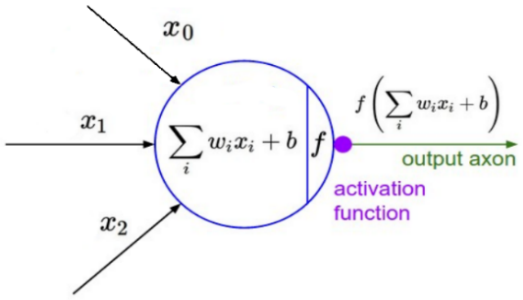
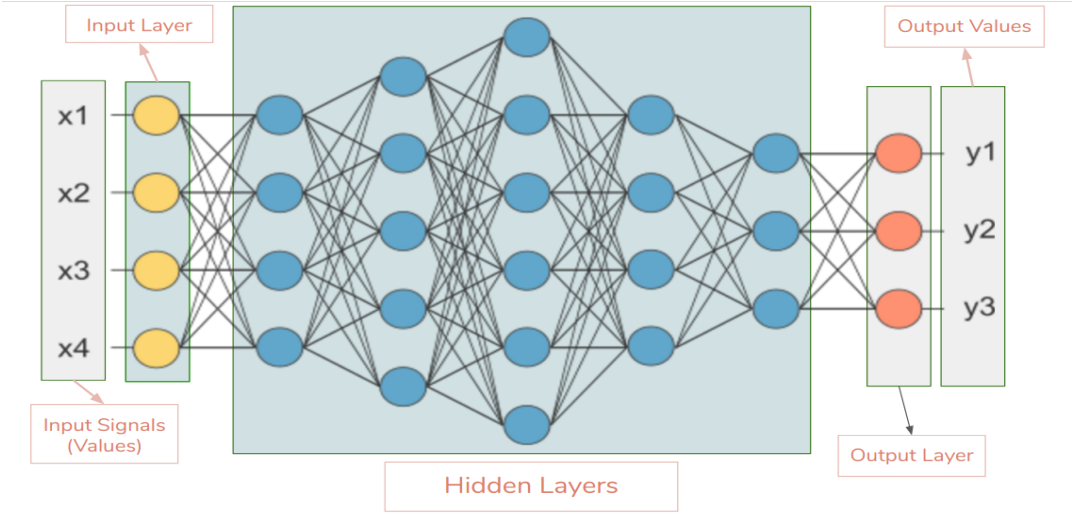
NEURAL NETWORKS

A neural network is a machine learning algorithm based on the model of a human neuron. The human brain consists of millions of neurons. It sends and process signals in the form of electrical and chemical signals. These neurons are connected with a special structure known as synapses. Synapses allow neurons to pass signals. From large numbers of simulated neurons forms the neural networks

Neural networks are typically organized in layers. Layers are being made up of many interconnected nodes which contain an activation function. A neural network is composed of input, hidden, and output layers - all of which are composed of nodes. Input layers take in a numerical representation of data (e.g. images with pixel specs), output layers output predictions, while hidden layers are correlated with most of the computation.



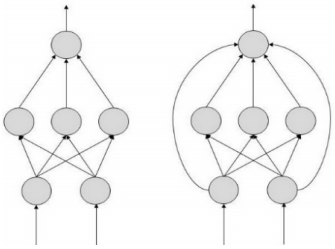


Types of Neural Networks

Feed forward Neural Networks

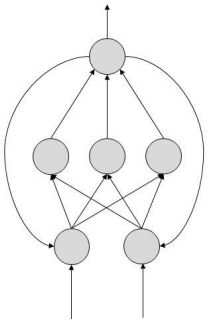
ANN and CNN are Feedforward Neural Networks. There are no feedback connections in which outputs of the model are fed back into itself. In this network flow of information is unidirectional. Also, no feedback loops are present in this.

ANN and CNN allow signals to travel one way only i.e. from input to output. Feed-forward ANNs extensively used in pattern recognition. Feedforward neural networks are ideally suitable for modeling relationships between a set of predictor or input variables and one or more response or output variables. In other words, they are appropriate for any functional mapping problem where we want to know how a number of input variables affect the output variable. The multilayer feedforward neural networks, also called multi-layer perceptron (MLP) are the most widely studied and used neural network model in practice.



Feed backward Neural Networks

Feedback (or recurrent or interactive) networks can have signals traveling in both directions by introducing loops in the network. Feedback networks are powerful and can get extremely complicated. Computations derived from earlier input are fed back into the network, which gives them a kind of memory.



Important Terms

Input layer- This is the first layer in the neural network. It takes input signals(values) and passes them on to the next layer. It doesn't apply any operations on the input signals(values) & has no weights and biases values associated. Brings the initial data into the system for further processing by subsequent layers of artificial neurons. Usually, the number of input nodes in an input layer is equal to the number of explanatory variables. 'input layer' presents the patterns to the network, which communicates to one or more 'hidden layers'. The nodes of the input layer are passive, meaning they do not change the data.

Hidden layer-A layer in between input layers and output layers where artificial neurons take in a set of weighted inputs and produce an output through an activation function. Hidden layers have neurons(nodes) which apply different transformations to the input data. One hidden layer is a collection of neurons stacked vertically. The Hidden layers apply given transformations to the input values inside the network. In this, incoming arcs that go from other hidden nodes or from input nodes connected to each node. It connects with outgoing arcs to output nodes or to other hidden nodes. In hidden layer, the actual processing is done via a system of weighted 'connections'. The values entering a hidden node multiplied by weights, a set of predetermined numbers stored in the program. The weighted inputs are then added to produce a single number.

Output layer- Hidden layers then link to an output layer . With this layer we can get the desired number of values and in a desired range. The output layer receives connections from hidden layers. It returns an output value that corresponds to the prediction of the response variable. In classification problems, there is usually only one output node. The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units. This layer is the last layer in the network & receives input from the last hidden layer.

Bias(Offset)- It is an additional parameter in the Neural Network which is used to adjust the output along with the weighted sum of the inputs to the neuron. Therefore, Bias is a constant which helps the model in a way that it can fit best for the given data. Every neuron has its own bias term. Bias helps in controlling the value at which activation function will trigger. Bias is like the intercept added in a linear equation.

The processing done by a neuron is thus denoted as:

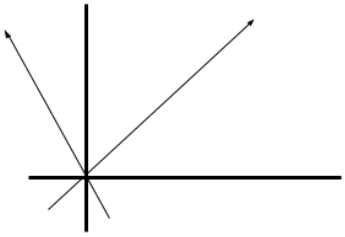
output = sum (weights * inputs) + bias

What is the need of bias?

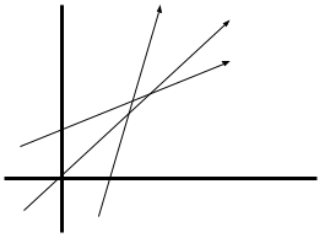
A simpler way to understand bias is through a constant c of a linear function

y =mx + c

It allows you to move the line down and up fitting the prediction with the data better. If the constant c is absent then the line will pass through the origin (0, 0) and you will get a poorer fit.



Y = mx



Y = mx + c

Due to absence of bias, model will train over point passing through origin only, which is not in accordance with real-world scenario. Also with the introduction of bias, the model will become more flexible

For Example: Suppose an activation function act () which get triggered on some input greater than 0.

input1 = 1, weight1 = 2, input2 = 2, weight2 = 2

output = input1*weight1 + input2*weight2

output = 6, suppose act(output) = 1

Now a bias is introduced in output as bias = -6, so the output become 0.

act (0) = 0. Thus activation function will not trigger.

Epoch - One forward pass and one backward pass of all the training examples.

Convergence - Convergence is when as the iterations proceed the output gets closer and closer to a specific value.

Fully Connected Layers- When all the nodes in the Lth layer connect to all the nodes in the (L+1)th layer we call these layers fully connected layers.

Model Optimizers- Optimization algorithms help us to minimize (or maximize) an objective function (another name for error function). Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction. We use various Optimization strategies and algorithms to update and calculate appropriate and optimum values of such model's parameters which influence our Model's learning process and the output of a Model. Some of the optimizers are SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam

Types of optimization algorithms

First Order Optimization Algorithms - These algorithms minimize or maximize a Loss Function E(x) using its Gradient values with respect to the parameters. Most widely used First order optimization algorithm is Gradient Descent.

Second Order Optimization Algorithms - Second-order methods use the second order derivative which is also called Hessian to minimize or maximize the Loss function. Since the second derivative is costly to compute, the second order is not used much

Why do we use partial derivative/first order derivative?

A Gradient is simply a vector and is represented by a matrix, which is simply a matrix consisting of first order partial Derivatives(Gradients). So to calculate a gradient we use partial derivative. The First order derivative tells us whether the function is decreasing or increasing at a particular point. First order Derivative basically give us a line which is Tangential to a point on its Error Surface

What's the difference between derivate and gradient?

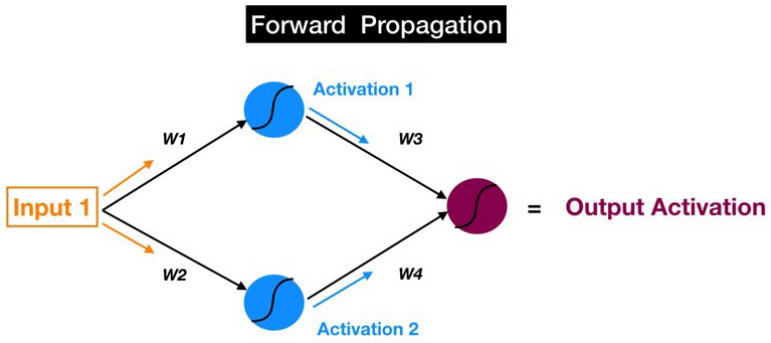
Hence summing up, a derivative is simply defined for a function dependent on single variables whereas a Gradient is defined for function dependent on multiple variables

Which Order Optimization Strategy to use

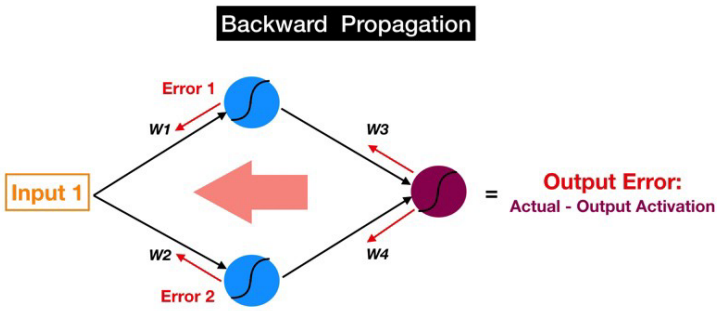
Now The First Order Optimization techniques are easy to compute and less time consuming, converging pretty fast on large data sets.

Second Order Techniques are faster only when the Second Order Derivative is known otherwise, these methods are always slower and costly to compute in terms of both time and memory

Forward Propagation - Forward propagation is a process of feeding input values to the neural network and getting an output which we call predicted value. Sometimes we refer forward propagation as inference. When we feed the input values to the neural network's first layer, it goes without any operations. Second layer takes values from first layer and applies multiplication, addition and activation operations and passes this value to the next layer. Same process repeats for subsequent layers and finally we get an output value from the last layer.



Backward Propagation - Backpropagation is the reverse. Except instead of signal, we are moving error backwards through our model.



The goal of my network is ultimately to minimize this loss by adjusting the weights and biases of the network. In using something called back propagation through gradient descent, the network backtracks through all its layers to update the weights and biases of every node in the opposite direction of the loss function.

Back-propagation is all about feeding loss backwards in such a way that we can fine-tune the weights based on which. In other words, every iteration of back propagation should result in a smaller loss function than before.

After forward propagation we get an output value which is the predicted value. To calculate error, we compare the predicted value with the actual output value. We use a loss function (mentioned below) to calculate the error value. Then we calculate the derivative of the error value with respect to each and every weight in the neural network. Back-Propagation uses chain rule of Differential Calculus. In chain rule first we calculate the derivatives of error value with respect to the weight values of the last layer. We call these derivatives, gradients and use these gradient values to calculate the gradients of the second last layer. We repeat this process until we get gradients for each and every weight in our neural network. Then we subtract this gradient value from the weight value to reduce the error value. In this way we move closer (descent) to the Local Minima (means minimum loss).

Gradient Descent - Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. It is an iterative optimisation algorithm to find the minimum of a function or find the coefficients for which the cost/loss function is minimum

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost). Parameters refer to coefficients in Linear Regression and weights in neural networks. Gradient Descent is a first-order optimization algorithm used to find local minima. This means it only takes into account the first derivative when performing the updates on the parameters.

Gradient Process is a method which comprises a vector of weights (or coefficients) where we calculate their partial derivative with respective to zero. The motive behind calculating their partial derivative is to find the local minima of the loss function (RSS), which is convex in nature. In this algorithm, we calculate the derivative of cost function in every iteration and update the values of parameters simultaneous

Activation function - Activation functions are used to introduce non-linearity to neural networks.

Their main purpose is to convert an input signal of a node in a ANN to an output signal. That output signal now is used as an input in the next layer in the stack.

It squashes the values in a smaller range.

The activation function serves two notable purposes:

-It captures non-linear relationship between the inputs

-It helps convert the input into a more useful output.

An activation function is a mathematical equation that determines the output of each element (perceptron or neuron) in the neural network. It takes in the input from each neuron and transforms it into an output, usually between one and zero or between -1 and one.

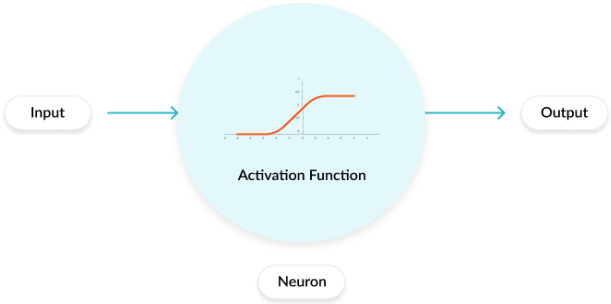
Why do we use activation functions?

Why don't we just use the initial dot product output without feeding it to the activation function? Well, without an activation function we will fail to introduce non-linearity into the network.

An activation function will allow us to model a response variable (target variable, class label, or score) that varies non-linearly with its explanatory variables. Non-linear means that the output cannot be reproduced from a linear combination of the inputs. Another way to think of it: without a non-linear activation function in the network, an artificial neural network, no matter how many layers it has, will behave just like a single-layer perceptron, because summing these layers would give you just another linear function. Neural networks rely on nonlinear activation functions—the derivative of the activation function helps the network learn through the backpropagation process

The question arises that why can't we do it without activating the input signal

If we do not apply an activation function, then the output signal would simply be a simple linear function. A linear function is just a polynomial of one degree. Now, a linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data. A Neural Network without Activation function would simply be a Linear Regression Model, which has limited power and does not performs good most of the times. We want our Neural Network to not just learn and compute a linear function but something more complicated than that. Also without activation function our Neural network would not be able to learn and model other complicated kinds of data such as images, videos, audio, speech etc.



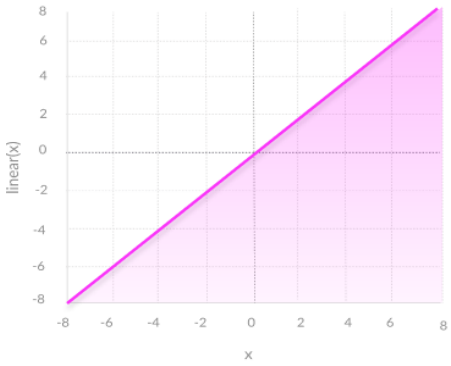
So why do we need Non-Linarites

Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function.

Hence it all comes down to this, we need to apply an activation function f(x) so as to make the network more powerful and add ability to it to learn something complex and complicated form data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a nonlinear activation, we are able to generate non-linear mappings from inputs to outputs. Just always remember to do: “Input times weights, add Bias and Activate”

Types of Activation functions

Linear or Identity Activation Function- It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input. In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no.



Limitations

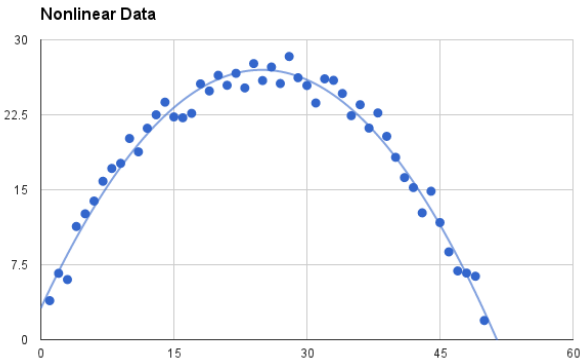
1. Not possible to use backpropagation (gradient descent) to train the model—the derivative of the function is a constant, and has no relation to the input, X. So it’s not possible to go back and understand which weights in the input neurons can provide a better prediction.

2. All layers of the neural network collapse into one—with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer (because a linear combination of linear functions is still a linear function). So a linear activation function turns the neural network into just one layer.

Equation: f(x) = x

Range: (-infinity to infinity)

Non-linear Activation Function - The Nonlinear Activation Functions are the most used activation functions. It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output. Nonlinearity helps to makes the graph look something like this



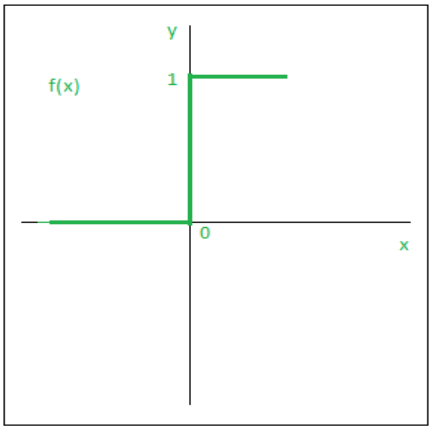
Examples of Activation Functions

Step Function

Step Function is one of the simplest kind of activation functions. In this, we consider a threshold value and if the value of net input say y is greater than the threshold then the neuron is activated.

F(x)=1, if x>0

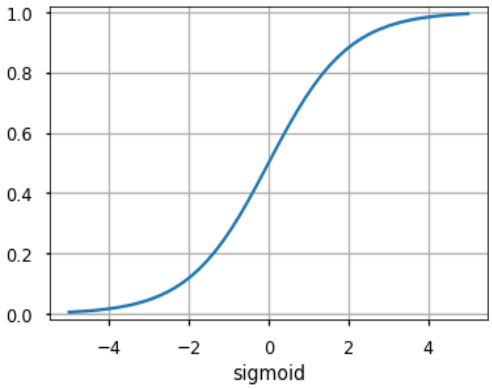
F(x)=0, if x<0



Sigmoid

A sigmoid function is a mathematical function having a characteristic “S”-shaped curve or sigmoid curve. Often, sigmoid function refers to the special case of the logistic function which generate a set of probability outputs between 0 and 1 when fed with a set of inputs. The sigmoid activation function is widely used in binary classification.

$$Sigmoid(x) = \frac{1}{1 + e^{(-x)}} = \frac{e^x}{1 + e^x}$$

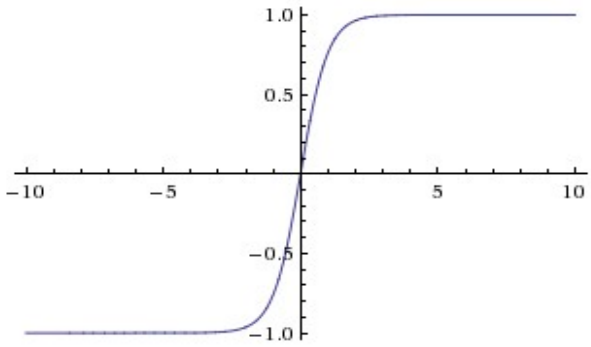


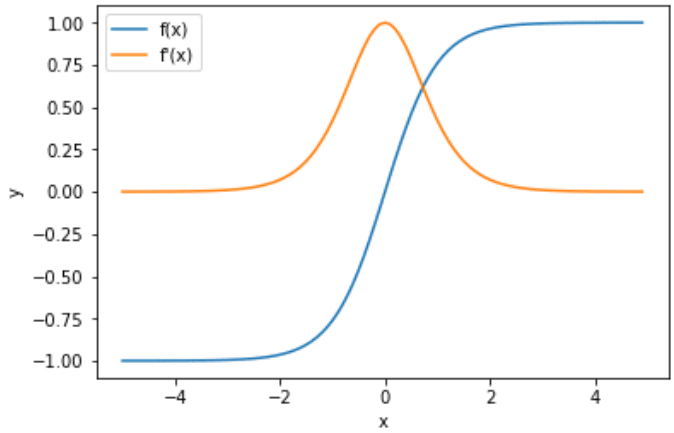
The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

Tan-h

An alternative to the logistic sigmoid is the hyperbolic tangent or tan-h function. Like the logistic sigmoid, the tan-h function is also sigmoidal (“S”-shaped), but instead outputs values that range [-1, 1]. Thus strongly negative inputs to the tan-h will map to negative outputs.

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2 \times Sigmoid(2x) - 1$$





The advantage of tanh over sigmoid is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

The tanh function is mainly used classification between two classes. Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

Softmax

Unlike the Sigmoid activation function, the Softmax activation function is used for multi-class classification. Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes.

$$P(y = j \mid z^{(i)}) = \phi_{softmax}(z^{(i)}) = \frac{e^{z^{(i)}}}{\sum_{j=0}^{k} e^{z_k^{(i)}}},$$

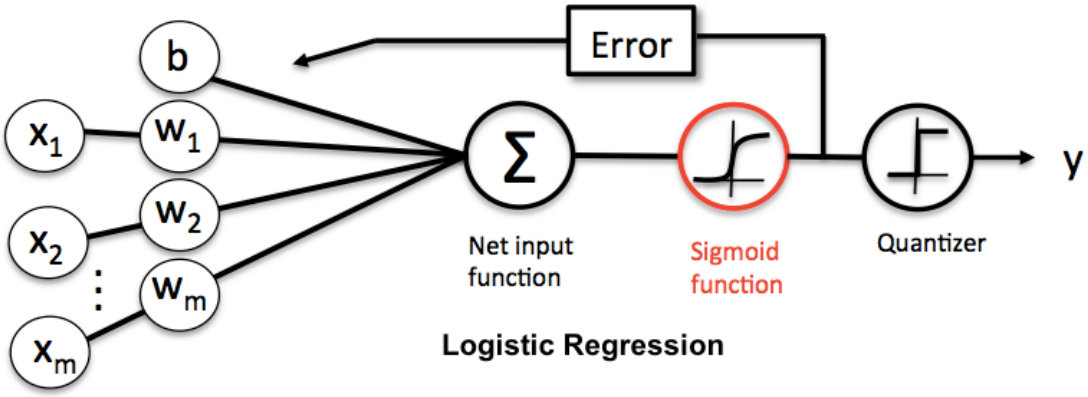
$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{l=0}^{m} w_l x_l = \mathbf{w}^T \mathbf{x}.$$

(w is the weight vector, x is the feature vector of 1 training sample, and w0 is the bias unit.)

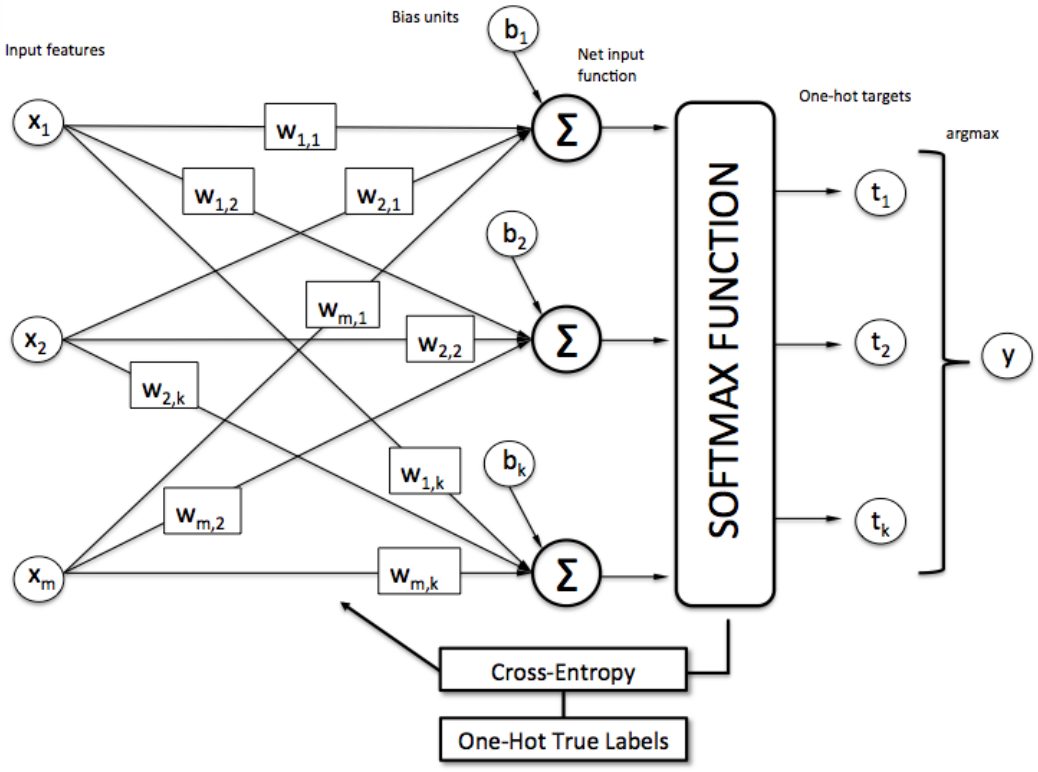
Now, this softmax function computes the probability that this training sample x(i) belongs to class j given the weight and net input z(i)

Example

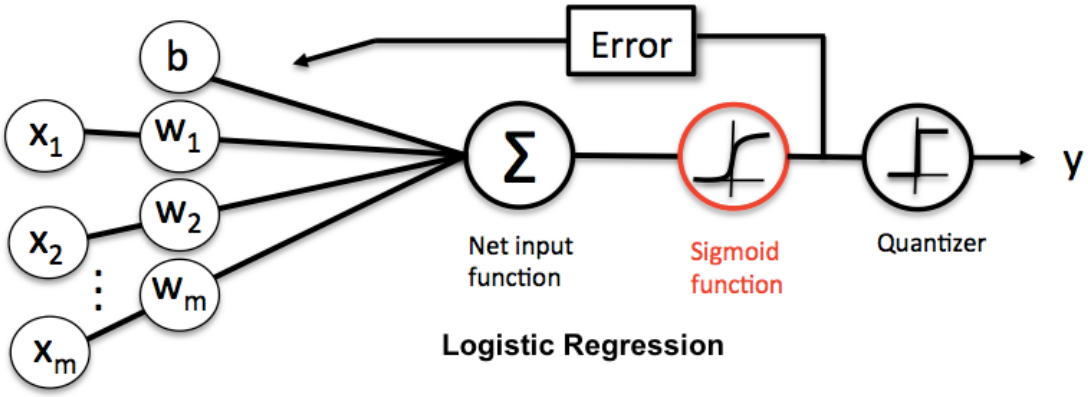
```
softmax:
 [[ 0.29450637  0.34216758  0.36332605]
 [ 0.21290077  0.32728332  0.45981591]
 [ 0.42860913  0.33380113  0.23758974]
 [ 0.44941979  0.32962558  0.22095463]]
```

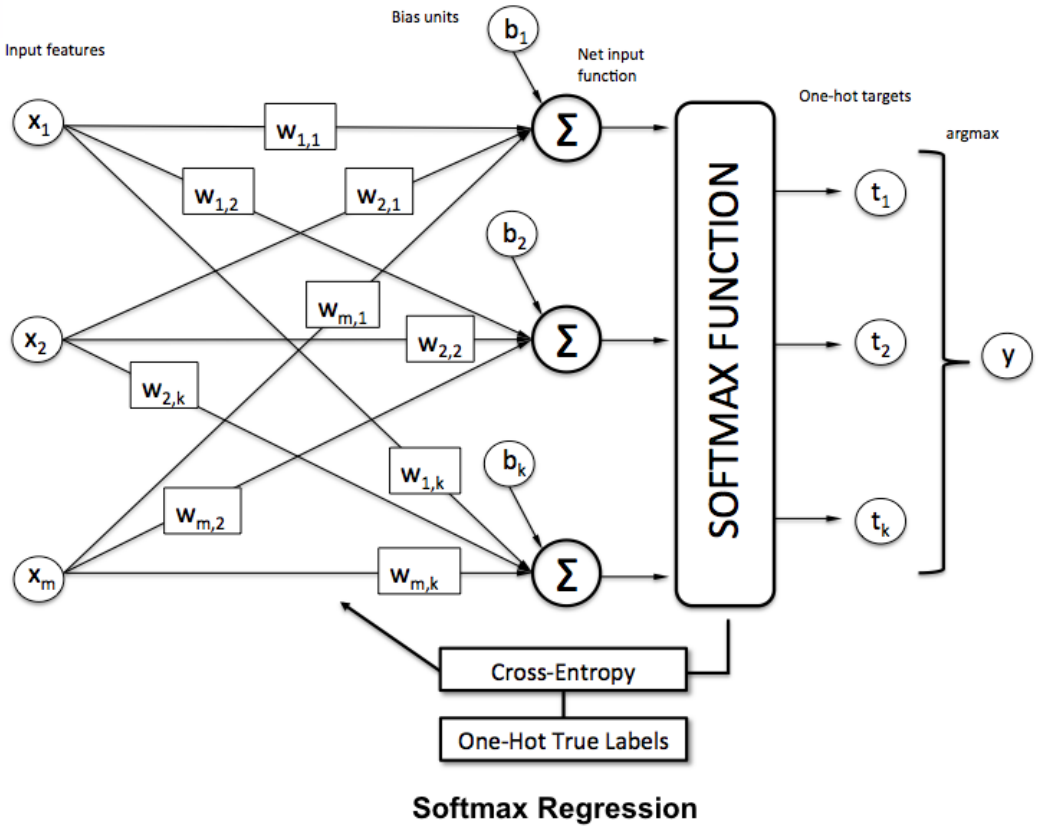


Logistic Regression



Softmax Regression





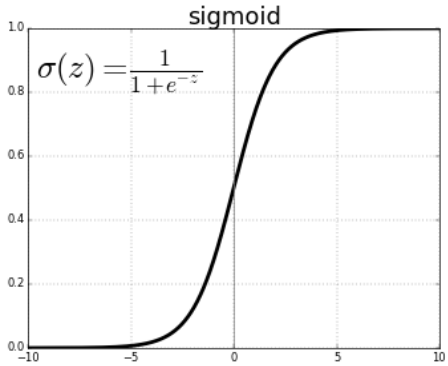
ReLU

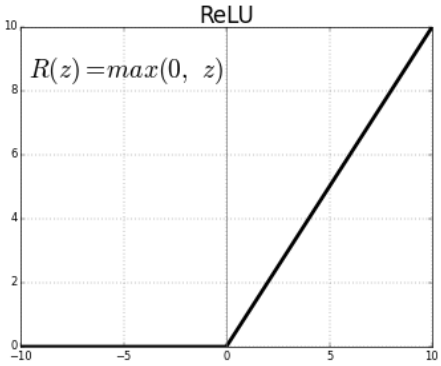
Instead of the sigmoid activation function, most recent artificial neural networks use rectified linear units (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to

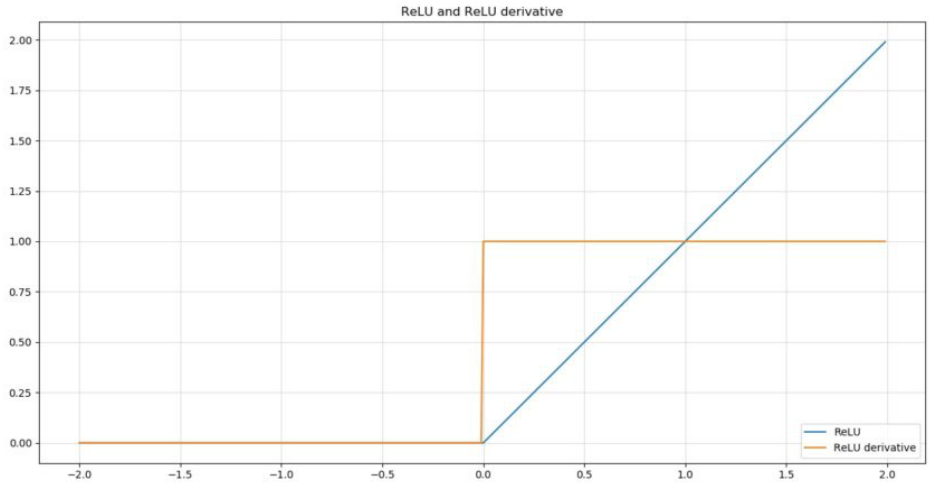
the input. It allows only positive values to pass through it. The negative values are mapped to zero. It allows for backpropagation.

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. What does this mean? If you look at the ReLU function if the input is negative it will convert it to zero and the neuron does not get activated.

$$ReLU(x) = max(0, x)$$







As you can see, the derivative of ReLU behaves differently. If the original input is < 0, the derivative is 0, else it is 1

Dying ReLU problem occurs when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn. The Dying ReLU refers to neuron which outputs 0 for your data in training set. This happens because

sum of weight * inputs in a neuron (also called activation) becomes <= 0 for all input patterns. This causes ReLU to output 0. As derivative of ReLU is 0 in this case, no weight updates are made and neuron is stuck at outputting 0. The unit dies when it only outputs 0 for any given input. When training with stochastic gradient descent, the unit is not likely to return to life, and the unit will no longer be useful during training. Such neurons are not playing any role in discriminating the input and is essentially useless. Over the time you may end up with a large part of your network doing nothing. Once a ReLU ends up in this state, it is unlikely to recover, because the function gradient at 0 is also 0, so gradient descent learning will not alter the weights.

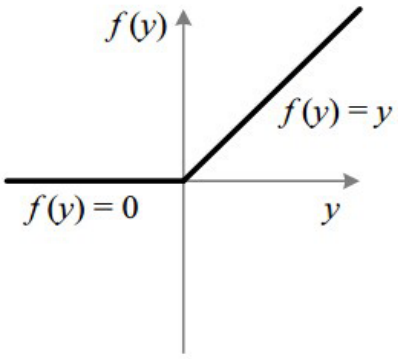
Leaky ReLU

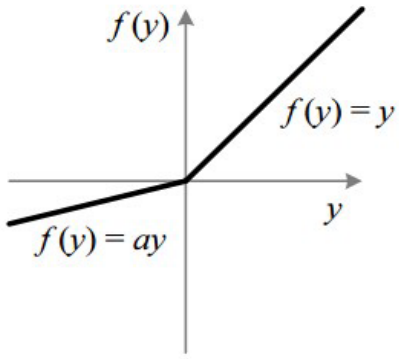
It is an attempt to solve the dying ReLU problem. This variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values

Leaky ReLU function is nothing but an improved version of the ReLU function. Instead of defining the ReLU function as 0 for x less than 0, we define it as a small linear component of x.

The leak helps to increase the range of the ReLU function. Usually, the value of a is 0.01 or so. When a is not 0.01 then it is called Randomized ReLU. Therefore, the range of the Leaky ReLU is (-infinity to infinity).

$$LeakyReLU(x) = \begin{cases} x, & \text{if } x > 0. \\ \alpha x, & \text{otherwise.} \end{cases}$$





Points to consider

- The sigmoid function suffers from vanishing gradient problem because the derivative of sigmoid ranges from 0 to 0.25 as its output ranges from 0 to 1

- The Tanh function also suffers from vanishing gradient problem because the derivative varies from 0 and 1 as its output ranges from -1 to 1.

- The ReLU function is highly computationally efficient but is not able to process inputs that approach zero or negative. It doesn't suffer from vanishing gradient problem. Its limitation is that it should only be used within Hidden layers of a Neural Network Model.

- The Leaky ReLU function has a small positive slope in its negative area, enabling it to process zero or negative values. Leaky ReLU allows the negative slope to be learned, performing backpropagation to learn the most effective slope for zero and negative input values.

- Sigmoid, Tanh and ReLU activation functions can all suffer from exploding gradient problems because if the weights are too high and derivative also becomes high, then exploding gradient problem may occur.

Vanishing Gradient Problem - Vanishing gradients lead to slow model convergence. Vanishing Gradient Problem occurs when we try to train a Neural Network model using Gradient based optimization techniques.

The gradients coming from the deeper layers have to go through continuous matrix multiplications because of the chain rule and as they approach the earlier layers, if they have small values (<1), they shrink exponentially until they vanish and make it impossible for the model to learn, this is the vanishing gradient problem.

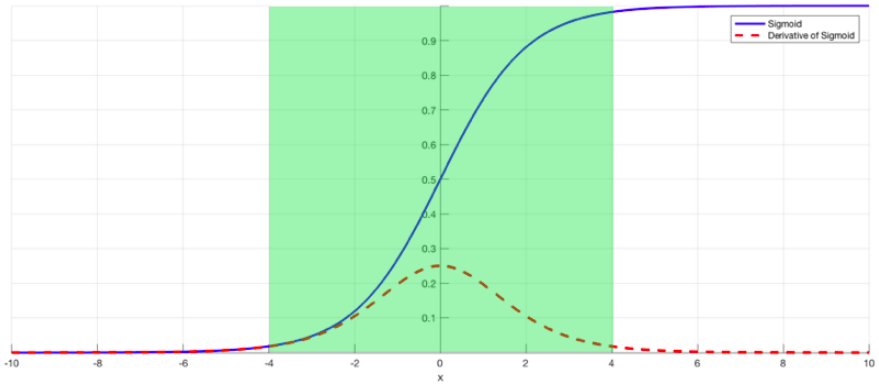
The term vanishing gradient refers to the fact that in a feedforward network (FFN) the back propagated error signal typically decreases (or increases) exponentially as a function of the distance from the final layer. As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

When n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers. When the gradient becomes negligible, subtracting it from original matrix doesn't makes any sense and hence the model stops learning. This problem is called as Vanishing Gradient Problem.

Why this problem occurs?

Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small. This is the reason why

we do not use Sigmoid and Tanh as Activation functions. Hence mostly nowadays we use RELU based activation functions



Why Earlier layers of the Network are so important to us?

Earlier layers in the Network are important because they are responsible to learn and detecting the simple patterns and are actually the building blocks of our Network. Obviously, if they give improper and inaccurate results, then how can we expect the next layers and the complete Network to perform nicely and produce accurate results.

What harm does it do to our Model?

- The Training process takes too long

Your deep Learning model may take longer time to train and learn from the data and sometimes may not train at all and show error. This results in less or no convergence of the neural network.

- Prediction Accuracy of the Model will decrease

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

How to deal with Vanishing Gradient?

Vanishing Gradient can be overcome with

- By the use of ReLU activation function.

- By the use of LSTM and GRU.

Exploding Gradient Problem - Exploding gradients are a problem when large error gradients accumulate and result in very large updates to neural network model weights during training. That is why the gradients of the cost with the respect to the parameters are too big. This leads the cost to oscillate around its minimum value.

When the magnitudes of the gradients accumulate, an unstable network is likely to occur, which can cause poor prediction results or even a model that reports nothing useful what so ever. When there are exploding gradients, an unstable network can result and the learning cannot be completed. The values of the weights can also become so large as to overflow and result in something called NaN values. NaN values, which stands for not a number, are values that represent an undefined or unpresentable values

How do You Know if You Have Exploding Gradients?

There are some subtle signs that you may be suffering from exploding gradients during the training of your network, such as:

The model is unable to get traction on your training data (e.g. poor loss).

The model is unstable, resulting in large changes in loss from update to update.

The model loss goes to NaN during training.

How to deal with Exploding Gradients?

Exploding Gradient can be overcome with

- By the use of RMSprop to adjust learning rate.

- By redesigning the network to have fewer layers.

- By the use of LSTM and GRU

Weight Initialization Techniques

Why do we need to select right initialization technique?

To achieve convergence in optimal time.

To prevent vanishing/exploding gradients.

Weight initialization is the most important step while training the neural network. If weights are high, it may lead to exploding gradient. If weights are low, it may lead to vanishing gradient.

Due to these issues, our model may take a long time to converge to global minima or sometimes it may never converge. When you are working with deep neural networks, initializing the network with the right weights can be the difference between the network converging in a reasonable amount of time and the network loss function not going anywhere even after hundreds of thousands of iterations

If the weights are too small, then the variance of the input signal starts diminishing as it passes through each layer in the network. The input eventually drops to a really low value and can no longer be useful

If the weights are too large, then the variance of input data tends to rapidly increase with each passing layer. Eventually it becomes so large that it becomes useless. Why would it become useless? Because the sigmoid function tends to become flat for larger values, as we can see the graph above. This means that our activations will become saturated and the gradients will start approaching zero.

All-zeros initialization

Of course, it is possible to initialize all neurons with all-zero weight vectors. However, this is a very bad idea, since effectively you'll start training your network while all your neurons are dead. It is considered to be poor practice by the deep learning community.

Recall that in a neuron, output=w·x+b.

Or:

output=w·x+b =w1x1+…+wnxn+b

Now, if you initialize w as an all-zeros vector, w1…wn will all be zero. And since anything multiplied by zero is zero, you see that with zero initialization, the input vector x no longer plays a role in computing the output of the neuron.

Zero initialization would thus produce poor models that, generally speaking, do not perform better than linear ones

Random initialization

It's also possible to perform random initialization. We can use two distributions for initializing the weights randomly - Gaussian distribution and Uniform distribution

Effectively, you'll simply initialize all the weights vectors randomly. Since they then have numbers > 0, your neurons aren't dead and will work from the start. You'll only have to expect that performance is (very) low the first couple of epochs simply because those random values likely do not correspond to the actual distribution underlying the data. With random initialization, you'll therefore see an exponentially decreasing loss, but then inverted – it goes fast first, and plateaus later.

There's however two types of problems that you can encounter when you initialize your weights randomly: the vanishing gradients problem and the exploding gradients problem. If you initialize your weights randomly, two scenarios may occur:

Your weights are very small. Backpropagation, which computes the error backwards, chains various numbers from the loss towards the updateable layer. Since 0.1 x 0.1 x 0.1 x 0.1 is very small, the actual gradient to be taken at that layer is really small (0.0001). Consequently, with random initialization, in the case of very small weights – you may encounter vanishing gradients. When your weights and hence your gradients are close to zero, the gradients in your upstream layers vanish because you're multiplying small values and e.g. 0.1 x 0.1 x 0.1 x 0.1 = 0.0001. Hence, it's going to be difficult to find an optimum, since your upstream layers learn slowly. That is, the farther from the end the update takes place, the slower it goes. This might yield that your model does not reach its optimum in the time you'll allow it to train.

In another case, you experience the exploding gradients scenario. In that case, your initialized weights are very much off, perhaps because they are really large, and by consequence a large weight swing must take place. Similarly, if this happens throughout many layers, the weight swing may be large and multiplications become really strong: 10^6·10^6·10^6·10^6=1024. The gradients may therefore also explode, causing number overflows in your upstream layers, rendering them untrainable (even dying off the neurons in those layers) Two things may happen then: first, because of the large weight swing, you may simply not reach the optimum for that particular neuron (which often requires taking small steps). Second, weight swings can yield number overflows in e.g. Python, so that the language can no longer process those large numbers. The result, Nan's (Not a Number), will reduce the power of your network.

Xavier (or Glorot) initialization

We use Gaussian distribution to randomly distribute these weights such that the mean of the distribution is zero and standard deviation is one. But the problem with this approach was that variance or standard deviation tend to change in next layers which lead to explode or vanish the gradients.

Deep neural networks face the difficulty that variance of the layer outputs gets lower the more upstream the data you go. It's best to have variances of ≈ 1 through all layers. This way, slow learning can be mitigated quite successfully.

With each passing layer, we want the variance or standard deviation to remain the same. This helps us keep the signal from exploding to a high value or vanishing to zero. In other words, we need to initialize the weights in such a way that the variance remains the same with each passing layer. This initialization process is known as Xavier initialization

In Xavier initialization technique, we need to pick the weights from a Gaussian distribution with zero mean and a variance of 1/N (instead of 1), where N specifies the number of input neurons.

Let's consider a linear neuron:

y = w1x1 + w2x2 + ... + wNxN + b

With each passing layer, we want the variance to remain the same. This helps us keep the signal from exploding to a high value or vanishing to zero. In other words, we need to initialize the weights in such a way that the variance remains the same for x and y. This initialization process is known as Xavier initialization.

How to perform Xavier initialization?

Just to reiterate, we want the variance to remain the same as we pass through each layer. Let's go ahead and compute the variance of y:

var(y) = var (w1x1 + w2x2 + ... + wNxN + b)

Let's compute the variance of the terms inside the parentheses on the right hand side of the above equation. If you consider a general term, we have:

var(wixi) = E(xi)2var(wi) + E(wi)2var(xi) + var(wi)var(xi)

Here, E () stands for expectation of a given variable, which basically represents the mean value. We have assumed that the inputs and weights are coming from a Gaussian distribution of zero mean. Hence the "E ()" term vanishes and we get:

var(wixi) = var(wi)var(xi)

Note that 'b' is a constant and has zero variance, so it will vanish. Let's substitute in the original equation:

var(y) = var(w1)var(x1) + ... + var(wN)var(xN)

Since they are all identically distributed, we can write:

var(y) = N * var(wi) * var(xi)

So if we want the variance of y to be the same as x, then the term "N * var(wi)" should be equal to 1. Hence:

N * var(wi) = 1

var(wi) = 1/N

std(wi)=1/sqrt(N)

There we go! We arrived at the Xavier initialization formula. We need to pick the weights from a Gaussian distribution with zero mean and a variance of 1/N, where N specifies the number of input neurons.

He initialization

It attempts to do the same but difference is related to the nonlinearities of the ReLU activation function, which make it non-differentiable at x=0.

We need to pick the weights from a Gaussian distribution with zero mean and a variance of 2/N, where N specifies the number of input neurons.

var(wi) =2/N

std(wi)=2/sqrt(N)

For Tanh based activating neural nets, the Xavier initialization seems to be a good strategy, which essentially performs random initialization from a distribution with a variance of $1/\sqrt{N}$

ReLU activating networks, which are pretty much the standard ones today, benefit from the He initializer - which does the same thing, but with a different variance, namely 2/N.

How does neural network works?

Artificial Neural Networks can be best viewed as weighted directed graphs, where the nodes are formed by the artificial neurons and the connection between the neuron outputs and neuron inputs can be represented by the directed edges with weights. The Artificial Neural Network receives the input signal from the external world in the form of a pattern and image in the form of a vector. These inputs are then mathematically designated by the notations x(n) for every n number of inputs. Each of the input is then multiplied by its corresponding weights (these weights are the details used by the artificial neural networks to solve a certain problem). In general terms, these weights typically represent the strength of the interconnection amongst neurons inside the artificial neural network. All the weighted inputs are summed up inside the computing unit (yet another artificial neuron). If the weighted sum equates to zero, a bias is added to make the output non-zero or else to scale up to the system's response. Here the sum of weighted inputs can be in the range of 0 to positive infinity

Training a neural network consists of 4 steps:

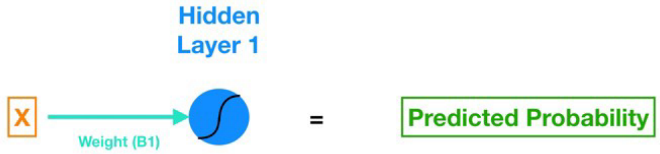
Forward propagation: Using the input X, weights W and biases b, for every layer we compute Z and A. At the final layer, we compute f(A^(L-1)) which could be a sigmoid, Softmax or linear function of A^(L-1) and this gives the prediction y_hat.

Compute the loss function: This is a function of the actual label y and predicted label y_hat. It captures how far off our predictions are from the actual target. Our objective is to minimize this loss function.

Backward Propagation: In this step, we calculate the gradients of the loss function f(y, y_hat) with respect to A, W, and b called dA, dW and db. Using these gradients, we update the values of the parameters from the last layer to the first.

Repeat steps 2–4 for n iterations/epochs till we feel we have minimized the loss function, without overfitting the train data

Example



$$Sigmoid\,(B_1 * X + B_0) = Predicted\ Probability$$

X is our input, the lone feature that we give to our model in order to calculate a prediction.

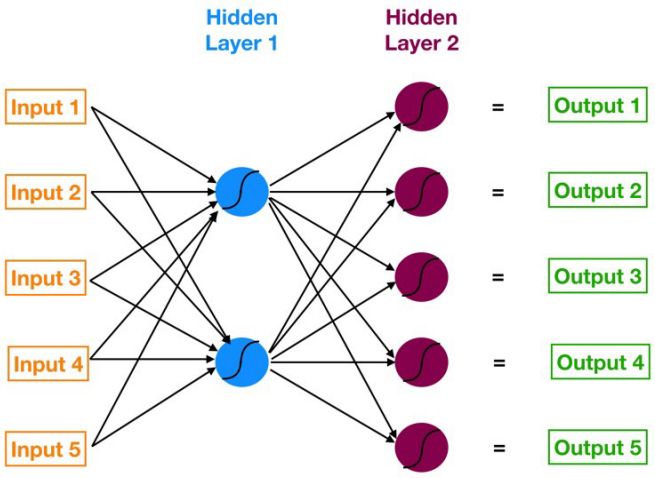
B1 is the estimated slope parameter of our logistic regression — B1 tells us by how much the Log Odds change as X changes. Notice that B1 connects the input X to the neuron in Hidden Layer 1.

B0 is the bias - very similar to the intercept term of regression. The difference is in neural networks, every neuron has its own bias term (but in regression, the model has a singular intercept term).

The neuron also includes a sigmoid activation function. Remember the sigmoid function is what we use to go from log-odds to probability (do a control-f search for “sigmoid” in my previous post).

And finally we get our predicted probability by applying the sigmoid function to the quantity (B1*X + B0)

Let’s Add a Bit of Complexity Now



Now let's calculate the outputs of each neuron in Hidden Layer 1 (known as the activations). We do so using the following formulas (W denotes weight, In denotes input).

Z1 = W1*In1 + W2*In2 + W3*In3 + W4*In4 + W5*In5 + Bias_Neuron1

Neuron 1 Activation = Sigmoid(Z1)

We can use matrix math to summarize this calculation (remember our notation rules — for example, W4,2 denotes the weight that lives in the connection between Input 4 and Neuron 2):

$$\begin{bmatrix} W_{1,1} & W_{2,1} & W_{3,1} & W_{4,1} & W_{5,1} \\ W_{1,2} & W_{2,2} & W_{3,2} & W_{4,2} & W_{5,2} \end{bmatrix} \times \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \\ X5 \end{bmatrix} + \begin{bmatrix} Bias1 \\ Bias2 \end{bmatrix} = \begin{bmatrix} Z1 \\ Z2 \end{bmatrix}$$

By repeatedly calculating [Z] and applying the activation function to it for each successive layer, we can move from input to output. This process is known as forward propagation. Now that we know how the outputs are calculated, it's time to start evaluating the quality of the outputs and training our neural network.

First let's think about what levers we can pull to minimize the cost function. In traditional linear or logistic regression, we are searching for beta coefficients (B0, B1, B2, etc.) that minimize the cost function. For a neural network, we are doing the same thing but at a much larger and more complicated scale.

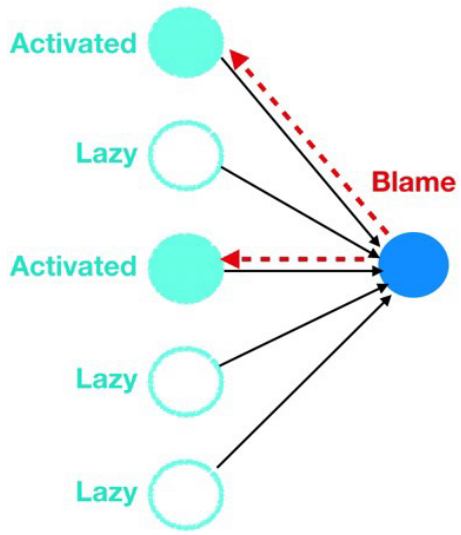
In a neural network, changing the weight of any one connection (or the bias of a neuron) has a reverberating effect across all the other neurons and their activations in the subsequent layers. That's because each neuron in a neural network is like its own little model.

We want to calculate the error attributable to each neuron (I will just refer to this error quantity as the neuron's error because saying "attributable" again and again is no fun) starting from the layer closest to the output all the way back to the starting layer of our model.

So why do we care about the error for each neuron? Remember that the two building blocks of a neural network are the connections that pass signals into a particular neuron (with a weight living in each connection) and the neuron itself (with a bias). These weights and biases across the entire network are also the dials that we tweak to change the predictions made by the model.

The magnitude of the error of a specific neuron (relative to the errors of all the other neurons) is directly proportional to the impact of that neuron's output (a.k.a. activation) on our cost function. So basically backpropagation allows us to calculate the error attributable to each neuron and that in turn allows us to calculate the partial derivatives and ultimately the gradient so that we can utilize gradient descent.

Well neurons, via backpropagation, are masters of the blame game. When the error gets back propagated to a particular neuron, that neuron will quickly and efficiently point the finger at the upstream colleague (or colleagues) who is most at fault for causing the error (i.e. layer 4 neurons would point the finger at layer 3 neurons, layer 3 neurons at layer 2 neurons, and so forth).



And how does each neuron know who to blame, as the neurons cannot directly observe the errors of other neurons? They just look at who sent them the most signal in terms of the highest and most frequent activations. Just like in real life, the lazy ones that play it safe (low and

infrequent activations) skate by blame free while the neurons that do the most work get blamed and have their weights and biases modified. Cynical yes but also very effective for getting us to the optimal set of weights and biases that minimize our cost function.