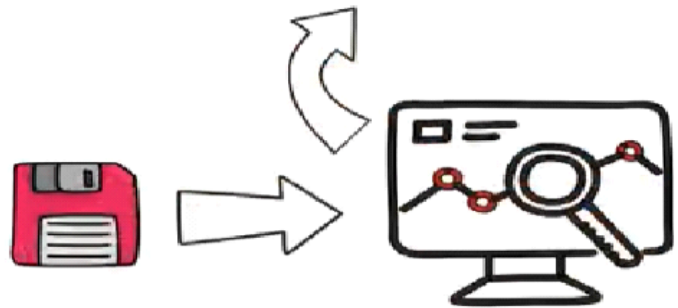


NEURAL NETWORKS

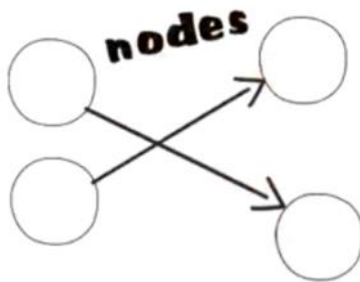
algorithms



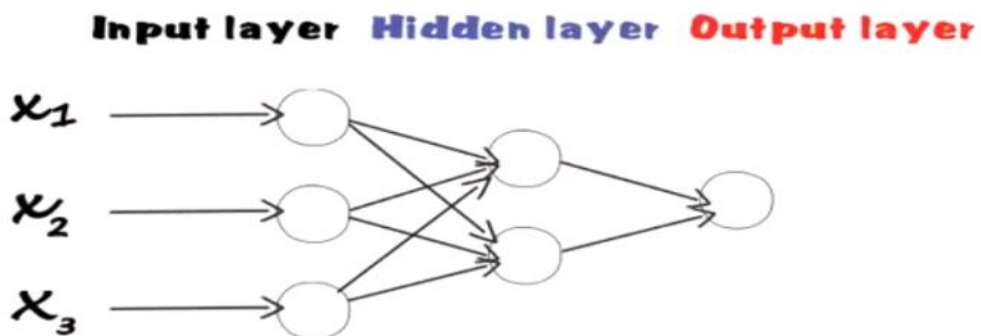
predictive models



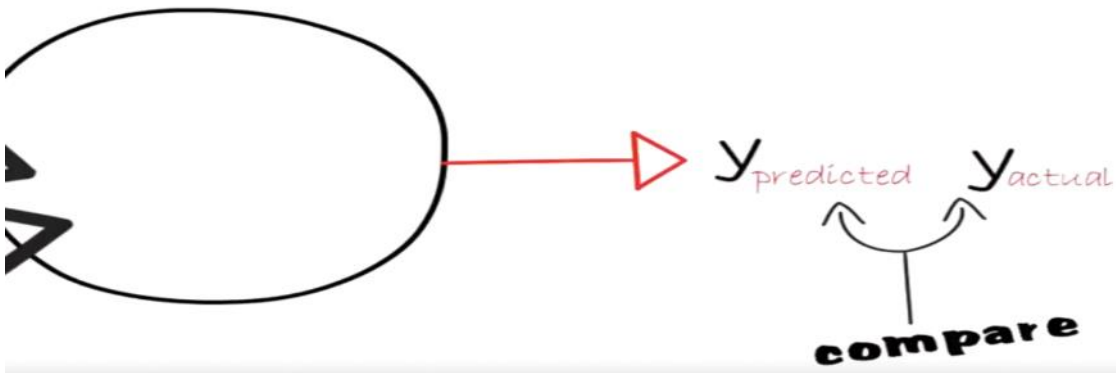
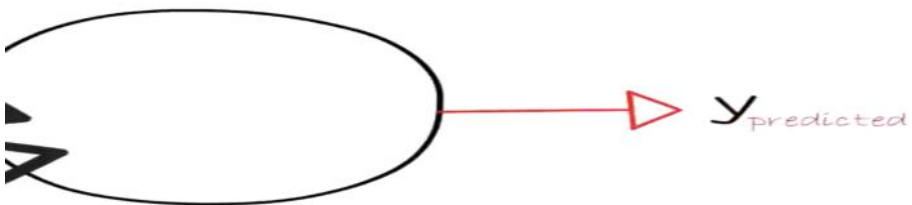
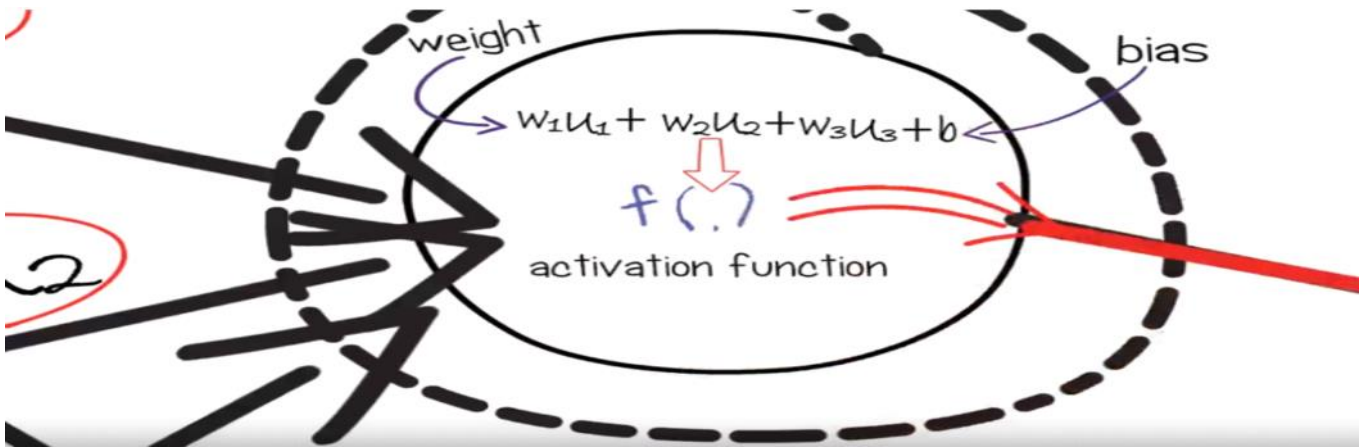
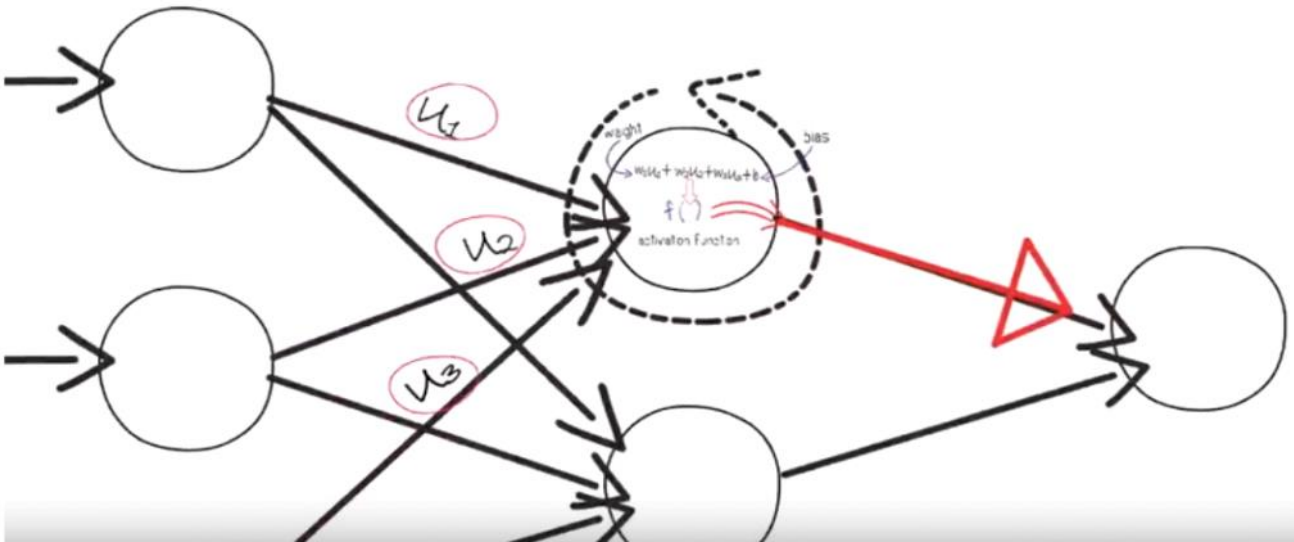
Neural Network consists of the interconnected processing unit is called node

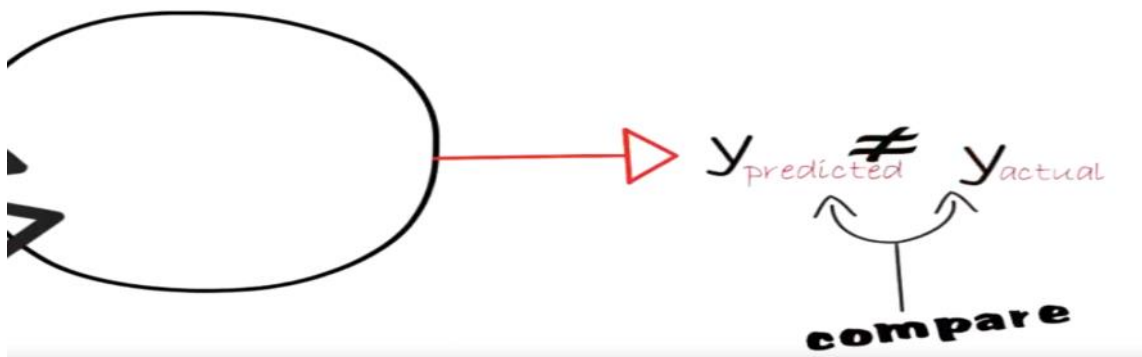


The Most common type of the ANN is Multilayer Perceptron.



Multilayer Perceptron





Compare the Predicted and actual , if it do not match than it adjust all the weight and repit the process again as below diagram.

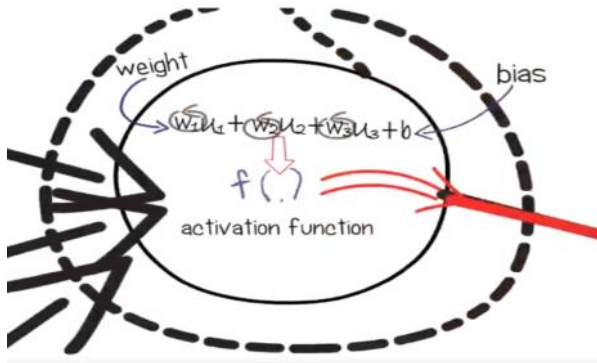
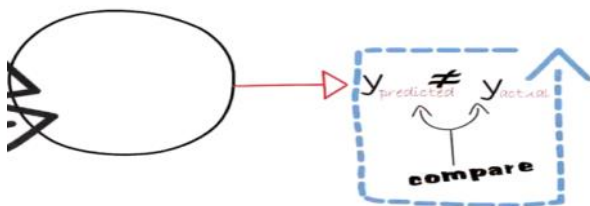
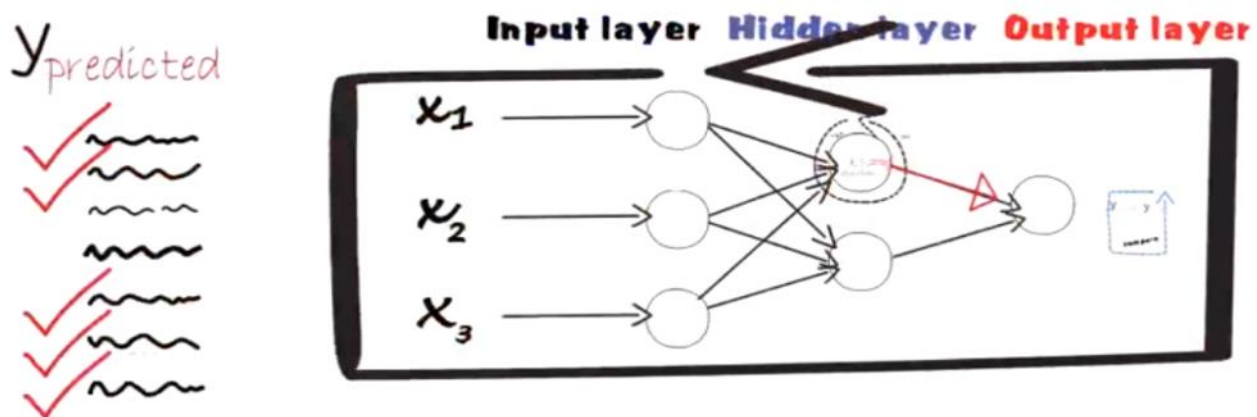


Fig: To Adjust the weight bcz Y-predicted is not same as Y-actual



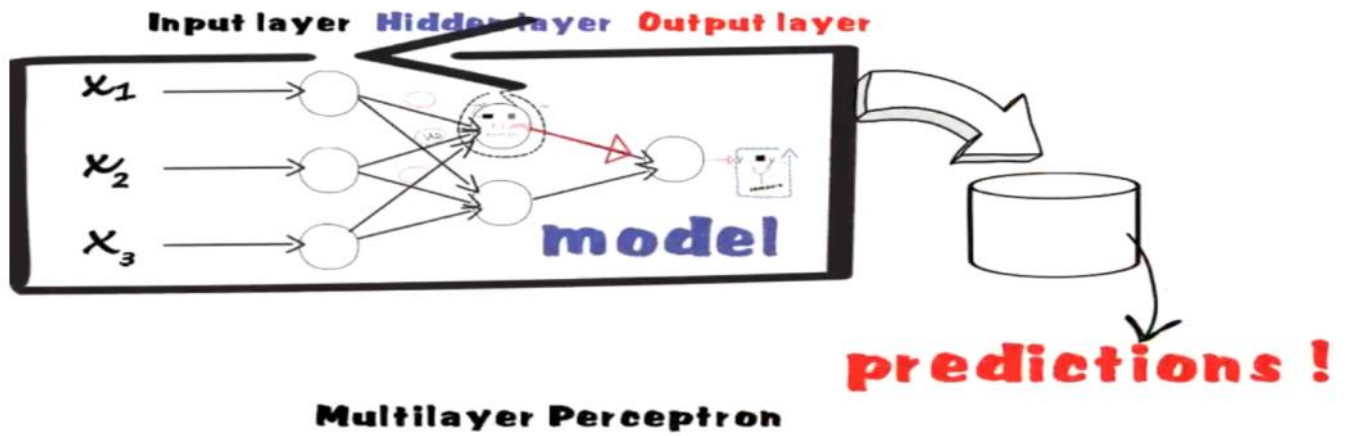
Repit the process by adjusting the weight as above.



Multilayer Perceptron

Above process will repit till to match maximum number with Y-Prediction and Y-actual

Finally to the production:



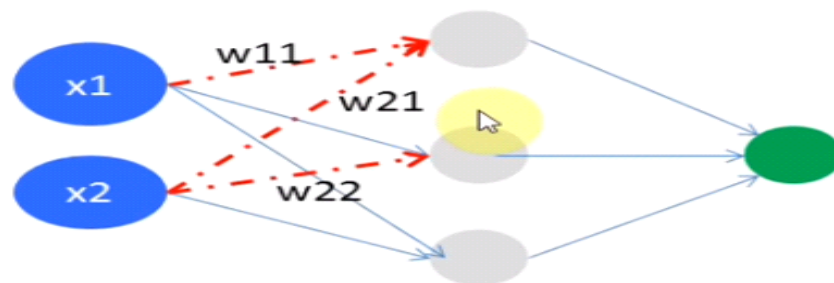
Backward propagation Algorithm

HOW DOES A NEURAL NETWORK MODEL WORK?

- Case updating
- Batch updating
- Weight and bias updation
- Intuitive understanding of it's action
- Stopping criteria
- Analyst decisions

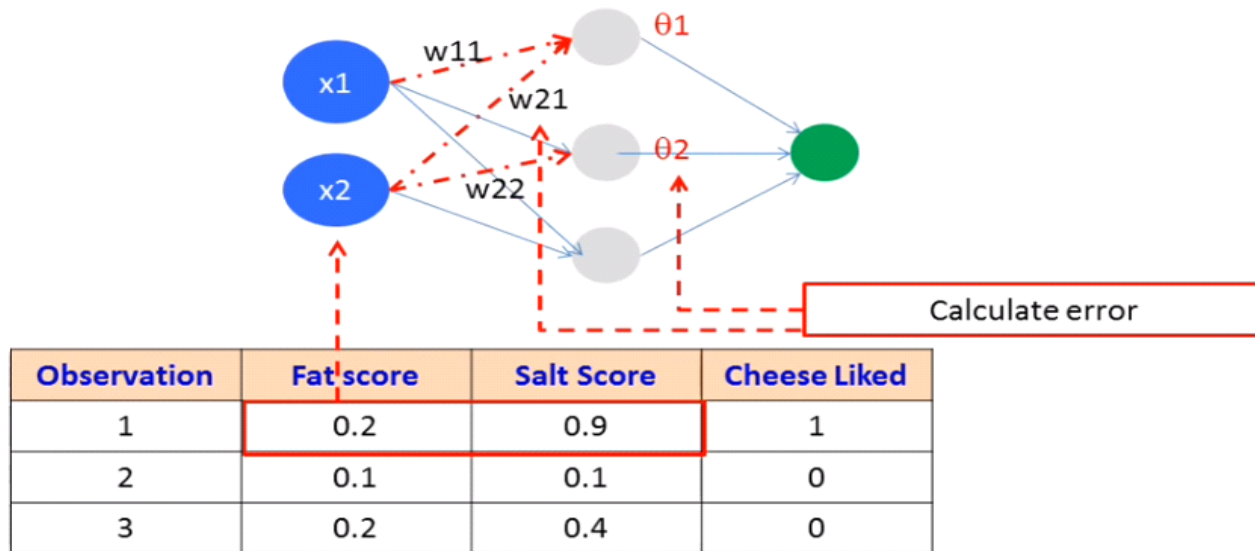
Neural Network is the Iterative approach.

Iterative approach



- Keep input nodes equal to number of independent variable
- Assign random weight to synapsis (links) – usually start with random number between -0.5 to 0.5
- Assign random biases – – usually start with random number between -0.5 to 0.5
- Calculate output $Y_p = Y_{\text{predicted}}$
- Calculate error $Y_a = Y_{\text{actual}} - Y_p$
- Back propagate error to adjust weights

Two ways of weight adjustment

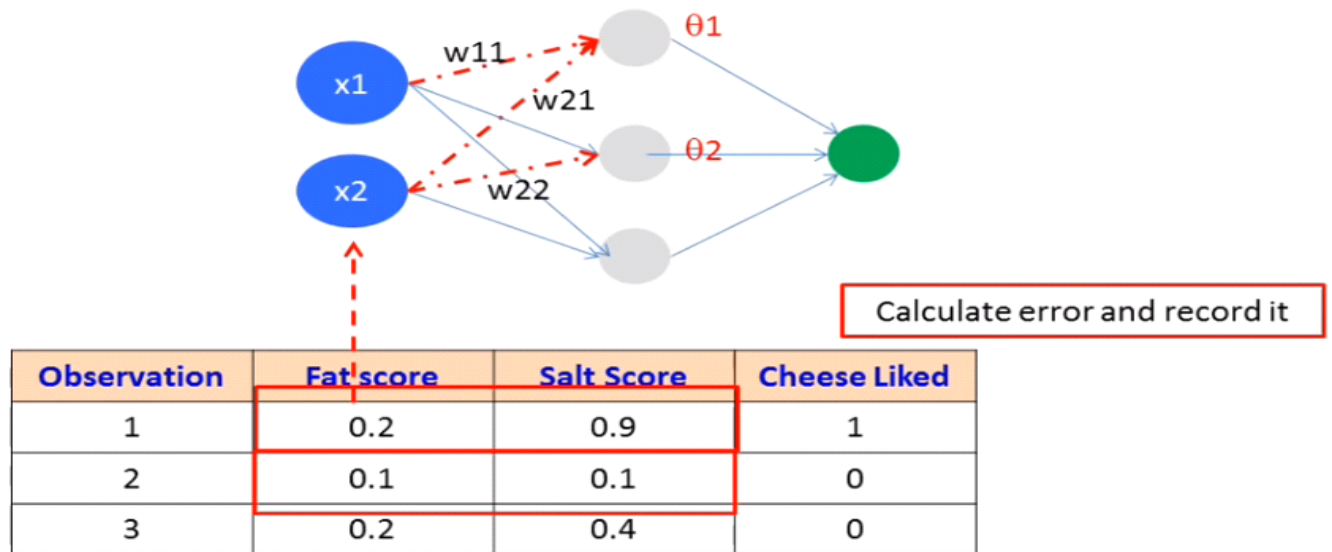


- Pass one observation at a time
- Calculate error
- Backward propagate to adjust weights and biases

Case Updating

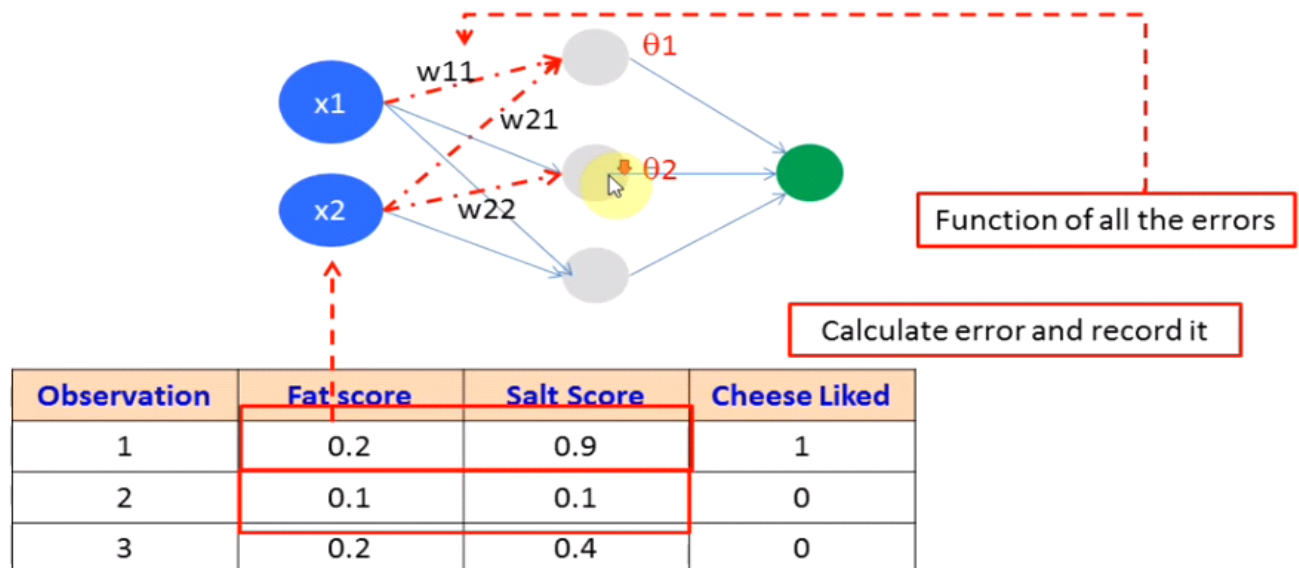
- Weights are updated after each record is run through the network
- Completion of all records through the network is one *epoch* (also called *sweep* or *iteration*)
- After one epoch is completed, return to first record and repeat the process

Batch updating



- Pass one observation at a time, calculate error and keep a record of error
- Pass another observation, calculate error and keep a record
- Adjust weights and biases with the function of all the errors, when all records have passed

Batch updating



- Pass one observation at a time, calculate error and keep a record of error
- Pass another observation, calculate error and keep a record
- Adjust weights and biases with the function of all the errors, when all records have passed

How errors are used...

- Say current weights and biases of neural net produces output \hat{y}_k
- The expected output is y_k
- Then the error is obtained as $err_k = \hat{y}_k(1 - \hat{y}_k)(y_k - \hat{y}_k)$
- So if the output is say 0.6, where expected output was 2.4, then error = $0.6 * (1 - 0.6) * (2.4 - 0.6)$
- Now this error is used to update weights and biases like

The diagram illustrates the weight update formula for a neural network. It shows two equations: $\theta_j^{new} = \theta_j^{old} + l(err_j)$ and $w_j^{new} = w_j^{old} + l(err_j)$. A red box highlights the old weights and biases (θ_j^{old} and w_j^{old}). A yellow circle highlights the new weights and biases (θ_j^{new} and w_j^{new}). A green box highlights the error term (err_j). A green arrow points from the error term to the learning rate l . A blue double-headed arrow indicates the relationship between the learning rate and the error term. A red arrow points from the old weights and biases to the new weights and biases. The text 'Old Weights And old biases' is written in red below the equations. The text 'Learning Rate (constant)' is written in blue below the equations. The word 'Error' is written in green to the right of the equations.

$$\theta_j^{new} = \theta_j^{old} + l(err_j)$$
$$w_j^{new} = w_j^{old} + l(err_j)$$

Old Weights
And old biases

Learning Rate
(constant)

Error

Why It Works

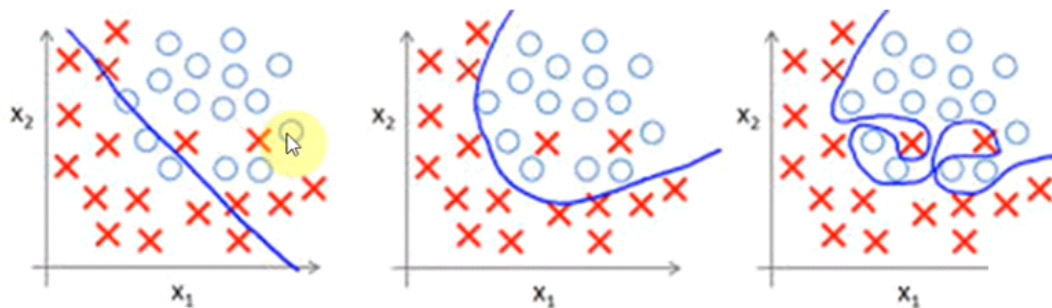
- Because errors are used to adjust the weight
- Adjustment is proportionate to the error
 - Big errors lead to big changes in weights
 - Small errors leave weights relatively unchanged
- Over many such updates, a given weight keeps changing until the error associated with that weight is negligible
- At this point weights change little

Stopping criteria

- When weights and bias hardly changes
- When errors are below a threshold
- When maximum number of iterations has reached

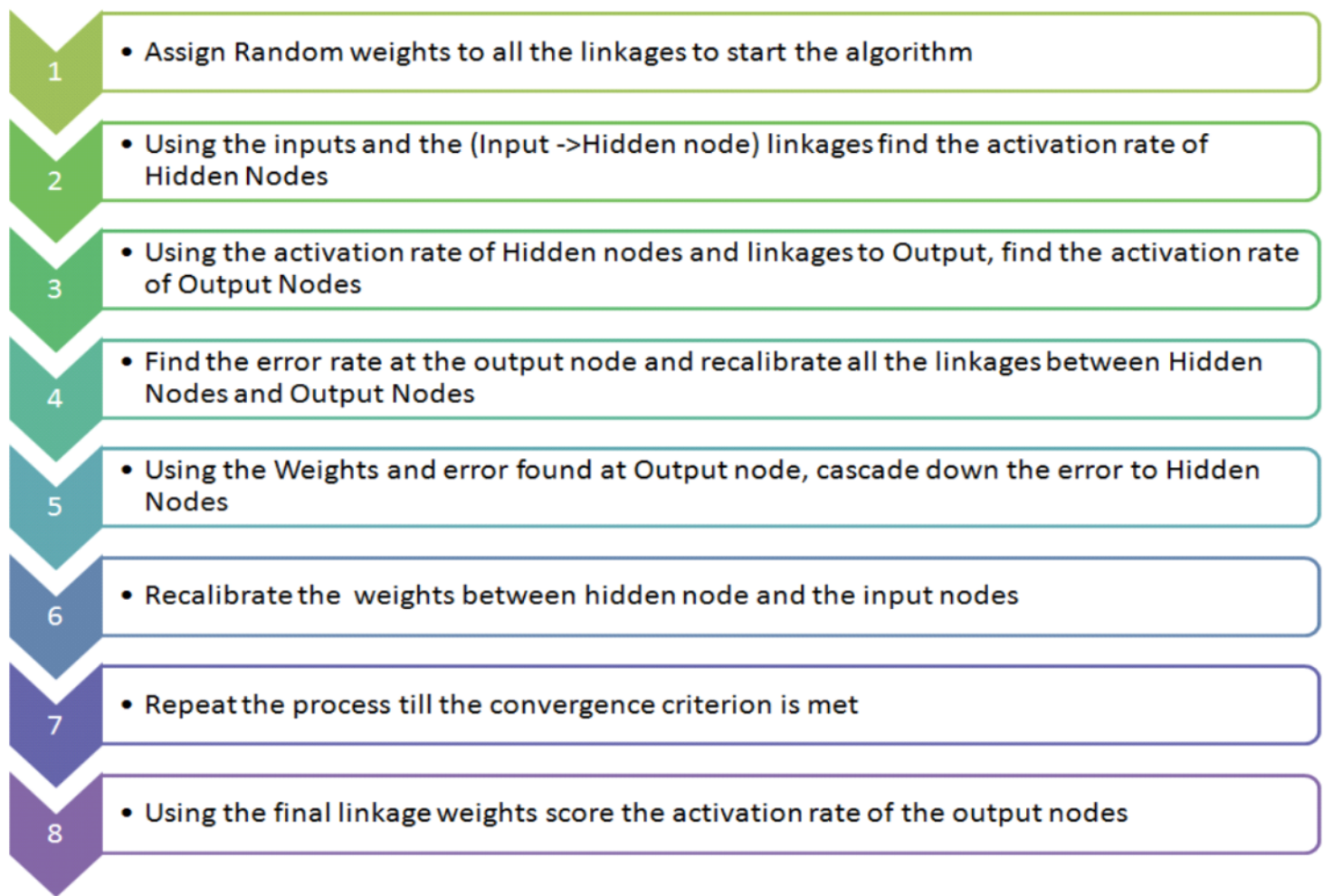
Analyst decisions

- How many internal layers should be there?
 - What should be the learning / optimization method to adjust the weights of synapses and biases?
 - How many nodes should be there on each layer?
 - What should the activation functions in different node?
 - How to avoid over fitting?
-
- Overfitting is like error modelling



Step to Step Process for ANN

Saturday, September 15, 2018 7:37 PM



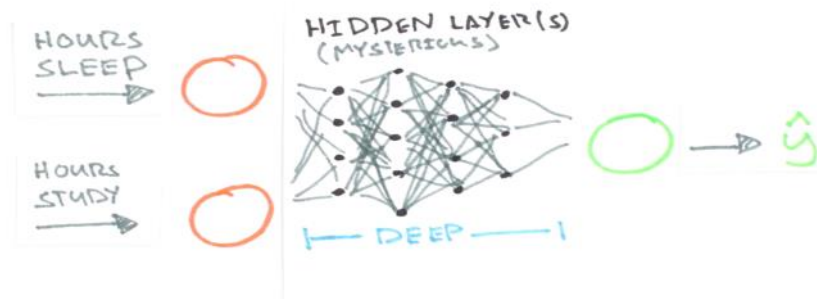
How to Build a Neural Network

[Artificial neural networks](#) are statistical learning models, inspired by biological neural networks (central nervous systems, such as the brain), that are used in [machine learning](#). These networks are represented as systems of interconnected “neurons”, which send messages to each other. The connections within the network can be systematically adjusted based on inputs and outputs, making them ideal for supervised learning.

Neural networks can be intimidating, especially for people with little experience in machine learning and cognitive science! However, through code, this tutorial will explain how neural networks operate. By the end, you will know how to build your own flexible, learning network, similar to **Mind**

Understanding the Mind

A neural network is a collection of “neurons” with “synapses” connecting them. The collection is organized into three main parts: the input layer, the hidden layer, and the output layer. Note that you can have n hidden layers, with the term “deep” learning implying multiple hidden layers.



Hidden layers are necessary when the neural network has to make sense of something really complicated, contextual, or non obvious, like image recognition. The term “deep” learning came from having many hidden layers. These layers are known as “hidden”, since they are not visible as a network output

The circles represent neurons and lines represent synapses. Synapses take the input and multiply it by a “weight” (the “strength” of the input in determining the output). Neurons add the outputs from all synapses and apply an activation function.

Training a neural network basically means calibrating all of the “weights” by repeating two key steps, forward propagation and back propagation.

Since neural networks are great for regression, the best input data are numbers (as opposed to discrete values, like colors or movie genres, whose data is better for statistical classification models). The output data will be a number within a range like 0 and 1 (this ultimately depends on the activation function—more on this below).

In forward propagation, we apply a set of weights to the input data and calculate an output. For the first forward propagation, the set of weights is selected randomly.

In back propagation, we measure the margin of error of the output and adjust the weights accordingly to decrease the error.

Neural networks repeat both forward and back propagation until the weights are calibrated to accurately predict an output.

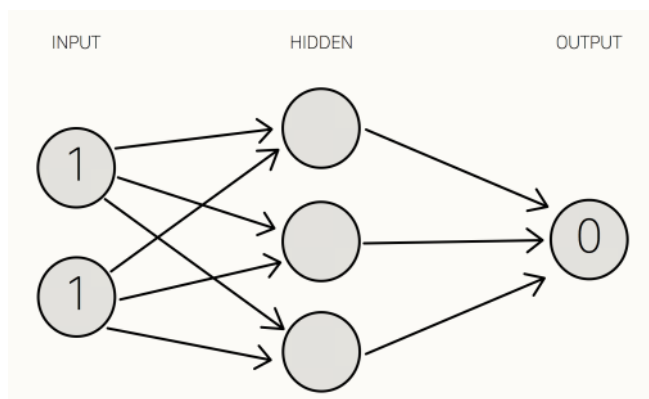
PROBLEM STATEMENT

neural network to function as an [“Exclusive or” \(“XOR”\) operation](#)

The XOR function can be represented by the mapping of the below inputs and outputs, which we’ll use as training data. It should provide a correct output given any input acceptable by the XOR function.

input output	
0, 0	0
0, 1	1
1, 0	1
1, 1	0

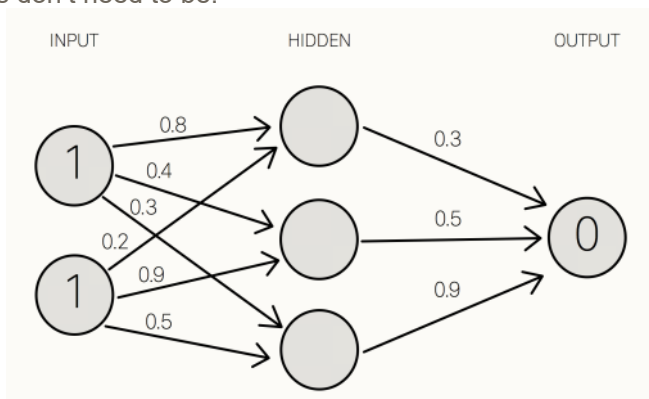
Let’s use the last row from the above table, $(1, 1) \Rightarrow 0$, to demonstrate forward propagation:



Note that we use a single hidden layer with only three neurons for this example.

We now assign weights to all of the synapses. Note that these weights are selected randomly (based on Gaussian

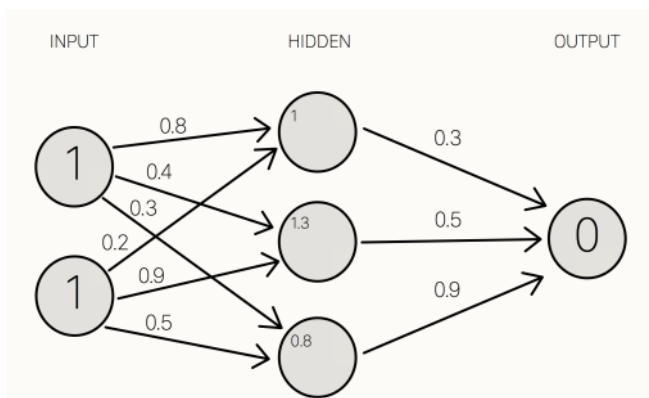
distribution) since it is the first time we're forward propagating. The initial weights will be between 0 and 1, but note that the final weights don't need to be.



We sum the product of the inputs with their corresponding set of weights to arrive at the first values for the hidden layer. You can think of the weights as measures of influence the input nodes have on the output.

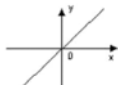
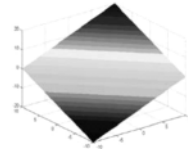
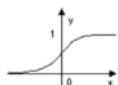
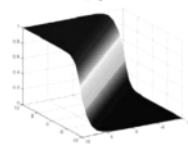
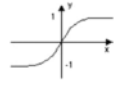
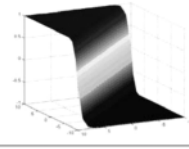
$$\begin{aligned} 1 * 0.8 + 1 * 0.2 &= 1 \\ 1 * 0.4 + 1 * 0.9 &= 1.3 \\ 1 * 0.3 + 1 * 0.5 &= 0.8 \end{aligned}$$

We put these sums smaller in the circle, because they're not the final value:

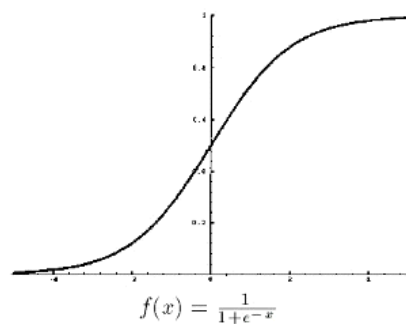


To get the final value, we apply the [activation function](#) to the hidden layer sums. The purpose of the activation function is to transform the input signal into an output signal and are necessary for neural networks to model complex non-linear patterns that simpler models might miss.

There are many types of activation functions—linear, sigmoid, hyperbolic tangent, even step-wise. If you want to know more know why one function is better than another, please check the like as above activation function. [Hyperlink](#).

Activation Function	Mathematical Equation	2D Graphical Representation	3D Graphical Representation
Linear	$y = x$		
Sigmoid (logistic)	$y = \frac{1}{1 + e^{-x}}$		
Hyperbolic tangent	$y = \frac{1 - e^{-2x}}{1 + e^{2x}}$		

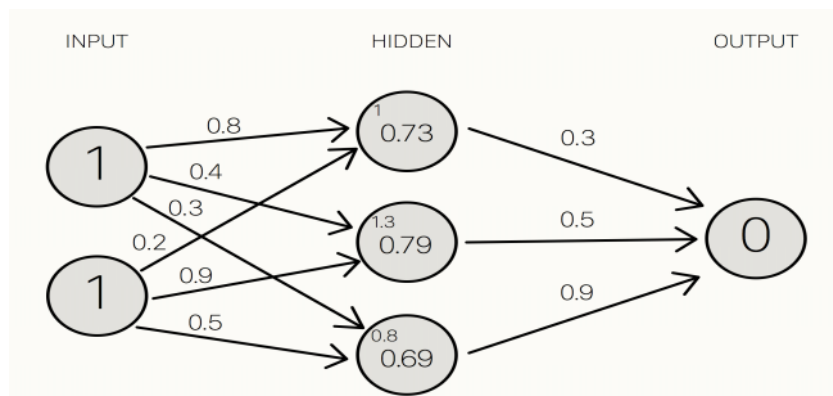
For our example, let's use the [sigmoid function](#) for activation. The sigmoid function looks like this, graphically:



And applying S(x) to the three hidden layer *sums*, we get:

S(1.0) = 0.73105857863
S(1.3) = 0.78583498304
S(0.8) = 0.68997448112

We add that to our neural network as hidden layer *results*:



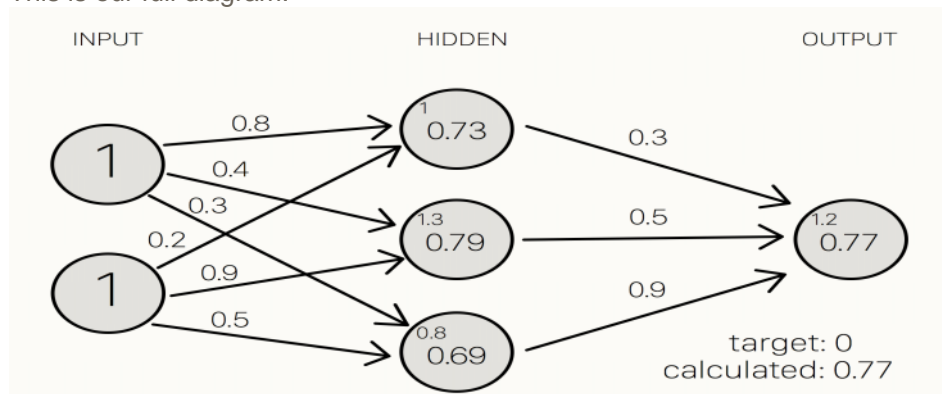
Then, we sum the product of the hidden layer results with the second set of weights (also determined at random the first time around) to determine the output sum.

$$0.73 * 0.3 + 0.79 * 0.5 + 0.69 * 0.9 = 1.235$$

..finally we apply the activation function to get the final output result.

$$S(1.235) = 0.7746924929149283$$

This is our full diagram:



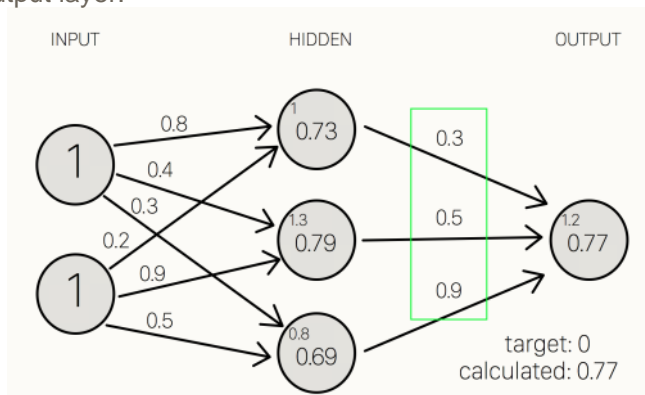
Because we used a random set of initial weights, the value of the output neuron is off the mark; in this case by +0.77 (since the target is 0). **If we stopped here, this set of weights would be a great neural network for inaccurately representing the XOR operation.**

Let's fix that by using back propagation to adjust the weights to improve the network!

Back propagation

To improve our model, we first have to quantify just how wrong our predictions are. Then, we adjust the weights accordingly so that the margin of errors are decreased.

Similar to forward propagation, back propagation calculations occur at each "layer". We begin by changing the weights between the hidden layer and the output layer.



Calculating the incremental change to these weights happens in two steps: 1) we find the margin of error of the output result (what we get after applying the activation function) to back out the necessary change in the output sum (we call this *delta output sum*) and 2) we extract the change in weights by multiplying *delta output sum* by the hidden layer results.

The *output sum margin of error* is the target output result minus the calculated output result:

$$\text{Output sum margin of error} = \text{target} - \text{calculated}$$

And doing the math:

Target = 0
 Calculated = 0.77
 Target - calculated = -0.77

To calculate the necessary change in the output sum, or *delta output sum*,

we take the derivative of the activation function and apply it to the output sum.

In our example, the activation function is the sigmoid function.

To refresh your memory, the activation function, sigmoid, takes the sum and returns the result:

$$S(\text{sum}) = \text{result}$$

So the derivative of sigmoid, also known as sigmoid prime, will give us the rate of change (or “slope”) of the activation function at the output sum:

$$S'(\text{sum}) = \frac{d\text{sum}}{d\text{result}}$$

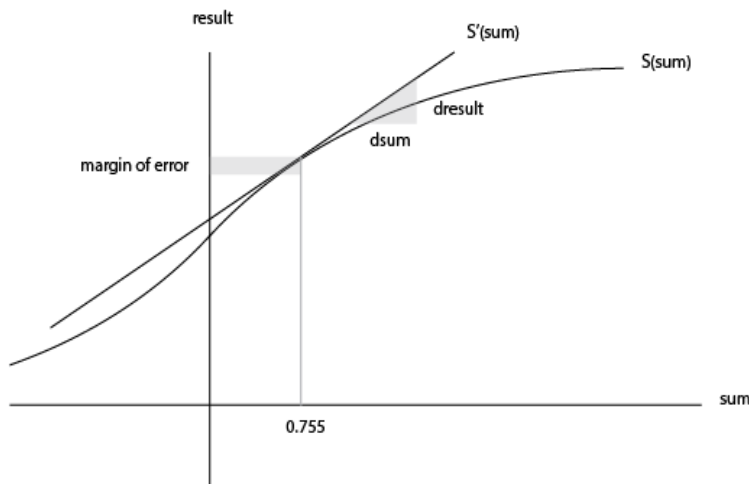
Since the output sum margin of error is the difference in the result, we can simply multiply that with the rate of change to give us the delta output sum:

$$\frac{d\text{sum}}{d\text{result}} \times (\text{target result} - \text{calculated result}) = \Delta\text{sum}$$

Conceptually, this means that the change in the output sum is the same as the sigmoid prime of the output result. Doing the actual math, we get:

$$\begin{aligned} \text{Delta output sum} &= S'(\text{sum}) * (\text{output sum margin of error}) \\ \text{Delta output sum} &= S'(1.235) * (-0.77) \\ \text{Delta output sum} &= -0.13439890643886018 \end{aligned}$$

Here is a graph of the Sigmoid function to give you an idea of how we are using the derivative to move the input towards the right direction. Note that this graph is not to scale.



Now that we have the proposed change in the output layer sum (-0.13), let's use this in the derivative of the output sum function to determine the new change in weights.

As a reminder, the mathematical definition of the output sum is the product of the hidden layer result and the weights between the hidden and output layer:

$$H_{\text{result}} \times w_{h \rightarrow o} = O_{\text{sum}}$$

The derivative of the output sum is:

$$\frac{dO_{\text{sum}}}{dw_{h \rightarrow o}} = H_{\text{results}}$$

..which can also be represented as:

$$dw_{h \rightarrow o} = \frac{dO_{\text{sum}}}{H_{\text{results}}}$$

This relationship suggests that a greater change in output sum yields a greater change in the weights; input neurons with the biggest contribution (higher weight to output neuron) should experience more change in the connecting synapse.

Let's do the math:

```
hidden result 1 = 0.73105857863
hidden result 2 = 0.78583498304
hidden result 3 = 0.68997448112
```

```
Delta weights = delta output sum / hidden layer results
Delta weights = -0.1344 / [0.73105, 0.78583, 0.68997]
Delta weights = [-0.1838, -0.1710, -0.1920]
```

```
old w7 = 0.3
old w8 = 0.5
old w9 = 0.9
```

```
new w7 = 0.1162
new w8 = 0.329
new w9 = 0.708
```

To determine the change in the weights between the *input and hidden* layers, we perform the similar, but notably different, set of calculations. Note that in the following calculations, we use the initial weights instead of the recently adjusted weights from the first part of the backward propagation.

Remember that the relationship between the hidden result, the weights between the hidden and output layer, and the output sum is:

$$H_{result} \times w_{h \rightarrow o} = O_{sum}$$

Instead of deriving for output sum, let's derive for hidden result as a function of output sum to ultimately find out delta hidden sum:

$$\frac{dH_{result}}{dO_{sum}} = \frac{1}{w_{h \rightarrow o}}$$

$$dH_{result} = \frac{dO_{sum}}{w_{h \rightarrow o}}$$

Also, remember that the change in the hidden result can also be defined as:

$$S'(H_{sum}) = \frac{dH_{sum}}{dH_{result}}$$

Let's multiply both sides by sigmoid prime of the hidden sum:

$$dH_{result} \times \frac{dH_{sum}}{dH_{result}} = \frac{dO_{sum}}{w_{h \rightarrow o}} \times \frac{dH_{sum}}{dH_{result}}$$

$$dH_{sum} = \frac{dO_{sum}}{w_{h \rightarrow o}} \times S'(H_{sum})$$

All of the pieces in the above equation can be calculated, so we can determine the delta hidden sum:

```
Delta hidden sum = delta output sum / hidden-to-outer weights * S'(hidden sum)
Delta hidden sum = -0.1344 / [0.3, 0.5, 0.9] * S'([1, 1.3, 0.8])
Delta hidden sum = [-0.448, -0.2688, -0.1493] * [0.1966, 0.1683, 0.2139]
Delta hidden sum = [-0.088, -0.0452, -0.0319]
```

Once we get the delta hidden sum, we calculate the change in weights between the input and hidden layer by dividing it with the input data, (1, 1). The input data here is equivalent to the hidden results in the earlier back propagation process to determine the change in the hidden-to-output weights. Here is the derivation of that relationship, similar to the one before:

$$I \times w_{i \rightarrow h} = H_{sum}$$

$$\frac{dH_{sum}}{dw_{i \rightarrow h}} = 1$$

$$dw_{i \rightarrow h} = \frac{dH_{sum}}{1}$$

Let's do the math:

input 1 = 1
input 2 = 1

Delta weights = delta hidden sum / input data
Delta weights = [-0.088, -0.0452, -0.0319] / [1, 1]
Delta weights = [-0.088, -0.0452, -0.0319, -0.088, -0.0452, -0.0319]

old w1 = 0.8
old w2 = 0.4
old w3 = 0.3
old w4 = 0.2
old w5 = 0.9
old w6 = 0.5

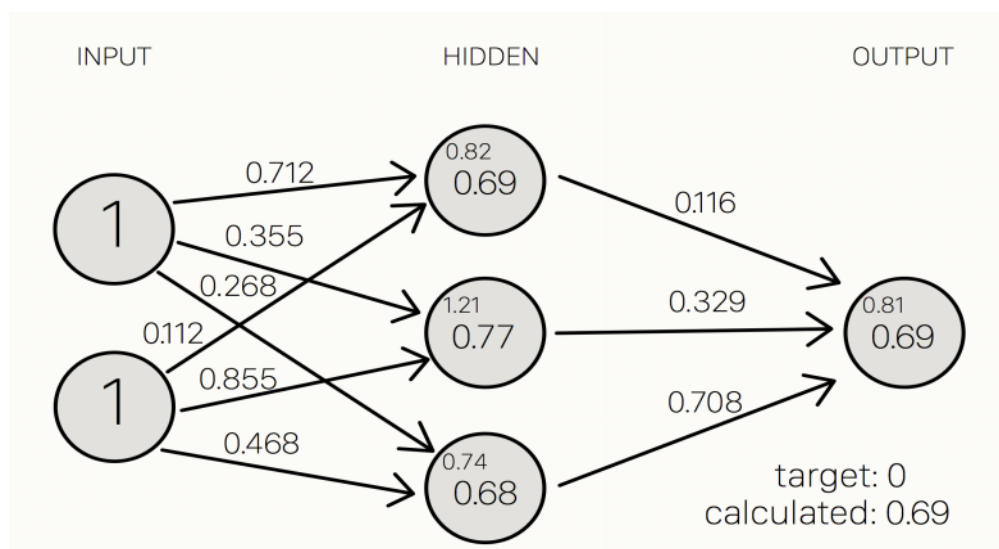
new w1 = 0.712
new w2 = 0.3548
new w3 = 0.2681
new w4 = 0.112
new w5 = 0.8548
new w6 = 0.4681

Here are the new weights, right next to the initial random starting weights as comparison:

old	new
w1: 0.8	w1: 0.712
w2: 0.4	w2: 0.3548
w3: 0.3	w3: 0.2681
w4: 0.2	w4: 0.112
w5: 0.9	w5: 0.8548
w6: 0.5	w6: 0.4681
w7: 0.3	w7: 0.1162
w8: 0.5	w8: 0.329
w9: 0.9	w9: 0.708

Once we arrive at the adjusted weights, we start again with forward propagation. When training a neural network, it is common to repeat both these processes thousands of times (by default, Mind iterates 10,000 times).

And doing a quick forward propagation, we can see that the final output here is a little closer to the expected output:



Through just one iteration of forward and back propagation, we've already improved the network!!

Introduction

Friday, September 14, 2018 7:58 PM

Recurrent Neural Networks (RNN) are a powerful and robust type of neural networks and belong to the most promising algorithms out there at the moment because they are the only ones with an internal memory.

RNN's are relatively old, like many other deep learning algorithms. They were initially created in the 1980's, but can only show their real potential since a few years, because of the increase in available computational power, the massive amounts of data that we have nowadays and the invention of LSTM in the 1990's.

Because of their internal memory, RNN's are able to remember important things about the input they received, which enables them to be very precise in predicting what's coming next.

This is the reason why they are the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more because they can form a much deeper understanding of a sequence and its context, compared to other algorithms.

Note:

Recurrent Neural Networks produce predictive results in sequential data that other algorithms can't

when do you need to use a Recurrent Neural Network ?

“Whenever there is a sequence of data and that temporal dynamics that connects the data is more important than the spatial content of each individual frame.”

Application where RNN Example:-

- software behind Siri Apple
- Google Translate

Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs

Friday, June 1, 2018 4:00 PM

Recurrent Neural Networks (RNNs) are popular models that have shown great promise in many NLP tasks. But despite their recent popularity I've only found a limited number of resources that thoroughly explain how RNNs work, and how to implement them. That's what this tutorial is about. It's a multi-part series in which I'm planning to cover the following:

1. Introduction to RNNs
2. Implementing a RNN using Python and Theano
3. Understanding the Backpropagation Through Time (BPTT) algorithm and the vanishing gradient problem
4. Implementing a GRU(gated recurrent unit -**GRU**)/LSTM RNN

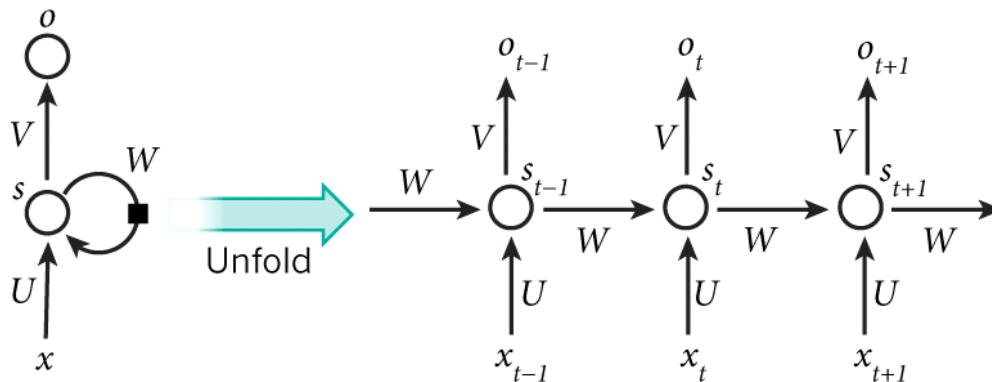
As part of the tutorial we will implement a **recurrent neural network based language model**. The applications of language models are two-fold: First, it allows us to score arbitrary sentences based on how likely they are to occur in the real world. This gives us a measure of grammatical and semantic correctness. Such models are typically used as part of Machine Translation systems.

Secondly, a language model allows us to generate new text (I think that's the much cooler application). Training a language model on Shakespeare allows us to generate Shakespeare-like text.

I'm assuming that you are somewhat familiar with basic Neural Networks. If you're not, you may want to head over to [Implementing A Neural Network From Scratch](#), which guides you through the ideas and implementation behind non-recurrent networks.

What are RNNs?

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 that's 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 depended on the previous computations. Another way to think about RNNs is that they have a "memory" which 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:



A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature

The above diagram shows a 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 a 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 a 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. $o_t = \text{softmax}(Vs_t)$.

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, a 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.

What can RNNs do?

RNNs have shown great success in many NLP tasks. At this point I should mention that the most commonly used type of RNNs are [LSTMs](#), which are much better at capturing long-term dependencies than vanilla RNNs are. But don't worry, LSTMs are essentially the same thing as the RNN we will develop in this tutorial, they just have a different way of computing the hidden state. We'll cover LSTMs in more detail in a later post. Here are some example applications of RNNs in NLP (by no means an exhaustive list).

Language Modeling and Generating Text

Given a sequence of words we want to predict the probability of each word given the previous words. Language Models allow us to measure how likely a sentence is, which is an important input for Machine Translation (since high-probability sentences are typically correct). A side-effect of being able to predict the next word is that we get a *generative* model, which allows us to generate new text by sampling from the output probabilities. And depending on what our training data is we can generate [all kinds of stuff](#). In Language Modeling our input is typically a sequence of words (encoded as one-hot vectors for example), and our output is the sequence of predicted words. When training the network we set

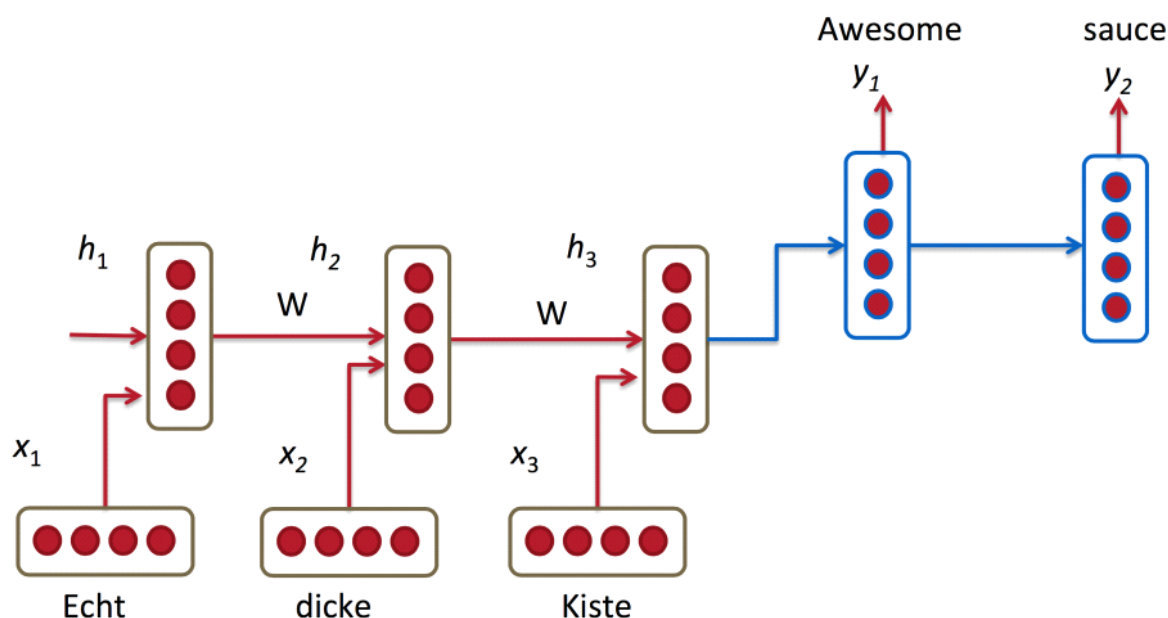
$o_t = x_{t+1}$
since we want the output at step t to be the actual next word.

Research papers about Language Modeling and Generating Text:

- [Recurrent neural network based language model](#)
- [Extensions of Recurrent neural network based language model](#)
- [Generating Text with Recurrent Neural Networks](#)

Machine Translation

Machine Translation is similar to language modeling in that our input is a sequence of words in our source language (e.g. German). We want to output a sequence of words in our target language (e.g. English). A key difference is that our output only starts after we have seen the complete input, because the first word of our translated sentences may require information captured from the complete input sequence.



Recurrent Neural Networks and LSTM

Friday, September 14, 2018 7:56 PM

Recurrent Neural Networks are the state of the art algorithm for sequential data and among others used by Apples Siri and Googles Voice Search. This is because it is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for Machine Learning problems that involve sequential data. It is one of the algorithms behind the scenes of the amazing achievements of Deep Learning in the past few years. In this post, you will learn the basic concepts of how Recurrent Neural Networks work, what the biggest issues are and how to solve them.

Table of Contents:

- Introduction
- How it works (Feed-Forward Neural Networks, Recurrent Neural Networks)
- Backpropagation Through Time
- Two issues of standard RNN's (Exploding Gradients, Vanishing Gradients)
- Long-Short Term Memory
- Summary

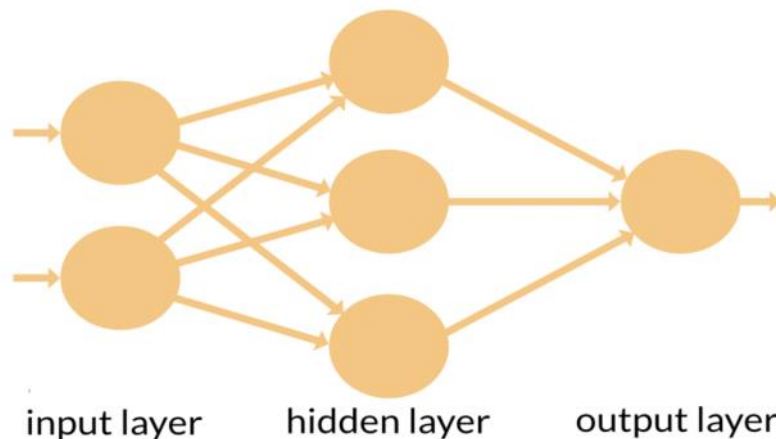
How they work

Friday, September 14, 2018 8:09 PM

We will first discuss some important facts about the „normal“ Feed Forward Neural Networks, that you need to know, to understand Recurrent Neural Networks properly.

But it is also important that you understand what sequential data is. It basically is just ordered data, where related things follow each other. Examples are financial data or the DNA sequence. The most popular type of sequential data is perhaps Time series data, which is just a series of data points that are listed in time order.

Feed-Forward Neural Networks



RNN's and Feed-Forward Neural Networks are both named after the way they channel information.

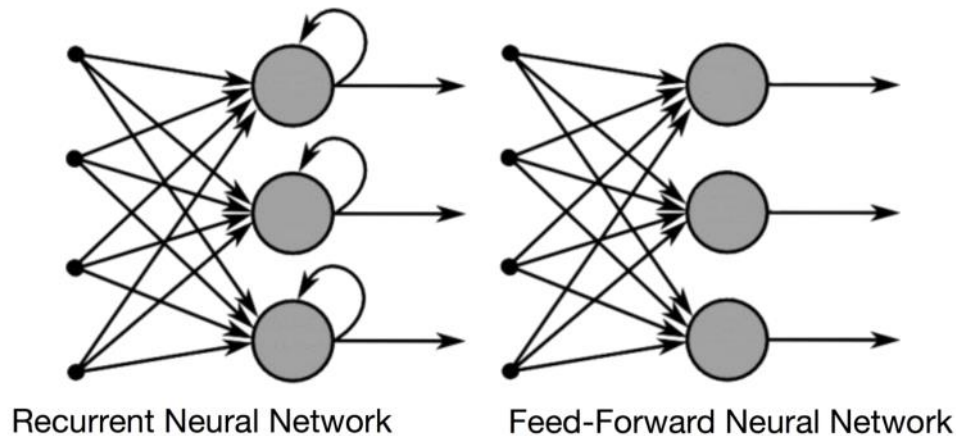
In a Feed-Forward neural network, the information only moves in one direction, from the input layer, through the hidden layers, to the output layer. The information moves straight through the network. Because of that, the information never touches a node twice.

Feed-Forward Neural Networks, have no memory of the input they received previously and are therefore bad in predicting what's coming next. Because a feedforward network only considers the current input, it has no notion of order in time. They simply can't remember anything about what happened in the past, except their training.

Recurrent Neural Networks

In a RNN, the information cycles through a loop. When it makes a decision, it takes into consideration the current input and also what it has learned from the inputs it received previously.

The two images below illustrate the difference in the information flow between a RNN and a Feed-Forward Neural Network.



A usual RNN has a short-term memory. In combination with a LSTM they also have a long-term memory

Deep Understanding of the internal flow

Imagine you have a normal feed-forward neural network and give it the word „neuron“ as an input and it processes the word character by character. At the time it reaches the character „r“, it has already forgotten about „n“, „e“ and „u“, which makes it almost impossible for this type of neural network to predict what character would come next.

A Recurrent Neural Network is able to remember exactly that, because of it's internal memory. It produces output, copies that output and loops it back into the network.

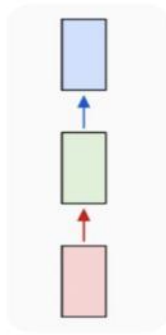
Recurrent Neural Networks add the immediate past to the present.

Therefore a Recurrent Neural Network has two inputs, the present and the recent past. This is important because the sequence of data contains crucial information about what is coming next, which is why a RNN can do things other algorithms can't.

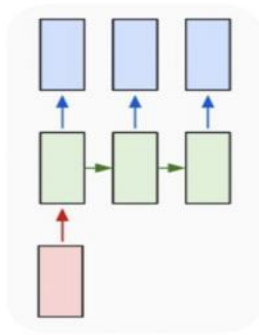
A Feed-Forward Neural Network assigns, like all other Deep Learning algorithms, a weight matrix to its inputs and then produces the output. Note that RNN's apply weights to the current and also to the previous input. Furthermore they also tweak their weights for both through gradient descent and Backpropagation Through Time

Also note that while Feed-Forward Neural Networks map one input to one output, RNN's can map one to many, many to many (translation) and many to one (classifying a voice).

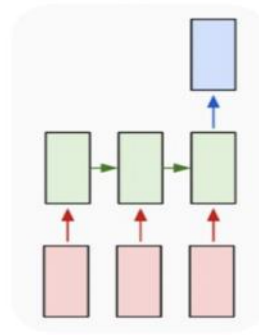
one to one



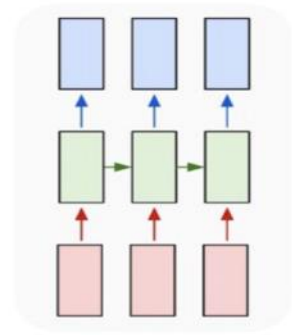
one to many



many to one



many to many



Backpropagation Through Time

Friday, September 14, 2018 8:25 PM

To understand the concept of Backpropagation Through Time you definitely have to understand the concepts of Forward and Back-Propagation first.

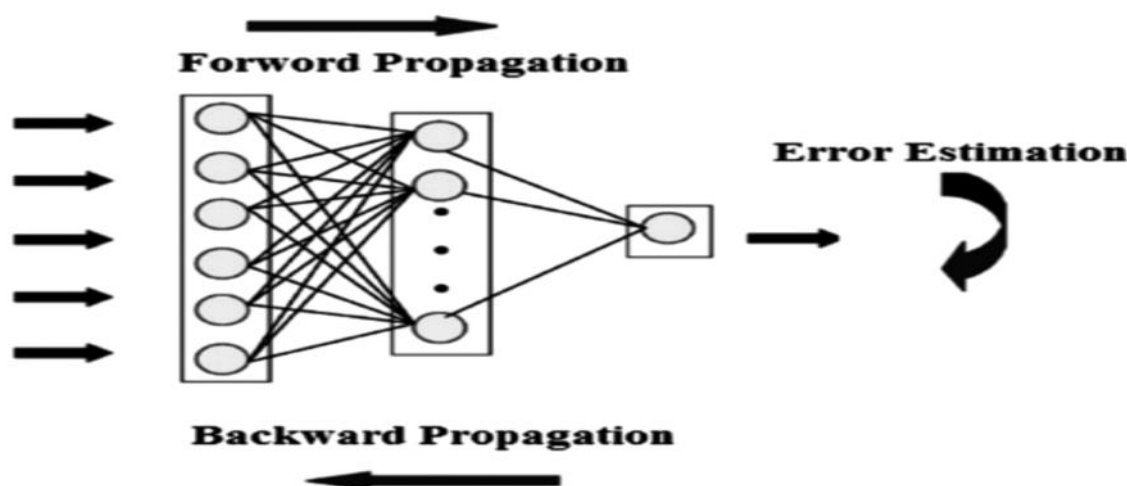
In neural networks, you basically do Forward-Propagation to get the output of your model and check if this output is correct or incorrect, to get the error.

Now you do Backward-Propagation, which is nothing but going backwards through your neural network to find the partial derivatives of the error with respect to the weights, which enables you to subtract this value from the weights.

Those derivatives are then used by Gradient Descent, an algorithm that is used to iteratively minimize a given function. Then it adjusts the weights up or down, depending on which decreases the error. That is exactly how a Neural Network learns during the training process.

So with Backpropagation you basically try to tweak the weights of your model, while training.

The image below illustrates the concept of Forward Propagation and Backward Propagation perfectly at the example of a Feed Forward Neural Network:

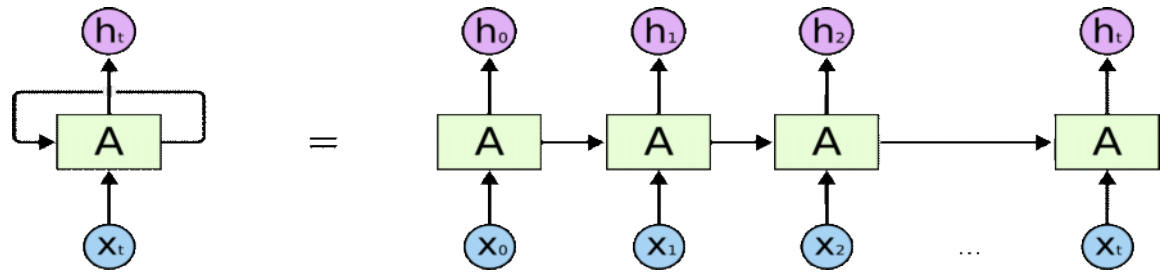


Backpropagation Through Time (BPTT) is basically just a fancy buzz word for doing Backpropagation on an unrolled Recurrent Neural Network. Unrolling is a visualization and conceptual tool, which helps you to understand what's going on within the network. Most of the time when you implement a Recurrent Neural Network in the common programming frameworks, they automatically take care of the Backpropagation but you need to understand how it works, which enables you to troubleshoot problems that come up during the development process.

You can view a RNN as a sequence of Neural Networks that you train one after another with backpropagation.

The image below illustrates an unrolled RNN. On the left, you can see the RNN, which is unrolled after the equal sign. Note that there is no cycle after the equal sign since the different timesteps are visualized and information gets passed from one timestep to the next. This illustration also shows why a RNN can be seen as

a sequence of Neural Networks.



If you do Backpropagation Through Time, it is required to do the conceptualization of unrolling, since the error of a given timestep depends on the previous timestep.

Note:

Within BPTT the error is back-propagated from the last to the first timestep, while unrolling all the timesteps. This allows calculating the error for each timestep, which allows updating the weights. Note that BPTT can be computationally expensive when you have a high number of timesteps.

Two issues of standard RNN's (Exploding Gradients, Vanishing Gradients)

Friday, September 14, 2018 8:36 PM

There are two major obstacles RNN's have or had to deal with. But to understand them, you first need to know what a gradient is.

A gradient is a partial derivative with respect to its inputs. If you don't know what that means, just think of it like this: A gradient measures how much the output of a function changes, if you change the inputs a little bit.

You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops to learning. A gradient simply measures the change in all weights with regard to the change in error.

Exploding Gradients

We speak of „Exploding Gradients“ when the algorithm assigns a stupidly high importance to the weights, without much reason. But fortunately, this problem can be easily solved if you truncate or squash the gradients.

Vanishing Gradients

We speak of „Vanishing Gradients“ when the values of a gradient are too small and the model stops learning or takes way too long because of that. This was a major problem in the 1990s and much harder to solve than the exploding gradients. Fortunately, it was solved through the concept of **LSTM** by Sepp Hochreiter and Juergen Schmidhuber

Long-Short Term Memory

Friday, September 14, 2018 8:40 PM

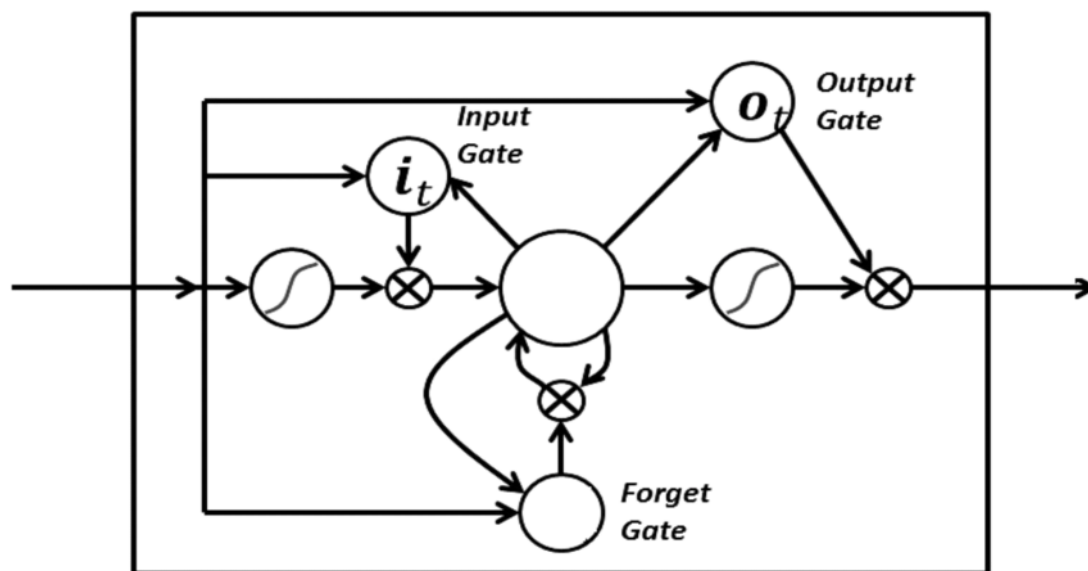
Long Short-Term Memory (LSTM) networks are an extension for recurrent neural networks, which basically extends their memory. Therefore it is well suited to learn from important experiences that have very long time lags in between.

The units of an LSTM are used as building units for the layers of a RNN, which is then often called an LSTM network.

LSTM's enable RNN's to remember their inputs over a long period of time. This is because LSTM's contain their information in a memory, that is much like the memory of a computer because the LSTM can read, write and delete information from its memory.

This memory can be seen as a gated cell, where gated means that the cell decides whether or not to store or delete information (e.g if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm. This simply means that it learns over time which information is important and which not.

In an LSTM you have three gates: input, forget and output gate. These gates determine whether or not to let new input in (input gate), delete the information because it isn't important (forget gate) or to let it impact the output at the current time step (output gate). You can see an illustration of a RNN with its three gates below:



The gates in a LSTM are analog, in the form of sigmoids, meaning that they range from 0 to 1. The fact that they are analog, enables them to do backpropagation with it.

For Multiclass classification I can use SoftMax

The problematic issues of vanishing gradients is solved through LSTM because it keeps the gradients steep enough and therefore the training relatively short and the accuracy high.

*** Neural networks: training with backpropagation

Saturday, September 15, 2018 11:34 PM

<https://www.jeremyjordan.me/neural-networks-training/> (VVI , you can refer for more math , how neural network works)

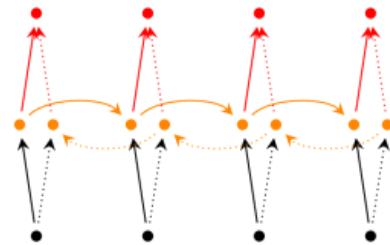
RNN Extensions

Friday, September 14, 2018 8:49 PM

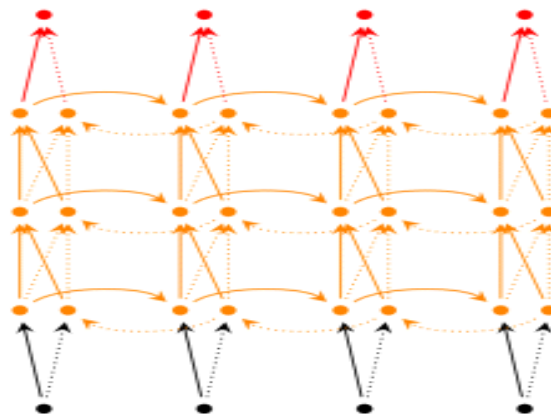
Over the years researchers have developed more sophisticated types of RNNs to deal with some of the shortcomings of the vanilla RNN model. We will cover them in more detail in a later post, but I want this section to serve as a brief overview so that you are familiar with the taxonomy of models.

Bidirectional RNNs are based on the idea that the output at time t

may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs.



Deep (Bidirectional) RNNs are similar to Bidirectional RNNs, only that we now have multiple layers per time step. In practice this gives us a higher learning capacity (but we also need a lot of training data).



LSTM networks are quite popular these days and we briefly talked about them above. LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. The memory in LSTMs are called *cells* and you can think of them as black boxes that take as input the previous state h_{t-1} and current input x_t .

Save Your Neural Network Model to JSON(PROD Release)

Saturday, September 15, 2018 11:52 PM

Install: `sudo pip install h5py`

```
# MLP for Pima Indians Dataset Serialize to JSON and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_json
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, kernel_initializer='uniform', activation='relu'))
model.add(Dense(8, kernel_initializer='uniform', activation='relu'))
model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")
# later...
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

Summary

Friday, September 14, 2018 8:45 PM

Now you have proper understanding of how a Recurrent Neural Network works, which enables you to decide if it is the right algorithm to use for a given Machine Learning problem.

Specifically, you learned what's the difference between a Feed-Forward Neural Network and a RNN, when you should use a Recurrent Neural Network, how Backpropagation and Backpropagation Through Time works, what the main issues of a RNN are and how a LSTM works.