Machine Learning for Finance Applications - Introduction to Deep Learning

Brian Clark

2020

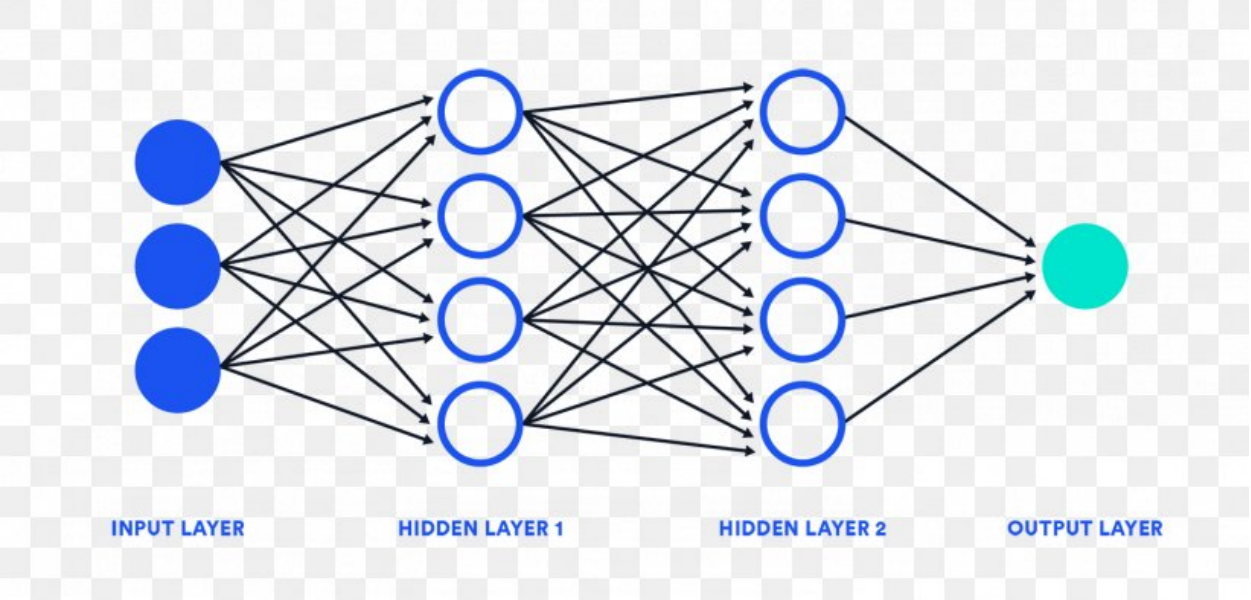
Contents

1 Introduction

This document provides a basic introduction to deep learning. It starts with a descritpion of a the basic unit of neural networks - the perceptron. It next discusses artificial neural nets (ANNs) in the context of a multi-layer perceptron. Finally, more advanced topics such as Recurrent Neural Nets (RNNs) are disucssed including the populat class of Long Short Term Memory (LSTM) neural nets.

In discussion of the basic neural net structures, the document will also cover how to train neural nets with a description of the commonly applied back propogation algorithm. We will also discuss some common methods for implementing back propogation including stochastic gradient descent (SGD) and adaptive gradient descent methods. Practical issues such as overfitting considerations and paramter selection will also be discussed.

Deep learning refers to a neural network that has multiple hidden layers. The below diagram is a visual depiction of a basic ANN. It consists of an input layer (X), two hidden layers, and output layer (Y). The idea behind deep learning is combine many layers of neurons (the dots) with simple linear functions to form a highly complex and non-linear model. The term "deep" refers to the number of hidden layers.

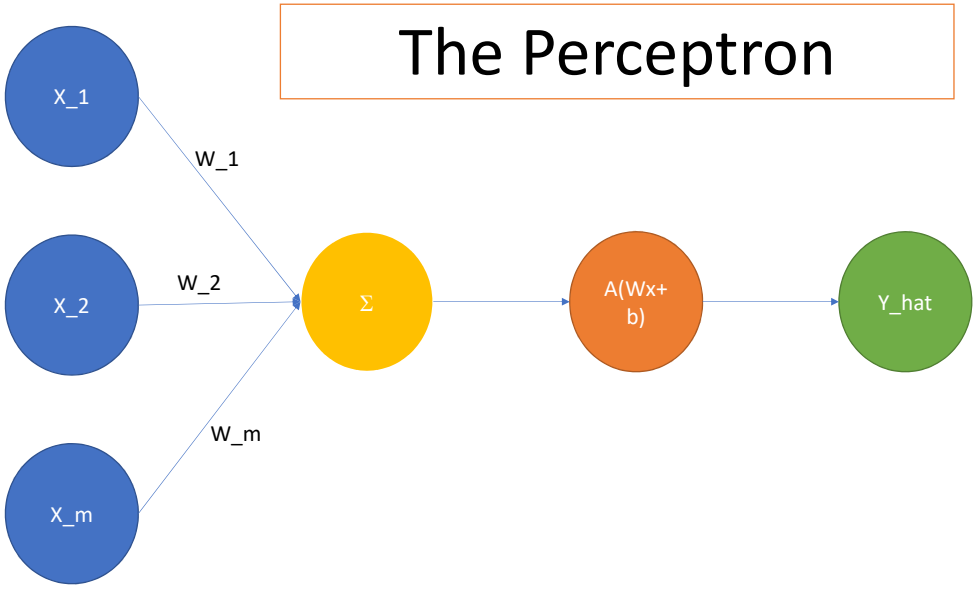


[1]

[1] Source for the figure: https://favpng.com/png_view/brain-artificial-neural-network-deep-learning-convolutional-neural-network-neuron-machine nnS6ikN0.

2 From the Perceptron to a Neural Network

The idea of a perceptron has been around since the 1950's. A perceptron forms the basis for today's complex neural networks. It can be thought of as a single neuron.

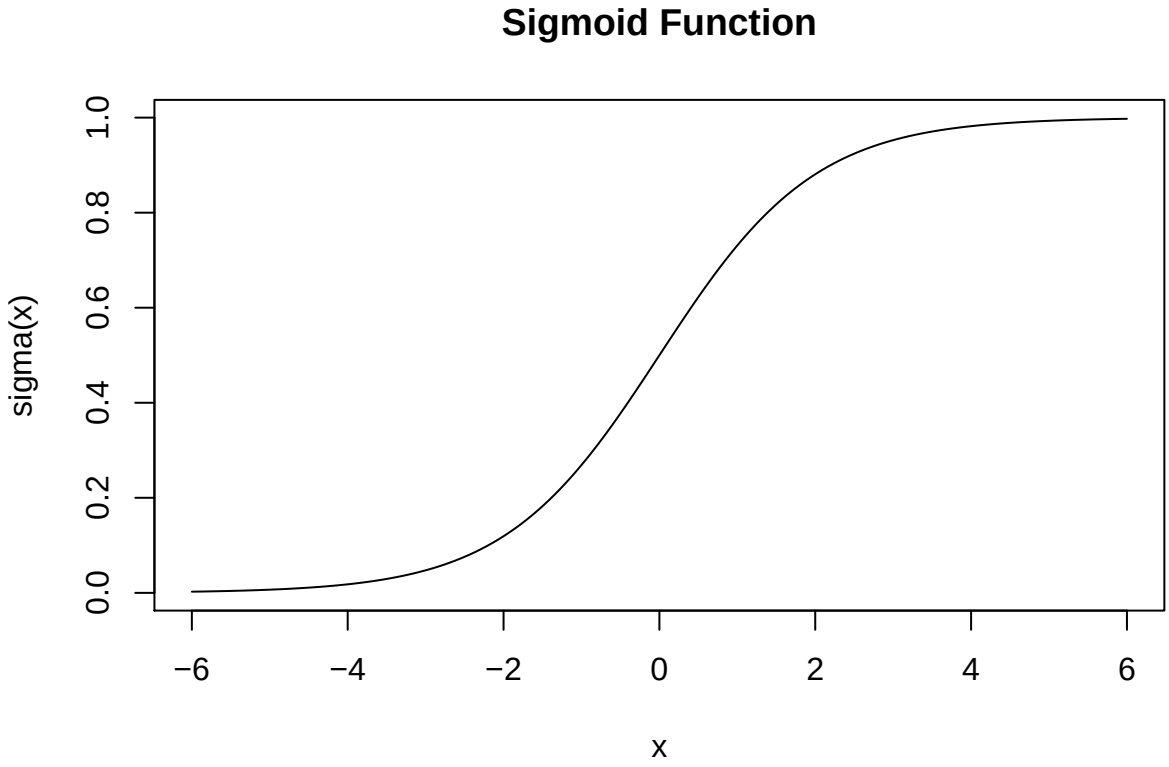


There are several components to the perceptron:

- Input layer: Consists of your feature space which we can express as a matrix X of dimension $n \times m$, where n is the number of observations and m is the number of features.
- Weights: There is a single weight on each connection. For the case of the simple perceptron, the weights can be expressed as a vector $W = [w_1, w_2, ..., w_m]'$.
- Sum function: The neuron aggregates the inputs as a linear combination $\sum_{i=1}^{m} x_i w_i$.
- Non-linearity: Non-linearity is introduced using a activation function, $g(W, X)$. In this example we will consider the sigmoid function, $\sigma(x) = (1 + e^{-x})^{-1}$. The sigmoid function is the basis for the logistic regression.
- The output layer: In this case, we will consider a binary outcome $y = g(w_0 + \sum_{i=1}^{m} x_i w_i)$. Note that the term w_0 was introduced and is commonly referred to as the bias. It serves the same purpose as a constant term in a linear regression.

A characteristic of neural nets is to have a non-linear activation function. Again, we are considering the sigmoid function which is as follows:

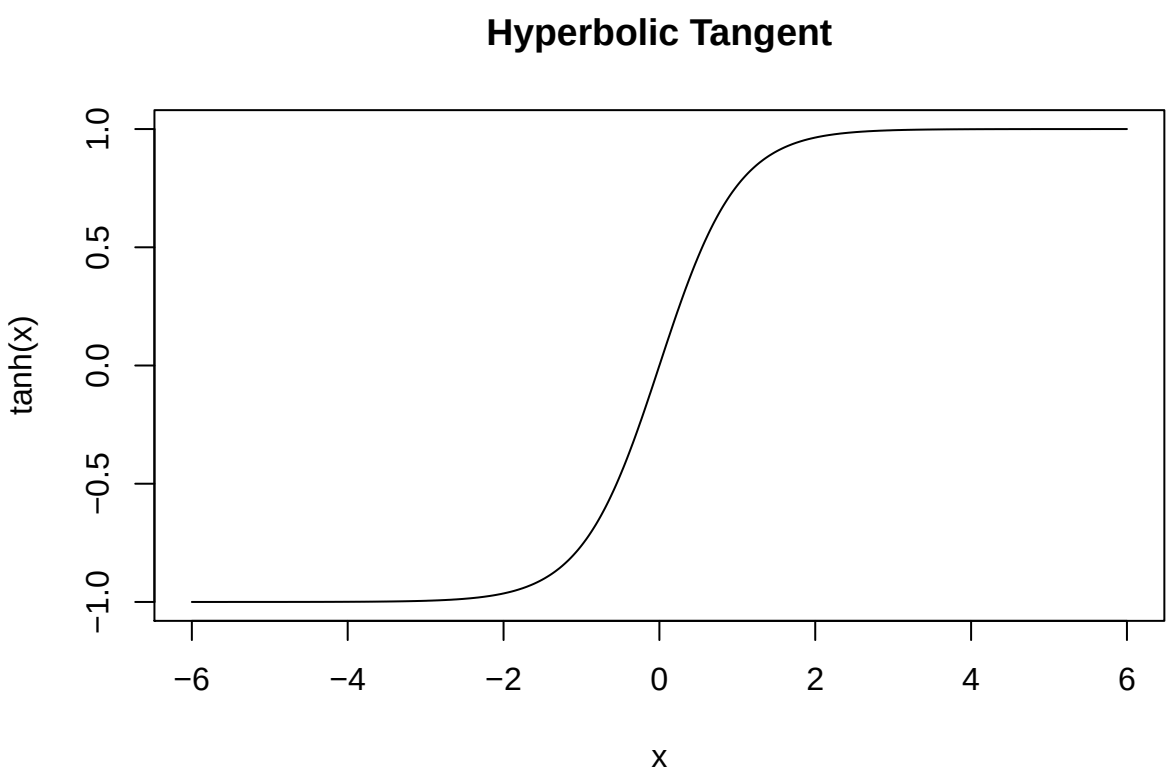
```r
rm(list=ls())
x <- seq(-6,6,length=1000)
sigmoid <- function(x){
  return(1/(1+exp(-x)))
}
y <- sigmoid(x)

plot(x=x,y=y,main="Sigmoid Function",
     xlab = "x", ylab="sigma(x)",
     type="l")
```



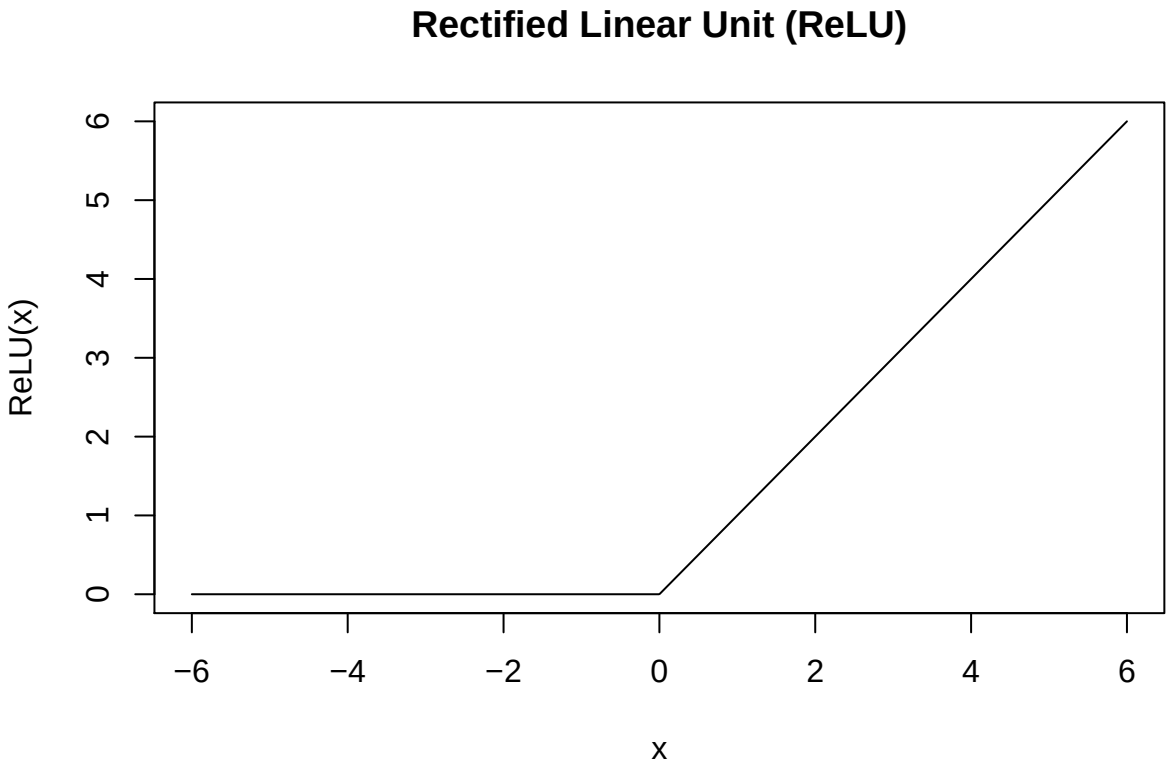
Notice how the sigmoid function is constrained to be between zero and one so it can represent a probablility. Hence, it is commonly used for classificaiton problems. Also, note that the probability is equal to 0.5 at $x = 0$. There are many other examples of activation functions (in place of the sigmoid function). Other include the $tanh(x)$ and the rectified linear unit or $ReLU$ functions.

```r
my.tanh <- function(x){
  return((exp(x) - exp(-x))/(exp(x) + exp(-x)))
}

my.relu <- function(x){
  return(pmax(x,0))
}

f.tanh <- my.tanh(x)
f.relu <- my.relu(x)
```

```
plot(x=x,y=f.tanh,main="Hyperbolic Tangent",
     xlab = "x", ylab="tanh(x)",
     type="l")
```




```
plot(x=x,y=f.relu,main="Rectified Linear Unit (ReLU)",
     xlab = "x", ylab="ReLU(x)",
     type="l")
```



Each of the above activation functions also has a derivative which is easy to compute:

$$\sigma(x)' = \sigma(x)(1 - \sigma(x)) \tag{1}$$

$$tanh(x)' = 1 - tanh(x)^2 \tag{2}$$

$$ReLU(x)' = 1, \text{ if } x > 0 \text{ and } 0 \text{ otherwise} \tag{3}$$

To solve the model, we first consider a first pass which is called forward propogation. The activation functions are important because they introduce non-linearity into the data.

We can write the equation of a perceptron as follows:

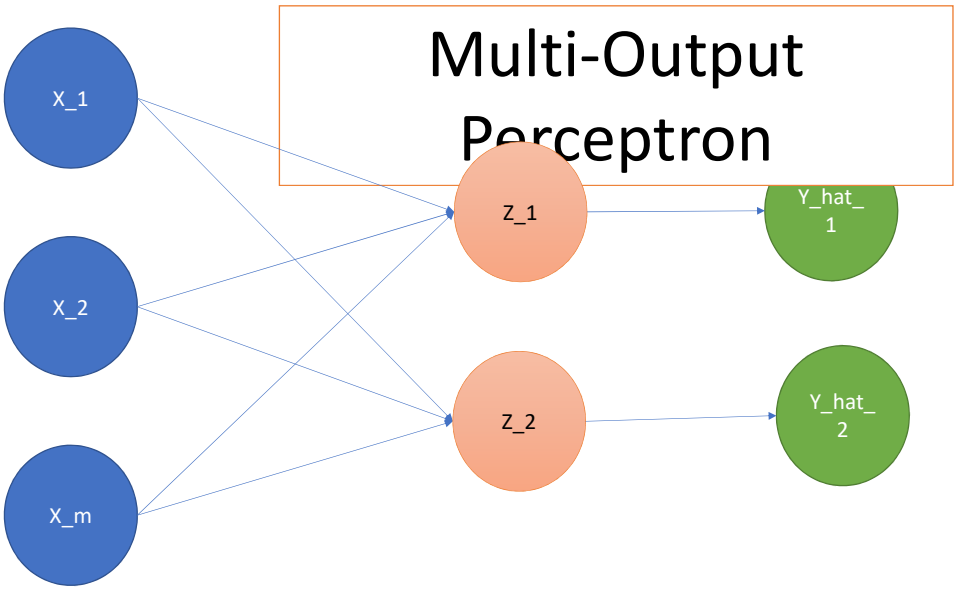
$$\hat{y} = g(w_0 + X'W), \tag{4}$$

or if we let $Z = X'W$,

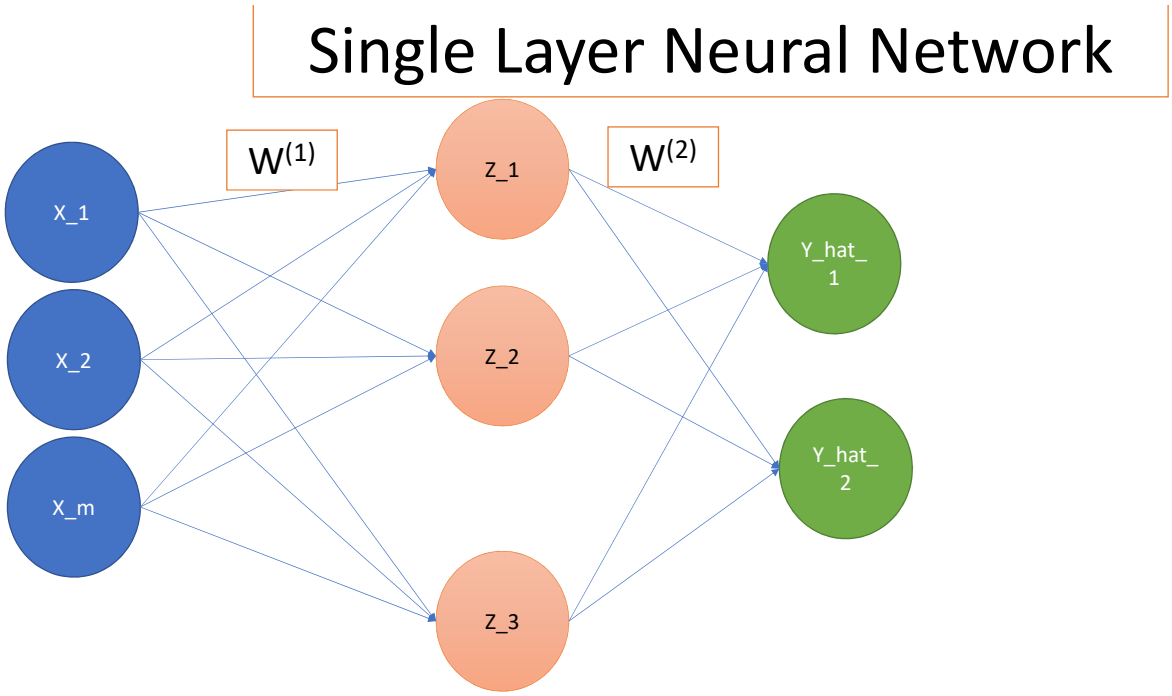
$$\hat{y} = g(w_0 + Z), \tag{5}$$

Thus, for any set of weights and inputs, we can obtain a prediction of our outcome. The question then becomes how do we find the weights to make the best prediction possible?

We can extend the single output perceptron for multiple otputs as well by simply adding an additional output.



Note that $z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$ and $\hat{y}_i = g(z_i)$ where again $g(z_i)$ is the activation function. We can now extend this to a single hidden layer neural network by adding a hidden layer.



We now have two weight matrices, $W_{(1)}$ and $W_{(2)}$. The hidden layer is defined as:

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}, \tag{6}$$

and the final output is defined as

$$\hat{y}_i = g(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} x_j w_{j,i}^{(2)}, \tag{7}$$

Again, note that each node in layer $L-1$ connects to each node in layer L. This is known as a dense connection. As you can see, the number of parameters (all the $w's$) needed to fit the model is growingvery quickly.

We the new layer is referred to as hidden because it is not directly observable. In the preceding perceptron example, the layers were all observable and in fact it was actually an example of a logistic or multinomial logistic regression. The idea behind deep learning is to increase the number of hidden layers. This increases the number of parameters and hence the flexibility of the model. For each hidden layer k, we can generalize the above formula to

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}. \tag{8}$$

3 Applying Neural Networks

In this section, we now address the key problem - **How do we train this model to make forecasts?**.

In essence, the solution to the problem is an extension of the regression problems we have already considered. The only real difference is in how we have to solve the problem to make it tractable as the number of parameters increases greatly.

Just as we saw for the other ML models, we start with a loss function. The idea of a loss function is to quanitfy the prediciton error for each training observation, hence we can define it for a binary output as follows

$$L(f(x^{(i)}; W), y_{(i)}) = y^{(i)} log(f(x^{(i)}; W)) + (1 - y^{(i)}) log(1 - f(x^{(i)}; W)), \tag{9}$$

where $y^{(i)}$ is the actual outcome and $f(x^{(i)}; W)$ is the model predicted value, $\hat{y}^{(i)}$. This loss function is known as the soft max cross entropy loss. If we were doing a regression problem, the loss function might be based on the $l_2 - norm$ as follows:

$$L(f(x^{(i)}; W), y_{(i)}) = (y^{(i)} - f(x^{(i)}; W)))^2. \tag{10}$$

Again, the latter is called the MSE loss and is useful for continuous outcome variables (i.e., regression problems). To solve the model, we want to find the set of weights that minimize the total loss across all $i = 1, ..., n$ observations. That is, we want a cost function that we can define as:

$$C(W) = \frac{1}{n} \sum_{i=1}^{n} L(f(x^{(i)}; W), y_{(i)}). \tag{11}$$

As such, the objective function becomes:

$$\mathbf{W}^* = \underset{\mathbf{W}}{\text{argmax}} \frac{1}{n} \sum_{i=1}^{n} L(f(x^{(i)}; W), y_{(i)}), \tag{12}$$

or simply

$$\mathbf{W}^* = \underset{\mathbf{W}}{\text{argmax}}\, C(\mathbf{W}). \tag{13}$$

Note that $\mathbf{W} = [\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, ..., \mathbf{W}^{(L)}]$, where L is the number of layers in the network and each individual $\mathbf{W}^{(i)}$ is a matrix of $m^{(L-1)} \times m^{(L)}$ dimensions. Thereofre, this problem can be thought of as an opitmization problem. We will consider solving the problem using a gradient descent.

3.1 Gradient Descent

This section reviews gradient descent methods for general multi-dimension optimization problems. The general problem is as follows:

$$\min_{\mathbf{W}} \mathbf{C}(\mathbf{W}) \tag{14}$$

Note that we will always state the problem as a minimization problem. The solution $\mathbf{W}^*$ should satisfy the following:

$$f(\mathbf{W}^*) \leq f(\mathbf{W}) \tag{15}$$

where, $\mathbf{W}^*$ is said to be the global optimimum.

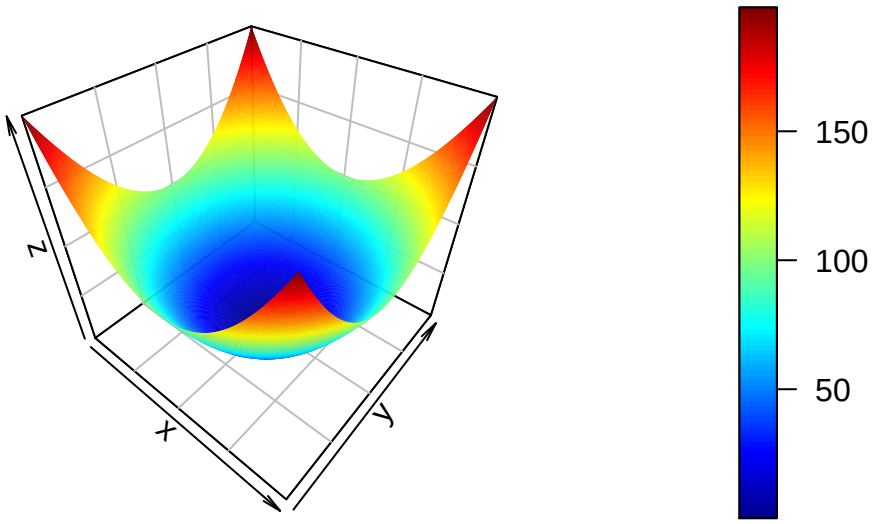
3.1.1 Example 1

The first example is to find the minimum of the following function:

$$F(w_1, w_2) = w_1^2 + w_2^2 \tag{16}$$

The following block of code plots the function, which is essentially a bowl shape:

```
# Clear memory:
rm(list=ls())

# Set Working Directory:
setwd("C:/Users/CLARKB2/Documents/Classes/2018-Fall/Financial Computation/R-vignettes")
if (!require("plotly")) install.packages("plotly")
if (!require("plot3D")) install.packages("plot3D")
library("plotly")
library("plot3D")

N <- 201

Fxy <- function(x,y){
  fx <- x^2 + y^2
  return(fx)
}


xmat <- matrix(seq(-10,10,length=N), nrow = N, ncol = N, byrow=FALSE)
ymat <- matrix(seq(-10,10,length=N), nrow = N, ncol = N, byrow=TRUE)
zmat <- Fxy(xmat,ymat)

# Interactive Plot:
p <- plot_ly(x=xmat,y=ymat,z=zmat) %>% add_surface() %>%
  layout(
    title = "3D Bowl Plot",
    scene = list(
      xaxis = list(title = "w1"),
      yaxis = list(title = "w2"),
      zaxis = list(title = "C(W)")
    ))

print(p)

surf3D(x=xmat,y=ymat,z=zmat,
       bty="b2",
       main="Bowl Plot C(W) = w1^2 + w2^2")
```



Note that the goal here is to find the mimimum of the above surface, which will be $W^* = [0, 0]$. The solution is quite simple in this example as there is only one local minima that is also the global minimum. As such, you will always find the true value of W^*, regardless of your starting values.

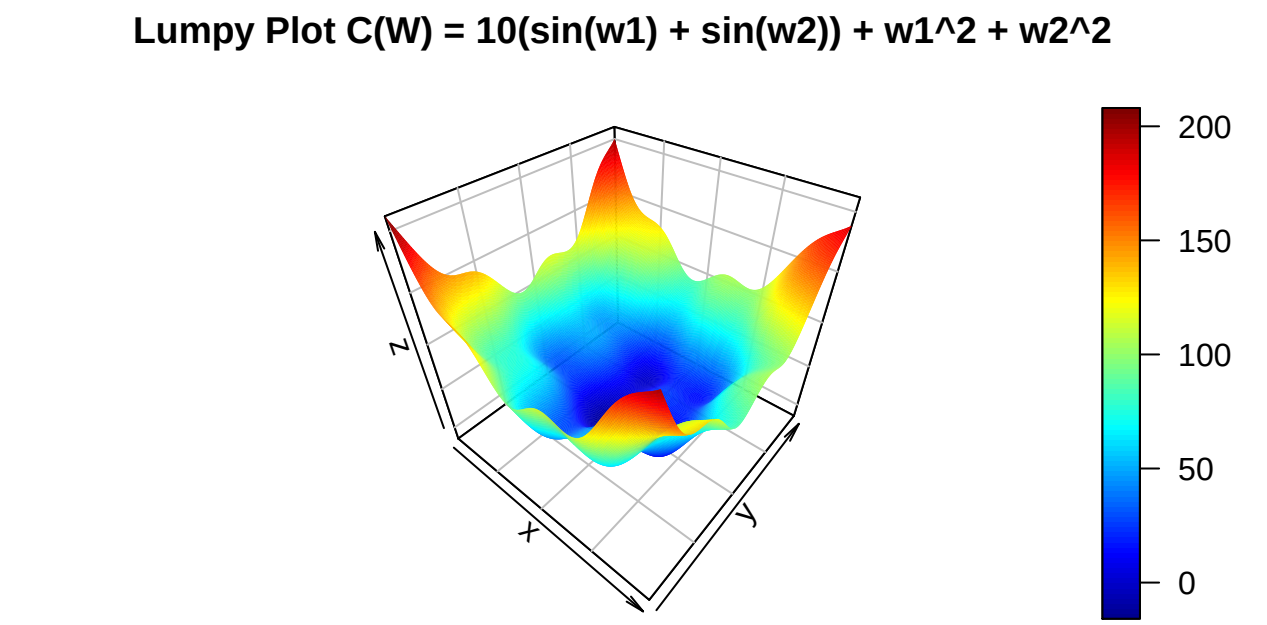
3.1.2 Example 2

Now consider the following equation:

$$C(W) = 10(sin(w_1) + sin(w_2)) + w_1^2 + w_2^2 \tag{17}$$

```
N <- 201

Fxy <- function(x,y){
  fx <- 10*(sin(x) + sin(y)) + x^2 + y^2
  return(fx)
}

xmat <- matrix(seq(-10,10,length=N), nrow = N, ncol = N, byrow=FALSE)
ymat <- matrix(seq(-10,10,length=N), nrow = N, ncol = N, byrow=TRUE)
zmat <- Fxy(xmat,ymat)

# Interactive Plot:
p <- plot_ly(x=xmat,y=ymat,z=zmat) %>% add_surface() %>%
  layout(
    title = "3D Lumpy Plot",
    scene = list(
      xaxis = list(title = "w1"),
```

```
      yaxis = list(title = "w2"),
      zaxis = list(title = "C(W)")
    ))

print(p)

surf3D(x=xmat,y=ymat,z=zmat,
       bty="b2",
       main="Lumpy Plot C(W) = 10(sin(w1) + sin(w2)) + w1^2 + w2^2")
```



Now, in this example, there are several local minima that you may find even though there is only one gloabl minimum ($W^* = [0, 0]$).

In principal, we can solve an unconstained optimization problem by finding a stationary point such that $\nabla C(\mathbf{W}^*) = \mathbf{0}$. Computation approaches to arrive at $\mathbf{W}^*$ are generally based on the generation of a sequence of points $\mathbf{W^{(k)}}$ that converge to $\mathbf{W}^*$.[2]

The algorithms start at some initial point, $\mathbf{W^{(0)}}$ and search "downhill." In particular, for each $\mathbf{W^{(k)}}$, we want to move in a search direction, $\mathbf{W^{(k)}}$, such that:

$$C(\mathbf{W^{(k)}} + \delta \mathbf{s^{(k)}}) < C(\mathbf{W^{(k)}}) \tag{18}$$

for some $\delta > 0$.

Consider the function $h(\delta) = C(\mathbf{W} + \delta \mathbf{s})$ which is a descent direction characterized by the following:

[2]Note that the index k in $\mathbf{W^{(k)}}$ refers to the k^{th} iteration of the optimization routine. That is, each $\mathbf{W^{(k)}} = [\mathbf{W}^{(k)(0)}, \mathbf{W}^{(k)(1)}, ..., \mathbf{W}^{(k)(L)}]$.

$$\frac{dh}{d\delta} = [\nabla C(\mathbf{W})]' s < 0 \tag{19}$$

which is to say that the slope is negative, i.e., we are moving toward the optimim (or at least "downhill").

A general algorithm for finding the minimum is then:

1. Find the descent direction $s^{(k)}$
2. Find a step length $\delta^{(k)}$
3. Update $\mathbf{W^{(k+1)}} = \mathbf{W^{(k)}} + \delta^{(k)}\mathbf{s^{(k)}}$
   Repeat steps (1-3) until some convergence criterion is met and return weights $\mathbf{W}$.

In a ML context, tha above process is commonly referred to as gradient descent. Note that δ is also referred to as the **learning rate** and choosing it is not a trivial process. Choosing δ too small or large can cause problems:

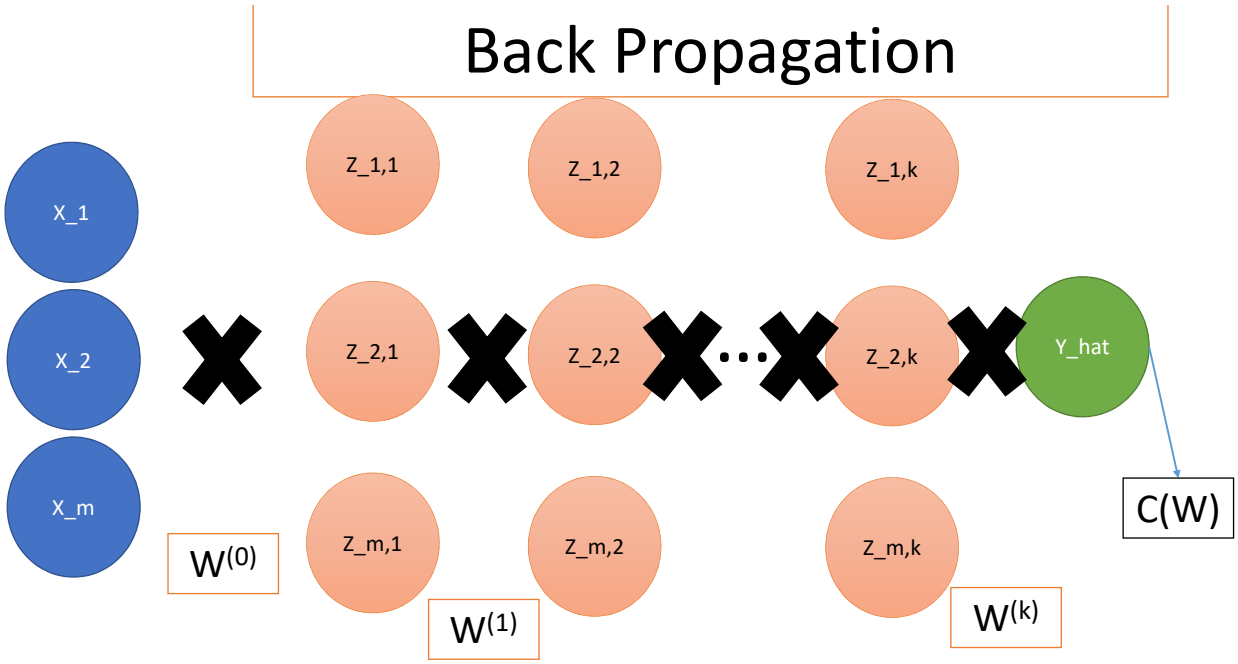
1. Small δ's slow the computation time because many steps are required.
2. Small δ's also increase the likelihood of getting caught in a local minimum.
3. Large δ's increase the likelihood that you will miss the minimum and potentially diverge from the optimal solution.

3.2 Back Propagation

The basic priciples in the above examples hold in a deep learning context. However, one difference from a method such as the steepest descent method commonly taught in a numerical methods course to solve an unconstrained convex optimization problem is that computing the gradient $\nabla C(\mathbf{W})$ is not straight-forward.

The reason is that the gradient of $C(\mathbf{W})$ is a vector of the partial derivaitves of $C(\mathbf{W})$ with respect to every individual weight $w_{i,k}$.

Let's consider a deep neural network depicted below. Note that the thick black X's mean that there are dense connections between each layer (i.e., every node in layer $L-1$ is connected to every node in layer L).



Again, the problem we want to solve is the following:

$$\mathbf{W}^* = \underset{\mathbf{W}}{\text{argmax}}\ C(\mathbf{W}).$$

(We want to choose the weights to optimize model performance.) In order to solve this problem, we conduct a gradient descent method. To understand the back propagation algorithm, it's probably best to start with an example.

Starting at the righ hand side of the plot, define the gradient of cost function with respect to the weights $\mathbf{W^{(k)}}$. For simplicity, let's assume $m = 3$ and ignore the bias term. Note tht m could vary across the network but we will assume that it does not. The gradient is defined as

$$\nabla \mathbf{C}(\mathbf{W}) = \begin{bmatrix} \left(\frac{\partial C}{\partial w_{1,1}^{(k)}}\right) \\ \left(\frac{\partial C}{\partial w_{2,1}^{(k)}}\right) \\ \left(\frac{\partial C}{\partial w_{3,1}^{(k)}}\right) \end{bmatrix}$$

Tto compute the individual components of the above gradient, notice that we need to use the chain rule because we want to gauge the impact of a small change in $w_{i,j}^{(k)}$ on the cost function $C(\mathbf{W})$, which of course depends on an intermediate term $\hat{y}$. Take the first term in the gradient. Using the chain rule, we can write

$$\nabla \mathbf{C}(\mathbf{W}) = \begin{bmatrix} \left(\frac{\partial C}{\partial w_{1,1}^{(k)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{1,1}^{(k)}}\right) \\ \left(\frac{\partial C}{\partial w_{2,1}^{(k)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{2,1}^{(k)}}\right) \\ \left(\frac{\partial C}{\partial w_{2,1}^{(k)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{3,1}^{(k)}}\right) \end{bmatrix}$$

Then, moving back one layer, we can see that the partial derivatives on layer $k-1$ need to be expanded further using the chain rule because the impact of changing any $w_{i,j}^{(k-1)}$ on $C(\mathbf{W})$ depends on the weights in layer k and $\hat{y}$. Also, note that we now have 9 partial derivatives:

$$\nabla \mathbf{C}(\mathbf{W^{(k-1)}}) = \begin{bmatrix} \left(\frac{\partial C}{\partial w_{1,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{1,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{1,3}^{(k-1)}}\right) \\ \left(\frac{\partial C}{\partial w_{2,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{2,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{2,3}^{(k-1)}}\right) \\ \left(\frac{\partial C}{\partial w_{3,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{3,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial w_{3,3}^{(k-1)}}\right) \end{bmatrix}$$

Take the upper left term, $\left(\frac{\partial C}{\partial w_{1,1}^{(k-1)}}\right)$, we can expand it using the chain rule:

$$\left(\frac{\partial C}{\partial w_{1,1}^{(k-1)}}\right) = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial w_{1,1}^{(k-1)}}$$

Expanding throughout the matrix yields:

$$\nabla \mathbf{C}(\mathbf{W^{(k-1)}}) = \begin{bmatrix} \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial w_{1,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2^{(k)}} \frac{\partial z_2^{(k)}}{\partial w_{1,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3^{(k)}} \frac{\partial z_3^{(k)}}{\partial w_{1,3}^{(k-1)}}\right) \\ \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial w_{2,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2^{(k)}} \frac{\partial z_2^{(k)}}{\partial w_{2,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3^{(k)}} \frac{\partial z_3^{(k)}}{\partial w_{2,3}^{(k-1)}}\right) \\ \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial w_{3,1}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2^{(k)}} \frac{\partial z_2^{(k)}}{\partial w_{3,2}^{(k-1)}}\right) & \left(\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3^{(k)}} \frac{\partial z_3^{(k)}}{\partial w_{3,3}^{(k-1)}}\right) \end{bmatrix}$$

Notice that there is a lot of duplicate values computed in the matrix $\nabla \mathbf{C}(\mathbf{W^{(k-1)}})$ that have already been computed in matrix $\nabla \mathbf{C}(\mathbf{W^{(k)}})$. In particular, the partial derivatives of $\frac{\partial C}{\partial \hat{y}}$ is already computed. Also, the derivatives with respect to state $z_i^{(k)}$ will be needed in the next iteration back (i.e., at layer $(k-2)$). This process will continue until the partial derivatives w.r.t. each of the weights is computed.

Without going into great detail, the idea behind back propagation is to use the above logic to reduce redundency in the compuattion of the overall gradient $C(\mathbf{W})$ across the entire network. The benefit of the algorthim is that it avoid unnecesary duplicate calculations and essentially breaks up the differential into parts that can be computed faster.

3.3 Training the Model in Practice

Any standard deep learning software has back propagation built in so we don't need to worry about the exact algorithm at this point. However, we do still need to understand some inputs that are not necessarily trivial. A usefule link for more information is the following: https://ruder.io/optimizing-gradient-descent/.

3.3.1 Gradient Descent Varieties

In practice, the gradient is very computationally expensive to compute. As such, it is not always feasible to compute for every data point on every iteration. As such, there are a few broad classes to choose from.

3.3.1.1 Batch Gradient Descent

Batch gradient descent computes the gradient of the cost function with respect to the weights for the eniter training dataset. It is the most accurate representation of the gradient, but can be very computationally expensive.

3.3.1.2 Stochastic Gradient Descent (SGD)

SGD performs the gradient at a randomly selected point i and updates the parameters. Thus, it updates parameters very frequently. The advantage is that it avoids finding the gradient of duplicate (or very similar) observations as is done in batch (or fill) gradient descent. The disadvantage is that it is a very high variance method. Therefore, we are again trading off bias and variance.

When implementing SGD, it is not uncommon to see wild variations in the value of the cost function since we are relying on a single point to make a decision. Also, SGD generallly requires a realtively small learning rate because of the extreme variance.

3.3.1.3 Mini-Batch Gradient Descent

As you might expect, there is a middle ground. Rather than using the full data or a single point, take a small set of point and measure the gradient at each of them. Then, average the results to make a deicision as to what direction to move. The benefits are obvious - we gain some speed raltive to the full gradient descent and reduce the variation associated with the SGD algorithm.

This approach is also well suited for parallelization in that many mini-batches can be distrubted across a GPU and all computed at once. This can lead to tremendous efficieny in speed gains.

3.3.2 Learning Rate

Once we have evaluated the gradient at each iteration (i.e., set of "guesses" at $\mathbf{W}$), we need to choose a step length. That is, how big of a step do we want? Above, we discussed the basic tradeoff involved when choosing a learning rate δ. In practice, we can choose a fixed rate or an adaptive rate. Generally, the adaptive rates are more effective since they change based on the parameters in the model.

A few potential methods of choosing an adaptive learning rate are described in the above link (and summarized here):

1. Learning rates can be adjusted by annealing. That is, reduce them accrding to a predfined schedule that may be a function of the value of the cost function. The idea is to reduce the step length as you converge on the solution (Momentum, Netserlov, and others).
2. Vary the learning rate based on which parameter(s) are being updated. For example, in cases where we have missing data or high-variance data (which can be more informative), you may want to take a larger step when infrequent data enters the cost function. (Adagrad, Adadelta, and others)

The same document also describes several different algorithms to adaptively choose a learning rate.

3.4 Overfitting

Overfitting in the context of deep learning is fundamentally similar to what we have seen in other ML contexts. We want to regularize the model in the sense that we want the model to be able to generalize to new test data. Below are a few basic options for regularizng a deep learning model.

3.4.1 Drop Out

This is a very similar approach in spririt to a random forest. The method works by randomly setting some of the activations to zero at each iteration. For example, with some probability, we set the value of $z_i^{(k)}$ to zero by mulitplying it by a Bernoulli [1,0] variable:

$$z_i^{(k)*} = B[0,1]\left[w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})w_{j,i}^{(k)}.\right. \tag{20}$$

The advantage is akin to what we saw for a random forest. It decreases the reliance on overly strong variables and reduces the likelihood that the model will overfit the trainign data. Typically, we can set the probability to a value of about 50%.

3.4.2 Early Stopping

Early stopping is as intuitive as it sounds. Essentially, we want to compare the training and test error during the trianing process. The training error should always be decreasing (except for random noise in methods such as SGD) but the test error will typically have a minimum point where the bais-variance tradeoff is optimized. By tracking a test statistic, we can avoid overfitting or training our model for an overly long period of time.

4 Recurrent Neural Networks

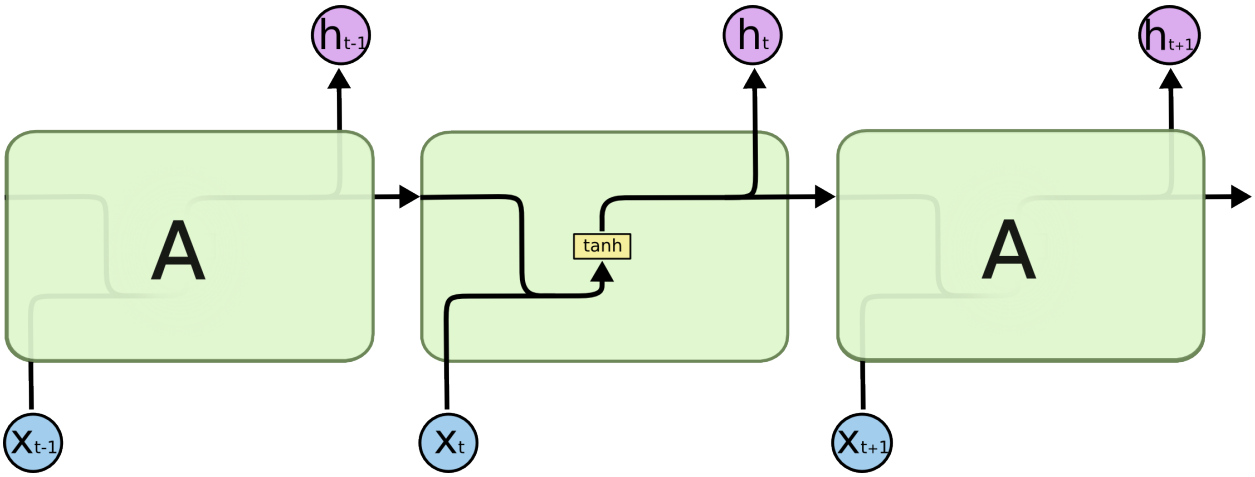
One drawback to the feed-forward class of neural netwroks discussed above is that there is not sense of time. You can see that because of the structure of the network, the timing of data is impossible to handle.

This is a major drawback for many financial applicaitons that require time-series data. FOr example, consider a model to predict stock returns. It is very plasusible that investors behave much differently in a recession than in an expansion so we should want to know what state we are in when trying to predict how agents make decisions.

One broad class of deep learning models equipped to deal with these issues are recurrnet neural networks (RNNs). They are also widely used in speech and text reading where the order of words and phrases matter when trying to predict what will come next - or in trying to understand context.

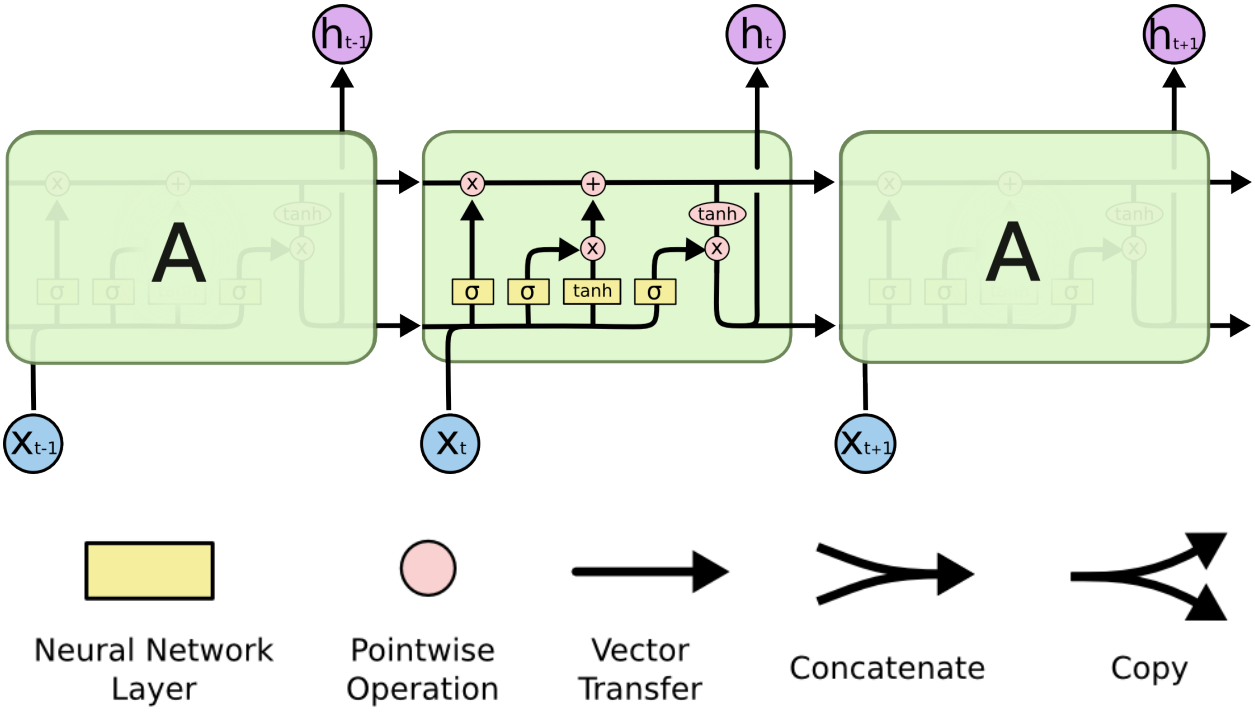
4.1 The Long-Short-Term Memory Model

The long-short term memory (LSTM) model is actually a class of models that attempt to do what the name suggests. Essentially, they are RNNs that have a memory. An excellent resource for understanding LSTM models is the following https://colah.github.io/posts/2015-08-Understanding-LSTMs/, which is where the following pictures are taken from. The first is a usual RNN structure.



In the above RNN structure you can see that new information arrives at each time period. Therefore, the predicitons preserve a memory of the past. However, one drawback is that they do not perform well in learning situations where there is a long time gap between the relevant information and the current decision.

LSTM models attempt to solve this problem by controlling how information is stored and explicitly laying out a structure to handle what information is retained and retired and how new information can enter the memory. Below is a depiction of a typical LSTM model.



[3]

The most important innovation is the top line that runs through the system. You can think of it as the "memory." The process is descibed in detail at https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[3]Images from https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

5 References