0 and 0.25.

$do_{21}/do_{11} = d\phi(z)/dz.dz/do_{11} = 0<=\phi(z<=0.25 * w_{21}$

Now, suppose I initialize weight w_{21} as 500. Considering my weights are higher, then I would get a very large value. And, if I replace that larger derivative value in w' update equation, it will become a very small value and this will lead to a lot of variations in the weights and it will never converge. That is why weight initialization is very important. This is an exploding gradient problem which is started off by faulty weight initialization.

Drop Out Layers and Regularization: When our neural network is very deep, then we have a lot of weights and bias parameters, due to which the model tends to overfit the data. How to fix this overfitting problem? Here, underfitting will never happen because of multiple layers of neurons present in it for training. What about overfitting?

There are 2 basic ways:

1. ***Regularization*** - L1 and L2
2. ***Drop out Layers*** - In 2014, the thesis was written by Nitish Srivastav and Jeoffery Hinton. How does drop out work?

Let us recall Random Forest, in which we use a subset of features to train each decision tree, this in turn reduces overfitting by each decision tree.

To implement drop out, we select a drop out ratio 0<=p<=1. We will be selecting a subset of features, a subset of hidden neurons and activation functions depending upon the drop out ratio ‘p’. It will randomly select certain features about that ratio and deactivate them. In backpropagation also, those which have been selected, their weights will be deactivated. Every time, with respect to this p value, features and neurons would be selected randomly.

If my training data is activated and deactivated, what about my test data? For predicting test data, everything would be activated. This probability value p will get multiplied with all the weights for the test data.

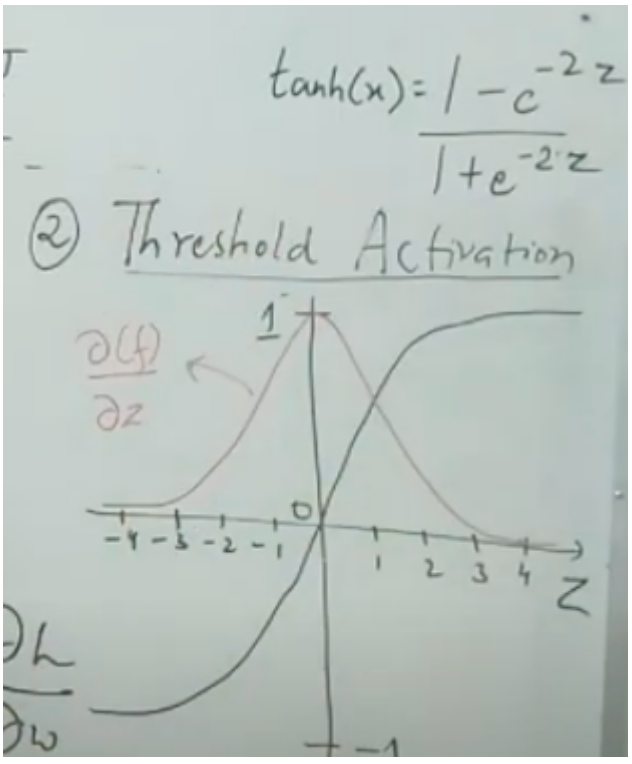
How do we select the p value?
One way is hyperparameter optimization. Whenever a deep neural network is overfitting, we should take a larger p value, maybe greater than 0.5.

RELU (Rectified Linear Unit) Activation function:

Drawbacks of other activation functions to pave way for RELU -

- Sigmoid activation function: As we studied earlier, the derivative of sigmoid would be specifically between 0 and 0.25.

- Threshold activation function:

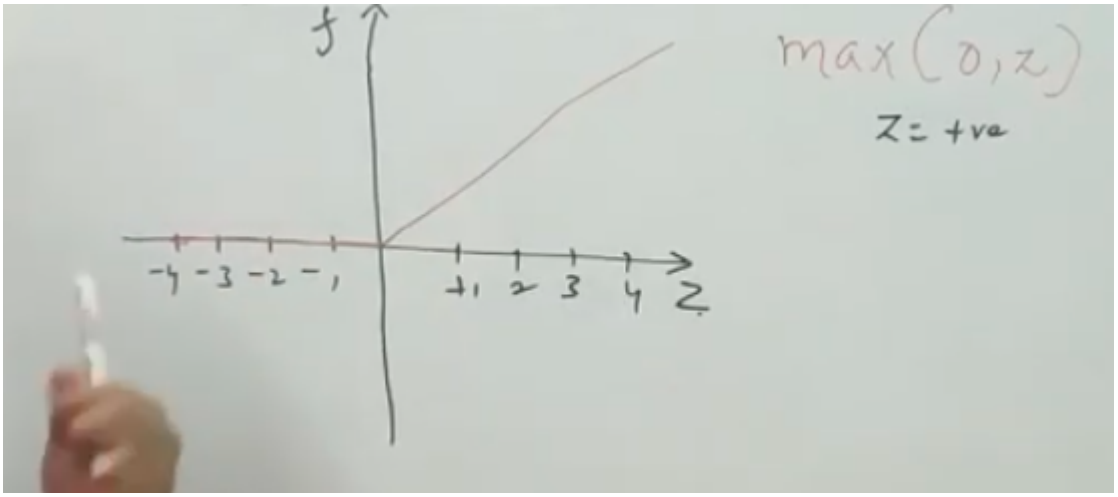


During the back propagation, the derivative of the threshold activation function ranges between 0 and 1.

Due to this constraint, the vanishing problem occurs in these activation functions.

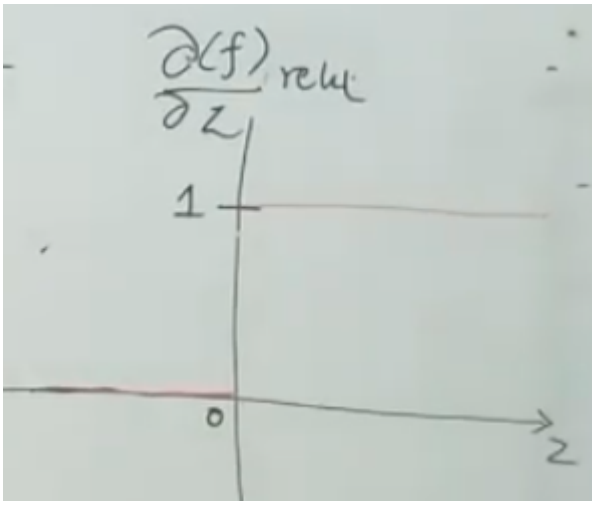
So, how does RELU work?

Equation: max(0,z)



The derivative for any point on the inclined line would always be 1. The derivative for any point / constant on the constant line would be 0. One cannot find out the derivative of 0 as it is not differentiable. This problem will be discussed shortly.

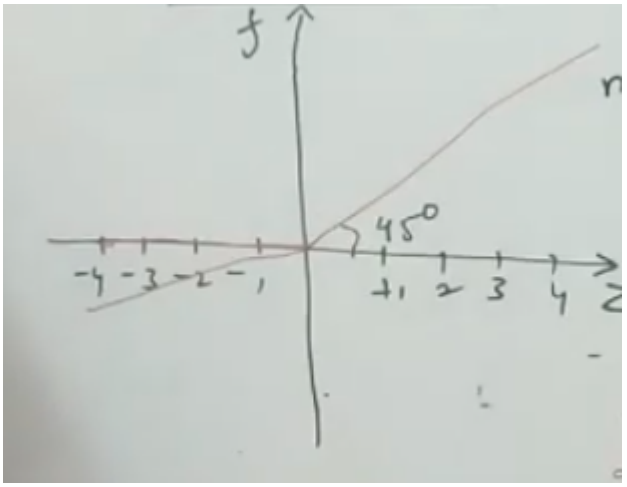
The derivative would look as follows:



But, if the derivative would be zero, then $w_{new} = w_{old}$ and the neuron would basically be a dead neuron or dead activation function which basically means no processing. We will handle this problem in leaky RELU.

How do we fix this dead neuron problem?

We will try to add some value to the constant part which leads to y = z when z>0 and 0.01(a) when z< 0



Whenever we do the derivative of the negative part, we would find that the output would be 0.01 and not 0. This is called Leaky RELU activation function.

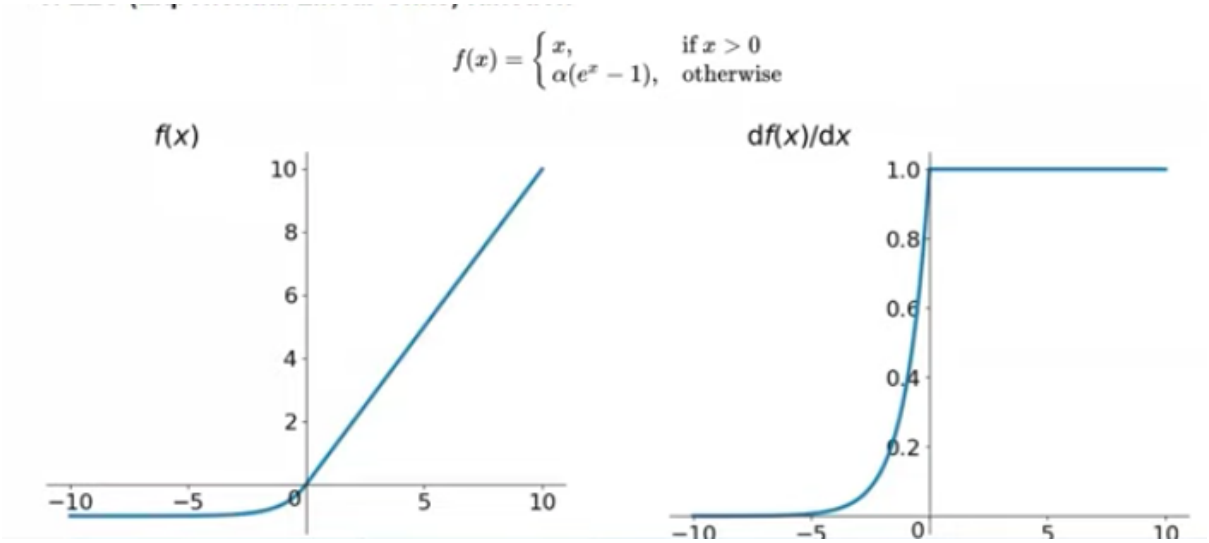
Choice of activation function:
In the last layer if it is a binary classifier, we use sigmoid and for a multi class classifier, we use Softmax activation function.

When I take a standard normal distribution data, the mean is 0 and s.d. Is 1. But, in the case of sigmoid, the data is not zero centered. This impacts the convergence time as the convergence time increases if the data is not zero centered.

For threshold function, the data is zero centered unlike the sigmoid function.

ReLU is the most common activation function used in hidden layers and it solves the vanishing gradient problem. To handle the dead neuron condition, we use leaky ReLU. To handle the zero derivative condition, again leaky ReLU can be used. But, the change in weights would then be a smaller number. If maximum weights are negative, with respect to the derivative , we would be multiplying with 0.01 several times. This will lead to a very small change in weights.

ELU (Exponential Linear Units) Function:



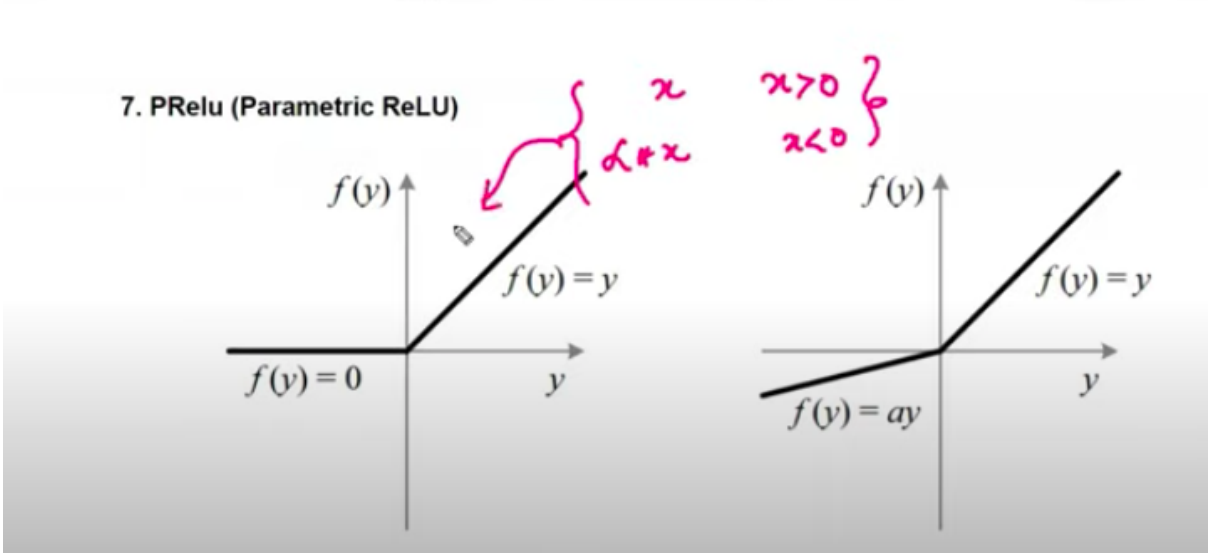
This alpha will be a learning parameter and can be treated as a hyperparameter. For a fixed alpha value, the ELU curve and its derivative will look like the ones shown in the above figure.

Here, we don't have to find out the derivative of 0 and hence one problem is solved.

ELU is proposed to solve the problems of the ReLU function. No dead ReLU issues. The mean of the output is close to 0 or zero-centered.

Since it has an exponential term, when we try to take the derivative of the function w.r.t. x during back propagation, the derivative will take some more time. It is computationally expensive. Similarly, Threshold Function Is Also Computationally Expensive.

PReLU function (Parametric ReLU):

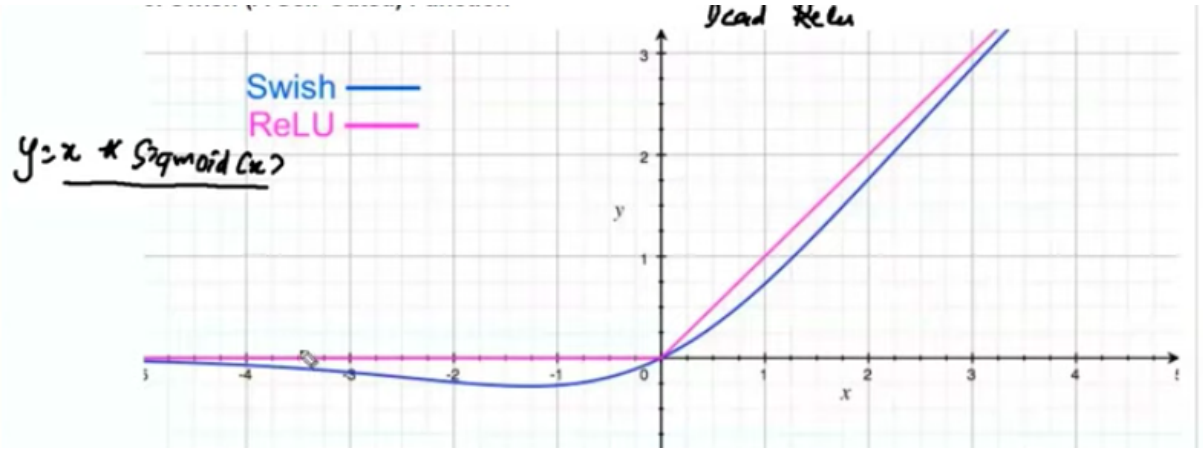


All the functionalities would be the same. If your alpha value is 0.01, this actually becomes leaky ReLU. If the alpha value is 0, it becomes a ReLU activation function.

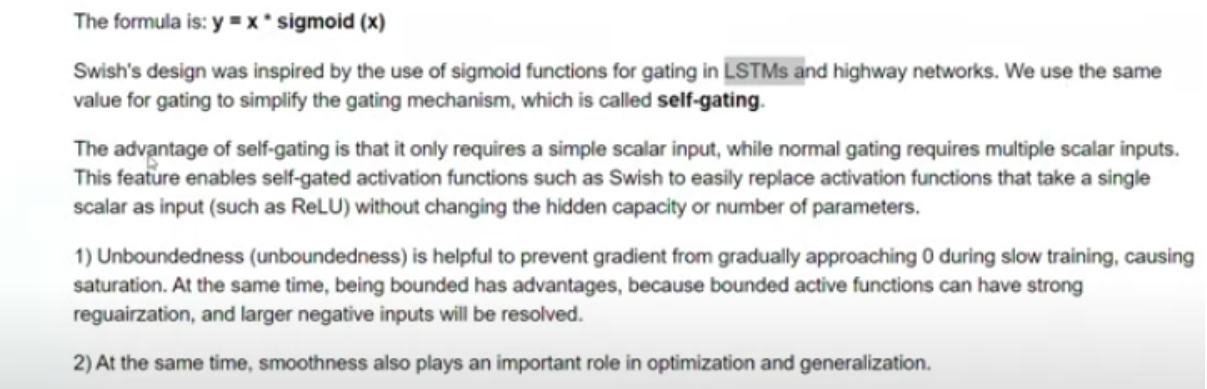
This alpha can be considered as a dynamic parameter which will change based on the activation function selected.

PReLU is an improved version of ReLU. In the negative region, it has a slow slope which also tackles dead neuron situations.

Swish ReLU:
It is used in LSTM. One uses this kind of activation function only when one has greater than 40 layers. Equation: y = x * sigmoid(x)



It also solves the dead neuron problem as it has non-zero value for the negative side.
It is called a self-gated function.


The formula is: **y = x * sigmoid (x)**

Swish's design was inspired by the use of sigmoid functions for gating in LSTMs and highway networks. We use the same value for gating to simplify the gating mechanism, which is called **self-gating**.

The advantage of self-gating is that it only requires a simple scalar input, while normal gating requires multiple scalar inputs. This feature enables self-gated activation functions such as Swish to easily replace activation functions that take a single scalar as input (such as ReLU) without changing the hidden capacity or number of parameters.

1) Unboundedness (unboundedness) is helpful to prevent gradient from gradually approaching 0 during slow training, causing saturation. At the same time, being bounded has advantages, because bounded active functions can have strong reguairzation, and larger negative inputs will be resolved.

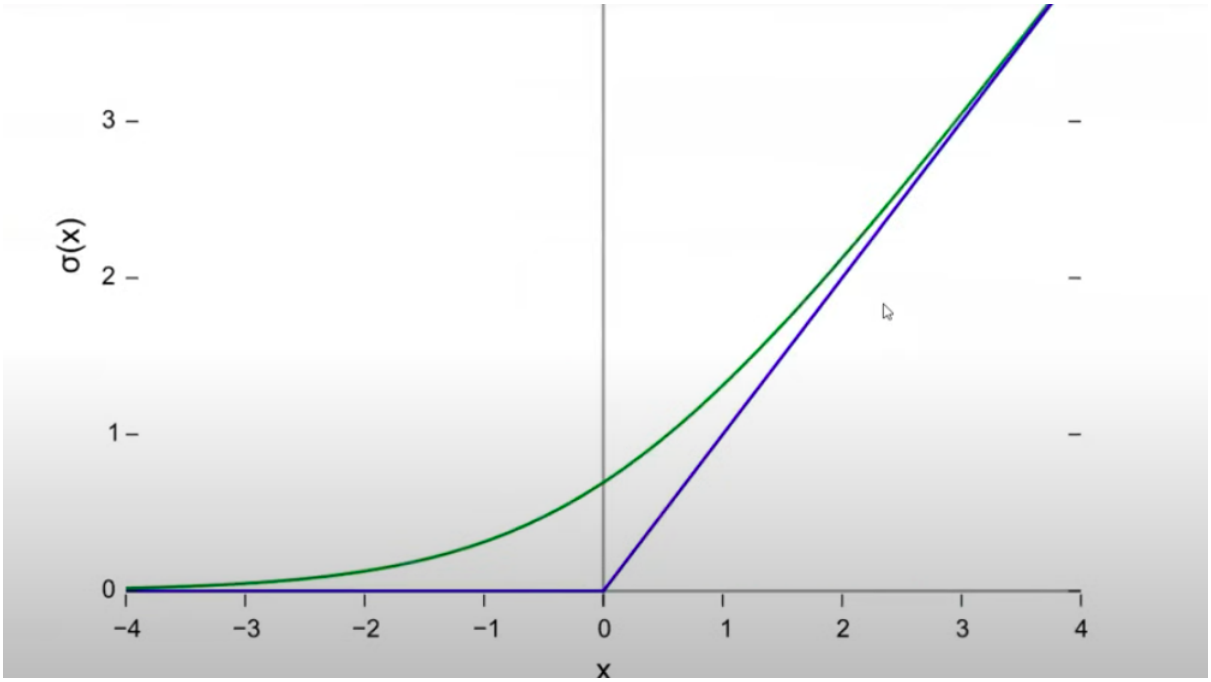
2) At the same time, smoothness also plays an important role in optimization and generalization.

Swish's design was inspired by the use of sigmoid functions for gating in LSTMs and highway networks. We use the same value for gating to simplify the gating mechanism, called self-gating.

Softplus function:
Derivative of 0 is not possible in certain scenarios. In such scenarios, using a ReLU function will not suffice. Then, in such a situation we use a softplus function.

Equation: f(x) = ln(1 + exp x)

The softplus function is similar to the ReLU function, but it is relatively smooth. It is unilateral suppression like ReLU. It has a wide acceptance range (0, inf).



Softmax Function:

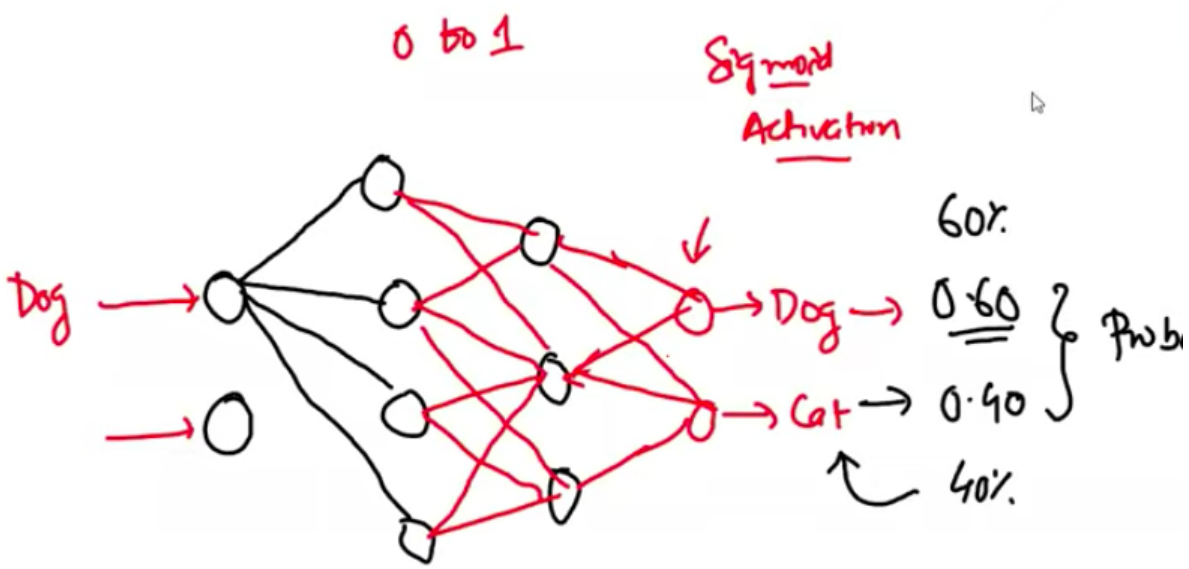
$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, j = 1, 2, \ldots, K$$

for an arbitrary real vector of length K, Softmax can compress it into a real vector of length K with a value in the range (0, 1), and the sum of the elements in the vector is 1.

It also has many applications in Multiclass Classification and neural networks. Softmax is different from the normal max function: the max function only outputs the largest value, and Softmax ensures that smaller values have a smaller probability and will not be discarded directly. It is a "max" that is "soft".

The denominator of the Softmax function combines all factors of the original output value, which means that the different probabilities obtained by the Softmax function are related to each other. In the case of binary classification, for Sigmoid, there are:

$$p(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}$$



What if I have three output layers? I cannot use the Sigmoid activation function as it ranges from 0 to 1. In that scenario, I use Softmax activation function.

Weight Initialization:

- Weights should be small
- Weights should not be same
- Weights should have good variance

Techniques for weights' initialization:

Suppose, there are 4 inputs to a particular neuron and 2 outputs from the neuron. Let us call fan_in for inputs and fan_out for the outputs.

- Uniform Distribution: W_{ij} ~ Uniform[-1/root(fan_in) , 1/root(fan_in)] : will work very nicely with sigmoid activation function

- Xavier/Gorat distribution: will work very nicely with sigmoid activation function
  - Xavier Normal - W_{ij} ~ N(0, σ), σ = root(2/(fan_in + fan_out))
  - Xavier Uniform - W_{ij} ~ Uniform[-root(6)/root(fan_in + fan_out) , root(6)/root(fan_in + fan_out)]

- He init : works well with ReLU activation function
  - He Uniform - W_{ij} ~ Uniform[-root(6)/root(fan_in), root(6)/root(fan_in)]
  - He Normal - W_{ij} ~ N(0, σ), σ = root(2/fan_in)

Stochastic Gradient Descent and Gradient Descent:

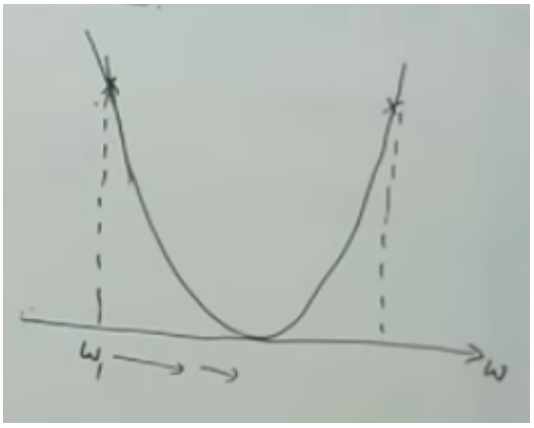
Epoch v/s Iteration:

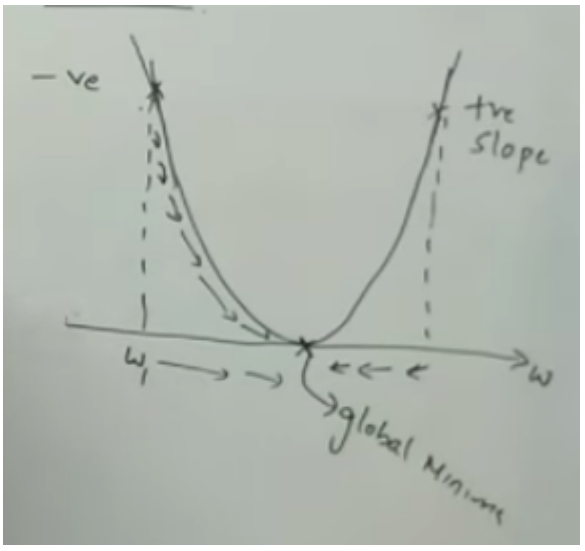
Suppose, we have in my dataset, 10K records. We have to train my model for 20 epochs. In every epoch, I will take 10K records and do updates for all the 10K records. Then, the loss function for gradient descent would be $\frac{1}{2}*\Sigma(y - y')^2$ for i = 1 to n

Iteration is the number of batches of data the algorithm has seen (or simply the number of passes the algorithm has done on the dataset). An epoch is the full pass of the training algorithm over the entire training set. Iterations per epoch = **Number of training samples ÷ MiniBatchSize** i.e. In how many iterations in an epoch the forward and backward pass takes place during training the network. Iterations = Iterations per epoch * Number of epochs.

Gradient Descent:

$w_{new} = w_{old} - \eta dL/dw_{old}$.





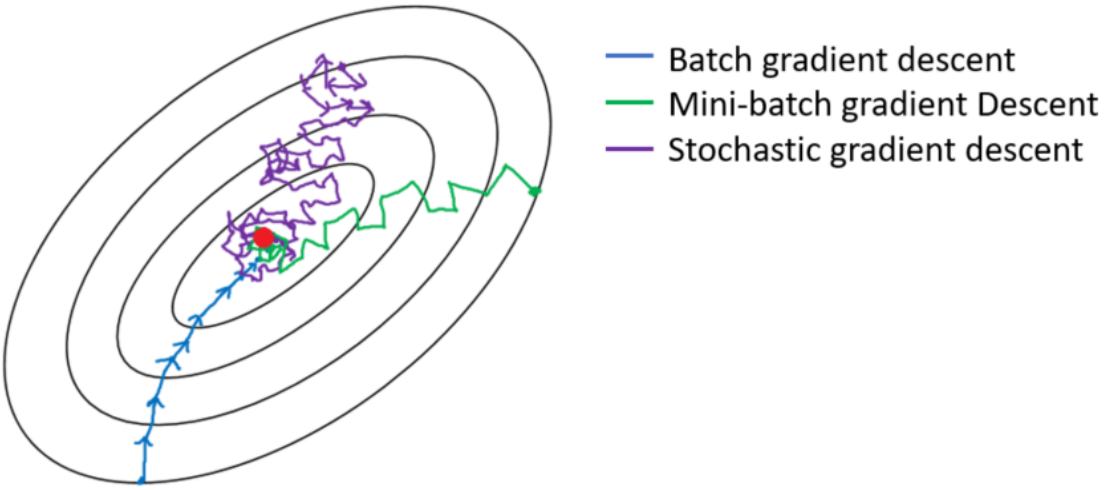
For weight updation, derivative of loss w.r.t. W_{old} → if I consider all the n data points to solve the loss derivative → the technique basically is called gradient descent.

If suppose, I am considering one point at a time for weight updation → then the technique is ***SGD (Stochastic Gradient Descent)***. If I consider k data points at a time where k < n then the technique is called ***Mini Batch SGD***.

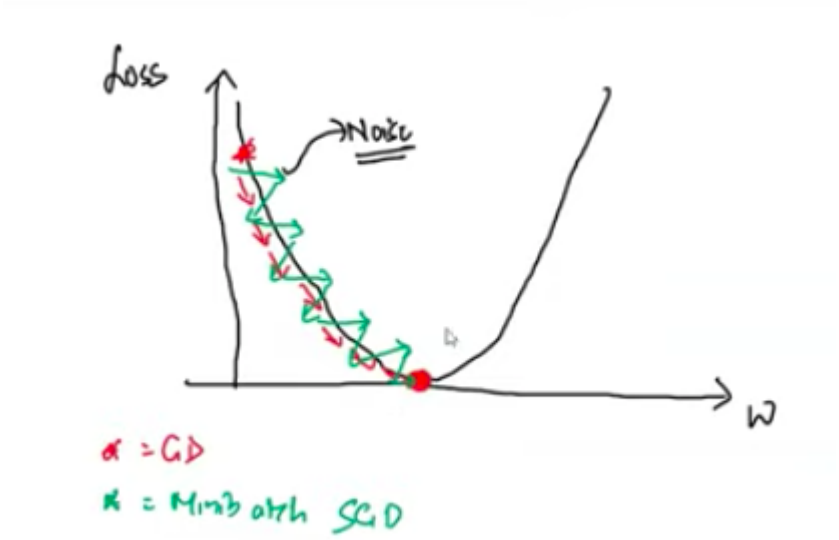
So, basically the loss function would vary as follows:

- Gradient Descent: $\Sigma(y - y')^2$ for i = 1 to n {becomes computationally expensive for a large number of records}
- Stochastic Gradient Descent: $(y - y')^2$ {for Linear Regression in machine learning}
- Mini-batch Gradient descent: $\Sigma(y - y')^2$ for i = 1 to k {for ANN, CNN}

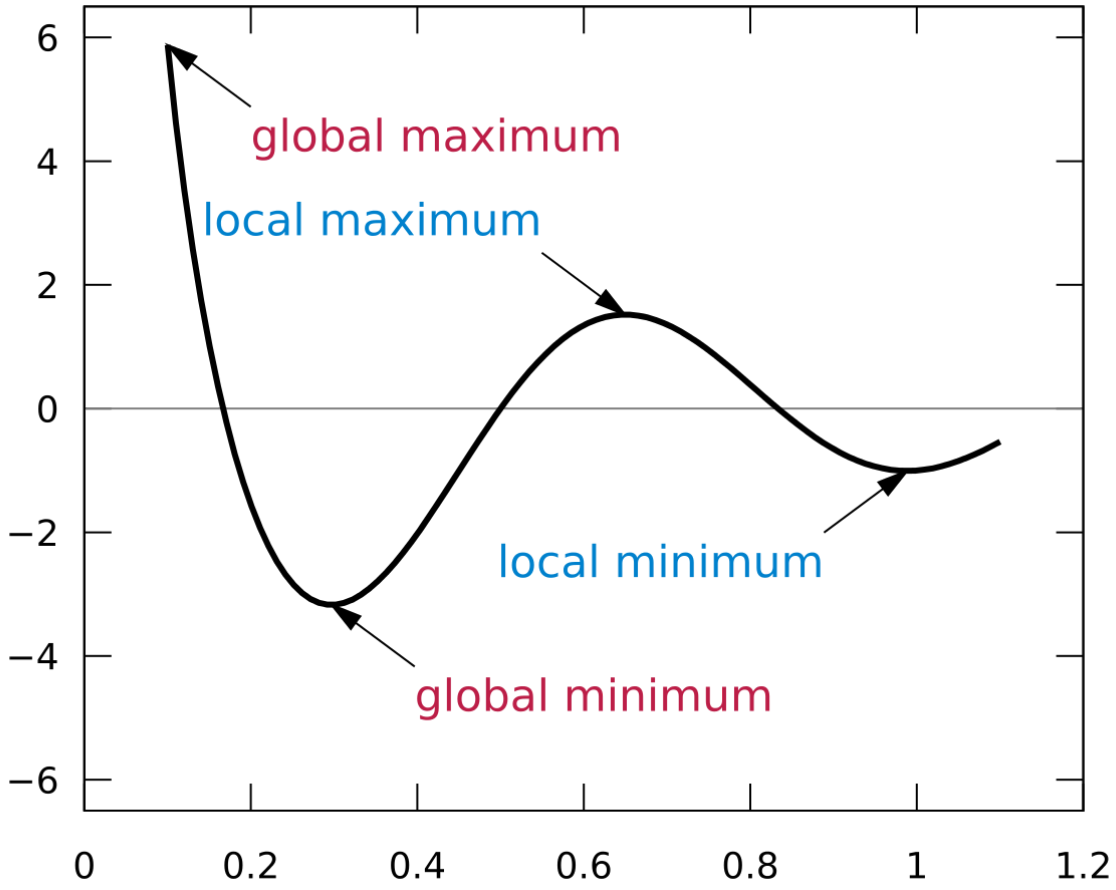
Problems if we are considering Mini-batch SGD:

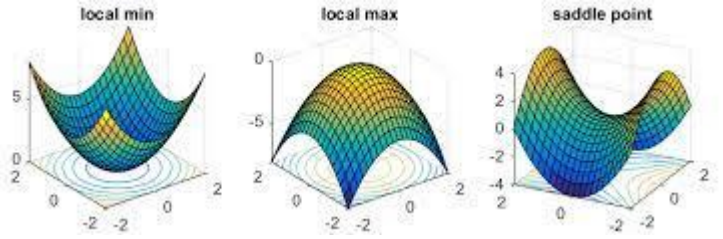


Assumption: $dL/dw_{\text{old for mini-batch SGD}} \sim dL/dw_{\text{old for GD}}$
Mini-batch SGD loss derivative is basically equivalent to sampling and the same thing for gradient descent is equivalent to population. They might be quite similar but might not be exactly equal.



Global minima vs. local minima:





- Gradient Descent: $\Sigma(y - y')^2$ for i = 1 to n
- Stochastic Gradient Descent: $(y - y')^2$
- Mini-batch Gradient descent: $\Sigma(y - y')^2$ for i = 1 to k ; we basically need to minimize this loss function. The point to which it is reduced finally is called global minima. A maxima is not present in an upward opening parabola. In the case of global minima, it will actually converge at least at one particular point, which is global minima.

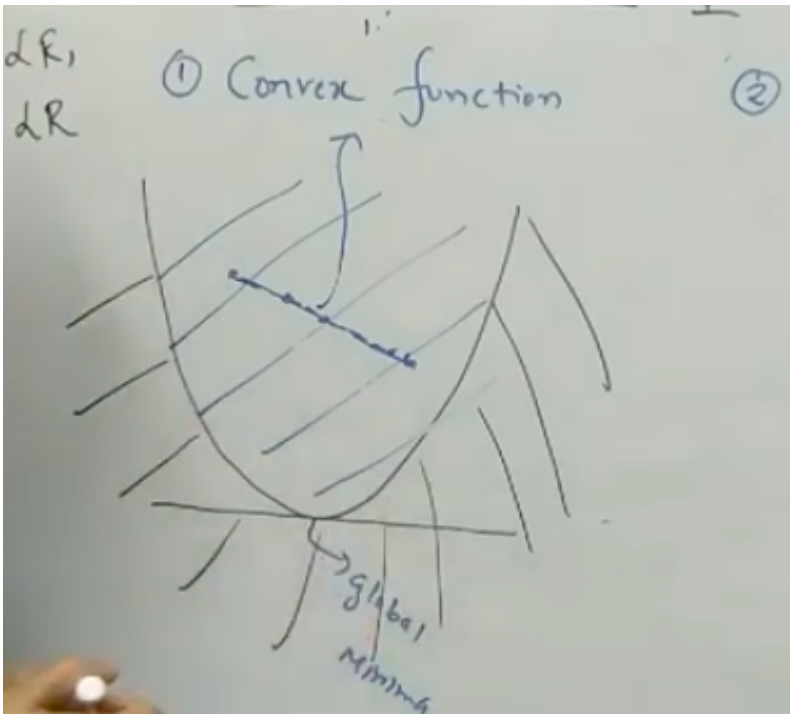
  Now, if we use a different loss function other than a mean square error loss function then we would get curves as shown above which might not resemble a parabola. The lowest point on such a curve would be a global minima and the lows other than global minima are called local minima because they are the best convergence points / minimas in that local region. On a similar note, local and global maximas are defined.

  Why are all these points important?

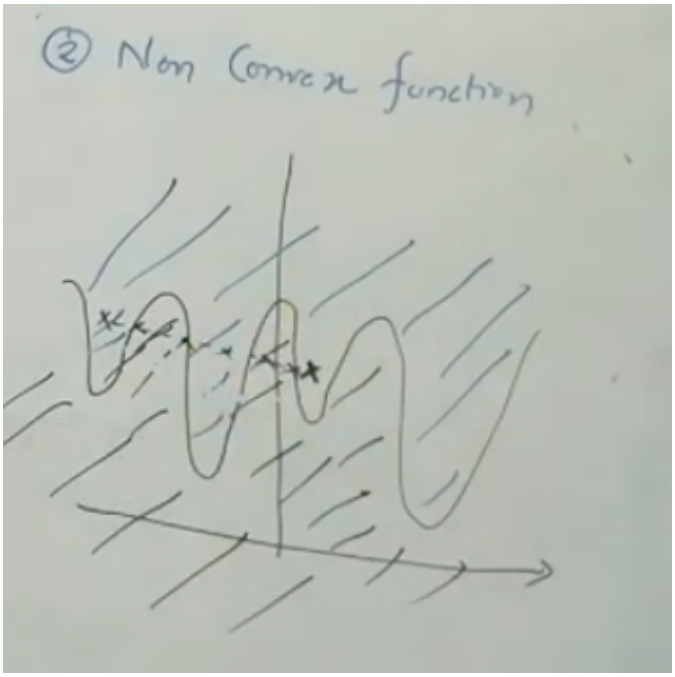
If we try to calculate the derivative of the loss function at all these points, it will be zero because the slope at this point is actually zero. Since, it is zero, there will not be any learning at these points. $w_{new} = w_{old}$

Now, the problem arises that we have still not reached our global minima and we are still at our local minima. How to tackle such a situation? We also need to handle the saddle points as the slope at these points is also zero.

First let us look at convex and non-convex functions:

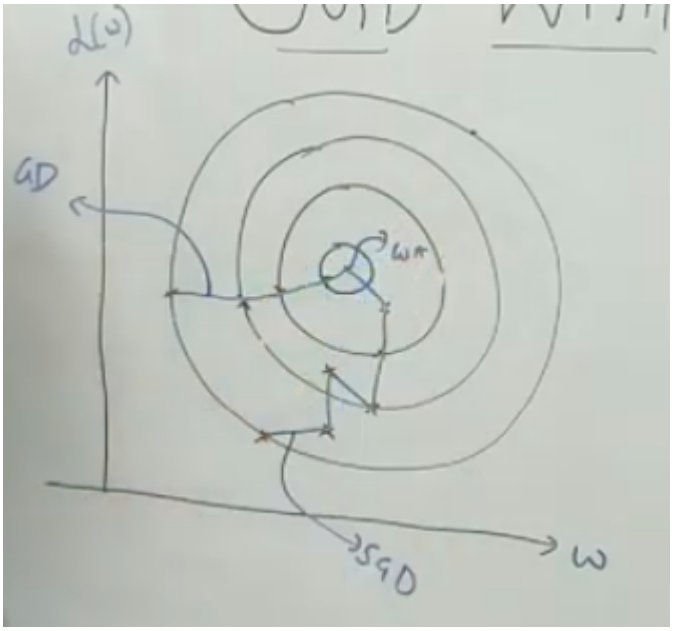


It will have exactly one minimum and would be able to divide into 2 regions.

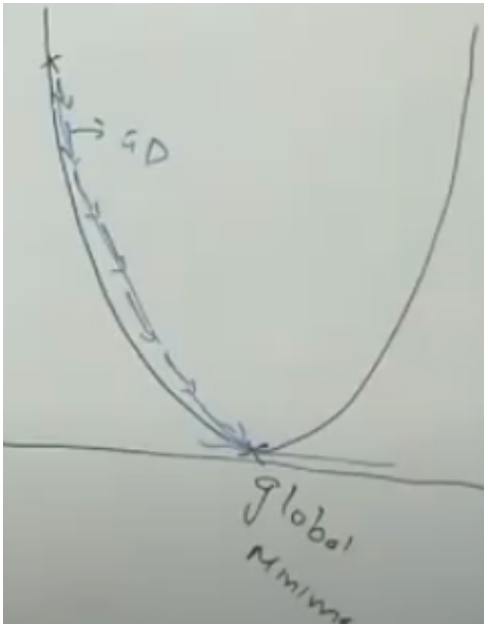


It occurs in most of the deep learning techniques. There are so many neurons being used and each neuron needs to converge. Thus, the non-convex curve occurs.

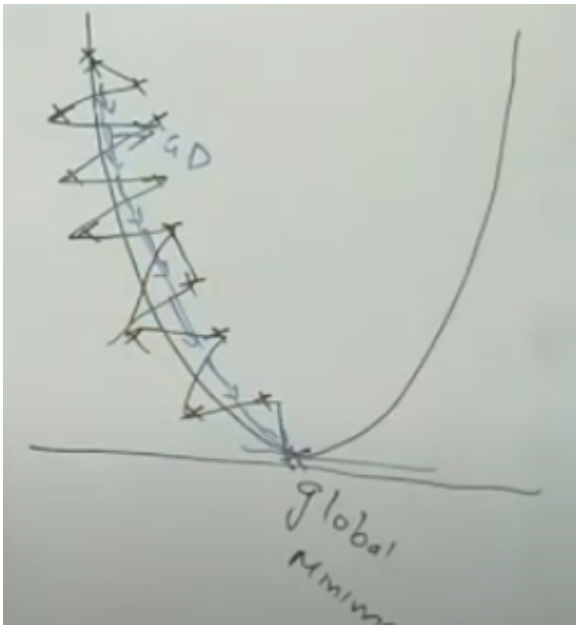
SGD with Momentum:



With SGD, there would be a lot of noise as shown in the above figure.
For Gradient Descent, with each iteration, it approaches the global minima as shown below:



If I am using SGD, then the curve looks as follows:



The problem here is that the convergence would take a lot of time and there would be a lot of noise. To tackle it , we would be using an ***exponentially moving average*** with SGD.
It is also called SGD with momentum.

We are considering various data points being generated at particular time instants, such as at time interval t1, we have a variable a1. Similarly, at t2, we have a variable a2…at t3 variable a3….and so on.

$V_{t1} = a_1$

$V_{t2} = \beta * V_{t1} + (1-\beta)a_2$ where we presume that β is 0.95. Usually, $0<=\beta<=1$
Thus, $V_{t2} = 0.95a_1 + 0.05a_2$ here, for the next point it is given less importance.

Now, if I select β as 0.1, then $V_{t2} = 0.1a_1 + 0.9a_2$ here, for the next point it is given more importance.

Thus, here β is a hyperparameter.

Now $V_{t3} = \beta * V_{t2} + (1-\beta)a_3 = \beta*(\beta*a_1 + (1-\beta)a_2) + (1-\beta)a_3$

This is basically called exponential weighted average. That is how the moving average is calculated.

This will reduce the noise which occurs in a batch SGD.

How do we apply the same formula in our batch SGD.
$w_{new} = w_{old} - \eta dL/dw_{old}$ If I apply momentum here, then the formula becomes as shown below:

This is my momentum $\rightarrow w_t = w_{t-1} - \eta V_{dw}$

Bias updation formula $\rightarrow b_t = b_{t-1} - \eta V_{db}$

How do we compute this V_{dw}?

$V_{dw} = \beta * V_{dw(t-1)} + (1-\beta)*dL/dw_{(t-1)}$

$V_{db} = \beta * V_{db(t-1)} + (1-\beta)*dL/db_{(t-1)}$

Usually, this β value is taken as 0.95. Initially V_{dw} and V_{db} will be initialized to zero.

Adaptive Gradient Descent (Adagrad) Optimizers:

We have already discussed Gradient Descent, SGD and mini-batch SGD. These basically help to reduce the loss function by modifying the weights.

Now, in Adagrad, we basically say that w_{new} can be treated as w_t and w_{old} can be treated as w_{t-1} .

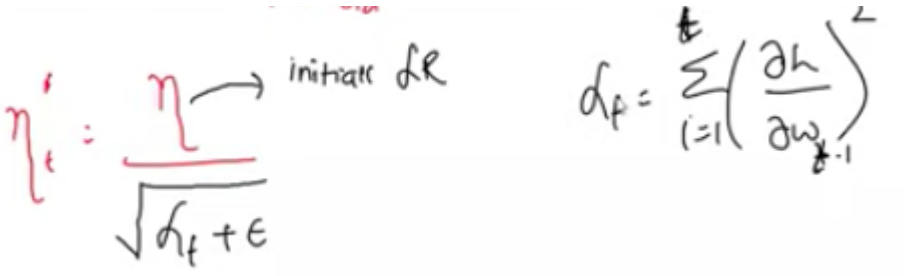
Till now the learning rate was fixed for all the optimizers. What happens if the learning rate is fixed? It will take the same sized steps to the global minima.

Can we use different learning rates for each and every neuron at each iteration?
There are basically two kinds of features: dense features and sparse features. Now, we know about the bag of words in NLP. It is an example of sparse features as many of the features here are zeroes and quite few values are non-zero. In dense features there are quite few zero values.

By using just one learning rate we cannot update the dense and sparse features correctly. So, the question comes: can we use a different learning rate at each iteration.

In case of Adagrad, the equation becomes:

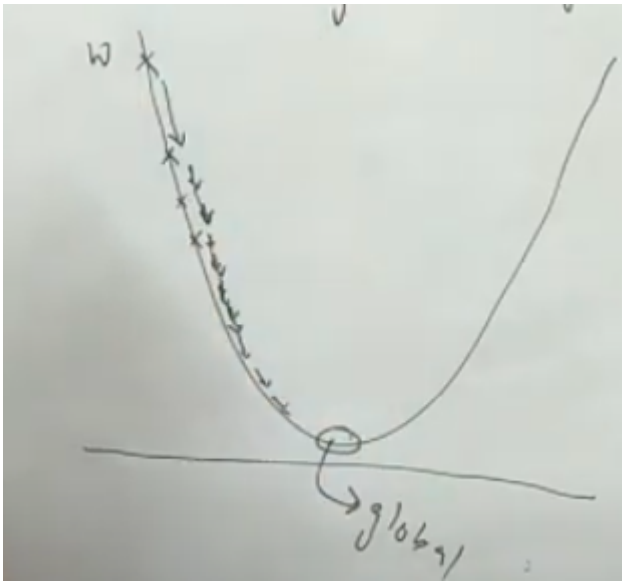
$w_t = w_{t-1} - \eta_t' dL/dw_{t-1}$ where in this learning rate value will change based on different iterations and weights.



How do we define the learning rate according to Adagrad optimizer:

Equation: $\eta_t' = \eta/root(\alpha_t + \varepsilon)$
Why do we need this ε value?
If our α_t will be zero, then our entire value of learning rate would be zero, thus our $w_t = w_{t-1}$
So, we use an ε value to keep the denominator of learning rate as non-zero.
It is always a small positive number.

What is this α_t ?
$\alpha_t = \mathbf{\Sigma}(dL/dw_i)^2$ for i = 1 to t
It basically indicates that for computing α_t , we would need to first find out the derivative of loss w.r.t . weights for previous iterations. Since we are squaring it, it will become a big number, then $\alpha_t + \varepsilon$ becomes a smaller number → η_t' becomes a smaller value as the number of iterations increases. We will be seeing that the weights will be updating slowly. Thus, we can deduce that initially the learning rate would be big and it would keep decreasing with the increasing iterations. The major use behind it is shown in the below diagram:



One disadvantage of Adagrad Optimizer is that sometimes this α_t becomes a very high number. As the number of iterations increases, sometimes it goes off limit. How do we handle it?

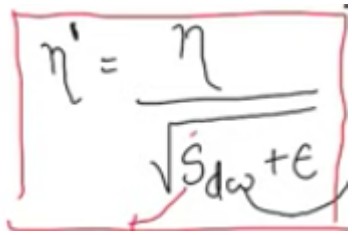
Adadelta and RMSprop (Root mean square propagation) Optimizers:

For Adagrad optimizer:

$w_t = w_{t-1} - \eta_t' dL/dw_{t-1}$ where in this learning rate value will change based on different iterations and weights.
Equation: $\eta_t' = \eta/root(\alpha_t + \varepsilon)$; $\alpha_t = \mathbf{\Sigma}(dL/dw_i)^2$ for i = 1 to t
Its major disadvantage is as the number of iterations increases, alpha increases to a quite high value.

To handle it, we basically use these two techniques: ***Adadelta and RMSprop***

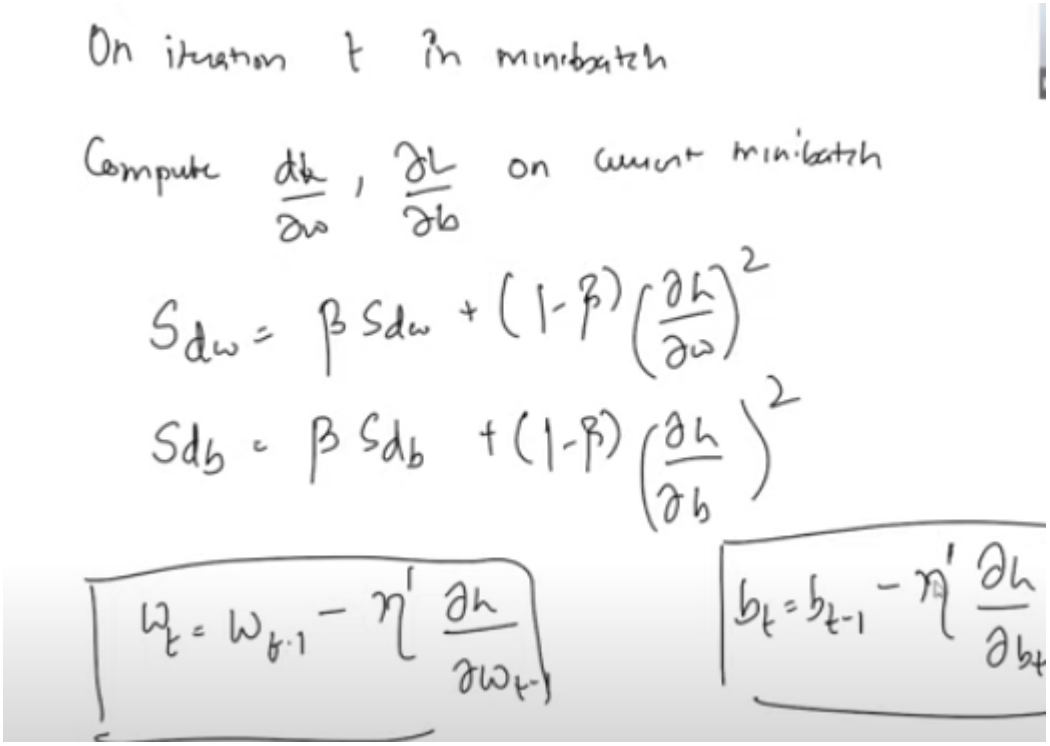
Why is α_t becoming higher because we are basically doing summation of squares of loss derivatives over i = 1 to t. If we somehow remove this square part, then the value might not increase to such high values. Some small changes are made in the above formula:



$\eta_t' = \eta/root(weightedAverage + \varepsilon)$; weightedAverage, $w_{Avg} = \beta w_{Avg(t-1)} + (1-\beta)(dL/dw_t)^2$
$w_t = w_{t-1} - \eta_t' dL/dw_{t-1}$
This learning rate will decrease very slowly.

This weighted average is also called decay average or exponential decay average. This prevents basically α_t to become very very high. Both Adadelta and RMSprop work on this principle only, it is just that they have been created by two different teams.

Implementation algorithm:



There were no problems with Adagrad and RMSprop but scientists found a better way while working with them. The next better way they found was Adam optimizer.

Let us discuss it below:

Adam Optimizer {Adaptive Moment Estimation}:

Momentum → smoothening
RMSProp → able to change the learning rate in an efficient manner

For momentum, we calculated the following:

$V_{dw} = \beta_1 * V_{dw(t-1)} + (1-\beta_1) * dL/dw_{(t-1)}$

$V_{db} = \beta_1 * V_{db(t-1)} + (1-\beta_1) * dL/db_{(t-1)}$

For RMSprop , we calculated s_{dw} and s_{db} .

Now let us go through the algorithm implementation:

1. Initialize $V_{dw}= 0$, $V_{db}= 0$, $s_{dw}= 0$,$s_{db}= 0$
2. On iteration t, compute dL/dw and dL/db using current mini batch
3. Compute V_{dw} and V_{db} using the above formulas
4. Compute s_{dw}, s_{db} using the following formulas:
   $s_{dw} = \beta_2 s_{dw} + (1-\beta_2)(dL/dw)^2$

   $s_{db} = \beta_2 s_{db} + (1-\beta_2)(dL/db)^2$

5. Bring in a new component - Bias correction. Put this computation in the weight updation formula as shown below:
   $w_t = w_{t-1} - (\eta * V_{dw} / root(s_{dw} + \varepsilon))$

   $b_t = b_{t-1} - (\eta * V_{db} / root(s_{db} + \varepsilon))$

   We basically brought in a smoothing factor and advantages of RMSprop.
   Bias correction can be applied as follows -

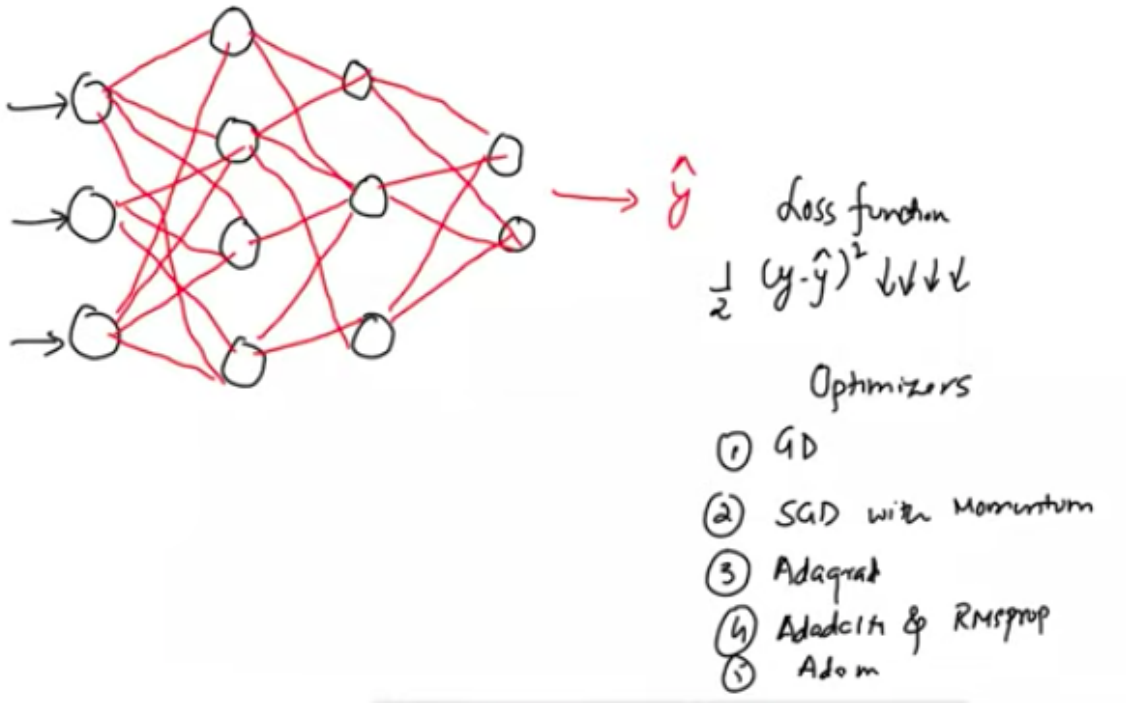
$V_{dw}^{correction} = V_{dw} / (1 - \beta_1^t)$

$V_{db}^{correction} = V_{db} / (1 - \beta_1^t)$

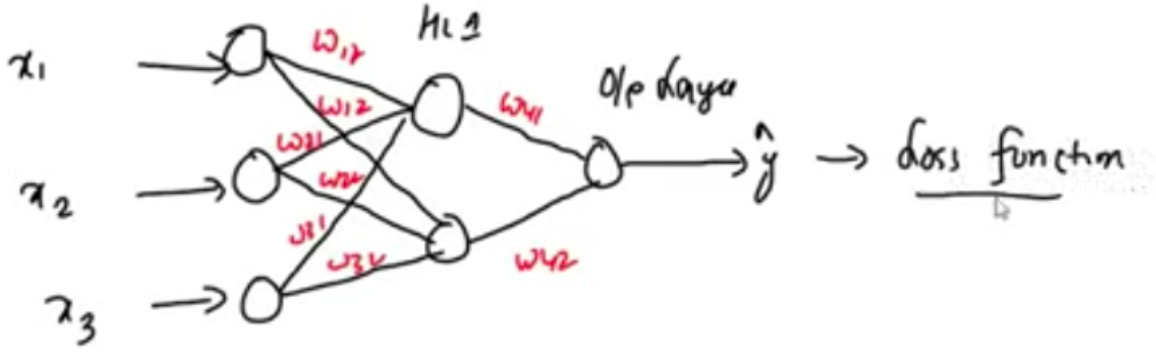
$s_{dw}^{correction} = s_{dw} / (1 - \beta_2^t)$

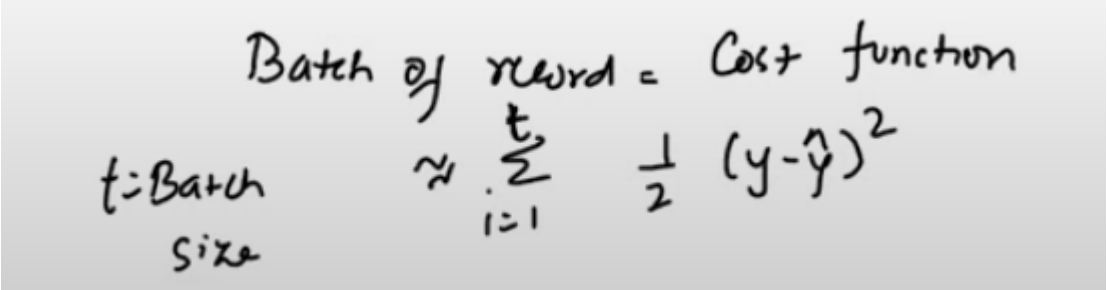
$s_{db}^{correction} = s_{db} / (1 - \beta_2^t)$

Different types of loss functions:



We solve two types of problems: classification and regression. The loss function used for these two kinds of problems are also of different types. This loss function is sometimes called an error function or cost function. If I am passing only one record at a time and then I am calculating the error each time, then we call it a loss function. For example, Stochastic Gradient Descent. On the other hand, when we give a batch of records and then we do the forward propagation and back propagation (t is my batch size) then summation of multiple errors done for t records is called a cost function.





<u>Different loss function in Regression problems</u>:

1. ***Squared Error Loss***: L = 1/n*(y - y')^2 ; T = 1/t*$\sum$(y - y')^2 for i = 1 to t where t is the batch size

   It is also called MSE (Mean squared error).

Advantages:

- This function is in the form of a quadratic equation. When we plot a quadratic equation. We get a gradient descent with only global minima.
- We don't get any local minima.
- The MSE loss penalizes the model for making large errors by squaring them.

Disadvantages:

- It is not robust to outliers.

2. ***Absolute Error Loss***: L = 1/n*|y - y'| ; T = 1/t*∑|y - y'| for i = 1 to t where t is the batch size
   It is also called MAE (Mean absolute error).

Advantages:

- The MAE is more robust to outliers as compared to MSE.

Disadvantages:

- Computation is very very difficult.
- It may have local minima.

3. ***Huber Loss***: Combine MAE and MSE. The loss is basically written as:

   L = ½*(y-y')^2 if |y - y'| <= δ ;
   L = δ|y - y'| - ½*δ^2 otherwise

   This δ value is a hyperparameter.

   This loss function solves the local minima issue that might arise in MAE. It also handles the outlier impact which occurs in MSE.

Loss functions for classification problems:

4. ***Cross Entropy***: Same loss equation as in logistic regression

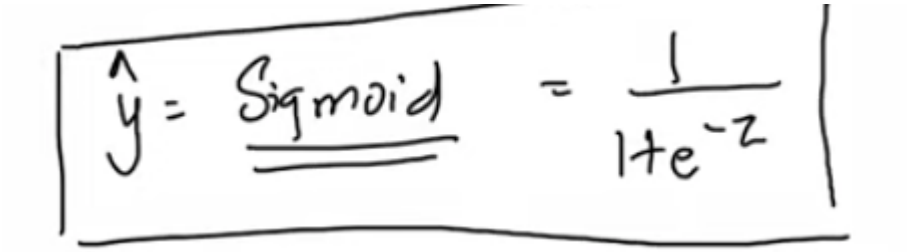
   Loss = -y*log(y') - (1-y)*log(1-y')

   This is specifically for Binary Cross Entropy.

   Let us consider if y = 0, then this equation will become:

   Loss = -log(1-y') if y = 0 ,
   Loss = -log(y') if y = 1

How do we find out y'? → By using the sigmoid activation function in the last layer.



This basically solves binary classification.

5. ***Multi Class Cross Entropy***: This is basically given by this equation –

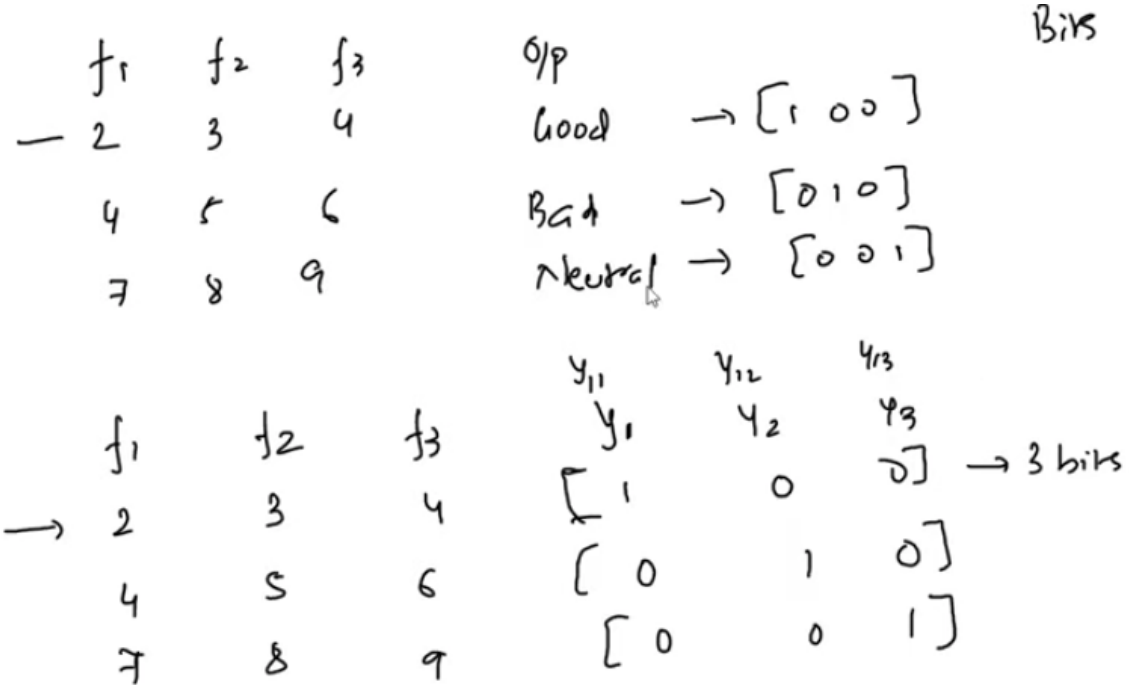
   $L(x_i , y_i) = - \sum y_{ij} * \log(y'_{ij})$ where j is from 1 to c

   The output will be multiclass.

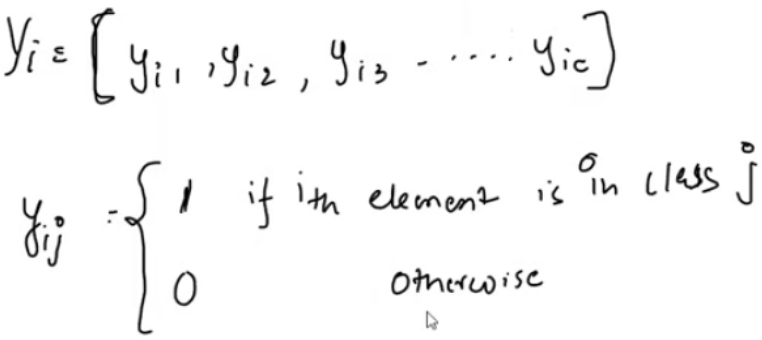
   Good, bad, Neutral → One hot encoding

   f1 f2 f3 o/p → o/p can be Good, bad, neutral

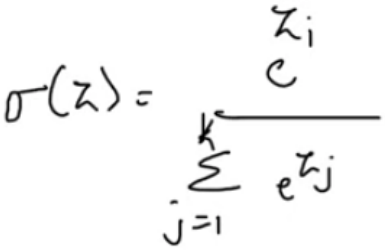
   Whenever we use this kind of cross entropy (represented by bits) → [010] then it is called categorical cross entropy.



Here, y_i is a one-hot encoded target vector. 'I' basically represents the row number.



How do we calculate y'_{ij}. We use the softmax activation function.



For the cost function, we would take the batch size.

**Can look into Keras module for the loss functions used*

Whenever you are applying ***Categorical cross entropy***, the output will be of the form [0.3 0.2 0.5]. We will be taking the highest probability category. In case of ***sparse categorical cross entropy***, then y' will be getting the maximum probability index.

Mean Absolute Percentage Error Loss, Mean absolute logarithmic error loss, cosine_similarity function etc. are other regression loss functions.

Environments – Python: 2.7, 3.5, 3.6, 3.7, 3.8, 3.9 versions
TensorFlow → Google brain → open source >= 2.0 or less than 2.0
Keras → TensorFlow >= 2.0 → Keras is integrated "wrapper" on top of tensorflow
PyTorch

Convolutional Neural Network:

Whenever one has image data → use CNN for recognition and object detection.

First we will see how image recognition is happening in the human brain?
The back part of the human brain - Cerebral cortex. In it we have a visual cortex. When we see an object, for example a dog, that information passes through our sensory organs and in this visual cortex we have layers (v1, v2, v3…). V1 layers for example are responsible for finding the edges of the object (dog). V2 determines if there is any other object and extracts information. Each layer works in the same manner as the information gets passed from one layer to another.

The V4 layer is responsible for reading the shape of the face.

What is convolution?

A grayscale image can be represented as a 4X4 matrix where each value ranges from 0 to 255. For a colored image, it will be represented by 3 layered (channels) 4X4X3 where each and every color channel would also range from 0 to 255. We can also scale these channels.

Now, let us consider a particular image which has been scaled down between 0 and 1. First half is completely white and the second half is completely black as shown below.

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

Each and every layer of the brain acts as a filter. Suppose V1 layer has this kind of filter, as shown below:

1	0	-1
2	0	-2
1	0	-1

This 3X3 matrix is a vertical edge filter.

Now, let us see how this convolution function actually works. We will get an output of 4X4. We would basically take this 3X3 matrix and place it over the 6X6 matrix.

We first multiply the first 3X3 sub-matrix with a 3X3 filter and that would be the first entry in the 4X4 matrix. We would take a stride of 1 here in the sub-matrix. Once the sub-matrix reaches the end of the horizontal line, it has to take one step down.

0	-4	-4	0

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

We can actually change all these values using min-max scalar.

255	0	0	255
255	0	0	255
255	0	0	255
255	0	0	255

The middle layer is basically your white side and the borders are the dark side. It basically is detecting the edge vertically between black and white.

Similarly, horizontal edge detector would look like this:

1	2	1
0	0	0
-1	-2	-1

Different filters would work on detecting different edges. Keras internally decides which all filters it needs based on our image size and number of filters specified.

But, how did we get a 4X4 result using a 3X3 filter on a 6X6 image?

We used the formula: n - f + 1 = 6 - 3 + 1 = 4.

But, this formula would change with the padding and stride information we provide initially. As we see that 6X6 image information got reduced to 4X4 matrix, which means some information was lost. To avoid this we specify padding.

Let us delve deeper into this.

Padding in CNN:

Now, to get a 6X6 output of the above example, n should be 8 if our filter is 3X3. How can we get a 8X8 matrix if we only have 6X6 in the input → we would use padding.

P = 1 → it basically tries to add a row above the top-most row, a row below the bottom-most row, a column to the left of the left-most column and a column to the right of the right-most column. This gives us an 8X8 image. Now, what value do we insert in these padded cells?

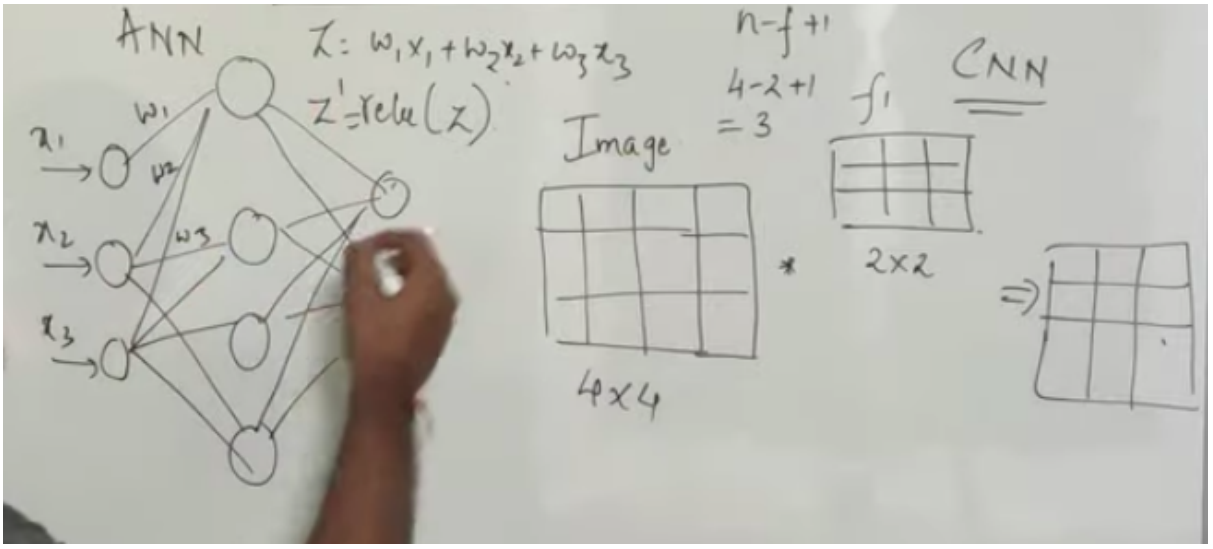
Method 1: Zero padding → just insert zeroes.
Method 2: Insert the nearest cell values.

How does the formula change if we add the padding?

The formula becomes: n + 2p - f + 1 = 6 + 2 - 3+ 1 = 6

We would be able to get more edges as there are more rows. You can continue to create a number of convolution operations.

Operation of CNN vs. ANN:



In ANN, we find the output and use backpropagation for improving the model and learning from the output matching. Same concept is applied in CNN. Once, we get a particular output, we apply ReLU activation function on each output field, and we finally improve the model using this process and get the output. We can also stack the convolution operations horizontally. Why stack horizontally? In our brain, we have layers to work on information on different aspects. These layers are also stacked horizontally. We can stack our convolution layers in the same way.

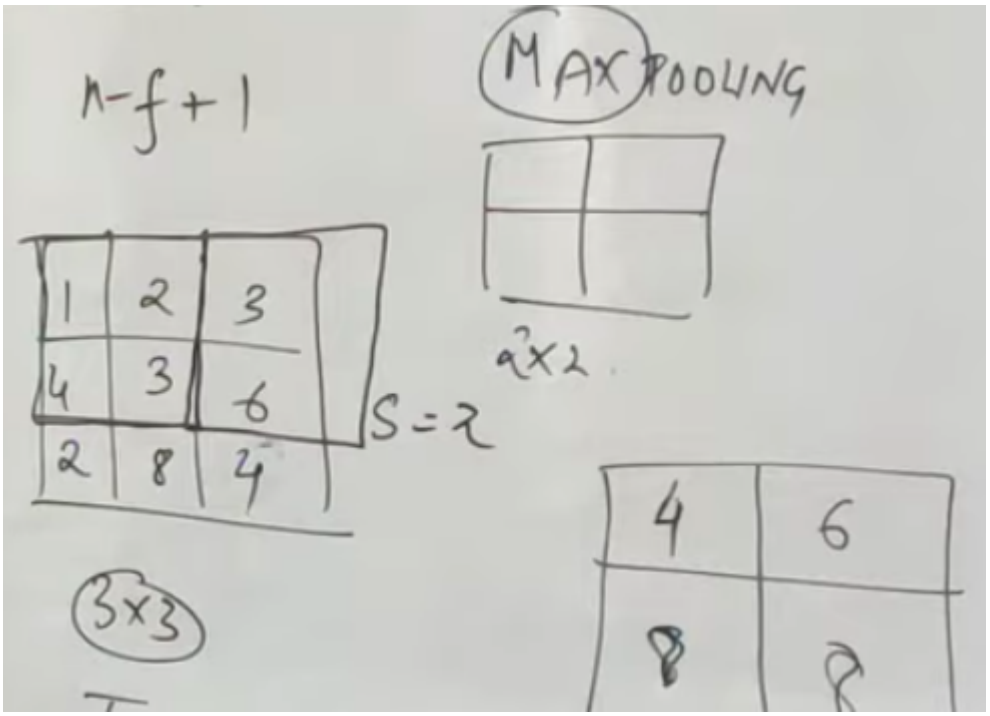
This entire one convolution layer can be stacked horizontally.
Location Invariant – Usually when our brain sees some faces, some of the neurons automatically get triggered. Suppose we have multiple dog faces in our image, so automatically this CNN should get triggered to detect these faces. This basically is done by max pooling layer. We will discuss it further in the below section.

Max Pooling:

Suppose I have a 4X4 image, we have 2X2 filter and padding, p = 0; stride, s = 1. We would get a 3X3 output as n +2p -f +1 = 3.

Suppose in this particular image, I have multiple dog images. And, suppose this is a filter that detects faces. As we go horizontally into a higher level, these faces should be able to be detected clearly. For this we use max pooling. We basically use Max pooling filters.

Initially we place it on the top of our outcome and the max value is selected from the 2X2 submatrix in the output.



The maximum intensity value is picked up from the output of the filter and is put in the result. This will help to detect the faces of dogs clearly as we go higher horizontally.

There are other pooling techniques, such as :

1. Min pooling
2. Average pooling

This can also be stacked horizontally in between the convolution layers. In transfer learning, we would see such models as well. In the backpropagation, the filter values as well as max pooling values will get updated.

Data Augmentation CNN:

$\{x_i, y_i\}$ → Cat and Dog images

One of these images go to the CNN → O/p is the label “Cat” or “Dog”

Data augmentation is some of the changes that can be applied to the data

For example, horizontal shifting, flipping, inverting vertically, zooming, adding some more noise.

The output would be the same but we would basically do different things to transform the same image using these particular properties.

Why is data augmentation important?
If I have very few images of cats and dogs, and I can use data augmentation techniques to get many more images from current data by adding some invariance. Whatever data is added, our model should be robust to detect the correct animal.

The filters would then work on these images and give good predictions by detecting edges and features.

Because of data augmentation, our model will become more robust and would be quite good in predicting the right outputs.

Create CNN Model Using Transfer Learning using Vgg 16, Resnet

ImageNet: https://en.wikipedia.org/wiki/ImageNet

You are using the state of art algorithms and you are changing the output layer and using it for solving your problem statement.

Recurrent Neural Network:

Used with NLP , spam classifier, time series forecasting.

Why exactly RNN?

If we take an example of NLP, we use techniques like BOW, TF-IDF, word2vec. My input data is text data.

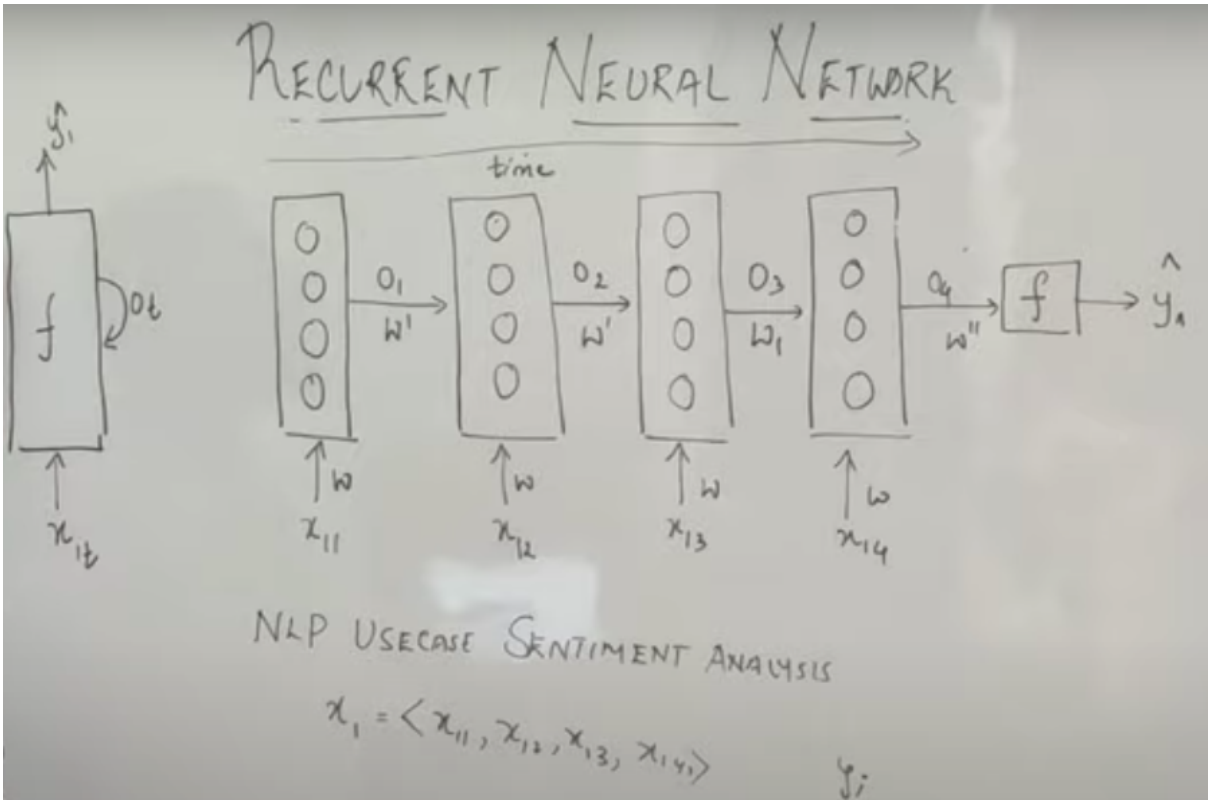
Vectors = {word1 word2 word3…..} which would be given values. Then, we apply Naive Bayes or some other classifier. The sequence information is basically discarded when we use these classifiers. Your accuracy might deflect towards the lower side.

Google assistant, Alexa etc. sequence information is very important - thus, RNN is used.

Second application of RNN - Time series
For example: sales data → predict output at a particular point. Techniques used ARIMA, SARIMAX (uses windows from some previous days). In this time series of data also, RNN would consider all previous data based on the LSTM conditions we have and predict the output at a particular time stamp.
Some of the applications - google image search, google translator (many-to-many RNN)
As we write down the sentences, it is converted keeping the sequence information intact.

Restaurant review is positive or not, Google search, understanding sarcasm etc. use RNN

Recurrent Neural Network Architecture:



Let us discuss the NLP use case -sentiment analysis engine. RNN at each instant will process one word. The preprocessing happens and the output of the first neuron will also be sent here along with the next word. Now, the word can be represented as a vector for some d-dimensional vector. Then we define weights, multiply the vector with weights and feed it to the first hidden layer at time instant t1.

The output o1 of the first hidden layer would be as follows:

$O1 = f(x_{i1}*w)$.

At t2, this output o1 would be fed to the hidden layer with the same initialized weights and next input word vector x12. Thus, $o2 = f(x_{12}*w + o1w1)$. At t3, x13 will get passed, $o3 = f(x_{13}*w + o2w1)$. Similarly, $o4 = f(x_{14}*w + o3w)$. Then, this output will go through an activation function (softmax) and we would compute our loss function (y' - y) and our main aim is to minimize this loss function.

So, we can see here, the sequence information is retained.

In the initial hidden layer also, we would add $o_{0\ input}$ (padded zeros). Thus, $o_1 = f(x_{i1}*w + o_0w')$. $o_2 = f((x_{12}*w + o_1w'))$. $o_3 = f((x_{13}*w + o_2w'))$. $o_4 = f((x_{14}*w + o_3w')$

Now, what will happen in the back propagation?

dL/dy' will be calculated and the weights will be updated. These weights can be updated in two ways:

$dL/dw'' = dL/dy' . dy'/dw''$
$dL/dw = dL/dy'.dy'/do_4 .do_4/dw$

We are basically finding the derivative and updating the weights. We will do this for several iterations, until we reach a global minimum.

One major problem with RNN -

Most of the time, we use activation functions such as sigmoid, ReLU. When the weights are getting updated at each instant during back propagation, the derivative will become a very small value due to which we will never be able to converge. This is basically called a vanishing gradient problem. If we are using an activation function such as ReLU, and our gradient or derivative is greater than 1, then it will never reach global minima. This is an exploding gradient problem. In such scenarios we use LSTM.

Suppose I have a sentence of 100 words, then during back propagation, if there is a vanishing gradient problem, there would be very less update in the weights and we might not be able to reach the global minima.

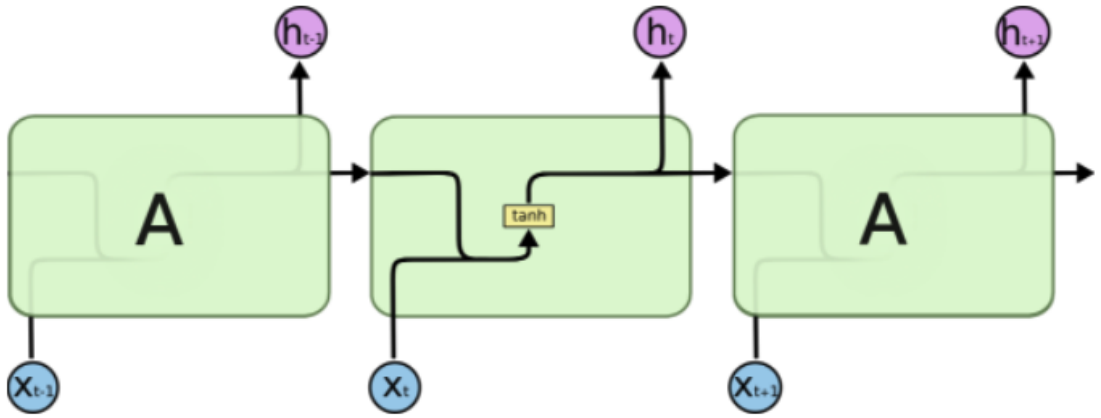
GTX GPU - Installing NVIDIA CUDA toolkit & CUDNN library Installation

1. Download cuda Toolkit 10.0 version (can be downloaded with tensorflow or PyTorch)
2. Download cuDNN (NVIDIA Deep learning library) for the same version
3. Set the path in the environment variable
4. Download Visual Studio community 2017
5. After installing visual studio, download everything for C++
6. Create an environment then install tensorflow-gpu

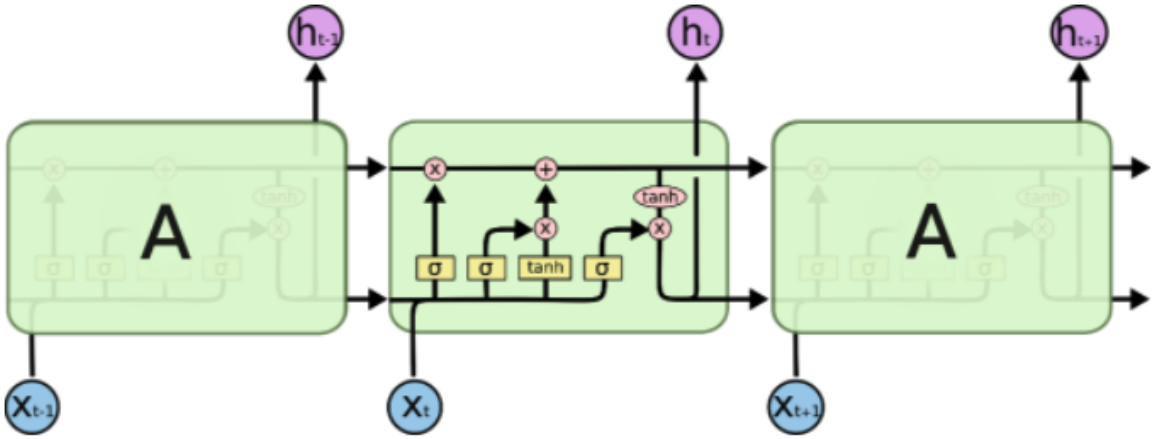
7. Install Keras

LSTM (Long Short Term Memory networks) Recurrent Neural Network:

In RNN as we discussed earlier, there is a problem of vanishing gradient at the time of back propagation due to which it never converges at a global minimum. Here, LSTM RNN comes into picture.

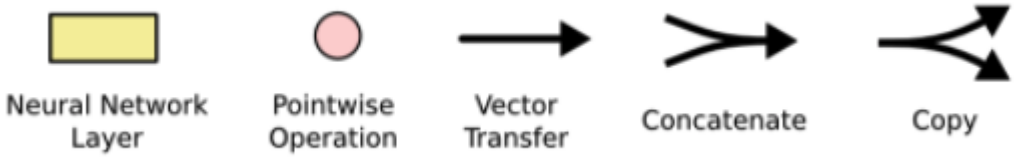


The repeating module in a standard RNN contains a single layer.



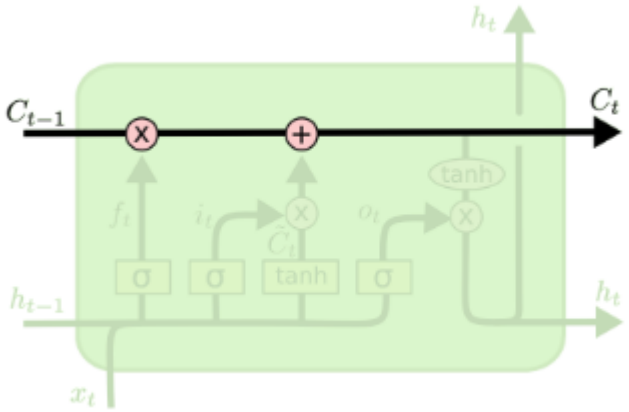
The repeating module in an LSTM contains four interacting layers.

1. Memory cell
2. Forget gate
3. Input gate
4. Output gate



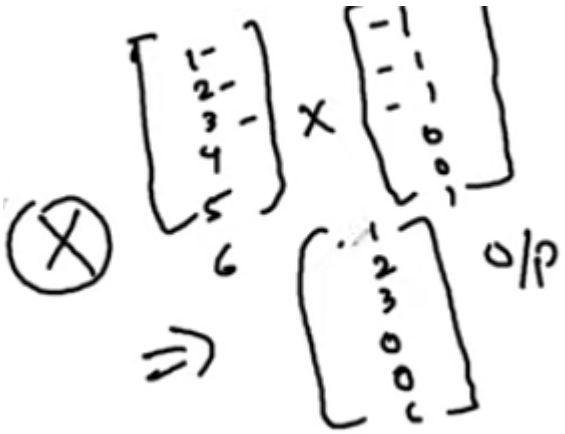
Reference: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Memory cell/Cell state is used for remembering and forgetting; how we remember or forget is based on the context of the input. Suppose, if I want to generate text.



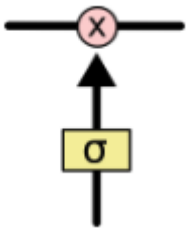
My sentence - “Hi, this is my blog. My specialization is Artificial Intelligence and Data Science.” RNN when it is training it should forget some of the previous information as the context of the information changes and it should be able to remember some of the useful information. It should be able to add some new information as well.

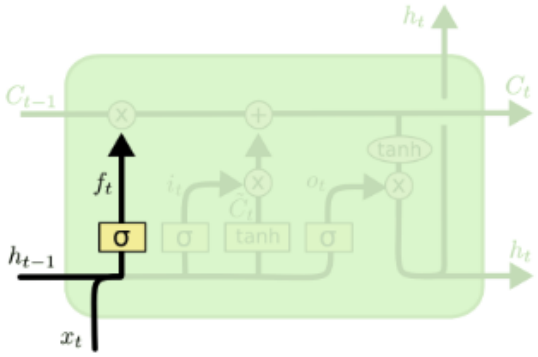
Pointwise operation - specific to location, we implement the operation, for example:



This 0, 0 in the o/p is the information forgotten basically.

Forget Gate -





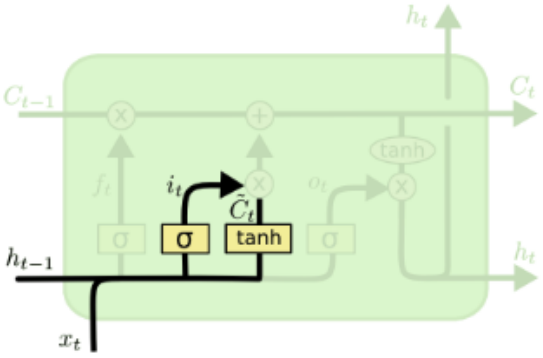
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Not similar, basically means there is a change in context. The previous vectors would be different from this particular vector.

Here, we will get more number of 1's, that means context has not changed much. Suppose, if the vector changes and we have a lot of zeros. Then, if we do many things, then it will remember some things and discard others.

Input layer:

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, Ct that could be added to the state. In the next step, we'll combine these two to create an update to the state



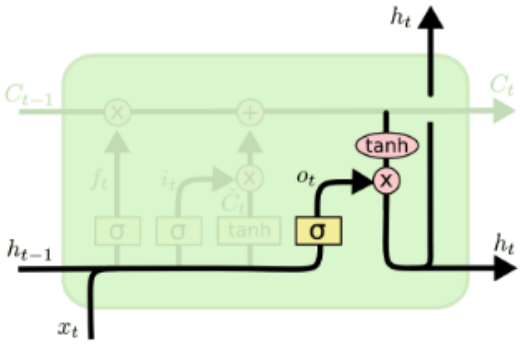
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Adding information to the memory cell.
Pointwise operation of -1 to +1
This entire layer is called an input layer.

It's now time to update the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add it∗C̃t. This is the new candidate values, scaled by how much we decided to update each state value.

Output Layer:



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

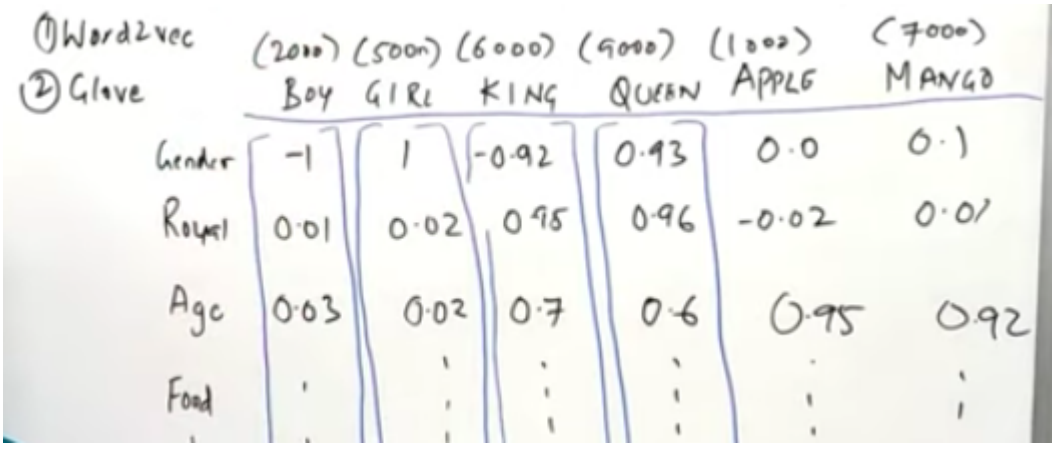
Practical applications of LSTM:

- Sequence to sequence
- Vec to sequence
- Vec to vec

Word Embedding:

We know about BagOfWords (where we do not get much semantic information); in TF-IDF we get some amount of semantic information. Here, word embedding can be of two types:

1. Word2vec
2. Glove



Converting the text data into vector representation, so that the algorithm will be able to generalize.

Suppose, I have a dictionary with 10,000 words. |v| = 10,000. Suppose a word Man is there at [5000] location. We can take a vector representation with zeros at all indices except 1 at 5000 index. Now, suppose, woman word is present at 9000 index; then all other indices in its vector representation would be zero except for 9000 index. This is basically a one-hot vector representation. Now, it is a very sparse matrix. If we are applying a deep learning algorithm or a machine learning algorithm, it is very difficult to understand it. There is not much semantic information also. Finding similarities would be difficult as well. Size of the vector is also big. The models would also not be able to give a very good result.

To handle this issue, there is a concept of word embeddings. We would try to convert the words into vectors based on some features. Now, suppose gender is a feature, it can be related to boy, girl, king , queen but has almost zero relation with apples and mango. Now, another feature - fruit will have some relation with apple and mango but no relation with boy, girl, king or queen. This way we define feature-wise representation for vectors.

We can choose to represent it as a 300 dimensional vector. This way we get a lower dimensional and dense matrix.

Similarity between two vectors can also be obtained by using cosine similarity. If I have a scenario: Boy → Girl; what will king refer to? Suppose, I consider a boy as x1 and girl as x2. If I do x1 - x2. Then, x1 - x2 = [-2, 0, 0….] Similarly, we can do King - Queen ; we would get a similar vector with other values almost equal to zero and the first value around -2. This Boy → Girl analogy based on gender corresponds to King → Queen based on gender.

We use cosine similarity, we would be able to find that this distance x1 - x2 and K-Q is very very less. Hence King → Queen as per analogy.

We will be creating these vectors based on the texts we have.

Suppose, if we reduce 300 dimensions to 2-dimensions, we would see that King and queen are near; boy and girl would be near. A word embedding basically helps to find out the most similar vectors.

To summarize, Word embeddings provide a dense representation of words and their relative meanings.They are an improvement over sparse representations used in simpler bags of word model representations.Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data.

First parameter with the help of one-hot encoding in keras is the vocabulary size. After that we get one-hot encoded representation. We will pass this whole value with respect to the embedding layer. The first parameter we give is how many dimensions we wish to consider.

Develop a Neural Network Like a Google Deep Learning Developer:

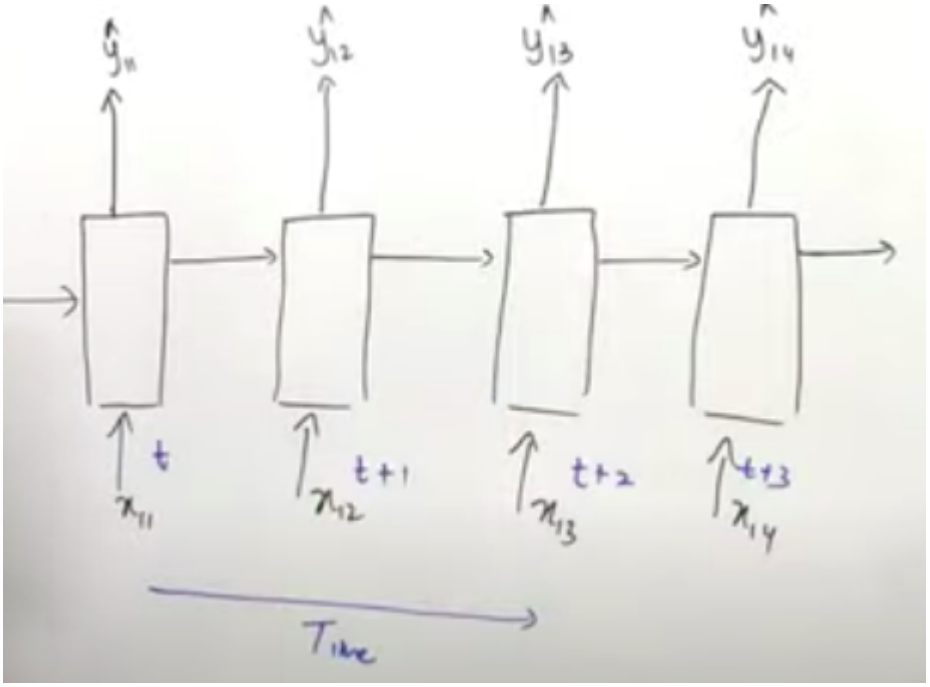
Whatever basic techniques we have learnt, we implement those basic techniques. Even if we implement transfer learning techniques, our accuracy improves slightly only. We should basically try to understand each and every layer working, for that we should try reverse engineering. But if we don’t have such powerful machines, then what should we do? Try creating a model on Google platform, then download it and we will use a tool to understand

the inside working and architecture of the model. Let us utilize this reverse engineering technique and develop a classification model with good accuracy.

First, we go to a teachable machine on google.com and create a model.

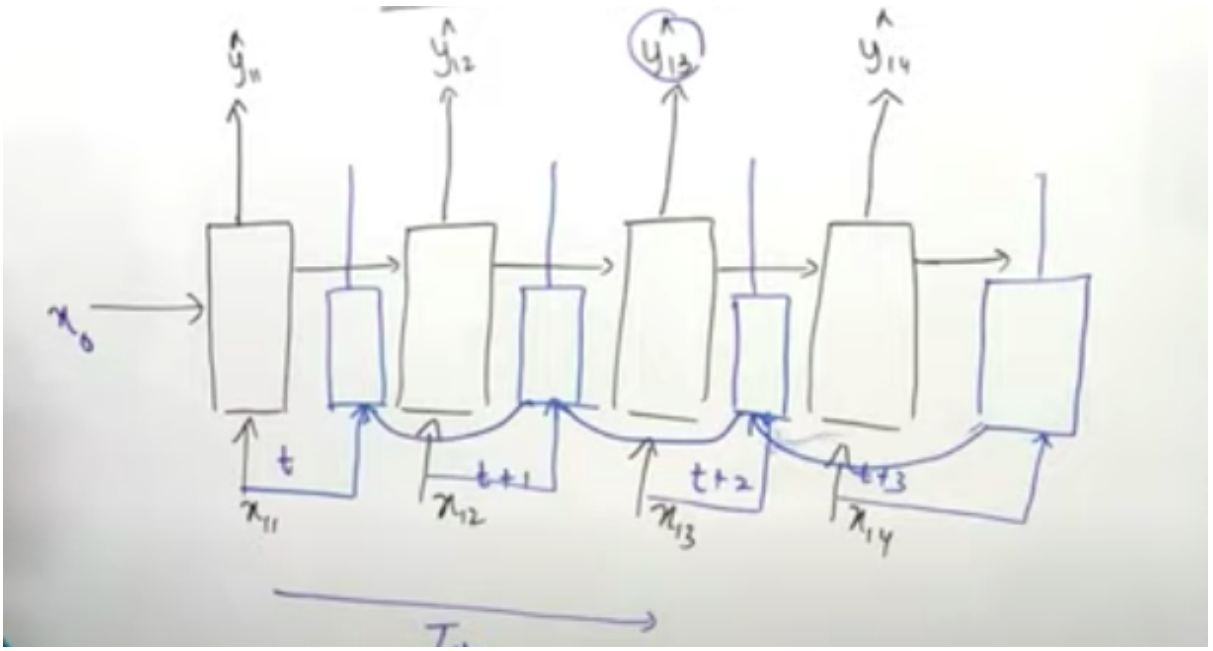
BiDirectional RNN:

Let us first consider a unidirectional RNN, as shown below:

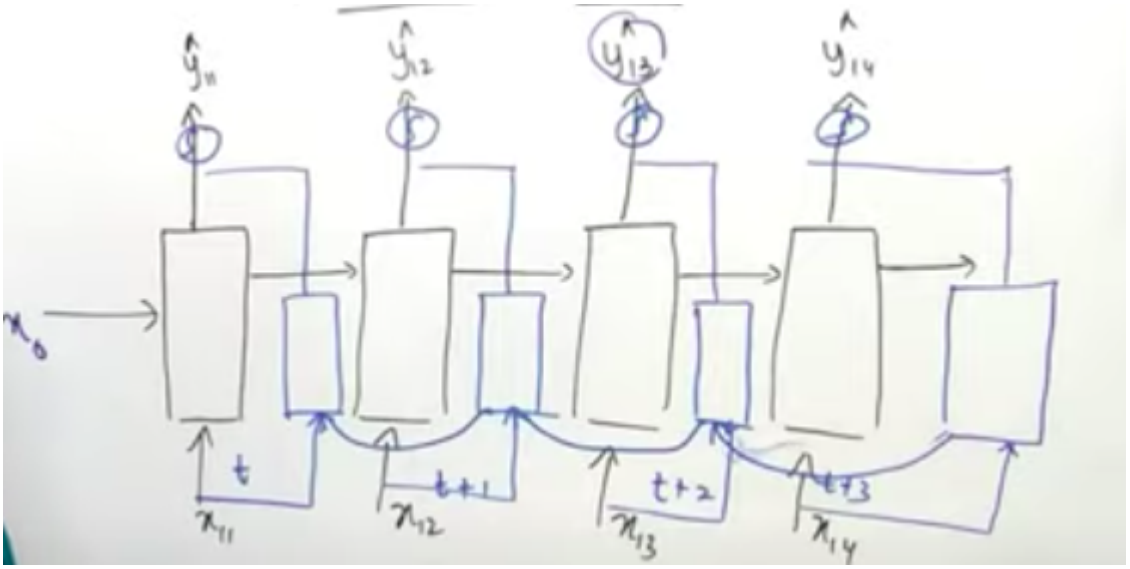


In this RNN, if we consider output y13, we need to have the information x11, x12 and x13. Now, x12 and x13 information is getting passed, so the output can be predicted easily. But, what if the output depended on the future context or words, then we need bidirectional RNN to compute the output using future words.

Basically, we call it backward directional RNN - we take the same number of hidden layers and we are going to reverse the direction and the information will get passed as shown below:



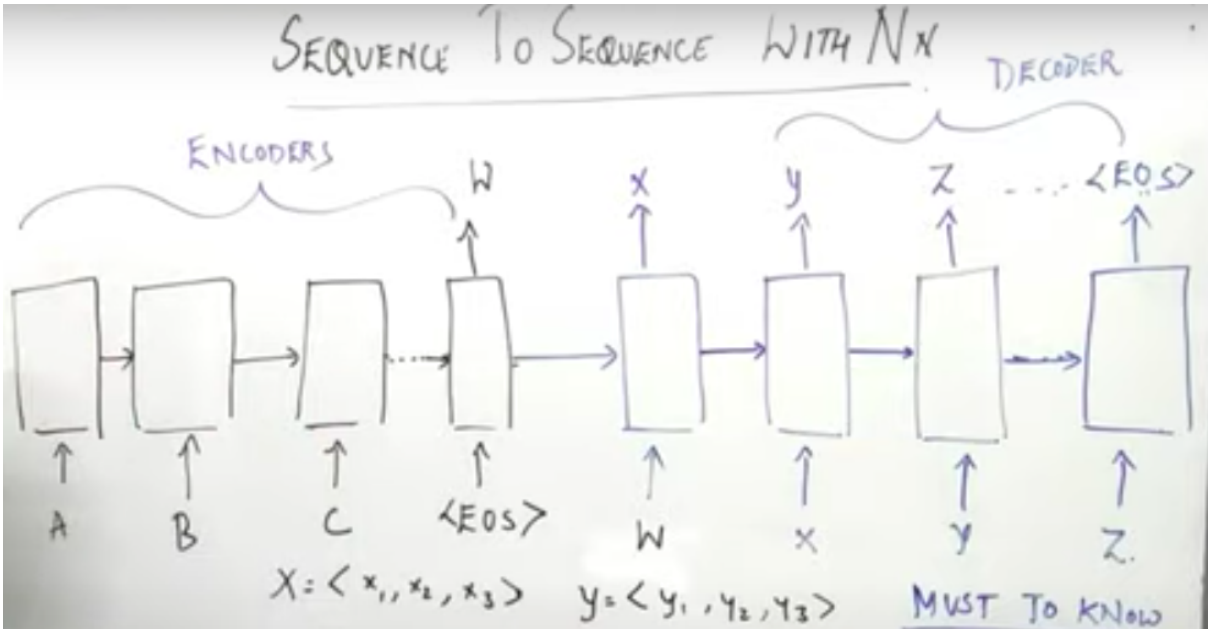
So, the inputs are connected to this reverse RNN as shown above and the output of these backward RNN gets feeded into forward RNN as shown below:



Bidirectional RNN can be used for future tasks in LSTM. It may not work well in speech recognition as we may not get the input all at once. It may give us less accuracy. It is very slow when compared to a regular RNN.

Sequence to sequence with Neural Networks: *How encoder and decoders actually work? What are attention models? What is sequence to sequence with Neural networks?*

If a model takes in some sequence of input and gives another sequence of output; for example, Language translation from English to French, Image captioning (Image to text); Google Image Search, Automated search and chat reply.



Basic things to know before going for encoders and decoders?

1. RNN
2. LSTM
3. GRU

How does sequence to sequence actually work?

Two main components:

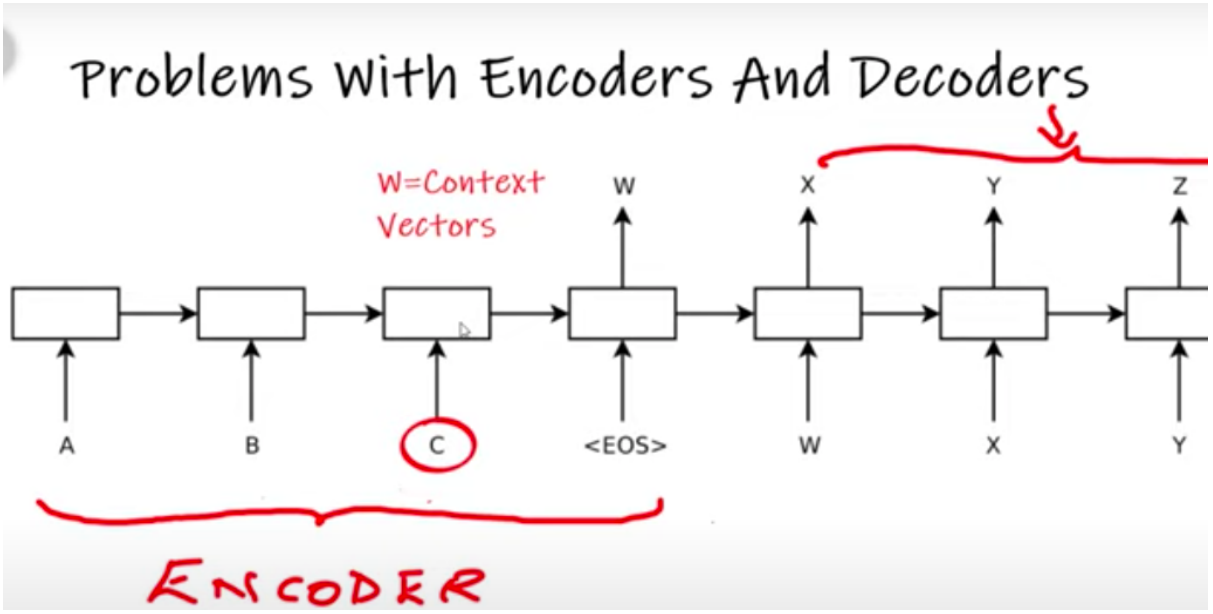
1. Encoders
2. Decoders

Suppose I talk about Language translation, in encoders basically output will not be present and it will be present at the last time stamp (End of stream). This will give an output (context vector). To get such a vector,we can use techniques such as Embedding layer, One hot representation or Word2vec. All the operations of Encoder would be similar to that of LSTM, just that none of the neural network layers would have the output but only the last EOS layer would have that. Suppose input is my english characters and output is french characters. We will get some context vector representing this input and this would be passed to the decoder as an input. With each time stamp sequential outputs would be generated and passed in decoders. This is how we would train the model. So, finally we will get EOS (End of stream) from the decoder. The input sequence and output sequence might be quite different in size. But, When I Am Doing Prediction for unseen data, I would basically get the probabilities similar to that we get in LSTM (Pr(y' |<x1,x2, x3, x4…..xn>) {i.e. Probability of y' given the inputs}. Then, I would compute the loss between y and y'. For this we will use an optimizer that will reduce the loss through back propagation. Mostly Adam optimizer is used.

Now, let us take another example: Image Captioning; here instead of encoder, we would have advanced CNN / Resnet and the last flattened layer can be taken as the context vector.

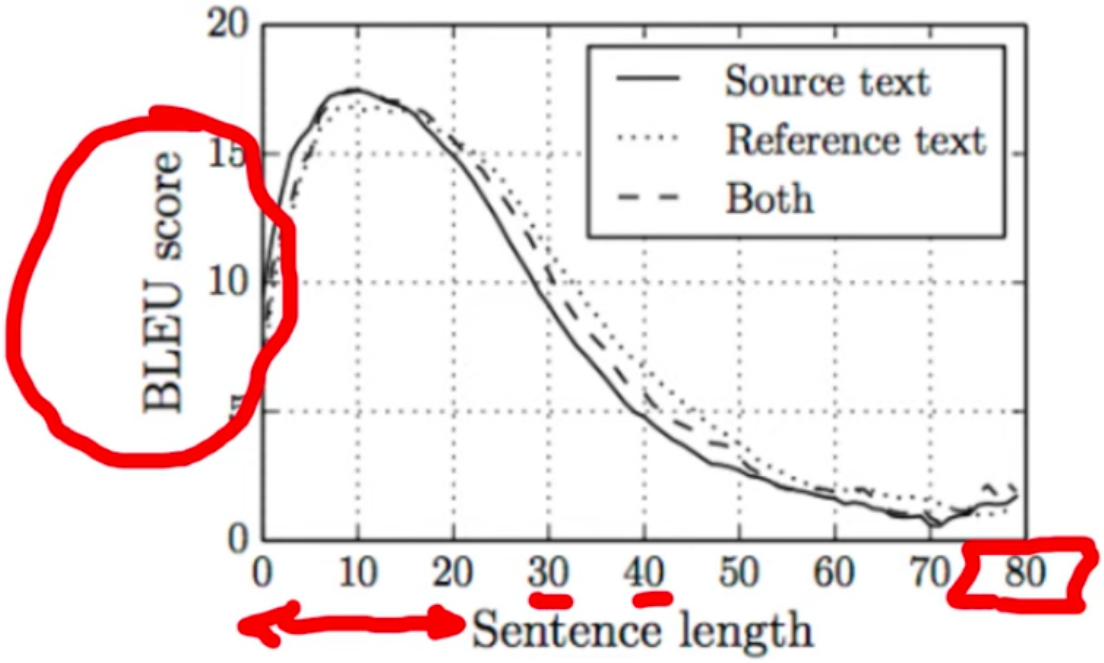
One disadvantage is whenever you have long sentences, the accuracy is not that good. There is a score called blue score. To handle this we would use a different kind of model called attention model in which instead of LSTM we will use Bidirectional LSTM model.

Problems with Encoders and Decoders:



Suppose if I was translating English to French, it was translating very well up to 100 words but was performing poorly for longer sentences.

Bleu Score



Loss in model skill with increased sentence length.
Taken from "On the Properties of Neural Machine Translation: Encoder-Decoder

A, B, C are all words inputs in the encoder shown above, we do not consider the output of each encoder layer. We only consider the states to generate w - context vectors. Now, if the sentences are quite long, then it does not retain the entire information, w - vector is not able to represent the entire input to the decoder and the translation is poor. It is quite similar to a human being told to memorize english sentences and convert them to french. It will not be able to capture the essence of all the words. Researchers experimented and found out a lower Bleu score. There are two major reasons -

a. We are taking output from just the last output layer from Encoder RNN. Thus, the last few inputs are highlighted more or have more weights.
b. It is not able to represent the whole information from the long sentence. It will have more context to the nearest words.

How can we actually solve it?

We use attention models.

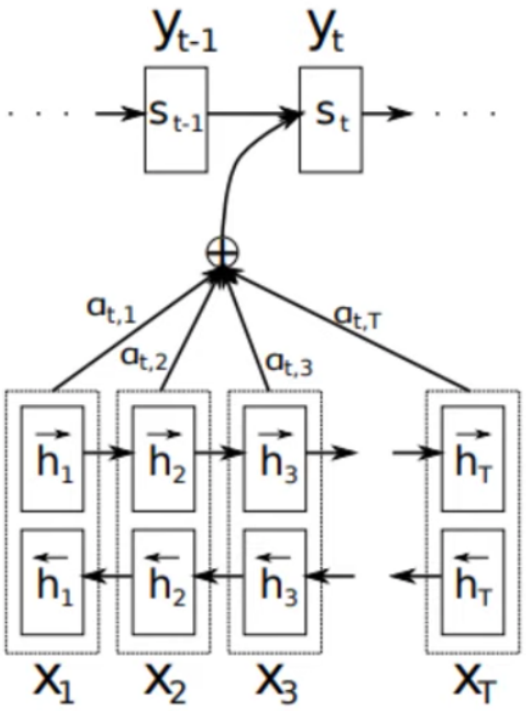
Imagine how human beings will translate. They might take a certain window-size of words and memorize - translate then memorize the next window of the same-size. This is the basic essence of attention models. It is basically a bi-directional model instead of a unidirectional model.

Attention models utilize bidirectional LSTM RNN.

Research Paper: https://arxiv.org/pdf/1409.0473.pdf

Attention Models:

The fix for the above problem with encoders-decoders is quite simple - to use bidirectional LSTM, RNN

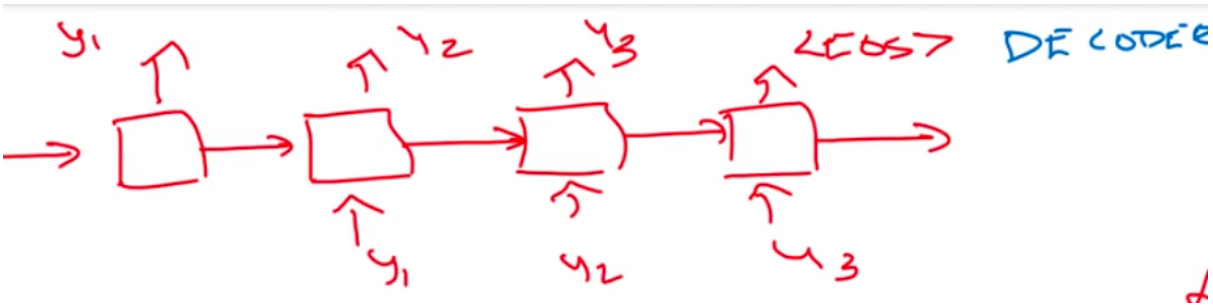


<--- ENCODER

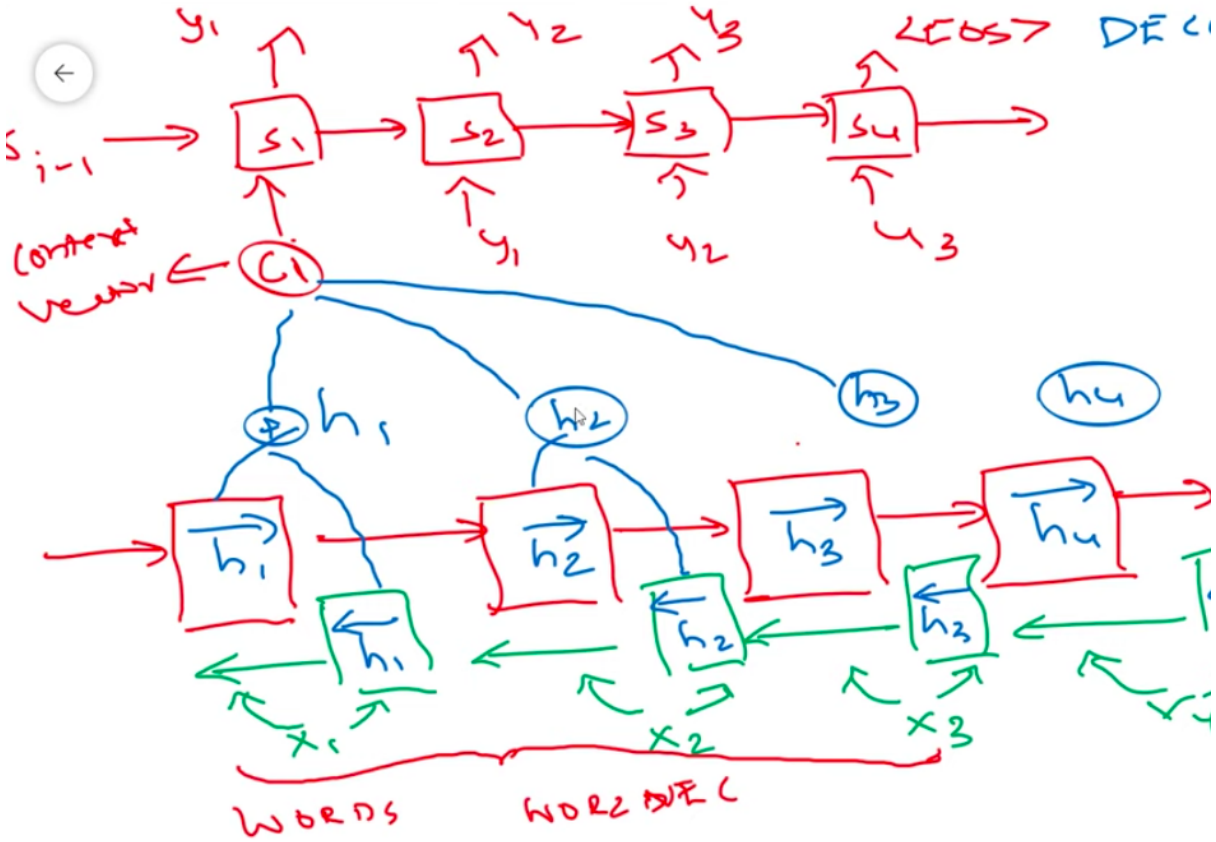
Instead of having one LSTM RNN, we will have two LSTM nodes. The information will be going in the opposite direction for the two LSTM.

Both the outputs of LSTM nodes would get combined, pass through an activation function and would give y' output. The input will get passed to both the nodes. The use of bidirectional RNN is basically to consider the future word information on which the translation output might be dependent. That is how we are fixing one problem by getting the future context of the whole information. This change is made in the encoder.

There would be one minor change for the decoder. In the decoder model, we would have a simple LSTM RNN only with just a minute change.



We have to consider a window: t(x). This window can be decided as a hyperparameter optimization. ‘s’ - state vector for each time state .



This context vector c1 is derived from h1, h2 and h3. And, the decoder finally gives output y1. This y1 may be dependent on x2 and/or x3 but it still is getting the indirect inputs from x2 and/x3 as needed through h1 as the encoder is a bidirectional LSTM.

$\alpha_{t,1}$ indicated that this alpha value is in context with time instant t and LSTM 1. These alphas are weights that have been initialized. The sum of all the alphas within a time frame (for example at an instance t = 3 ; window size is 3, we are using alpha1, alpha2, alpha3 → their sum should be equal to 1 when initialized randomly)

Basically, we are giving attention to some of the words based on the tx parameter. Then, we are translating it to a sentence in another language.

Coming to the math, there are two more conditions – how are these alpha values getting initialized? The sum of alpha values at any time instant should be equal to 1.

The context vector is being generated by multiplying the output of the bidirectional encoder layer with α. Then, it is summed over i.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

This is the characteristic of a feedforward neural network. That means I can also change the alpha value by back propagation.

It is quite similar to ANN where the inputs are h1, h2 etc. and the previous output of decoder. The weights are alpha1, alpha2, etc. The weights are computed by a softmax function such that the sum of all alphas at any time is 1. The output is c which is the input to the decoder that yields the output y'. The scope of Tx is that it will let you define the length of the timestamp.

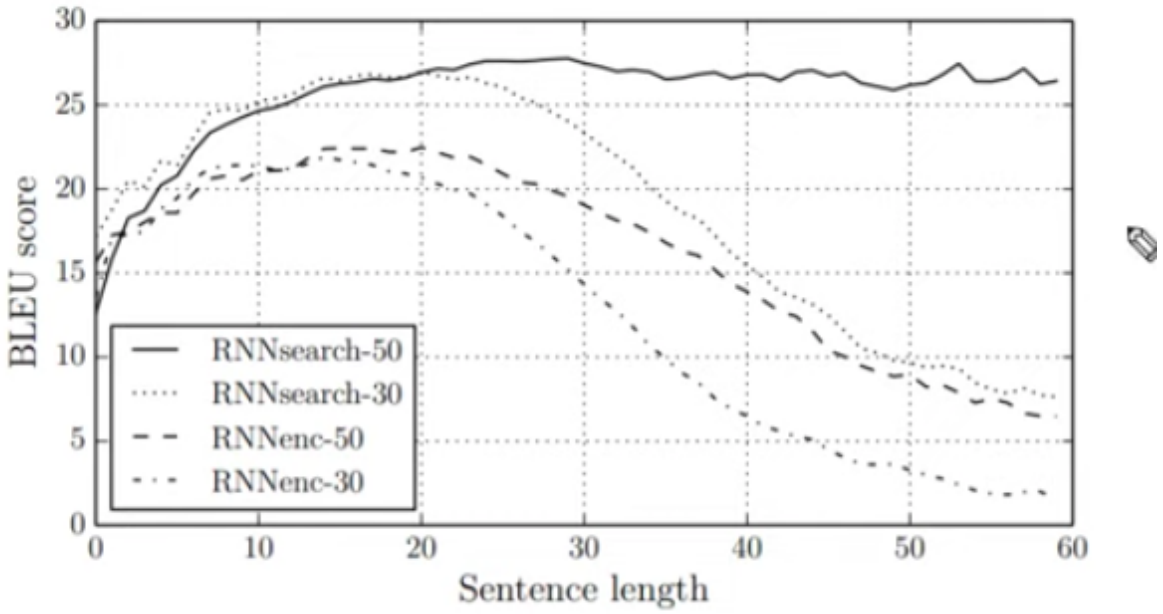
The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

where

$$e_{ij} = \boldsymbol{a}(s_{i-1}, h_j)$$

One point to note is that the entire LSTM RNN is based on probability.



After inserting the bidirectional RNN, the accuracy improved to a large extent as shown in the Bleu score above.

The window size here will be taken as a hyperparameter and it will be changed based on the translation we are doing.

Article for reference: Attention is all that you need!

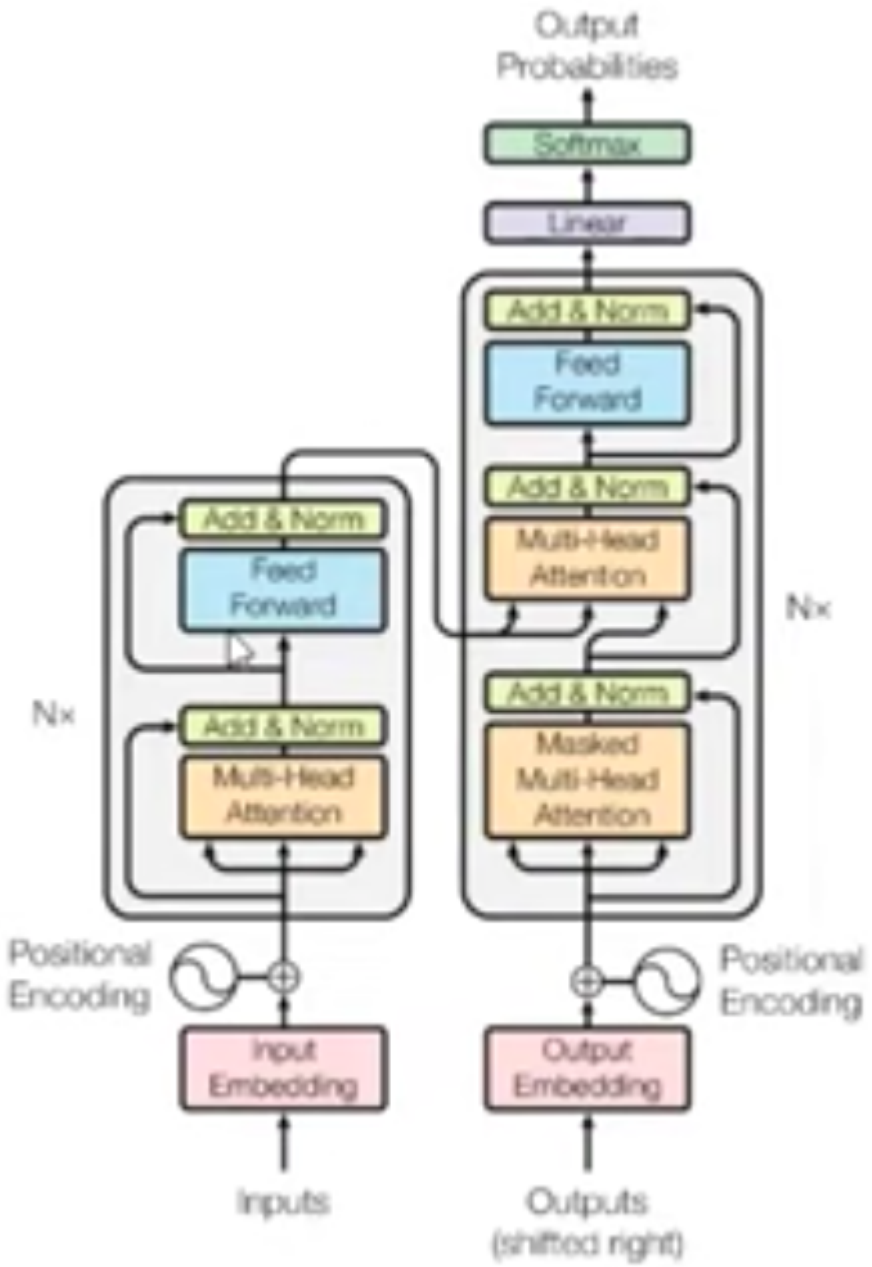


Figure 1: The Transformer - model architecture.

Unless we are stuck in some problem, we do not find a solution. So, at the time, people were facing issues in translating language (from one language to another). They were not able to translate their languages. This translation is not a one-to-one translation. One has to understand the sentences, the context and then generate a new word with respect to context to translate. Second problem, a question answering system - such as a chatbot. Meaning of cricket is different in English as per context - a game or an animal. Similarly, the market is inclined to be a bear. Here, bear does not refer to the animal “bear” but it is the contextual nature of the financial stock market. Suppose, one needs to find out the sentiment of a word → one cannot find the sentiment without understanding the entire context. For example, a video feedback - The video was good but the picturization was not that good. The system

used to get confused in understanding if it is a positive or a negative feedback. The earlier neural systems were unable to solve such problems.

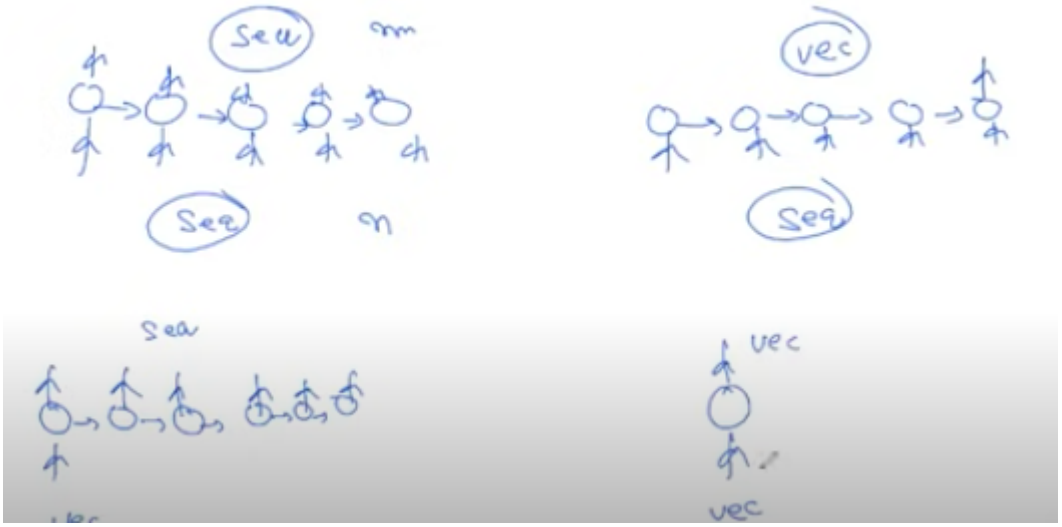
To solve such problems, people started finding new neural networks. At that time, CNN and transfer learning CNN based models had already developed. But these models were not able to solve the above problems. Because it was trying to train the weights based on the loss and optimizer used. So, we were basically trying to train the model by sending data from output to the front in backwards direction. We are trying to send the data from one end to another end.

But, in these problems we were not able to send the output data backwards using CNN as there was a lot of lag in which the information was not retained and it got lost. Thus, CNN was not just supposed to train the weight but also retain the context and other information to answer. It failed to do so. And , thus also failed to change the answer based on the contextual change. Hence, CNN and models based on it were not able to understand a context and thus were good for images but not for language or text-based problems as mentioned above.

The first model which was found to solve such problems was LSTM (Long short term memory). Popularly known LSTM networks are RNN and LSTM. People thought that we would give some input and we would be able to get some output. What if we try to send some of the output to the input. With that I will be able to understand what I will be able to do this time. Another kind of RNN was discovered as GRU. It was able to take feedback from the output to the input. The network was able to understand the context and have some memory. In the Gated Recurrent Unit, we were able to find a forget gate, a memory gate and an output gate. Forget gate was able to give which was already found, input gate was able to bring in something new and output gate was able to give the output. Now, people started facing some issues. If suppose we have to do a language translation, google translator understands a language and gives its equivalent translation in another language. These networks were not good enough to standalone take multiple inputs, understand them and generate output.

To find better solutions for such problems, researchers researched and a paper was published - Sequence to Sequence Learning with Neural Networks. This could solve such problems to a certain extent.

Whenever we try to work with text, we face different situations. Somewhere we have multiple inputs and we are looking for varied outputs as a sequence. This is a seq to seq model. Sometimes you are giving a sequence of inputs and expecting just one output, then that is a seq to vec model. Another case could be of a vec to seq model. Another kind could be vec to vec model. Here we are not talking about neurons but cells. Each cell would contain a forget gate, a memory gate and an output gate.



We can try to combine each and every cell and get the seq-to-seq output. Here, we are using RNN / GRU as a cell. But it could not solve our problems. Then researchers thought that if we could separate the entire network of inputs and the entire network of outputs separately, would we be able to perform better. This led to Sequence to sequence neural networks.

The concept of encoder and decoders came into picture with the idea of separating the input and output networks completely. Firstly the model trained by back propagating the loss to reduce it and finally predicted after training. But here also people faced problems. The data was being propagated in just one direction. The output of encoder-decoder model is computed by undergoing the following equations:

$$h_t = \text{sigm}\left(W^{\text{hx}} x_t + W^{\text{hh}} h_{t-1}\right)$$
$$y_t = W^{\text{yh}} h_t$$

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

Whenever we talk about a sentence, some kind of word before as well as a word matters. For example: Market is bearish and is going to be slow. Now, just bearish does not make sense but with context it does. So, in case of a sentence even a future word matters for defining the context and thus translation. Thus people started thinking about a newer, better approach.

They proposed using a bidirectional LSTM or RNN. This led to the attention models.

Another reference paper: Effective modeling of Encoder-Decoder Architecture for Joint Entity and Relation Extraction

In this model also, the base is the same, but we try to use a unidirectional LSTM, or a bidirectional LSTM, or a stacked LSTM.

Now, why are we learning these models? Instead we can just learn BERT and transformer models. To appreciate the model really well, we need to understand what led to that particular model and what was its need.

Earlier, we saw a basic encoder-decoder where our data was being passed from the previous network to the next network and the output of the last layer of encoder was a context vector. It was trying to transfer that context vector to the decoder model which would be responsible to provide some kind of output based on the input it gets. Then, the loss is computed and back propagated. Hare, the problem was that the data is propagated in one direction only. Another issue is that we are trying to get the context of all the inputs at the very end. It would have been possible that not all the inputs were contributing towards the outcome. We were just trying to get one context vector that was a summarization of all the inputs. Sometimes it matters and sometimes it does not. We need a network that understands what I need as an input to get a particular output. Here is when the attention model comes into picture. We will be able to try and understand, and pick those words which are needed for a particular prediction model.

Reference paper: Neural Machine Translation By Jointly Learning to Align and Translate

Our story for the attention model starts from the above paper. People said that they were going to use the Encoder Decoder model but they would use a bidirectional RNN / LSTM model as the base rather than a unidirectional RNN model.

The neural network in between the encoder and decoder side conveys which parts will get activated. It will focus and train certain weights.

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \tag{5}$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

where

$$e_{ij} = a(s_{i-1}, h_j)$$

Tx is the number of attention inputs which we are going to take inside the neural network. It is a hyperparameter.

**can check out examples of use cases on Stanford university SQuAD dataset (Stanford Question Answering Dataset) - best model performance by Albert model

Reference articles: Attention-based Recurrent Neural Networks for Question Answering;

Attention-based RNN Model for Speech Recognition;

Video-based attention model;

Image Capturing by Attention model

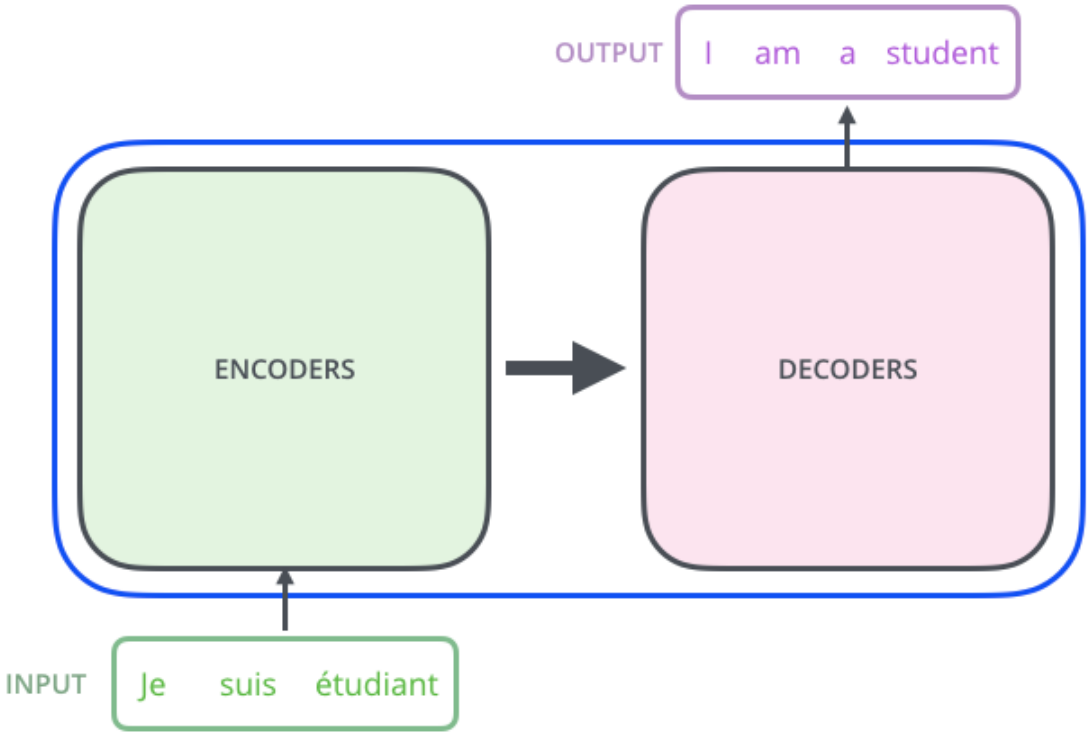
Transformers:

Article / Paper referred - Attention is All You need (Link - https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)

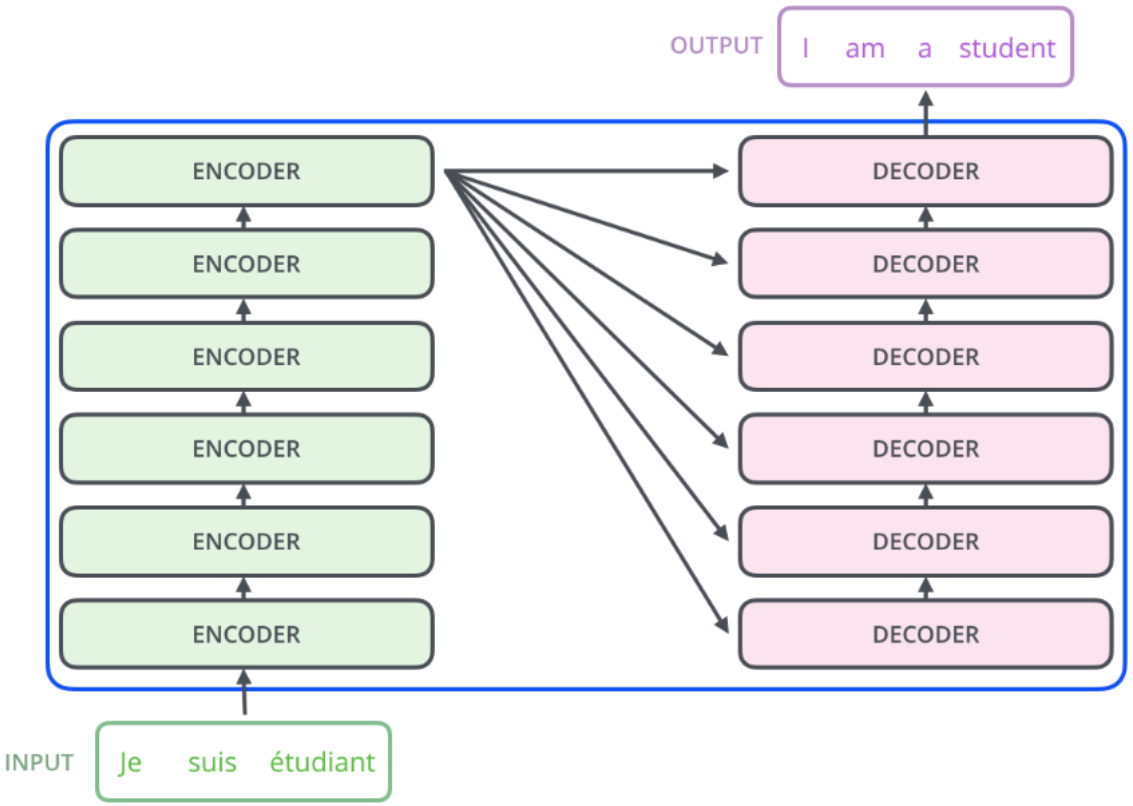
Another reference: http://jalammar.github.io/illustrated...

Black-box model (High-level look):

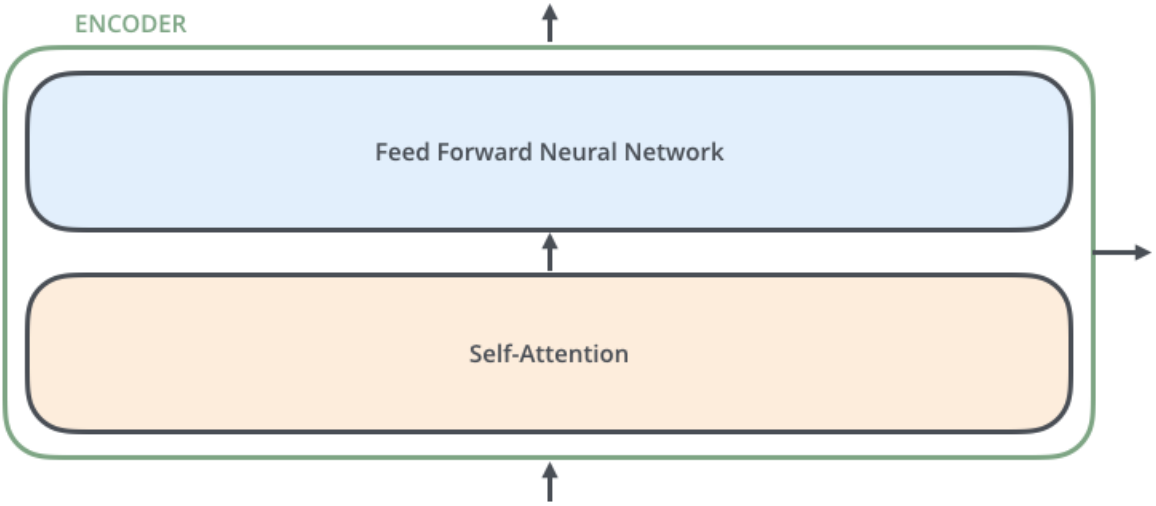
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.



The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements - 6 gave good results for the researchers; it is a kind of hyperparameter). The decoding component is a stack of decoders of the same number.



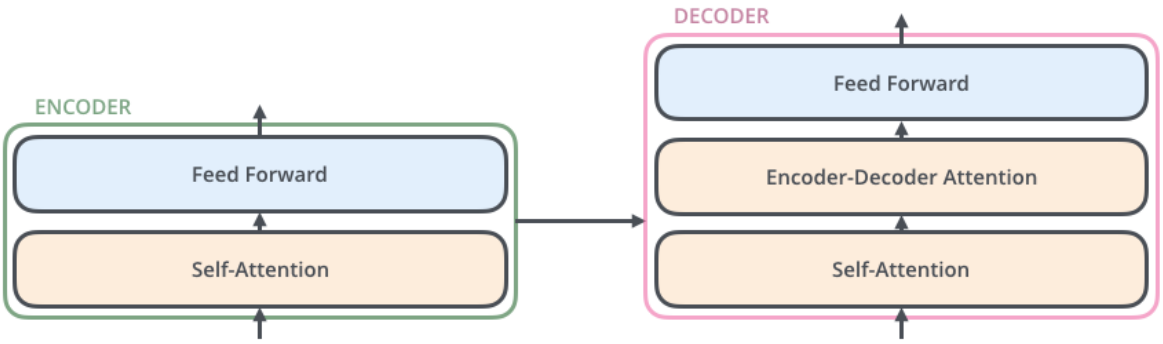
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sublayers:



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar to what attention does in seq2seq models).

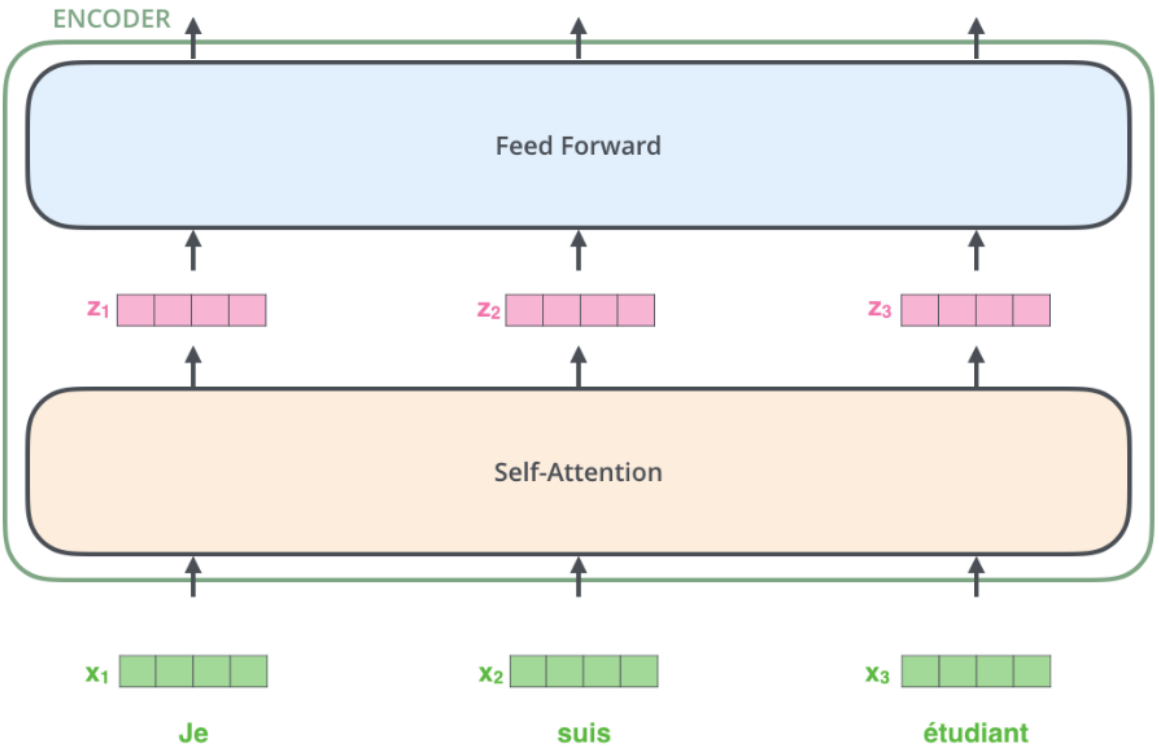


As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is a hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

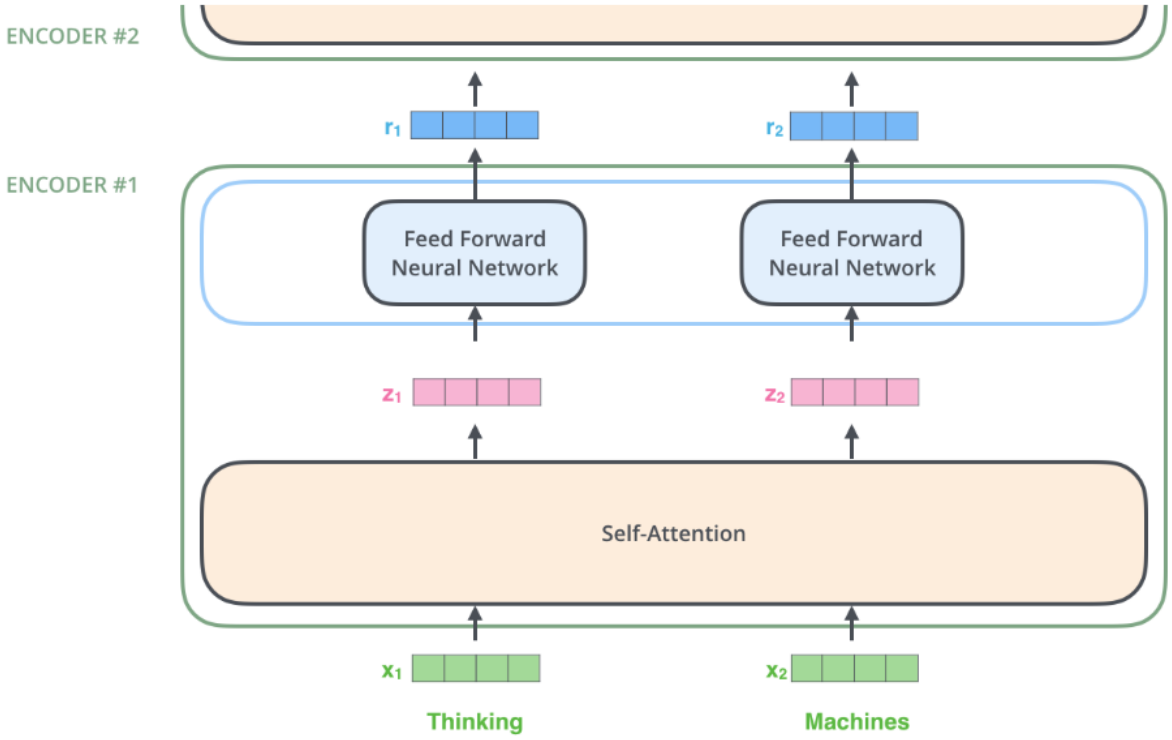


Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

Now We're Encoding!

As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



Self-attention - What is it?

Say the following sentence is an input sentence we want to translate:

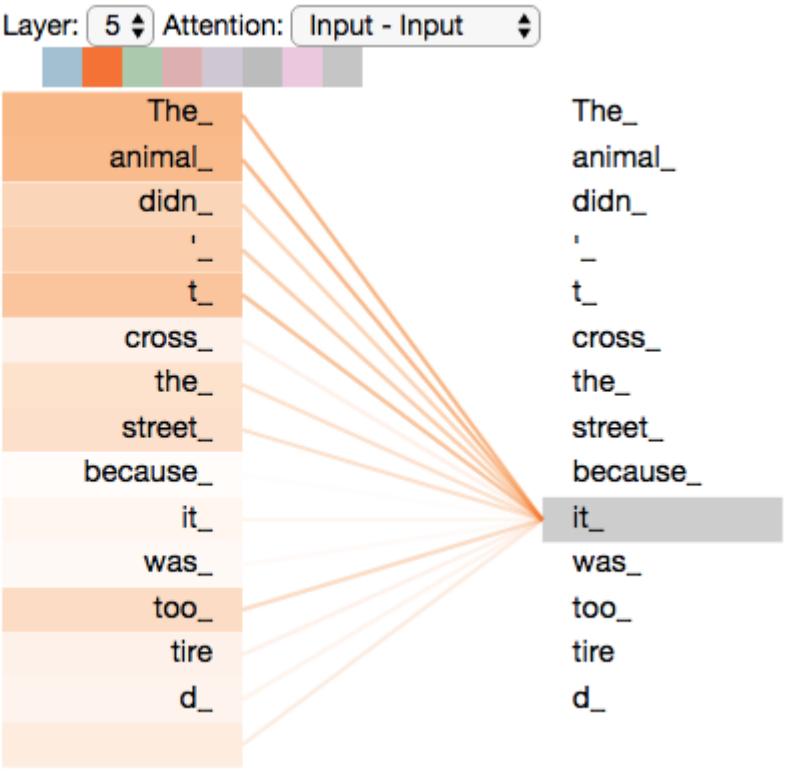
”The animal didn't cross the street because it was too tired”

What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

If you’re familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it’s processing. Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.

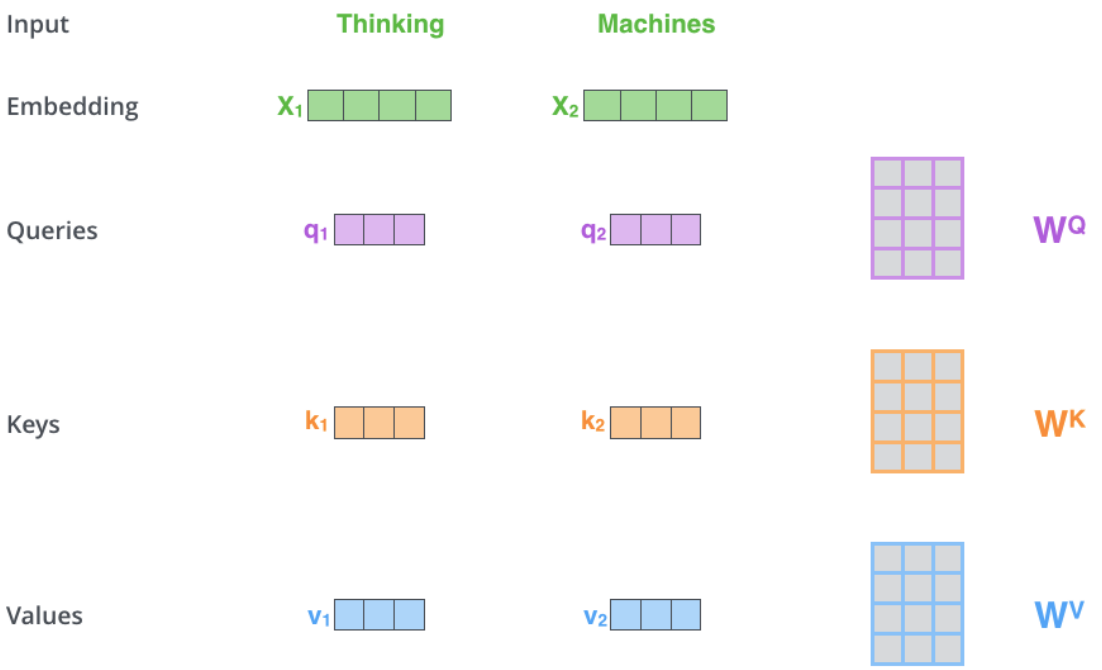


Self-attention in Detail:

Let's first look at how to calculate self-attention using vectors, then proceed to look at how it's actually implemented – using matrices.

The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't HAVE to be smaller, this is an architecture choice to make the computation of multi headed attention (mostly) constant.



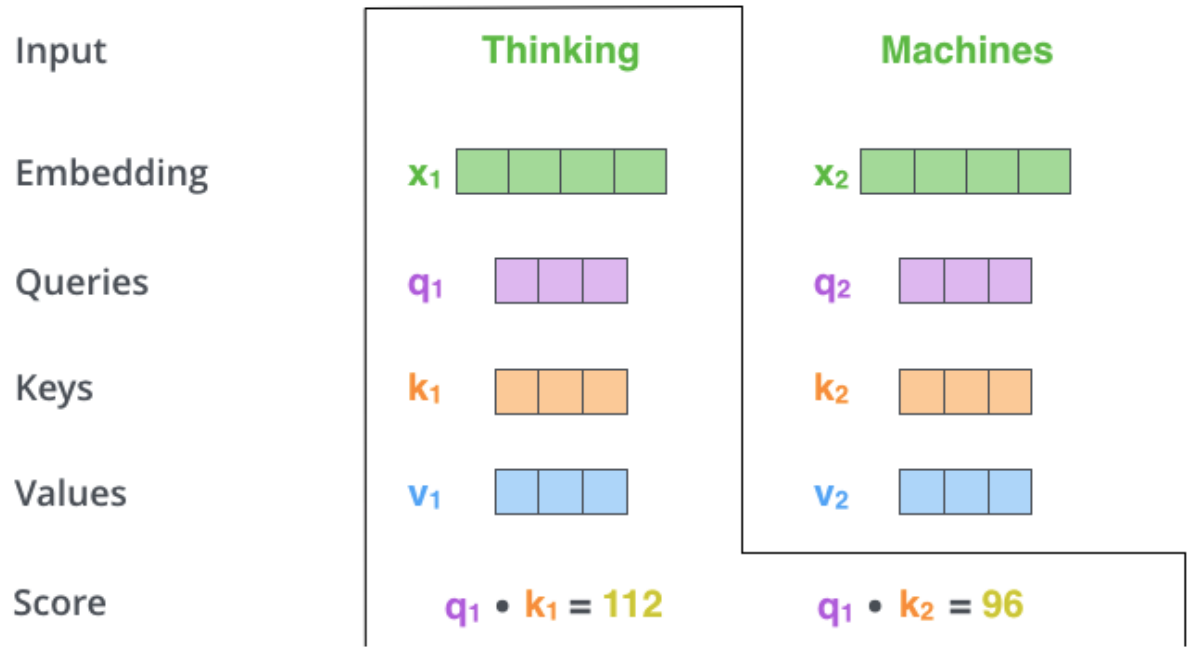
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

What are the "query", "key", and "value" vectors?

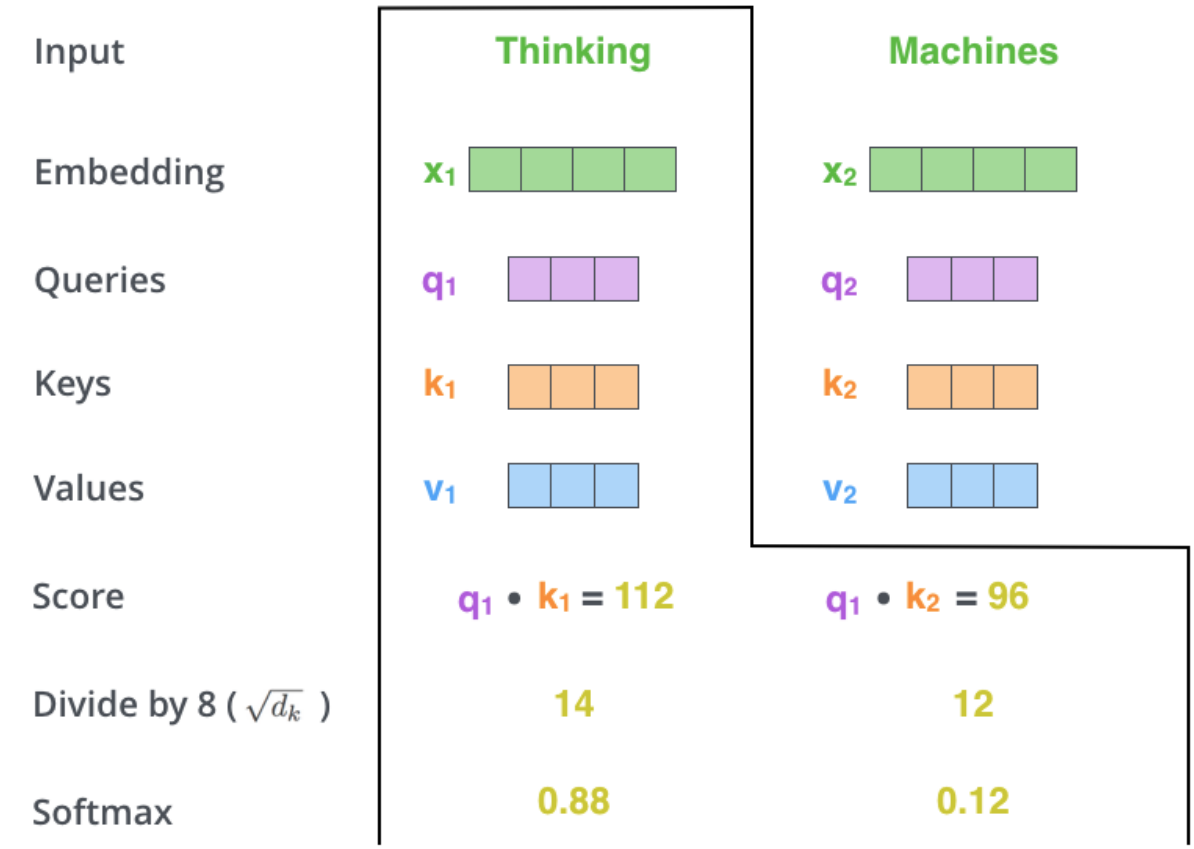
They're abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you'll know pretty much all you need to know about the role each of these vectors plays.

The **second step** in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.



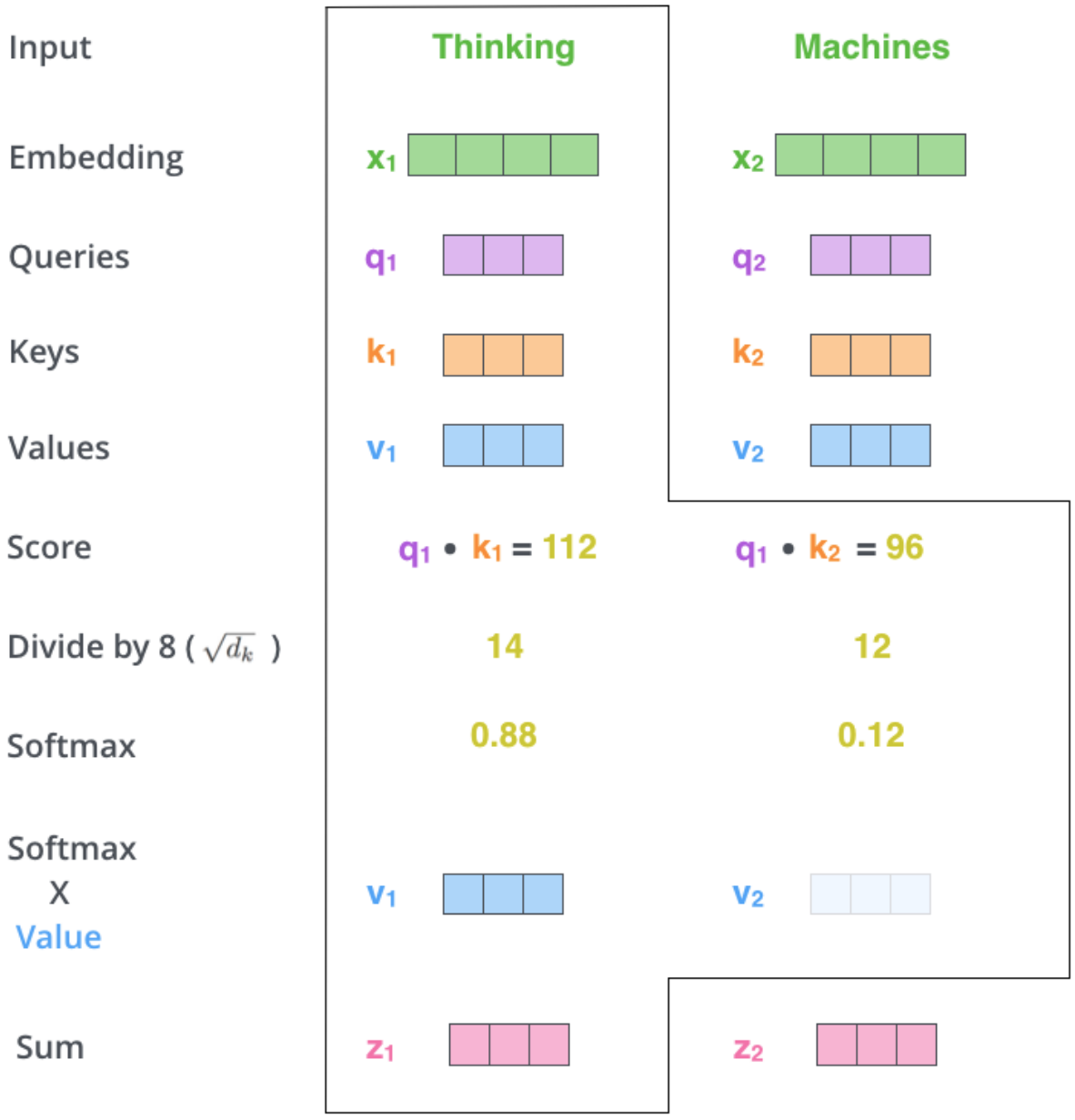
The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.



This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

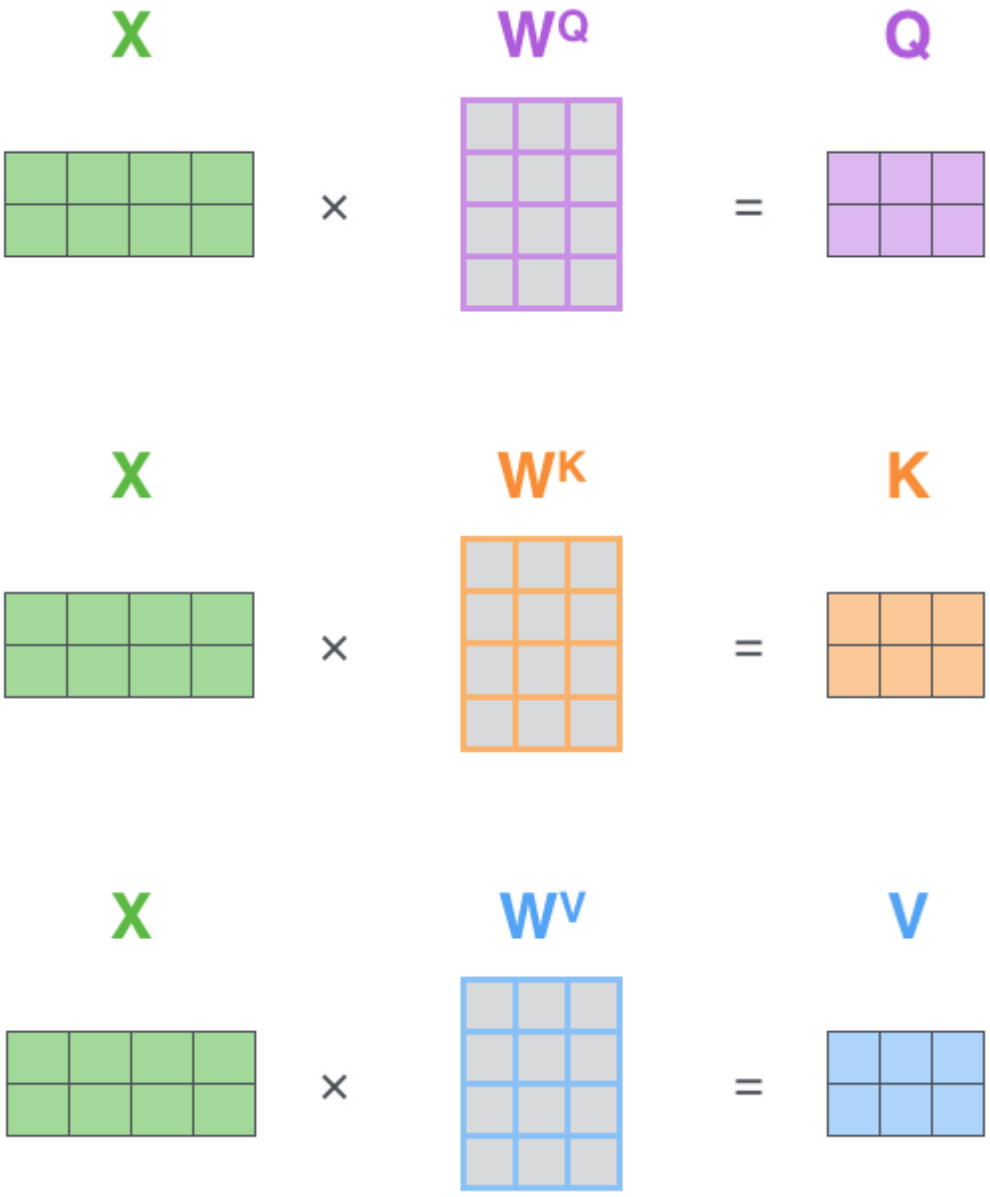


That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is

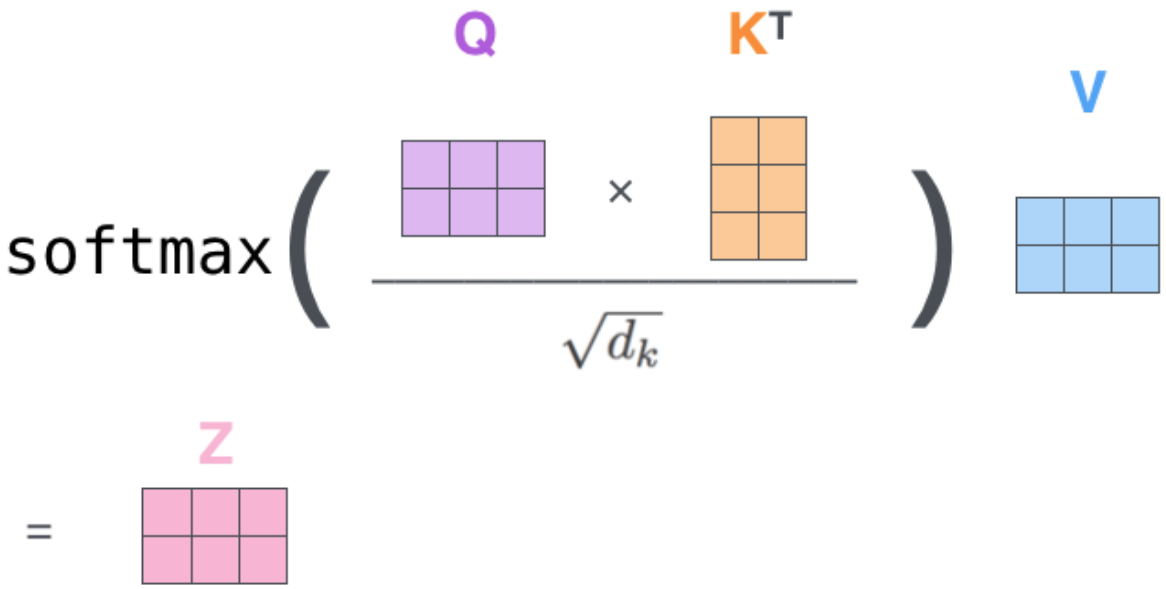
done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

Matrix Calculation for self-attention:

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).



Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



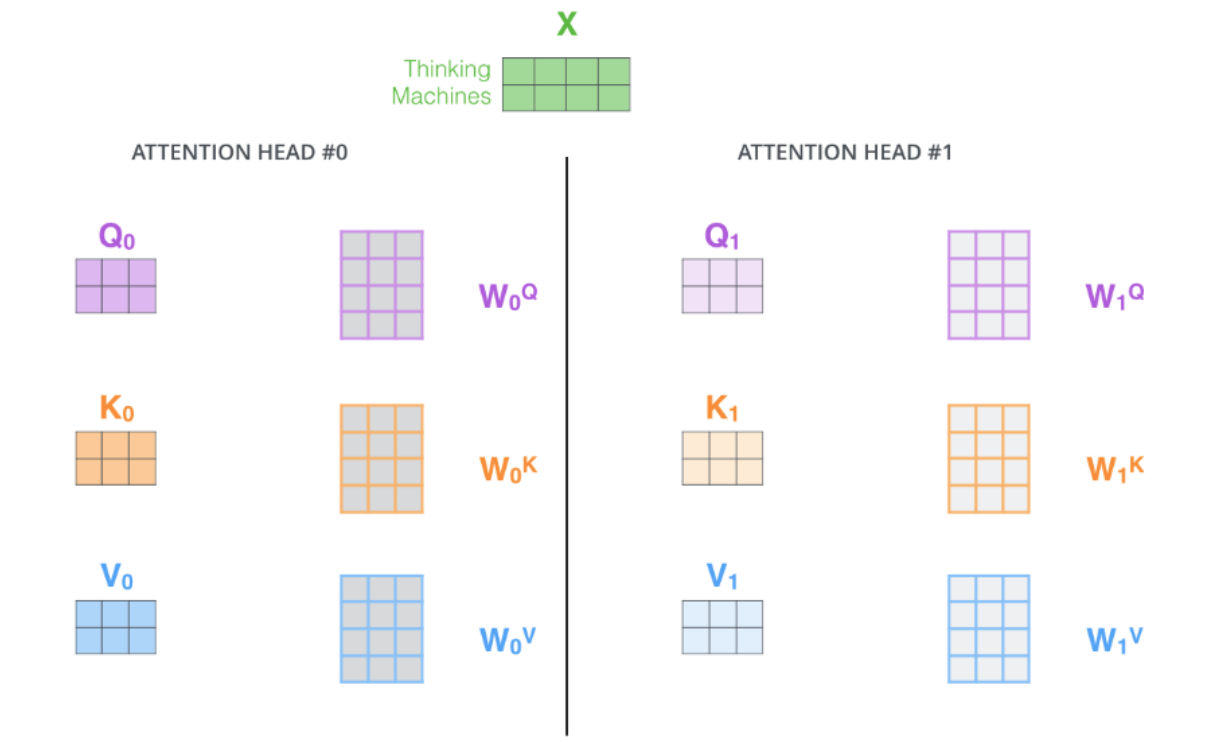
The self-attention calculation in matrix form

We have used single Wq, Wk and Wv for all the words being sent in the input. But, it poses some problems with it as the word dependencies are not correctly identified. That is why we go ahead with multi-headed attention.

Going in-depth: The beast with many heads

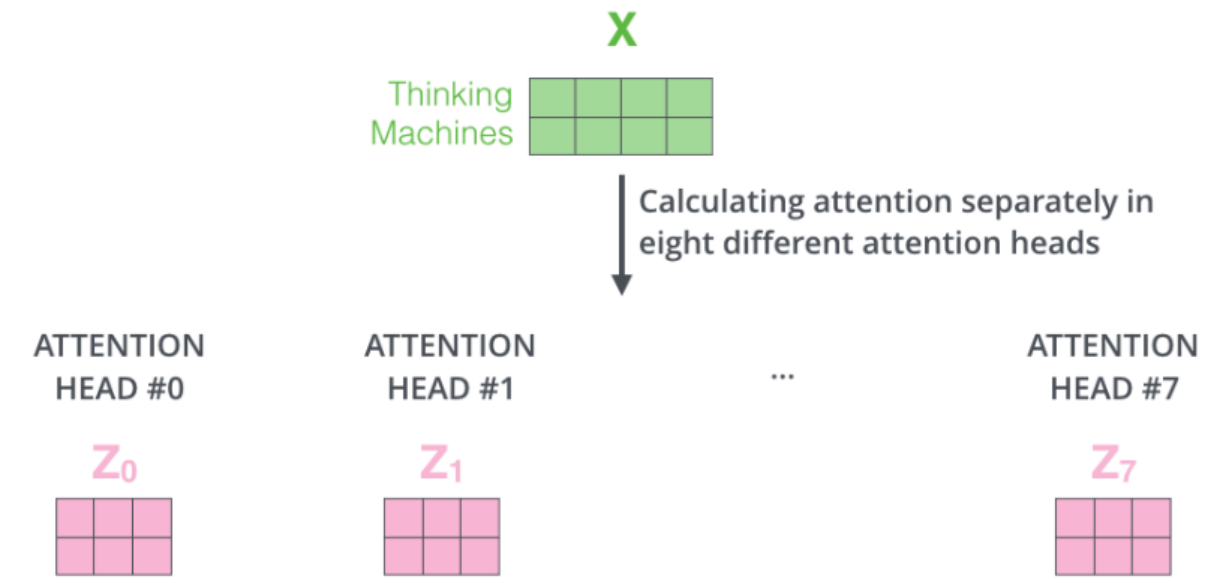
The paper further refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. It would be useful if we're translating a sentence like "The animal didn't cross the street because it was too tired", we would want to know which word "it" refers to.
2. It gives the attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.



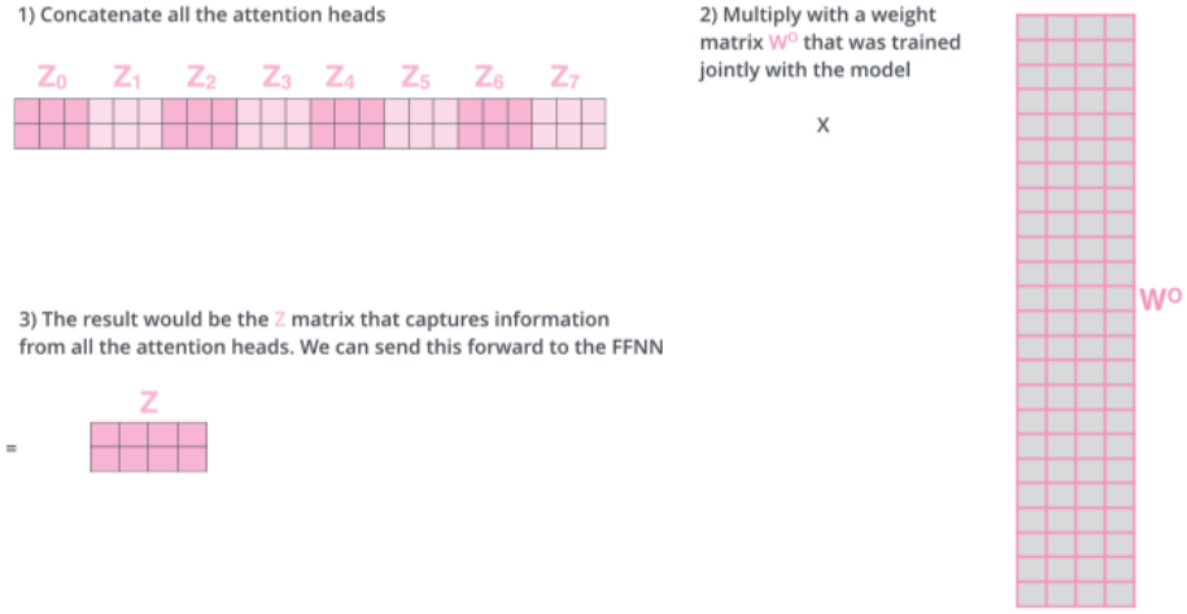
ith multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices

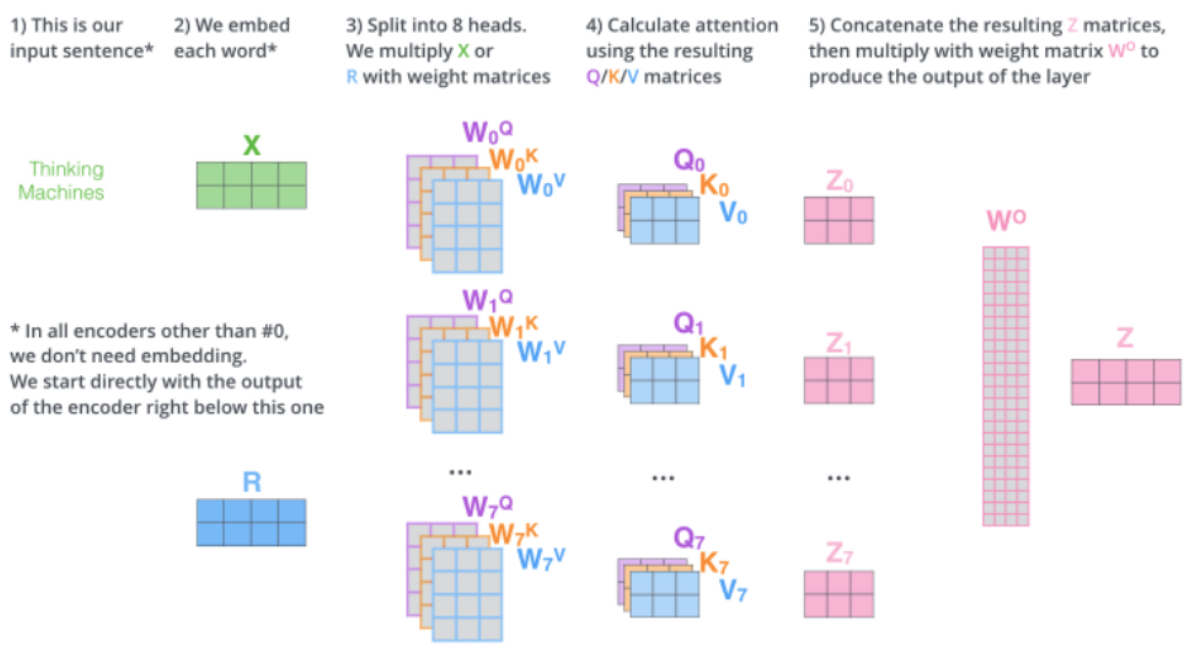


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

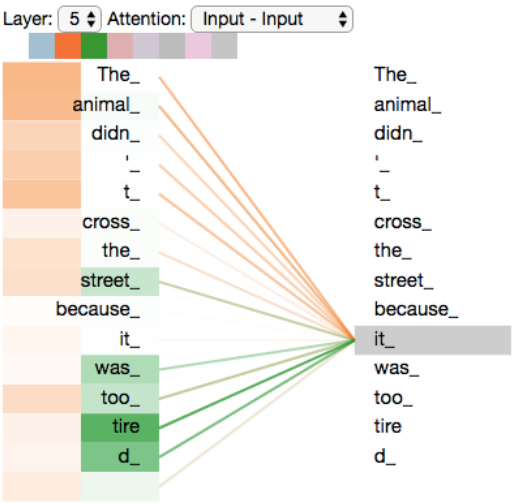
How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.



That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place

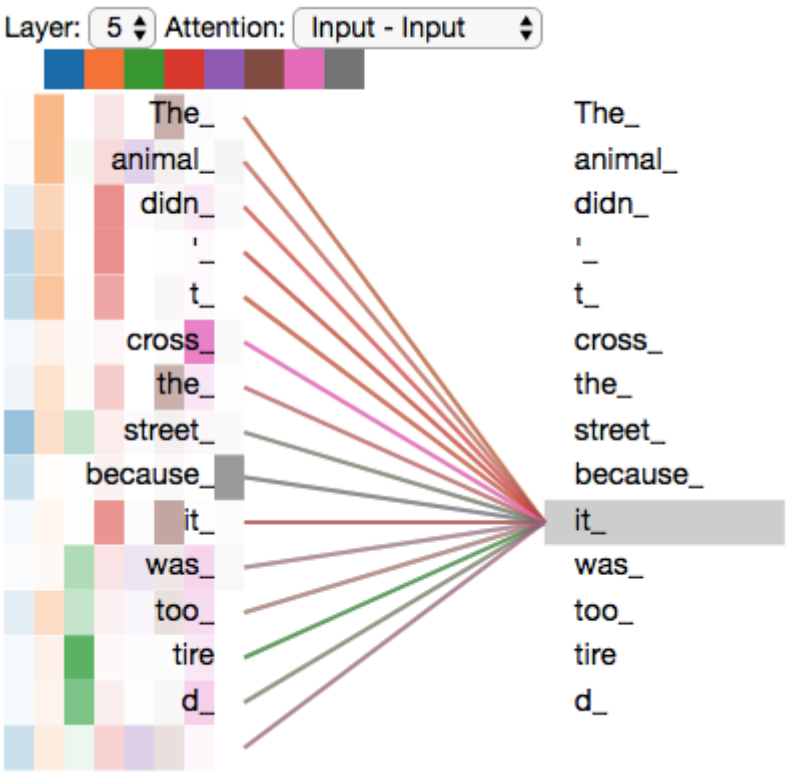


Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

If we add all the attention heads to the picture, however, things can be harder to interpret:

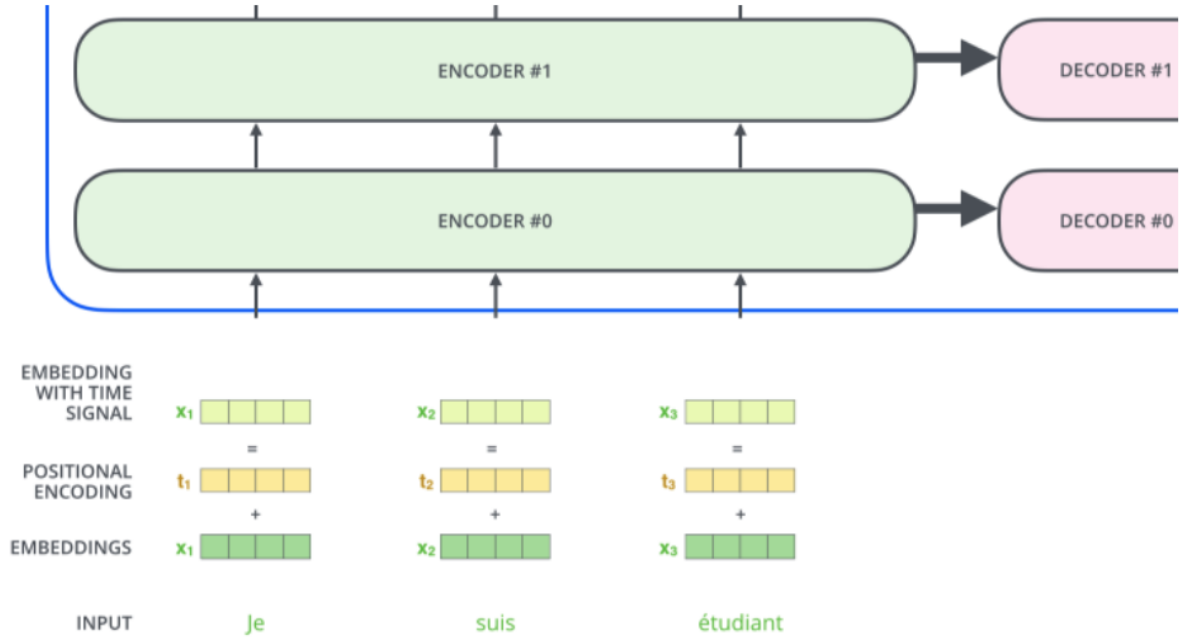


Representing The Order of The Sequence Using Positional Encoding

One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

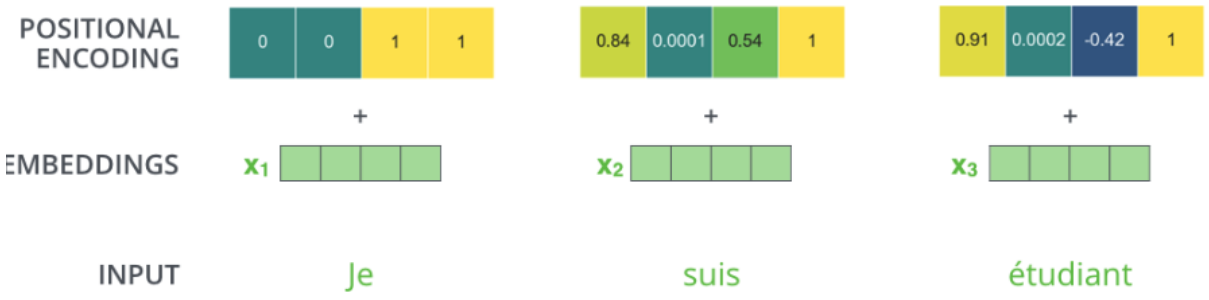
To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that

adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

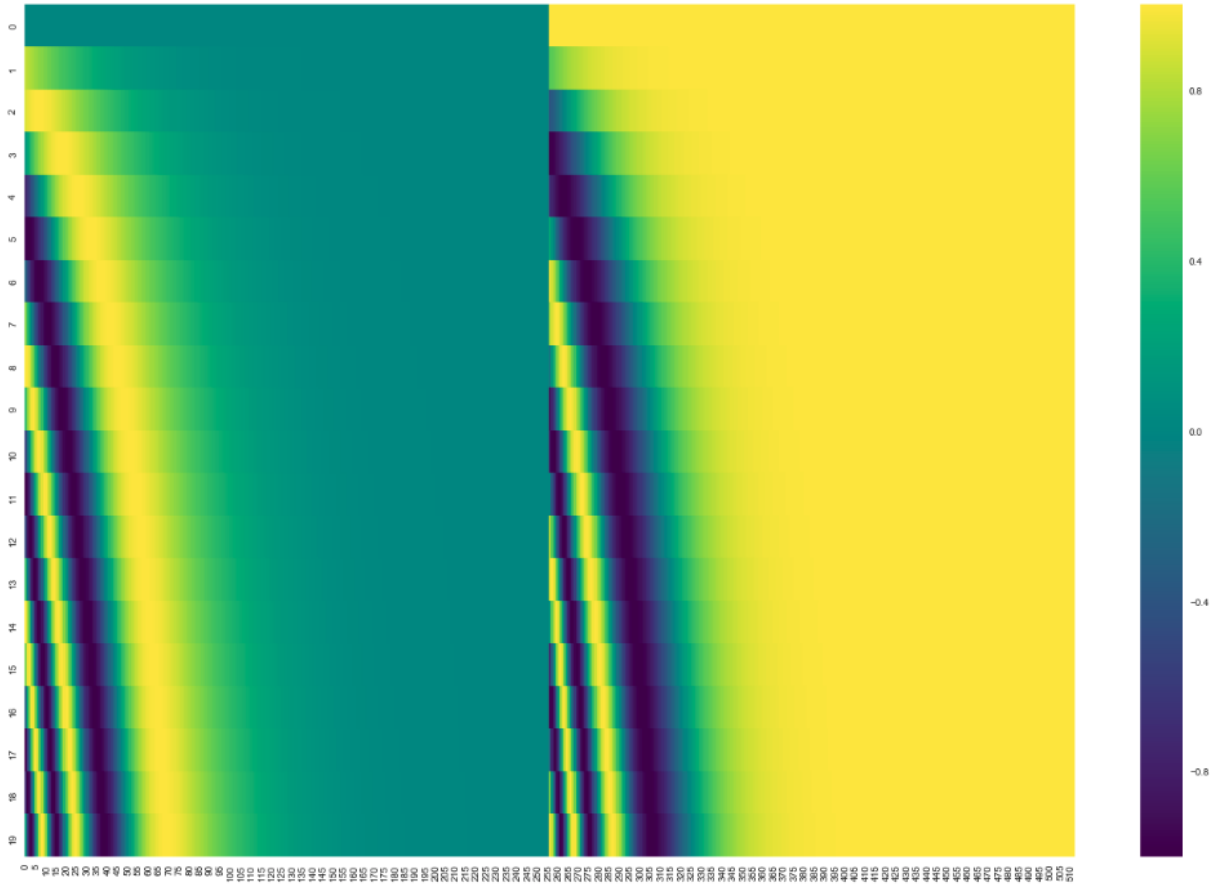
If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



A real example of positional encoding with a toy embedding size of 4

What might this pattern look like?

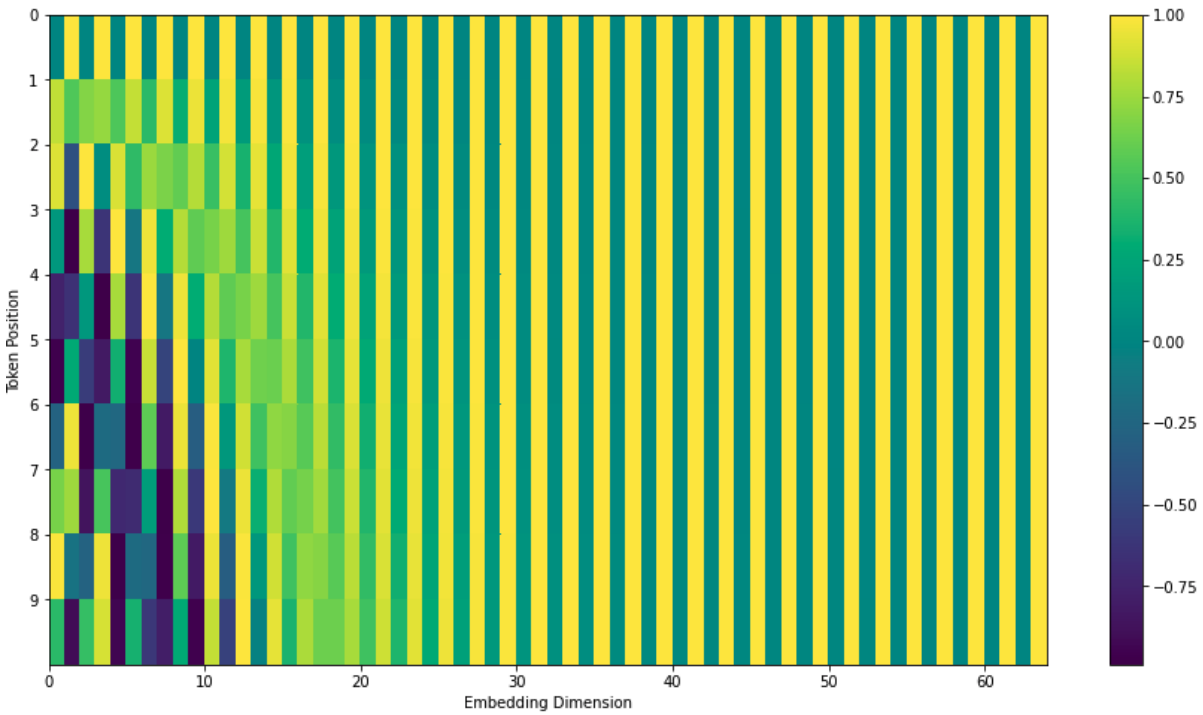
In the following figure, each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

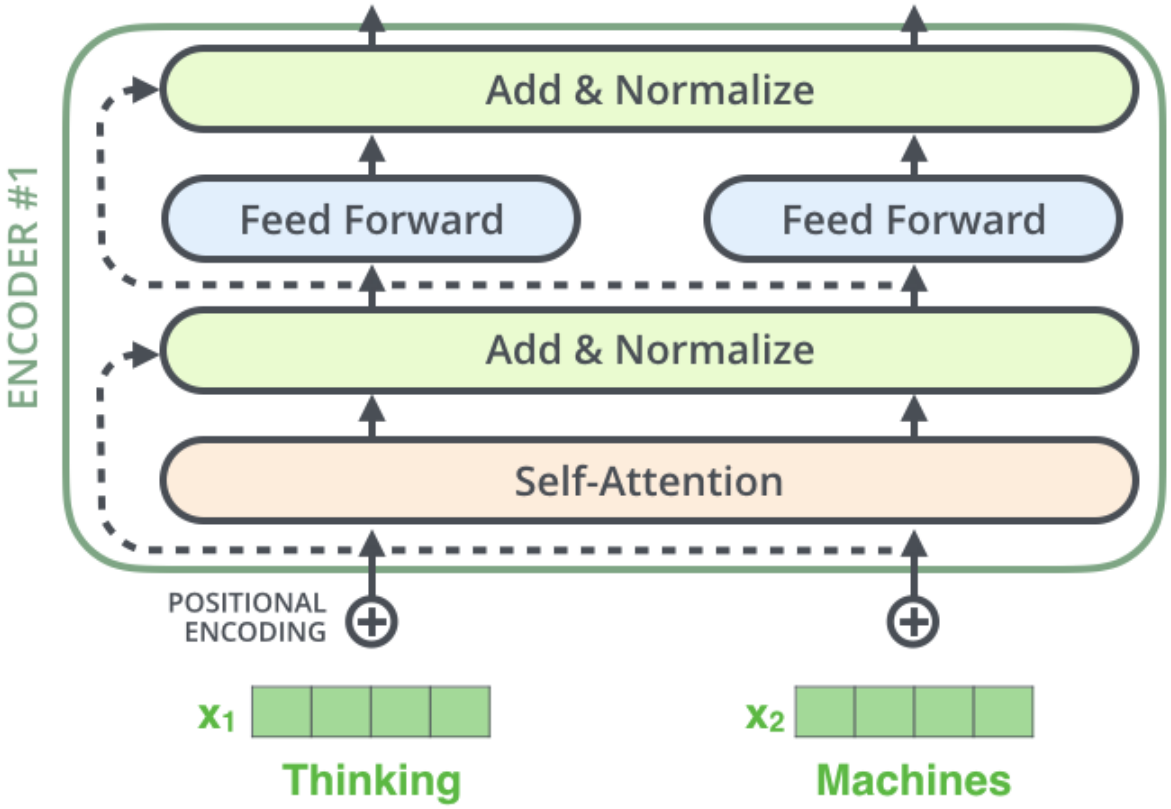
The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in `get_timing_signal_1d()`. This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of sequences (e.g. if our trained model is asked to translate a sentence longer than any of those in our training set).

July 2020 Update: The positional encoding shown above is from the Tranformer2Transformer implementation of the Transformer. The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. Here's the code to generate it:

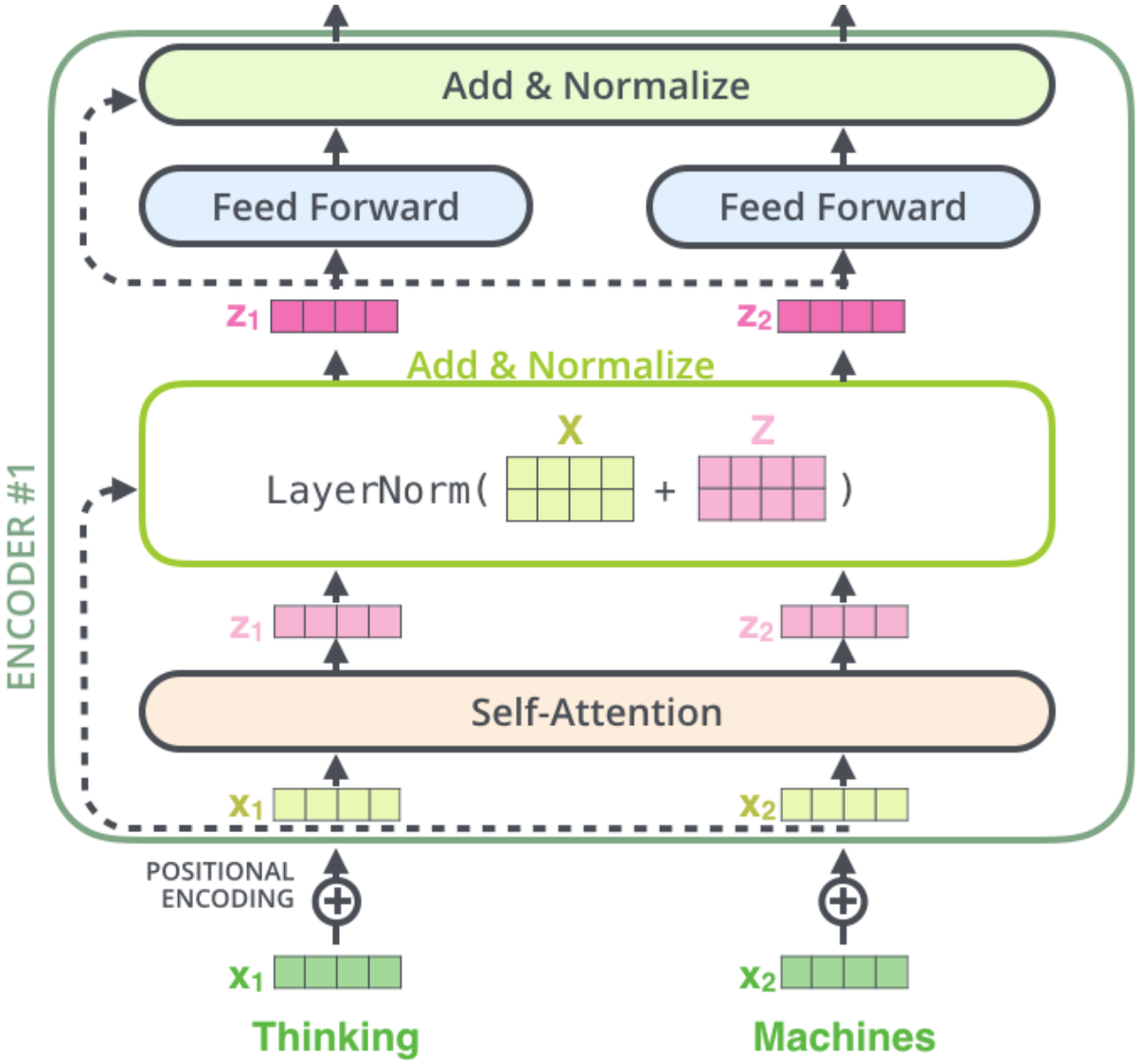


The Residuals

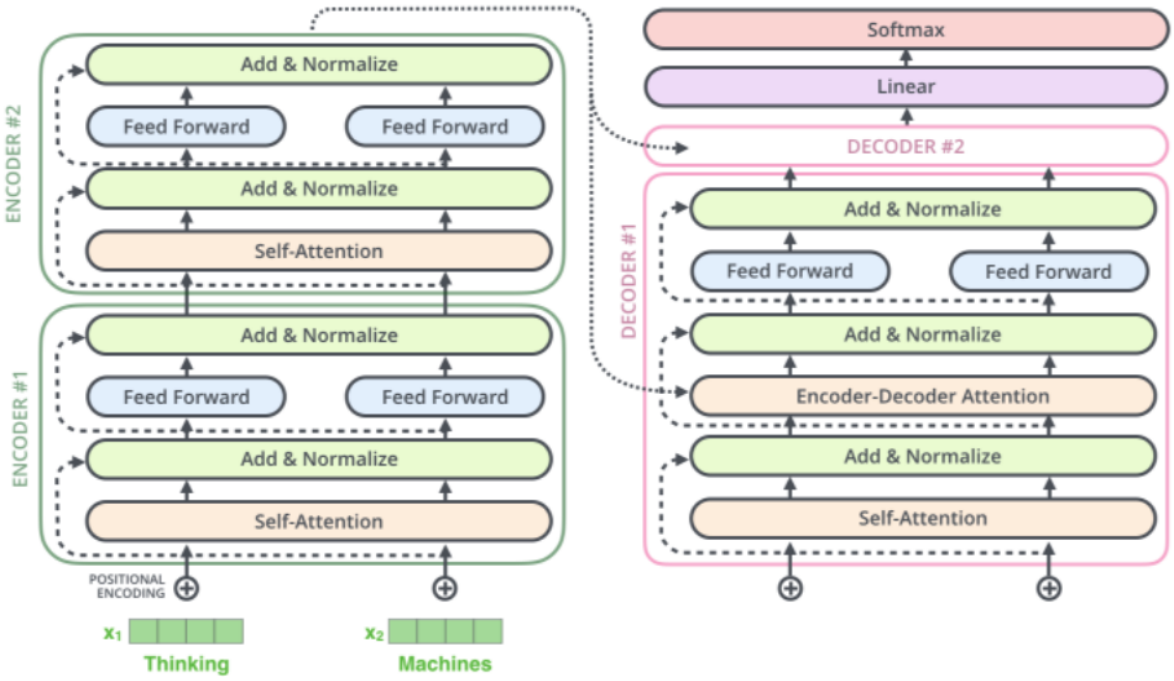
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:



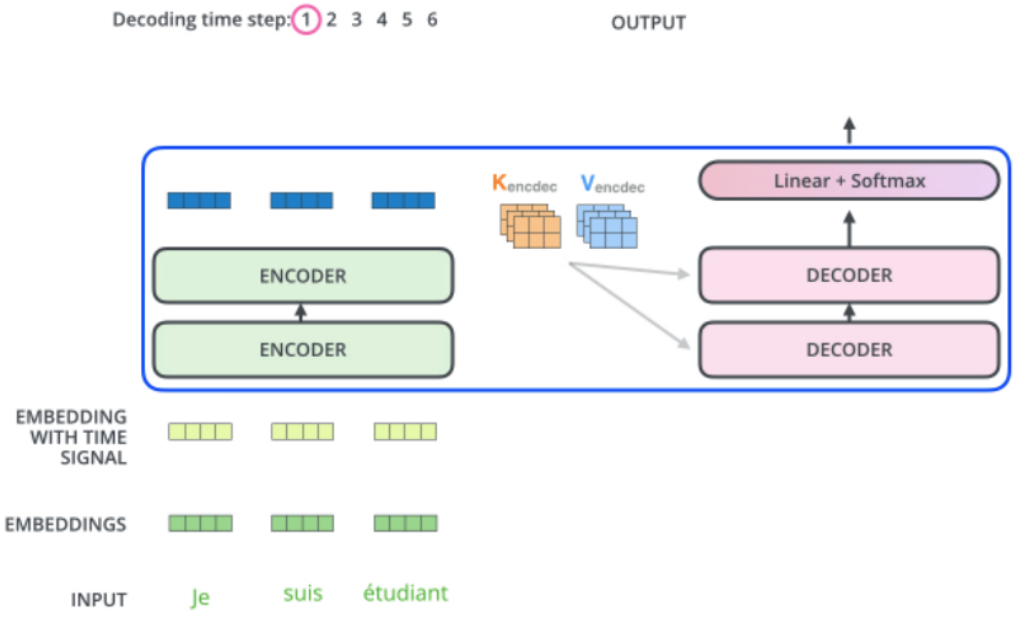
This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



The Decoder Side

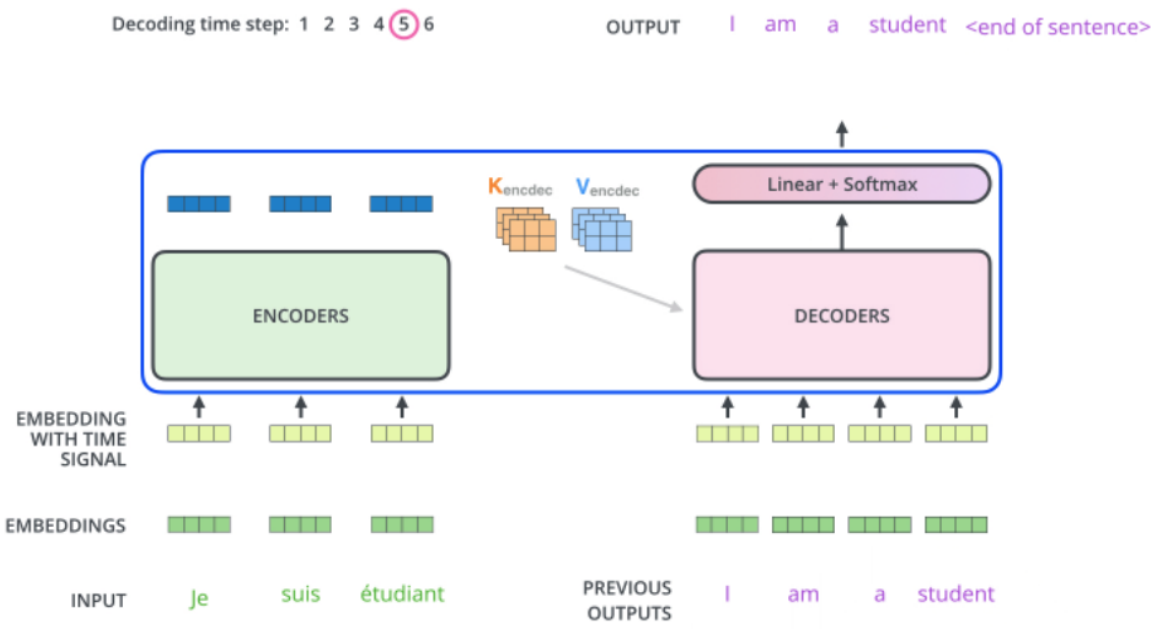
Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

The encoder starts by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.



The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to `-inf`) before the softmax step in the self-attention calculation.

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

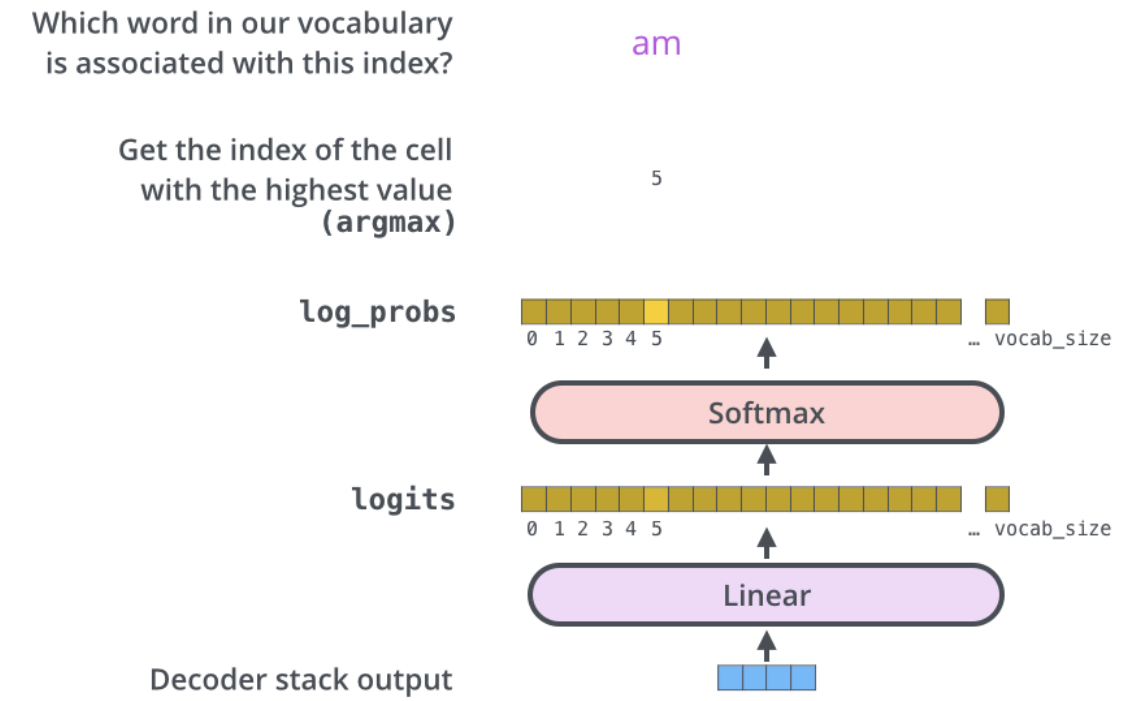
The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That’s the job of the final Linear layer which is followed by a Softmax Layer.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

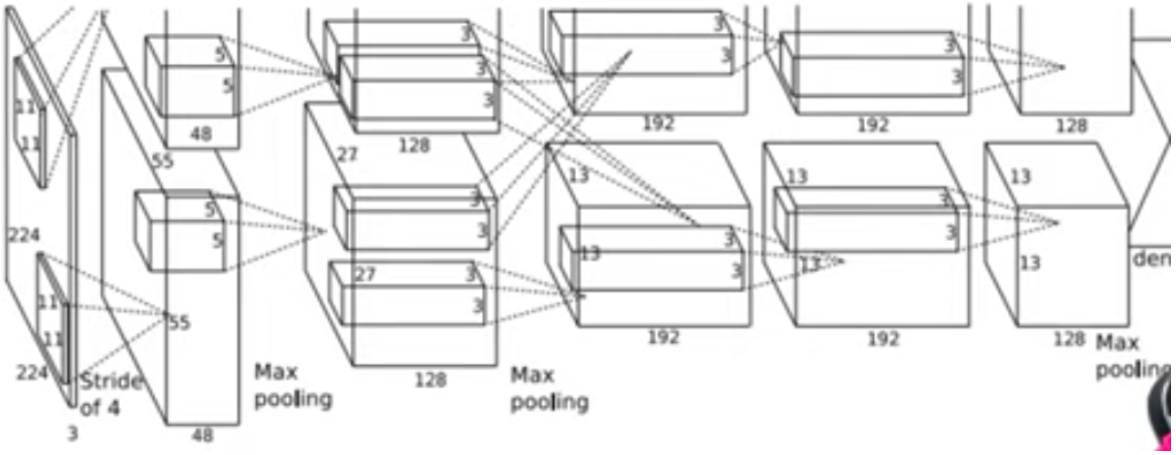
Let’s assume that our model knows 10,000 unique English words (our model’s “output vocabulary”) that it’s learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

Advanced CNN: Alexnet Architecture In-depth (TRANSFER LEARNING)



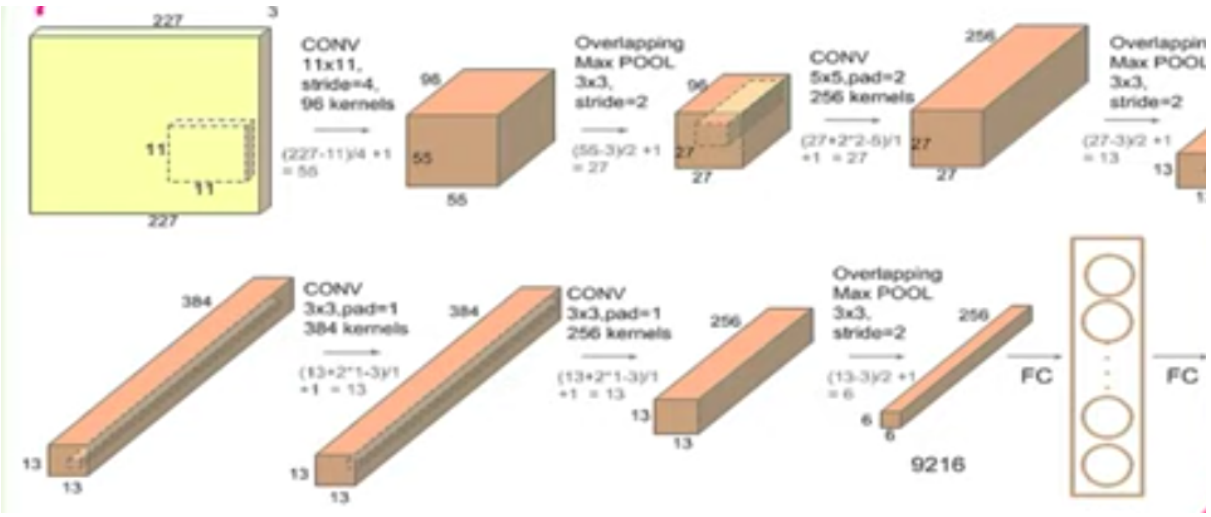


Image: 227X227X3 (RGB Channel)

First Conv layer: 96 filters (filter size, f: 11X11 and stride: 4) - 55X55X96 (Calculation -> (227 - 11) / 4 + 1 = 55 i.e (n + 2p - f)/ s + 1

Max Pooling layer: Pool (3X3 and stride = 2) (55 + 0 - 3 / 2) + 1 = 27; 27X27X96

Second Conv layer: 5X5 filter size; pad = 2; 256 kernels -> (27 + 4 - 5)/1 +1 = 27 → 27X27X256

Max Pooling layer: Pool (3X3 and stride = 2) (27 + 0 - 3 / 2) + 1 = 13; 13X13X256

Third Conv layer: 3X3 filter size; 384 kernels; pad = 1 → 13+ 2 - 3/1+1 = 13; 13X13X384

Fourth Conv layer: 3X3 filter size; 384 kernels; pad = 1 → 13X13X384

Fifth Conv layer: 3X3 filter size; 256 kernels; pad = 1 → 13X13X256

Max Pooling layer: Pool (3X3 and stride = 2) 13 - 3/2 + 1 = 6; 6X6X256 {i.e. 9216}

Fully connected layer (FC1): 4096 nodes {dense nodes}

Fully connected layer (FC2): 4096 nodes {dense nodes}

Output layer (1000 categories) - softmax function

XXVI. Natural Language Processing(NLP)

Machine Learning Libraries: SpaCy, NLTK
Deep learning Libraries: PyTorch, Keras, TensorFlow
BERT, Transformers: Hugging Face Libraries