**CHAPTER 5: Neural Networks**

**Theory**

In the previous chapter, we studied what is decision tree, what are ensemble learning techniques, and gradient boosting. In this chapter, we will learn Feature Selection methods in machine learning, Loss functions and Gradient Descent in Machine learning, and Neural Networks.

Let’s start our chapter with Feature Selection Techniques.

**Feature Selection Techniques in Machine Learning**

**Introduction**

When building a machine learning model in real-life, it’s almost rare that all the variables in the dataset are useful to build a model. Adding redundant variables reduces the generalization capability of the model and may also reduce the overall accuracy of a classifier. Furthermore, adding more and more variables to a model increases the overall complexity of the model.

The goal of feature selection in machine learning is to find the best set of features that allows one to build useful models of studied phenomena.

The techniques for feature selection in machine learning can be broadly classified into the following categories:

**Supervised Techniques:** These techniques can be used for unlabelled, and are used to identify the relevant features for increasing the efficiency of supervised models like classification and regression.

**Unsupervised Techniques:** These techniques can be used for unlabelled data. From a taxonomic point of view, these techniques are classified as under.

* Filter methods
* Wrapper methods
* Embedded methods
* Hybrid methods

**Filter methods**

Filter methods pick up the intrinsic properties of the features measured via univariate statistics instead of cross-validation performance. These methods are faster and less computationally expensive than wrapper methods. When dealing with high-dimensional data, it is computationally cheaper to use filter methods.

Let’s discuss some of these techniques:

1. **Chi-square Test:**

The Chi-square test is used for categorical features in a dataset. We calculate Chi-square between each feature and the target and select the desired number of features with the best chi-square scores. In order to correctly apply the chi-squared in order to test the relation between various features in the dataset and the target variable, the following conditions have to meet: the variables have to be categorical, sampled independently and values should have an expected frequency greater than 5.

**Importing the dataset and Necessary libraries**

Graphical user interface, text, table

Description automatically generated

Graphical user interface, text, application

Description automatically generated

We will be using the ‘titanic” dataset for the illustration of the chi-square feature selection method.

Let’s filter out all the categorical variables to perform the chi-square test

Graphical user interface, text, application

Description automatically generated

We can see “sex” & “embarked” variables have category data. Let’s perform label encoding on sex and embarked columns.

A picture containing chart

Description automatically generated



Table

Description automatically generated

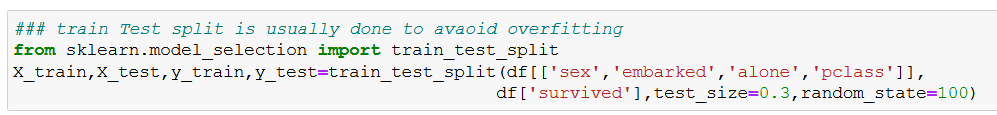
Let’s also perform label encoding on the ‘alone’ variable too.



A picture containing text

Description automatically generated

Now, lets split data into train and test data.



Now, we can perform feature selection using chi-square test

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

**Observation**

‘Sex’ columns are the most important column when compared to the output feature ‘Survived’

1. **Correlation Coefficient**

Correlation is a measure of the linear relationship of 2 or more variables. Through correlation, we can predict one variable from the other. The logic behind using correlation for feature selection is that the good variables are highly correlated with the target. Furthermore, variables should be correlated with the target but should be uncorrelated among themselves.

If two variables are correlated. We can predict one from the other. Therefore, if two features are correlated, the model only really needs one of them, as the second one does not add additional information. We will use the Person Correlation here.

**Importing necessary libraries and data**

A picture containing text

Description automatically generated

We are using a mobile dataset to demonstrate the correlation coefficient-based feature selection.

A picture containing text, indoor, screenshot

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Application, table, Excel

Description automatically generated

Text

Description automatically generated

Chart

Description automatically generated with medium confidence

We need to set an absolute value, say 0.5 as the threshold for selecting the variables. If we find that the predictor variables are correlated among themselves, we can drop the variables which have a lower correlation coefficient value with the target variable. We can compute multiple correlation coefficients to check whether more than two variables are correlated to each other. This phenomenon is known as multicollinearity.

1. **Information gain**

Information gain calculates the reduction in entropy from the transformation of a dataset. It can be used for feature selection by evaluating the information gain of each variable in the context of the target variable.

Graphical user interface, text, application

Description automatically generated

1. **Information gain**

Information gain calculates the reduction in entropy from the transformation of a dataset. It can be used for feature selection by evaluating the information gain of each variable in the context of the target variable.

Graphical user interface, text, application

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Text

Description automatically generated with medium confidence

Chart, bar chart

Description automatically generated

1. **Variance Threshold**

The variance threshold is a simple baseline approach to feature selection. It removes all features which variance doesn’t meet some threshold. By default, it removes all zero-variance features, i.e., features that have the same value in all samples, we assume that features with a higher variance may contain more useful information, but note that we are not taking relationship between feature variables or feature and target variables into account, which is one of the drawbacks of filter methods.

Graphical user interface, text, application

Description automatically generated

**Wrapper Methods**

Wrapper requires some method to search the space of all possible subsets of features, assessing their quality by learning and evaluating a classifier with that feature subset. The feature selection process is based on a specific machine learning algorithm that we are trying to fit on a given dataset. It follows a greedy search approach by evaluating all the possible combinations of features against the evaluation criterion. The wrapper methods usually result in better predictive accuracy than filter methods.

Let’s discuss some of these techniques:

1. **Forward Feature Selection**

This is an iterative method wherein we start with the best performing variable against the target. Next, we select another variable that gives the best performance in combination with the first selected variable. This process continues until the pre-set criterion is achieved.

Graphical user interface, text, application

Description automatically generated

1. **Backward Feature Elimination**

This method works exactly opposite to the Forward Feature Selection method. Here, we start with all the features available and build a model. Next, we select the variable from the model which gives the best evaluation measure value. This process is continued until the pre-set criterion is achieved.

Graphical user interface, text, application

Description automatically generated

1. **Recursive Feature Elimination**

‘Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model). The goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a coef\_attribute or through a feature\_importance\_ attribute.

Then, the least important features are pruned from the current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

Text

Description automatically generated

**Embedded Methods**

These methods encompass the benefits of both the wrapper and filter methods, by including interactions of features but also maintaining reasonable computational cost. Embedded methods are iterative in the sense that takes care of each iteration of the model training process and carefully extract those features which contribute the most to the training for a particular iteration.

Let’s, discuss some of these techniques here:

1. **LASSO Regularization (L1)**

Regularization consists of adding a penalty to the different parameters of the machine learning model to reduce the freedom of the model, i.e. to avoid over-fitting. In linear model regularization, the penalty is applied over the coefficients that multiply each of the predictors. From the different types of regularization, LASSO of L1 has the property that can shrink some of the coefficients to zero. Therefore, that feature can be removed from the model.

Graphical user interface, text

Description automatically generated

1. **Random Forest Importance**

Random Forest is a kind of Bagging algorithm that aggregates a specified number of decision trees. The tree-based strategies used by random forests naturally rank by how well they improve the purity of the node, or in other words a decrease in the impurity (Gini impurity) over all trees. Nodes with the greatest decrease in impurity happen at the start of the trees, while the least decrease in impurity occurs at the end of trees. Thus, by pruning trees below a particular node, we can create a subset of the most important features.

Text

Description automatically generated

Chart, bar chart

Description automatically generated

So, these are the few feature selection techniques we use in the machine learning process.

**Loss Functions**

Loss functions are at the heart of the machine learning algorithms we love to use. but I have seen the majority of the beginners and the enthusiastic become quite confused regarding how when where to use them.

there are not difficult to understand and will enhance you are understanding of machine learning algorithms infinitely. So, what are loss functions and how can you grasp their meaning?

**What are lass functions?**

A loss function maps decisions to their associated costs. deciding to go up the slope will cost us energy and time. deciding to go down will benefit us therefore, it has a negative cost.

In supervised machine learning algorithms, we want to minimize the error for each training example during the learning process. this is done using some optimization strategies like gradient descent and this error comes from the loss function.

**What is the difference between a loss function and the cost function?**

I want to emphasize this here - although cost function and loss function is synonym and users interchangeably, they are different.

A loss function is a single training example. it is also sometimes called an error function.

A cost function, on the other hand, is the average loss over the entire training data set. the optimization strategies aim at minimizing the cost function.

**Types of loss functions**

Loss functions are broadly classified into three types

* Regression loss function
* Binary classification loss function
* Multi-class classification loss function

let's discuss each one briefly

**Regression loss function**

You must be quite familiar with regulation at this point. it deals with modeling a linear relationship between a dependent variable, Y, and several independent variables, X’s. thus, we essentially fit a line in space on this variable.

A picture containing chart

Description automatically generated

We will use the given data points to find the coefficients, a0, a1, …. An.

Chart, scatter chart

Description automatically generated

We will use the famous Boston housing data set for understanding this concept. and to keep this simple we will use only one feature - the average number of rules for dwelling(X) - to predict the dependent variable – Median value(Y) of Houses in $1000’s

Chart, scatter chart

Description automatically generated

We will use gradient descent as an optimization strategy to find the regression line. I will not go into the details about gradient descent but here is a reminder of the weight update rule:

Text

Description automatically generated

Here, theta\_j is the weight to be updated, alpha is the learning rate and J is the cost function. the cost function is parameterised by Theta. our aim is to find the value of Theta which yields minimum overall cost.

I have defined the steps that we will follow for each loss function below:

1. Write the expression for our predictor function, f(X), and identify the parameters that we need to find.
2. identify the loss to use for each training example
3. find the expression for the cost function - the average loss on all examples
4. find the gradient of the cost function with respect to each unknown parameter
5. decide on the learning rate and run the weight update rule for a fixed number of iterations\
6. **Squared Error Loss**

Squared error loss for each training example, also known as L2 Loss, is the square of the difference between the actual and the predicted values:

Diagram

Description automatically generated with low confidence

The corresponding cost function is the mean of these squared errors (MSE)

1. **Absolute Error Loss**

Absolute Error for each training example is the distance between the predictor and the actual values, irrespective of the sign. Absolute Error is also known as the L1 Loss.

A picture containing text, watch, clock, gauge

Description automatically generated

as I mentioned above, the cost is the mean of these absolute errors (MAE).

Text

Description automatically generated

**The MAE cost is more robust to outliers as compared to MSE**.

1. **Huber Loss**

The Huber Loss combines the best properties of MSE and MAE. it is quadratic for small errors and is linear otherwise. it is defined by its delta parameters

Text

Description automatically generated

**Binary Classification Loss Functions**

The name is pretty self-explanatory. binary classification refers to assigning an object into one of two classes. This classification is based on a rule applied to the input feature vector. for example, classifying an email as spam or not spam based on, say its subject line is binary classification.

1. **Binary Cross Entropy Loss**

let us start by understanding the term ‘entropy’. Generally, we use entropy to indicate disorder or uncertainty. it is measured for a random variable X with probability distribution p(X):

A picture containing table

Description automatically generated

The negative sign is used to make the overall quantity positive.

A greater value of entropy for a probability distribution indicates a greater uncertainty in the distribution. Likewise, a smaller value indicates a more certain distribution.

this makes binary cross-entropy suitable as the loss function - **you want to minimize its value.** We use binary cross-entropy loss for classification models which output a probability *p*.

Then, the cross-entropy loss for output label y (can take values 0 and 1) and predictive probability *p* is defined as:

A picture containing text

Description automatically generated

This is called Log-Loss. To calculate the probability *p*, we can use the sigmoid function. Here, z is a function of our input features:

A picture containing text, clock

Description automatically generated

The range of the sigmoid function is [0, 1] which makes it suitable for calculating the probability

Chart, histogram

Description automatically generated

1. **Hinge Loss**

***Hinge loss is primarily used to support victor Machine (SVM) classifiers with class levels-1 and 1.***

Hinge Loss not only penalizes the wrong predictions but also the right predictions that are not confident.

Hinge loss for any input-output pair (x, y) is given as:

A picture containing text, clock, watch, gauge

Description automatically generated

Hinge loss simplifies the mathematics for SCM while maximizing the loss. it is used when we want to make real-time decisions with not a laser-sharp focus on accuracy

**Multi-Class Classification Loss Function**

Emails are not just classified as spam or not spam (this isn’t the 90’s anymore!). They are classified into various other categories – work, home, social, promotions, etc. this is a Multi-Class Classification use case

1. **Multi-Class Cross Entropy Loss**

The multiclass cross-entropy loss is a generalization of the binary cross-entropy. the loss for input vector Xi and the corresponding one-hot encoded target vector Yi is:

Text

Description automatically generated with low confidence

Where's the Softmax function to find the probabilities Pij:

Text

Description automatically generated

“Softmax is implemented through a neural network layer just before the output layer. the softmax layer must have the same number of notes as output layer”.

Diagram

Description automatically generated

Finally, our output is a class with maximum probability for a given input

1. **KL-Divergence**

The Kullback-Liebler Divergence is a measure of how a property distribution differs from another distribution. A KL-divergence of Zero indicates that the distributions are identical.

Text, letter

Description automatically generated

Notice that the divergence function is not symmetric.



This is why KL-Divergence cannot be used as a distance metric.

I will describe the basic approach of using KL-Divergence as a loss function without getting into it's math. we want to approximate the true probability distribution p of our target variables with respect to the input features, given some approximate distribution Q. since KL-Divergence is not symmetric, we can do this in two ways:

Text

Description automatically generated

The first approach is used in supervised learning, the second is used in reinforcement learning. KL-Divergence is functionally similar to multiclass cross-entropy and is also called relative entropy of P with respect to Q:

Text, letter

Description automatically generated with medium confidence

We specify the “kullback\_leibler\_divergence” as the value of the loss parameter in the compile() function as we did before with the multi-class cross-entropy loss.

**Gradient Descent**

**What is Gradient Descent?**

The million-dollar question!

Let’s say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?

Take a moment to think about this before you read on.

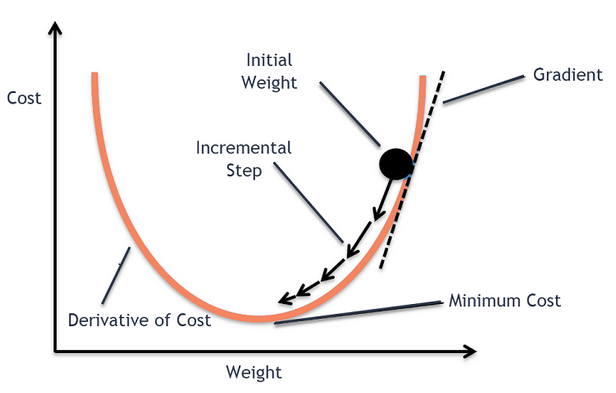
The best way is to observe the ground and find where the land descends. From that position, take a step in the descending direction and iterate this process until we reach the lowest point.



Source: Clairvoyant

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local minimum of the function, and the procedure is called Gradient Ascent.

Gradient descent was originally proposed by CAUCHY in 1847. It is also known as the steepest descent.



Source: Clairvoyant

The goal of the gradient descent algorithm is to minimize the given function (say cost function). To achieve this goal, it performs two steps iteratively:

1. **Compute the gradient** (slope), the first order derivate of the function at that point
2. **Make a step (move) in the direction opposite to the gradient**, opposite direction of slope increase from the current point by alpha times the gradient at that point

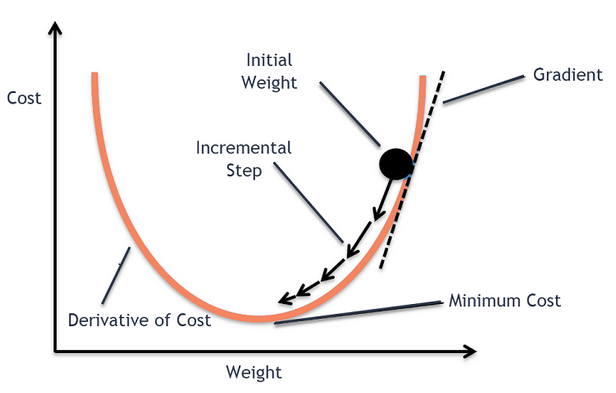
**Gradient descent algorithm**

**Wnew = Wold – (α \***∂E∂W**)**

So, why we need a derivative? If you see in the above diagram, to find whether the slop of the weight is positive or negative we have drawn a tangent. The derivate here in the formula helps us to find the positive or a negative slope of the weight.

There are two cases here

**Case 1: Positive slop**

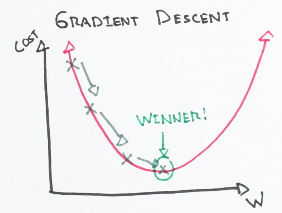


Here the weight which is initialized has a high loss and we need to update our weight such that we minimize the loss. Here the slope of the derivate or gradient is positive.

**Wnew = Wold – (α \***∂E∂W**)**

We will multiply our positive gradient with a learning rate, which generally ranges from 0.01 to 1. We subtract the product from the old weight to get a new weight. We iterate this process until we reach out a global minimum point.

**Case 2: Negative Slope**

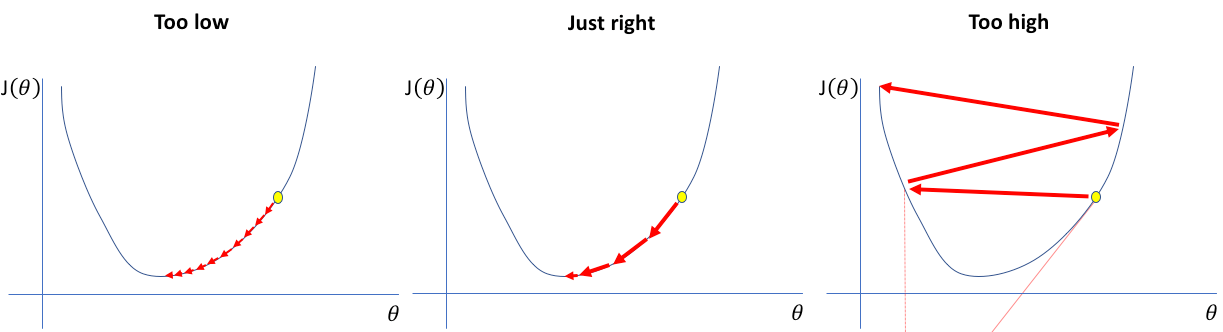


Here the slope of the initialized weight is negative, and we have to reach global minima,

**Wnew = Wold – (α \***∂E∂W**)**

We will multiply the negative slop with a negative alpha which becomes positive. So we will add the product to the old weight to get a new weight. We iterate this process until we reach the global minimum.

The important point to be noted over here is, how fast or how slow you reach the global minima depends on the learning rate we choose.



A small learning rate requires many updates before reaching the minimum point. Too large of a learning rate causes drastic updates which lead to divergent behaviors. The optimal learning rate swiftly reaches the minimum point.

**Disadvantages:**

Gradient descent uses the data of the entire training set to calculate the gradient of the cost function to the parameters. Because this method calculates **the gradient for the entire data set in one update, the calculation is very slow**, it will be very tricky to encounter a large number of data sets, and you cannot invest in new data to update the model in real-time.

**What is Artificial Neural Network?**

An Artificial Neural Network is inspired by biological computational models of brain and biological neural networks. In nutshell, an Artificial Neural Network (ANN) is a computational representation of the human neural network that regulates human intelligence, reasoning, and memory. However, why should we necessarily emulate a human brain system to develop efficient algorithms?

The main rationale behind using ANN is that neural networks are efficient in complex computations and hierarchical representation of knowledge. Neurons connected by axons and dendrites into complex neural networks can pass and exchange information, store intermediary computation results, produce abstractions, and divide the learning process into multiple steps. The computational model of such a system can thus produce very efficient learning processes similar to biological ones.

Diagram

Description automatically generated

Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of the interconnected perceptron.

The first layer is the input layer. Each node in this layer takes an input and then passes its output as the input to each node in the next layer. There are generally no connections between nodes in the same layer and the last layer produces the outputs.

We call the middle part the hidden layer. These neurons have no connection to the outside (e.g. input or output) and are only activated by nodes in the previous layer.

Diagram, schematic

Description automatically generated

Think of deep learning as the technique for learning in neural networks that utilizes multiple layers of abstraction to solve pattern recognition problems. In the 1980s, most neural networks were single layers due to the cost of computation and availability of data.

**Why is Deep Learning Important?**

Diagram

Description automatically generated

Over the past few years, Deep learning architectures and algorithms have impressive advances in fields such as image recognition and speech processing.

Their application to Natural Language Processing (NLP) was less impressive at first but now proven to make significant contributions, yielding state-of-art results for some common NLP tasks. Named entity recognition (NER), part of speech (POS) tagging, or sentiment analyses are some of the problems where neural network models have outperformed traditional approaches. The progress in machine translation is perhaps the most remarkable among all.

**McCulloch and Pitts Neuron**

In 1943, McCulloch and Pitts introduced a mathematical model of a neuron. It consisted of three components:

1. A set of weights ***wi*** corresponding to synapses (inputs)
2. An ***adder*** for summing input signals; analogous to the cell membrane that collects charge
3. An ***Activation Function*** for determining when the neuron fires, based on accumulated input.

The neuron model is shown mathematically below. On the right are input nodes {xi}, usually expressed as a vector. The strength with which the inputs are able to deliver the signal along the synapse is determined by their corresponding weights {***wi***}. The adder then sums the inputs from all the synapses.

Text

Description automatically generated

The parameter ***𝜃*** determines whether or not the neuron fires given a weighted input of ***h***. If it fires, it returns a value **y = 1**, otherwise **y = 0**. For example, a simple **activation function** is using 𝜃 as a simple fixed threshold:

A picture containing chart

Description automatically generated

This activation function may take any of several forms, such as a logistic function.

A picture containing text, clock

Description automatically generated

A single neuron is not interesting, nor useful, from a learning perspective. It cannot learn; it simply receives inputs and either fire or not. Only when neurons are joined as a network can they perform useful work.

Learning takes place by changing the weights of the connections in a neural network, and by changing the parameters of the activation functions of neurons.



Each neuron is independent of the others in the perceptron, in the sense that its behavior and performance depend only on its own weights and threshold values, and not of those for the other neurons. Though they share inputs, they operate independently.

The number of inputs and outputs is determined by the data. Weights are stored as an **N x K** matrix, with **N** observations and **K** neurons, with ***𝑤𝑖𝑗*** specifying the weight on ***i\*th*** *observation on the* ***j\**th** neuron.

Shape, polygon

Description automatically generated

In order to use the perceptron for statistical learning, we compare the outputs ***𝑦𝑗*** from each neuron to the observation target ***𝑡𝑗*,** and adjust the input weights when they do not correspond (*e.g.* if a neuron fires when it should not have).



We use this difference to update the weight *𝑤𝑖𝑗*, based on the input and the desired **learning rate**. This results in an update rule:



After an incremental improvement, the perceptron is shown the training data again, resulting in another update. This is repeated until the performance no longer improves. Having a learning rate of fewer than one results in a more stable learning rate, though this stability is traded off against having to expose the network to the data multiple times. Typical learning rates are in the 0.1-0.4 range.

An additional input node is typically added to the perceptron model, which is a constant value (usually -1, 0, or 1) that acts analogously to an intercept in a regression model. This establishes a baseline input for the case when all inputs are zero.

Shape

Description automatically generated

**Learning with Perceptron**

1. Initialize weights *𝑤𝑖𝑗* to small, random numbers.
2. For each t in T, iterations
   1. compute activation for each neuron *j\* connected to each input vector \*i*

Diagram

Description automatically generated

* 1. update weights



This algorithm is





The solution to fitting more complex (i.e. non-linear) models with neural networks is to use a more complex network that consists of more than just a single perceptron.

An MLP consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer.

Diagram, schematic

Description automatically generated

Except for the input nodes, each node is a neuron that uses a nonlinear activation function.

In any neural network, there are 3 layers present:

1. **Input Layer:** It functions similarly to that of dendrites. The purpose of this layer is to accept input.
2. **Hidden Layer:** These are the layers that perform the actual operation.
3. **Output Layer:** It functions similarly to that of axons. The purpose of this layer to transmit the generated output.

One thing to be noted here is that in the above diagram we have 2 hidden layers. But there is no limit on how many hidden layers should be here. It can be as low as 1 or as high as 100 or maybe even 1000!

Types of Artificial Neural Networks

Generally, there are two types of ANN. Such as Feed Forward and Back Propagation.

**Feed Forward Artificial Neural Networks**

In this network flow of information is unidirectional. A unit used to send information to another unit that does not receive any information. Also, no feedback loops are present in this. Although, used in recognition of a pattern. As they contain fixed input and outputs.

Diagram

Description automatically generated

Text

Description automatically generated

In the input layer, we have four neurons to feed values from four features. The number of neurons in the input layer should be equal to the attributes or features in the dataset.

In between the input layer and output layer, there are hidden layers based on the type of model. Here’s how the neural network computes the data in three simple steps:

1. **Multiplication of Weights and inputs**: The input is multiplied by the assigned weight values; let’s take a neuron ‘**E**’ from the hidden layer for an instance.

(X1 \* W1) = (0.5 \* 3) 🡺 1.5

(X2 \* W4) = (2.8 \* 1) 🡺 -0.75

(X3 \* W7) = (0 \* 5) 🡺 1

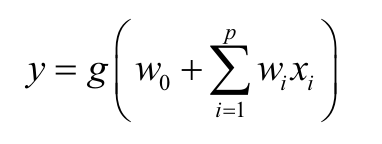
(X4 \* W10) = (-0.1 \* -4) 🡺 0.4

1. **Adding the biases:** In the next step, the product found in the previous step is summed up to a single value by adding some amount of bias.

(X1 \* W1) + (X2 \* W4) + (X3 \* W7) + (X4 \* W10) + bias

Weighted\_sum = 1.5+ (-0.75) + 1 + 0.4 + (-2) 🡺 2.7

1. **Activation:** The activation function helps to normalize the output of each neuron to a range between 1 and 0 (or) between -1 and 1.



**Output Signal:** Finally, the weighted sum obtained is turned into an output signal by feeding the weighted sum into an activation function (also called the transfer function). The output signal in the output layer will be numerical for the regression problem and Class label for the classification problem.

A similar calculation is done for all the subsequent neurons in all the hidden and output layers; remember that each neuron in all the layers will have an activation function except the input layer.

Let’s understand what are the different types of Weight Initialization techniques and different types of Activation functions

**Weights Initialization Techniques**

Before we get to see the different types of weight initialization techniques, let’s understand what are weights in a neural network?

**Weights** are the parameter within a neural network that transforms input data within the network’s hidden layers. Weights control the signal (or strength of the connection) between two neurons. In other words, a weight decides how much influence the input will have on the output.

We’ll have a look at some of the basic initialization practices in the use and some improved techniques that must be used in order to achieve a better result. Following are some techniques generally practiced to initialize parameters.

* Zero initialization
* Random initialization

**Zero Initialization**

In general practice biases are initialized with 0 and weights are initialized with random numbers, what if weights are initialized with 0?

In order to understand let us consider we applied the sigmoid activation function for the output layer.

Chart, line chart

Description automatically generated

If all the weights are initialized with 0, the derivative with respect to loss function is the same for every “W” in W[1], thus all weights have the same value in subsequent iterations. This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model. An important thing to keep in mind is that biases have no effect whatsoever when initialized with 0.



Let us consider a neural network with only three hidden layers with ReLU activation function in hidden layers and sigmoid for the output layer.

Using the above neural network on the dataset ‘make circles’ from sklearn.datasets, the result obtained as follows:

For 15000 iterations, loss = 0.6931, accuracy = 50%

Chart, scatter chart

Description automatically generated

Clearly, zero initialization isn’t successful in classification.

**Random initialization:**

Assigning random values to weights is better than just 0 assignments. But there is one thing to keep in my mind is that what happens if weights are initialized high values or very low values and what is a reasonable initialization of weight values.

1. If weights are initialized with very high values the term np.dot(W,X)+b becomes significantly higher and if an activation function like sigmoid() is applied, the function maps its value near to 1 where the slope of gradient changes slowly and learning takes a lot of time.
2. If weights are initialized with low values it gets mapped to 0,, where the case is the same as above.



A neural network is the same as earlier, using this initialization on the dataset ‘make circles’ from sklearn.datasets, the result obtained as the following:

For 15000 iterations, loss = 0.3827, accuracy = 86%

Chart

Description automatically generated

This solution is better but doesn’t properly fulfill the needs so, let us see a new technique.

**New Initialization techniques**

As we saw above that with zero initialization of weights, no significant result is obtained even if we use appropriate initialization of weights it is probable that the training process is going to take a longer time. There are certain problems associated with it:

1. If the model is too large and takes many days to train then what
2. What about the vanishing/exploding gradient problem

These were some problems that stood in the path for many years but in 2015, He et al. proposed activation aware initialization of weights (for ReLU) that was able to resolve this problem. ReLU and Leaky ReLU also solve the problem of vanishing gradient.

**He initialization:** we just simply multiply random initialization with

Text

Description automatically generated

To see how effective this solution is, let us use the previous dataset and neural network we took for the above initialization and results are:

For 15000 iterations, loss = 0.0735, accuracy = 96%

Chart

Description automatically generated

Surely, this is an improvement over the previous techniques.

There are also some other techniques other than He initialization in use that is comparatively better than old techniques and are used frequently.

Xavier initialization: It is the same as He initialization but it is used for the ‘tanh’ activation function, in this method 2 is replaced with 1.

Diagram, schematic

Description automatically generated

Some also use the following technique for initialization:

Table

Description automatically generated

These methods serve as good starting points for initialization and mitigate the chances of exploding or vanishing gradient. They set the weights neither too much bigger than 1 nor too much less than 1. So, the gradients do not vanish or explode too quickly. They help avoid slow convergence, also ensuring that we do not keep oscillating off the minima. There exist other variants of the above, where the main objective again is to minimize the variance of parameters.

We have seen what are weights, how weights are initialized, and different types of weights initialization techniques. Let’s move ahead and see what happens in a neuron, what is an activation function and the roles of different activation functions.



**Why do we need Non-linear activation functions?**

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

**Mathematical proof:-**

Suppose we have a Neural net like this:-

Diagram

Description automatically generated

Elements of the diagram:-

**Hidden layer i.e. layer 1:-**

Z(1) = W(1)X + b(1)

a(1) = Z(1)

Here,

* Z(1) is the vectorized output of layer 1.
* W(1) is the vectorized weights assigned to neurons of the hidden layer i.e. W1, W2, W3, and W4.
* X be the vectorized input features i.e. i1 and i2.
* B is the vectorized bias assigned to neurons in the hidden layer i.e b1 and b2.
* a(1) is the vectorized form of any linear function.

**Note:** We are not considering activation function here

**Layer 2 i.e. output layer:-**

Z(2) = W(2)a(1) + b(2)

a(2) = Z(2)

**Note:** Input for layer 2 is output from layer 1

**Calculation at Output layer:**

Putting value of Z(1) here

Z(2) = (W(2) \* [W(1)X + b(1)]) + b(2)

Z(2) = [W(2) \* W(1)] \* X + [W(2)\*b(1) + b(2)]

Let,

[W(2) \* W(1)] = W

[W(2)\*b(1) + b(2)] = b

Final output:

Z(2) = W\*X + b

This is again a linear function.

This observation results again in a linear function even after applying a hidden layer, hence we can conclude that doesn’t matter how many hidden layers we attach in the neural network, all layers will behave the same way because the composition of two linear functions is a linear function itself. A neuron cannot learn with just a linear function attached to it. A non-linear activation function will let it learn as per the difference w.r.t error.

**Popular types of activation functions and when to use them**

1. **Binary Step Function**

The first thing that comes to our mind when we have an activation function would be a threshold-based classifier i.e. whether or not the neuron should be activated based on the value from the linear transformation.

In other words, if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e. its output is not considered for the next hidden layer.

Graphical user interface, text

Description automatically generated with medium confidence

Chart, box and whisker chart

Description automatically generated

This is the simplest activation function, which can be implemented with a single if-else condition in python.

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Output:

Shape, rectangle

Description automatically generated

The binary step function can be used as an activation function while creating a binary classifier. As you can imagine, this function will not be useful when there are multiple classes in the target variable. That is one of the limitations of a binary step function.

Moreover, the gradient of the step function is zero which causes a hindrance in the backpropagation process. That is, if you calculate the derivative of f(x) w.r.t x, it comes out to be 0.

A picture containing text

Description automatically generated

Chart

Description automatically generated

Gradients are calculated to update the weights and biases during the backpropagation process. Since the gradient of the function is zero, the weights and biases don’t update.

1. **Linear Function**

We saw the problem with the step function, the gradient of the function became zero. This is because there is no component of x in the binary step function. Instead of a binary function, we can use a linear function. We can define the function asShape, rectangle

Description automatically generated

Chart, line chart

Description automatically generated

Here the activation is proportional to the input. The variable ‘a’, in this case, can be any constant value. Let’s quickly define the function in python:

Graphical user interface

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

Output:

Graphical user interface, text

Description automatically generated

What do you think will be the derivative in this case? When we differentiate the function w.r.t X, the result is the coefficient of X, which is a constant.

Graphical user interface

Description automatically generated with low confidence

A picture containing calendar

Description automatically generated

Although the gradient here does not become zero, it is a constant which does not depend upon the input value x at all. This implies that the weights and biases will be updated during the backpropagation process but the updating factor would be the same.

In this scenario, the neural network will not really improve the error since the gradient is the same for every iteration. The network will not be able to train well and capture the complex patterns from the data. Hence, the linear function might be ideal for simple tasks where interpretability is highly desired.

The linear activation function is used at just one place i.e. output layer for solving the regression problem. Even in this case, the neural net must have any non-linear function at hidden layers.

1. **Sigmoid**

The next activation function that we are going to look at is the sigmoid function. It is one of the most widely used non-linear activation functions. Sigmoid transforms the value between the range 0 and 1. Here is the mathematical expression for sigmoid

Diagram

Description automatically generated

Chart, histogram

Description automatically generated

A noteworthy point here is that unlike the binary step and linear functions, sigmoid is a non-linear function. This essentially means when I have multiple neurons having a sigmoid function as their activation function, the output is non-linear as well. The output range of the sigmoid function is 0 to 1.

Here is the python code for defining the function in python

Text, letter

Description automatically generated

Text

Description automatically generated

Output:

Text

Description automatically generated

Additionally, as you can see in the graph above, this is a smooth S-shaped function and is continuously differentiable. The derivative of this function comes out to be (Sigmoid(X)\*(1-Sigmoid(X)). Let’s look at the plot of its gradient.

Text

Description automatically generated with low confidence

Chart, line chart

Description automatically generated

The gradient values are significant for range -3 and 3 but the graph gets much flatter in other regions. This implies that for values greater than 3 or less than -3, we have very small gradients. As the gradient value approaches zero, the network is not really learning.

Additionally, the sigmoid function is not symmetric around zero. So the output of all the neurons will be of the same sign. This can be addressed by scaling the sigmoid function which is exactly what happens in the “tanh” function.

1. **Tanh Function**

The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values, in this case, is from -1 to 1. Thus the inputs to the next layers will not always be of the same sign. The tanh function is defined as

A picture containing text, watch

Description automatically generated

Diagram

Description automatically generated

In order to code this in python; let us simplify the previous expression.

Text

Description automatically generated with medium confidence

Text

Description automatically generated

And here is the python code for the same:

Text, letter

Description automatically generated

Text

Description automatically generated with low confidence

Output:

Text

Description automatically generated with low confidence

As you can see, the range of values is between -1 to 1. Apart from that, all other properties of tanh function are the same as that of the sigmoid function. Similar to the sigmoid, the tanh function is continuous and differentiable at all points.

Let’s have a look at the gradient of the tanh function.

Chart, line chart

Description automatically generated

The gradient of the tanh function is steeper as compared to the sigmoid function.

1. **ReLU**

The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. The plot below will help you understand this better.

Text, letter

Description automatically generated

Chart, line chart

Description automatically generated

For the negative input values, the result is zero that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions. Here is the python function for ReLU:

Graphical user interface, text, application

Description automatically generated

Chart

Description automatically generated with low confidence

Output:

Shape

Description automatically generated

Let’s look at the gradient of the ReLU function.

Graphical user interface, text, application

Description automatically generated

Chart

Description automatically generated

If you look at the negative side of the graph, you will notice that the gradient value is zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated. This is taken care of by the ‘Leaky’ ReLU function.

1. **Leaky ReLU**

Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that, for the ReLU function, the gradient of 0 for x<0, which would deactivate the neurons in that region.

Leaky ReLU is defined to address this problem. Instead of defining the ReLU function as 0 for negative values of X, we define it as an extremely small linear component of X. Here is the mathematical expression

Text

Description automatically generated

Chart, line chart

Description automatically generated

By making this small modification, the gradient of the left side of the graph comes to be a non-zero value. Hence we would no longer encounter dead neurons in that region. Here is the derivative of the leaky ReLU function.

Text

Description automatically generated

Chart

Description automatically generated

Since Leaky ReLU is a variant of ReLU, the python code can be implemented with a small modification.

Text

Description automatically generatedChart

Description automatically generated with low confidence

Output:

Shape, rectangle

Description automatically generated

1. **Parameterized ReLU**

This is another variant of ReLU that aims to solve the problem of gradient’s becoming zero for the left half of the axis. The parameterized ReLU, as the name suggests, introduces a new parameter as a slope of the negative part of the function. Here’s how the ReLU function is modified to incorporate the slope parameter.

Text

Description automatically generated

Diagram, schematic

Description automatically generated

When the value of ‘a’ is fixed to 0.01, the function acts as the Leaky ReLU function. However, in the case of parameterized ReLU function, ‘a’ is also a trained parameter. The network also learns the value of ‘a’ for faster and more optimum convergence.

The derivative of the function would be the same as the Leaky ReLU function, except the value 0.01 will be replaced with the value of ‘a’.

A picture containing rectangle

Description automatically generated

The parameterized ReLU function is used when the leaky ReLU function still fails to solve the problem of dead neurons and the relevant information is not successfully passed to the next layer.

1. **Softmax**

The softmax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems.

The softmax function can be used for multiclass classification problems. This function returns the probability for a data point belonging to each individual class. Here is the mathematical expression of the same

A picture containing text

Description automatically generated

While building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance, if you have three classes, there would be three neurons in the output layer. Suppose you got the output from neurons as [1.2, 0.9, 0.75].

Applying the softmax function over these values, you will get the following result [0.42, 0.31, 0.27]. These represent the probability for the data point belonging to each class.

**Note:** The sum of all the values is 1.

Text

Description automatically generatedA picture containing text

Description automatically generated

Output:

A picture containing shape

Description automatically generated

**Which activation function is the best to use?**

Now that we have seen so many activation functions, we need some to know which activation function should be used in which situation.

‘Sigmoid’ & ‘tanh’ functions are for binary classification and they cannot be a possible use for the multiclass tasks we would be using the ‘softmax’ activation function, as it would output a vector of probabilities of each class. We mostly use Sigmoid & Softmax in our output layer.

**Which activation would be better to use in the hidden layer part of a multi-layered neural net?**

If you use the sigmoid activation function then you would be hit by the vanishing gradient problem (The vanishing gradient problem is when the gradient becomes so small in the earlier layers of a deep neural network to the point that it barely affects the weights of the earlier layers thus failing at optimizing our initial weights).

Should we use tanh function then? Not necessarily, it’s true that the tanh function is an enhanced version of the sigmoid function but the vanishing gradient problem is still persistently there.

So the solution to our persistent problem is the ReLU activation function, but how does it solve the vanishing gradient problem?

The fact that the derivative of the ReLU activation function is 1 if its inputs are positive will assure that the gradient will not get infinitely smaller passing from one layer to another.

But, what if our inputs are negative? Here we get stuck with what we call the “dying ReLU” problem (the dying ReLU problem is when the ReLU activation function always outputs the same value (zero) for any input, which means it takes no role in discriminating between inputs) Once a ReLU ends up in this state, it is unlikely to recover because the function of the gradient is ‘0’ is always ‘0’, so gradient descent learning will not alter weights).

Here comes the leaky ReLU/Parametric ReLU to rescue and instead of outputting a flat-out zero for the negative values. The leaky ReLU multiplies the negative value by an alpha parametric value (alpha = 0.01).

Once we get our predicted value from the output layer we will calculate the loss using the loss function.

**Backpropagation in neural network**

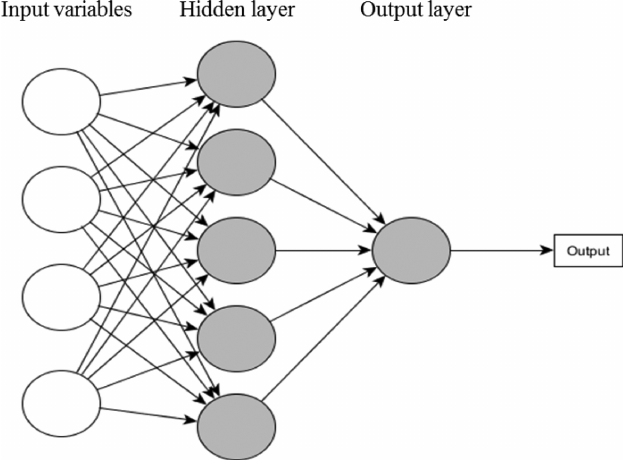
Backpropagation in a neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

**How Back Propagation Algorithm Works**

Let’s say we want to use the neural network to predict house prices. For our understanding purpose here, we will take a subset dummy dataset having four input variables and six observations here with input having a dimension of 4\*5:

| **X1** | **X2** | **X3** | **X4** |
| --- | --- | --- | --- |
| 0.00632 | 18 | 2.31 | 0 |
| 0.02731 | 0 | 7.07 | 0 |
| 0.02727 | 0 | 7.07 | 0 |
| 0.03237 | 0 | 2.18 | 0 |
| 0.06905 | 0 | 2.18 | 0 |

The neural network for this subset data looks like below:



The architecture of the neural network is [4, 5, 1] with:

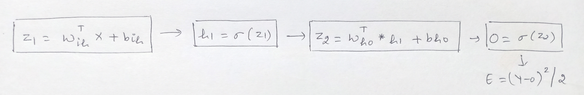
* 4 independent variables, Xs in the input layer
* 5 nodes in the hidden layer, and
* Since we have a regression problem at hand, we will have one node in the output layer.

A Neural Network operates by:

1. Initializing the weights with some random values, which are mostly between 0 and 1.
2. Compute the output to calculate the loss or the error term.
3. Then, adjust the weights so that to minimize the loss.

We repeat these steps until have reached the optimum solution of the minimum loss function or exhausted the predefined epochs (i.e. the number of iterations)

Now, the computation graph after applying the sigmoid activation function is:



Building on this, the first step in Backward Propagation to calculate the error. In our regression problem, we shall take the loss function

(Y-Yhat)^22

Where:

* Y is the actual value
* Yhat is the predicted value.

For simplicity, replacing Yhat with 0, so the error E becomes

(Y-0)^22

Our goal is to minimize the error that is clearly dependent on Y, which is the actual observation values, and on the output, which is further dependent on the:

* Input values
* Coefficient or Weights of the input variables
* Biases, the activation function, and Optimizers

Now, we can neither change the input variables nor the actual Y values. However, we can change the other factors. The activation function and the optimizers are the tuning hyperparameters – and we can change these based on our requirements.

The other two factors: the coefficients or weights of the input variables **(Wi’s)** and the biases **(bho, bih)** are updated using the optimizer algorithm with the following equation:

**Wnew = Wold – (α \***∂E∂W**)**

Where:

* **Wnew** = The new weight of Xi
* **Wold** = The old weight of the Xi
* **α** = learning rate
* ∂E∂Wis the partial derivate of the error for each of the Xs. It is the rate of change of error to the change in weight.

In the backpropagation, we adjust these weights in the output. The weights and biases between the respective input, hidden and output layers we have are **Wih, bih, Who, and bho:**

* **Wih:** weight between the input and hidden layer
* **bih** : bias between the input and hidden layer
* **Who** : weight between the hidden and the output layer
* **bho** : bias between the hidden and the output layer

In the first iteration, we randomly initialize the weights. In the second iteration, we change the weights of the hidden layer that is closest to the output layer. In this case, we go from the output layer, hidden layer, and then to the input layer.

**Contribution of each Weight and Bias on the Error**

Now, we have to calculate how much each of these weights **(Wi’s)** and biases **(bi’s)** contribute to the error term. For this, we need to calculate the rate of change of error to the respective weights and bias parameters.

In other words, we need to compute the terms: ∂E∂Wih**,** ∂E∂bih**,** ∂E∂Who**,** and ∂E∂bho

This is not a direct task. It is a series of steps involving the **Chain Rule**.

**The Weight, Who, between the hidden and output layer**:

From the above graph we can see that the error E is not directly dependent on the **Who**.

* The error term is dependent on the Output O
* Output O is further dependent on **Z2**, and
* **Z2** is dependent on Who

Therefore we employ the chain rule to compute the rate of change in error to the change in weight **Who** and it becomes:

∂E∂Who **=** ∂E∂O **\*** ∂O∂Z2 **\*** ∂Z2∂Who

  Now, we take the partial derivatives of each of these individual terms:

(Y-0)^22

* The partial derivate of error with respect to output is:

∂E∂O **= 2\*(Y-O)\*(-1) = (0 – Y)**

* The partial derivative of Output with respect to **Z2**, as output O = Sigmoid of **Z2** and the derivative of sigmoid is:

∂O∂Z2 **= sigmoid(Z2)\*(1-sigmoid(Z2)) = 0\*(1-0)**

* The partial derivative of **Z2** with respect to **Who** is:

∂Z2∂Who **= (WhoT\* h1 + bho)/Who**

**= (WhoT \* h1)/** ∂**Who + (bho/Who)**

∂Z2∂Who **= h1 + 0 = h1**

    Therefore,

∂E∂Who **=** ∂E∂O **\*** ∂O∂Z2 **\*** ∂Z2∂Who

    Becomes

∂E∂Who **= (0-Y) \* 0 \* (1-0) \* h1**

Similarly, we will calculate the contribution for each of the other parameters in this manner.

For the bias, **bho,** between the hidden and the output layer:

∂E∂bho **=** ∂E∂O **\*** ∂O∂Z2 **\*** ∂Z2∂bho

∂E∂bho **= (0 – Y) \* 0 \* (1 – 0) \* 1**

**The weight, Wih, between the input and hidden layer:**

From the above graph we can see that the terms are dependent as below:

* The error term is dependent on the Output **O**
* Output **O** is dependent on **Z2**
* **Z2** this time is dependent on **h1**
* **h1** is dependent on **Z1**, and
* **Z1** is dependent on **Wih**

∂E∂Wih **=** ∂E∂O **\*** ∂O∂Z2 **\*** ∂Z2∂h1 **\*** ∂h1∂Z1 **\*** ∂Z1∂Wih

So, this time, apart from the initial above

∂E∂O**,** ∂O∂Z2

We have the partial derivatives as follow:

* The partial derivative of **Z2** with respective to **h1** is:

∂Z2∂h1 **= (WhoT\* h1 + bho)/h1**

**= (WhoT \* h1)/** ∂**h1 + (bho/h1)**

∂Z2∂h1 **= Who + 0 = Who**

* The partial derivative of h1 with respect to **Z1**, as h1 = Sigmoid of **Z1** and the derivative of Sigmoid is:

∂h1∂Z1 **= sigmoid(Z1)\*(1-sigmoid(Z1)) = h1\*(1-h1)**

* The partial derivate of **Z1** with respect to **Wih** is: X

∂Z1∂W1h **= (WihT\* X + bih)/Wih**

**= (WihT\* X)/** ∂**Wih + (bih /Wih)**

∂Z2∂h1 **= X + 0 = X**

Hence, the equation after plugging the partial derivative value is:

∂E∂Wih **=** ∂E∂O **\*** ∂O∂Z2 **\*** ∂Z2∂h1 **\*** ∂h1∂Z1 **\*** ∂Z1∂Wih

∂E∂Wih **= (0 – Y) \* 0 \* (1 – 0) \* Who \* h1\*(1-h1) \* X**

Now, that we have computed these terms we can update the parameters using the following respective update equations:

* **Wih = Wih – (α \*** ∂E∂Wih**)**
* **bih = bih – (α \*** ∂E∂bih**)**
* **Who = Who – (α \*** ∂E∂Who**)**
* **bho = bho – (α \*** ∂E∂bho**)**

**Optimizers**

We have discussed Gradient Descent at the beginning of our chapter. However, there are few more optimizing algorithms

**Why optimizers are required?**

Instead of changing weights manually, optimizers can update weights automatically in small increments and help to find the minimum value of the loss/cost function. **Magic of DL!!**

Finding the minimum value of the cost function requires iterating through the dataset many times and thus requires large computational power.

 There are different optimization techniques in deep learning

1. Momentum
2. Adagrad
3. Adadelta
4. RMSprop
5. Adam

Let’s understand how each optimizer algorithm works

**SGD (Stochastic Gradient Descent)**

Compared with the Gradient Descent calculation of gradients with all data at one time, SGD updates the gradient of each sample with each update. For large datasets, there may be similar samples, so Gradient Descent calculates the gradient. There will be redundancy and SGD is updated only, there is no redundancy, it is faster, and new samples can be added.

**Disadvantages**

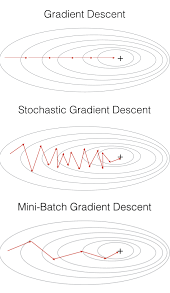
However, because SGD is updated more frequently, the cost function will have severe oscillations. When we decrease the learning rate slightly, the convergence of SGD and GD is the same.

**MBGD (Mini-batch Gradient Descent)**

MBGD uses a small batch of samples, that is, n samples to calculate each time. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. It can make full use of the highly optimized matrix operations in the deep learning library for more efficient gradient calculations.

**Disadvantages**

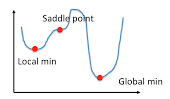
MBGD does not guarantee good convergence, if the learning rate is too small, the convergence rate will be slow, If it is too large, the loss function will oscillate or even deviate at the minimum value. One measure is to set a **larger learning rate**. When the change between two iterations is lower than a certain threshold, the learning rate is reduced.



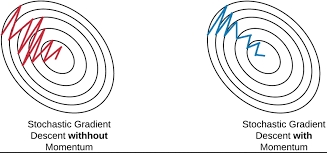
However, the setting of this threshold needs to be written in advance to adapt to the characteristics of the data set.

In addition, this method is to apply the **same learning rate** to all parameter updates. If our data is sparse, we would prefer to update the features with a lower frequency. For **non-convex functions**, it is also necessary to avoid trapping at the local minimum or saddle point, because the error around the saddle point is the same, the gradients of all dimensions are close to 0, and SGD is easily trapped here.

**Saddle points** are the curves, surfaces, or hypersurfaces of a saddle point neighborhood of a smooth function are located on different sides of a tangent to this point. For example, this two-dimensional figure looks like a saddle: it curves up in the x-axis direction and down in the y-axis direction, and the saddle point is (0, 0).



**Momentum,** which simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.



**Adagrad**

Adagrad is an algorithm for gradient-based optimization which adapts the learning rate to the parameters, using low learning rates for parameters associated with frequently occurring features, and using high learning rates for parameters associated with infrequent features.

So, it is well-suited for dealing with sparse data.

But the same update rate may not be suitable for all parameters. For example, some parameters may have reached the stage where only fine-tuning is needed, but some parameters need to be adjusted a lot due to the small number of corresponding samples.

Adagrad proposed this problem, an algorithm that adaptively assigns different learning rates to various parameters among them. The implication is that for each parameter, as its total distance updated increases, its learning rate also slows.

**Adadelta**

There are three problems with the Adagrad algorithm

* The learning rate is monotonically decreasing.
* The learning rate in the late training period is very small.
* It requires manually setting a global initial learning rate.

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad’s aggressive, monotonically reducing the learning rate.

It does this by restricting the window of the past accumulated gradient to some fixed size of w. running average at time t then depends on the previous average and the current gradient.

In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

**RMSProp**

The full name of RMSProp algorithm is called **Root Mean Square Prop**, which is an adaptive learning rate optimization algorithm proposed by Geoff Hinton.

RMSProp tries to resolve Adagrad’s radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.

Adagrad will accumulate all previous gradient squares, and RMSprop just calculates the corresponding average value, so it can alleviate the problem that the learning rate of the Adagrad algorithm drops quickly.

The difference is that RMSProp calculates the **differential squared weighted average of the gradient**. This method is beneficial to eliminate the direction of large swing amplitude and is used to correct the swing amplitude so that the swing amplitude in each dimension is smaller. On the other hand, it also makes the network function converge faster.

In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

RMSProp divides the learning rate by the average of the exponential decay of squared gradients

**Adam**

**Adaptive Moment Estimation (Adam)** is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop.

* Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

* Adam can be viewed as a combination of Adagrad and RMSprop,(Adagrad) which works well on sparse gradients, and (RMSProp) which works well in online and non-stationary settings respectively.

* Adam implements the **exponential moving average of the gradients** to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients.

* Adam is computationally efficient and has very little memory requirement.

* Adam optimizer is one of the most popular and famous gradient descent optimization algorithms.

**How to choose optimizers?**

* If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
* RMSprop, Adadelta, Adam have similar effects in many cases.
* Adam just added bias-correction and momentum on the basis of RMSprop,
* As the gradient becomes sparse, Adam will perform better than RMSprop.

**Overall, Adam is the best choice.**

SGD is used in many papers, without momentum, etc. Although SGD can reach a minimum value, it takes longer than other algorithms and may be trapped in the saddle point.

If faster convergence is needed, or deeper and more complex neural networks are trained, an adaptive algorithm is needed.

By this time you would have got a clear understanding of neural networks, how the learning happens in backpropagation, different activation functions, and optimizers to build an efficient neural network.

**What is Pattern Recognition?**

Pattern recognition is a process of finding regularities and similarities in data using neural networks. Now, these similarities can be found based on statistical analysis, historical data, or the already gained knowledge by the machine itself.

a pattern is a regularity in the world are in abstract notions. if we discuss sports, a description of a type would be a pattern. if a person keeps watching videos related to football YouTube would not recommend them chess tutorial videos.

**Examples:** speech recognition, identification, multimedia document recognition, and automatic medical diagnosis.

Before searching for a pattern there are some certain steps end the first one is to collect the data from the real world. the collected data needs to be filtered and pre-processed so that its system can extract the features from the data. then based on the type of data system will choose the appropriate algorithm among classification, regression, and regression to recognize the pattern.

**How does it work?**

A pattern recognition system will receive some input from the real world with sensors. such a system can work with any type of data: images, texts, videos, or numbers.

After receiving some information as the input, the algorithm starts to pre-process the data. that is segmenting something interesting from the background. for example, when you are given a photo of a park and familiar face or any object that attracts the user’s attention is this is pre-processing.

While the data is in the pre-processing phase it is important to filter the noise from the main data set. depending on the working function of the application, the filter algorithm will change. for example, consider a face recognition system where the system is collecting the images for training purposes.

In order to process the data, it will first convert the images from RGB to grayscale. Also, the system doesn't need other areas than the face. so to filter out unwanted portions of the images and replace them with white or black backgrounds some filter mechanisms are required. once those filter mechanisms are used on the data it will be easier for the system to extract features from the filtered images.

**Feature Extraction**

It is a process of uncovering some characteristic traits that are similar to more than one data sample. the diversified information may be general features, which are evaluated to is further processing. for example, in image recognition, the extracted features will contain information about the grey shade, texture, shape, or context of the image. this is the main information used in image processing. these methods of future extraction and the extracted features are application-dependent.

After extracting the features from the processed data, the result of the pattern recognition system will be either a class segment or cluster segment or predicted values.

**Neural Approach**

There are quite a few approaches for pattern recognition like statistical, syntactical, and neural. The statistical approach is nothing but to collect historical data and based on the observations and analysis from those data new patterns are recognized. the syntactical approach is also known as the structural approach as it mainly relies upon subpatterns called primitives like words.

An artificial neural network is a computing system that tries to stimulate the working function of a biological neural network of human brains. in this network, all the neurons are well connected and that helps to achieve massive parallel distribution. the input units receive various forms and structures of information based on an internal weighting system and the neural network attempts to learn about the information presented to produce one output error.

the advantage of neural networks is their adaptive learning, self-organization, and fault-tolerance capabilities. for these outstanding capabilities, neural networks are used for pattern recognition applications. and artificial neural network initially goes through a training phase where it learns to recognize patterns in data, whether visually, orally, or textually. some of the best neural models are backpropagation, high order Nets, time-delay neural networks, and recurrent nets.

A picture containing chart

Description automatically generated

Normally, only feedforward networks are used for pattern recognition. feedforward means that there is no feedback to the input. similarly, to the way that human beings learn from mistakes, neural networks also could learn from their mistakes by giving feedback to the input patterns. this kind of feedback would be used to reconstruct the input patterns and make them free from error; thus, increasing the performance of neural networks. Of course, it is very complex to construct search types of neural networks. These kinds of networks are called auto-associative networks. this complexity of constructing the network can be avoided by using backpropagation algorithms. during this supervisor’s face, the network compares its actual output producer with what it was meant to produce – the desired output.

The difference between both outcomes is adjusted using backpropagation. this means that the network works backward, going from the output unit to the input units to adjust the weight of its connections between the units until the difference between the actual and desired outcome produces the lowest possible error. local minima is one of the main problems associated with backpropagation algorithms. in addition, neural networks have issues associated with hyperparameters like learning rate, architectural selection, feature representation, modularity, and scaling. Though there are problems and obstacles, the application of neural networks has spread everywhere.

Some of the real-world applications are

* Image processing, segmentation, and analysis.
* Computer vision
* Speech recognition
* Fingerprint identification
* Medical diagnosis
* Stock market analysis

**Practical Implementation of Neural Networks**

**Case Study**

**Credit Card Fraud Detection using Artificial Neural Networks**

Credit card fraud is when someone uses another person’s credit card or account information to make unauthorized purchases or access funds through cash advances. credit card fraud doesn't just happen online; it happens in brick-and-mortar stores, too. as a business owner, you can avoid serious headaches and - unwanted publicity - by recognizing potentially fraudulent dental use of credit cards in your payment environment.

**Three challenges surrounding credit card fraud**

1. it's not always easy to agree on ground truth for what ‘fraud’ means
2. regardless of how you define ground truth, the vast majority of chargers are not fraudulent.
3. most merchants aren't experts at evaluating the business impact of fraud.

**Problem Statement:**

the credit card fraud detection problem includes modeling past credit card transactions with the knowledge of ones that turns out to be a fraud. this model is then used to identify whether a new transaction is fraudulent or not. our aim here is to detect 100% of fraudulent transactions while minimizing the incorrect fraud classifications.

**Importing required libraries and data**

Text

Description automatically generated

Text

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

**Exploratory Data Analysis**



Table

Description automatically generated

Graphical user interface, application

Description automatically generated

Text

Description automatically generated

Let’s visualize the target variable

Graphical user interface, text

Description automatically generated

Chart, bar chart

Description automatically generated

Graphical user interface, text

Description automatically generated

Notice how imbalanced is our original data set! most of the transactions are not a fraud. if we use this data frame as a base for our predictive models and analysis, we might get a lot of errors and our algorithms will probably overfit since it will “assume” that most transactions are not a fraud. but we don't want our model to assume we want our model to detect patterns that give the science of fraud!

Determine the number of fraud and valid transitions in the entire data set.

A picture containing graphical user interface

Description automatically generated

Schematic

Description automatically generated with medium confidence

How different is the amount of money used in different transaction classes?

Graphical user interface

Description automatically generated

Table

Description automatically generated

There is not much difference in basic statistics between fraud and non-fraud transactions.

Do fraudulent transactions occur more often during a certain time frames?

Graphical user interface, text

Description automatically generated

Chart, histogram

Description automatically generated

Even the time seems pretty much similar between the fraud and non-fraud transactions.

Visualizing the correlation using heatmap.

Chart, histogram

Description automatically generated

Highest Correlations come from:

Text, letter

Description automatically generated

* While these correlations are high, I don't expect it to run the risk of multicollinearity.
* The correlation matrix shows also that none of the V1 to V28 PCA components have any correlation to each other.

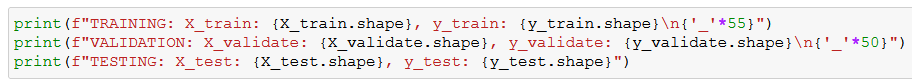
**Data Pre-processing**

Time and Amount should be scaled as the other columns.

Text, application

Description automatically generated

Train and Test Split



Text, table

Description automatically generated

Let’s create a function to evaluate the performance of the model on train and test.

A picture containing table

Description automatically generated

Creating a sequential model object.

Graphical user interface, text

Description automatically generated

We have created a “Sequential” model object, then we have added the layers accordingly. As you can see, we have used ‘relu’ activation function in our hidden layers and “sigmoid” in our output layer.

Table

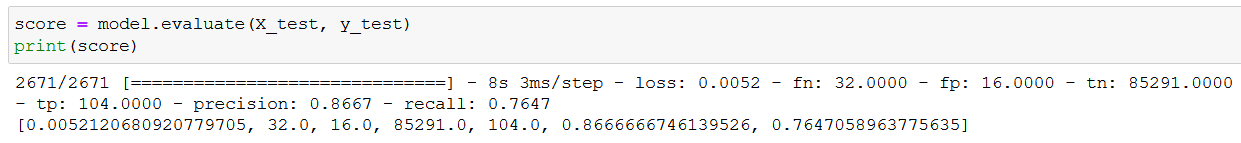
Description automatically generated

The total parameters used in the neural network are 142,849. The trainable parameters are 141, 313 and non-trainable parameters are 1,536.

Graphical user interface, text, application

Description automatically generated

Model Evaluation on the test set.



Text

Description automatically generated

Chart, line chart

Description automatically generated

We can see the training loss and validation loss have decreased parallel, which is a sign of a generalized model.

Table

Description automatically generated

Excellent! The model has worked perfectly in all the evaluation metrics.

**Natural Language Processing with Neural Networks**

**Introduction**

Let me start this topic with a question – “working love learning we on deep”, did this make any sense to you? Not really – read this one – “we love working on deep learning”. Made perfect sense! A little jumble in the words made the sentence incoherent. Well, can we expect a neural network to work sense out of it? Not really! If the human brain was confused about what it meant I am sure a neural network is going to have a tough time deciphering such text.

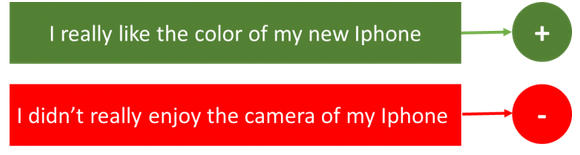
There are multiple such tasks in everyday life that get completely disrupted when their sequence is disturbed. For instance, language as we saw earlier – the sequence of words defines their meaning, a time series data – where time defines the occurrence of events, the data of a genome sequence – where time defines the occurrence of events, the data of a genome sequence – where every sequence has a different meaning. There are multiple such cases wherein the sequence of information determines the event itself. If we are trying to use such data for any reasonable output, we need a network that has access to some prior knowledge about the data to completely understand it. Recurrent neural networks thus come into play.

**Need for a Neural Network dealing with Sequences**

Before we deep dive into the details of what a recurrent neural network is, let’s ponder a bit on if we really need a network especially for dealing with sequences in information. Also, what are the kinds of tasks that we can achieve using such networks?

The beauty of recurrent neural networks lies in their diversity of application. When we are dealing with RNNs they have a great ability to deal with various input and output types.

* **Sentiment Classification** – This can be a task of simply classifying tweets into positive and negative sentiment. So here the input would be a tweet of varying lengths, while output is of a fixed type and size.



* **Image Captioning** – Here, let’s say we have an image for which we need a textual description. So we have a single input – the image, and a series or sequence of words as output. Here the image might be of a fixed size, but the output is a description of varying lengths.



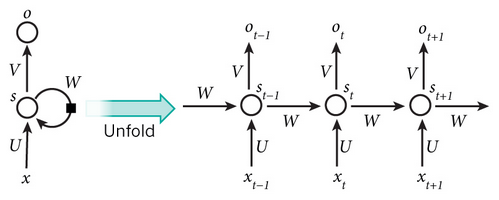
* **Language Translation** - This basically means that we have some text in a particular language let’s say English, and we wish to translate it in French. Each language has it’s own semantics and would have varying lengths for the same sentence. So here the inputs, as well as outputs, are of varying lengths.



So RNNs can be used for mapping inputs to outputs of varying types, lengths and are fairly generalized in their application. Looking at their applications, let’s see how the architecture of an RNN looks like.

**What are Recurrent Neural Networks?**

The idea behind RNNs is to make use of sequential information. In a traditional neural network, we assume that all inputs (and outputs) are independent of each other. But for many tasks, those are a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations. Another way to think about RNNs is that they have a “memory” that captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps (more on this later). Here is what a typical RNN looks like:



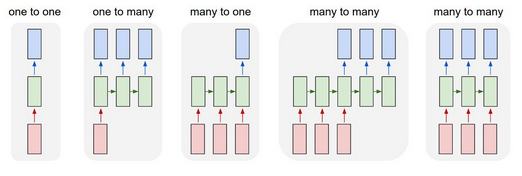
The above diagram shows an RNN being unrolled (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in an RNN are as follows:

* x\_t is the input at time step t. For example, x\_1 could be a one-hot vector corresponding to the second word of a sentence.
* s\_t is the hidden state at time step t. It’s the “memory” of the network. s\_t is calculated based on the previous hidden state and the input at the current step: s\_t=f(Ux\_t + Ws\_{t-1}). The function f usually is nonlinearity such as tanh or ReLU. s\_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
* o\_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. o\_t = \mathrm{softmax}(Vs\_t).

There are a few things to note here:

* You can think of the hidden state s\_t as the memory of the network. s\_t captures information about what happened in all the previous time steps. The output at step o\_t is calculated solely based on the memory at time t. As briefly mentioned above, it’s a bit more complicated in practice because s\_t typically can’t capture information from too many time steps ago.
* Unlike a traditional deep neural network, which uses different parameters at each layer, an RNN shares the same parameters (U, V, W above) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.
* The above diagram has outputs at each time step, but depending on the task this may not be necessary. For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word. Similarly, we may not need inputs at each time step. The main feature of an RNN is its hidden state, which captures some information about a sequence.

**Different types of RNN**



**One-to-one:**

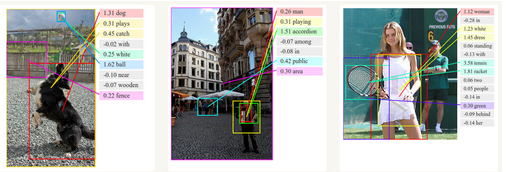
This also called Plain/Vaniall Neural networks. It deals with a fixed size of the input to the fixed size of Output where they are independent of previous information/output.

Ex: Image classification.

**One-to-Many:**

It deals with a fixed size of information as input that gives the sequence of data as output.

Ex: Image Captioning takes an image as input and outputs a sentence of words.



**Many-to-One:**

It takes a Sequence of information as input and outputs a fixed size of the output.

Ex: Sentiment analysis where a given sentence is classified as expressing positive or negative sentiment.

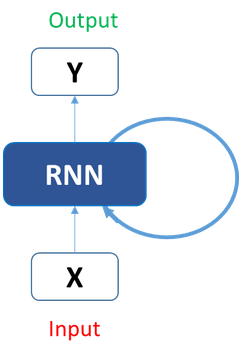
**Many-to-Many:**

It takes a Sequence of information as input and processes it recurrently outputs a Sequence of data.

Ex: Machine Translation, where an RNN reads a sentence in English and then outputs a sentence in French.

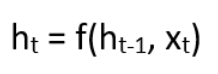
**Understanding a Recurrent Neuron in Detail**

Let’s take a simple task at first. Let’s take a character level RNN where we have the word “Hello”. So we provide the first 4 letters i.e. h,e,l,l, and ask the network to predict the last letter i.e. ’o’. So here the vocabulary of the task is just 4 letters {h,e,l,o}. In real case scenarios involving natural language processing, the vocabularies include the words in the entire Wikipedia database or all the words in a language. Here for simplicity, we have taken a very small set of vocabulary.



Let’s see how the above structure be used to predict the fifth letter in the word “hello”. In the above structure, the blue RNN block applies something called a recurrence formula to the input vector and also its previous state. In this case, the letter “h” has nothing preceding it, let’s take the letter “e”. So at the time the letter “e” is supplied to the network, a recurrence formula is applied to the letter “e” and the previous state which is the letter “h”. These are known as various time steps of the input. So if at time t, the input is “e”, at time t-1, the input was “h”. The recurrence formula is applied to e and h. and we get a new state.

The formula for the current state can be written as –



Here, Ht is the new state, ht-1 is the previous state while xt is the current input. We now have a state of the previous input instead of the input itself, because the input neuron would have applied the transformations on our previous input. So each successive input is called as a time step.

In this case, we have four inputs to be given to the network, during a recurrence formula, the same function and the same weights are applied to the network at each time step.

Taking the simplest form of a recurrent neural network, let’s say that the activation function is tanh, the weight at the recurrent neuron is Whh and the weight at the input neuron is Wxh, we can write the equation for the state at time t as –

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06005300/eq2.png

The Recurrent neuron in this case is just taking the immediately previous state into consideration. For longer sequences, the equation can involve multiple such states. Once the final state is calculated we can go on to produce the output

Now, once the current state is calculated we can calculate the output state as-

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06005750/outeq.png

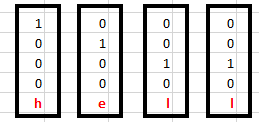
Let me summarize the steps in a recurrent neuron for you-

1. A single time step of the input is supplied to the network i.e. xt is supplied to the network
2. We then calculate its current state using a combination of the current input and the previous state i.e. we calculate ht
3. The current ht becomes ht-1 for the next time step
4. We can go as many time steps as the problem demands and combine the information from all the previous states
5. Once all the time steps are completed the final current state is used to calculate the output yt
6. The output is then compared to the actual output and the error is generated
7. The error is then backpropagated to the network to update the weights(we shall go into the details of backpropagation in further sections) and the network is trained

Let’s take a look at how we can calculate these states in Excel and get the output.

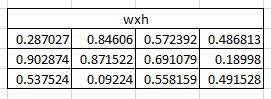
**Forward Propagation in a Recurrent Neuron in Excel**

Let’s take a look at the inputs first –



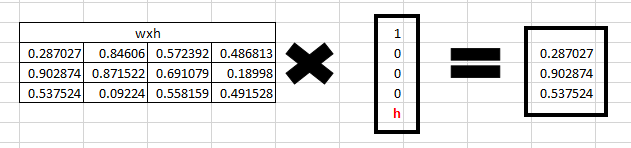
The inputs are one-hot encoded. Our entire vocabulary is {h,e,l,o} and hence we can easily one-hot encode the inputs.

Now the input neuron would transform the input to the hidden state using the weight wxh. We have randomly initialized the weights as a 3\*4 matrix –



**Step 1:**

Now for the letter “h”, for the hidden state, we would need Wxh\*Xt. By matrix multiplication, we get it as –



**Step 2:**

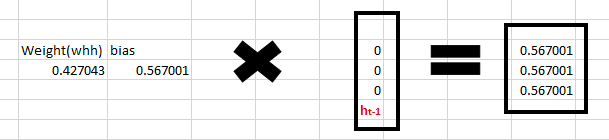
Now moving to the recurrent neuron, we have Whh as the weight which is a 1\*1 matrix as https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06013320/WHH.png

and the bias which is also a 1\*1 matrix as

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06013447/bias.png

For the letter “h”, the previous state is [0,0,0] since there is no letter prior to it.

So to calculate ->  (whh\*ht-1+bias)

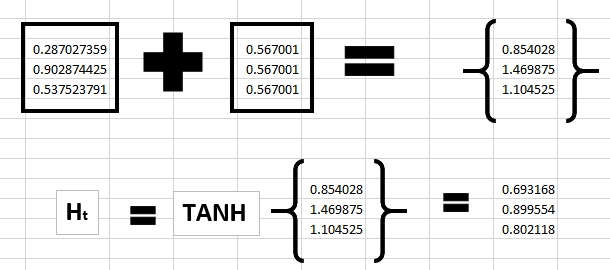


**Step 3:**

Now we can get the current state as –

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06014059/eq21.png

Since for h, there is no previous hidden state we apply the tanh function to this output and get the current state –

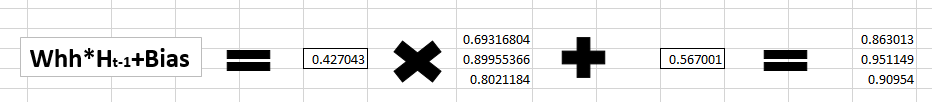


**Step 4:**

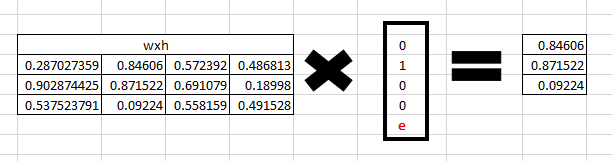
Now we go on to the next state. “e” is now supplied to the network. The processed output of ht, now becomes ht-1, while the one-hot encoded e, is xt. Let’s now calculate the current state ht.

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06005300/eq2.png

Whh\*ht-1 +bias will be –

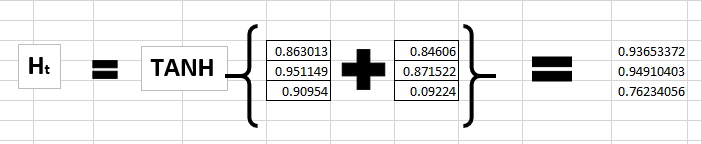


Wxh\*xt will be –



**Step 5:**

Now calculating ht for the letter “e”,

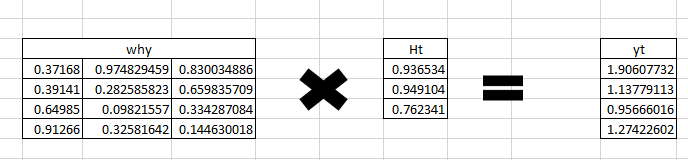


Now this would become ht-1 for the next state and the recurrent neuron would use this along with the new character to predict the next one.

**Step 6:**

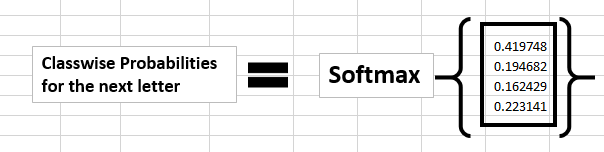
At each state, the recurrent neural network would produce the output as well. Let’s calculate yt for the letter e.

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/12/06005750/outeq.png



**Step 7:**

The probability for a particular letter from the vocabulary can be calculated by applying the softmax function. so we shall have softmax(yt)



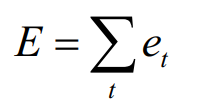
If we convert these probabilities to understand the prediction, we see that the model says that the letter after “e” should be h, since the highest probability is for the letter “h”. Does this mean we have done something wrong? No, so here we have hardly trained the network. We have just shown it in two letters. So it pretty much hasn’t learned anything yet.

Now the next BIG question that faces us is how does Backpropagation work in the case of a Recurrent Neural Network. How are the weights updated while there is a feedback loop?

**Backpropagation through TIME of RNN**

Earlier we introduced the forward propagation method of RNN, so how are the weigh parameters W, U, and V of the RNN updated?

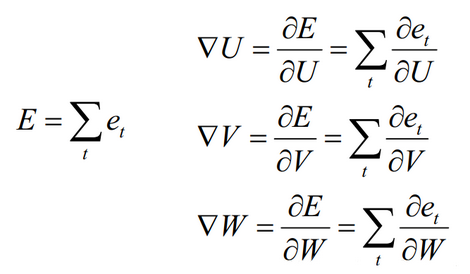
Each output will have a value Ot error value, Et the total error may be expressed as:



The loss function can use either the cross-entropy loss function or the squared error loss function.

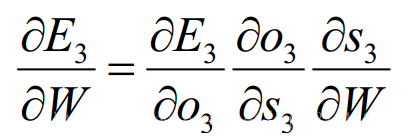
Because the output of each step does not only depend on the network of the current step but also the state of the previous steps, then this modified BP algorithm is called Backpropagation Through Time ( BPTT ), which is the reverse transfer of the error value at the output end. The gradient descent method is updated.

That is, the gradient of the parameter is required:

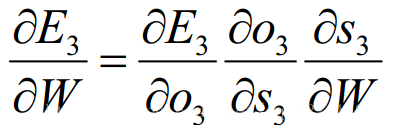


First, we solve the update method of W. From the previous update of W, it can be seen that it is the sum of the partial derivatives of the deviations at each moment.

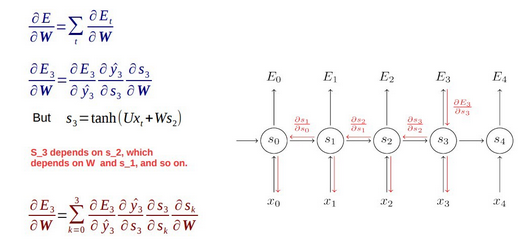
Here we take time t = 3 as an example. According to the chain derivation rule, we can get the partial derivative at time t = 3 as:



At this time, according to the formula,



We will find that in addition to W, S3 is also related to S2 at the previous moment.



For S3, expand directly to get the following formula:

https://lh4.googleusercontent.com/maNqyCyBu-wbjB6jj8YzTqasrvThimijYbkvtPn4N2ojA7tGPebPruNyELuorv9UmerQyiQYAGOxEMQR1weiY7ffcmN3hDxxN4SCf0_pjpg24041-f7VQwDOggJVo0dEN7JEIb4

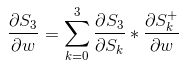
For S2, expand directly to get the following formula:

https://lh3.googleusercontent.com/VZlPWei9Omfhc1STU5HSSEANmbA4YMbUrWzNT6NlgH0UzNcyimxgHcNyupWJmrDLg3U9NJNBKI_V3DvOj3O511eY5IvLhDcFS68BUVRZAZl9x14Zw0PvyCS8p61H1gShF9VCOWk

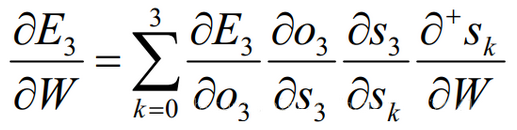
For S1, expand directly to get the following formula:

https://lh4.googleusercontent.com/km9-FRaj98ejMTBgxpBRbey8cOTyKQPEWKvU6vsN3P6tz0XthI23Z_ytfpto79P-f_2VlQ-DbbV-A9NKC4zXdyHwKrmlWTh9x0kY1utK261maP7nQTpwi9wZF26g0IfeKeu7Du0

Combine the above three formulas to get:



This gives the formula:

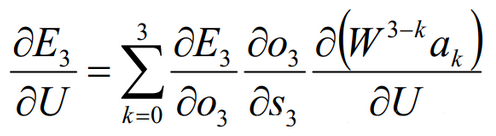


What is to be explained here is

https://lh6.googleusercontent.com/ygewDBjptPLbfN_OPca5IBP_bvzMOx05HVz1ctMv0YS1VL9bvn3Dj7fRGFOVcCKVh9yRzzliX7JPU1yV8zzSRxaLEVsY5FhVlbLDXu81LnNOcankCZaIUb0TKW5ch9DJWZT4oeE

that it means that S3 directly differentiates W without considering the effect of S2 (that is, for example, y = f (x) \* g (x) is the same as the derivative of x)

The second is the update method for U. Since the parameter U and W are similar, they will not be described here, and the specific formula finally obtained is as follows:



Finally, give the updated formula of V (V is only related to output O):

https://lh4.googleusercontent.com/BYKp-pQansPx8fjjBAkzWnAx5UdVCbo-AxvHsMHOt6C65ryUiOx4Ce-ahuL_R4U44QMPdro42V5X_afsv_PeqqAPxwjH5kdDcPN79Wv99BQvVDwbgpXi5WptkAKkenTqwJyg9Ns

**Going little deeper**

Let’s focus on one error term et.

You’ve calculated the cost function et, and now you want to propagate your cost function back through the network because you need to update the weights.

Essentially, every single neuron that participated in the calculation of the output, associated with this cost function, should have its weight updated in order to minimize that error. And the thing with RNNs is that it’s not just the neurons directly below this output layer that contributed but all of the neurons far back in time. So, you have to propagate all the way back through time to these neurons.

The problem relates to updating wrec (weight recurring) – the weight that is used to connect the hidden layers to themselves in the unrolled temporal loop.

For instance, to get from xt-3 to xt-2 we multiply xt-3 by wrec. Then, to get from xt-2 to xt-1 we again multiply xt-2 by wrec. So, we multiply with the same exact weight multiple times, and this is where the problem arises: when you multiply something by a small number, your value decreases very quickly.

As we know, weights are assigned at the start of the neural network with the random values, which are close to zero, and from there the network trains them up. But, when you start with wrec close to zero and multiply xt, xt-1, xt-2, xt-3, … by this value, your gradient becomes less and less with each multiplication.

**What does this mean for the network?**

The lower the gradient is, the harder it is for the network to update the weights and the longer it takes to get to the final result.

For instance, 1000 epochs might be enough to get the final weight for the time point t, but insufficient for training the weights for the time point t-3 due to a very low gradient at this point. However, the problem is not only that half of the network is not trained properly.

The output of the earlier layers is used as the input for the further layers. Thus, the training for the time point t is happening all along based on inputs that are coming from untrained layers. So, because of the vanishing gradient, the whole network is not being trained properly.

To sum up, if wrec is small, you have a vanishing gradient problem, and if wrec is large, you have exploding gradient problem.

For the vanishing gradient problem, the further you go through the network, the lower your gradient is and the harder it is to train the weights, which has a domino effect on all of the further weights throughout the network.

That was the main roadblock to using Recurrent Neural Networks. But let’s now check what the possible solutions to this problem are.

Solutions to the vanishing gradient problem In case of exploding gradient, you can:

* stop backpropagating after a certain point, which is usually not optimal because not all of the weights get updated;
* penalize or artificially reduce gradient;
* put a maximum limit on a gradient.

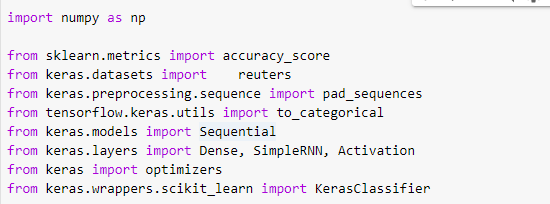
In case of vanishing gradient, you can:

* initialize weights so that the potential for vanishing gradient is minimized;
* have Echo State Networks that are designed to solve the vanishing gradient problem;
* have Long Short-Term Memory Networks (LSTMs).

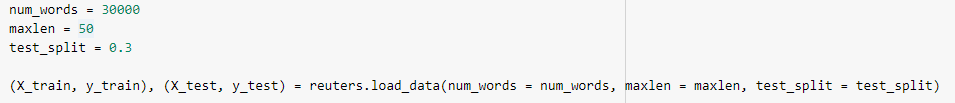
**Building Classifiers using the Reuters Dataset**

**Simple RNN**

Importing the necessary packages



Setting parameters for data load

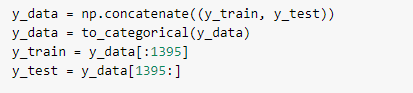


We are padding the data

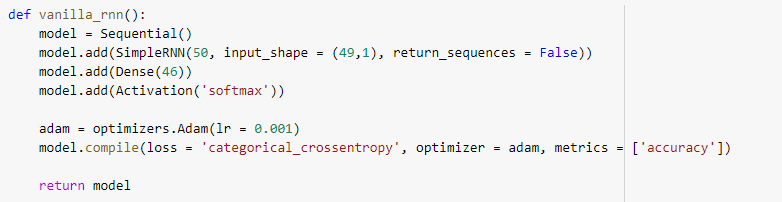
https://lh5.googleusercontent.com/fcr6esyZkTTXFIssw5U7d6y7xjiMEEYzwI2a6_cqm-yUwHX6ZTvHBW6MQa1XNsRuIBUOJ3GN0E2yQ1q2EQzfaVcU1iRyI41EJzo3UoCaOBZ_Uojr58QU7bkX1jcy9fDyK9iYU6U

Converting the train and test files to a numpy array

https://lh3.googleusercontent.com/NrCkM3uZP1GY0L9gKmuI7NSwQQ40ppfuKbmjU_WwRK4u2F54eJwcmvnmVWri8NTOoKLD2CAkB7e4pobut6LHMjSiDi5m5qB4Aa2wcU4EYBOWeolOE8t_YjrkY6Miti8OSNlUtaY



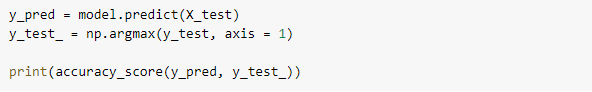
Creating a function to fit an RNN model with a simple RNN layer



Fitting the model using Keras classifier

https://lh3.googleusercontent.com/SQtuI9WgtbZlpvSNQ3uSdKrRFomOUobxlGfQRE75HPiYJSeQZPl3fXXsDZVHSbQnbqipg_faS-OOzxCNqST8sy-ZhugPzVNQtwesazfH485lxdozpdUXJMHZ_dCppE0ZKiZoO1s

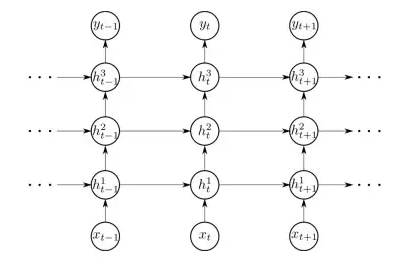
Making the predictions on the test data



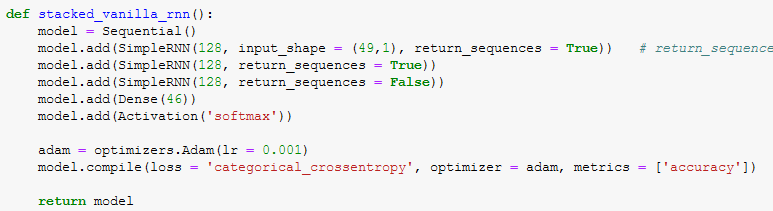
https://lh3.googleusercontent.com/I2cn_5YY7EK3Nzp9XzCjvNFEFtbaRtqMAXc00F9kCXCqpJH3Ks26R4Q0yXBL26LpPl_NburUL-SvAOQkVj2OPvjHH80eYwEK2g-DSUYHGVk7SMF1A6mXmZ6A71kYeEULYu5KECE

We got an accuracy of 75% with a simple RNN layer.

**Stacked RNN**



We will be creating a RNN model with multiple stacked RNN layers to check the performance of increased layers.

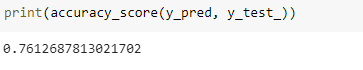


Let fit the model on the train set.

https://lh5.googleusercontent.com/f-EHUi375N9w_lU54WiW_bgguiZIaTpF1nx56LdA23lEZHkBL1AzMZ5Jm1fpl-HRO5uMeqgWQXQCZ3UZ-O3Y03ZLS4d4HezjD2yGNrFTLmwW3T2O__S6L1N-Rq4tKYEqSQ6dSEI

Test the model performance on the test set.

https://lh5.googleusercontent.com/nDe4kitMis9xYv0Ixor_YfXhj-0YYGxSW8prUYo9iJeolthH22tJsrhwjwhtcZPbUchIwGXUs0spA9OKfZrEi2FutJsnZg3_RYreMZ6tPR4qEyyK-UjNrC71IVwNdZm_yuE5izE



We can see the accuracy sore of the model has just increased by 1%. The stacked layers haven’t increased our model performance significantly.

This is because the more the deep of the RNN increases; there is a high chance of facing a Vanishing and Exploding gradient in RNN networks.

We overcome this problem the researcher has come up with an extraordinary algorithm called Long Short Term Memory (LSTM).

The use of RNN in real-time is very minute but the use of LSTM is very extensive in many real-time use cases.

**Text Analytics**

We will use NLP for text analytics. There are many libraries available for NLP in python. We will focus on the two most important ones:

* Natural Language Toolkit (NLTK)
* Spacy

Before we dive into text analytics using NLTK or Spacy, now when we process text there are a few terminologies that we need to understand.

**Tokenization**

Tokenization is the process of breaking down a given paragraph of text into a list of sentences or words. When a paragraph is broken down into a list of sentences, it is called sentence tokenization. Similarly, if the sentences are further broken down into a list of words, it is known as Word tokenization.

**Example:**

A picture containing text, whiteboard

Description automatically generated

Below is a given paragraph, let’s see how tokenization works on it:

A screenshot of a computer

Description automatically generated with medium confidence

**Sentence Tokenization**

Text

Description automatically generated

**Word Tokenization**

A screenshot of a computer

Description automatically generated with low confidence

**Stemming and Lemmatization**

Many words that are used in a sentence are not always used in their base form but are used as per the rules of grammar.

Example:

* Running 🡪 Run (base word)
* Runs 🡪 Run (base word)
* Ran 🡪 Run (base word)

Although the underlying meaning is the same, the form of the base word changes to preserve the correct grammatical meaning.

Stemming and lemmatization are basically used to bring such words to their basic form so that the words with the same base are treated as the same words rather than treated differently.

The only difference in Stemming and Lemmatization is the way in which they change the word's base form.

* **Stemming**

Stemming means mapping a group of words to the same stem by removing prefixes or suffixes without giving any value to the “grammatical meaning” of the stem formed after the process.

**Example:**

Computation 🡪 Comput

Computer 🡪 Comput

Hobbies 🡪 hobbi

We can see that steaming tries to bring the word back to its base word but the base word may or may not have the correct grammatical meaning.

* **Lemmatization**

Lemmatization also does the same thing as stemming and tries to bring a word to its base form, but unlike stemming it does keep in account the actual meaning of the base word i.e. the base word belongs to any specific language. The “base word” is known as “Lemma”.

**Example:**

Finally 🡪 Final

Final 🡪 Final

Finalized 🡪 Final

**Stop Words**

Stop words are words that are very common in occurrence such as ‘a’, ‘an’, ‘the’, ‘at’ etc. We ignore such words during the preprocessing part since they do not give any important information and would just take additional space. We can make our custom list of stop words as well if we want.

A picture containing text

Description automatically generated

**Parts Of Speech Tagging (POS Tagging) and Chunking**

As the name suggests, it is a method of tagging individual words on the basis of their parts of speech.

Part-of-speech tagging also called grammatical tagging or word-category disambiguation is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context i.e., its relationship with adjacent and related words in a phrase, sentence or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

There are 9 parts of speech in grammars, but in NLP there are more than 9 POS tags based on a different set of rules, such as:

Table

Description automatically generated Table

Description automatically generated

**Example:**

Diagram

Description automatically generated

Chunking can be used to make data more structured by giving a specific set of rules. Chunking is also known as a shallow parser.

Timeline

Description automatically generated

**Named Entity Recognition (NER)**

In chucking, we read that we can set rules to keep different POS tags under one single user-defined tag. One such form of chunking in NLP is known as Named Entity Recognition.

Diagram

Description automatically generated

**Example:**

Diagram

Description automatically generated with low confidence

**N-gram** is the sequence of n words from a given text/document.

When,

* + n = 1, we call it a “unigram”
  + n = 2, it is called a “bigram”
  + n = 3, it is called a “trigram”.

Let’s understand this with an example:

Text1 = “I went to have a cup of coffee but I ended up having lunch with her.”

**Unigram**

[I, went, to, have, a, cup, of, coffee, but, I, ended, up, having, lunch, with, her]

**Bigram**

[I went], [went to],[to have],[have a],[a cup],[cup f],[of coffee],[coffee but],[but I],[I ended],[ended up], [up having],[having lunch],[lunch with],[with her]

**Tri-gram**

[I went to], [went to have], [to have a], [have a cup],[ a cup of], [cup of coffee],[ of coffee but],[ coffee but I],[but I ended],[I ended up],[ended up having],[up having lunch],[having lunch with],[lunch with her].

**Note:** We can clearly see that BOW model is nothing but n-gram model when n = 1.

**Skip-grams**

Skip grams are a type of n-grams where the words are not necessarily in the same order as are in the given text. i.e., some words can be skipped.

**Example:**

Text2 = “I don’t understand, what is the problem here?”

1-skip 2-grams (we have to make 2-gram while skipping 1 word)

[I understand, don’t what, understand is, what the, is the problem, here].

**Sentiment Analysis**

**Case Study**

This project is based on the data set which is composed of about 20k tweet's to train sentiment predictor

**Importing all the necessary libraries**

Text

Description automatically generated

**Importing Data**

A picture containing text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

**Basic Exploratory Data Analysis**

Graphical user interface, text, application

Description automatically generated

Graphical user interface

Description automatically generated

There are three different classes of target variables.

Table

Description automatically generated

Let’s check the null values

Graphical user interface, application, Word

Description automatically generated

There are one missing value, lets impute the missing value



**Data Cleaning**

Text, timeline

Description automatically generated with medium confidence

Text

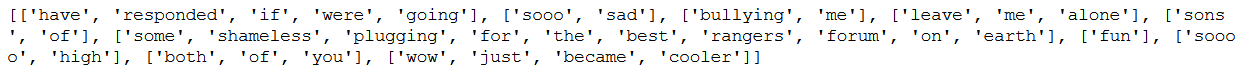
Description automatically generated

We removed all the URLs, Emails, newline characters, and single quotes using the above function.

Let’s convert the sentence into words using the below function.

Graphical user interface, text, application, email

Description automatically generated



Now, let’s do detokenization

A picture containing shape

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

**Data labeling**

Graphical user interface, text, application, email

Description automatically generated

Text

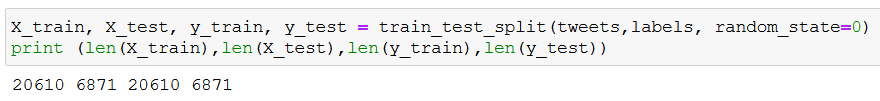
Description automatically generated

We have done tokenization and padding in the above code.

Table

Description automatically generated with medium confidence

**Train and Test Split**



**Model building**

Text

Description automatically generated

A picture containing calendar

Description automatically generated

I have built the model using only 5 epochs and the loss has come down to 42% from 74%. You can try to play with epochs and validate the model performance.

Let’s load the model with what we saved and predict on the test set.

A picture containing text

Description automatically generated

Text

Description automatically generated

**Confusion matrix**

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

Chart, treemap chart

Description automatically generated

Most of the tweets are correctly predicted. We can improve our model with an increase in epochs.

**SUMMARY**

We started with understanding feature selection techniques, gradient descent algorithms then we moved on to the neural networks where we studied perceptron, weights initialization techniques, loss functions, optimization techniques, and how weights get updated in backpropagation. Finally, we studied how neural networks work in NLP, what is RNN, and different types of RNN.

**Program Assignment**

Build a sentiment analysis engine to review and predict sentiment using the Anti-Money Laundering.

Follow the below step by step approach:

* Data importing
* Data Exploring
* Data Cleaning
* Model building
* Hyperparameter tuning
* Model evaluation

**Assessment**

**Choose the appropriate option**

1. **Which of the following statements correctly represents a real neuron?**
2. A neuron has a single input and a single output only
3. A neuron has multiple inputs but a single output only
4. A neuron has a single input but multiple outputs
5. A neuron has multiple inputs and multiple outputs
6. All of the above statements are valid
7. **In a neural network, knowing the weight and bias of each neuron is the most important step. If you can somehow get the correct value of weight and bias of each neuron, you can approximate any function. What would be the best way to approach this?**
8. Assign random values and pray to God they are correct.
9. Search every possible combination of weights and biases till you get the best value.
10. Iteratively check that after assigning a value how far you are from the best values, and slightly change the assigned values to make them better.
11. None of these
12. **Which is the sequence of the following tasks in a perceptron?**
13. Initialize weights of perceptron randomly
14. Go to the next batch of the dataset
15. If the prediction does not match the output, change the weights
16. For a sample input, compute an output
17. 1 2, 3, 4
18. 4, 3, 2, 1
19. 3, 1, 2, 4
20. 1, 4, 3, 2
21. **Which of the following gives non-linearity to a neural network?**
22. Stochastic Gradient Descent
23. Rectified Linear Unit
24. Convolution Function
25. None of the above
26. **In training a neural network, the loss does not decrease in the few starting epochs.**

Chart, shape

Description automatically generated with medium confidence

**The reasons for this could be:**

1. **The learning rate is low**
2. **Regularization parameter is high**
3. **Stuck at local minima**

**What according to you are the probable reasons?**

1. 1 and 2
2. 2 and 3
3. 1 and 3
4. Any of these

**Fill in the spaces with appropriate answers**

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Gradient technique is more advantageous when the data is too big to handle in RAM simultaneously?
2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, propagates the error information from the end of the network to all the weights inside the network.
3. Gradient descent algorithm aims to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ the cost function.
4. If the activation function is ReLU and the input value is negative, what would be the output of this neuron?
5. Which activation function help you to overcome the dying ReLU problem?

**True or False**

1. Can a neural network model the function (y=1/x)?
   1. True
   2. False
2. The number of neurons in the output layer should match the number of classes (where the number of classes is greater than 2) in a supervised learning task.
   1. True
   2. False
3. Y = ax^2 + bx + c (polynomial equation of degree 2)

Can this equation be represented by a neural network of the single hidden layer with a liner threshold?

* 1. True
  2. False

1. RNN cannot be used for sequential data.
   1. True
   2. False
2. The large learning rate in the deep neural networks helps to reach global minima soon
   1. True
   2. False

**Solutions for Assessment**

**Choose the appropriate options answers**

1. E
2. C
3. D
4. B
5. D

**Fill in the spaces with appropriate answers**

1. Stochastic Gradient Descent
2. Backpropagation
3. Minimize
4. Zero
5. Leaky ReLU/Parameterized ReLU

**True or False**

1. True
2. False
3. False
4. False
5. False