**Recurrent Neural Networks**

So far, we've been looking at convolutional neural networks and models that allows us to analyze the spatial information in a given input image. CNN's excel in tasks that rely on finding spatial and visible patterns in training data.

In this and the next couple lessons, we'll be reviewing RNN's or recurrent neural networks. These networks give us a way to incorporate **memory** into our neural networks, and will be critical in analyzing sequential data. RNN's are most often associated with text processing and text generation because of the way sentences are structured as a sequence of words!

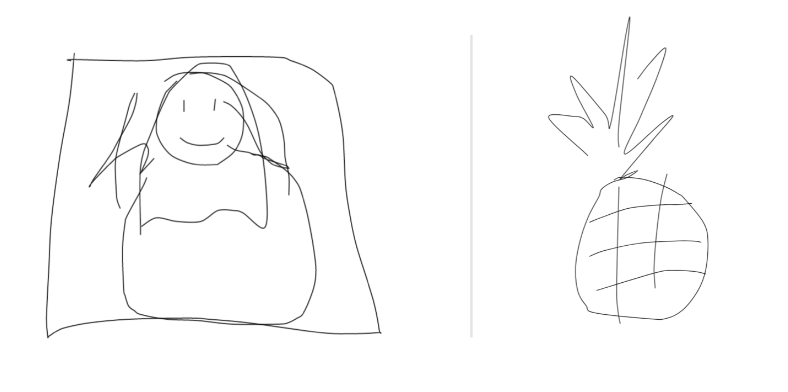
**RNN's and Text Generation**

At the end of this lesson, you will be tasked with creating a TV script generator; a model that takes in a series of words as input and outputs a likely *next* word, forming a text, one word at a time.

Similarly, RNN's can be used to generate text given *any* other data. For example, you can give an RNN a feature vector from an image, and use it to [generate a descriptive caption](https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/03-advanced/image_captioning). Image captions are used to create accessible content and in a number of cases where one may want to read about the contents of an image.

**Sketch RNN**

One of my favorite use cases for RNN's is in generating drawings. [Sketch RNN (demo here)](https://magenta.tensorflow.org/assets/sketch_rnn_demo/index.html) is a program that learns to complete a drawing, once you give it something (a line or circle, etc.) to start!

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/066d068f-f601-4386-a2ec-d8ba685d8c52)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/066d068f-f601-4386-a2ec-d8ba685d8c52)

[Sketch RNN example output. Left, Mona Lisa. Right, pineapple.](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/066d068f-f601-4386-a2ec-d8ba685d8c52)

It's interesting to think of drawing as a sequential act, but it is! This network takes a starting line or squiggle and then, having trained on a number of types of sketches, does its best to complete the drawing based on your input squiggle.

Next, you'll learn all about how RNN's are structured and how they can be trained! This section is taught by Ortal, who has a PhD in Computer Engineering and has been a professor and researcher in the fields of applied cryptography and embedded systems.

The neural network architectures you've seen so far were trained using the current inputs only. We did not consider previous inputs when generating the current output. In other words, our systems did not have any **memory** elements. RNNs address this very basic and important issue by using **memory** (i.e. past inputs to the network) when producing the current output.

As mentioned in this video, RNNs have a key flaw, as capturing relationships that span more than 8 or 10 steps back is practically impossible. This flaw stems from the "**vanishing gradient**" problem in which the contribution of information decays geometrically over time.

What does this mean?

As you may recall, while training our network we use **backpropagation**. In the backpropagation process we adjust our weight matrices with the use of a **gradient**. In the process, gradients are calculated by continuous multiplications of derivatives. The value of these derivatives may be so small, that these continuous multiplications may cause the gradient to practically "vanish".

**LSTM** is one option to overcome the Vanishing Gradient problem in RNNs.

Please use these resources if you would like to read more about the [Vanishing Gradient](https://en.wikipedia.org/wiki/Vanishing_gradient_problem) problem or understand further the concept of a [Geometric Series](https://socratic.org/algebra/exponents-and-exponential-functions/geometric-sequences-and-exponential-functions) and how its values may exponentially decrease.

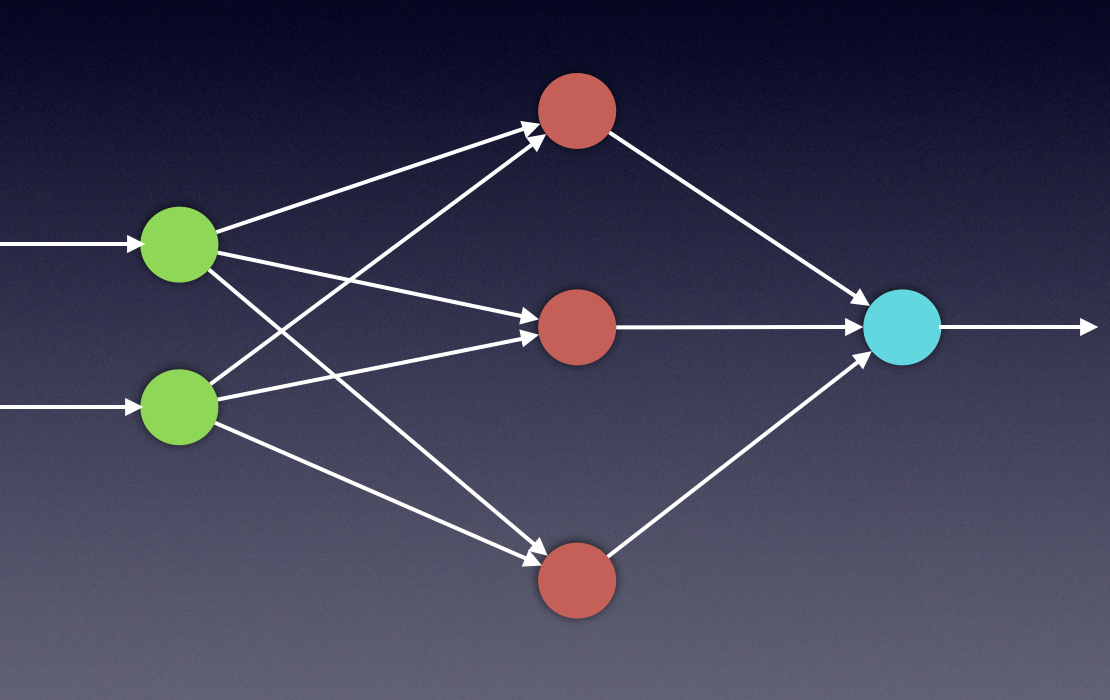
If you are still curious, for more information on the important milestones mentioned here, please take a peek at the following links:

* [TDNN](https://en.wikipedia.org/wiki/Time_delay_neural_network)
* Here is the original [Elman Network](http://onlinelibrary.wiley.com/doi/10.1207/s15516709cog1402_1/abstract) publication from 1990. This link is provided here as it's a significant milestone in the world on RNNs. To simplify things a bit, you can take a look at the following [additional info](https://en.wikipedia.org/wiki/Recurrent_neural_network#Elman_networks_and_Jordan_networks).
* In this [LSTM](http://www.bioinf.jku.at/publications/older/2604.pdf) link you will find the original paper written by [Sepp Hochreiter](https://en.wikipedia.org/wiki/Sepp_Hochreiter) and [Jürgen Schmidhuber](http://people.idsia.ch/~juergen/). Don't get into all the details just yet. We will cover all of this later!

As mentioned in the video, Long Short-Term Memory Cells (LSTMs) and Gated Recurrent Units (GRUs) give a solution to the vanishing gradient problem, by helping us apply networks that have temporal dependencies. In this lesson we will focus on RNNs and continue with LSTMs. We will not be focusing on GRUs. More information about GRUs can be found in the following [blog](https://deeplearning4j.org/lstm.html). Focus on the overview titled: **GRUs**.

NEXT

**Feedforward Neural Network - A Reminder**

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/ebec6246-752c-49ef-bf74-423b8c3683f3)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/ebec6246-752c-49ef-bf74-423b8c3683f3)

The mathematical calculations needed for training RNN systems are fascinating. To deeply understand the process, we first need to feel confident with the vanilla FFNN system. We need to thoroughly understand the feedforward process, as well as the backpropagation process used in the training phases of such systems. The next few videos will cover these topics, which you are already familiar with. We will address the feedforward process as well as backpropagation, using specific examples. These examples will serve as extra content to help further understand RNNs later in this lesson.

The following couple of videos will give you a brief overview of the **Feedforward Neural Network (FFNN)**.

OK, you can take a small break now. We will continue with FFNN when you come back!

As mentioned before, when working with neural networks we have 2 primary phases:

**Training**

and

**Evaluation**.

During the **training** phase, we take the data set (also called the *training set*), which includes many pairs of inputs and their corresponding targets (outputs). Our goal is to find a set of weights that would best map the inputs to the desired outputs. In the **evaluation** phase, we use the network that was created in the training phase, apply our new inputs and expect to obtain the desired outputs.

The training phase will include two steps:

**Feedforward**

and

**Backpropagation**

We will repeat these steps as many times as we need until we decide that our system has reached the best set of weights, giving us the best possible outputs.

The next two videos will focus on the feedforward process.

You will notice that in these videos I use subscripts as well as superscript as a numeric notation for the weight matrix.

For example:

* W\_k*Wk*​ is weight matrix k*k*
* \ W\_{ij}^k *Wijk*​ is the ij*ij* element of weight matrix k*k*

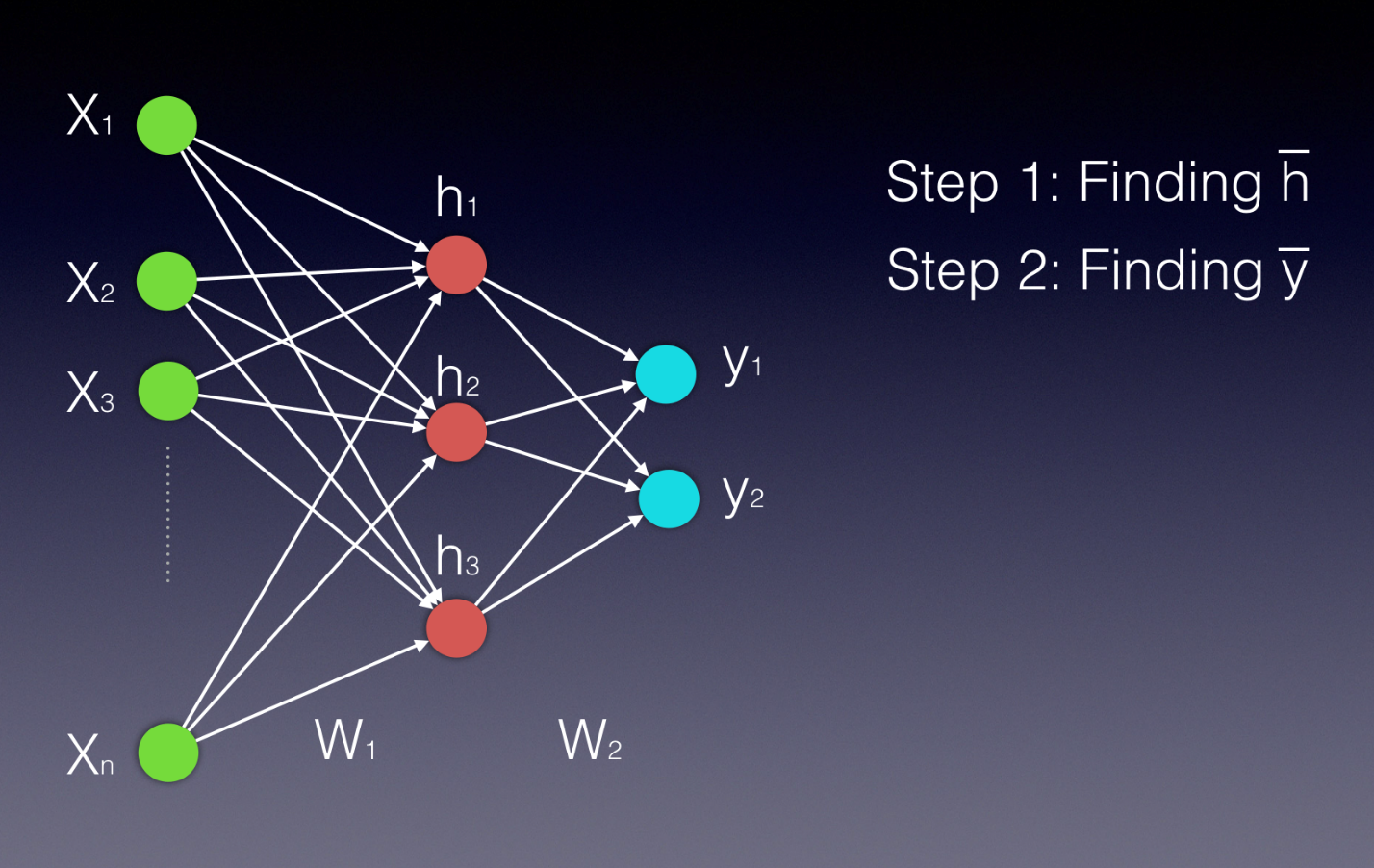
# Feedforward

In this section we will look closely at the math behind the feedforward process. With the use of basic Linear Algebra tools, these calculations are pretty simple!

If you are not feeling confident with linear combinations and matrix multiplications, you can use the following links as a refresher:

* [Linear Combination](http://linear.ups.edu/html/section-LC.html)
* [Matrix Multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication)

Assuming that we have a single hidden layer, we will need two steps in our calculations. The first will be calculating the value of the hidden states and the latter will be calculating the value of the outputs.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

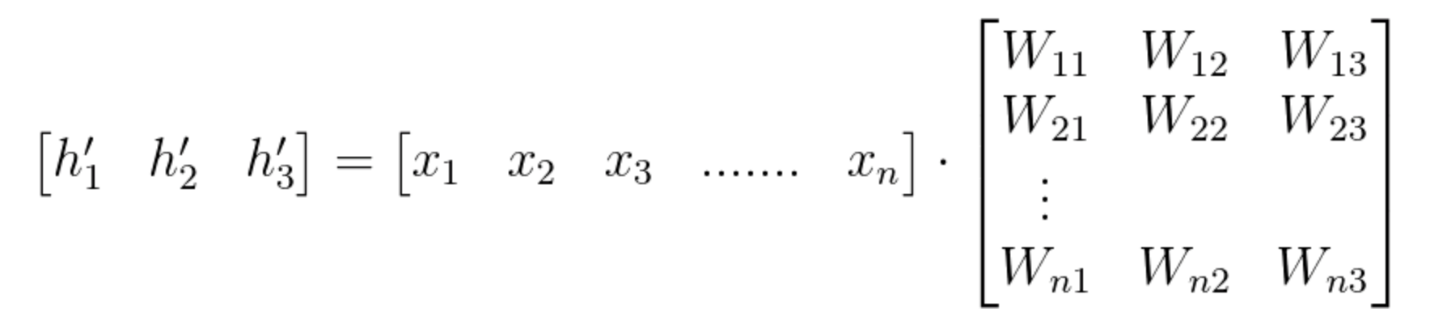
Notice that both the hidden layer and the output layer are displayed as vectors, as they are both represented by more than a single neuron.

Our first video will help you understand the first step- **Calculating the value of the hidden states**.

As you saw in the video above, vector h' of the hidden layer will be calculated by multiplying the input vector with the weight matrix W^{1}*W*1 the following way:

\bar{h'} = (\bar{x} W^1 )*h*′¯=(*x*¯*W*1)

Using vector by matrix multiplication, we can look at this computation the following way:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

[Equation 1](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

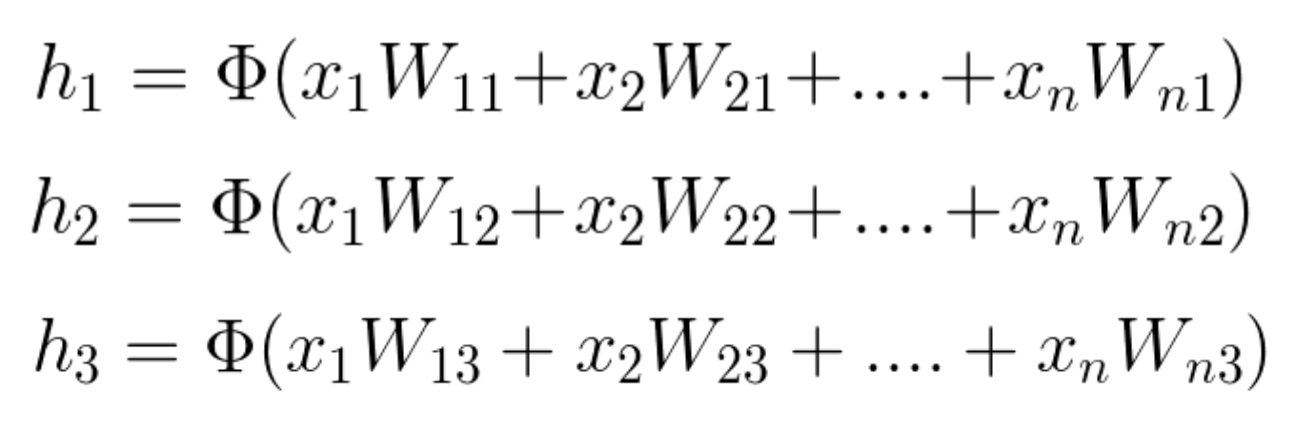
After finding h'*h*′ , we need an activation function (\PhiΦ) to finalize the computation of the hidden layer's values. This activation function can be a Hyperbolic Tangent, a Sigmoid or a ReLU function. We can use the following two equations to express the final hidden vector \bar{h}*h*¯:

\bar{h} = \Phi(\bar{x} W^1 )*h*¯=Φ(*x*¯*W*1)

or

\bar{h} = \Phi(h')*h*¯=Φ(*h*′)

Since W\_{ij}*Wij*​ 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)

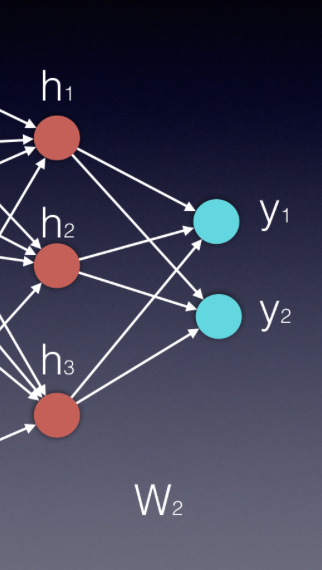
[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

[Equation 2](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

More information on the activation functions and how to use them can be found [here](https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions)

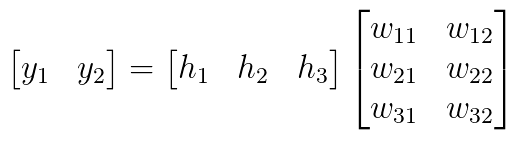
This next video will help you understand the second step- **Calculating the values of the Outputs**.

As you've seen in the video above, the process of calculating the output vector is mathematically similar to that of calculating the vector of the hidden layer. We use, again, a vector by matrix multiplication, which can be followed by an activation function. The vector is the newly calculated hidden layer and the matrix is the one connecting the hidden layer to the output.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

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}*h*¯ and the matrix is W^2*W*2, therefore \bar{y}=\bar{h}W^2*y*¯​=*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).

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

[Equation 3](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/29a326a7-32b9-46bb-93aa-10a09acef7c0)

The two error functions that are most commonly used are the [Mean Squared Error (MSE)](https://en.wikipedia.org/wiki/Mean_squared_error) (usually used in regression problems) and the [cross entropy](https://www.ics.uci.edu/~pjsadows/notes.pdf) (usually used in classification problems).

In the above calculations we used a variation of the MSE.

The next few videos will focus on the backpropagation process, or what we also call stochastic gradient decent with the use of the chain rule.

# Backpropagation Theory

Since partial derivatives are the key mathematical concept used in backpropagation, it's important that you feel confident in your ability to calculate them. Once you know how to calculate basic derivatives, calculating partial derivatives is easy to understand.  
For more information on partial derivatives use the following [link](http://www.columbia.edu/itc/sipa/math/calc_rules_multivar.html)

For calculation purposes in future quizzes of the lesson, you can use the following link as a reference for [common derivatives](http://tutorial.math.lamar.edu/pdf/Common_Derivatives_Integrals.pdf).

In the **backpropagation** process we minimize the network error slightly with each iteration, by adjusting the weights. The following video will help you understand the mathematical process we use for computing these adjustments.

If we look at an arbitrary layer k, we can define the amount by which we change the weights from neuron i to neuron j stemming from layer k as: \Delta W^kΔ*Wk*\_{ij}*ij*​.

The superscript (k) indicates that the weight connects layer k to layer k+1.

Therefore, the weight update rule for that neuron can be expressed as:

W\_{new} = W\_{previous} +\Delta W^k*Wnew*​=*Wprevious*​+Δ*Wk*\_{ij}*ij*​

Equation 4

The updated value \Delta W\_{ij}^kΔ*Wijk*​ is calculated through the use of the gradient calculation, in the following way:

\Delta W\_{ij}^k=\alpha (-\frac{\partial E}{\partial W})Δ*Wijk*​=*α*(−∂*W*∂*E*​), where \alpha*α* is a small positive number called the**Learning Rate**.

Equation 5

From these derivation we can easily see that the weight updates are calculated the by the following equation:

W\_{new}= W\_{previous} +\alpha (-\frac{\partial E}{\partial W} )*Wnew*​=*Wprevious*​+*α*(−∂*W*∂*E*​)

Equation 6

Since many weights determine the network’s output, we can use a vector of the partial derivatives (defined by the Greek letter Nabla \nabla∇) of the network error - each with respect to a different weight.

W\_{new}= W\_{previous}+\alpha \nabla\_W(-E)*Wnew*​=*Wprevious*​+*α*∇*W*​(−*E*)

Equation 7

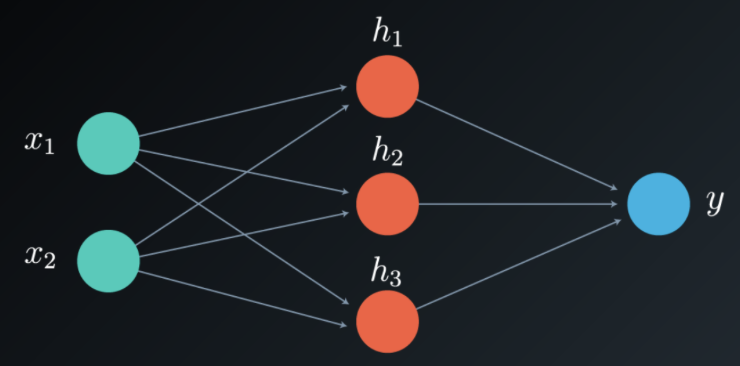
Here you can find other good resources for understanding and tuning the Learning Rate:

* [resource 1](http://blog.datumbox.com/tuning-the-learning-rate-in-gradient-descent/)
* [resource 2](http://cs231n.github.io/neural-networks-3/#loss)

The following video is given as a refresher to **overfitting** . You have already seen this concept in the Training Neural Networks lesson. Feel free to skip it and jump right into the next video.

# Backpropagation- Example (part a)

We will now continue with an example focusing on the backpropagation process, and consider a network having two inputs [x\_1, x\_2][*x*1​,*x*2​], three neurons in a single hidden layer [h\_1, h\_2, h\_3][*h*1​,*h*2​,*h*3​] and a single output y*y*.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

The weight matrices to update are W^1*W*1 from the input to the hidden layer, and W^2*W*2 from the hidden layer to the output. Notice that in our case W^2*W*2 is a vector, not a matrix, as we only have one output.

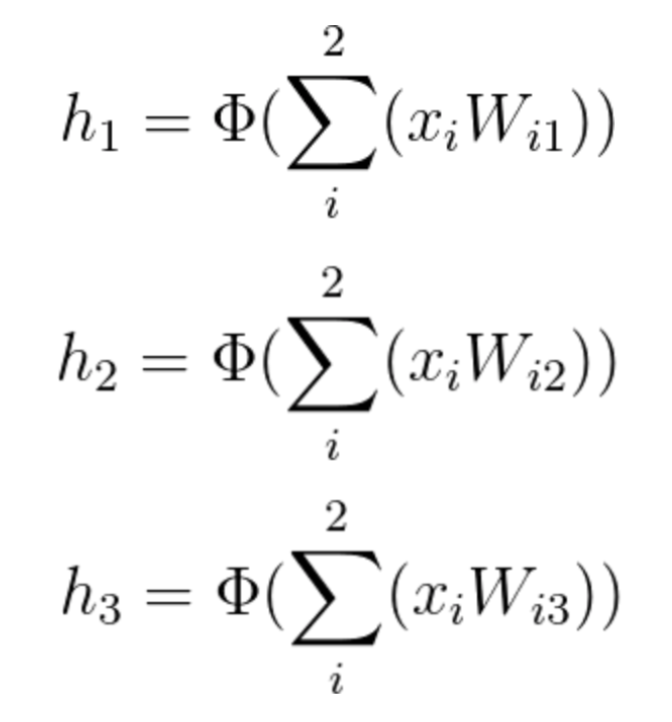
The chain of thought in the weight updating process is as follows:

To update the weights, we need the network error. To find the network error, we need the network output, and to find the network output we need the value of the hidden layer, vector \bar {h}*h*¯.

\bar{h}=[h\_1, h\_2, h\_3]*h*¯=[*h*1​,*h*2​,*h*3​]

Equation 8

Each element of vector \bar {h}*h*¯ is calculated by a simple linear combination of the input vector with its corresponding weight matrix W^1*W*1, followed by an activation function.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

[Equation 9](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

We now need to find the network's output, y*y*. y*y* is calculated in a similar way by using a linear combination of the vector \bar{h}*h*¯ with its corresponding elements of the weight vector W^2*W*2.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

[Equation 10](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

After computing the output, we can finally find the network error.

As a reminder, the two Error functions most commonly used are the [Mean Squared Error (MSE)](https://en.wikipedia.org/wiki/Mean_squared_error) (usually used in regression problems) and the [cross entropy](https://www.ics.uci.edu/~pjsadows/notes.pdf) (often used in classification problems).

In this example, we use a variation of the MSE:

E=\frac{(d-y)^2}{2}*E*=2(*d*−*y*)2​,

where d*d* is the desired output and y*y* is the calculated one. Notice that **y** and **d** are not vectors in this case, as we have a single output.

The error is their squared difference, E=(d-y)^2*E*=(*d*−*y*)2, and is also called the network's **Loss Function**. We are dividing the error term by 2 to simplify notation, as will become clear soon.

The aim of the backpropagation process is to minimize the error, which in our case is the Loss Function. To do that we need to calculate its partial derivative with respect to all of the weights.

Since we just found the output y, we can now minimize the error by finding the updated values \Delta W\_{ij}^kΔ*Wijk*​. The superscript k indicates that we need to update each and every layer k.

As we noted before, the weight update value \Delta W\_{ij}^kΔ*Wijk*​ is calculated with the use of the gradient the following way:

\Delta W\_{ij}^k=\alpha (-\frac{\partial E}{\partial W})Δ*Wijk*​=*α*(−∂*W*∂*E*​)

Therefore:

\Delta W\_{ij}^k=\alpha (-\frac{\partial E}{\partial W})=-\frac{\alpha}{2} \frac{\partial (d-y)^2}{\partial W\_{ij}}=-2 \frac{\alpha}{2}(d-y) \large \frac{\partial (d-y)}{\partial W\_{ij}}Δ*Wijk*​=*α*(−∂*W*∂*E*​)=−2*α*​∂*Wij*​∂(*d*−*y*)2​=−22*α*​(*d*−*y*)∂*Wij*​∂(*d*−*y*)​

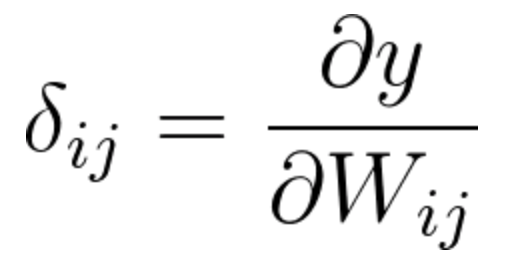
which can be simplified as:

\Delta W\_{ij}^k=\alpha(d-y) \frac{\partial y}{\partial W\_{ij}}Δ*Wijk*​=*α*(*d*−*y*)∂*Wij*​∂*y*​

Equation 11

(Notice that d*d* is a constant value, so it’s partial derivative is simply a zero)

This partial derivative of the output with respect to each weight, defines the gradient and is often denoted by the Greek letter \delta*δ*.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

[Equation 12](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/00d69708-34de-45a2-a93d-4213f0cea51e)

We will find all the elements of the gradient using the chain rule.

If you are feeling confident with the **chain rule** and understand how to apply it, skip the next video and continue with our example. Otherwise, give Luis a few minutes of your time as he takes you through the process!

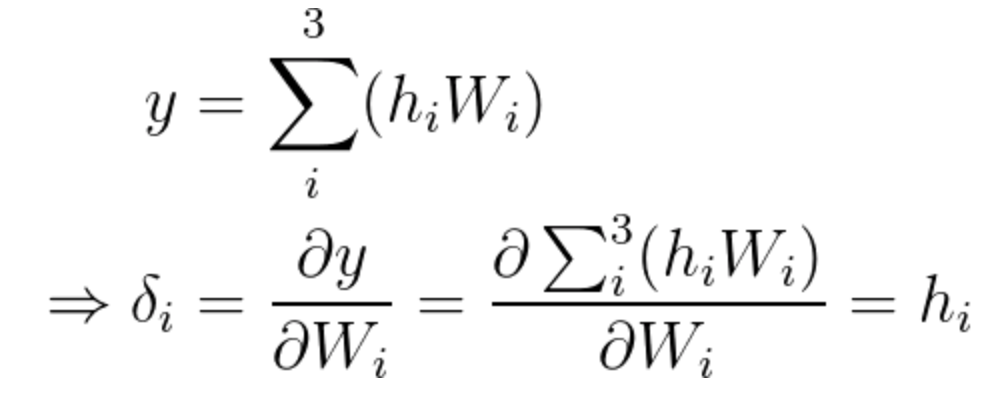
# Backpropagation- Example (part b)

Now that we understand the chain rule, we can continue with our **backpropagation** example, where we will calculate the gradient

In our example we only have one hidden layer, so our backpropagation process will consist of two steps:

Step 1: Calculating the gradient with respect to the weight vector W^2*W*2 (from the output to the hidden layer).   
Step 2: Calculating the gradient with respect to the weight matrix W^1*W*1 (from the hidden layer to the input).

**Step 1** (Note that the weight vector referenced here will be W^2*W*2. All indices referring to W^2*W*2 have been omitted from the calculations to keep the notation simple).

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 13](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

As you may recall:

\large\Delta W\_{ij}=\alpha(d-y) \frac{\partial y}{\partial W\_{ij}}Δ*Wij*​=*α*(*d*−*y*)∂*Wij*​∂*y*​

In this specific step, since the output is of only a single value, we can rewrite the equation the following way (in which we have a weights vector):

\large\Delta W\_i=\alpha(d-y) \frac{\partial y}{\partial W\_i}Δ*Wi*​=*α*(*d*−*y*)∂*Wi*​∂*y*​

Since we already calculated the gradient, we now know that the incremental value we need for step one is:

\Delta W\_i=\alpha(d-y) h\_iΔ*Wi*​=*α*(*d*−*y*)*hi*​

Equation 14

Having calculated the incremental value, we can update vector W^2*W*2 the following way:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 15](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

**Step 2** (In this step, we will need to use both weight matrices. Therefore we will not be omitting the weight indices.)

In our second step we will update the weights of matrix W^1*W*1 by calculating the partial derivative of y*y* with respect to the weight matrix W^1*W*1.

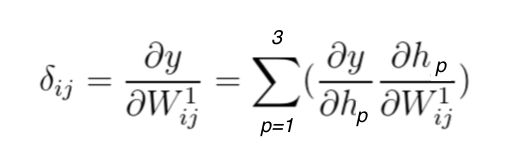
The chain rule will be used the following way:

obtain the partial derivative of y*y* with respect to \bar{h}*h*¯, and multiply it by the partial derivative of \bar{h}*h*¯ with respect to the corresponding elements in W^1*W*1. Instead of referring to vector \bar{h}*h*¯, we can observe each element and present the equation the following way:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

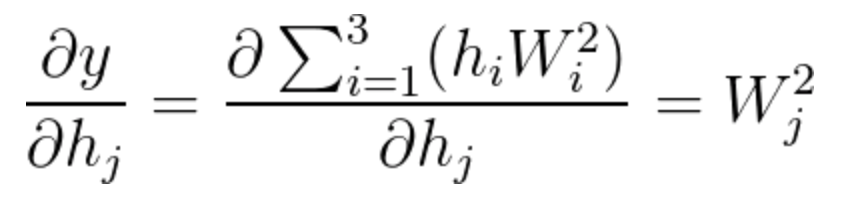
[Equation 16](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

In this example we have only 3 neurons the the single hidden layer, therefore this will be a linear combination of three elements:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 17](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

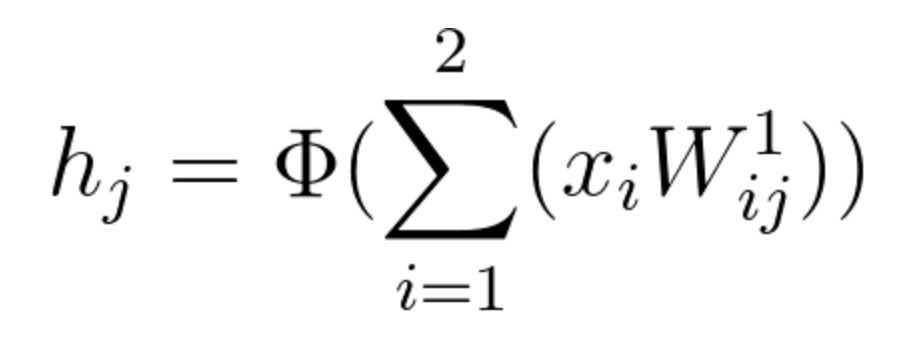
We will calculate each derivative separately. \frac{\partial y}{\partial h\_j}∂*hj*​∂*y*​ will be calculated first, followed by \frac{\partial h\_j}{\partial W^1\_{ij}}∂*Wij*1​∂*hj*​​.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 18](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

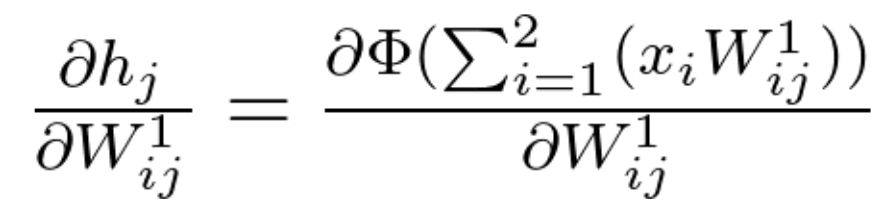
Notice that most of the derivatives were zero, leaving us with the simple solution of \frac{\partial y}{\partial h\_{j}}=W^2\_j∂*hj*​∂*y*​=*Wj*2​

To calculate \frac{\partial h\_j}{\partial W^1\_{{ij}}}∂*Wij*1​∂*hj*​​ we need to remember first that

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

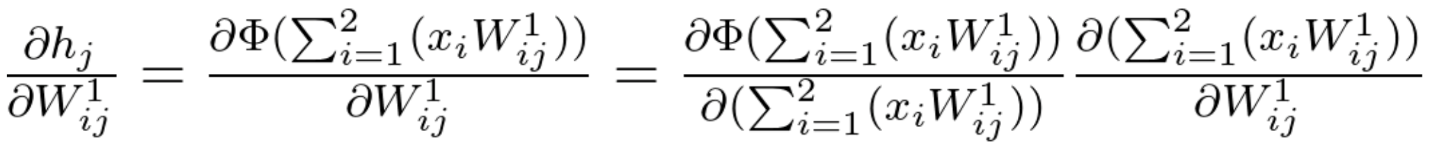
[Equation 19](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

Therefore:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 20](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

Since the function \ h\_j *hj*​ is an activation function (\PhiΦ) of a linear combination, its partial derivative will be calculated the following way:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 21](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

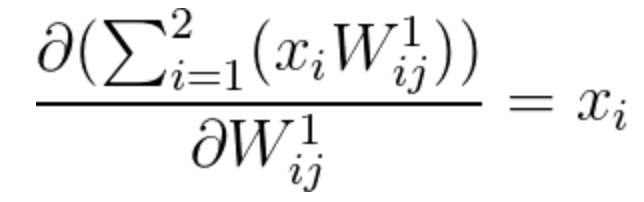
Given that there are various activation functions, we will leave the partial derivative of \PhiΦ using a general notation. Each neuron j will have its own value for \PhiΦ and \Phi'Φ′, according to the activation function we choose to use.

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 22](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

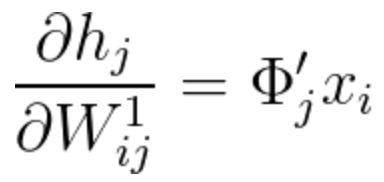
The second calculation of equation 21 can be calculated the following way:

(Notice how simple the result is, as most of the components of this partial derivative are zero).

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

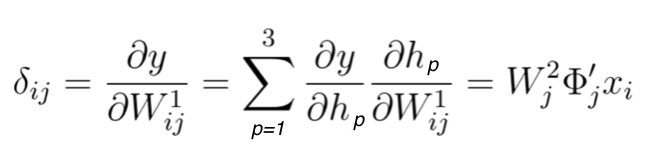
[Equation 23](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

After understanding how to treat each multiplication of equation 21 separately, we can now summarize it the following way:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

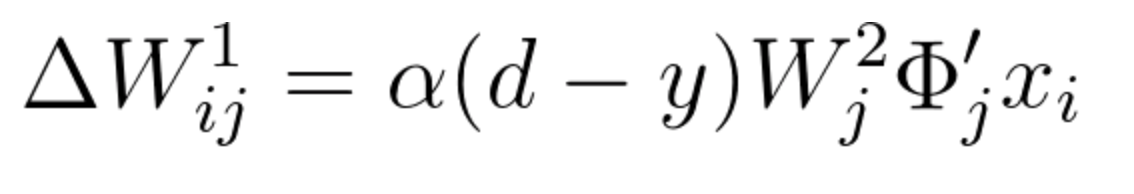
[Equation 24](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

We are ready to finalize **step 2**, in which we update the weights of matrix W^1*W*1 by calculating the gradient shown in equation 17. From the above calculations, we can conclude that:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 25](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

Since  , when finalizing step 2, we have:

[[](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

[Equation 26](https://classroom.udacity.com/nanodegrees/nd101/parts/0d85c39f-2ac0-49b3-98b0-fd40e9180cf4/modules/dd68b0bb-dd81-436f-bb04-e2a34ed349b8/lessons/74236975-4329-4704-9890-85b51f3f35fa/concepts/514cf09c-9651-4877-8b2c-d651bca3cd16)

Having calculated the incremental value, we can update vector W^1*W*1 the following way:

​+*α*(*d*−*y*)*Wj*2​Φ*j*′​*xi*​

Equation 27

After updating the weight matrices we begin once again with the Feedforward pass, starting the process of updating the weights all over again.

This video touches on the subject of Mini Batch Training. We will further explain things in our **Hyperparameters** lesson coming up.