RNN

In RNN, in many applications involve temporal dependencies or dependencies over time.

What does that mean?

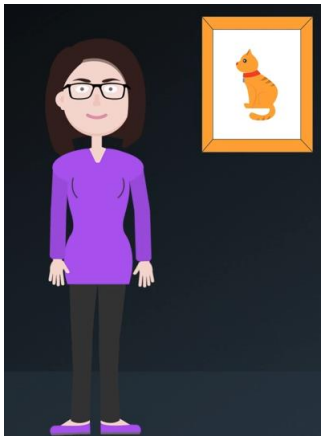
It means our current output depend not only on the current input but also on past inputs.

So, if I want to make dinner tonight, let`s see, I had pizza yesterday so maybe I should consider a salad.

Essentially, what we will have is a network that is similar to feedforward networks that we have seen before **but with addition of memory**.

You may have noticed that in the applications you have seen before only the current input matter.

For example, classifying a picture is this a cat.



But perhaps this is not a static cat.

Maybe when the picture was shot, the cat was moving From a single image.

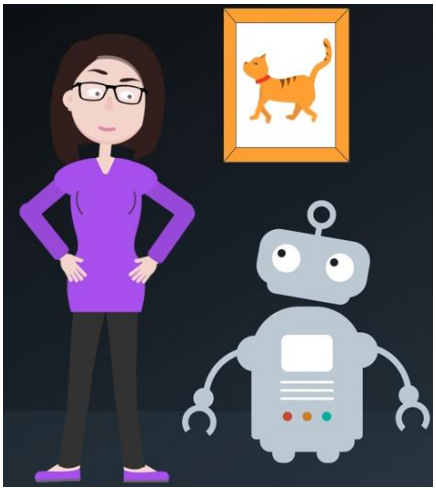
It may difficult to determine it is indeed walking or maybe it`s running.

Maybe it`s just a very talented cat standing in a funny pose with two feet up in the air.

When observe the cat frame by frame, we remember what we see before so we know if the cat is still of if it`s walking.

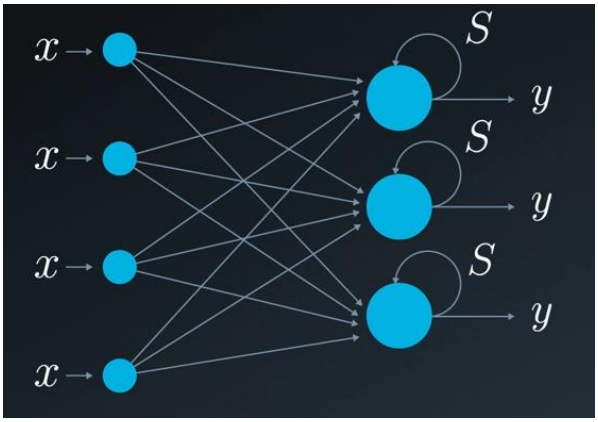
We can also distinguish between **walking or running**.

But can the machine do the same ?



This is where recurrent neural network or RNN come in.

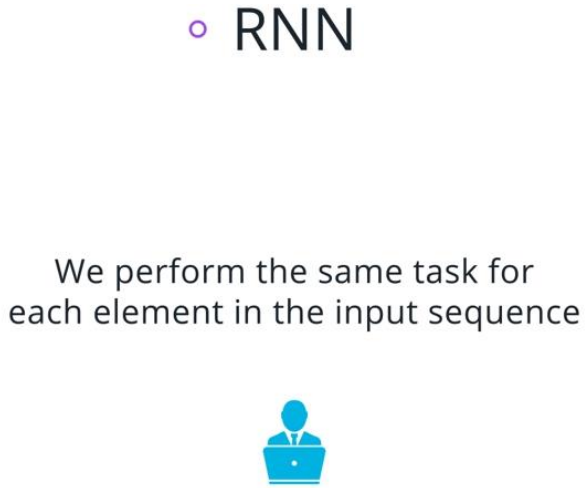
RNNs are artificial neural network, that can capture temporal dependencies, which are dependencies over time.



RNN

If look up the definition of the word recurrent, you will find that it simply means occurring often or repeatedly. So why are these networks called recurrent neural networks?

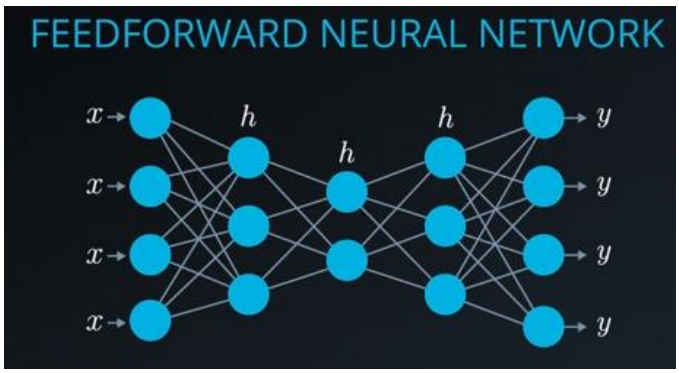
It`s simply because with RNNs we perform the same task for each element in the input sequence.



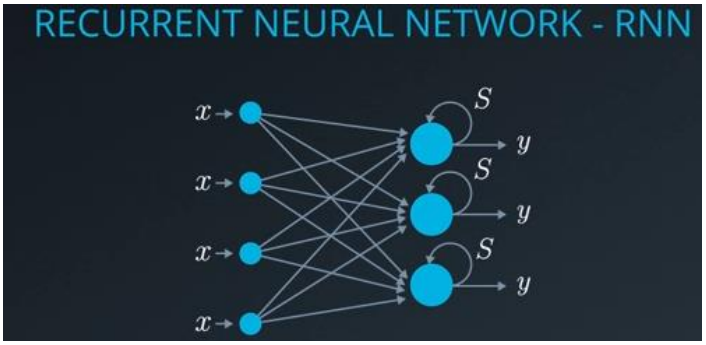
In this lesson, we will start with a bit of history.

It`s really interesting to see how this field has evolved.

We will remined ourselves what feedforward networks are.



You will see that RNNs are based on very similar ideas to those behind feedforward networks.



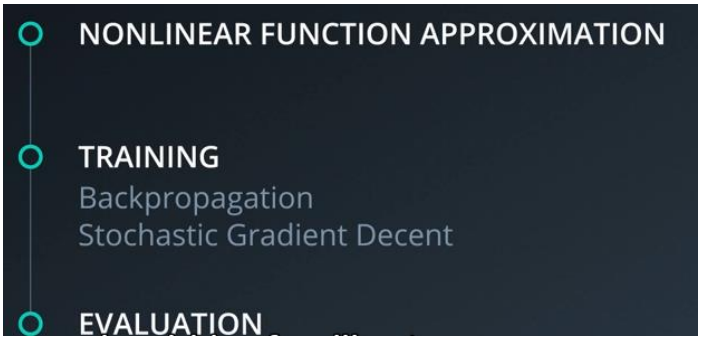
RNN

So, Once we have a clear understanding of the fundamentals, we can easily understand the next steps.

While focusing on feedforward networks, you will learn about,

1- Non-Linear function approximation, training using backpropagation, or What we call stochastic gradient descent and evaluation

All these should be familiar to you.



Our main focused of course will be RNNs

I have given you a good example of why we need RNNs.

Remember the cat, but there are many other applications, we will focus on those well.

We will look a simple of RNN also known as Elman network, and learn how to train the network.

We will also understand the limitations of simple RNNs and how they can be overcome by using what **we call LSTMs (Long short term memory).**

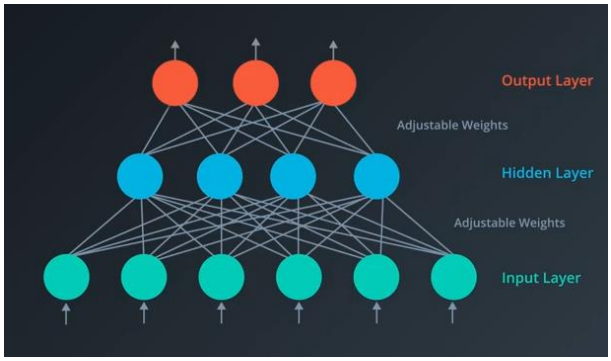
Don`t be alarmed, you don`t need to remember all these names right now We will slowly progress into each concept and talk about all of them in much detail later.

RNNs History

After the First way in Artificial neural network in Mideast.

It became a clear that feedforward networks are limited since they are unable to capture temporal dependencies, which as we said before are dependencies that change over time.

Modeling temporal data is critical in most real-world applications, since neural signals like **speech** and **video** have time varying properties and are characterized by having dependencies across time.



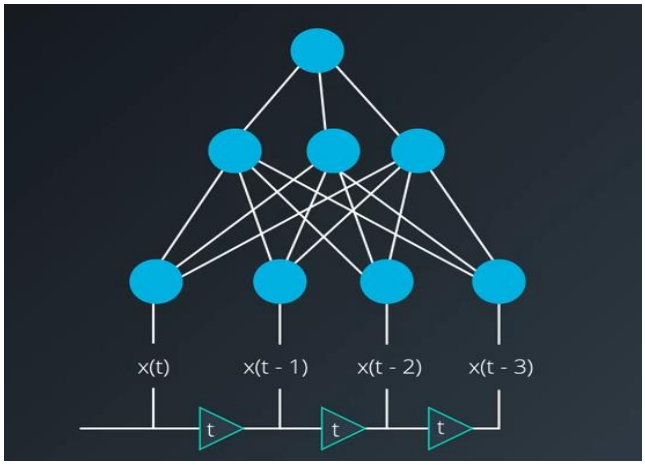
Feedforward Neural Network FFNN (1980'S)

By the way, biological neural networks have recurring connection, so applying recurrence to artificial feedforward neural networks made natural sense.

1- **The first attempt to add memory to neural networks were the Time Delay Neural Networks, or TDNNs in Short.**
2- **And TDNNs input from past timesteps were introduced to the network input, changing the actual external inputs.**

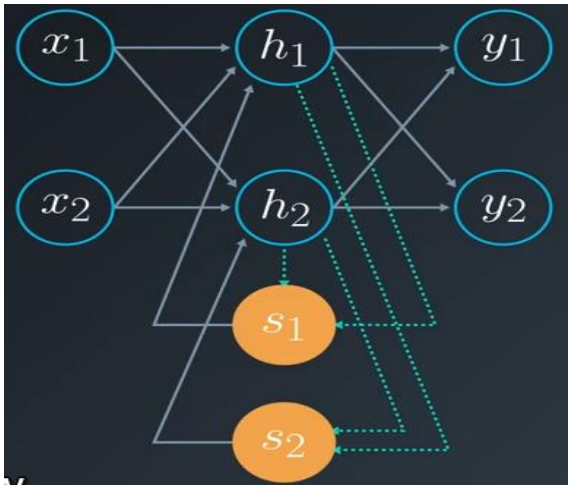
This had the advantage of clearly allowing the network to look beyond the current timestep

But also introduce to clear Disadvantage, Since the temporal dependences were limited to the window of the chosen.



Time Delay Neural Networks TDNN (1989)

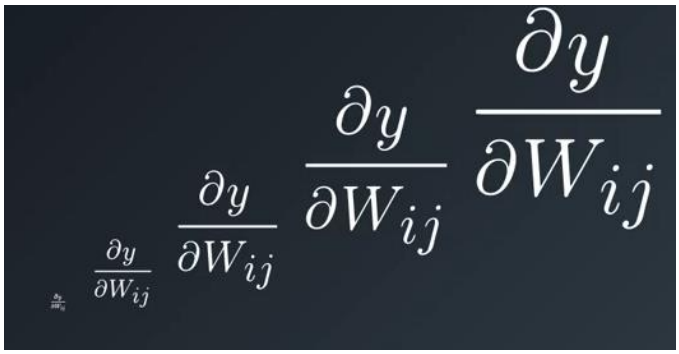
Simple RNN, also known as Elman networks and Jordan networks, were next to follow



Simple RNN Elman network (1990)

We will talk about all those later in the lesson.

It was recognized in the early 90s that all of these network suffer from what we call, the **vanishing gradient problem, in which contribution of information decayed geometrically over time.**



So, capturing the relationship that spanned more than **eight** or **ten steps** back practically impossible

Despite the elegance of those networks, they all had this key flaw.

We will discuss the vanishing gradient problem in detail later.

You can also find more information about the topic right after the video.

In the mid 90s, Long Short-Term Memory cells or LSTMs in short were invented to address this very problem.

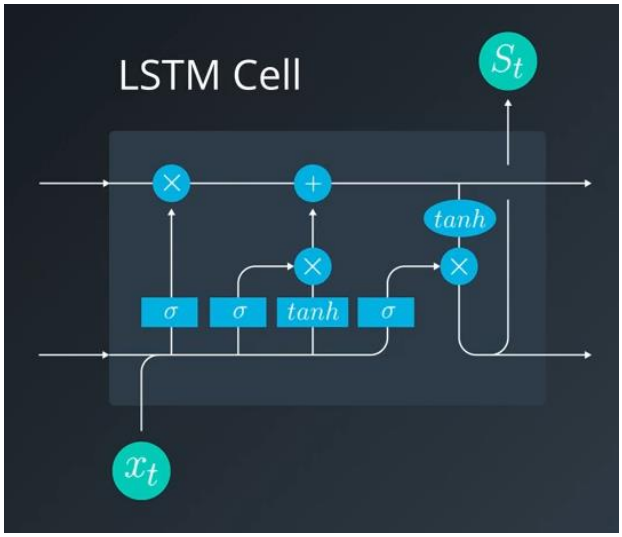
The key novelty in LSTLMs was the idea that some signals what we call state variables can be kept fixed by using gates and reintroduced or not an appropriate time in the future.

In this way, arbitrary time intervals can be represented and temporal dependencies can be captured.

Don`t alarmed by this sketch.

We will get into all of the LSTM details later in this lesson.

Variations on LSTMs such as Gated Recurrent Networks or GRUs in Short further refined this theme and nowadays represent another mainstream approach to realizing RNNs.



RNN application Final

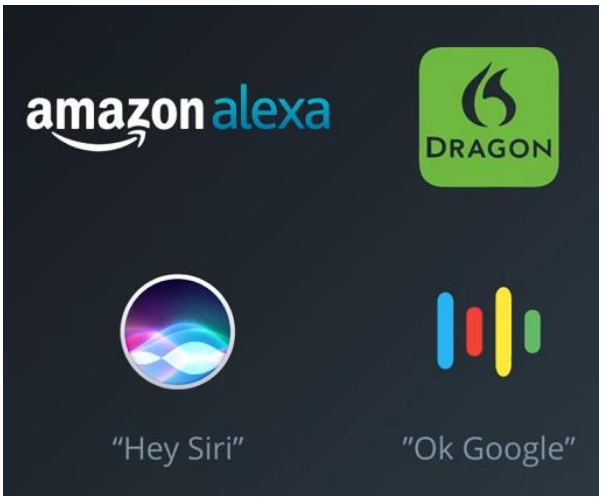
To give an idea of how useful in LSTM.

The world`s leading tech companies are all using RNNs and LSTMs in their applications Let`s take a took at some of those.

Speech recognition, where a sequence of data samples extracted from an audio signal is continuously mapped to text.

Goode examples to

1- **Google Assistant**
2- **Apple Siri**
3- **Amazon Alexa**
4- **Nuances' Dragon solutions**

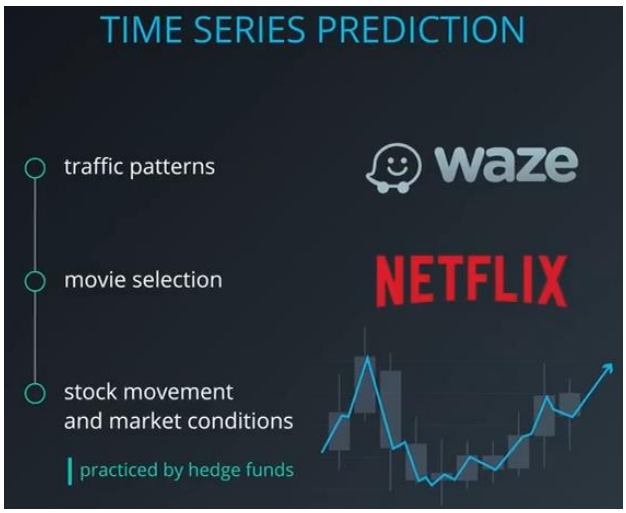


All of these use RNN as a part of their speech recognition software.

Time series predications, where we predict traffic pattern on specific road to help drivers optimize their paths like they do in WAZE

Or predicting what movie a consumer will want to watch next like they do Netflix

Predicating stock price movements based on historical pattern of stock movements and potentially other market conditions that change over time.



This is practiced by most quantities hedge funds.

RNN

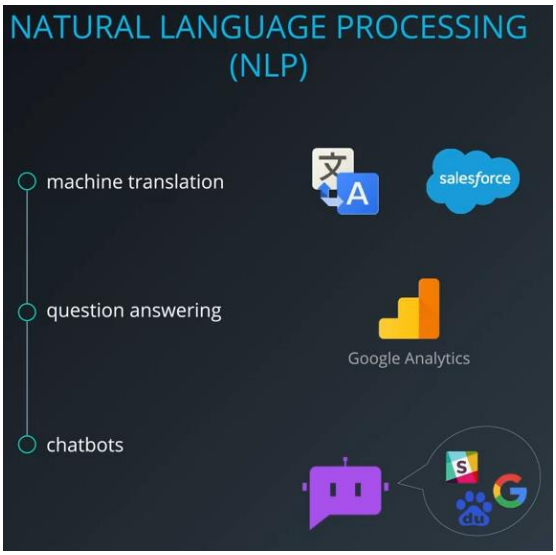
Natural Language processing or NLP in Short such as for example

1- **machine translation used by Google or Salesforce**
2- **Questions answering By Google Analytics (if you have got question about your app, you will soon be able to ask Google Analytics directly)**
3- **Have you head of chatbots**

Many Companies such as

1- **Google**
2- **Baidu**
3- **Slack**

Using RNN to drive their Natural language Processing engines for such applicators.



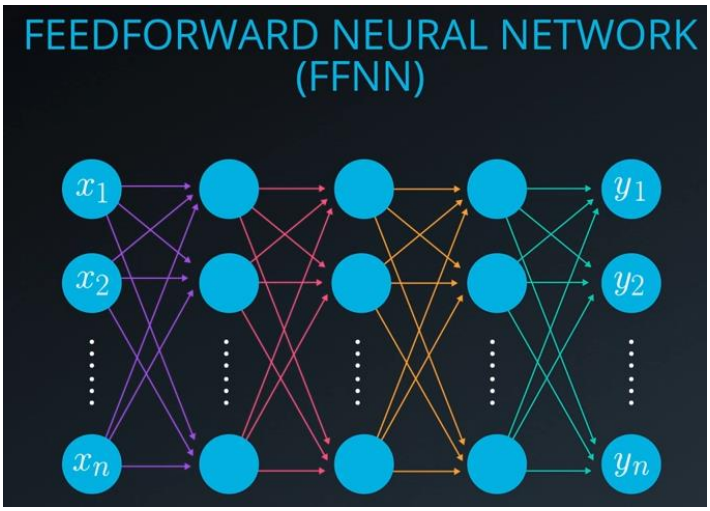
The last applications will focus on is gesture recognition, where we observe a sequence of video frames to determine which gesture the user has made.

For example, waiving your right arm or swinging it.

Here are a few gesture recognition technology companies and companies that are applying gesture recognition in their products.

FFNN Reminder

Before diving in RNN, Let`s remember the process we use in feedforward neural network.

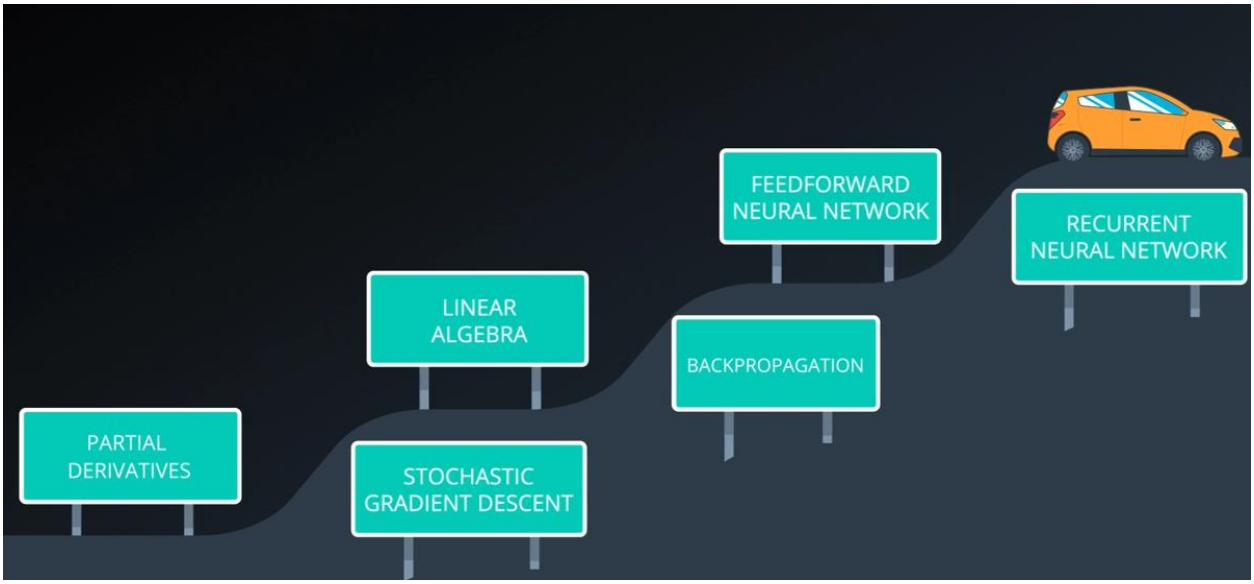


You can have any many hidden layers between the inputs and the outputs, but for simplicity, we will start with a single hidden layer. We will remind ourselves why, when and how it used.

After we have a clear understanding the mathematical background as well,

We will take another step and open the door to RNNs.

You will see that once you have a solid understanding in the basic feedforward network the step towards RNNs in a simple one.



Some of you may be familiar with the concept of Convolutional neural networks or CNN in short.

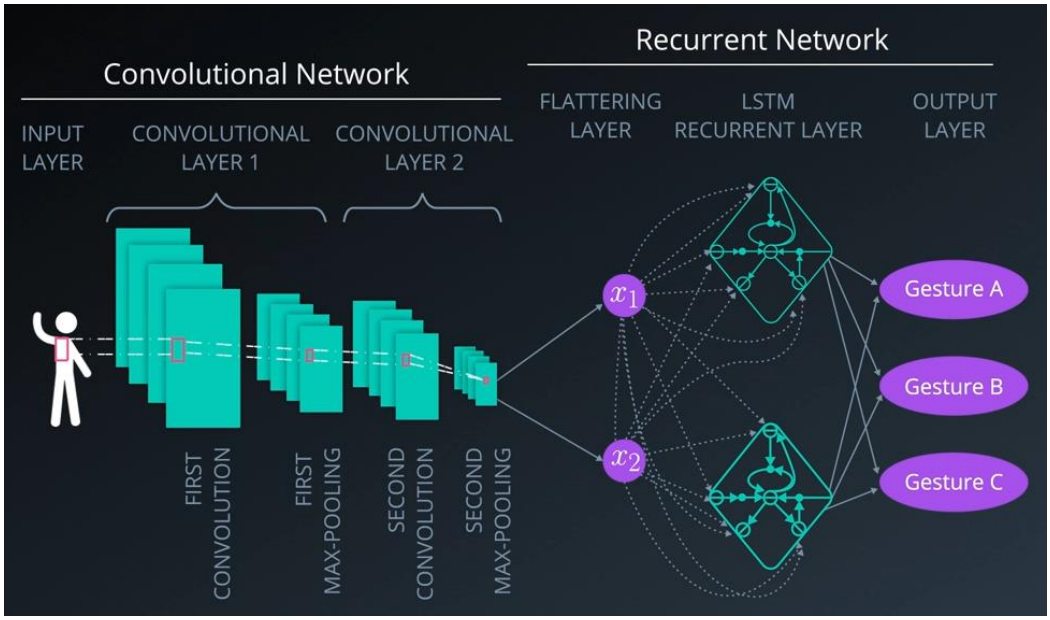
When implementing your neural network net, you will find that you can combine these techniques.

For example.

One can use CNNs in the **first few layers** for the purposes of feature extraction, and then RNNs in the final layer where memory needs to be considered.

A popular application for this is in gesture recognition.

But no warries, if you are not familiar with CNNs that's okay.



When working on a feedforward neural network, we actually simulate an artificial neural network by using a Liner function approximation.

That Function will act as a system that has

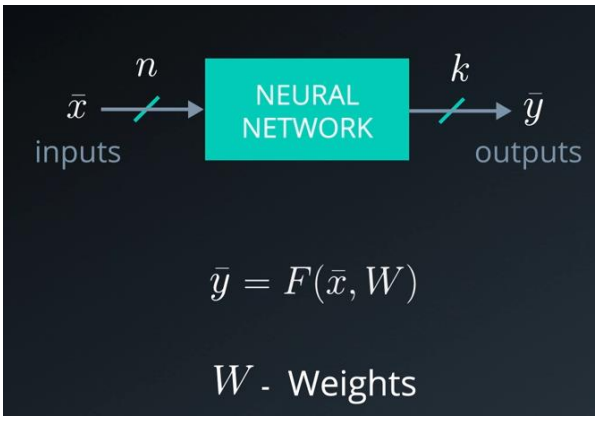
1- **n number of inputs**
2- **weights**
3- **k number of outputs**

We will use x as the input vector and y as the output vector.

Inputs and output can also be

1- **many-to-many**
2- **many-to-one**
3- **one-to-many**

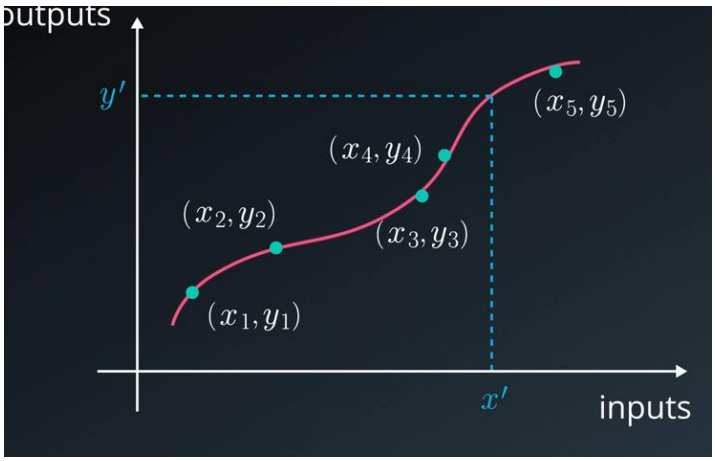
Feedforward neural network



So, when neural network essentially works as a non-linear function approximator, What we do is we try to smooth function between given points x1 and y1 , x2,y2 ……..

And so on in such way that when we have the a new input **x prime** we can find the new output **y prime**

We will elaborate on these non-linear function approximations later in the lesson.

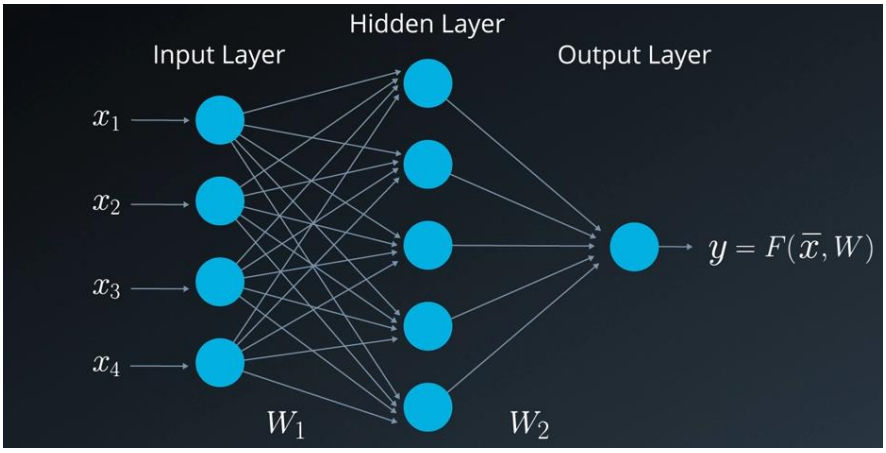


There are two main type of applications

1- **Classifications, where we identify of a set of categories a new input belongs to. For example an image classifications where the neural network receives as an input an image, and can know if it`s a cat.**

2- **The other applications is regression, where we approximate a function, so the network produces continuous values following a supervised training process. A Simple example can be time series forecasting, where we predict the price of stock tomorrow based in price of the stock over the past over days. The input to the network would be a five values representing the price of stock for each of the past five days, and the output we want is tomorrow`s price.**

Our task in neural networks to find the best set of weights that yield a good output where x represent the inputs and w represents the weights.

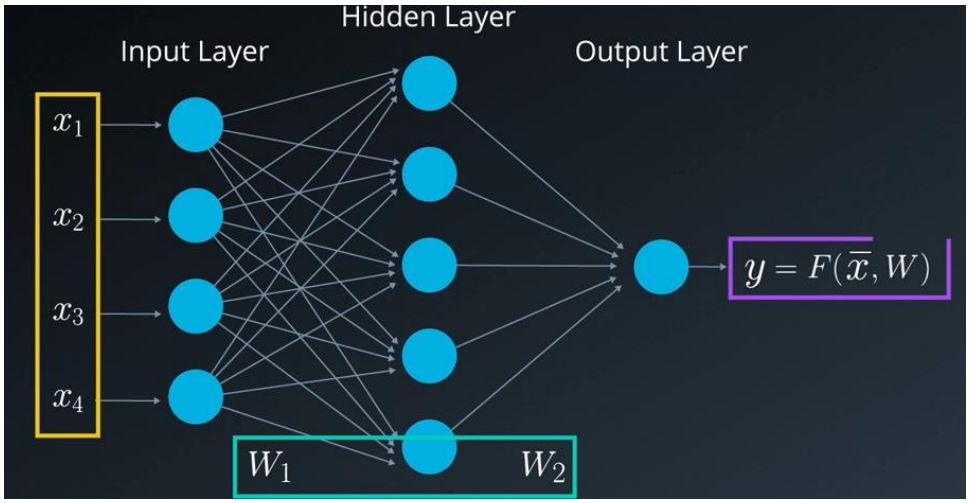


We start with random weights. In feedforward neural networks, we have static mapping from the inputs to the outputs.

It is static mapping

we use word static we have no memory and the output depends only on the **inputs and the weights**.

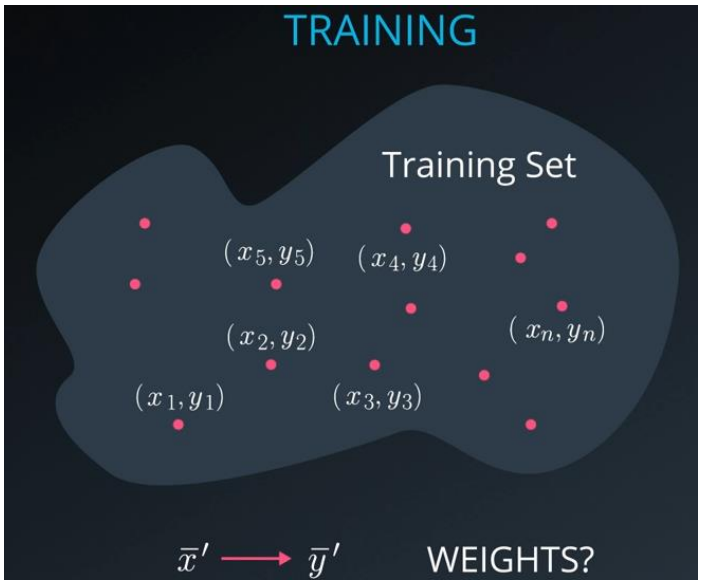
In the other words, for the same input and same weights, we always receive the same output.



Generally speaking, when working with neural networks, we have to primary phases

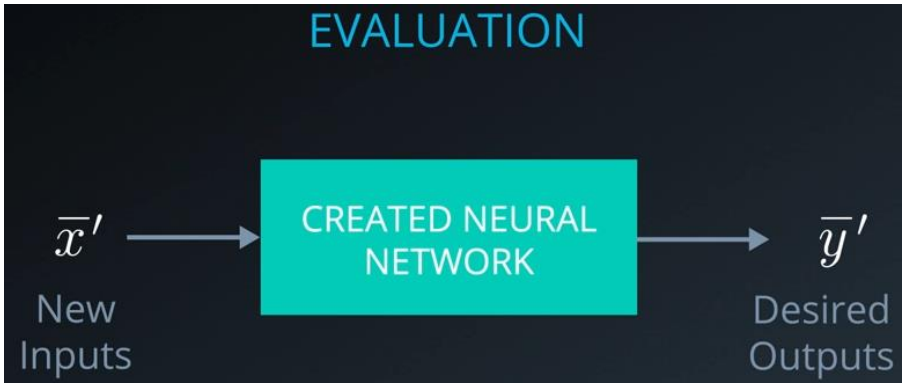
1- **Training**
2- **Evaluation**

In training phase, we take the dataset called the training set which includes many pairs of inputs and their corresponding target or outputs and the Goal is to find a set of weights that would best map the inputs to the desired outputs.



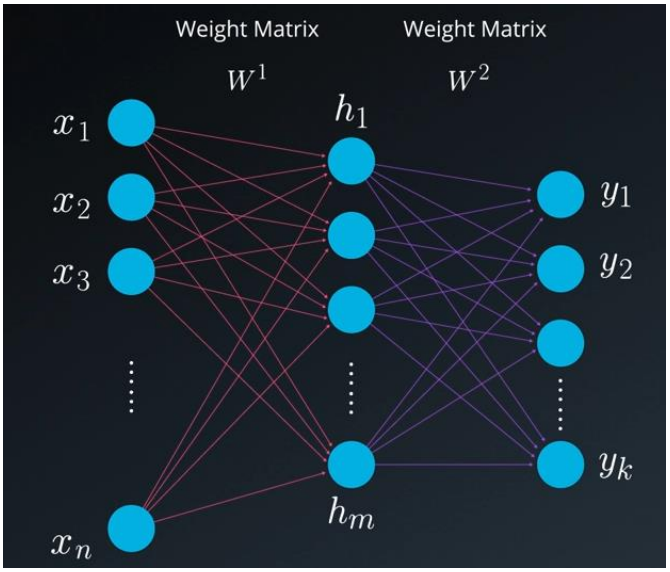
In other words, the goal of the training phase is to yield a network that generalizes beyond the train set.

We used the network that was created in the training phase apply our new input and expect to obtain the desired outputs.



Let`s look an model of ANN, we have only a single a hidden layer.

The inputs are each connected to the neurons in the hidden layer and the neurons in the hidden layer ere each connected to the neurons in the output layer where each neuron there represents a single output.



We can look as it as a collection of mathematical functions.

1- Each input connected mathematically to a hidden layer of neurons through a set of weights we need to modify
2- And each hidden layer neurons is connected to the output layer in a similar way.

$$h_i = F(x_i, W_{ij}^1)$$

$$y_i = F(h_i, W_{ij}^2)$$

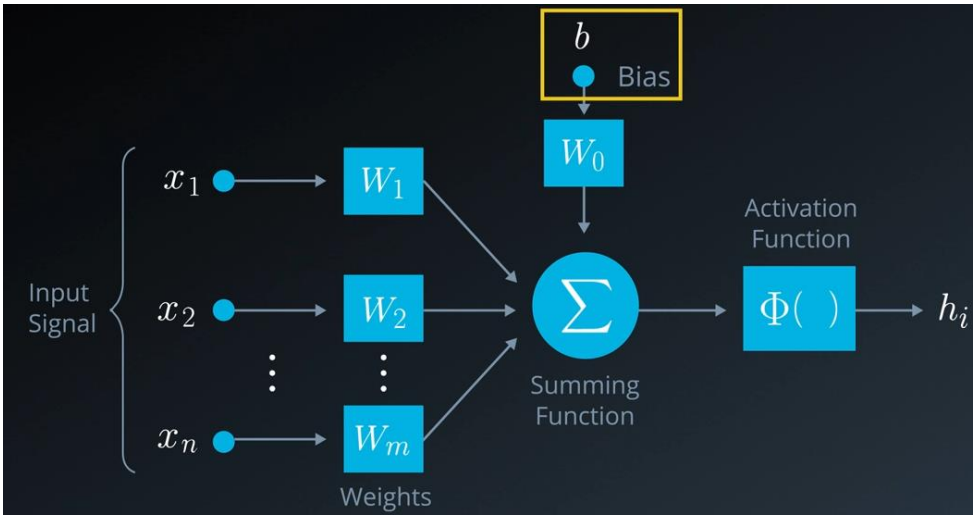
There is not limit to the number of inputs, number of hidden neurons in a layer, and number of outputs.

Nor are there any correlations between those numbers, so we can have

1- **n inputs**
2- **m hidden neurons**
3- **k outputs**

in a closer, even more simplistic look we can see **that each input is multiplied by its corresponding weight and added at the next layer`s neuron** with a **bias** as well.

The bias is the external of the parameter of the neuron and can be modeled by adding an external fixed value input.



The entire summation will usually go through **an activation function** to the next layer or the output layer.

But what is goal?

We can look at out system as a block box that has

1- **n inputs**
2- **k outputs**

our goal is to design the system in such a way that it will give us the correct output y for a specific X.

Our job is to decide what`s inside this black box.

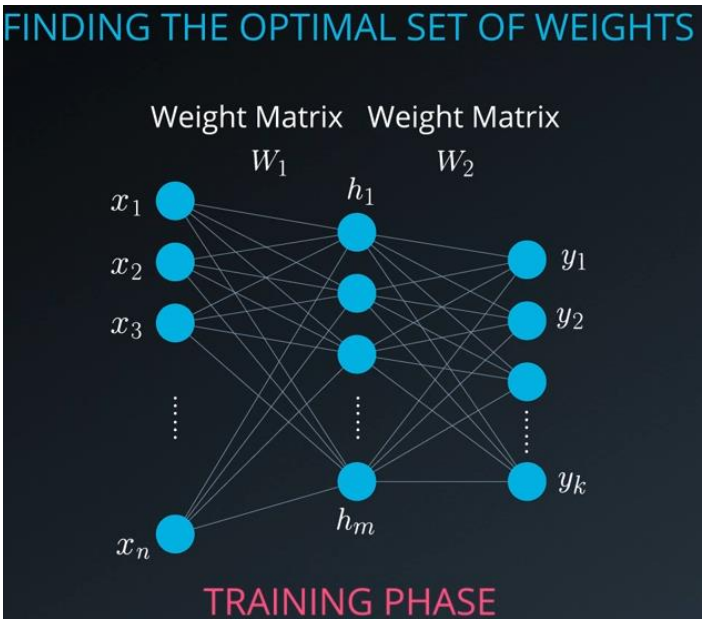
We know that we will use **artificial neural networks** and need to train it to eventually have a system that will yield the correct output to a specific input.

Well, correct, most of the time.

Essentially, what we really want is to find the optimal set of weights?

The optimal set of weights connecting the input to the hidden layer and the optimal set of weights connecting the hidden layer to the output. We will never have a perfect estimation, but we can try to be a close to it as we can.

To do that, we need to start a process you are already familiar with and that is a training phase.



So, look at the training phase, where we will find the best set of weights for our system.

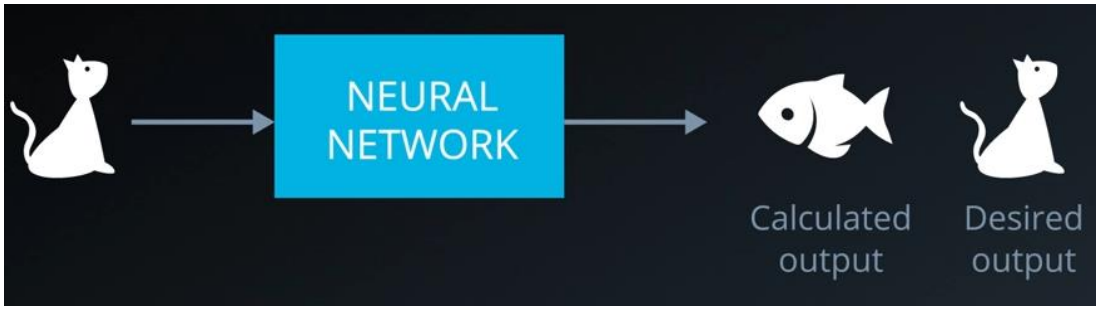
This phase will include two steps

1. **Feedforward**
2. **Backpropagation**

Which will repeat as many times as need until we decide that out system is as best it can be.

In the feedforward part, we will calculate the output of the system.

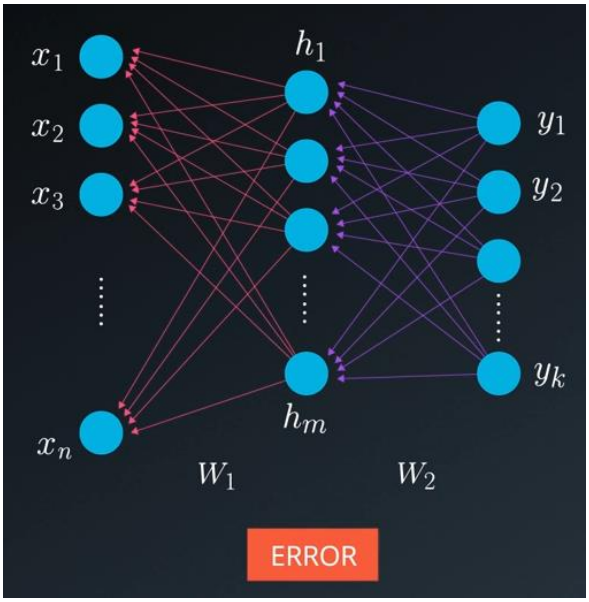
The output will be compared to the correct output giving us an indication of the error.



There are a few way to determine the error.

We will look at it mathematically in a bit

In backpropagation part, we will change weights as we try to minimize the error.



And start the feedforward process all over again.

Let`s look at feedforward

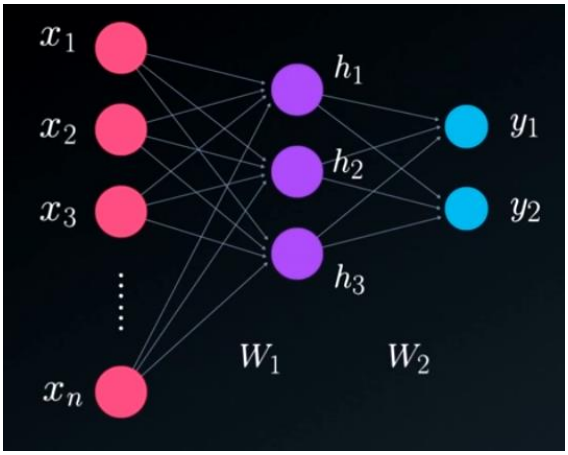
To make our computations easier, let`s decide to have

1- n inputs
2- Three neurons in a single hidden layer
3- Two outputs

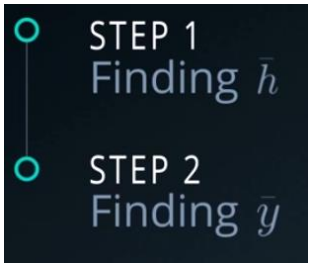
By the way, in practice, we can have thousands of neurons in a single hidden layer.

1- **We will use w_1 as the set of weights from x to h**
2- **And w_2 as the set of weights from h to y.**

Since we have only one hidden layer, we will have two steps in each feedforward cycle.

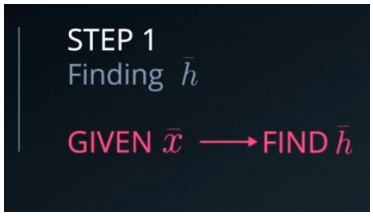


1- **Step one, we will be finding h from a given input and a set of weights W_1.**
2- **And Step two, we will finding the output y from the calculated h and the set of weights W_2.**



You will find that the other the use of non-linear activation functions, which I will took about soon, all of the calculation involve linear combination of inputs and weights. We will use matrix multiplications.

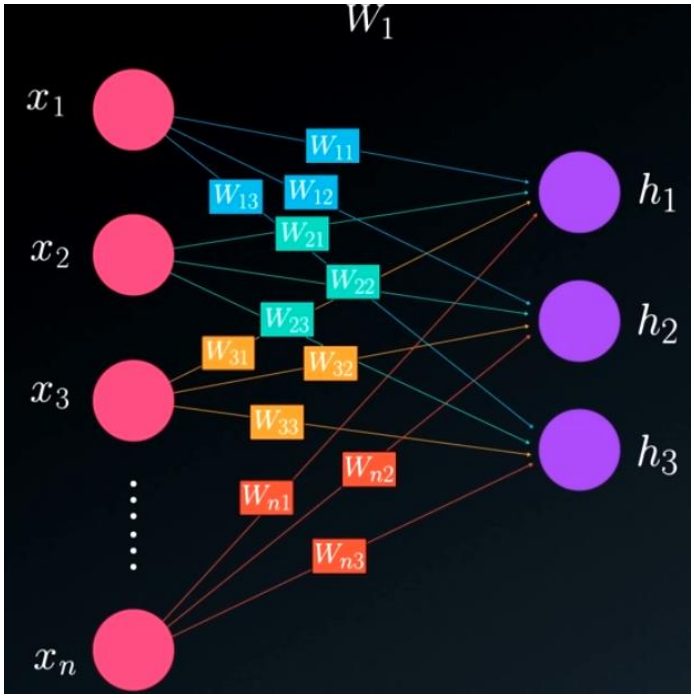
Let`s start with step number one, fining h.



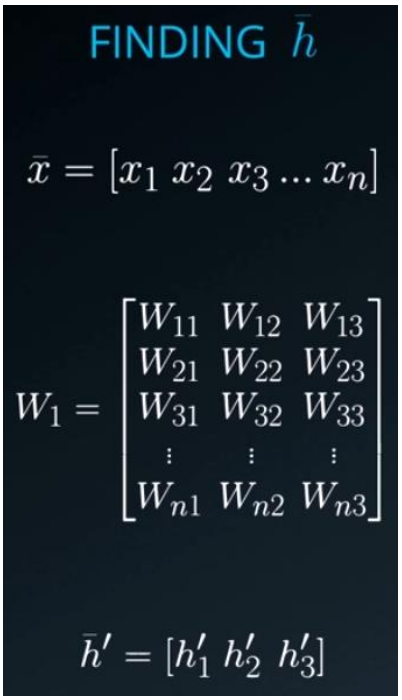
Notice that if we have more than one neuron in the hidden layer, which is usually the case, h is actually a vector.

We will have our initial inputs x , x is also vector and we want to find the values of the hidden neurons h. Each input is connected to each neuron in the hidden layer.

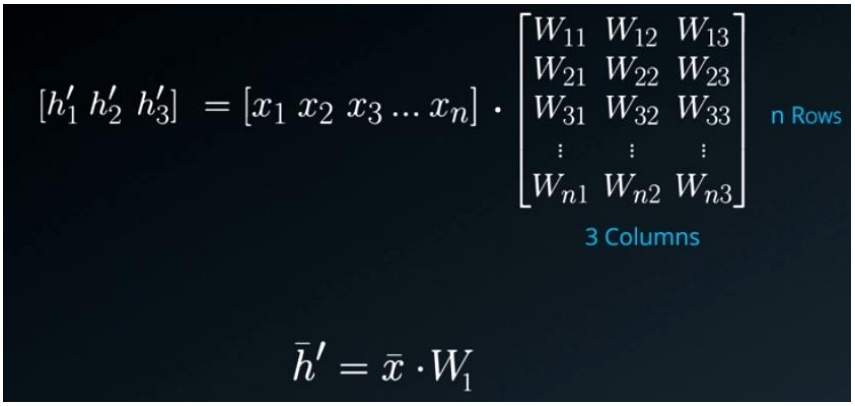
For simplicity, we will use the following indices: W_11 connects x_1 to h_1.



The vector of the inputs x_1,x_2 all the way up to x_n is multiplied by the weight matrix W_1 give us the hidden neurons.



Each vector h equals vector x multiplied by the weight matrix, W_1.



1- In this case, we have a weight matrix with n rows
2- As we have n inputs
3- And three columns, as we have three neurons in the hidden layer

If we multiply the input vector by the weight matrix, you will have a simple linear combination for each neuron in the hidden layer giving us vector h

So for example,

H_1 will be x_1 times W_11 plus x_2 times W_21 and so on.

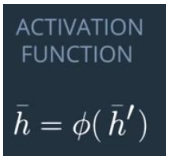
But we are not done with calculating the hidden layer yet

Notice the prime symbol I have been used?

I used it to remind us that are not done with finding h yet.

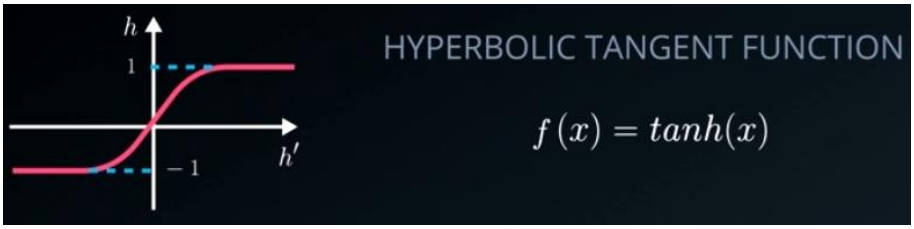
We are almost there but not quite yet.

To make sure that the values of h do not explode or increase too much in size, we need to use the activation function usually denoted by the Greek letter, phi.



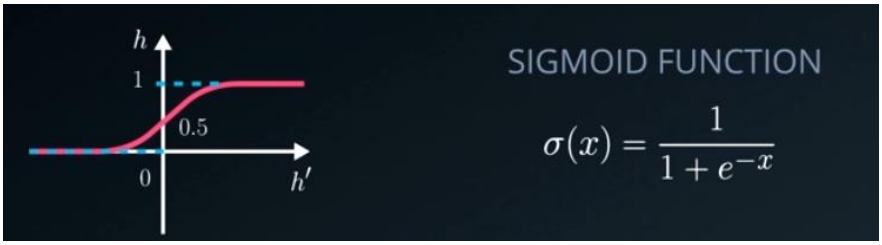
We can use hyperbolic tangent.

a- Using this function will ensure that our outputs are between one and negative one.



We can also used a sigmoid.

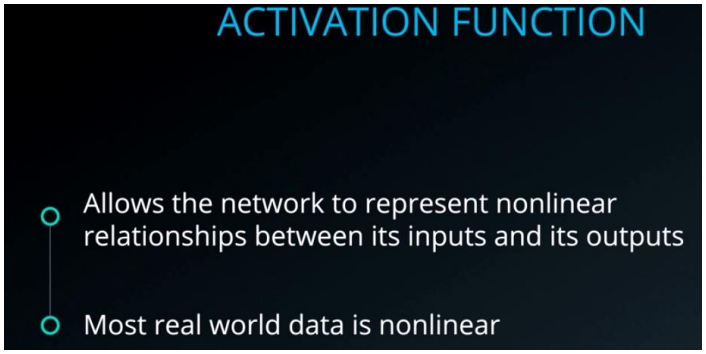
Using this function will ensure that our outputs are between one and zero.



We can also used a rectified linear unit or in short a Relu Function, where the negative values are nulled and positive value remain as they are.

Each activation function has its advantages and disadvantages.

What they all share is that they allow the network to represent non-linear relationships between its inputs and its outputs. And this is very important since most real world data is non-linear.

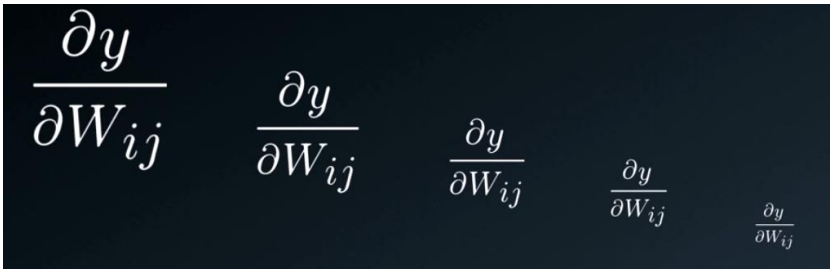


Mathematically, the linear combination and activation function can simply be written as h equals to the output of an activation function of the input vector multiplied by the corresponding weight matrix.

$$\bar{h} = \phi(\bar{x} \cdot W_1)$$

The vanishing Gradient problem.

Using this function can be bit tricky as contribute to the vanishing gradient problem that before.



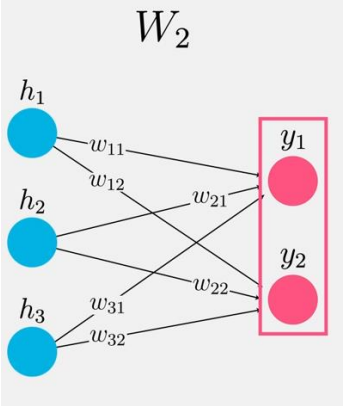
We finish here step one

Step 2 Finding y prime

And will know start with step number two which is finding the output y , by using the value of h, we just calculated.

Since we have more than one output, y will be a vector as well.

We have our initial inputs h and want to find the values of the output y.



Mathematically, idea is identical to what we just saw in step number one.

We now we have different inputs.

1- **We call them h**
2- **Different weight matrix we call it W2**
3- **The output will be vector y**

$$\overline{h} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

3 rows

2 columns

$$\overline{y} = \begin{bmatrix} y_1 & y_2 \end{bmatrix}$$

Notice that the weight matrix has three rows, as we have three neurons in the hidden layer and two columns, as we have only two outputs.

And again, we have a vector of matrix multiplication.

1- **Vector h , multiplied by the weight matrix W2 give us the output vector y.**

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

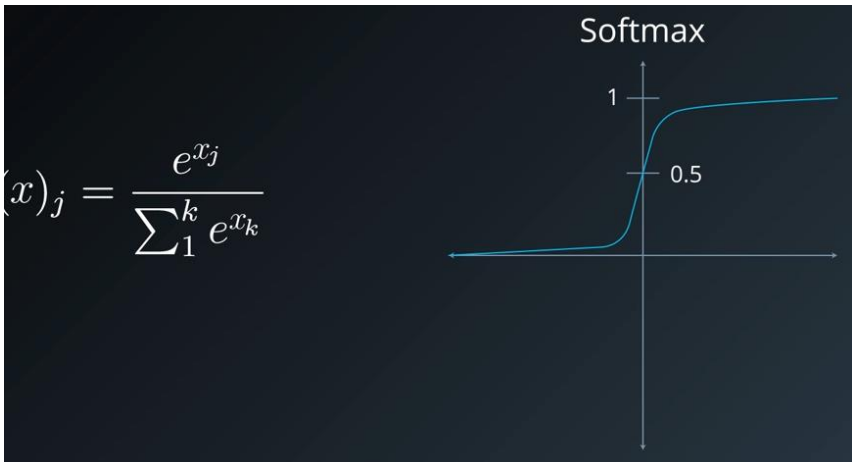
We can put it in a simple equation, where y equals h times W. Once we have the outputs,

$$\overline{y} = \overline{h} \cdot W_2$$

We don`t necessary need an activation function.

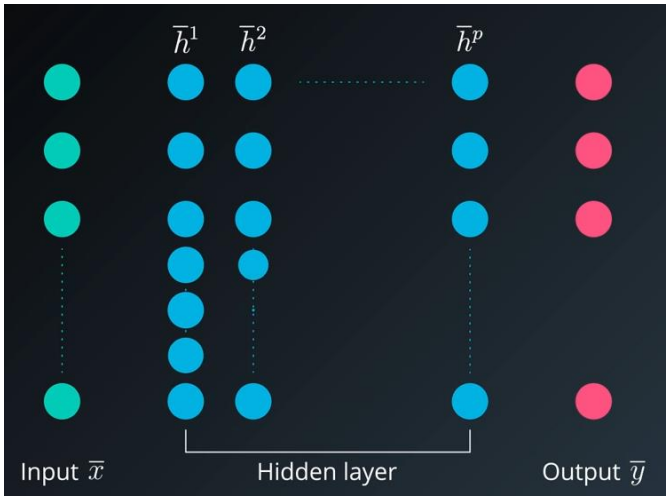
In some applications, it can be beneficial to use for example, a **SoftMax function**, what we call sigma x,

1- If we want the output values to be between zero and one.

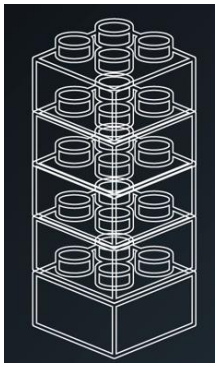


To have a good approximation of the output y ,we need more than one level of hidden layers. **Maybe even 10 or more.**

In this picture I was general number P the number of neurons in each layer can change from one layer to the next, and the mentioned before, can be thousands.



Essentially, you can look at these neuron as building blocks or lego pieces that can be stacked.



So how is this done mathematically ?

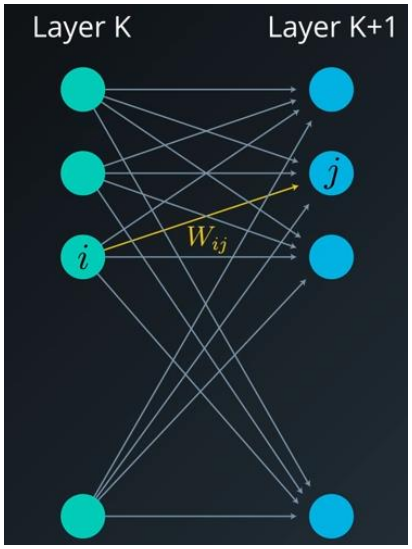
A Simple vector by matrix multiplication, followed by activation function,

$$\overline{h}^{p} = \phi(\overline{x} \cdot W_{p})$$

Where the vector indicate the inputs, and the matrix indicates the weights connecting one layer to the next.

To generalize this model, let`s look into one random K Layer

The weight W_{ij} from the K layer to the K plus 1 corresponds to the I F input going into to j F output.

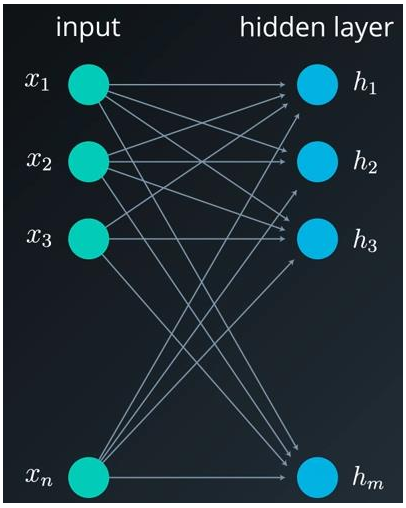


RNN

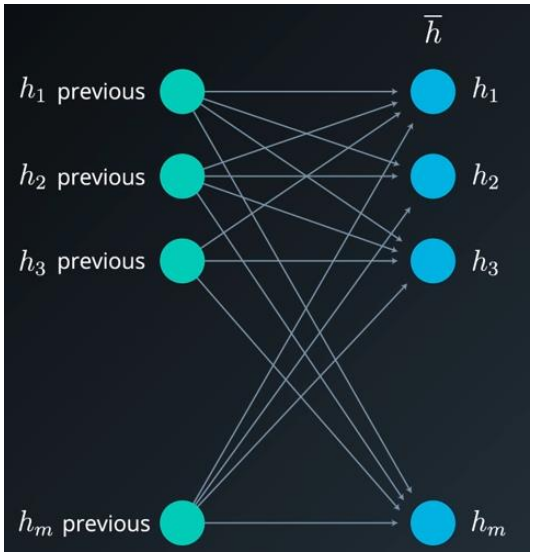
You might want to get a pencil for this one, as these important mathematical derivations. As you follow you math, pause the video , write the derivations in your notes

Try to do so throughout the entire lesson. Let`s treat layer K, as the input x and Layer K plus one as the outputs of the hidden layer h.

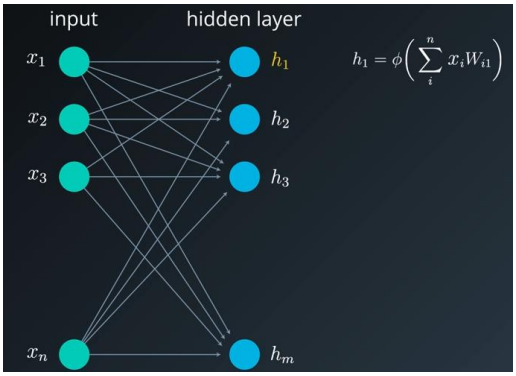
We will have n_x inputs and m_h outputs. By the way



If we are not dealing with inputs but rather than the outputs of the previous hidden layer, the only thing that will change is the letter that we choose to use but the calculations will be the same.

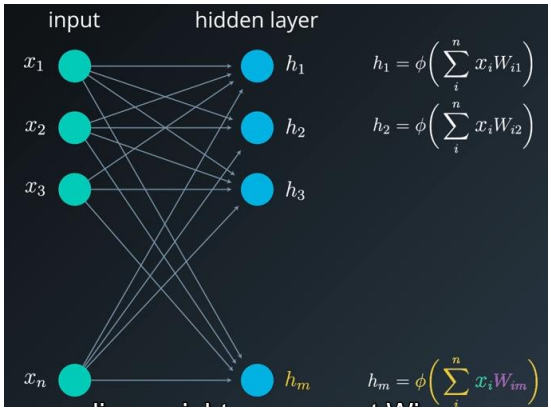


So to make the notation simple, Let`s just stay **x.h1 is the output of an activation function of a sum,**

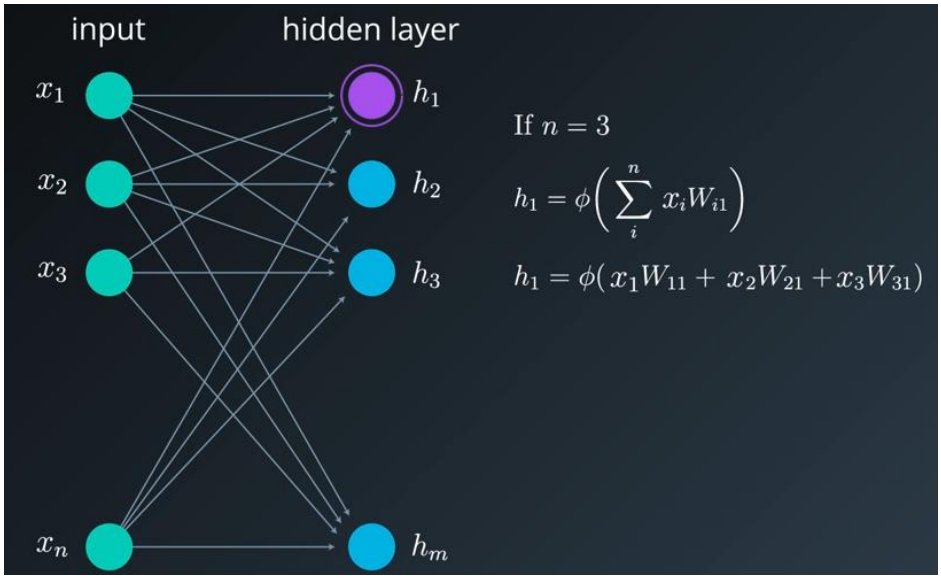


Where the sum is a multiplication of each xi, by corresponding weight component Wi1.

The same way hm is the output of an activation function of a sum, and the sum is multiplication of each input xi, by it corresponding weight component W_{im}



For example, if we have three inputs and we want to calculate h1 it will be the output an activation function following linear Combination.

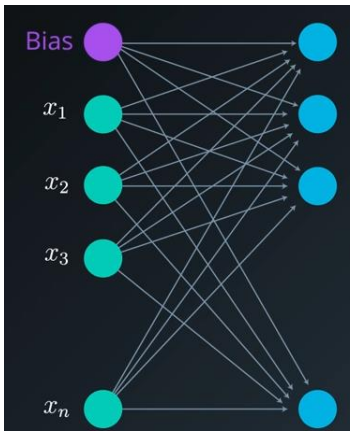


These single element calculations will be helpful in understanding back propagation, which is why understand them as well.

But as before, we can also look at these calculations as a vector by matrix multiplication.

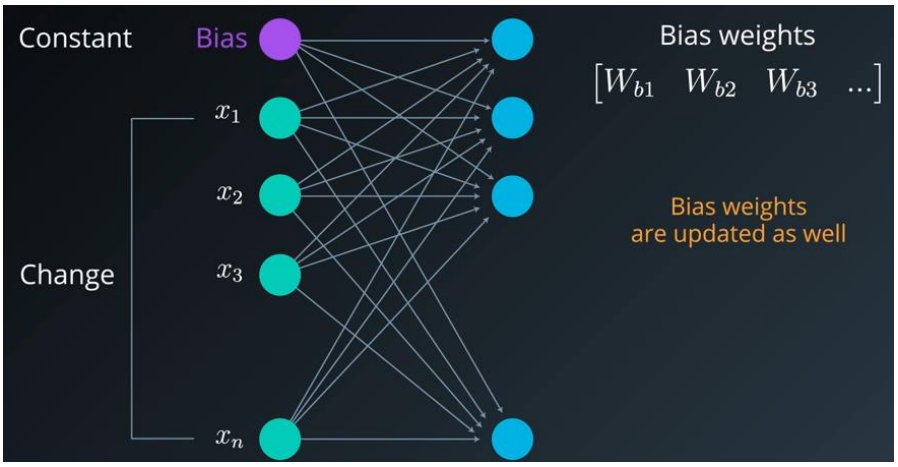
$$\overline{h} = \phi\left(\overline{x} \cdot W\right)$$

By the way you probably noticed that I didn`t emphasize the bias input here.



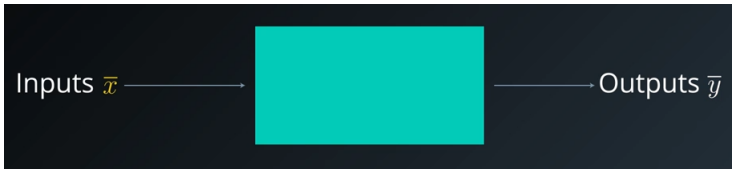
The bias does not change any of these calculations. Simply consider it as a constant input usually one that also connected to each of the neurons of the hidden layer by a weight.

The only difference between the bias and any other input, is the fact that it remains the same as each of the other inputs change. And just as all the other inputs, the weights connecting it to the next layer are updated as well.

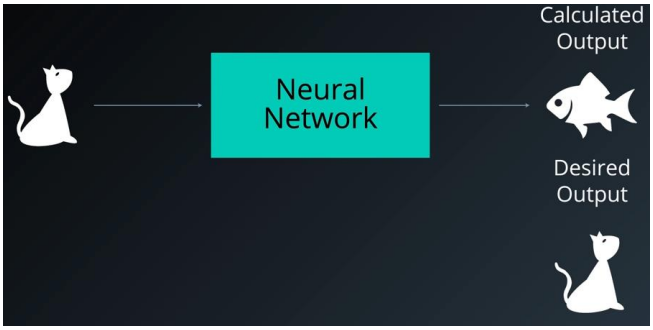


What is the goal again?

Our goal is to find the best **set of weights**, that will give us at the end the desired **output**. Right ? At the end, we want find a system that will give us **the correct output**, for a **specific input.**

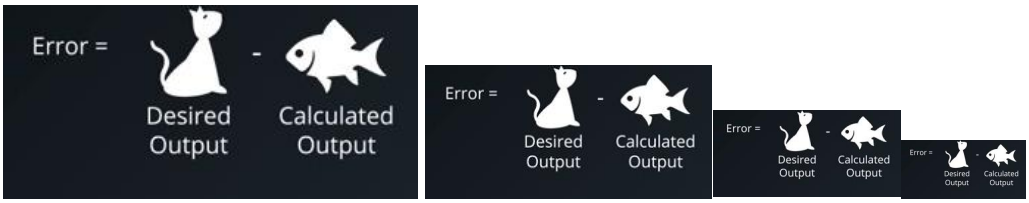


In the training phase, we actually know the output of a given input. We calculate the output of the system in order to adjust the weights.



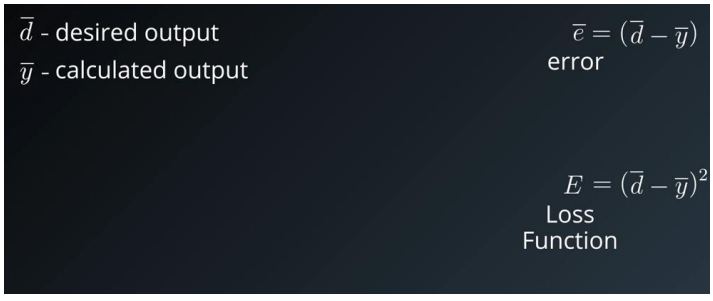
We do that by finding the error and trying to minimize it.

Each iteration of the training phase will decrease the error just a bit, until we eventually decide that the error is small enough.



Let`s focus the an intuitive error calculations, Which is simply finding the difference between the calculated, and the desired output. This is our basic error. For our backpropagation calculations,

1- **We will use the square error which also called the loss function**



After finding h' , we need an activation function (Φ) to finalize the computation of the hidden layer's values. This activation function can be a Hyperbolic Tangent, a Sigmoid or a ReLU function. We can use the following two equations to express the final hidden vector $\bar{h}$:

$\bar{h} = \Phi(\bar{x}W^1)$

or

$\bar{h} = \Phi(h')$

Since W_{ij}
represents the weight component in the weight matrix, connecting neuron **i** from the input to neuron **j** in the hidden layer, we can also write these calculations in the following way:
(notice that in this example we have *n* inputs and only 3 hidden neurons)

$$h_1 = \Phi(x_1W_{11}+x_2W_{21}+....+x_nW_{n1})$$

$$h_2 = \Phi(x_1W_{12}+x_2W_{22}+....+x_nW_{n2})$$

$$h_3 = \Phi(x_1W_{13}+x_2W_{23}+....+x_nW_{n3})$$

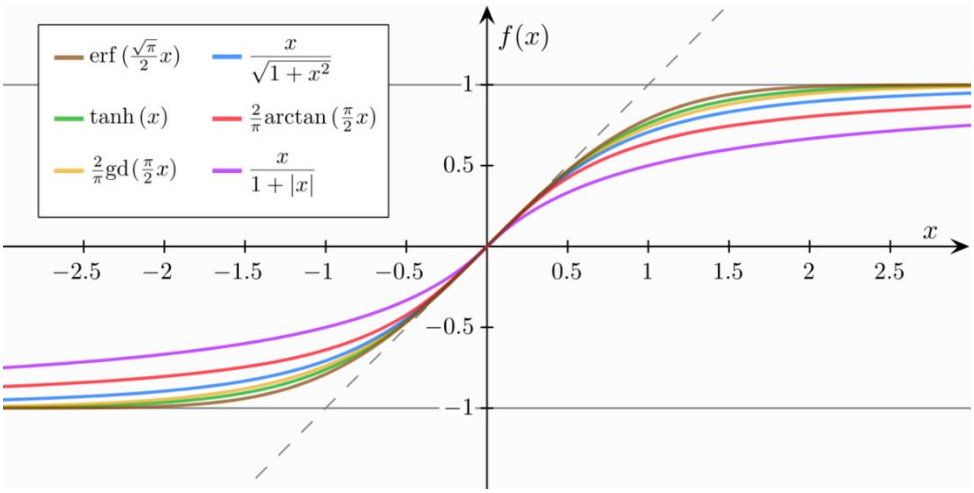
Essentially, each new layer in an neural network is calculated by a vector by matrix multiplication, where the vector represents the inputs to the new layer and the matrix is the one connecting these new inputs to the next layer.

In our example, the input vector is $\bar{h}$ and the matrix is W^2, therefore $\bar{y} = \bar{h}W^2$. In some applications it can be beneficial to use a softmax function (if we want all output values to be between zero and 1, and their sum to be 1).

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Refresher: The Sigmoid Function

The sigmoid function has been widely used in machine learning intro materials, especially for the logistic regression and some basic neural network implementations. However, you may need to know that the sigmoid function is not your only choice for the activation function and it does have drawbacks.



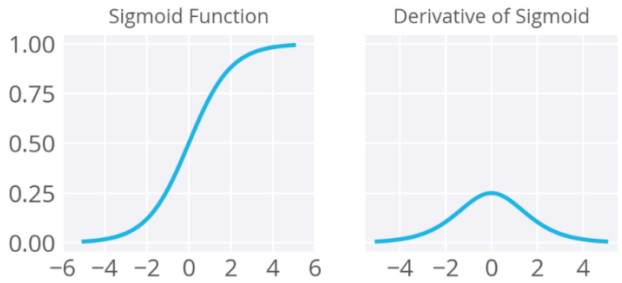
Let's see what a sigmoid function could benefit us.

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

By taking this formula, we can get the derivative of the sigmoid function, note that for shortening the formula, here f(x) is the sigmoid function.

$$f'(x) = f(x)(1-f(x))$$

Despite that, such a result from the derivate is easy to calculate and save times for building models, the sigmoid function actually forces your model "losing" knowledge from the data. Why? Think about the possible maximum value of the derivative of a sigmoid function.



For the backpropagation process in a neural network, it means that your errors will be squeezed by (at least) a quarter at each layer. Therefore, deeper your network is, more knowledge from the data will be "lost". Some "big" errors we get from the output layer might not be able to affect the synapses weight of a neuron in a relatively shallow layer much ("shallow" means it's close to the input layer).

Due to this, sigmoids have fallen out of favor as activations on hidden units.

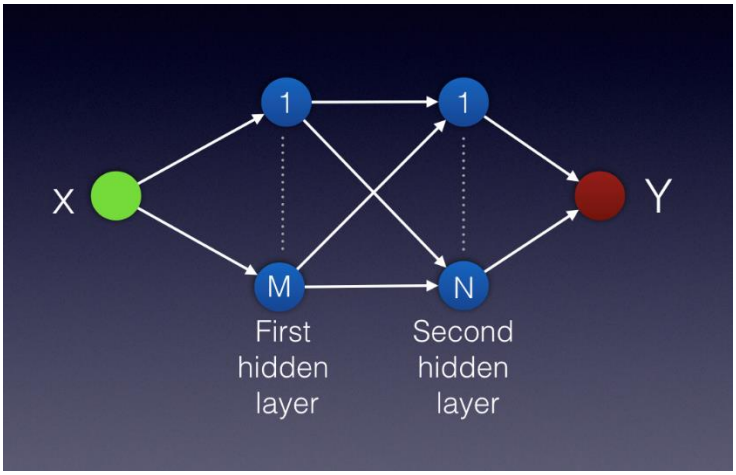
Quiz

The following picture is of a feedforward network with

- A single input x
- Two hidden layers
- A singe output

The first hidden layer has M neurons.

The second hidden layer has N neurons.



What is the total number of multiplication operations needed for a single feedforward pass?

MN

M+N

M+N+2MN

M+N+NM

There isn't enough information to solve this question

Solution : M+N+NM

Solution

To calculate the number of multiplications needed for a single feedforward pass, we can break down the network to three steps:

- Step 1: From the single input to the first hidden layer
- Step 2: From the first hidden layer to the second hidden layer
- Step 2: From the second hidden layer to the single output

Step 1

The single input is multiplied by a vector with M values. Each value in the vector will represent a weight connecting the input to the first hidden layer. Therefore, we will have **M** multiplication operations.

Step 2

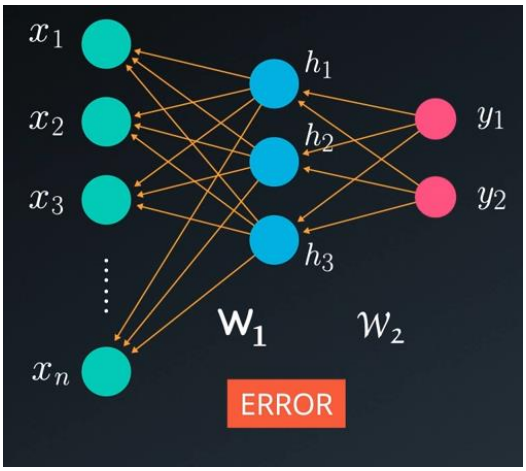
Each value in the first hidden layer (M in total) will be multiplied by a vector with N values. Each value in the vector will represent a weight connecting the neurons in the first hidden layer to the neurons in the second hidden layer. Therefore, we will have here M times N calculations, or simply **MN** multiplication operations.

Each value in the second hidden layer (N in total) will be multiplied once, by the weight element connecting it to the single output. Therefore, we will have **N** multiplication operations.

In total, we will add the number of operations we calculated in each step: **M+MN+N** .

Backpropagations

Know we complete feedforward, received an output, and calculated the error, we are read to go backwards in order to change our weights with a goal of decreasing the network error.



Going backwards from the output to the input while changing the weights, is a process we call back propagation,

Which essentially stochastic gradient descent computed using the chain rule.

If you are not familiar or comfortable with back propagation yet, this section will hep you out.

We will now be a little more mathematical.

I find it fascinating to see how math comes to life, how mathematical calculations eventually lead us to implementing in this case a neural network, Which is main building block of Ai.

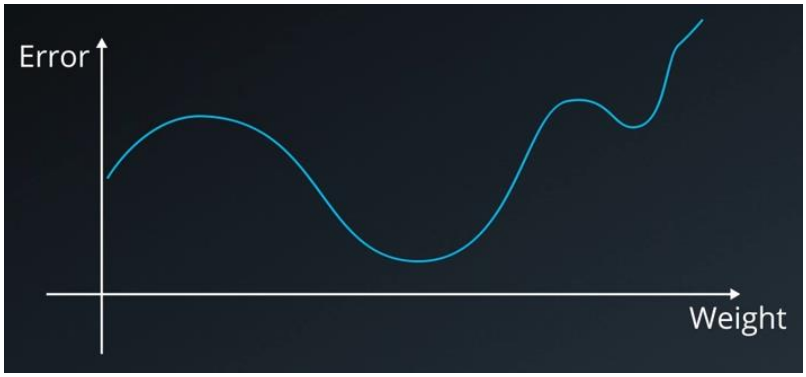
To be able to implement a basic neural network, won`t doesn’t really need to deeply a mathematical understanding as now we have a open source tools but  to really understand how it works and to optimize our applications, it`s always important to know the math.

This is where I want to ask again to add these old school techniques and with notes sa I drive the math I really believe that as you write your own notes you will feel more confident with the math since it will be with your own handwriting.

Our goal to find a set of weights that minimize the network error.

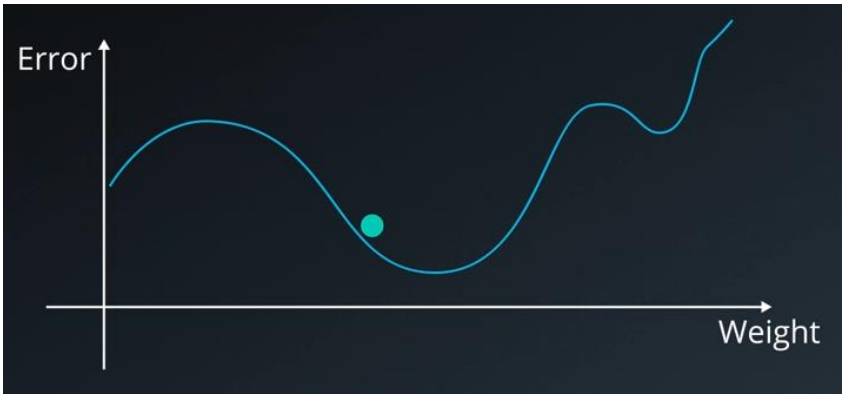
So how do we that?

Well, we use an iterative process presenting the network with one input at a time from our training set.



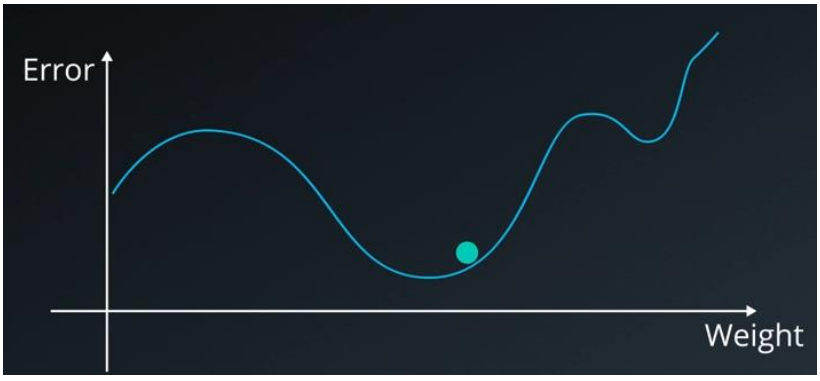
As I mentioned before during the feedforward pass for each input, we calculate the networks error.

We can then use the error slightly change the weights in the correct direction, each time reducing the error by just a bit.



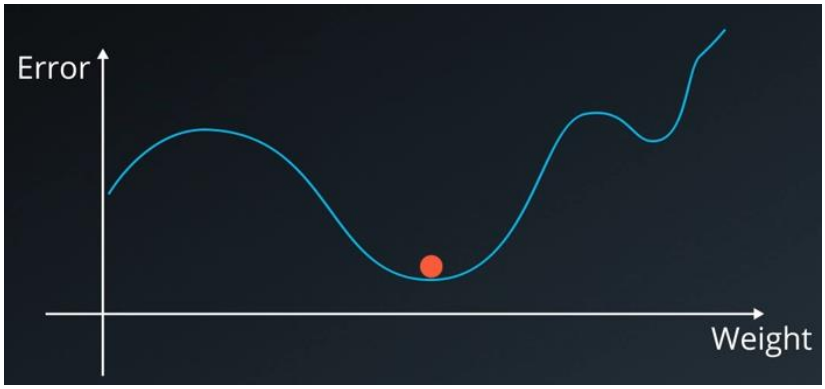
We continue to do so until we determine that the error is small enough.

So how small is small enough?



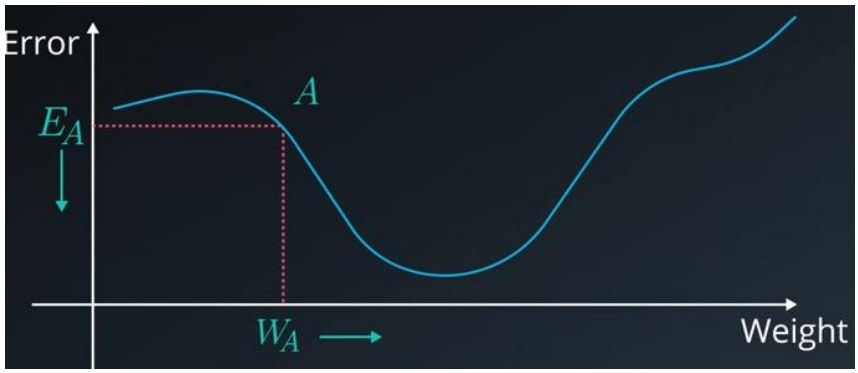
How do we know if we have a good enough mapping from the inputs to the outputs?

Well, there`s is know simple answer to that.

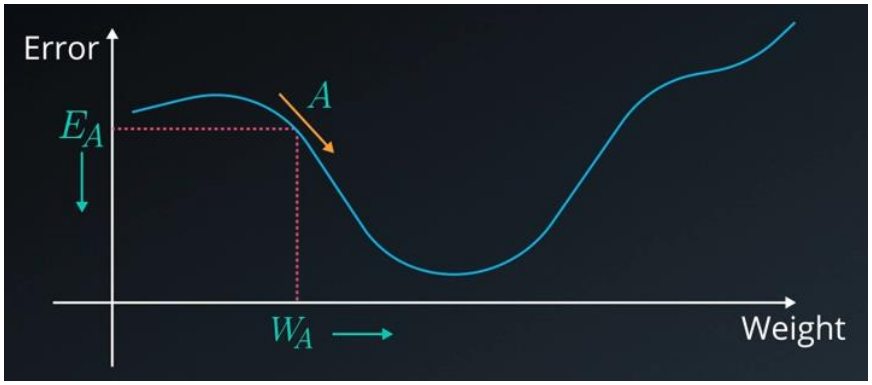


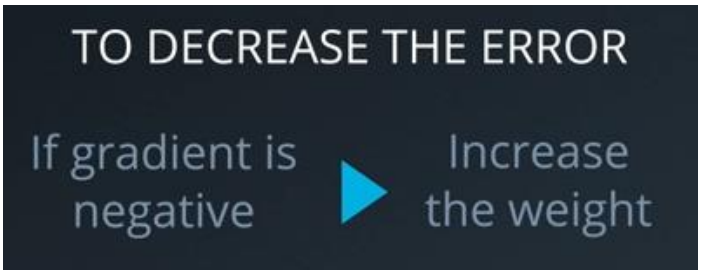
You Will find practical solutions to some of these questions in our next section on over-fitting.

Imagine the network with only one weight W , Assume that during a certain point in the training process, The weight has a value of W_A and the error at Point A is E_of_A.

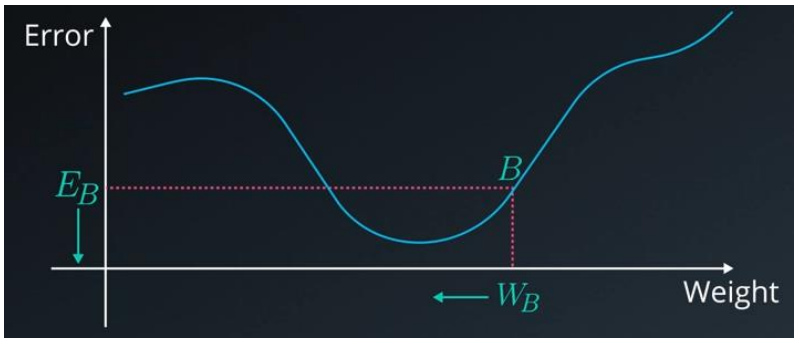


To reduce the error, we need to increase the value of he weight, W_A . Since the gradient or , **in other words the derivative which the slope of the curve at Point A is negative since it's pointing down we need to change the weight its negative direction so that we actually increase the value of** W_A

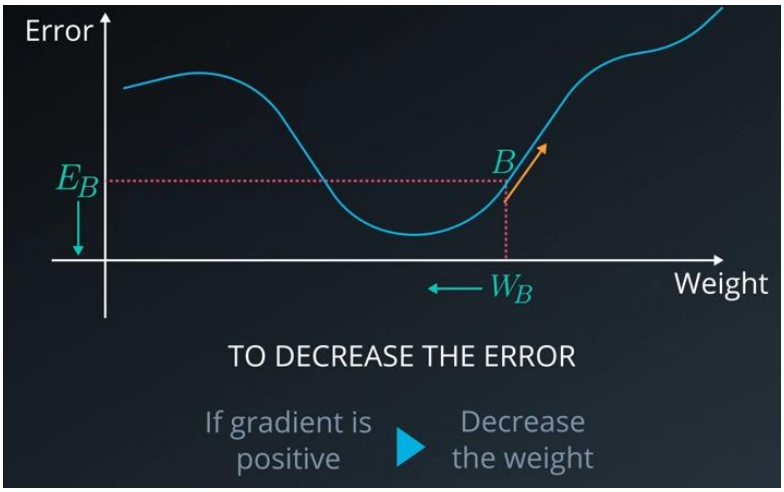




If, on the other hand, the weight has a value of W_B with a network being Eof B to reduce the error we need to decrease the weight W_B.



If you look at the gradient and point B you will see that it`s positive. In this case, changing the weight by talking a step in the negative direction of the gradient would mean that we are correctly decreasing the value of the weight.



The case we looked at was of a single weight which was oversimplifying the more piratical case where the neural network as many weights.

$$W_{new} = W_{previous} + \alpha\left(-\frac{\partial E}{\partial W}\right)$$

1- We can summarize the weight update process using this equation where alpha is the learning rate the step size
2- Where also see the weight W for the reasons I just mentioned changes in the opposite direction of the partial derivations of the error with respect to W.

Why we looking at partial derivatives And the answer is simply because the error is a function of many variables and the partial derivative let`s measure how the error is impacted by each weight separately.

You can find a good resources on how to turn the learning rate at the end of this video.

You will also find a full a hyper parameter section following this online lesson

Wights update

Since many weights determined the network`s output, we can use a vector of the partial derivatives of the network error, each with respect to a different weight.

$$W_{new} = W_{previous} + \alpha\nabla_W(-E)$$

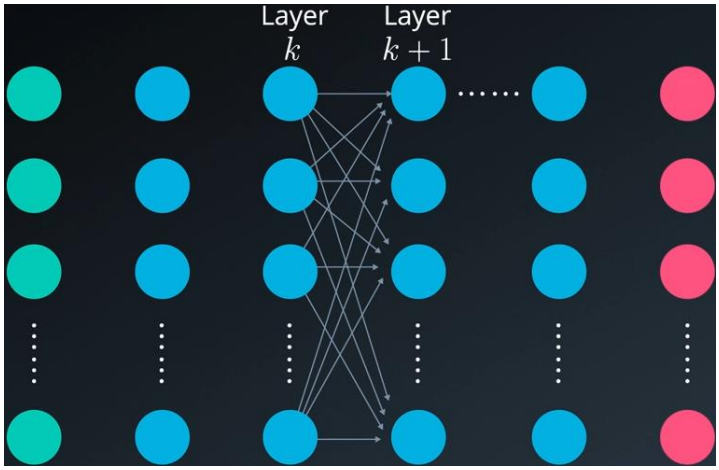
The strange inverted triangular Symbol, if you haven't seen it before is the gradient.

This gradient is the vector of partial derivatives of the error with respect to each of the weights.

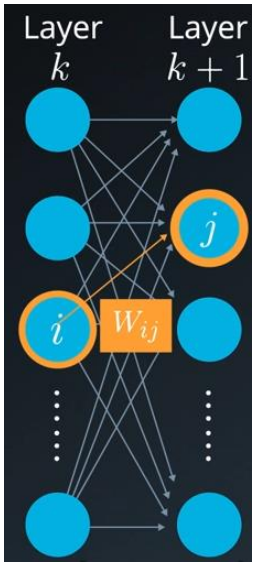
For the purpose of proper notation, we will have a quite a few indicates here.

In this illustration which should be familiar by now.

We are focusing on the connections between layer k and layer k_plus_1.



The weight W_{ij} connects neuron I in layer k to neuron J in the layer k_plus_1.



Let`s call the amount by which we change or update weight , W_{ij} , delta_of _ W_{ij}.

Weight Update

$$\Delta W_{ij}^{k}$$

The superscript K, indicates that the weights connects layer k to k_plus_1 or another words, it originated from layer K.

Calculating this delta weight of W_{ij} is straightforward. It equal to learning rate multiplied by the partial divertive of the error with respect to weight, W_{ij} in each layer.

$$\Delta W_{ij}^{k} = -\alpha \frac{\partial E}{\partial W_{ij}^{k}}$$

We will take the negative of that term for reasons I just mentioned before.

1- **Basically, backpropagation boils down to calculating the partial derivative of the E with respect to each of the weights**

$$\Delta W_{ij}^{k} = -\alpha \frac{\partial E}{\partial W_{ij}^{k}}$$

2- **And then adjusting the weights according to the calculated value of delta of W_{ij}.**

$$W_{new} = W_{previous} + \Delta W_{ij}^{k}$$

3- **This calculation are done for each layer.**

Let`s look at our system one more time. For the error, we will use the loss function which is simply the desired output minus the network output, all that squared. Also dividing this error term by 2 for calculation simplicity that you will see later. Now we have all our equations defined we can dive into the math with an example.

$$E = \frac{(\overline{d} - \overline{y})^2}{2}$$

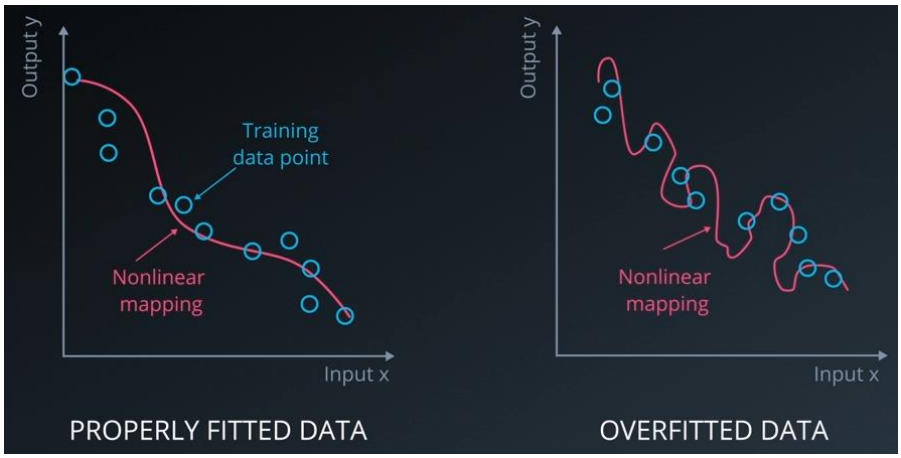
$\overline{d}$ – Desired Output

$\overline{y}$ – Calculated Output

Overfitting

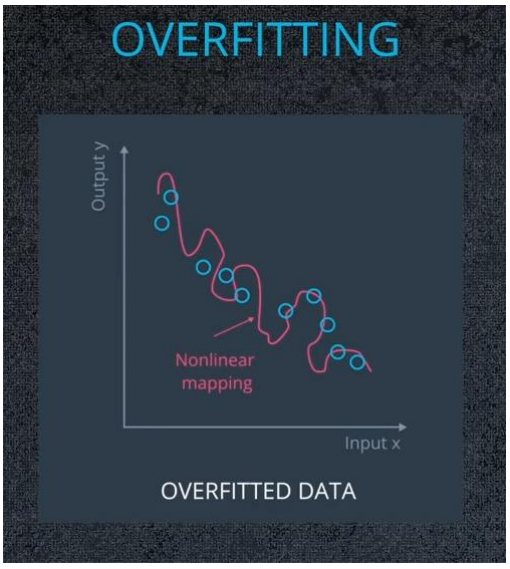
When we minimize the network error using backpropagation. We may either probably fit the model to the data or overfit.

Generally speaking, when we have a finite training set, there`s risk overfitting.



Overfitting means that our model will fit the training data too closely.

In other words, we over trained the model or the network to fit our data. As result we unintentionally also model the noise or random elements of the training set.

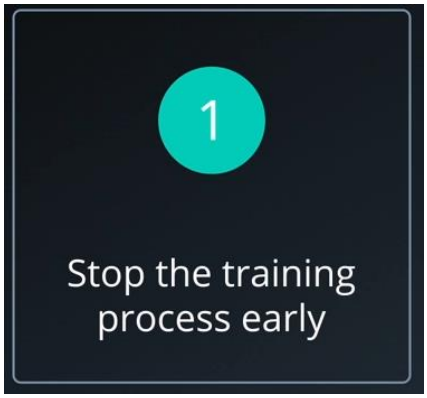


The model will not generalize well

If that happens, our model will not generalize well when tested on new inputs.

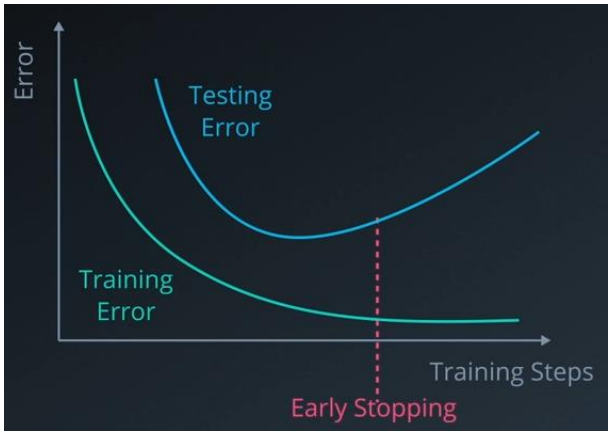
There are two generally two main approaches to addressing the overfitting problem.

1- **The first is to stop the training process early,**
2- **And Second is to use of regularization.**



When stop the training process early, we do that in the region where the network begins to overfit. By doing so, we reduce degradation in the performance on the test set.

It would be ideal if we knew precisely when we should stop the training processes. However, that is often difficult to determine.



One way to determining when to stop the training is by carving a small data set out of the out of the training which we will call the validation set.

Assuming that the accuracy of the valiation set is simliar to that of the test set, we can use it to estimate when the training should stop.

The drawback of this approach is that we end up with fewer samples to train our model on, so our training set is smaller.

An Alternative mainstream approach to mitigating overfitting is to use regularization.

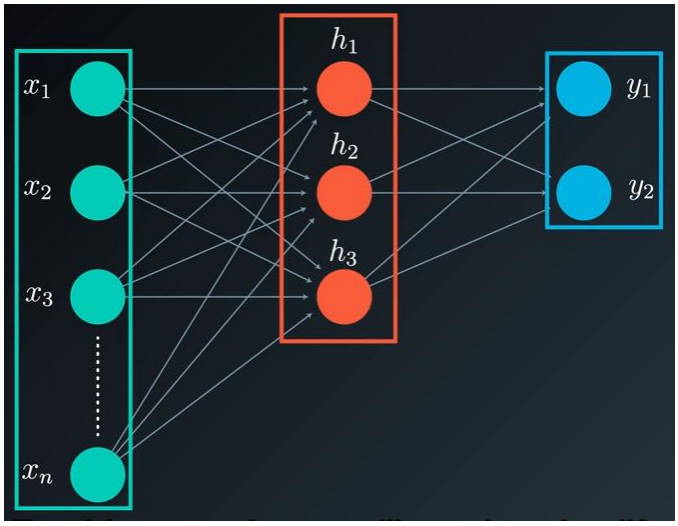
Regularization means that we impose a constraint on the training of the network such that better generalization can be achieved.

Dropout is widely used regularization scheme which helps in the that manner.

Backpropagation

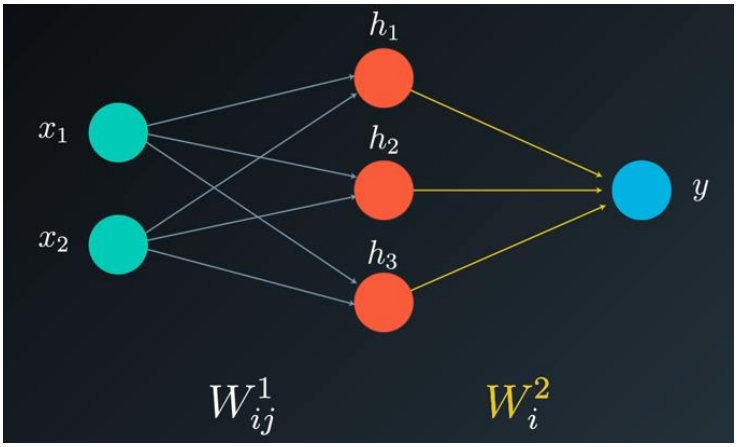
Remember our feedforward

We had n inputs, three neurons in the hidden layer , and two outputs.



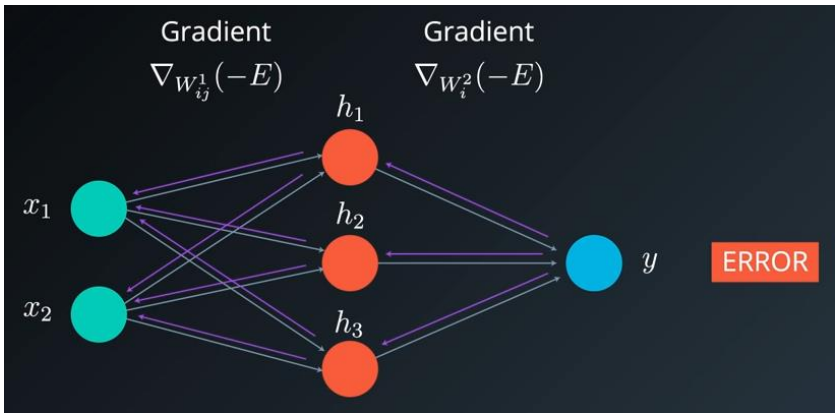
For this example, we will need to simplify things even more and look at a model with two inputs,

X_1 and x_2 and a single output y, we will have a weight matrix , W_1 from the input to the hidden layer, the element of the matrix will be W_{ij} just as before. We also have a weight matrix vector W_2, from the hidden layer to the output.

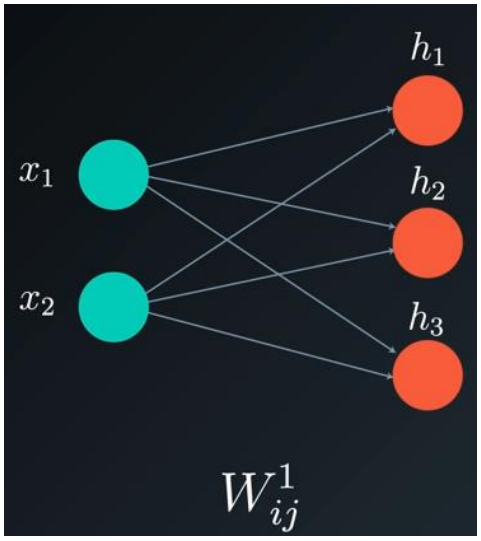


Notice that it`s a vector and not a matrix as we only have one output. Don`t forget your pencil and notes.

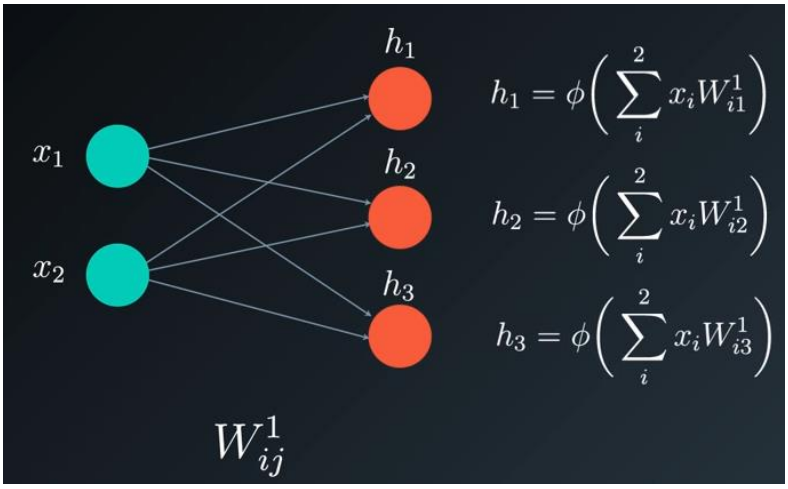
We will begin with a feedforward pass of the inputs across the network, then calculate the output, and based on the error use backpropagation to calculate the partial derivatives.



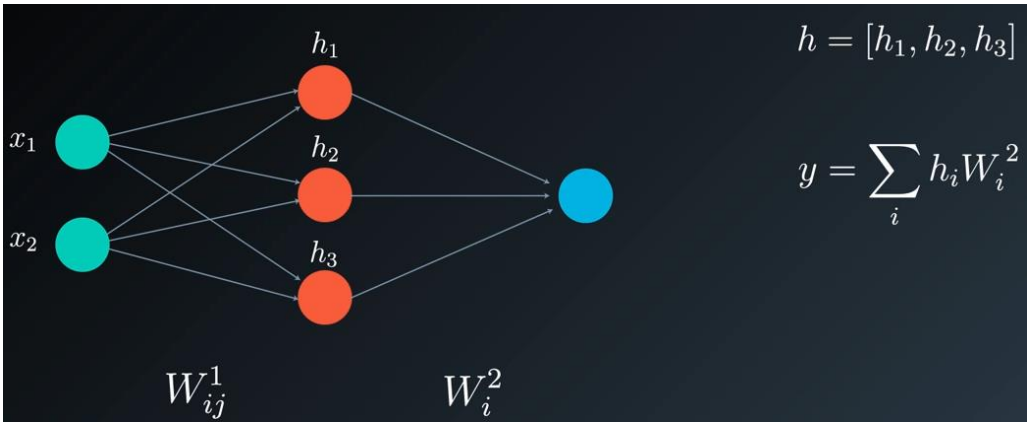
1- Calculating the values of the activations at each of the three hidden neurons is simple.



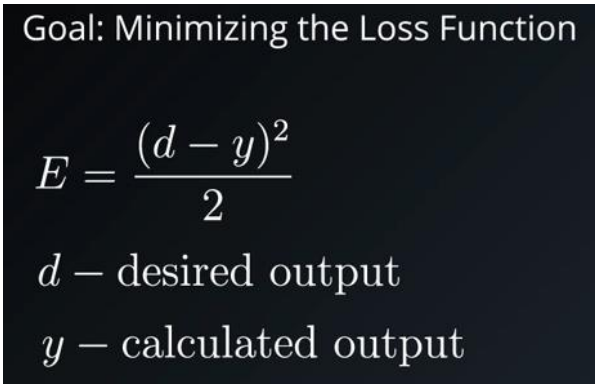
2- We have a linear combination of the inputs with a corresponding weight element of the matrix W_1. All that is followed by an activation function.



3- The output are a dot product of the activation of the previous layer vector H with the weights of W2.



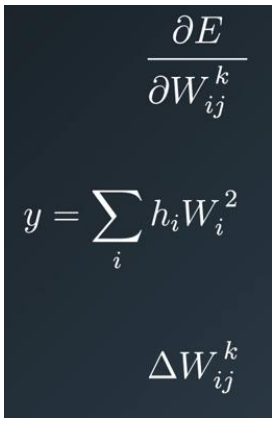
As I mentioned before, the aim of the back propagation process is to minimize, in our case , the loss function or square error.



To do that, we need to calculated the partial derivatives of the square error with respect to each of the weights.

Since we just found the output, we can minimize the error by finding the updated values of delta of

W_ij and of course on every case, we need to do so for every single layer K.



The value equals to the negative of alpha multiplied by the partial derivative of the loss function e.

Since the error is polynomial finding its derivative is immediate by using basic calculus, we can seek that this incremental value is simply the learning rate alpha multiplied by d minus y and the partial derivatives of y with respect to each of the weights.

If you asking yourselves, what happen to the desired output D? Well, that was a constant value,

So it`s partial derivatives was simply a zero.

$$\Delta W_{ij} = -\alpha \frac{\partial E}{\partial W_{ij}} = -\alpha \frac{\partial \frac{(d-y)^2}{2}}{\partial W_{ij}} \quad => \quad \Delta W_{ij} = \alpha (d-y) \frac{\partial y}{\partial W_{ij}}$$

$$E = \frac{(d-y)^2}{2}$$

Notice that I used the chain rule here? in the beginning this video I mentioned that back propagation is actually stochastic gradient descent with the use of the chain rule. Well know you have a gradient.

$$\text{Gradient} \quad \delta = \frac{\partial y}{\partial W_{ij}}$$

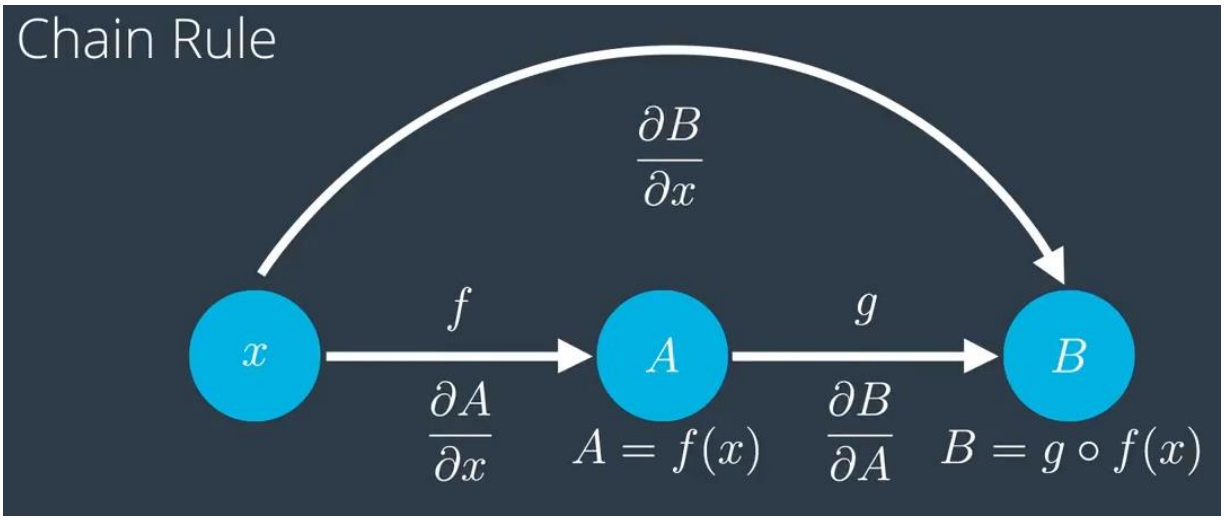
1- **The symbol that we use for the gradient is usually lowercase delta.**
2- **The partial derivative of the calculated output defines the gradient and we will find it by using the chain rule.**

Remain themselves what the chain rule is and how it`s used. When comfortable with actual calculation.

Rule chain

So we before starting the derivative let`s do a refresher on the chain rule which is the main technique we will use to calculate them.

The chain rule says, if you have a variable x on function f that you apply to x to get f of x, which we are going call A, and the another function g, which to apply to f of x to get g of f of x , which we are gonna call B the chain rule says, if you want to find the partial derivative of B with respect to x that`s just a partial derivative of B with respect to A Times the partial derivative of A with respect to X.



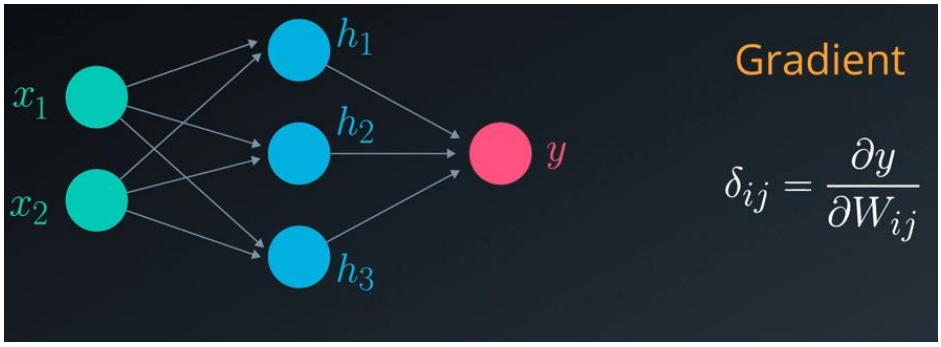
So it literaly says, when composing functions, that dervatives just muliplty, and that`s gonna be super for us becaues feed forwarding is literaly composing a bunch of functions, and back propagation is literaly talking the derivatives at each piece and since talking the derivaitve of a composition is the same as mutliplying the partial derivatives, then all we are gonna do is mutliply a bunch of patial derviatives to get what we want Pretty simple right

$$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A}\frac{\partial A}{\partial x}$$

Backpropagation- Example

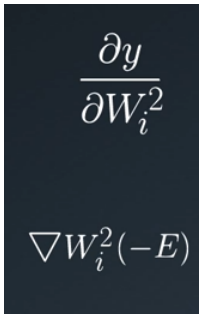
We need to calculated the gradient we will do that one step at a time.

In our example we only have one hidden layer. So back propagation process will have two steps.

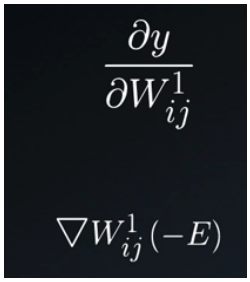


Let`s be more precise now and decide that the gradient calculated for each element IJ in the matrix is called delta of ij.

1- In step number one
   a. **We calculated the gradient with respect to the weight vector W2 from the output to the hidden layer.**



2- And in Step two
   a. **We calculated the gradient with respect to the weight matrix W1 from the hidden layer to the input.**



$$\text{Find } \frac{\partial y}{\partial W_i^2}$$

Ok let`s start with step one we already calculated y.

$$y = \sum_i^3 h_i W_i^2$$

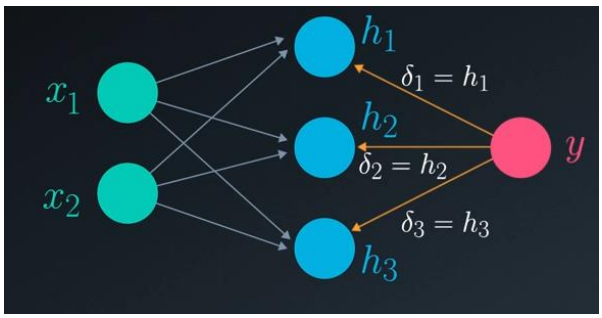
We will find its partial derivative with respect to the weight vector W2.

$$\frac{\partial y}{\partial W_i^2} = \frac{\partial(\sum_i^3 h_i W_i^2)}{\partial W_i^2} = h_i$$

Given that y is a linear summation over terms you will find that the gradient is simply the value of the corresponding activation h, as all the terms were zero.

This is hold for

1- **Gradient one**
2- **Gradient two**
3- **Gradient three**

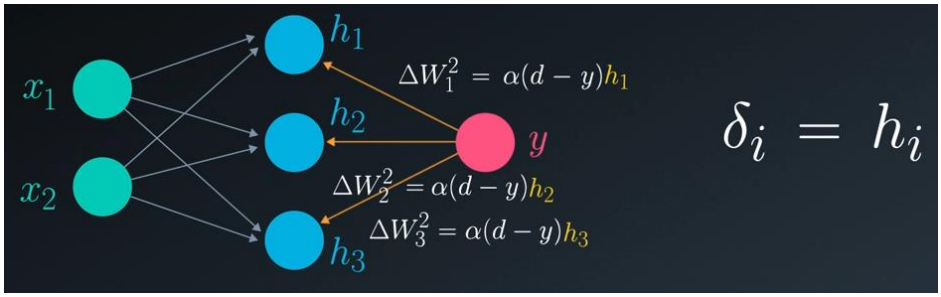


You probably noticed that we have only one index for delta and that`s because we have a single output.

Previously, we saw that the incremental value delta of W_{ij} equals to the learning rate alpha multiplied by d minus y and multiplied by the gradient.

$$\Delta W_i = \alpha(d-y)\delta_i$$

So at the second layer, the incremental value of delta of Wi equals alpha multiplied by d minus y and by h_i.



In step two

We want to update to weights of layer one by calculating the partial derivative of y with respect to the weight matrix W1.

Here is where things get a little more interesting. When the calculating the gradient, with respect to weight matrix W1 we need to chain rule.

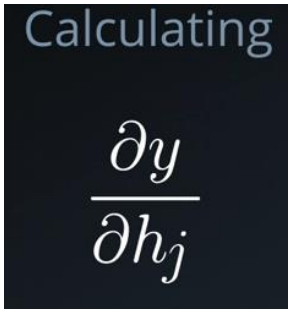
The approach, obtain the partial derivative of y with respect to h and multiply it by the partial derivative of h with respect to the corresponding elements in W1.

In this example, we only we have three neurons in the single layer hidden layer.

Therefore, this will be a linear combination of three elements.

Step 2

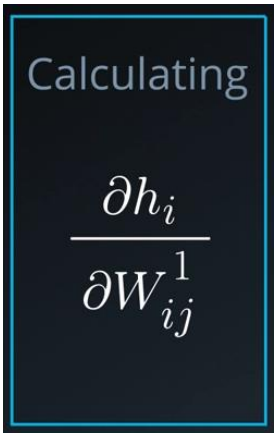
Let`s calculated each derivative separately.



Since y is a linear combination of h and its corresponding weights, its partial derivative with respect to h will be weight elements a vector W2.

$$\frac{\partial y}{\partial h_j} = \frac{\partial \sum_{i=1}^{3}(h_i W_i^2)}{\partial h_j} = W_j^2$$

Now, what is the partial derivative of each elements of vector h with respect to its corresponding weights in matrix W1?



We have already considered each element of vector h separately.

1- **So here is h1**
2- **We also have h2**
3- **And h3**

$$h_1 = \phi(\sum_{i}^{2} x_i W_{i1}^1) \qquad h_3 = \phi(\sum_{i}^{2} x_i W_{i3}^1)$$

$$h_2 = \phi(\sum_{i}^{2} x_i W_{i2}^1)$$

If we generalize this, each element j is an **activation function** of a corresponding linear combination.

Finding its partial derivative means finding the partial derivative of the activation function and multiplying it by the partial derivative of the linear combination.

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} \frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

All of course, with respect to the correct elements of the wights W1.

Feel free to pause anytime you need to catch up on your notes.

As I said before, there are various activation functions.

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} \frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

So let`s just call the partial derivative of the activation function f1 prime.

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j(\sum_{i=1}^2 (x_i W_{ij}^1))}{\underbrace{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))}_{\Phi_j'}} \frac{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))}{\partial W_{ij}^1}$$

Each neuron will have its own value of fi and fi prime, according to activation function you use.

The partial derivative of the linear combination with respect to W_{ij} is simply x1 since all of the other component are zero.

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j(\sum_{i=1}^2 (x_i W_{ij}^1))}{\underbrace{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))}_{\Phi_j'}} \frac{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))}{\underbrace{\partial W_{ij}^1}_{x_i}}$$

So the partial derivative of h with respect to the weight matrix W1 is simply fi prime at neuron j multiplied by x1.

$$\frac{\partial h_j}{\partial W_{ij}^1} = \Phi_j' x_i$$

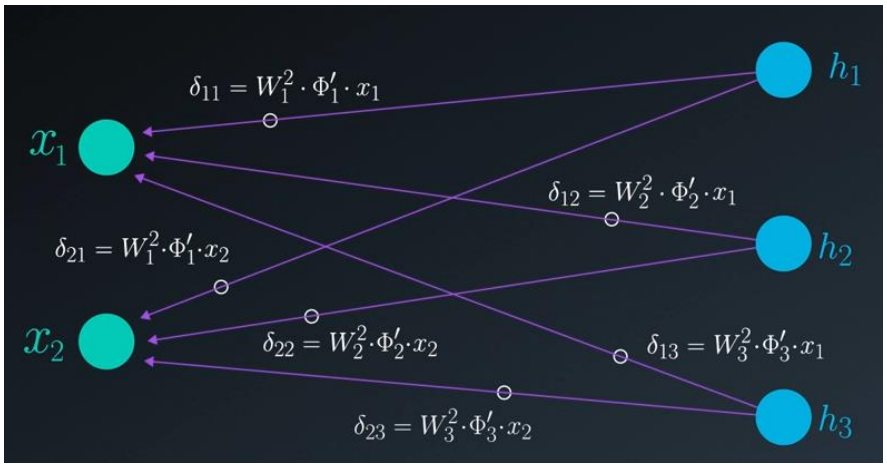
We now have all the pieces required for step number two, given us the gradient delta of ij.

We know that the gradient of the output y with respect to each element in matrix W1 is the multiplication of these two partial derivatives that we just calculated.

Since in the example,

We have two inputs and three hidden neurons, we will have six gradients to calculate,

1- **delta one one**
2- **delta one two**
3- **delta one three**
4- **delta two one**
5- **delta two two**
6- **delta two three**



1- **After finding the gradient in step two**
2- **Finding the incremental value of W_{ij} is immediate.**

$$\Delta W_{ij} = \alpha \cdot (d - y) \cdot \delta_{ij}$$

Again , in the case of the loss function that we are using here, it`s simply the multiplication of the gradient by the alpha and by d minus y

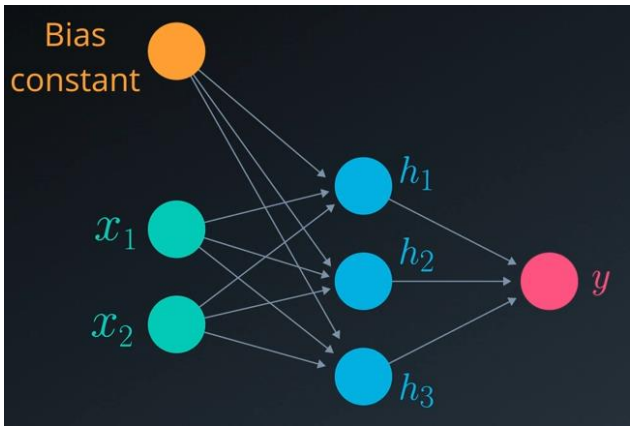
And the end the back propagation part, each element in the weight matrix can be updated by the incremental values we calculated using these two steps.

If you have more layers which is usually the case, we will have more steps.

$$W_{new}^1 = W_{previous}^1 + \Delta W_{ij}^1$$

You can imagine that the process becomes more complicated. Luckily we have the programming tools for that.

In all of these calculations we did not emphasize the biased input as it does not change any of these concepts we covered.



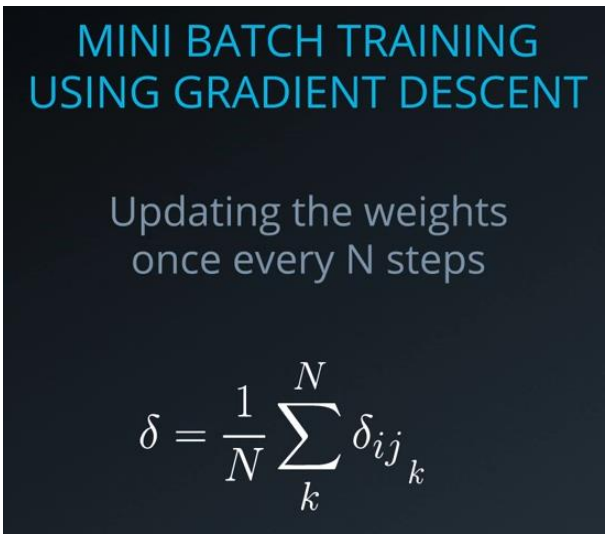
As I mentioned before, simply consider the bias is a constant input that is also connected to each of the neurons of the hidden layers by weigh.

The only difference between the bias and the other inputs is the fact that it remains the same as each of the other inputs change.

In this example, for each new input we updated the weights after each calculation of the output.

It is often beneficial to update the weights once every and steps

This is called mini batch training and involves averaging the changes to the weights over multiple steps before actually updating the weights.



There are two mini primary reasons for using mini batch training.

1- **The first is to reduce the complexity of the training process since fewer computations are required**
2- **The second and more important is that when we average multiple possibly noisy changes to the weights, we end up with a less noisy correction.**

This means that the learning process may actually converge faster and more accurately.

Recurrent neural network

Everything we have seen so far prepared us for this moment.

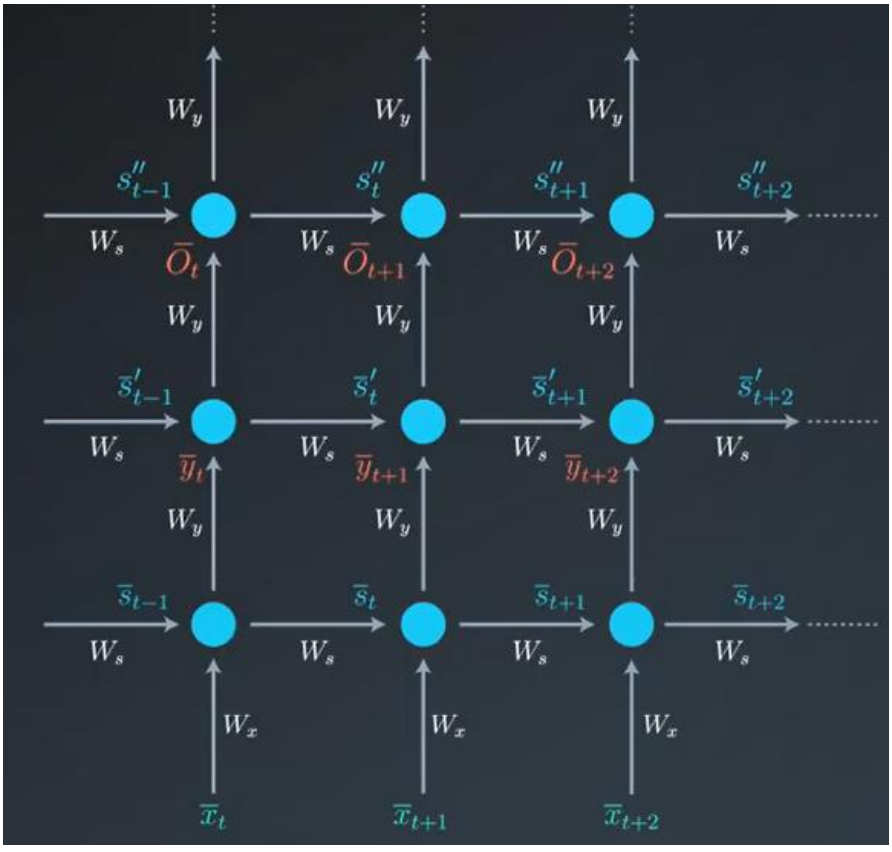
We want we feedforward process, as well as back propagation process in much detail.

As I mentioned before, if you look up the definition of the word Recurrent, you find simply that it simply means occurring often or repeatedly So why are these networks called Recurrent Neural Networks?

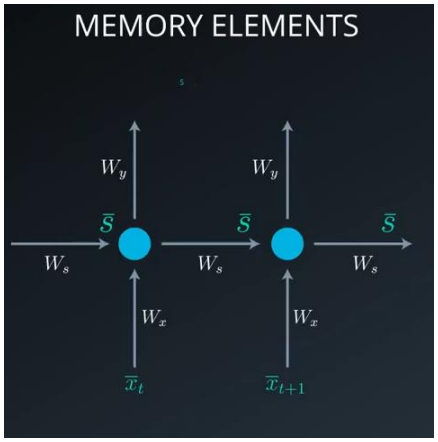
It simply because with RNN`s, we perform the same task for each element in the input sequence.

We will see a lot more of this sketch later.

RNN`s also attempt to address the need for the capturing information and previous inputs by maintaining internal memory elements,

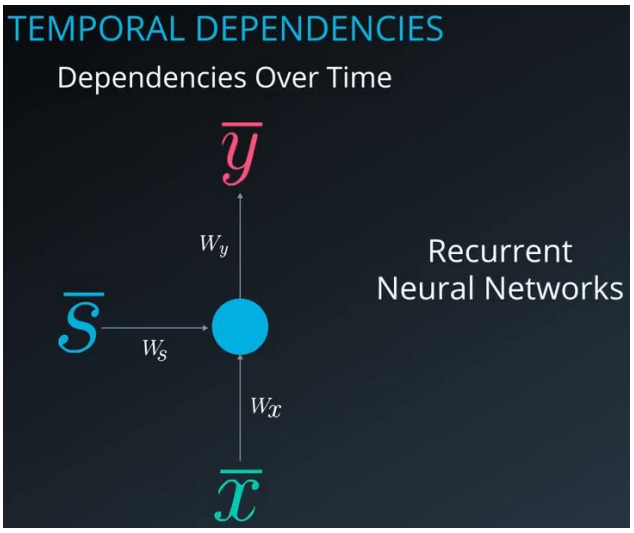


By maintaining internal memory elements, also known as State.



Many applications have temporal dependencies.

Meaning, that the current output depends not only on the current input, but also on a memory which takes into account past inputs for cases like this we need the recurrent neural networks.



RNN

A good example for the use of RNN is predicting the next word in a sentence, which typically requires looking at the last few words rather than only the current one.

We also mentioned quite a few other categories of applications

1- **Sentiment analysis**
2- **Speech Recognition**
3- **Time Series Prediction**
4- **Natural Language Processing**
5- **Gesture Recognition**

Frankly, applications of our RNN`s are propping up almost everyday making it challenging to keep up.

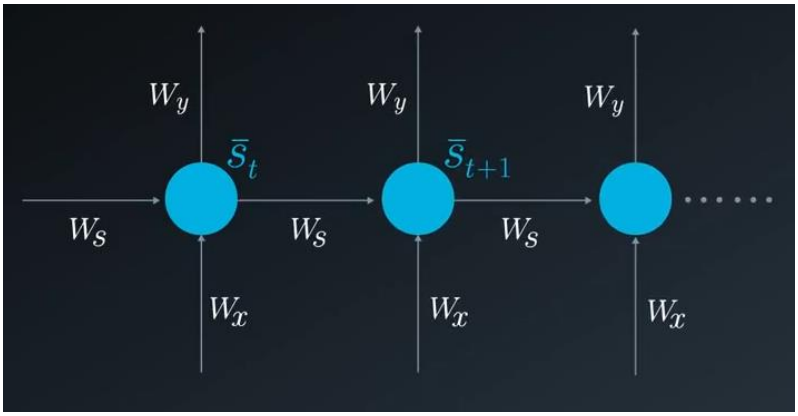
So how should we think about this new neural network?

How are the training, and evolution phases changed?

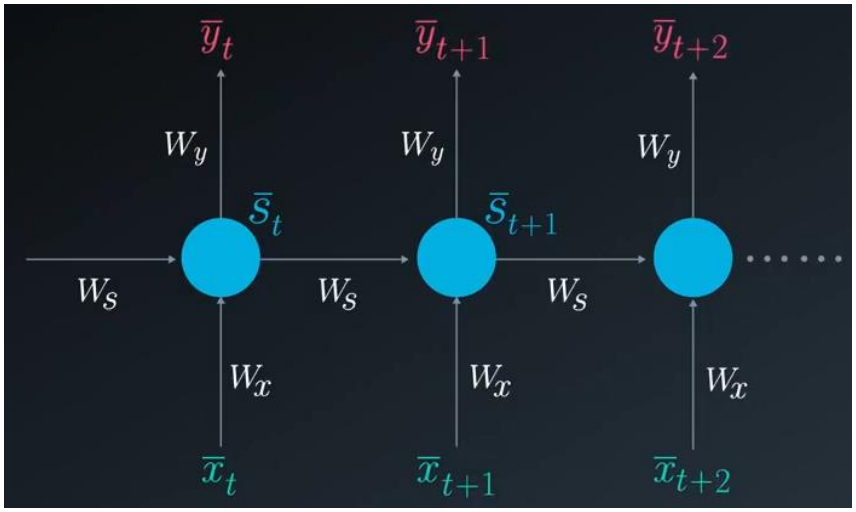
Well RNN`s are based on the same principles behind feedforward neural networks, Which is why we spend so much time reminding ourselves of the latter, and feedforward neural network.

The first, is the manner by which we define our inputs and outputs.

Instead of training the network using a single-input, single-output at each time step,

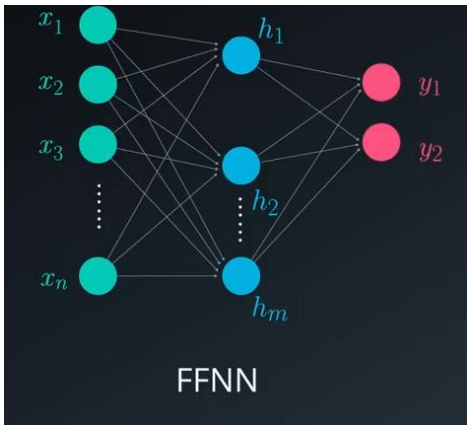


We train with sequence since previous input matter.



The second difference, stems from the memory elements that RNN`s host. Current inputs, as well as activation of neurons serve as inputs to the next time step. S

in feedforward neural networks, we saw a flow of information from the input to the output to the output without any feedback.



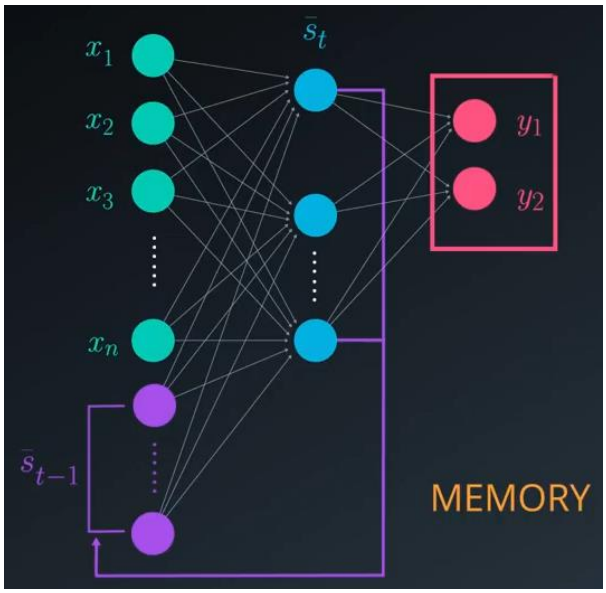
Now, that feedforward scheme changes, and includes the feedback or memory elements.

We will consider memory defined, as the output of the hidden layer, which will serve as an additional input to the network at the following training step.

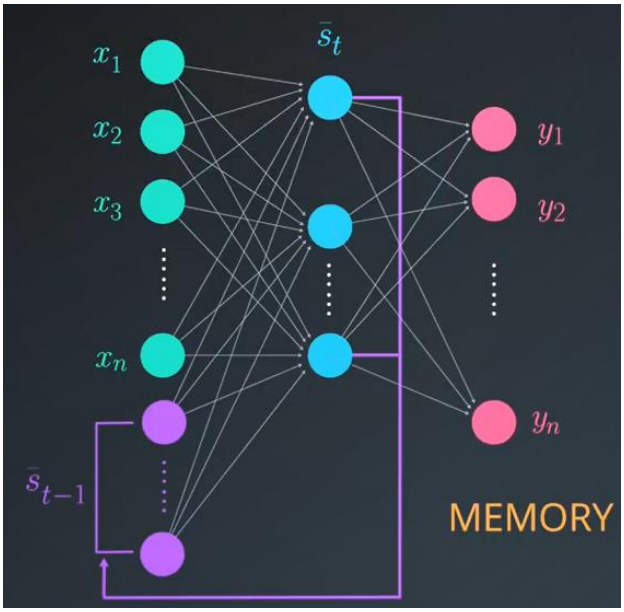
We will no longer use H as the output of the hidden layer, but S for state , referring to a system with memory.

The basic schemes of RNN is called simple RNN and also known as an Elman Network.

Notice that in this illustration, I only used two outputs.

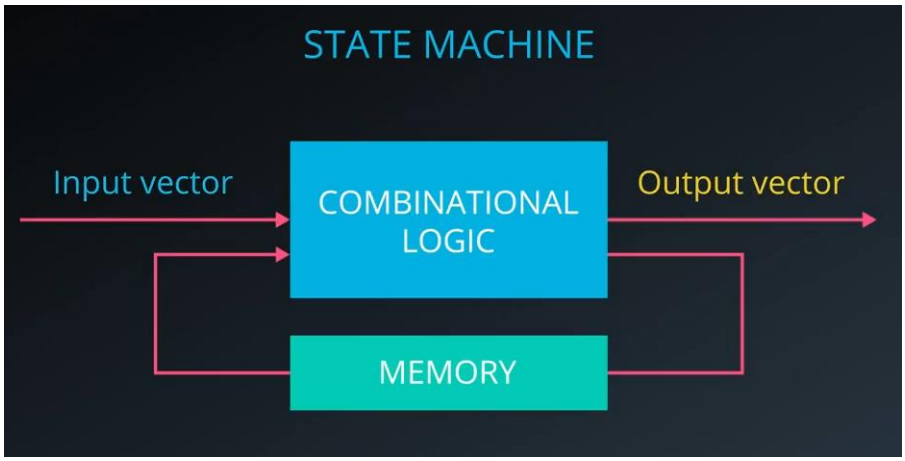


Well, you can have many outputs as well, certainly more than two. But to simplify the sketch a bit, let`s just stay with two or now.



For those of you who come from engineering or computer science backgrounds, this will probably remind you of a simple state machine with combination of logic and memory.

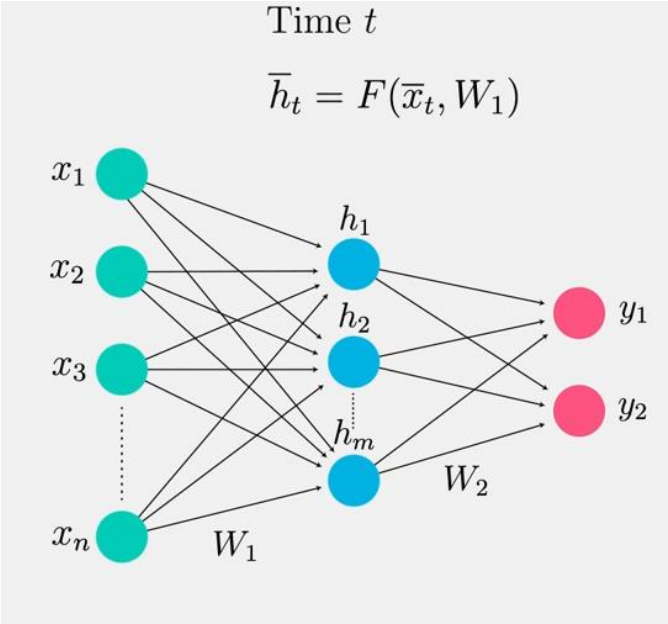
The outputs depend not only on the external inputs, but also on previous inputs which come from memory cells.



The RNN is conceptually similar.

But the beauty in our case is the system will train itself and learn how to optimize the weight matrix to realize the network.

In Feedforward Neural network the output at any time is a function of the current input and the weights alone.



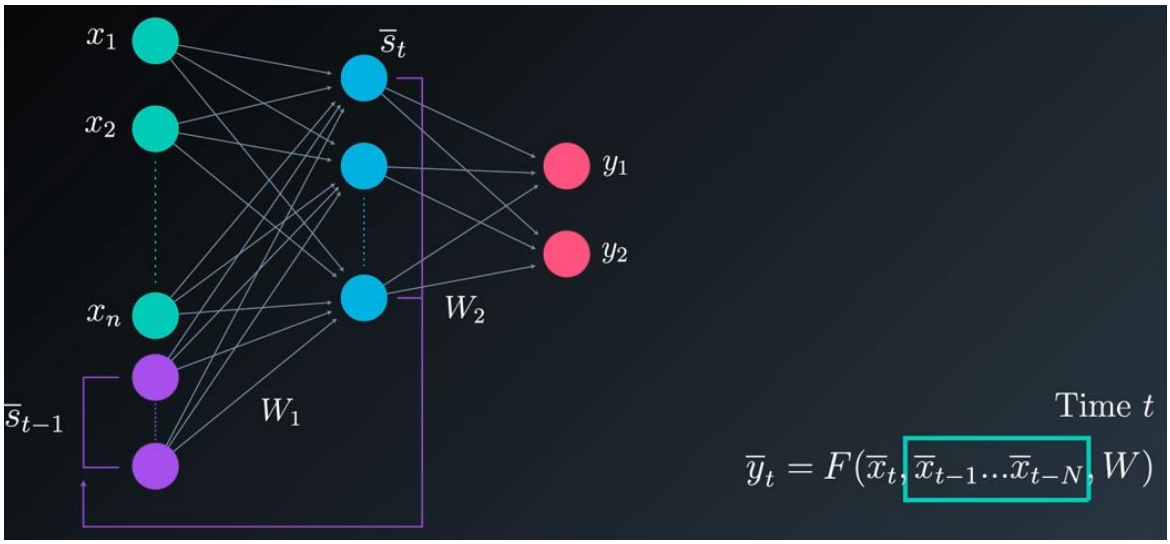
We assume that the inputs are independent of each other. Therefore, there is no significant to the sequence.

So we actually train the system by randomly drawing inputs and target pairs

In RNNs, our output at time t, depends not only on the current input and weights, but also on pervious inputs.

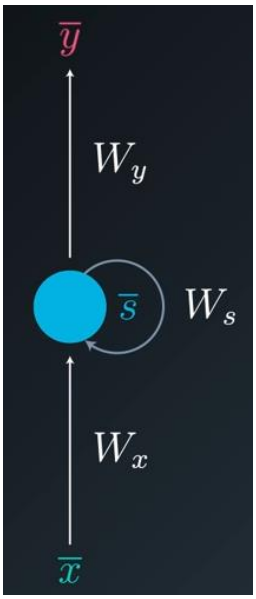
Such as the input to Time T minus 1 , Time minus 2 so on.

Let`s look at our neural network again.



RNN

We have an input x and output y and a hidden layer s. Remember S? it stand for State which is a term we use when system has memory. W_x represent the weight matrix, connecting the inputs to the state layer. W_y represent the weight matrix, Connecting the state to the output and W_s represent the weight matrix, Connecting the state from the previous timestep to the state in the next timestep.



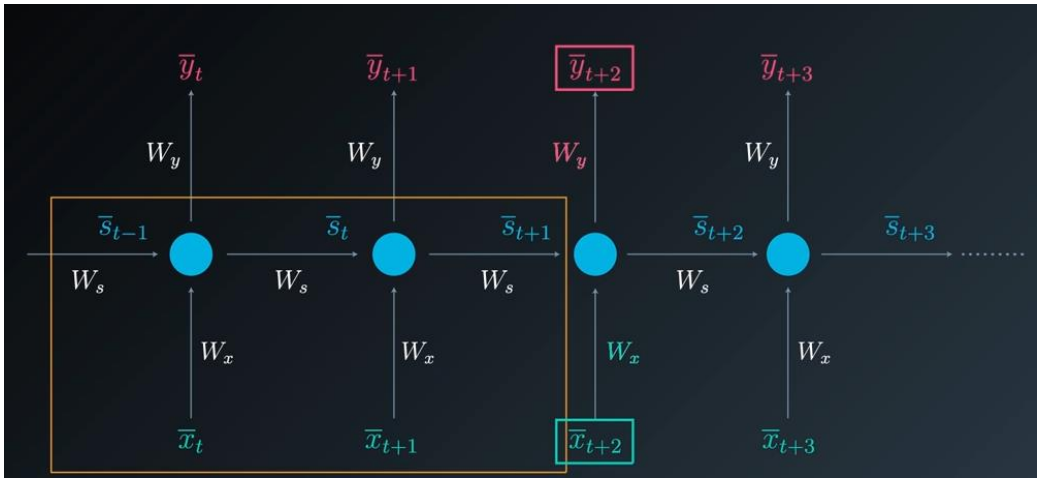
Notice that the single s is fed back to the system. In every timestep, the system will look the same.

This is what we call the folded model.

Since the input is spread over time and we performed the same task for every element in the sequence, we can unfold the model in time and represent it the following way.

Just as an example, you can see that the output at Time T plus 2 , depends on the input at time T plus 2 on the way matrices and also on all previous inputs.

"Unfolded Model "

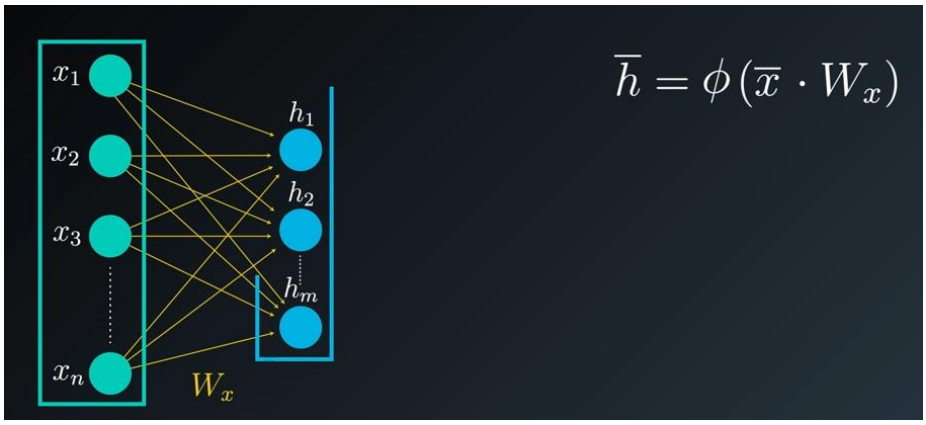


Let`s use the following definitions,

1- **x of t is the input vector at time t**
2- **y of t is the output vector at time t**
3- **s of t is the hidden state vector at time t.**

In feed forward neural networks , we use the activation function to obtain the hidden layer h.

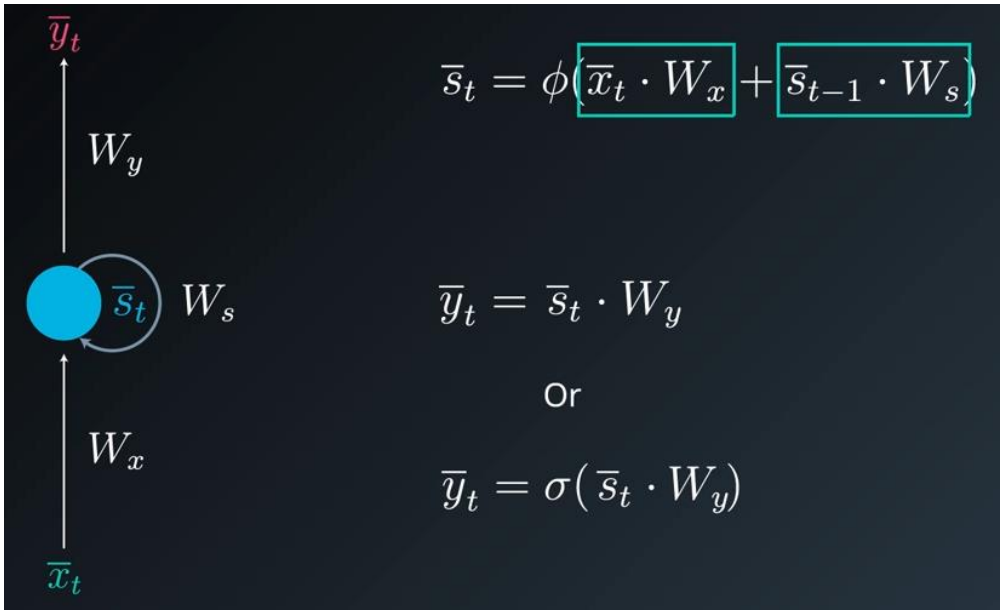
And all we need are the input and the weight matrix connecting the inputs to the hidden layer.



In RNNs, we use an activation function to obtain s, but with a slight twist the input to the activation function is now the sum of one, the product of the inputs and their and corresponding weight matrix, Wx and two, the product of the previous activation values and their corresponding weight matrix Ws.

The output vector is calculated exactly the same as in feedforward neural networks. It can be a linear combination of the inputs to each output node with a corresponding weight matrix Wy or for

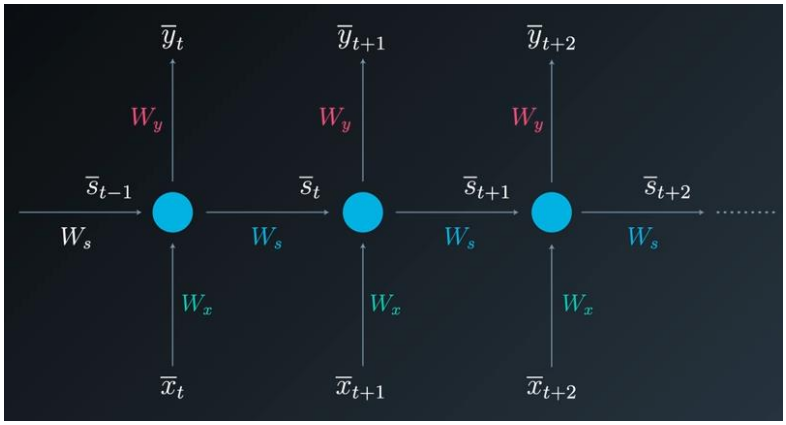
Example a SoftMax function of the same linear combinations.



Notice, that RNNs share the same parameters during each timestep. So although intuitively, it appears that RNNs are more complicated than feedforward neural networks since they have a memory.

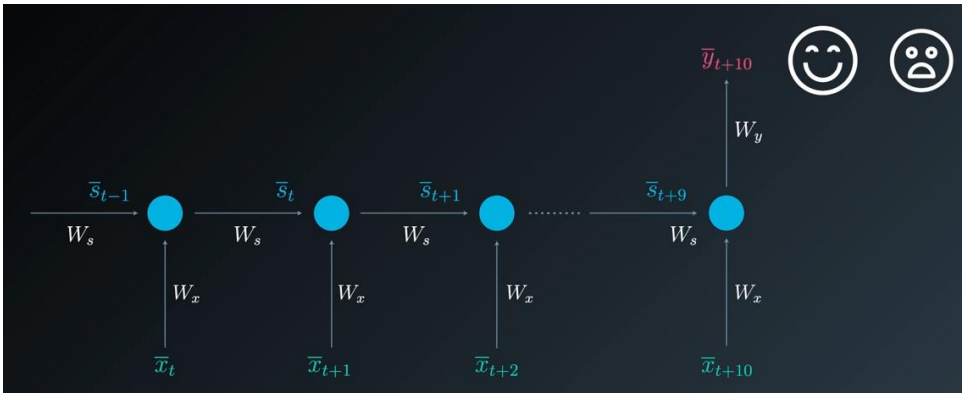
In practice, the number of parameters that need to be learned remains modest.

The unfolding scheme we mentioned is generic and we can be adjusted according to the neural network architecture we aim to build.

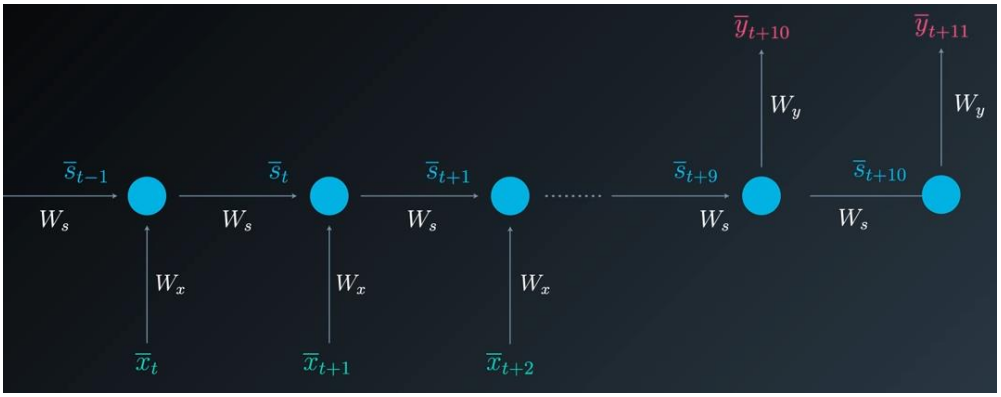


We can decide how many inputs and outputs we need, For example in sentiment analysis,

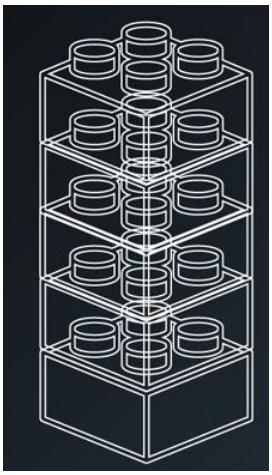
We can have many inputs and a single output that spans the spectrum of happy to sad.



Another example can be time series predictions, where we have many input and many outputs which are not necessarily aligned.

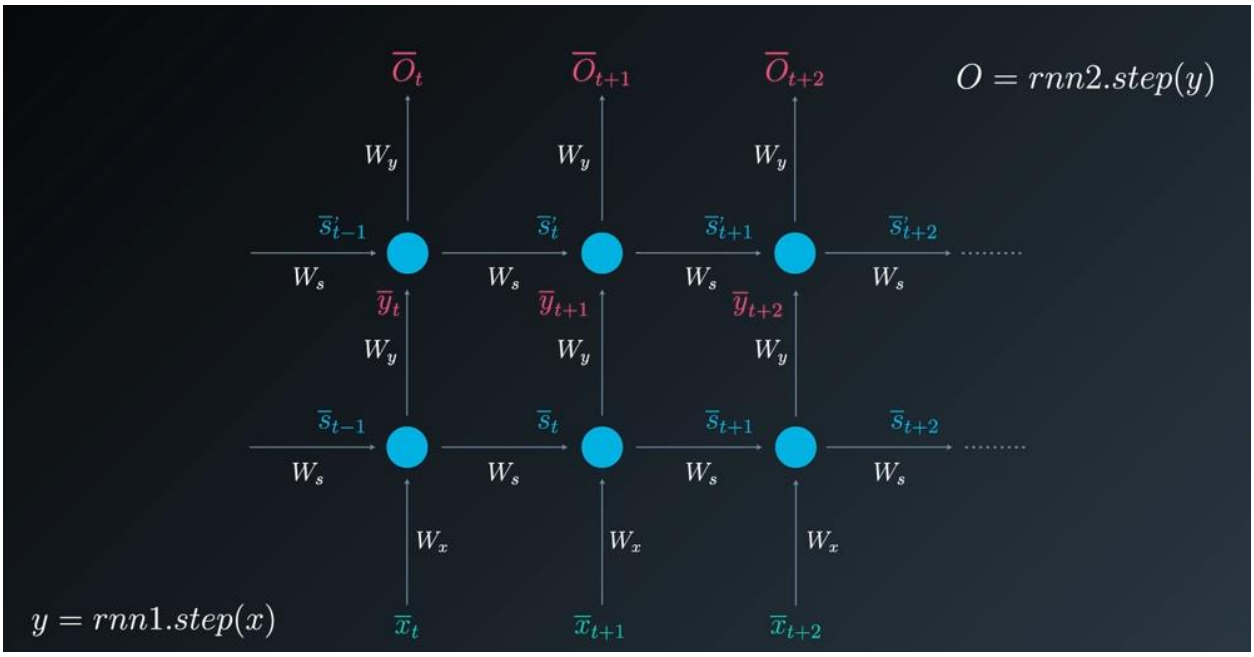


RNNs can be stacked like Legos, just as feedforward neural networks can.

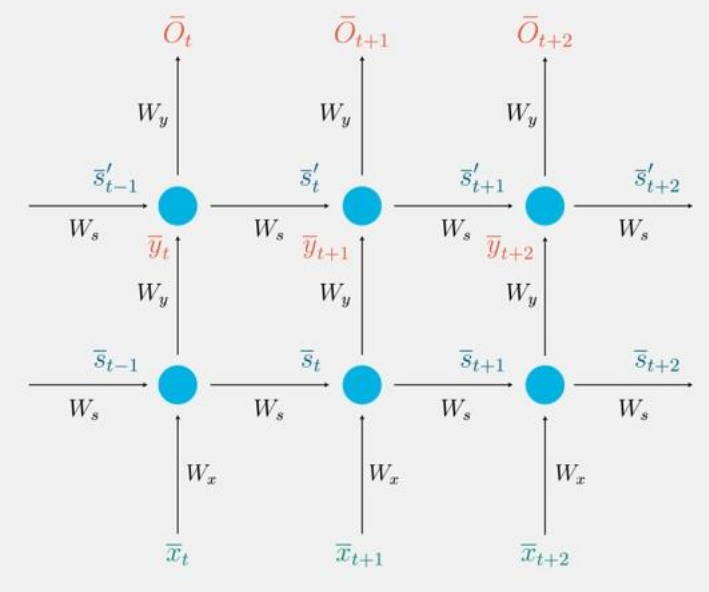


For example, We can have the output of a single RNN level, say vector y become the input to a second layer whose is vector o.

Each layer operates independently of the other layers, as architecturally it doesn`t matter where the inputs come from or where the outputs are headed to.



The unfolding Time scheme can be confusing So let`s go back for a bit, Look at it closely,



And see what`s actually going on there. First, we will talk the Elman network, and titled by 90 degrees counter-clockwise.

As in RNN we usually display the flow of information from the bottom to the top.

In the case of a single hidden layer without stacking further, this is how the unfolded model may look like.

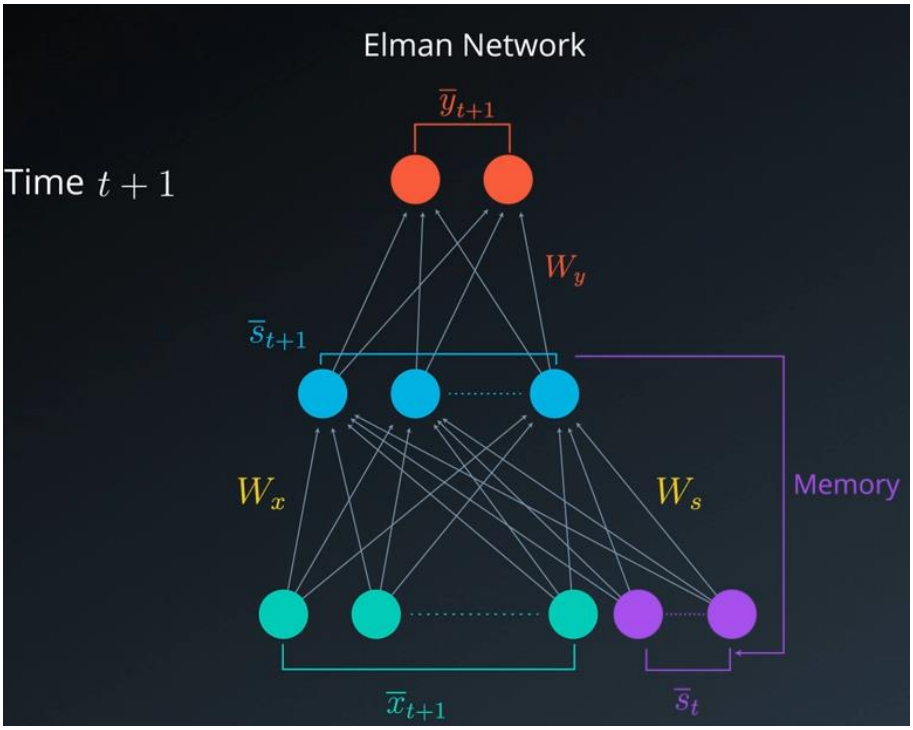
At any given time t, we will have input an vector X to t, connected to the state by the weight matrix WX.

State vector S of t minus one, Connected to the state by the weight matrix WS, X of t and s of t minus to gather, helped produce the desired state vector S of t, weight matrix WY in return helps produce the output vector Y of t.

Now at time t plus one the system will have two different input vectors,

1- **vector X of t plus one,**
2- **and previous state vector S of t, which was our hidden layer activation in the previous time step.**

But the weights remained the same as it`s the same system, only a different time.

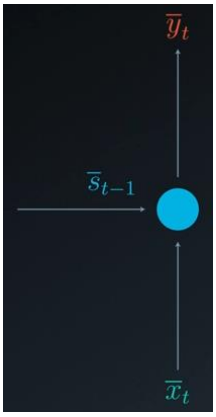


To avoid messy Sketches, we can use the unfolded scheme which is visibly cleaner with fewer connecting lines.

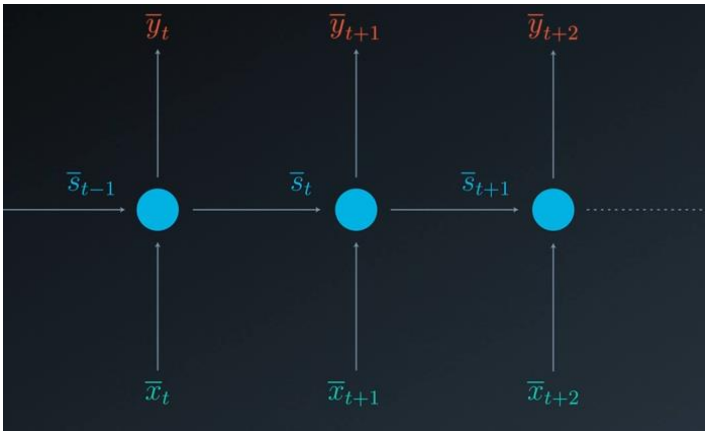
So to present the system time t plus one and so on, we can simply connect it this way.

Add the weights to the sketch, and we have our complete unfolded system where each arrow represents a few if not many variables.

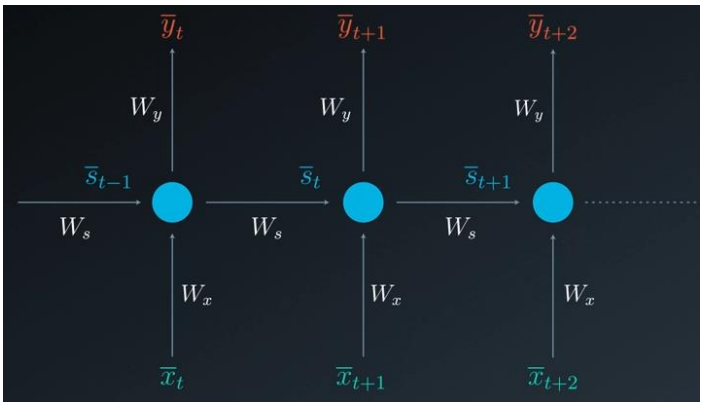
Step 1



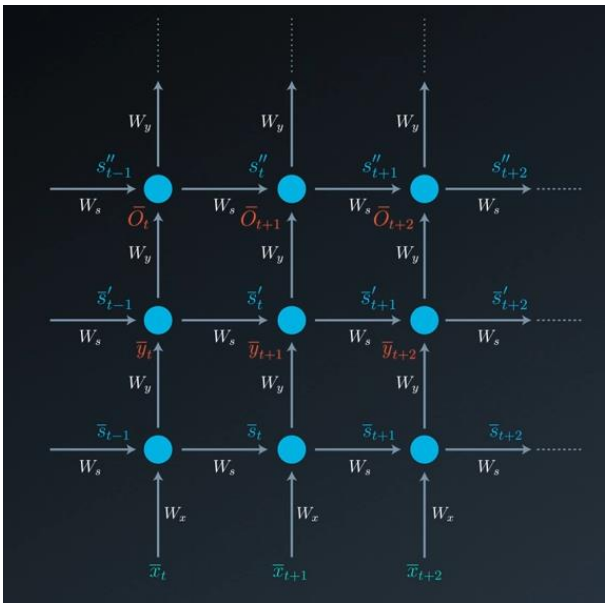
Step 2



Step 3 >> Add the weights in the sketch



Understanding this principle, and realizing that we can stack up any arbitrary number of layers, enables the easy sketching of RNN networks of any number of layers.

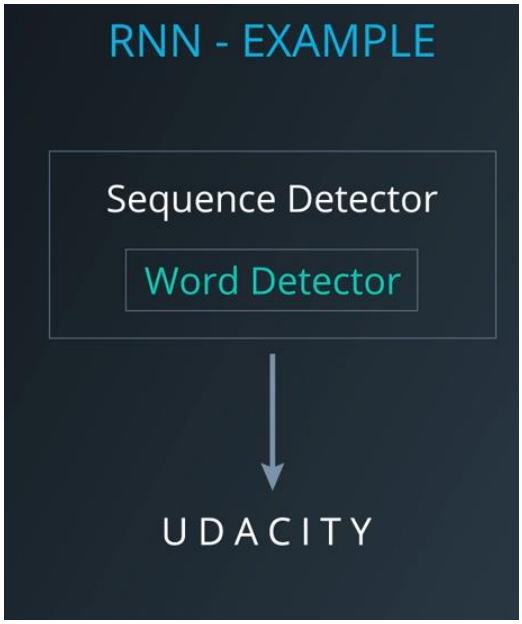


Let`s continue with conceptual RNN example

Assume that we want to build a sequence detector, and let`s decide that our sequence detector will track letters.

So we actually build a word detector.

And more specifically we want our network to detect the word, Udacity



So we before we start, we need to make a few decisions.

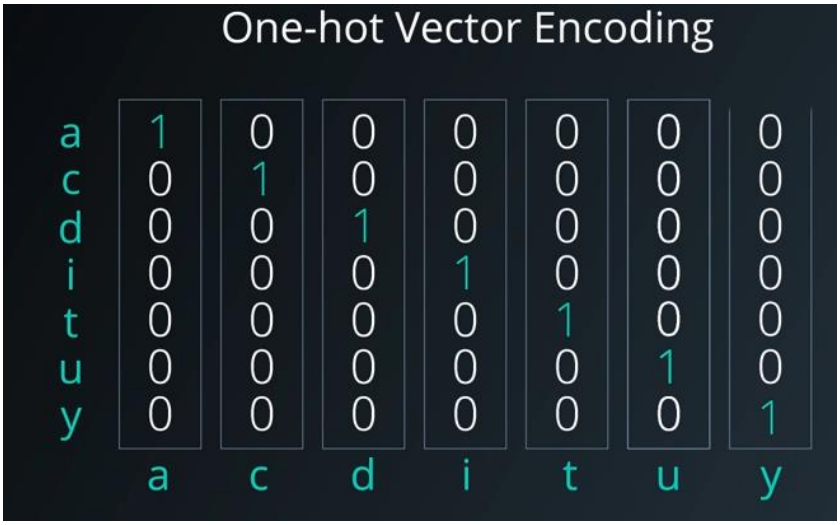
1- **We can define the input by using a one-hot vector encoding, containing seven binary values.**

Here I use the letters in ascending order, but that`s arbitrary.

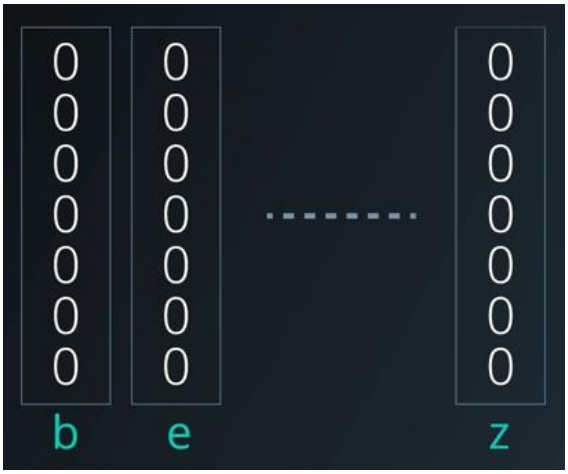
Each letter will be represented by one in the index it corresponding to, and zero everywhere else.

So for example, the letter A will be represented by one followed by six zeros

Here are the rest of the letters that we need.



The letters not appearing in the word Udacity can be addressed by a simple vector containing all zeros, Like these letters, for example.



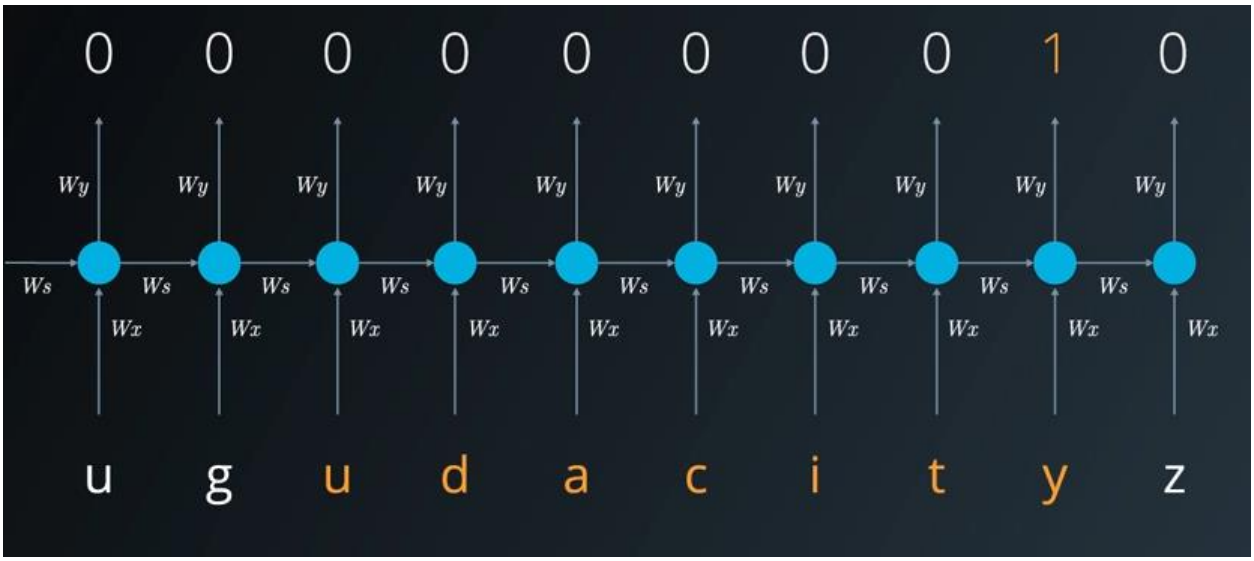
The sequencing we are trying to detect will be of length seven with inputs in the following order:

1- **U**
2- **D**
3- **A**
4- **C**
5- **I**
6- **T**
7- **Y**

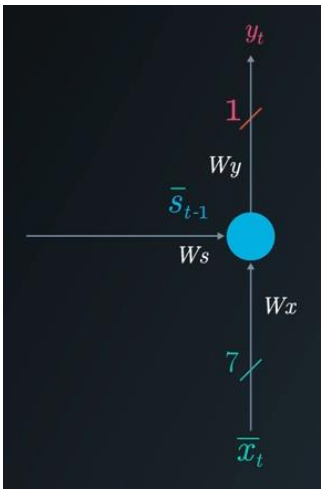
We can train the system by feeding it random letters at each timesteps.

Creating a sequence of inputs demonstrated here from left to right. Occasionally, we will also insert the word Udacity. We set the target values, the outputs to the zero all the time except for when the last letter Y of the sequence Udacity enters the system. Only then the target will be set to one.

Again, the system will acknowledge all inputs and will respond with a target output of one only when the desired sequence is detected.



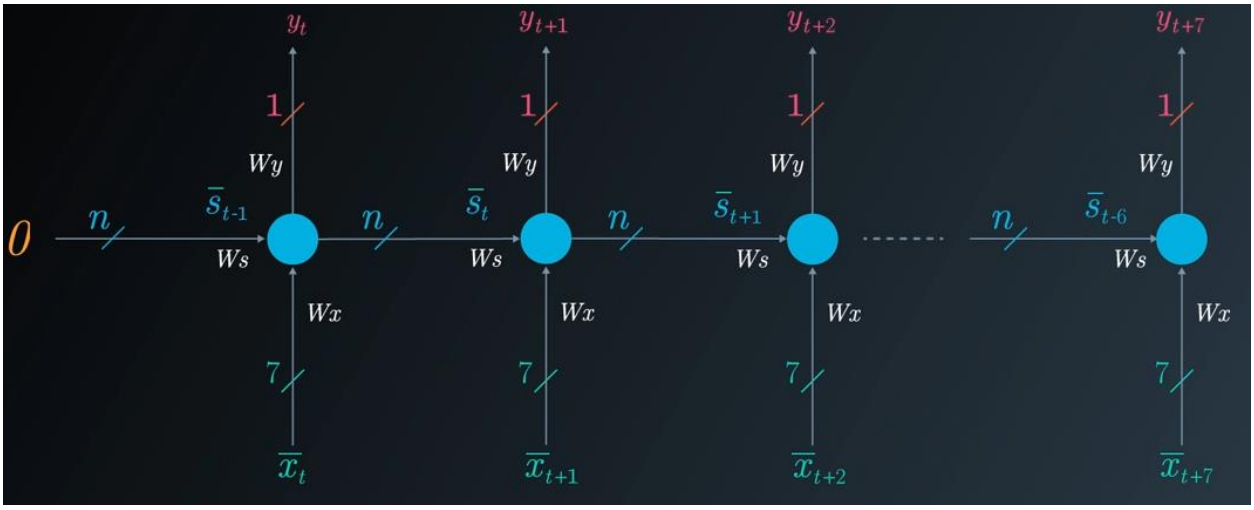
Sketching our network in an unfolded form we have an input vector of seven values a single output and a state.



The state can have any number of hidden neurons.

In the illustration, we will use n to leave things generic.

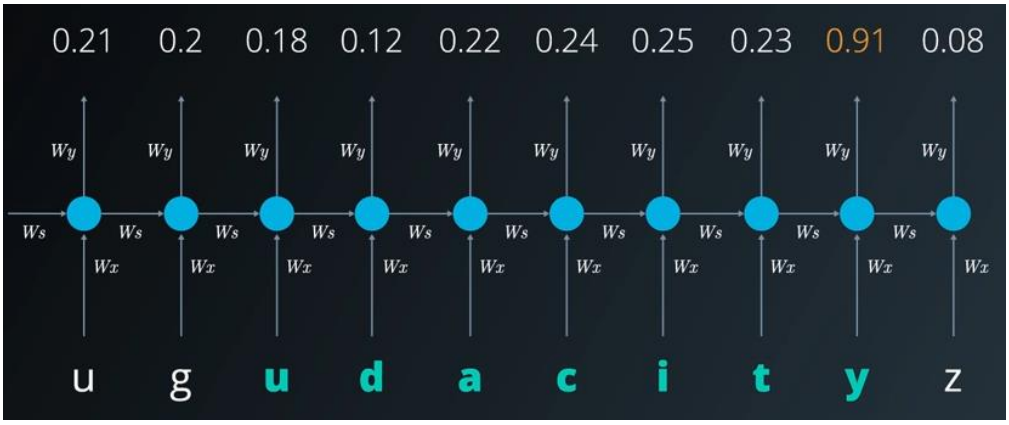
The first state vector is usually set to zero, allowing the next state to evolve as inputs come in. When training the network, we set the targets to either zero or one.



The target will be zero when the word Udacity is not detected, and one when it is.

1- **If the train our system on targets that are either zero or one, we except that the output will also take on values between zero and one**

After training our system and optimizing the weights, we would except that when the sequence Udacity appears, the output will signal that a sequence has been detected by talking on a value close to one, like 0.9 in this case.



Practically speaking we can set a threshold of say 0.9.

Threshold > 0.9

And decide that whenever the output exceeds this threshold value, the sequence of interest has been detected.

The number 0.9 in this case, by the way we selected as an arbitrary example.

In our example the RNN, was trained to recognize sequence of seven inputs like letters in the word Udacity.

However, we could have trained the RNN to recognize sequences of letters that have a different length, like five in the word Happy.

Generally speaking, the RNN can deal with varying sequence lengths.

So how do we train this network? Or in the other word, how to optimize its weights to minimize the output error?

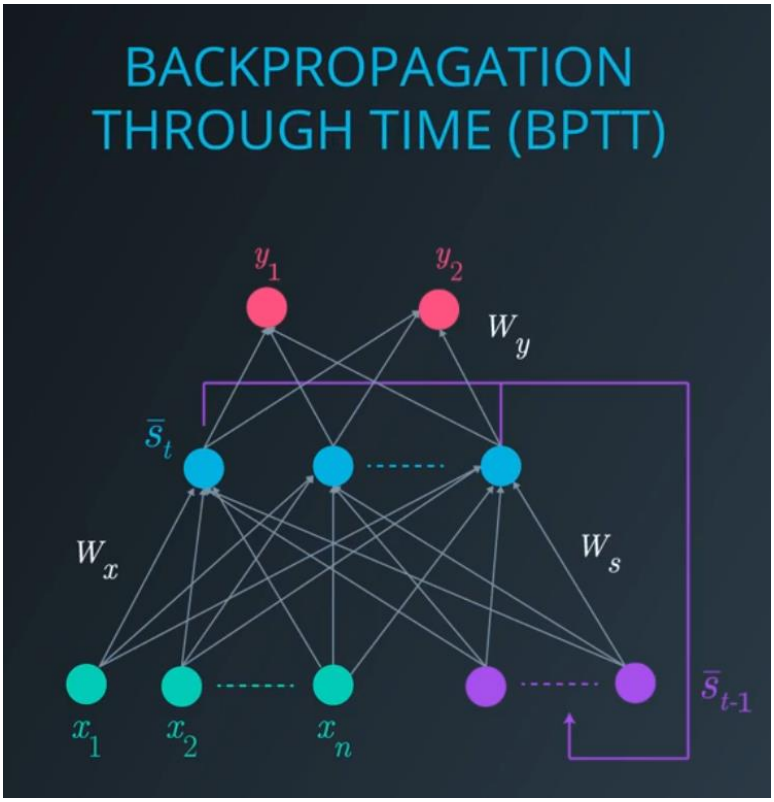
We do that using backpropagation through time, and we will be learning all about that in the next set of videos.

Deeper conceptual understanding of RNNs but how do we train such networks?

How can we find a good set of weights that would minimize the error?

You will see that our training framework will be similar to what we have seen before with a slight change in the backpropagation algorithm.

When training RNNs, we use what we call back propagation through time.

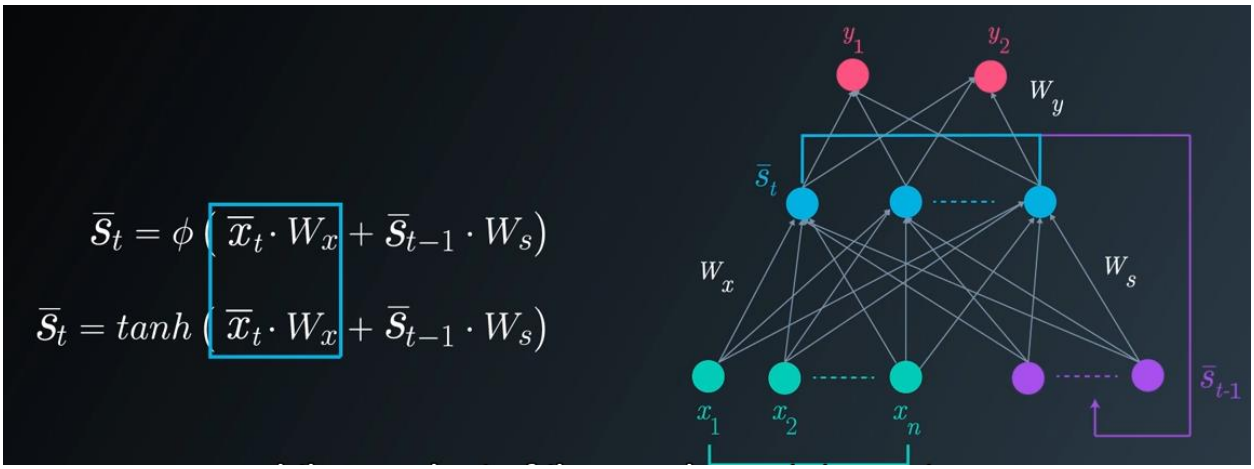


For simplicity reasons, let`s decide that from this point, whenever I refer to partial derivatives, I will simply say derivatives, as I will need to refer to those quite often. And before start just small reminder

If you feel that taking notes has been working for you, please continue to do so.

To better understand back propagation through time, we need a few notational definitions.

1- **The state vector S_of_t is given by applying an activation function let`s say a hyperbolic tangent for example, to sum of the product of the input vector X_of_t with a weight matrix Wx, and the product of the previous state vector S_of_t minus 1 with the weight matrix Ws.**



2- **The output at time t simply equals to the product of the state S_of_t, with wight matrix Wy, unless you are also using a soft using max function for example.**

$$\overline{y}_t = \sigma(\overline{s}_t \cdot W_y)$$

$$\overline{y}_t = \overline{s}_t \cdot W_y$$

3- **And the loss function E_of_t equals to the square of the difference between the**
   a. **Desired**
   b. **Network output at the time t**

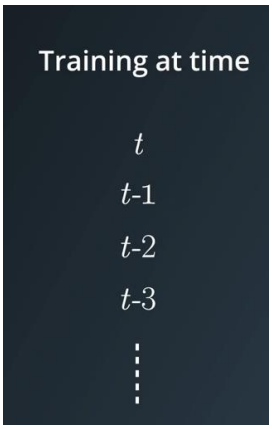
In the previous lessons you have seen other error function such as

1- **Cross entropy for example.**

But for consistency, we will stay with the same error we have seen so far in this lesson.

In Back propagation through time, we don`t independently train the system at a specific time t.

What we do, is we train the system at a specific time t as well as take into account all that has happened before.

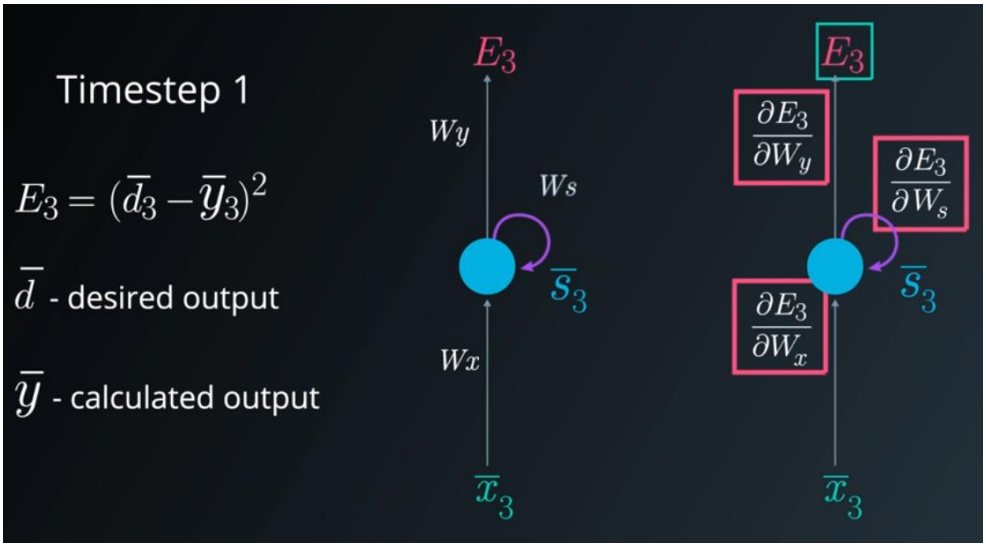


For example, assume that we are at timestep, t equals 3.

Our square error remains as before the square of the difference the desired output and the network output,

In this case at time t equal three.

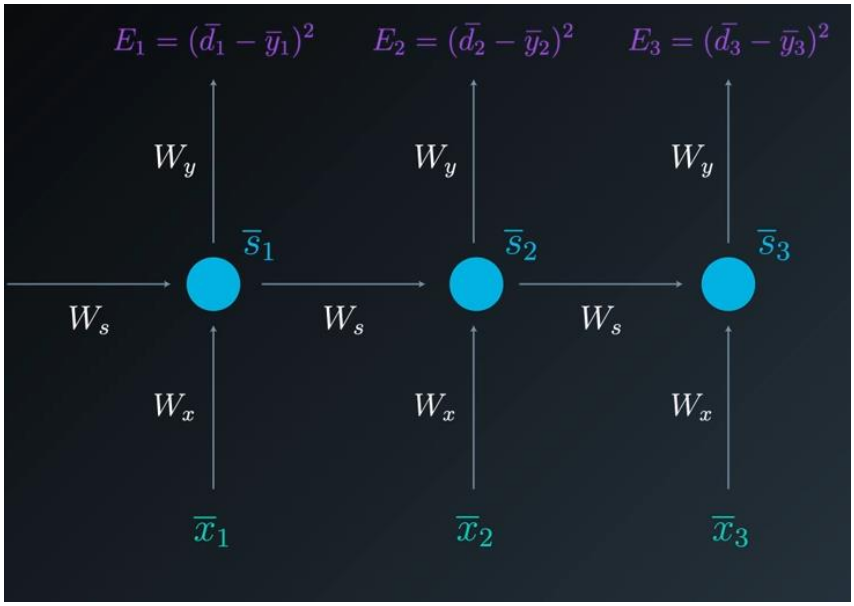
This is the folded scheme at this particular time. To update each weight matrix, we need to find the derivatives of the loss function at time t equals three as a function of the weight matrices.



In the other words, we will modify each matrix using gradient descent just as we did before in feedforward neural networks , but the addition , we also need to consider the previous timesteps.

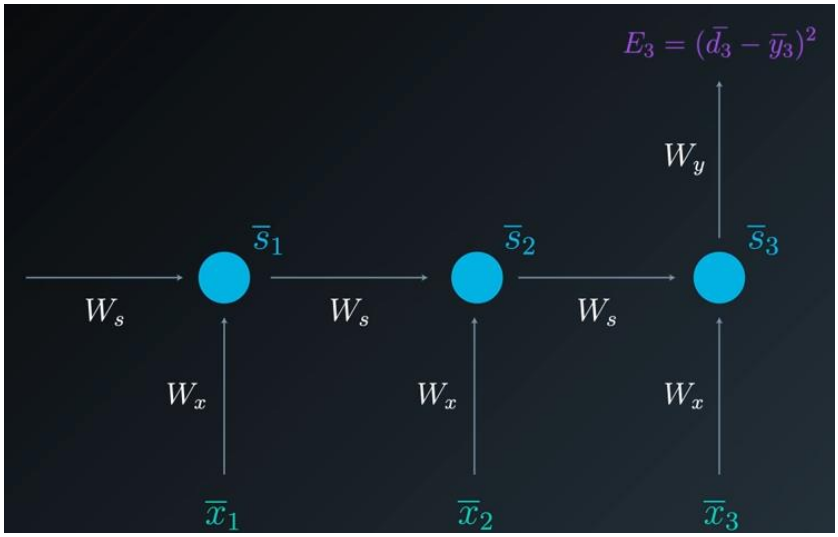
To better understand how to continue with this process of back propagation thought time we will unfold this model .

So let`s unfold the model in time and clean the sketch a bit, and focus on the third time step.



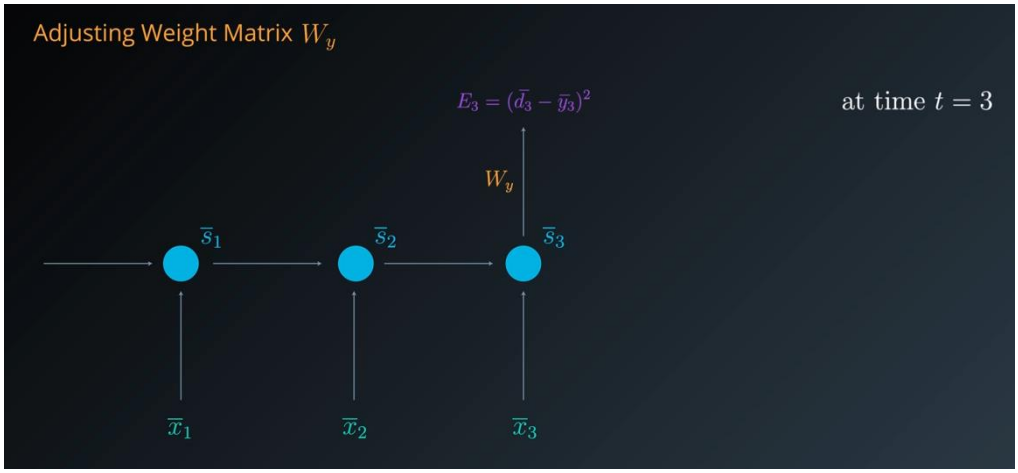
In this model we have three weight matrices we want to modify.

1-

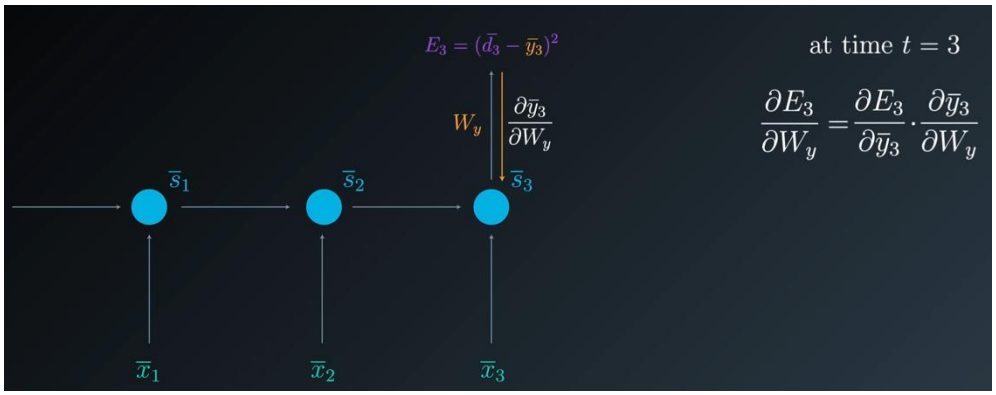


1- **The weight matrix Wx linking the network inputs to the state or the hidden layer**
2- **The weight matrix Ws connecting one stage to the next**
3- **The weight matrix Wy connecting this state to the output**

Let`s start with adjusting Wy which is most straightforward to obtain. At time T equals 3 ,

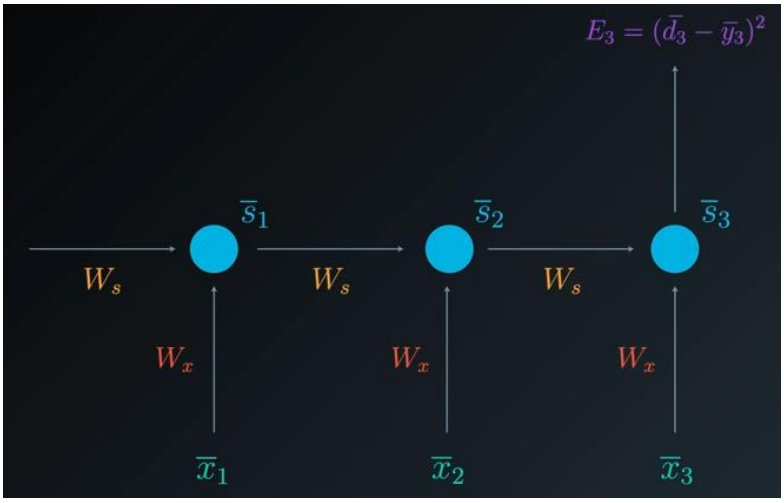


The derivative of the squared error with respect to Wy is found by a simple one step chain rule, and equals to derivative of the squared error with respect to output multiplied by the derivative of the output with respect to weight matrix Wy.



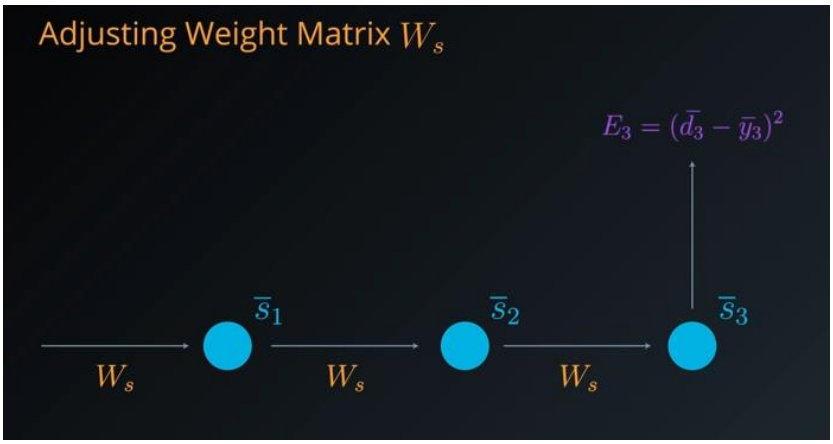
As always, these derivatives will be calculated according to each element of the weight matrix.

To adjust the other two matrices,



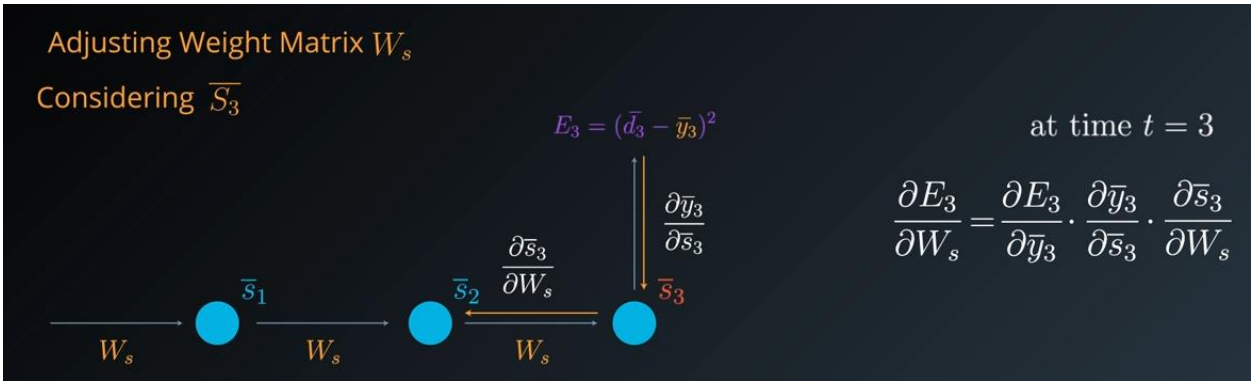
We will need to use backpropagation Through time, and doesn`t matter which one we choose to adjust first.

1- **Let`s choose to focus on the weight matrix Ws, the weight matrix connecting one state to the next and remove anything from the sketch.**



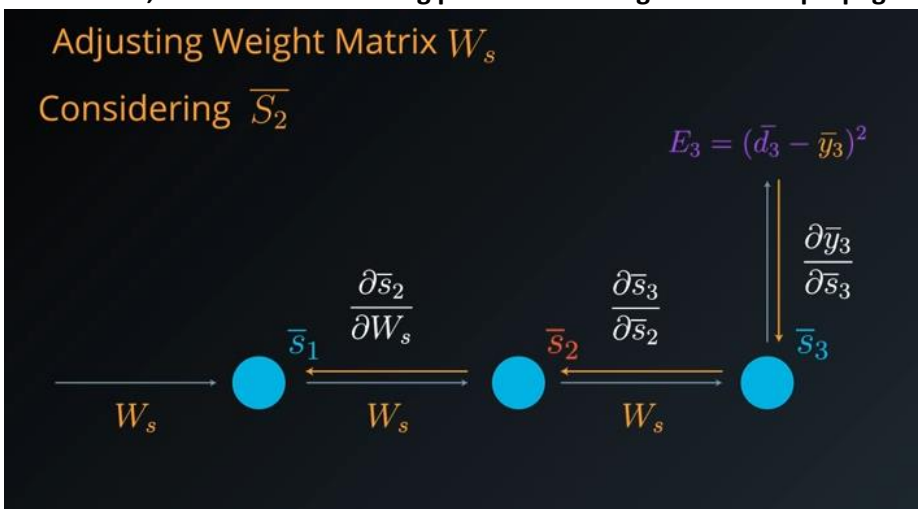
At first glance, it may seem that when finding the derivatives with respect to Ws , we only consider state S3.

This way the derivates of timestep T equals 3, simply equals the derivates of the squared error with respect to the output, multiplied by the derivative of the output with respect to S3, and muliypiled by the derivative of S3with respect to the matrix Ws.

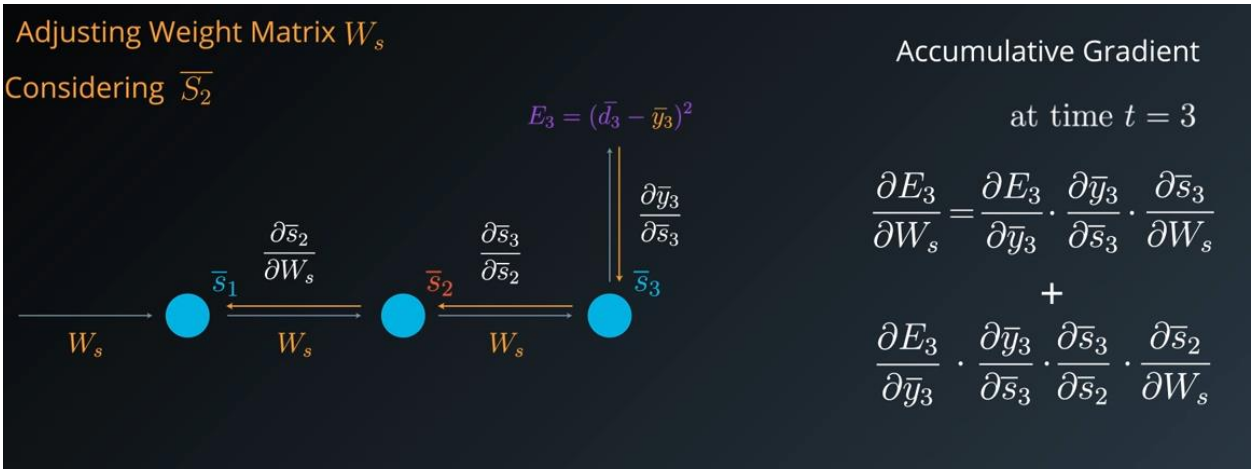


But S3 also depends on S2 and S1 the previous states, So we can`t really stop three. We also take into account what has happened before, and add to the contribution to our calculation. So we will continue calculating the gradient, knowing that we will need to accumulate the contributions, originating from each of the previous states.

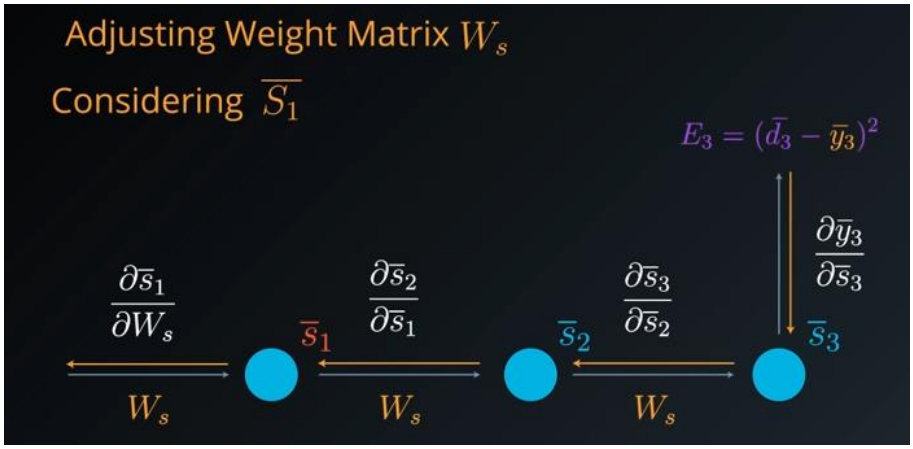
2- When we consider S2, we have the following path contributing to the back propagation.

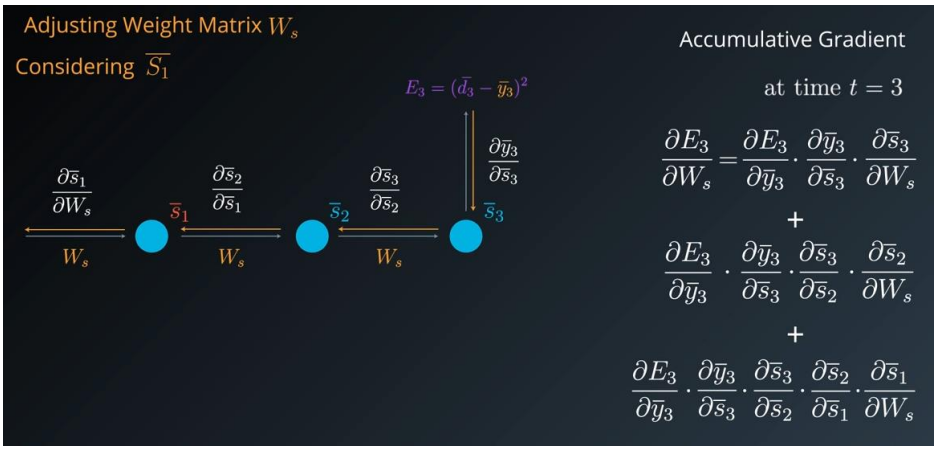


We can clearly see that S3 depend on S2 given us the derivatives calculations using chain rules, all the way back to the derivative of S2 with respect to the matrix Ws.



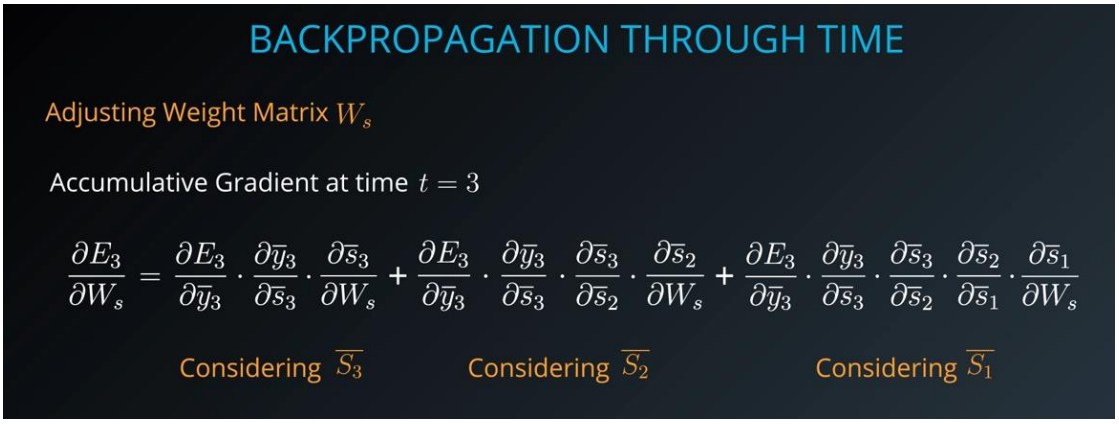
But we are not done yet, we need to go more step back to the first state S1, giving us the calculations again the chain rule, all the way to the derivative of S1 with respect to the matrix Ws.



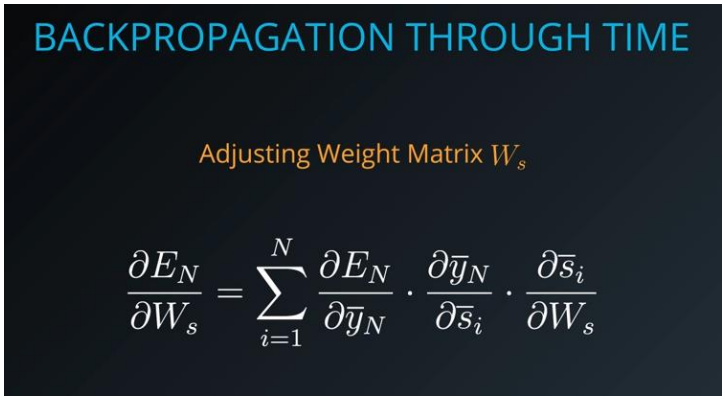


Let`s look again the accumulative gradient we now have using backpropagation Through time , Which calculated considering all the state vectors that we have,

1- **State vector S3**
2- **State vector S2**
3- **State vector S1**

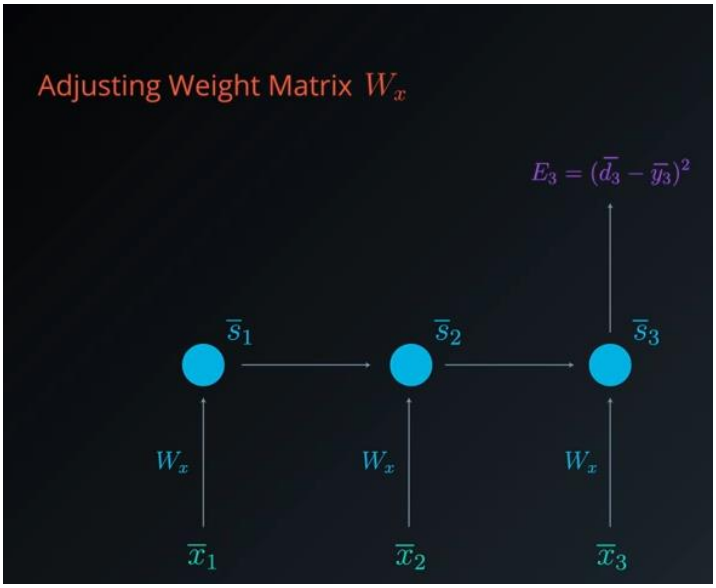


Generally speaking we consider multiple time steps back, and need general framework to define Backpropagation Through time, For purpose of Changing Ws.



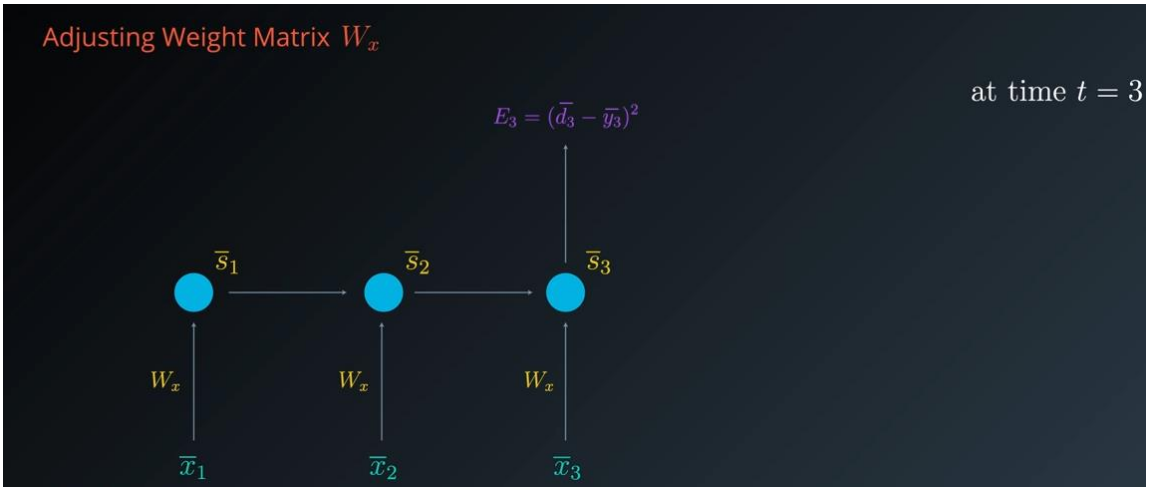
We still have adjust Wx The weight matrix connecting the input layer to the hidden or state layer.

Let`s simplify the sketch and leave only what we need. You will follow see that the process we follow do adjust Wx will be very similar to the one we use when updating Ws.



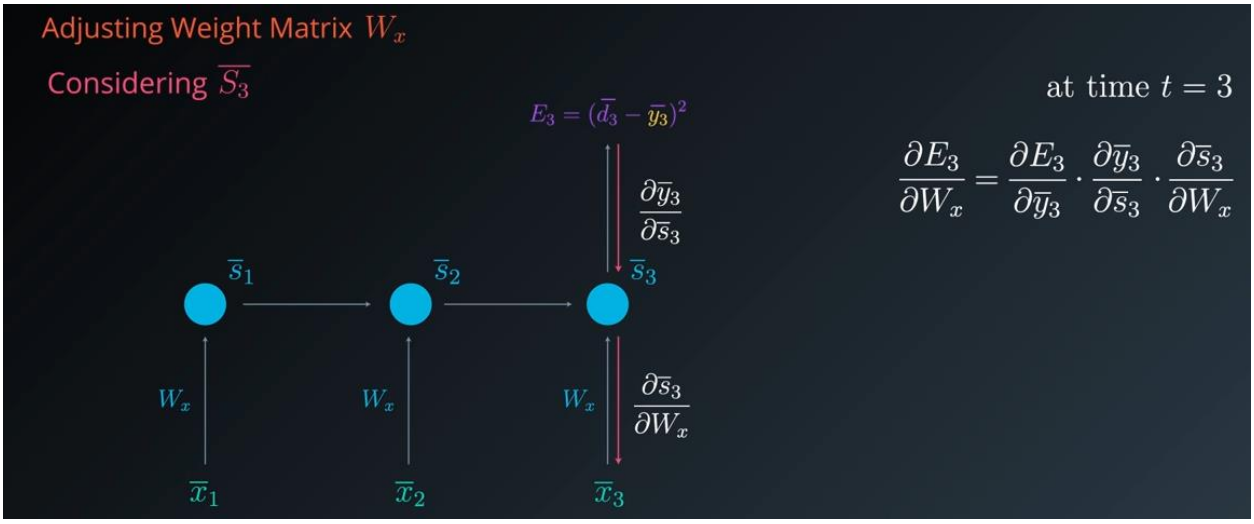
Having said that let`s go the process in detail.

1- **If we look at timestep, t equals three, the error with respect to the matrix Wx depends not only on vector S3 but also on S2 and its predecessor S1, which all affected by the same matrix Wx.**



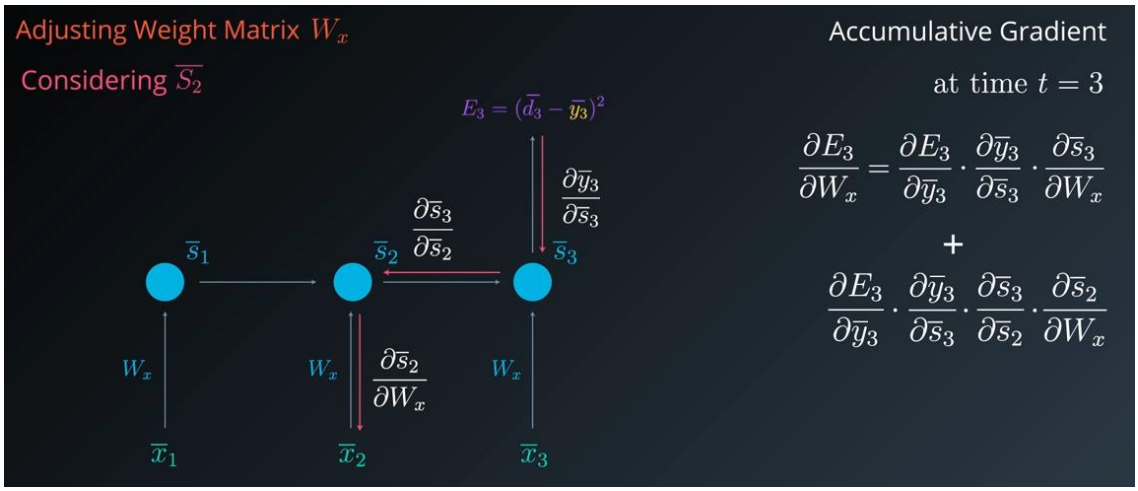
At first glance, it may seem that we need to consider only vector S3. So derivative of timestep t equals three, So the derivative of timestep t equals three by using the chain rule of course

Simply equals to the derivative of the square error with respect to the output y3 multiplied by the derivative of the output with respect to S3. And finally multiplied by the derivative of S3 with respect to the matrix Wx.



But let`s go beck for a bit as we said S3 also depend on S2 and S1 which are all affected by the same matrix Wx. So the gradient that we are looking for is not only the product of three derivatives we just saw but it is the accumulation of all of the contribution originating from each of the previous states.

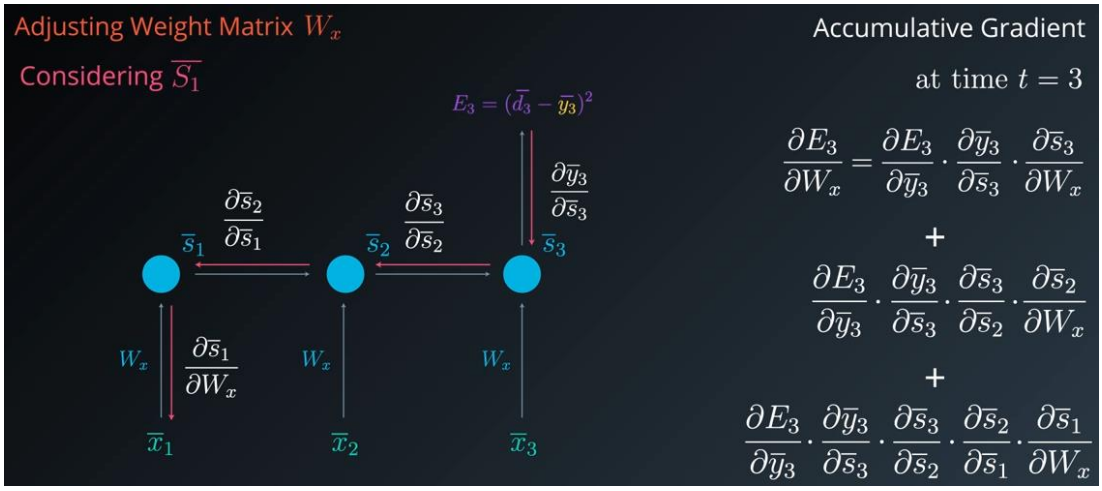
So, let`s consider the previous S2. Again by using chain rule, we can see the following path giving us an additional contributions to overall gradient.



But we are not done yet.

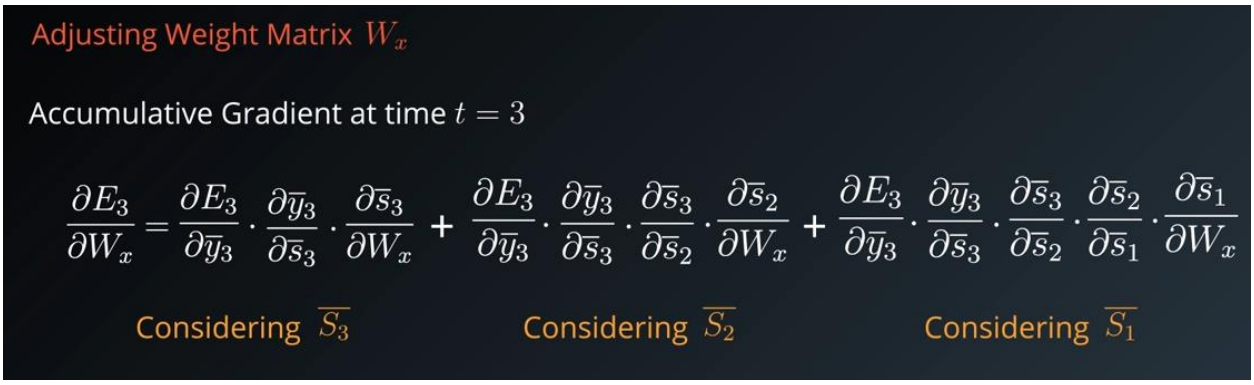
We have one more state, S1 , to consider. And we will add its contributions to the overall accumulative gradient.

Starting from the output and backpropagation to the first state , we will provide the following additional component to the overall gradient.



Let`s look again at the accumulative gradient we have using back propagation through time which calculated all the state vector we have

1- **State S3**
2- **State S2**
3- **State S1**



This is the complete gradient needed for the purpose of correctly updating the matrix Wx.

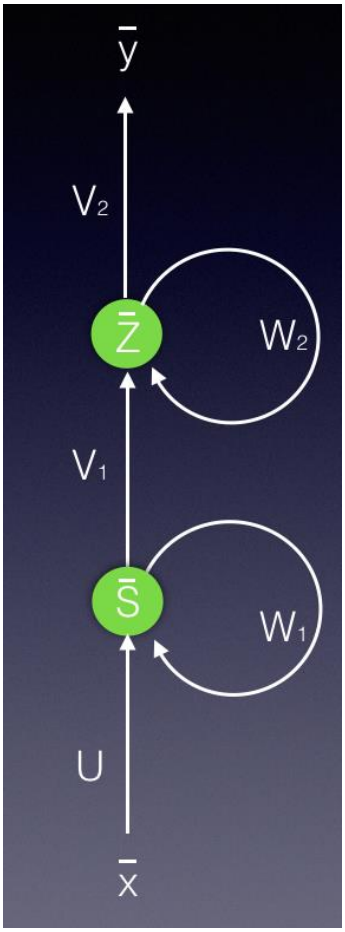
Generally speaking, we need to consider multiple past timesteps and not just three as in this example and need general Framework to define backpropagations through time for the purpose of updating Wx.

Adjusting Weight Matrix W_x

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \cdot \frac{\partial \bar{s}_i}{\partial W_x}$$

Notice the similarities between the calculations of $\frac{\partial E_3}{\partial W_s}$ and $\frac{\partial E_3}{\partial W_x}$. Hopefully after understanding the calculation process of $\frac{\partial E_3}{\partial W_s}$, understanding that of $\frac{\partial E_3}{\partial W_x}$ was straight forward.

Quiz :



RNN

Consider the above folded RNN Model. Both states **S** and **Z** have multiple neurons in each layer. The mathematical derivation of state **Z** at time t is:

Equation A

Equation B

Equation C

Equation D

Equation A $$z_t = \phi(s_t v_1 + z_{t-1} w_2)$$

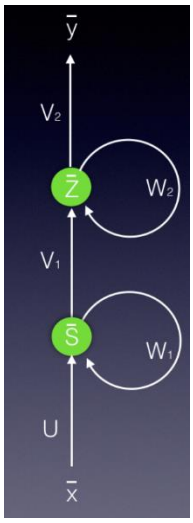
Equation B $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_t w_2)$$

Equation C $$z_t = \phi(s_t v_1 + z_t w_2)$$

Equation D $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_{t-1} w_2)$$

Solution

$\bar{z}$ and $\bar{s}$ are vectors, as we indicate that they have multiple neurons in each layer. Using this logic we can understand that equations A and C are incorrect. Since w_2 connects the hidden state $\bar{z}$ to itself, we know that we need to consider the previous timestep here. Therefore only equation D is the correct one.



A folded RNN model

Lets look at the same folded model again (displayed above). Assume that the error is noted by the symbol **E**. What is the update rule of weight matrix V1 at time t, over a single timestep ?

Equation A

Equation B

Equation C

Equation D

Equation A $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v}_1}$$

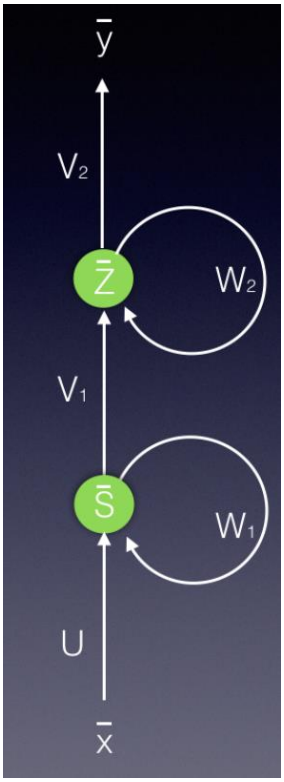
Equation B $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v}_1}$$

Equation C $$\Delta v_1 = \frac{\partial E_t}{\partial v_1} = \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v}_1}$$

Equation D $$\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{v}_1}$$

Solution

Equation B Is the only equation with the correct derivation of the chain rule with the proper use of the learning rate.



A folded RNN model

Lets look at the same folded model again (displayed above). Assume that the error is noted by the symbol **E**. What is the update rule of weight matrix U at time t+1 (over 2 timesteps) ?
Hint: Use the unfolded model for a better visualization.

Equation A

Equation B

Equation C

Equation A

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$

Equation B

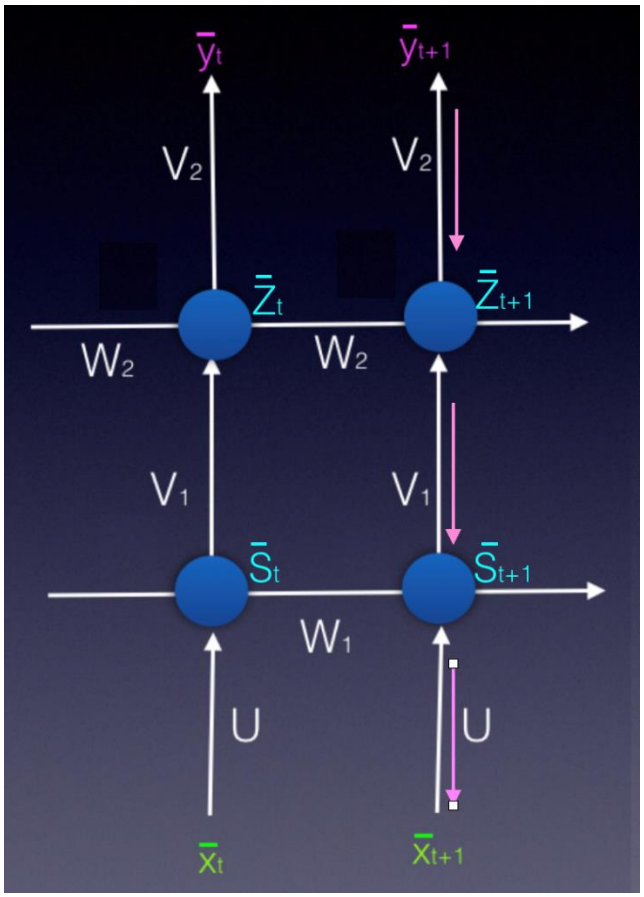
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Equation C

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Solution

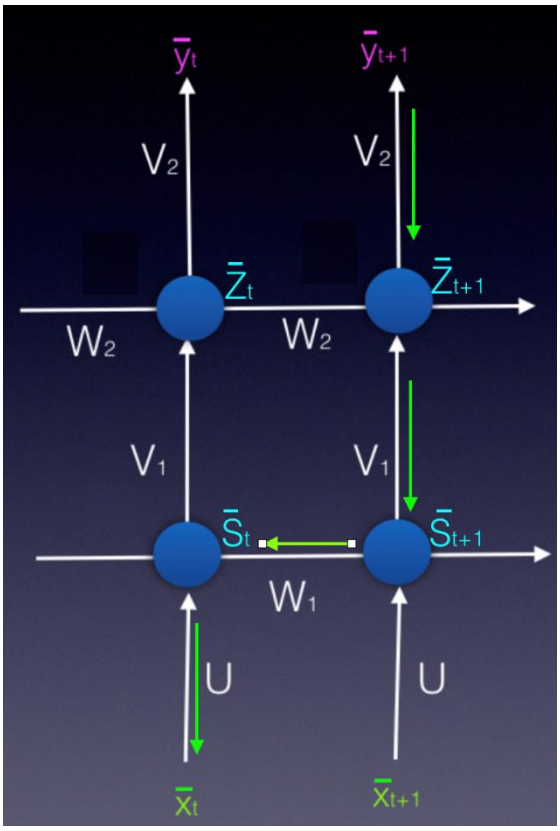
To understand how to update weight matrix U, we will need to unfold the model in time. We will unfold the model over two time steps, as we need to look only time t and time t+1. The following three pictures will help you understand the **three** paths we need to consider. Notice that we have two hidden layers that serve as memory elements, so this case will be different than the one we saw in the video, but the idea is the same. We will use **BPTT** while applying the chain rule.



The first path to consider

The following is the equation we derive using the first path:

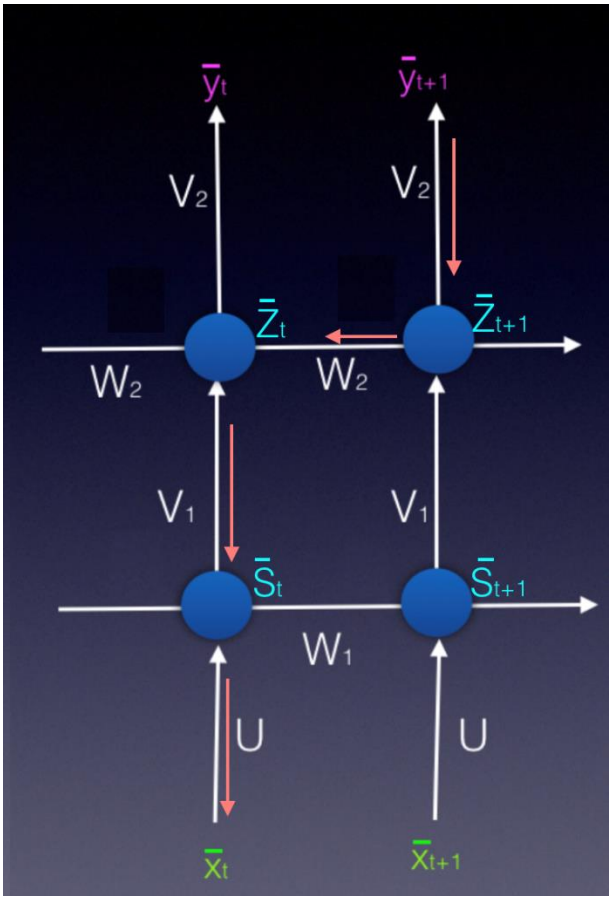
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$



The second path to consider

The following is the equation we derive using the second path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$



The third path to consider

The following is the equation we derive using the third path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Finally, after considering all three paths, we can derive the correct equation for the purposes of updating weight matrix U, using BPTT:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$

$$+$$

$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

$$+$$

$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

The final answer for BPTT Quiz 3

is section is given as bonus material and is not mandatory. If you are curious how we derived the final accumulative equation for BPTT, this section will help you out.

In the previous videos, we talked about **Backpropagation Through Time**. We used a lot of partial derivatives, accumulating the contributions to the change in the error from each state. Remember?
When we needed a general scheme for the BPTT, I simply displayed the equation without giving you further explanations.

As a reminder, the following two equations were derived when adjusting the weights of matrix W_sW_s and matrix W_xW_x:

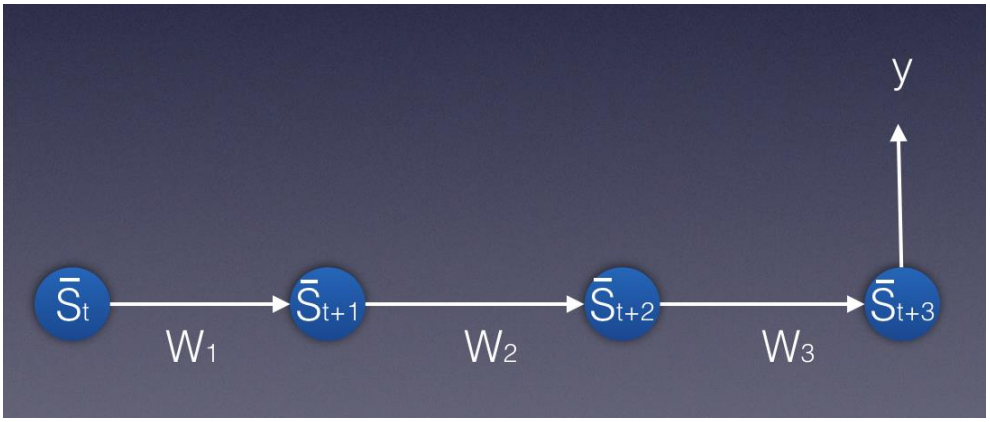
$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Equation 48: BPTT calculations for the purpose of adjusting Ws

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_x}$$

Equation 49: BPTT calculations for the purpose of adjusting Wx

To generalize the case, we will avoid proving equation 48 or 49, and will focus on a general framework.
Let's look at the following sketch, presenting a portion of a network:



In the picture above, we have four states, starting with s_ts_t.
We will initially consider the three weight matrices W_1W_1,W_2W_2 and W_3W_3 as three different matrices.

Using the chain rule we can derive the following three equations:

Equation 50 (Equation set)

$$\frac{\partial y}{\partial W_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W_3}$$

$$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W_2}$$

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W_1}$$

Equation 50 (Equation set)

In **Backpropagation Through Time** we accumulate the contributions, therefore:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W_1} + \frac{\partial y}{\partial W_2} + \frac{\partial y}{\partial W_3}$$

Equation 51

Since this network is displayed as *unfolded in time*, we understand that the weight matrices connecting each of the states are identical. Therefore:

$\mathrm{W\_1}W_1=\mathrm{W\_2}W_2=\mathrm{W\_3}W_3$

Lets simply call it weight matrix $\mathrm{W}W$. Therefore:

$\mathrm{W\_1}W_1=\mathrm{W\_2}W_2=\mathrm{W\_3}W_3=\mathrm{W}W$

Equation 52

From *equation 52*, *equation 51* and the *set of equations 50* we derive that:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 52 summarizes the mathematical procedure of BPTT and can be simply written as:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 53

Notice that for i=t+1$i=t+1$, we derive the following:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 54

With the use of the chain rule we can derive the following equation (displayed in *set of equations 50*).

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 55

A general derivation of the BPTT calculation can be displayed the following way:

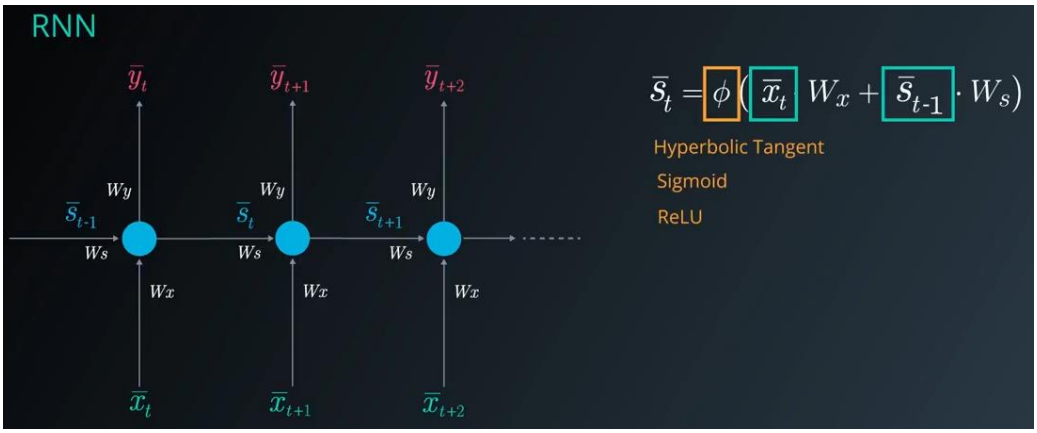
$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 55

We summarize to what will discuss.

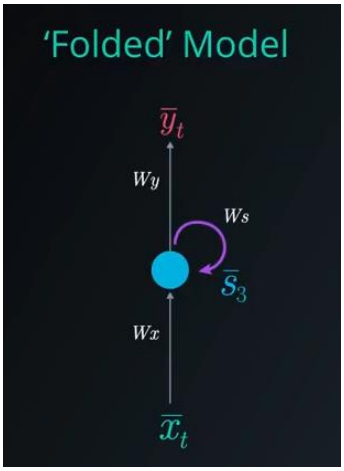
We know understand that in RNNs. the current state depends on the inputs as well as on the previous states, with use the activation function like the

1- **Hyperbolic Tangent**
2- **The Sigmoid**
3- **RelU**



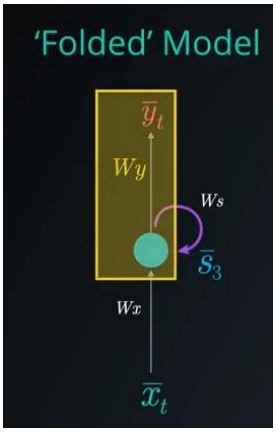
The current output is a simple linear combination of the current state elements with corresponding weight matrix.

We can also using the SoftMax function to calculate the outputs.

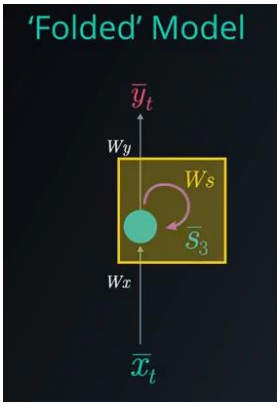


This is the folded form of the RNN scheme. When we have only one hidden layer, we need to update three matrices

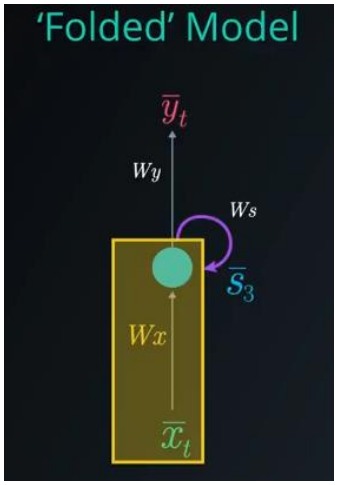
1- **Wy connecting the state to the output**



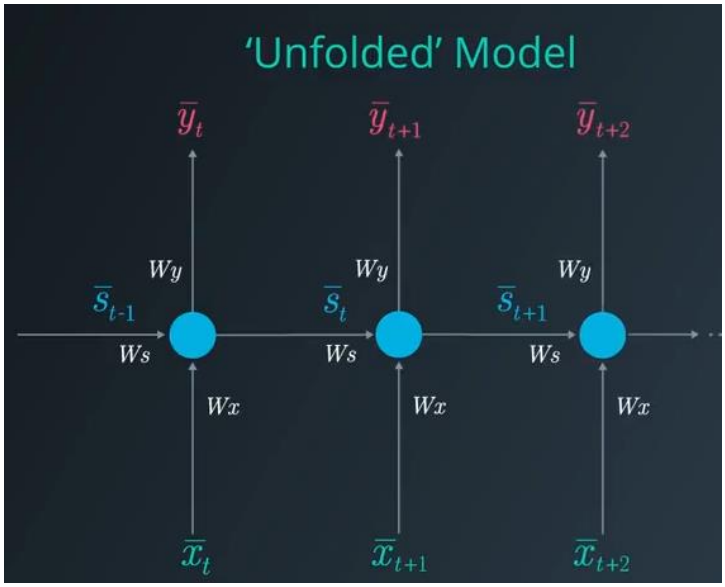
2- **Ws connecting to state to itself**



3- Wx connecting the input the state



This is the unfolded Model and it is the one we have been mostly using.

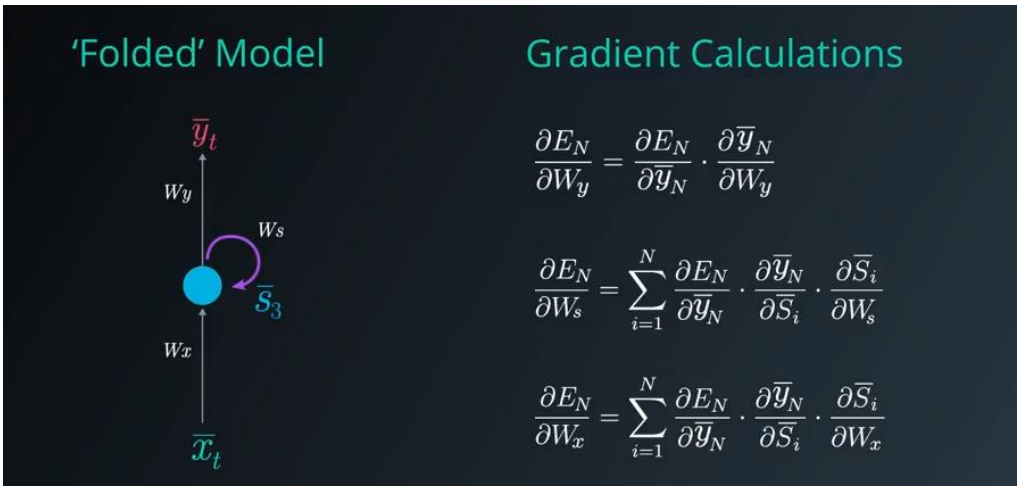


Let`s look at the gradient calculations again.

1- **Calculating the gradient of the squared error or what we call the loss function, with respect to the weight matrix Wy, was straightforward.**

And by the way, you can choose to use other error function of course.

When calculating the gradient with respect to Ws and Wx, We need to more careful and consider wat happened in previous timesteps accumulating all of these contributions.



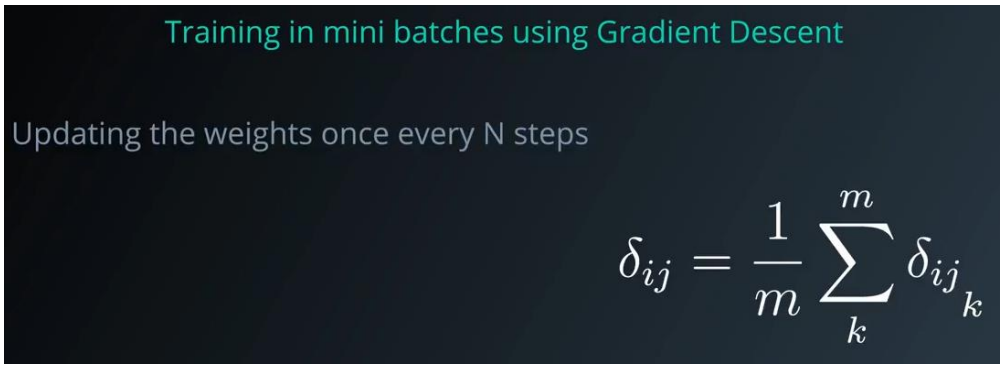
For these calculations, we used a process called backpropagation Through time.

As we discussed earlier, When using backpropagation we can choose to use mini batches We can do the same in RNN.

Updating the weights in Backpropagation Through Time, can be performed periodically in batches as well as opposed to once every input sample.

As reminder , we calculate the gradient for each step but not update the weights right away

We can choose to update once every fixed number of steps. For example, 20.



1- **The helps reduce the complexity of the training process and can also remove noise from the weights updates, since averaging a set of noisy samples tends to yield a loss noisy value.**

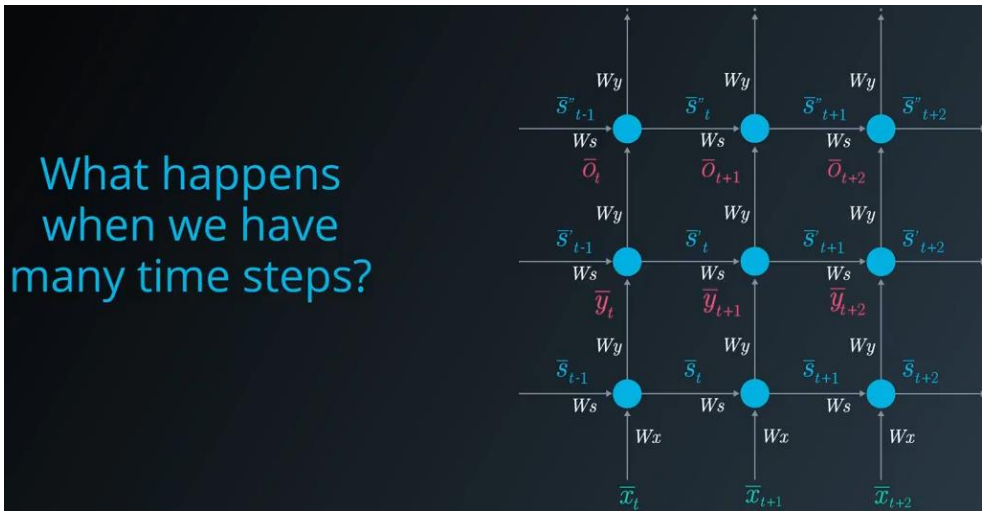
You may ask yourself, what happens when we have many time steps? Add not just a few as we had in our previous example. Remember the Backpropagation Through time example?

We had only three time steps.

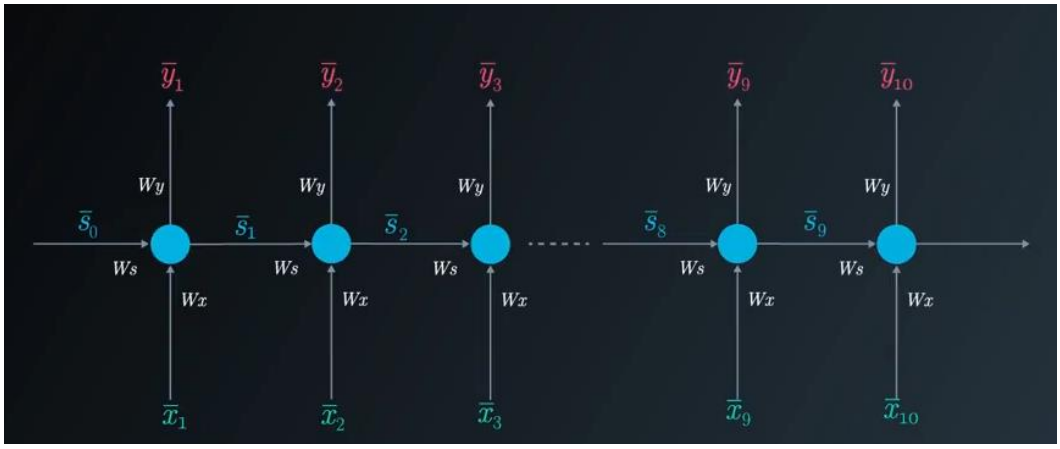
We may also have more than one hidden layer. So what happen then ?

Can we simply accumulate the gradient contributions from each of these time steps ?

The simple answer to that is no we can`t



Studies have shown that for up to a small number of time steps say, eight or ten, we can effectively use this method.

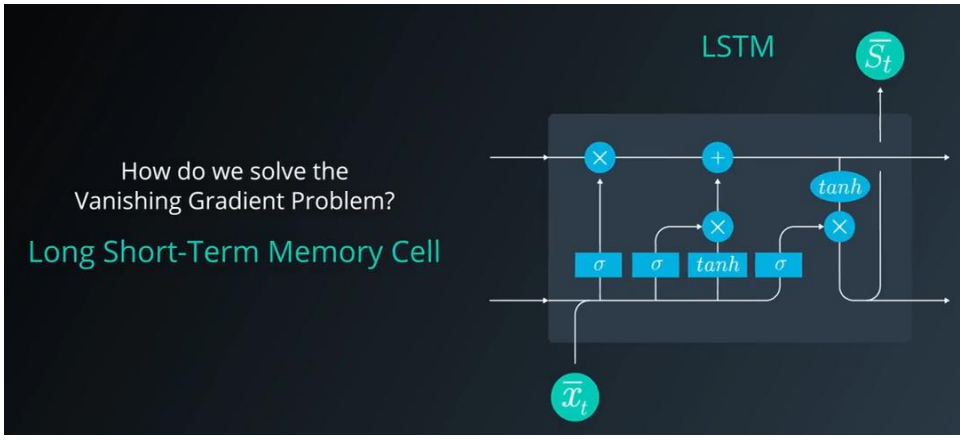


1- **If we backpropagate further, the gradient will become too small which is known as the Vanishing Gradient problem where the contributions of the information decays geometrically over time**
2- **Or in other word temporal dependencies that span many time steps, for example, more than eight or nine or ten time steps, will effectively be disguarded by the network.**

So how do we address this the Vanishing Gradient Problem?

1- **Long Short-Term Memory Cell or LSTM`s in short were invented specifically to address**

This issue and they will be the topic of our next set of videos.



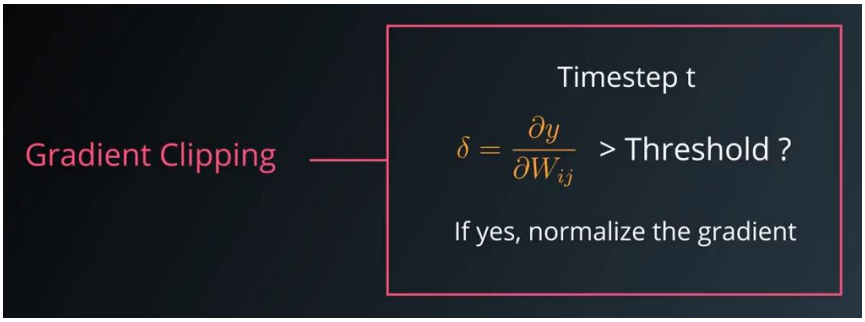
The other scenario to be aware of is that exploding Gradient, in which the value of the gradient grows uncontrollably.

Luckily a simple scheme called gradient clipping practically resolves this issue .

Basically, what we do, is we check in each timestep whether the gradient exceeds a certain threshold.

1- **If it does, we normalize the successive gradient.**

Normalize means that we penalize super large gradient, more than those that are slightly larger than our threshold.



Clapping large gradient this way helps avoid the exploding Gradient problem.

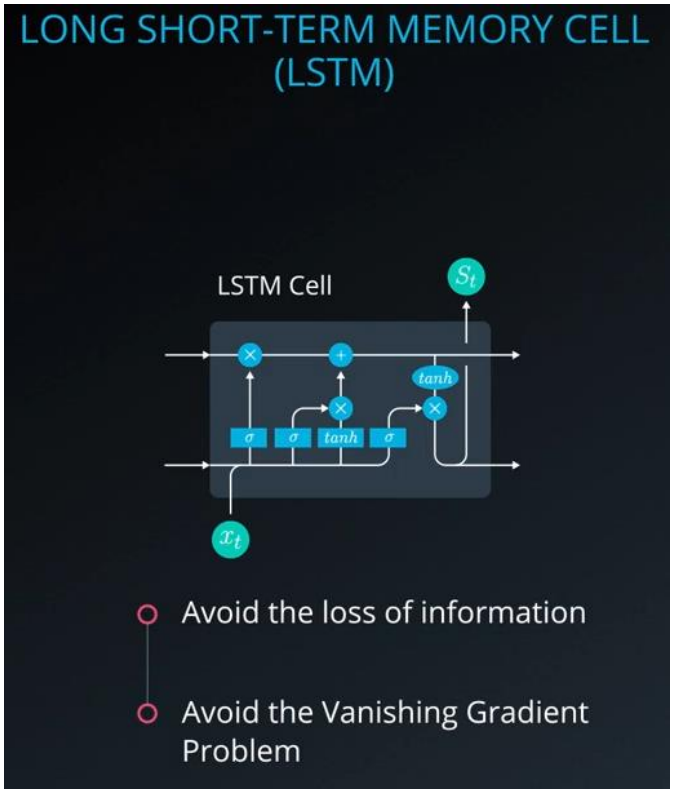
The long short memory cell or LSTM cells, were proposed in 1997 by Sepp Hocreiter and jurgen Shmidhuber

1- **The goal of the cell its to overcome the vanishing gradient problem.**
2- **You will see that it allows certain inputs to be latched , or stored for long periods of time without forgetting them as would be the case in RNNs.**

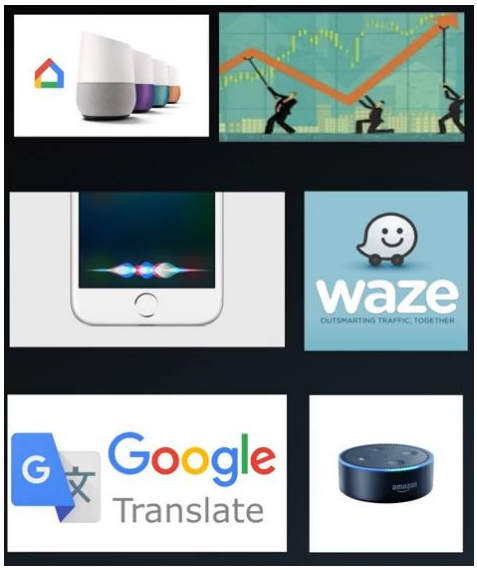
When calculating the gradient using backpropagation through time, the gradients stemming back many timesteps can be come negligible for the same reason, the partial derivative of the error can also become negligible. This is fundamental problem in RNN.

You will see the in LSTM we want to avoid the loss of information or the vanishing gradient problem by intentionally latching on to some information over many timesteps.

Remember information over long periods time is easily achieved using these methods.

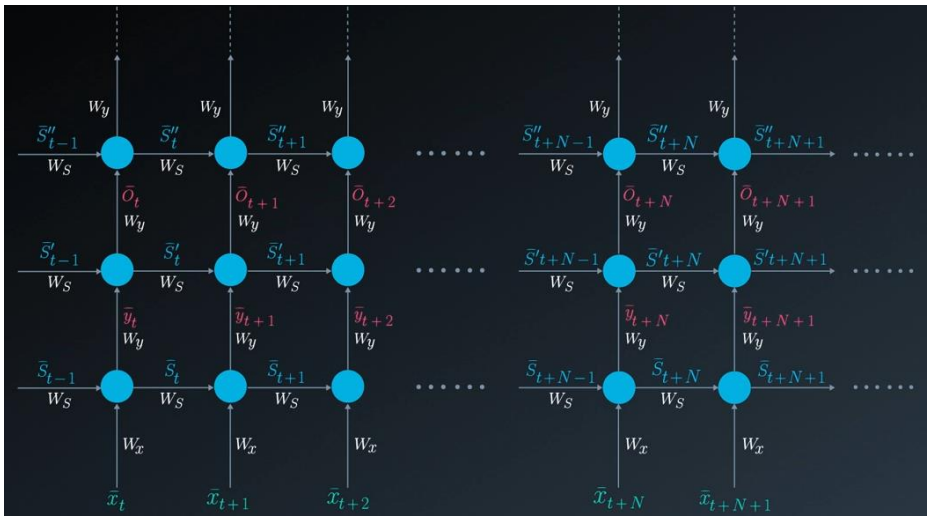


Many real-world applications such as Google`s language translations tools, or amazon Alexa are powered by LSTM.

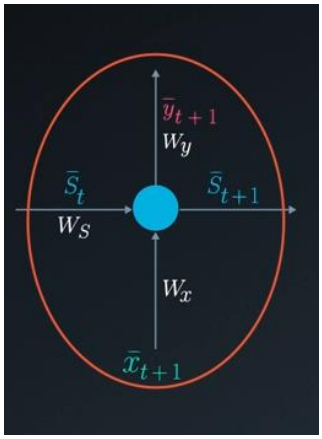


In reality, most of the RNN applications we mentioned are moving towards implementation using LSTMs.

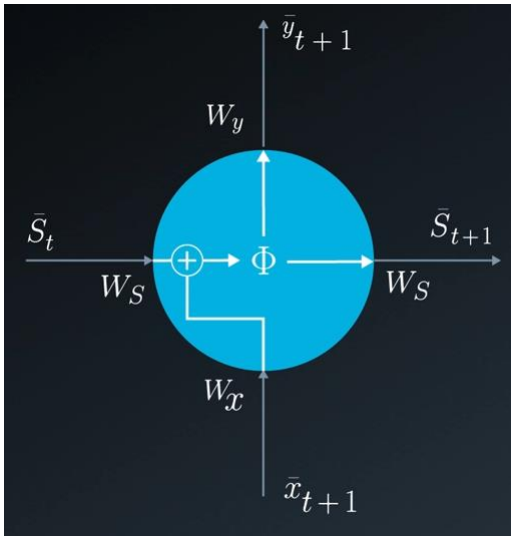
To understand the difference between LSTM and RNN system.



And zoom on one neuron in a hidden layer, and understand how we calculated S of T plus one

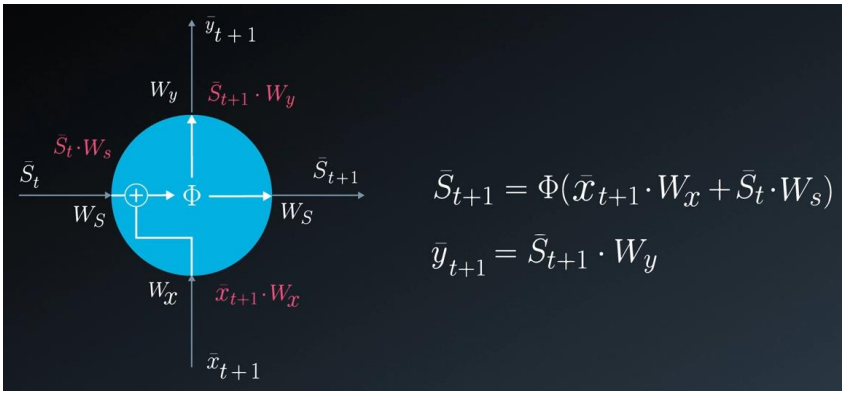


Zooming in a bit more will help us remember how the calculations are done.



As you may recall, the next state was calculated through a simple activation function, let`s say a hyperbolic tangent, of a linear combination of different inputs, and corresponding weight matrices.

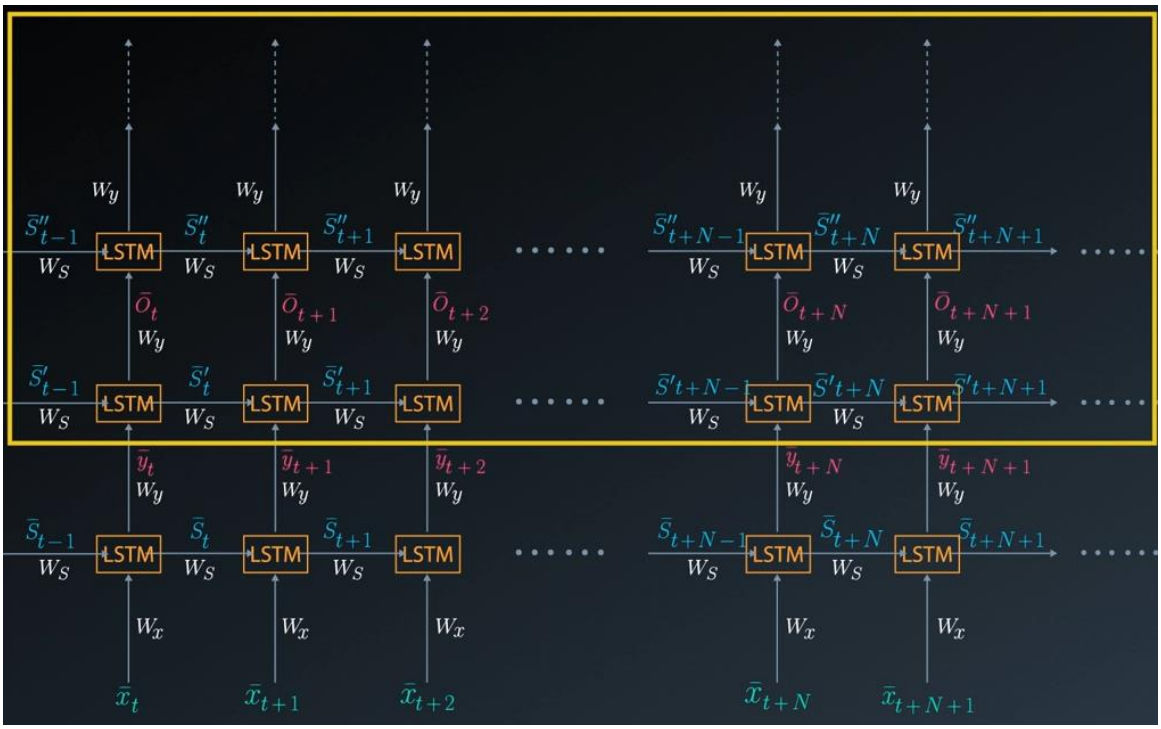
1- **The output was calculated as a simple linear combination as well.**



Using LSTM, we no longer have basic computations as we had here in a single neuron.

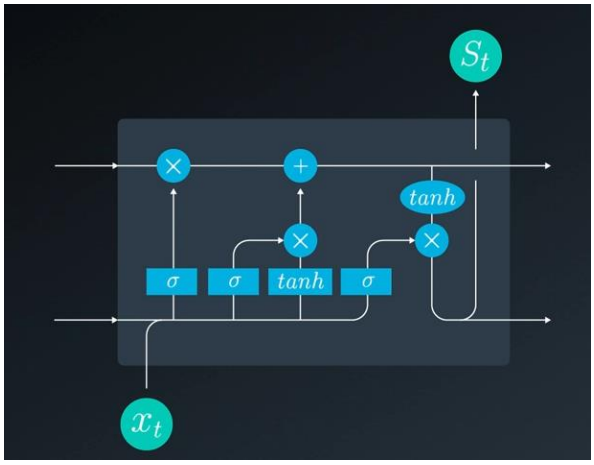
Zooming out again our system will have a very similar layout.

The neurons of the hidden states are replaced with LSTM cells, and can be stacked like lego pieces , just as before

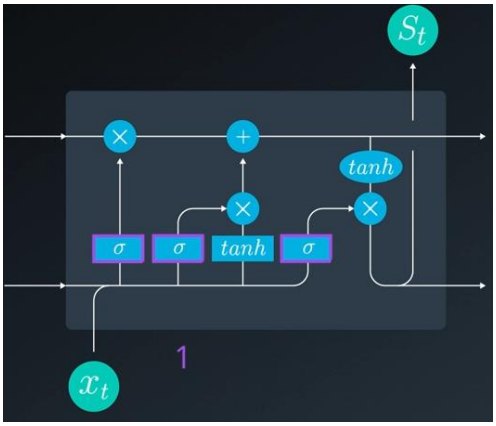


If we zoom in on one cell, we will find that we no longer have a single calculation, but we have four separate ones,

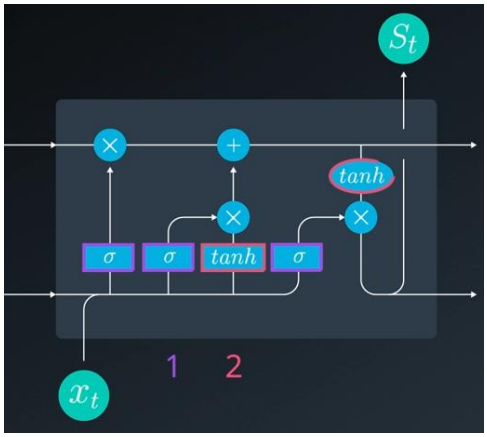
This is our LSTM cell,



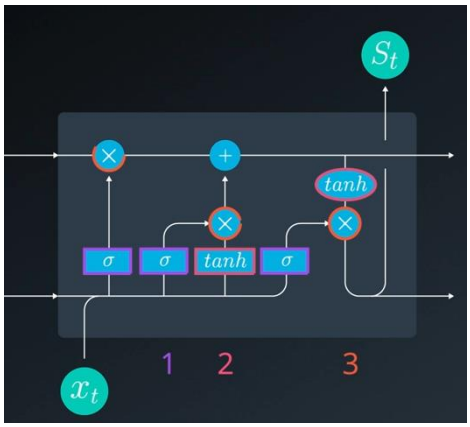
1- **This is our first calculations,**



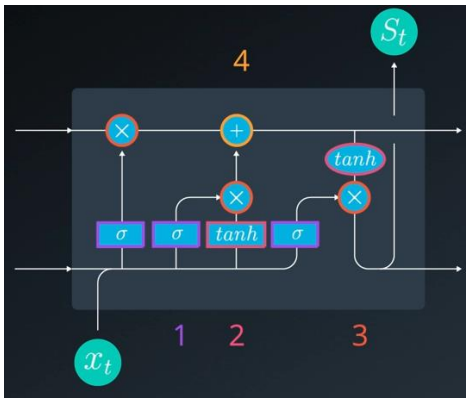
2- **Our Second**



3- **Our Third**



4- **Finally, Fourth**



The LSTM network allows a recurrent system to learn over many timesteps while learning the network using the same backpropagation principles.

In these cases, we are no longer considering a few timesteps back as we did in RNNs, but consider over 1,000

The cell is fully differentiable, meaning that all of its function have a gradient or a derivative that we can calculate

1- **These function are a sigmoid**
2- **Hyperbolic tangent**
3- **Multiplication**
4- **Addition**

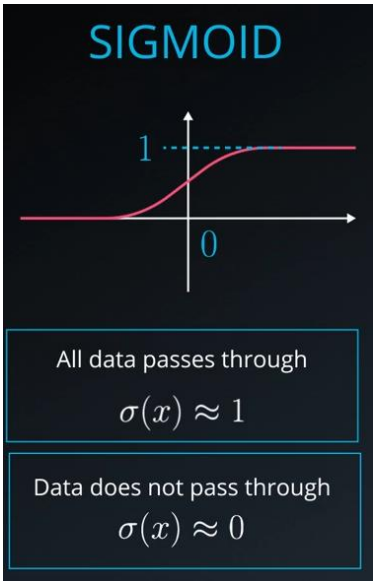
This is allow us to use backpropagation, or stochastic gradient descent when updating the weights.

The main idea with LSTM cells, is that can decide which information to remove of forget which information to store and when use it.

The cell can also help decide when to move the previous states information to the next

We just saw that the LSTM cell has three sigmoid.

1- **The output of each sigmoid is between zero and one. Having the data flow through a sigmoid intuitively answer the following questions.**
   a. **Do we let all the data flow through when the output of the sigmoid is one or close to it**
   b. **Or do we force the output to be zero where none of data flows through.? And that would be if the output of the sigmoid is zero or close to it**



1- **These three sigmoid act as mechanism to filter what goes into the cell if at all,**

2- **what is retained within the cell?**

3- **What pass through to its output.**

The key idea in LSTMs is that these three gating functions are also trained using backpropagation by adjusting the weights that feed into them.