Course 1: Neural Networks and Deep Learning

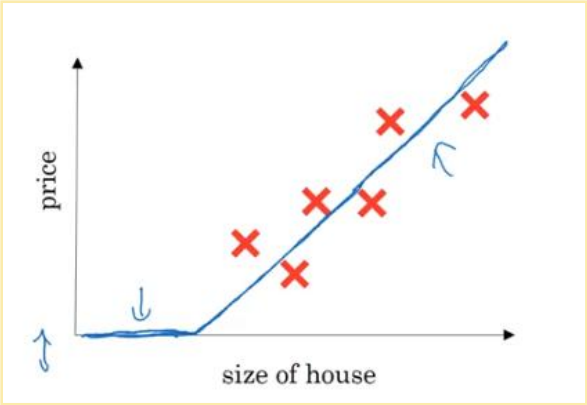
Content

1. Introduction to Deep Learning

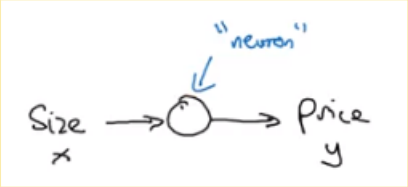
Starting about 100 years ago, the electrification of our society transformed every major industry, every ranging from transportation, manufacturing, to healthcare, to communications and many more. And today, we see a surprisingly clear path for AI to bring about an equally big transformation. And of course, the part of AI that is rising rapidly and driving a lot of these developments, is deep learning. So today, deep learning is one of the most highly sought after skills and technology worlds.

1.1. What is a neural network?

Let's suppose we have some points relating size of a house and its price. If we know linear regression, we can fit a straight line to the points. Since price cannot be negative, we can bend the curve when price reaches 0.



We can think of this as a **very simple neural network**:

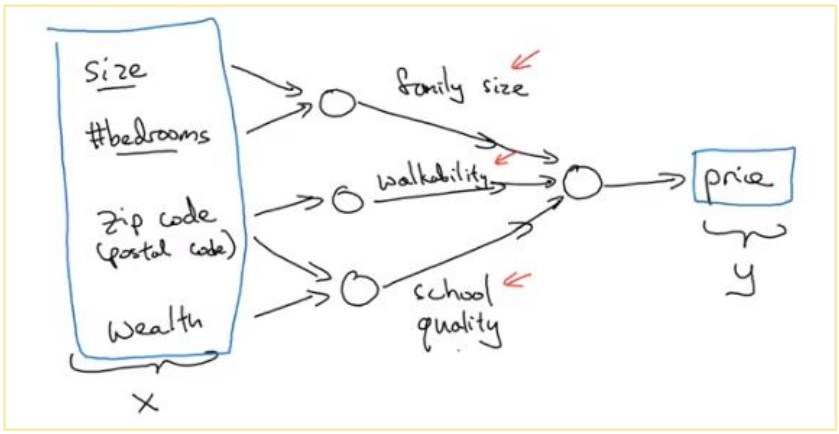


And all the neuron does is it inputs the size, computes this linear function, takes a max of zero, and then outputs the estimated price.

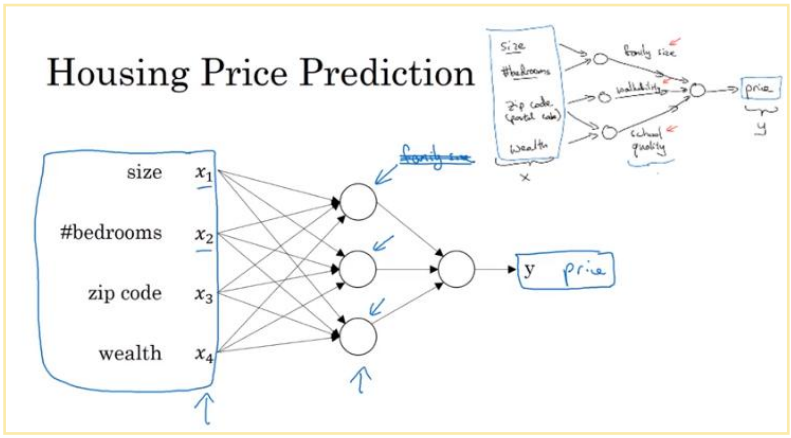
In the neural network literature, we can see this function a lot. This function is called a **ReLU** function which stands for rectified linear unit. "Rectified" just means taking a max of 0 which is why you get a function shape like this. We'll see them again later in the course.

A larger neural network is then formed by taking many of the single neurons and stacking them together. For example, in the image below, we can have more features like size, number of bedrooms, zip code and wealth. Then, we can combine these features with single neurons (for example, using ReLU (or any other slightly non-linear function) to obtain other features (for example, size and number of bedrooms can serve as a proxy for family size). With these new features we can

then predict the price. The **magic** of NN is that you only have to provide x and not the "intermediate features". They will figure out by themselves.



What we actually implement is this:



But, rather than saying the hidden nodes represent family size, and that it is connected to the first two features, we're going to **let the NN decide** whatever it wants it to be, and whatever connections may it have (if we state that every feature is connected to every node we talk about **density layers**). And it turns out that given enough training examples with both x and y, neural networks are **remarkably good** at figuring out functions that accurately map from x to y.

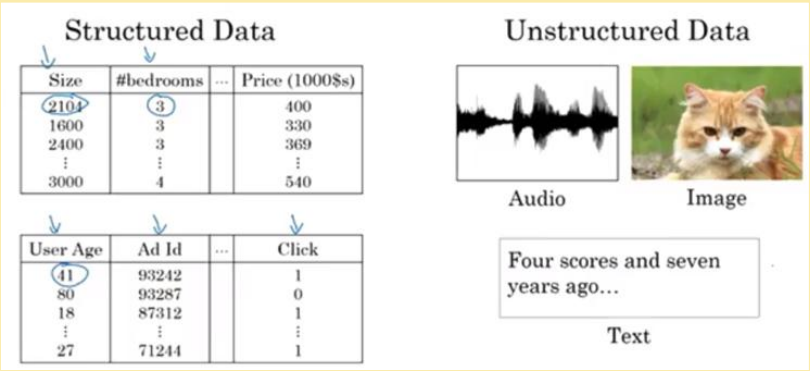
1.2. Supervised Learning with Neural Networks

Almost all the economic value created by neural networks has been through one type of machine learning, called supervised learning. Here we can see some examples:

Input(x)	Output (y)	Application
Home features	Price	Real Estate
Ad, user info	Click on ad? (0/1)	Online Advertising
Image	Object (1,...,1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info	Position of other cars	Autonomous driving

And the corresponding neural network types that work well in each application.

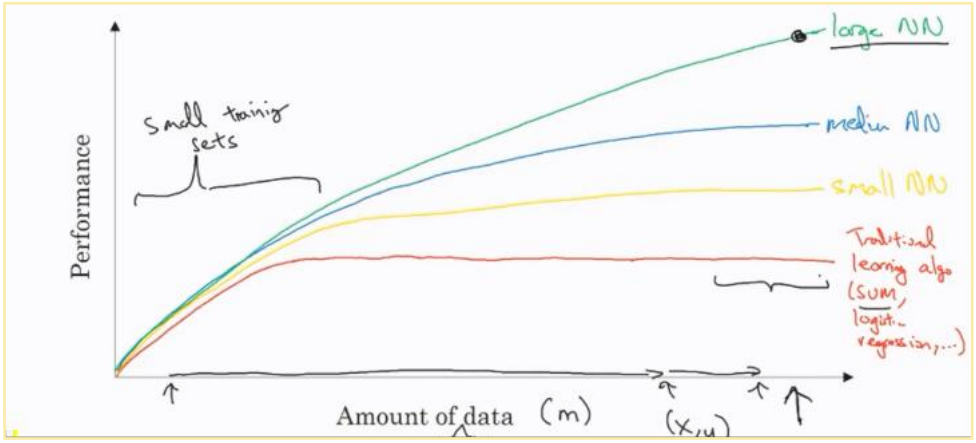
We can apply these techniques with either **structured** or **unstructured** data:



So neural networks have transformed supervised learning and are creating tremendous economic value. It turns out though, that the basic technical ideas behind neural networks have mostly been around, sometimes for many decades. So why is it, then, that they're only just now taking off and working so well?

1.3. Why is Deep Learning taking off?

Let's say we plot a figure where on the horizontal axis we plot the amount of data we have for a task and let's say on the vertical axis we plot the performance on above learning algorithms such as the accuracy:



The ability to scale from a data, computation or algorithmic view is what has turned NN so powerful. In addition, as training a NN is an iterative process, faster computation leads to higher efficiency.

2. Neural Network Basics

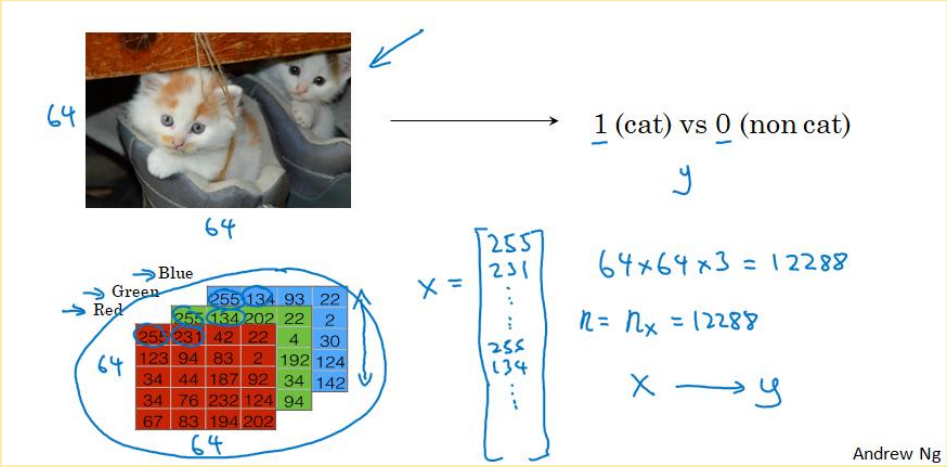
2.1. Logistic Regression as a Neural Network

2.1.1. Binary Classification

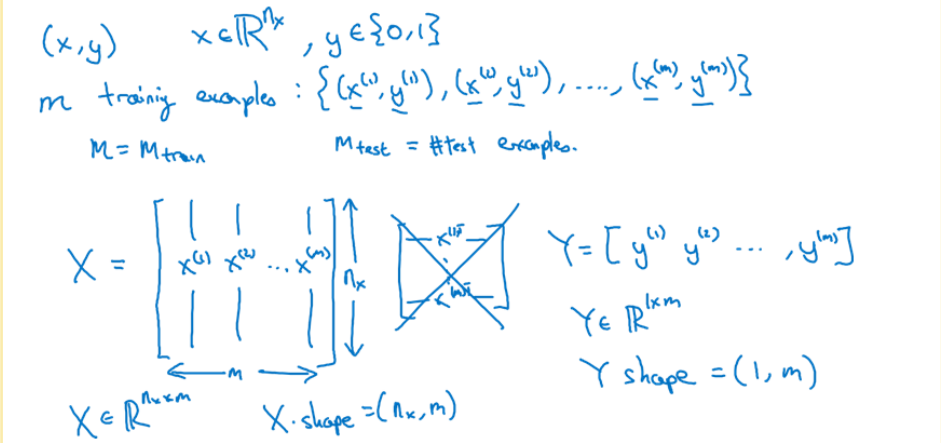
Before going into neural networks in detail and explain the logistic regression seen as a NN, let's talk about the problem we want to solve: binary classification.

For example, let's classify images between cats or non cats. In an image, we have three channels (red, green and blue) and a 64x64 matrix on each one. Each element represents the intensity of each pixel in each channel.

We'll transform these matrices into a feature vector:



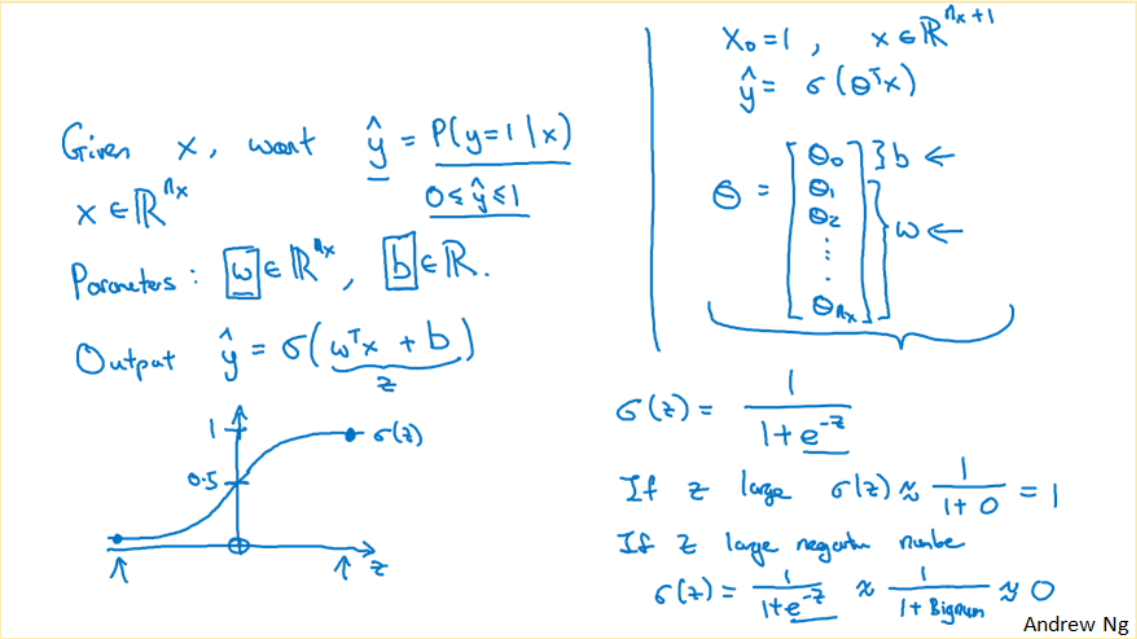
We'll also see some notation:



2.1.2. *Logistic Regression*

This is a learning algorithm that you use when the output labels Y in a supervised learning problem are all either **zero** or **one**.

The output will be the linear regression functional form applied to the sigmoid function.



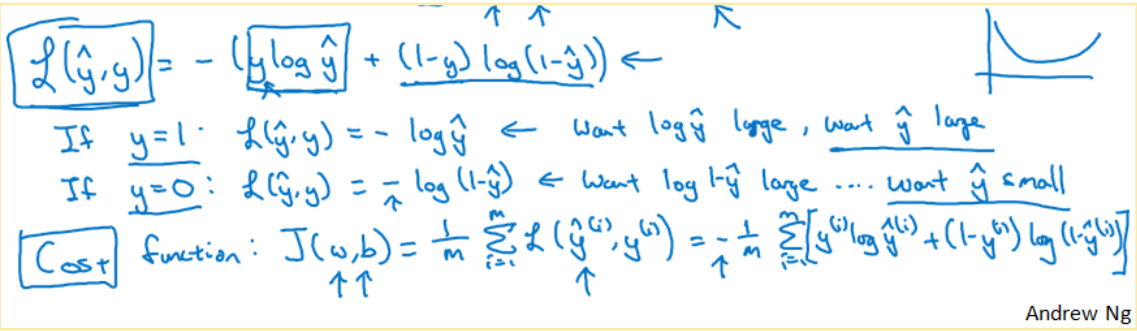
The top right annotations show a different way of notating a logistic regression model. However, in this course we will not use it.

We will need to estimate the parameters *w* and *b*

The next step is to define a cost function to define the model parameters. We could define this function:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

But it would lead us to a non-convex optimization problem. For that reason, we'll use this one instead:



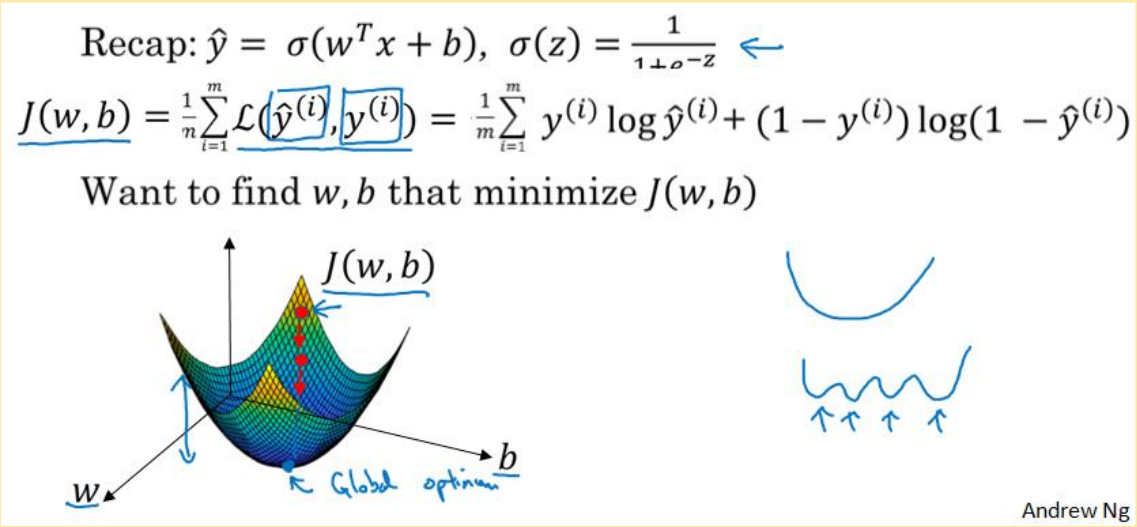
And why does this make sense? If $y = 1$, we want to predict a high $\hat{y}$. The higher it is, the less loss we'll have. The same logic applies when $y = 0$.

Once we have defined the **loss** function L, for a single data point, we need to define the **cost** function that we will compute for all the training points. That is the cost function J.

It turns out that, as we have said, the logistic regression is a way of seeing a simple neural network.

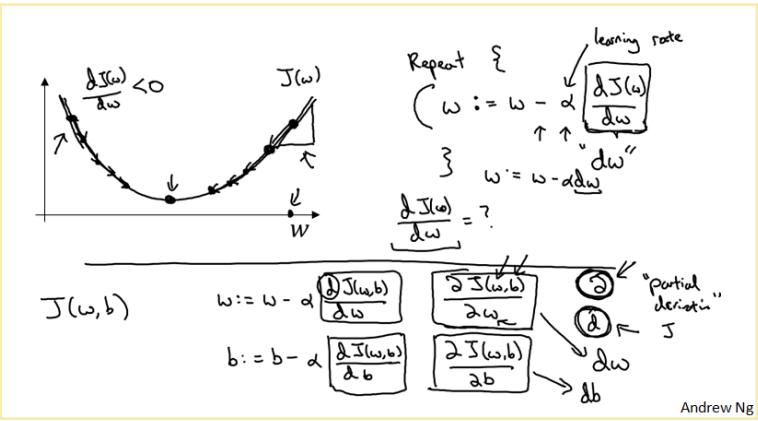
2.1.3. Gradient Descent

Let's see how we can use the **gradient descent** algorithm to train (or learn) the w and b parameters on our training set.



That is the cost function represented in a two-dimensional space. We want to minimize the loss by varying the w and b parameters. The cost function as we have defined it is convex, so we can initialize the parameters at some value. Since this function is convex, we can initialize at any value, since it will always reach for the minimum.

Let's project this example into a two dimensional space and forget b for a moment:

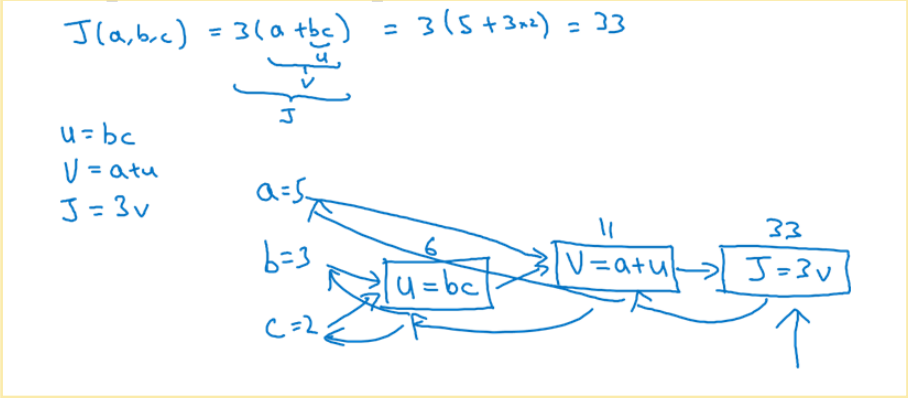


By iterating we can reach near the minimum.

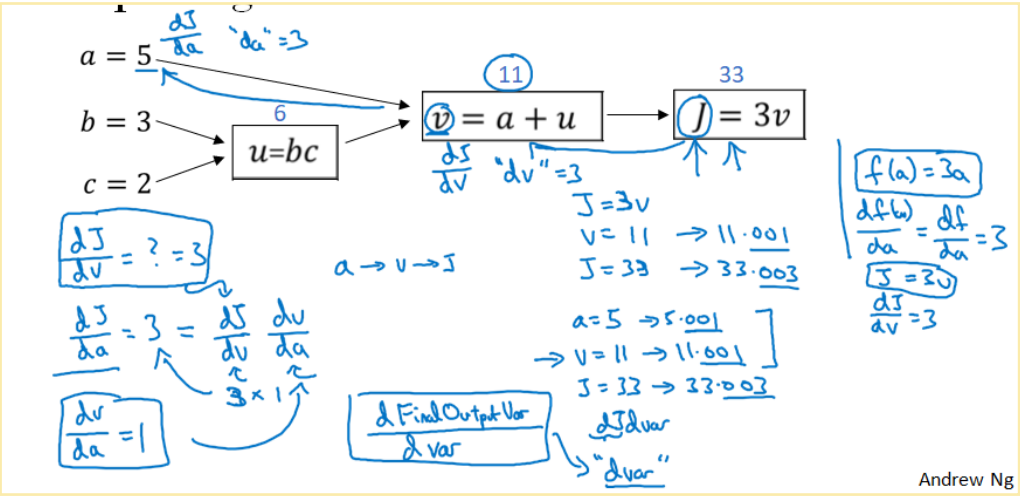
2.1.4. Computation graph

The computations of a neural network are organized in terms of a forward pass or a **forward propagation** step, in which we compute the output of the neural network, followed by a backward pass or **back propagation step**, which we use to compute gradients or compute derivatives. The computation graph explains why it is organized this way.

Let's suppose we have a function J that is calculated as the image shows. We can divide that calculation into three smaller calculations and then put them into a graph:



The left-to-right process can help us compute the cost function value. Now, let's suppose we need to calculate the derivative of J with respect to v. Or what is the same, if v changed a little bit, how would the value of J change?



We had that $v = 11$. If we change it by 0.001, we then increase J by 0.003. That is because the derivative is 3.

Now, if we wanted to calculate the derivative of J with respect to a, we could use the calculus' chain rule to calculate dJ/dv and then dv/da to finally get dJ/da. This

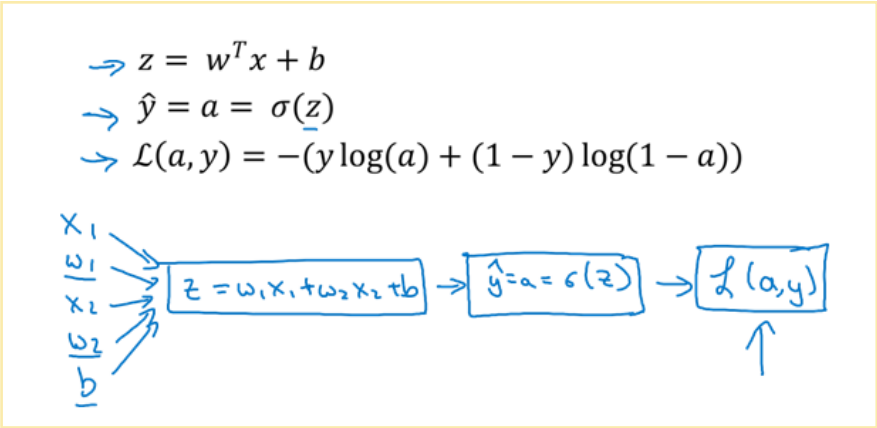
means we can go from right to left doing back propagation steps to compute derivatives.

As a matter of notation, when we code, if we calculate the derivative of J with respect to some variable, we will call it "*dvar*".

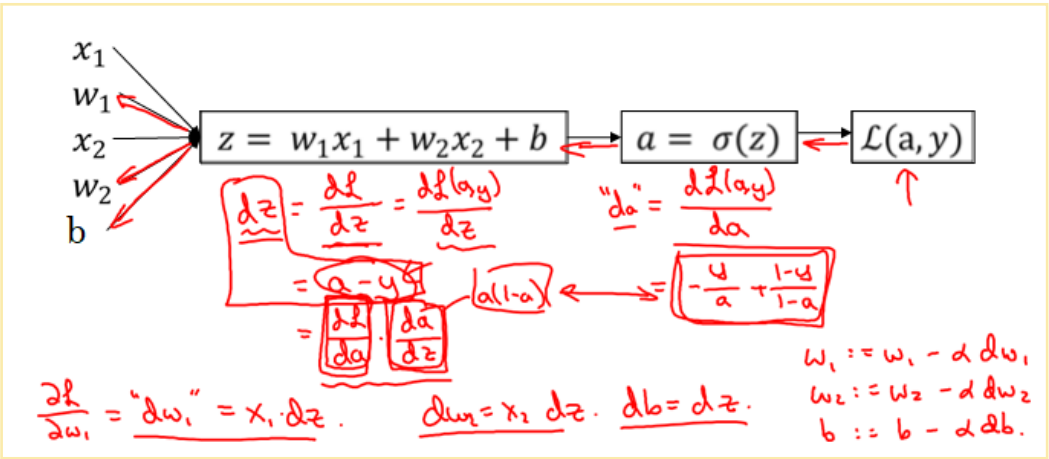
So, as a summary, we can do **forward propagation** to compute the value of the **cost function**, and **backpropagation** to compute its **derivative**.

2.1.5. Logistic Regression Gradient Descent

Let's recap the logistic regression model:

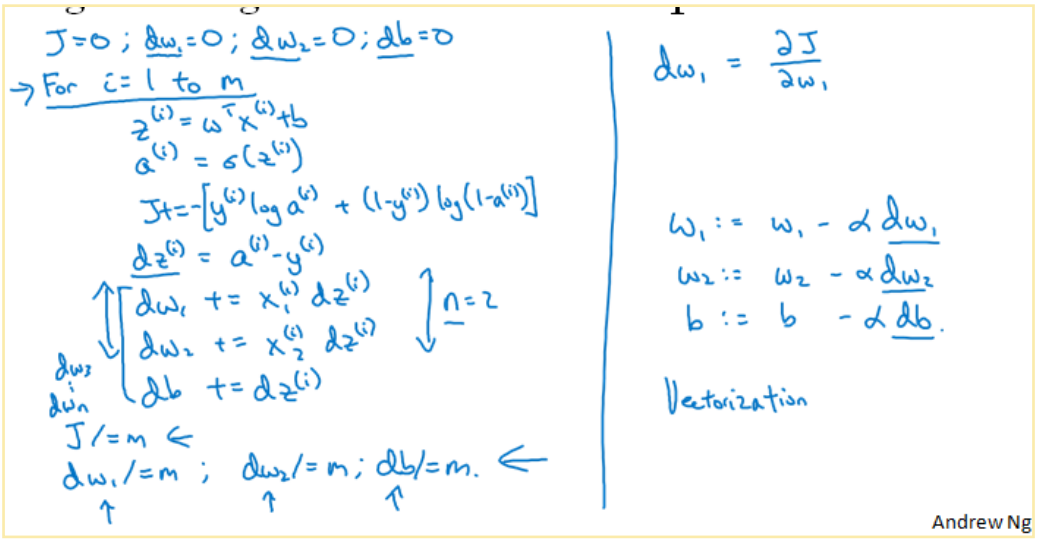


This is our computation graph. We want to change the parameters (underlined) to minimize the loss.



We are applying the chain rule in the particular case of logistic regression. So, if we wanted to implement gradient descent, we would need to calculate dz (notation for $dL/dz) = a - y$ and then $dL/dw_1 = x_1 * dz$. And w_2 and b are updated as in the image.

To do this for the full training dataset, we'll do:



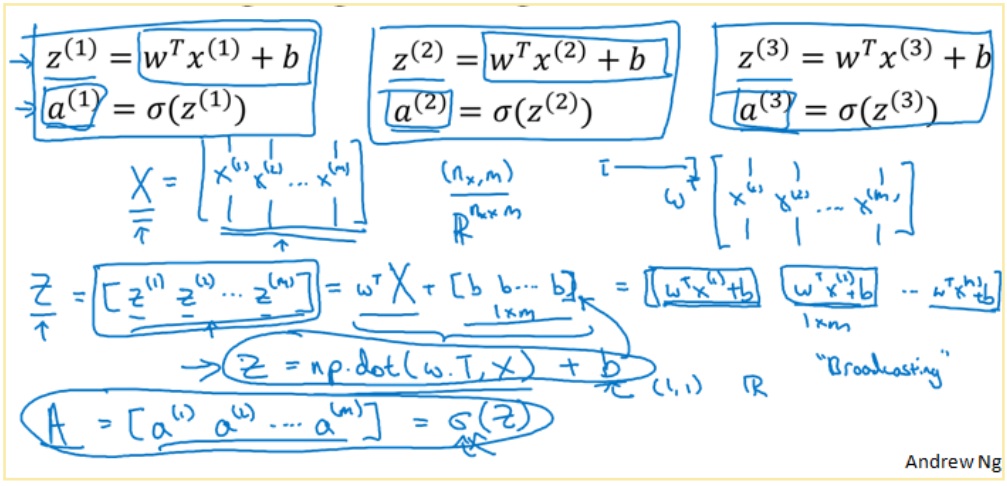
We obtain that right-side formulas to update the parameters. The values used in them come from the left-side, which is a representation of the gradient descent algorithm.

Having explicit for loops makes algorithms run **less efficiently**. We'll use vectorization techniques to fasten these processes.

2.1.6. Vectorization

Using vectorization to, as an example, do a matrix product, takes 300 times less time than with an explicit for loop. We'll need to remember to vectorize our code when implementing neural networks. In addition, GPUs are really better than CPU when parallelizing operations and executing SIMD (single instruction multiple data).

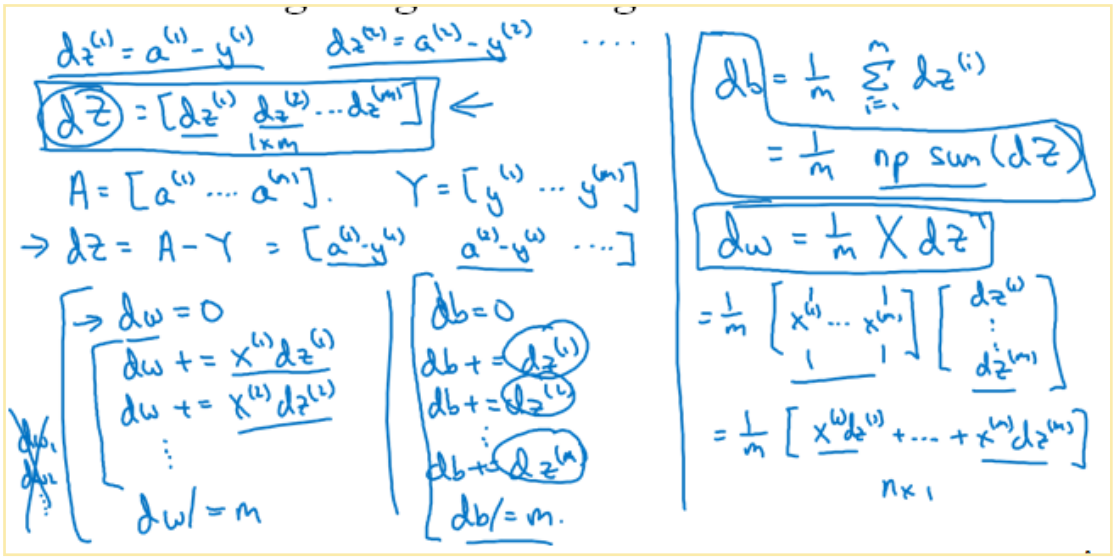
We'll see how to vectorize the logistic regression algorithm.



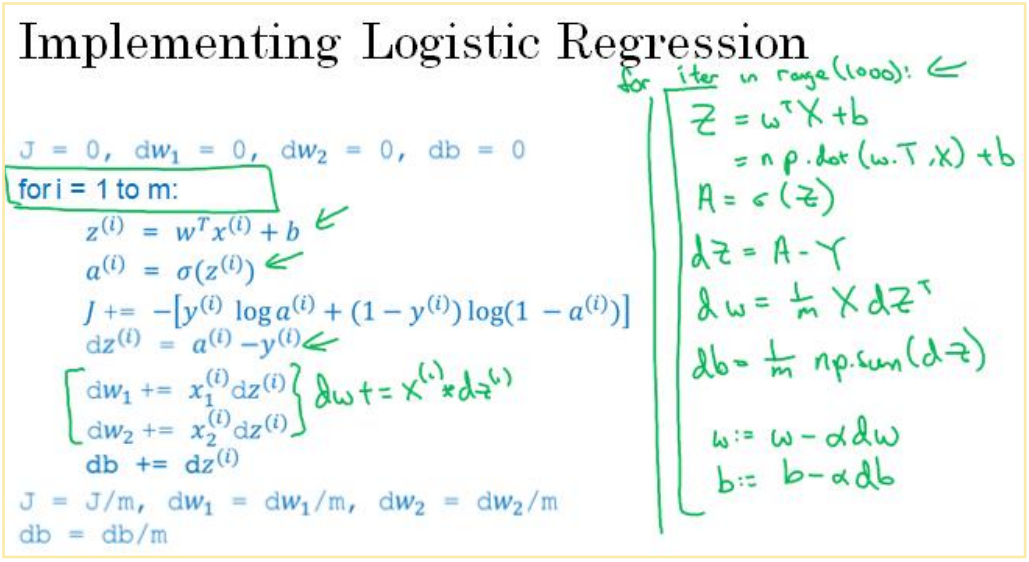
We can calculate the *Z* vector in a simple operation (without loops) and then compute the sigmoid function to obtain *A*.

Note that, in order to compute, for example, *A*, we need to apply the sigmoid function to every element. But with vectorization, we can do that in a parallelized way instead of in a sequential way.

We can also compute the gradient calculations with vectorization:



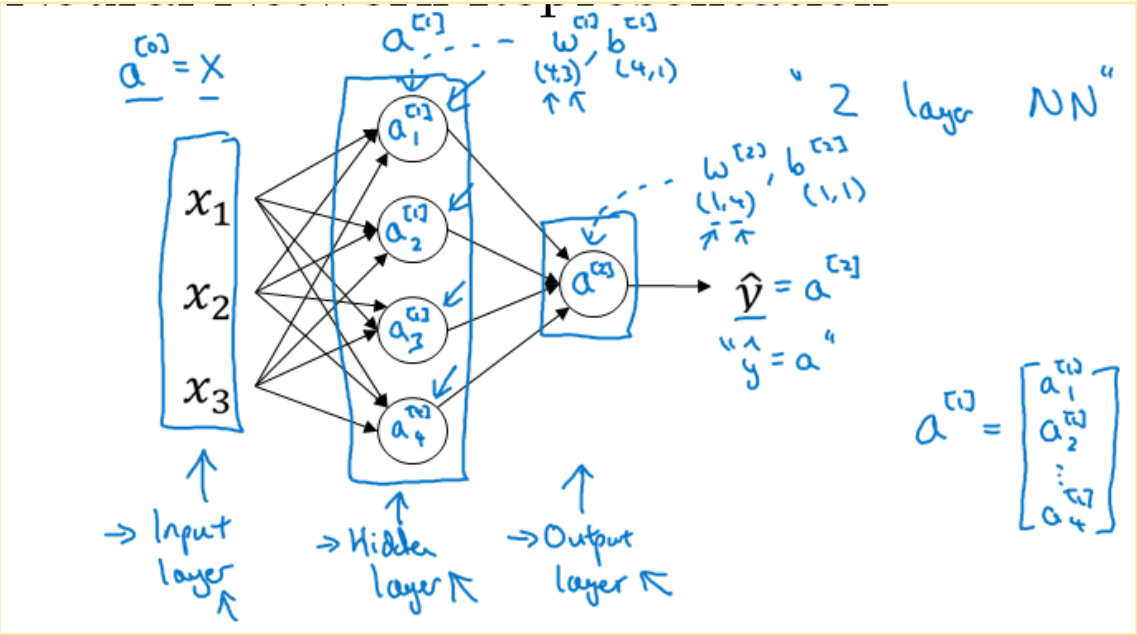
Now, as a summary, let's check how to implement logistic regression in a vectorized way:



On the left we have our highly inefficient non vectorized implementation. On the right, the efficient version.

3. Shallow Neural Network

Let's see how a two layer NN looks like:



First of all, we'll fix some notation. $a^{[0]}$ represents the activation vector of the layer 0 (or *input* layer). We are calling it a because, in the case of sigmoid activation functions, it represents the sigmoid transformation as we have seen in the previous lessons. This layer corresponds to the feature values.

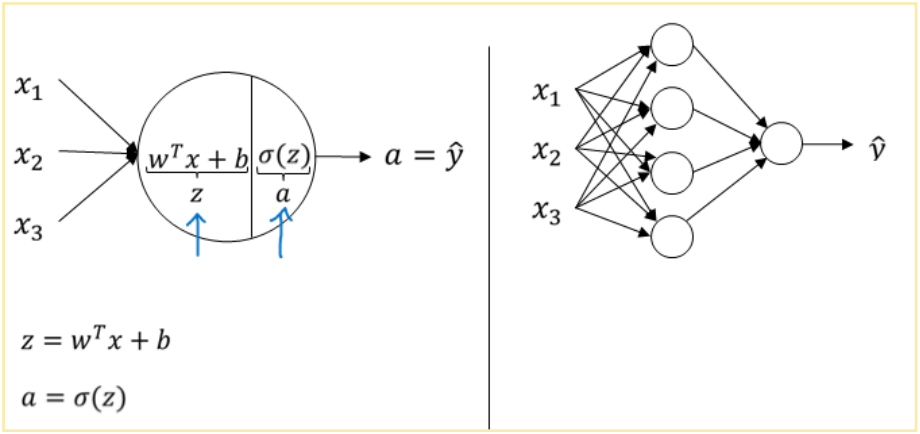
Then, we have the hidden layer which is composed by 4 neurons. Each neuron has its own activation term. This a term has, as well, its own w and b parameters.

Finally we have the output layer, which will correspond to $\hat{y}$.

Now, we'll see how exactly the neural network computes the output:

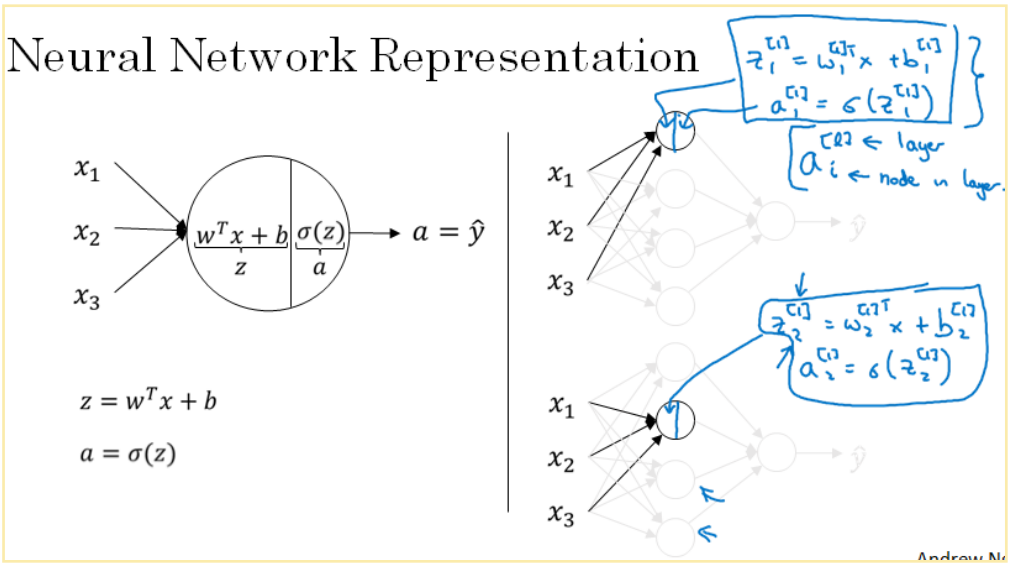
3.1. Computing a Neural Network's Output

Basically, a NN with sigmoid activation functions computes the logistic regression a lot of times.

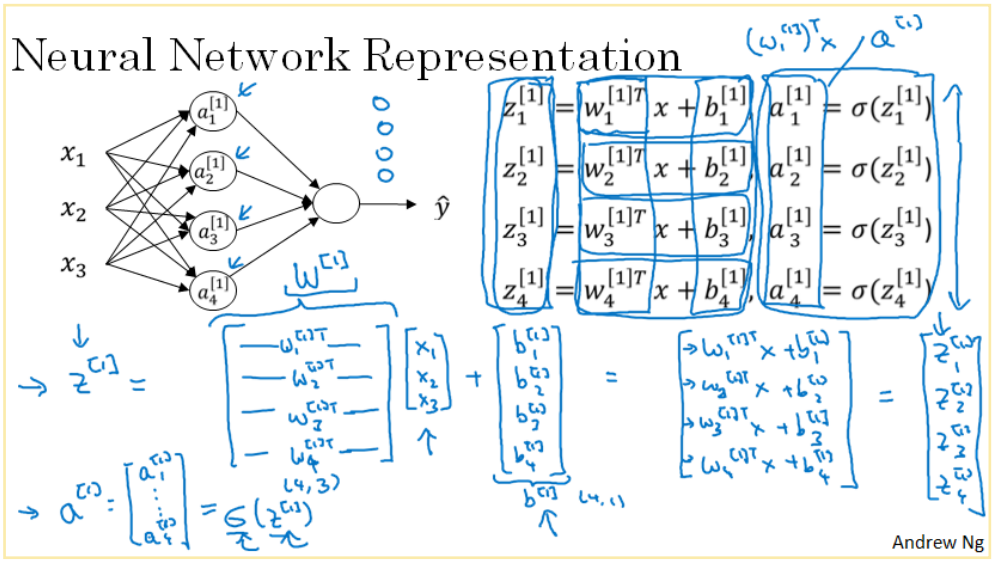


We've said before that the circle in logistic regression represents two computation steps: the computation of z and then the activation as a sigmoid function of z.

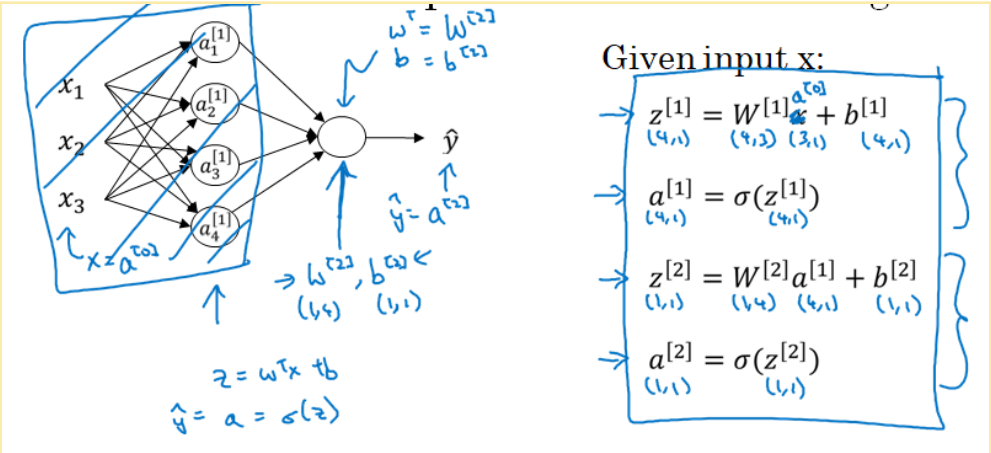
Let's focus on one node in the hidden layer.



z is calculated and then a is in every node. Here we have the calculations:



Recall that for each node, we have a w parameter vector. For that reason we get a W matrix of (4,3) shape.

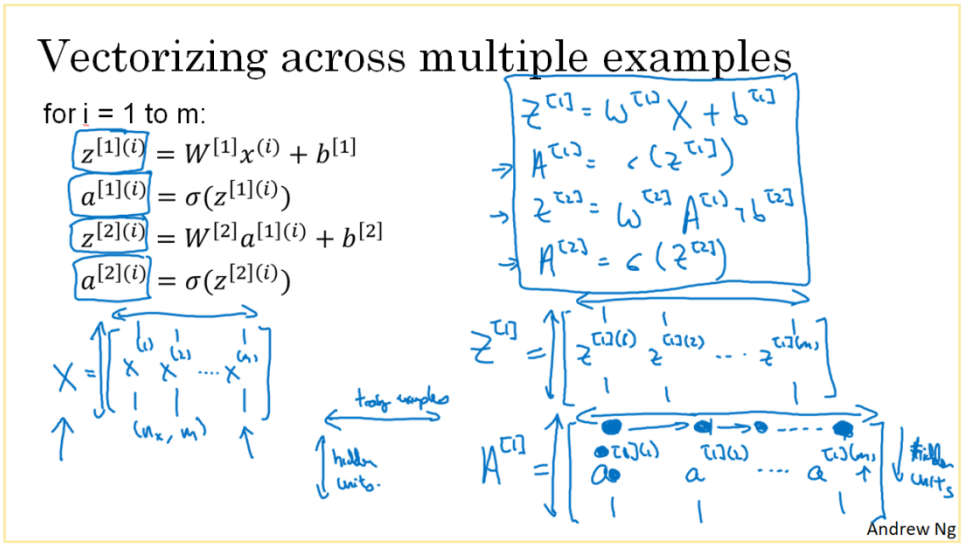


Then, for the output layer, we again have a logistic regression that takes as an input the activations of the nodes of the hidden layer.

To compute the output of the NN we only need those four lines of code. Recall that this neural network takes as an input a single data point with three features. We will now expand the calculation to the whole training set.

Recall that m represents the number of training data points. A simple solution would be to do a for loop for every training example. That way we'd get m and $\hat{y}$ values, one for each training data point.

Also recall that our training data matrix is like in the following picture:



That is, we have it transposed with respect to what we would normally expect. This means each column is a training data point, and each row corresponds to the value of a feature for each training example.

In the NN with the simple data point we had a single column of the X matrix. Now we are just stacking up in columns the different data points. We can do the same thing to the Z and A matrices.

In addition, we can interpret the A matrix as the following: as we can see on the bottom right corner of the image, if we move horizontally we are moving through the data points, and moving vertically means moving through nodes in the layer. Recall that we will have an A matrix for every layer.

A similar intuition holds to the Z matrix.

3.2. Activation functions

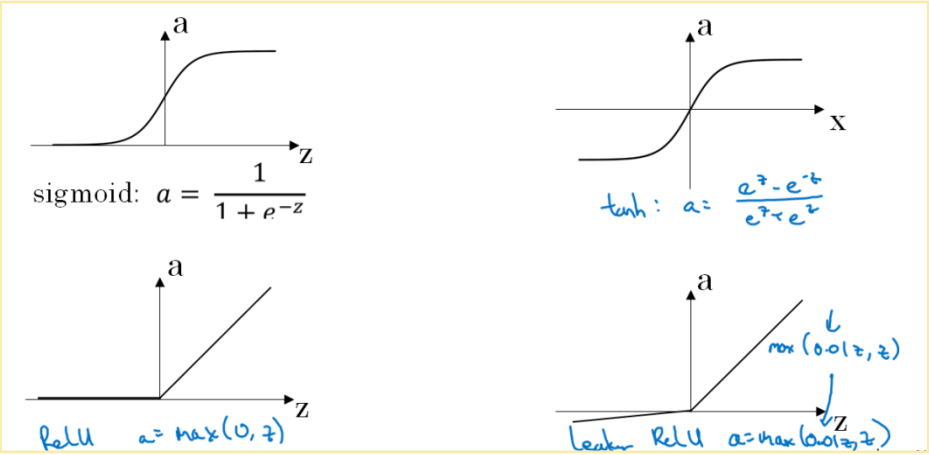
When we build our neural network, one of the choices we get to make is what **activation function** to use in the hidden layers, as well as what is the output unit of our neural network.

We have been using sigmoid function, but other ones can perform better (i.e. ReLU, tanh…). In fact, the *tanh* almost always works better than the sigmoid function. The activation functions **can be different in different layers**.

Some rules of thumb when choosing activation functions are:

- If we have a binary classification problem, the output layer should have a sigmoid activation function.
- Don't use sigmoid function in hidden layers because tanh function outperforms it.
- When using the tanh function, for really big or really small values of z we get a really small slope. This means the gradient will be small, and the NN will learn slowly.
- This does not happen with the ReLU function. For half of the space, we have a derivative really different to 0.
- Another choice is the leaky ReLu, that deals with the fact that when using the ReLU function, the other half of the space has a 0 derivative, meaning the network won't learn.

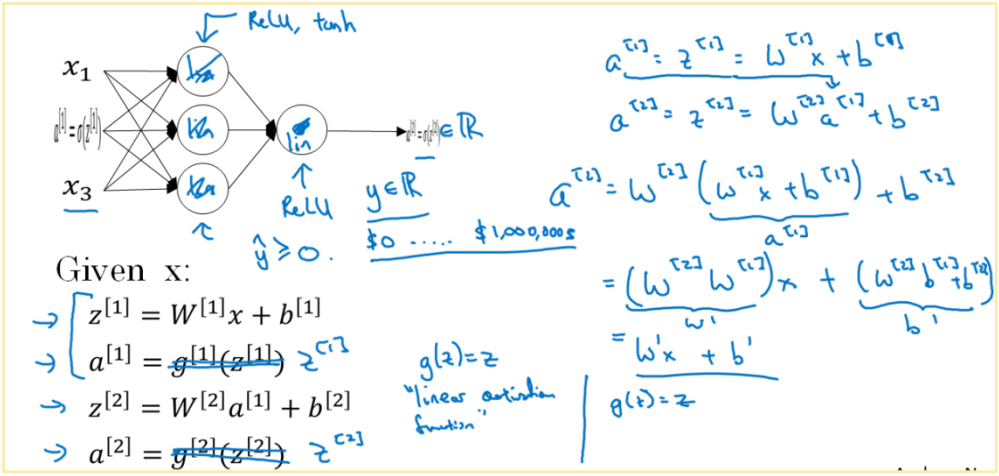
Here we have a recap of the different activation functions:



There is not a specific guideline on which to use. It will depend on the problem.

3.3. Why do we need non-linear activation functions?

We **do** need to pick **non-linear functions** in order to have good results. Why?

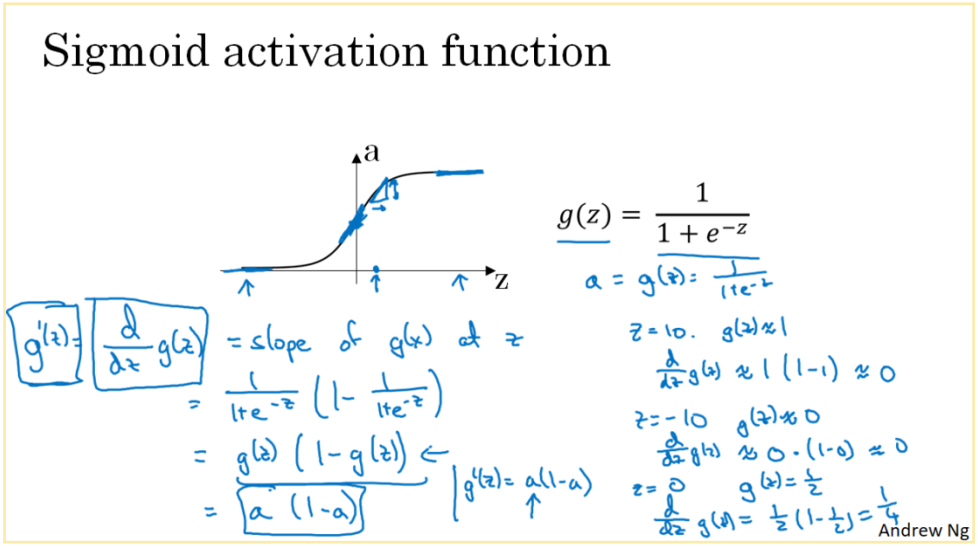


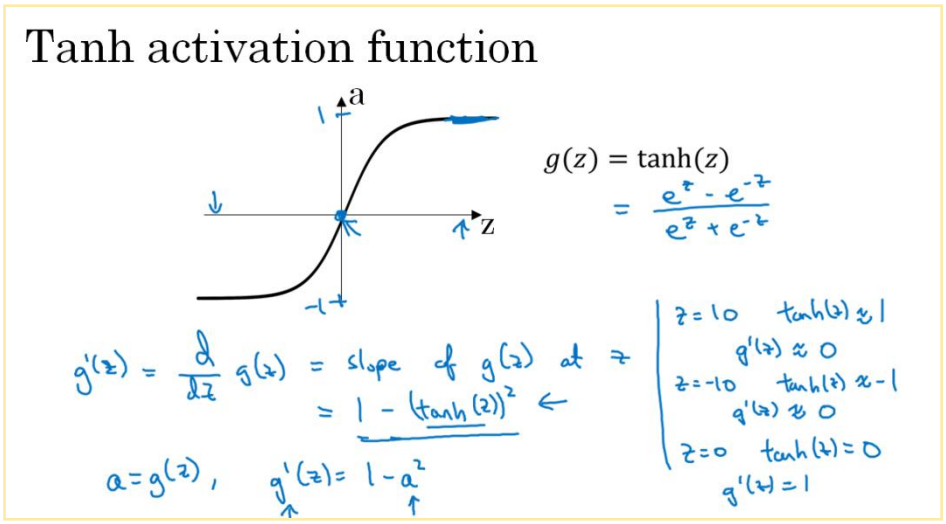
It is simple. If we used linear activation functions, the output would be a linear combination of the input. This means we wouldn't need to use hidden layers. We would have a linear regression model.

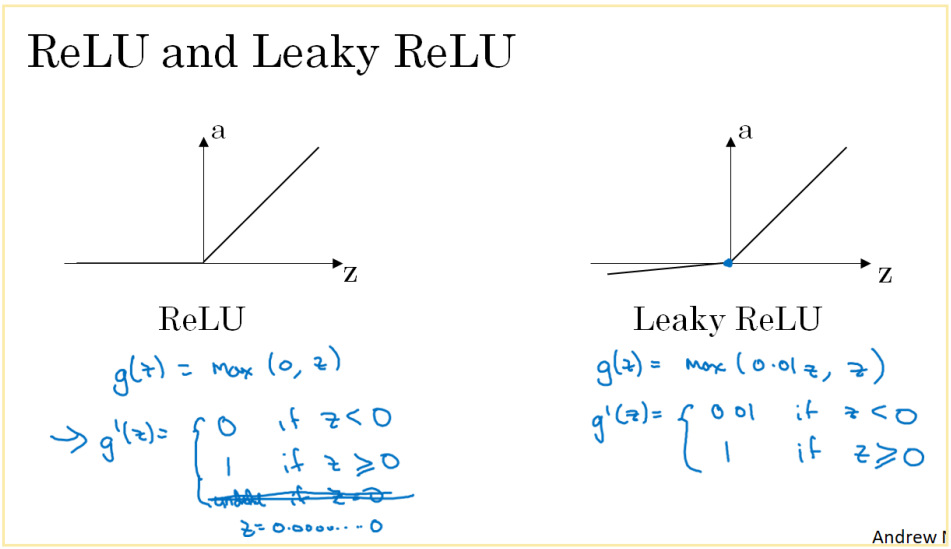
The only case in which we would be interested in using a linear activation function is when we are in a regression problem. There, the output layer should have this function.

3.4. Derivatives of activation functions

When we implement backpropagation we need to compute the derivative of the activation functions. The derivatives are:



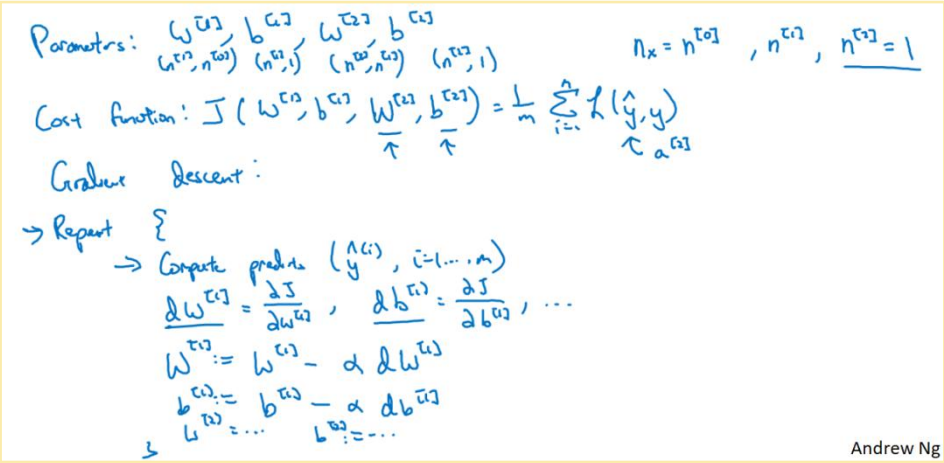




With regard to the corner of the ReLU function, it will be no problem since we would not be able to compute the slope only when $z = 0.0000000000$. In practice, we just give the value 1 to that case.

3.5. Gradient descent for Neural Networks

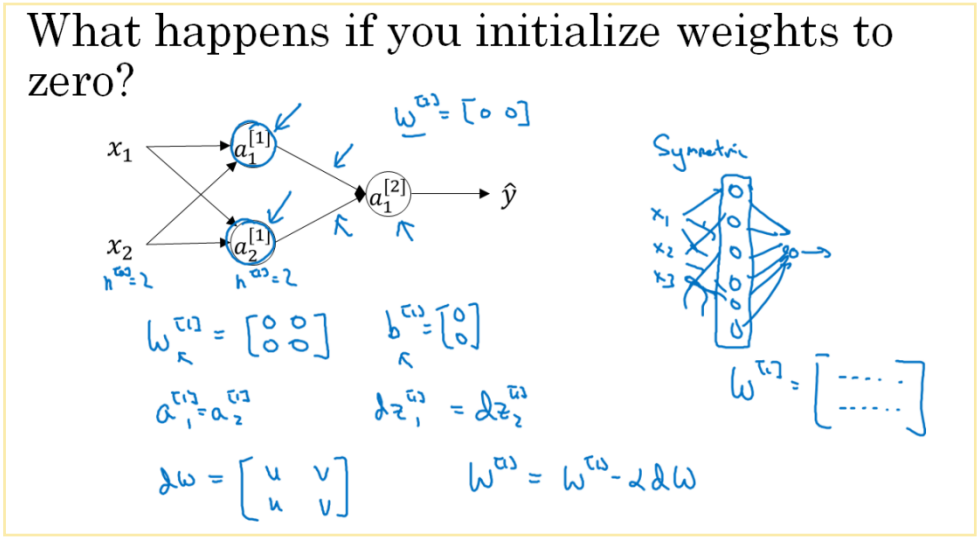
In this section we'll see how to implement gradient descent for neural networks with hidden layers. The process is as follows:



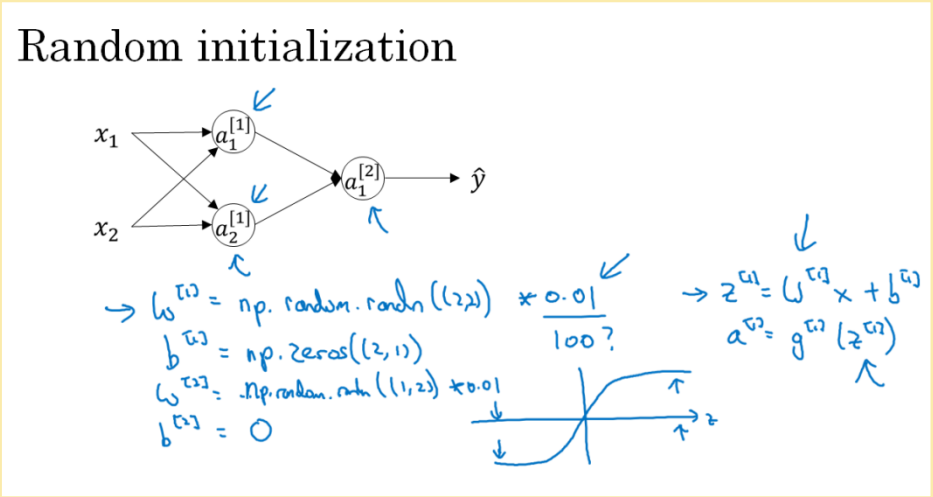
The last thing we need to explain is the initialization of the parameters. It turns out to be a really important task when training our NN. Let's see why:

3.6. Random Initialization

It is important to initialize the weights **randomly**. If we initialize them to 0, after all the iterations, all the hidden units will compute the same function. So it will be like having a single hidden unit.



We'll initialize the weights randomly:

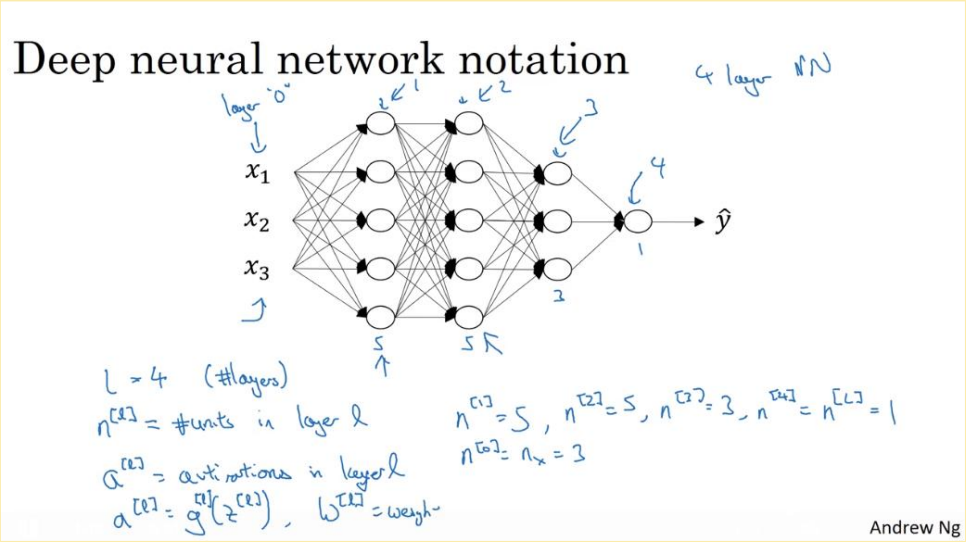


In addition, we would like the values to be small, in order to avoid, for example in the *tanh* function, large values of z leading to small slopes.

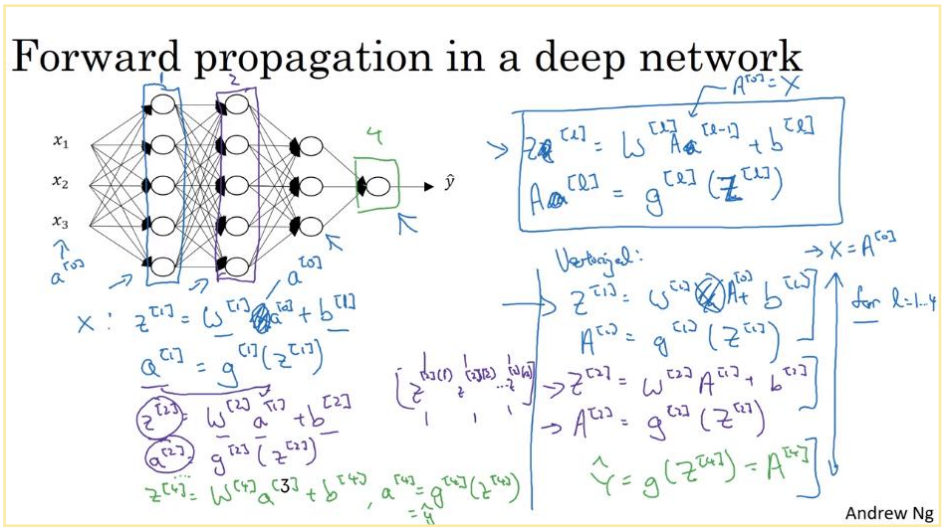
4. Deep Neural Networks

We'll use the knowledge we have acquired for shallow networks (backpropagation and forward propagation, initialization, etc…) to build deep neural networks.

We'll see some notation for a deep neural network:



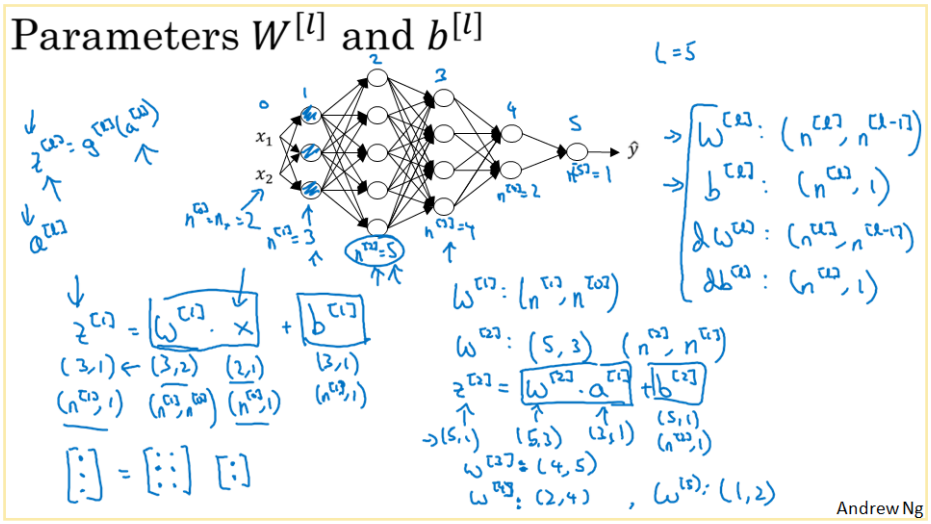
Now, we'll work out how forward propagation is done for a deep NN. It is simple: the output of the layer (activations of the layer) is the input for the next one. In the next figure we have on the left the forward propatgation equations for a simple training example, and on the right we have them for the whole training dataset.



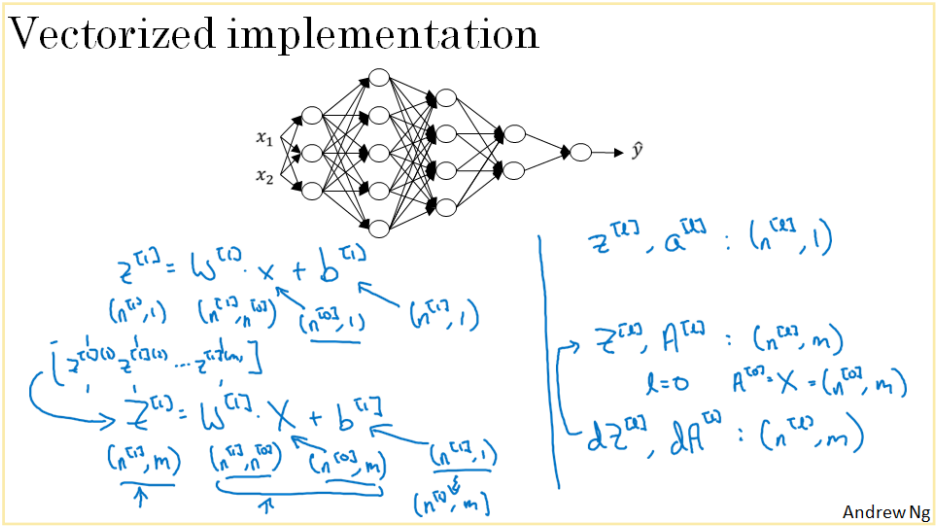
One important thing is that we have to correctly define the dimensions of the matrices involved. We'll see how to do that:

4.1. Getting the dimensions right

Here we show the dimension of the different vectors involved:



When implementing backpropagation, dW and db should have the same dimensions as W and b. Now, if we vectorize:

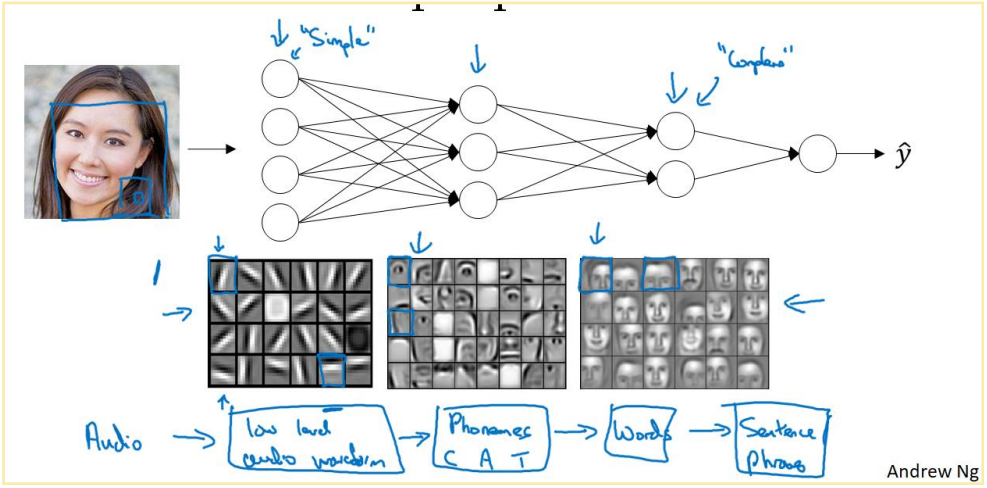


We have figured out the dimensions of Z, A, dZ and dA.

It is a good practice to check the dimensions and it will save us from many bugs.

Once we have seen these details, let's ask a more general question: Why are deep neural networks so effective, and why do they outperform shallow representations?

4.2. Why deep representations?

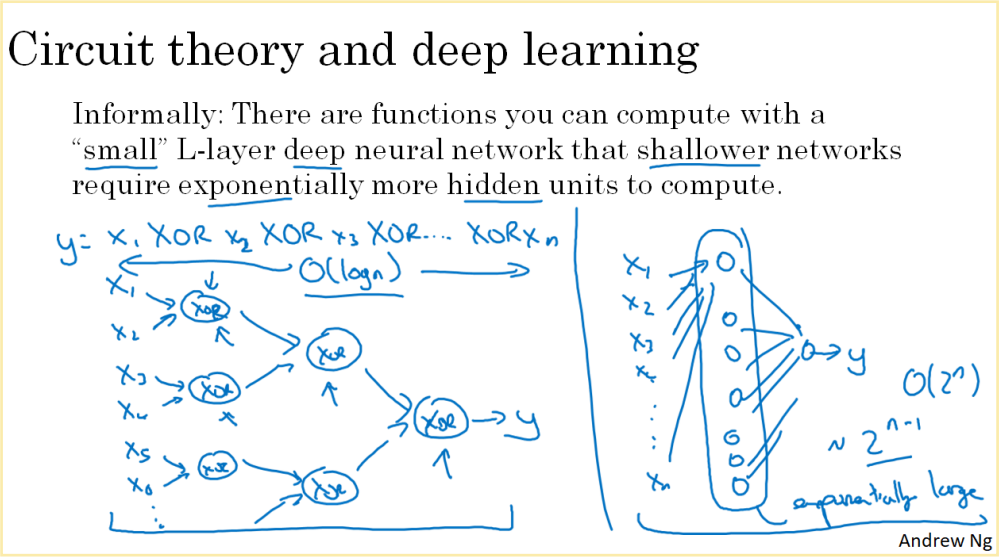


If we have a face detection neural network, we can think of it as the following: the first layer is able to recognize edges. The second layer, by combining edges, can recognize parts of the face. And by combining parts of the face, the last layer can detect faces.

This means, the earlier layers are detecting simple functions like edges. And then, by composing them together in later layers it can learn more complex functions.

And this does not only apply to face recognition. The same simple-to-complex process can be used, for example, in speech recognition.

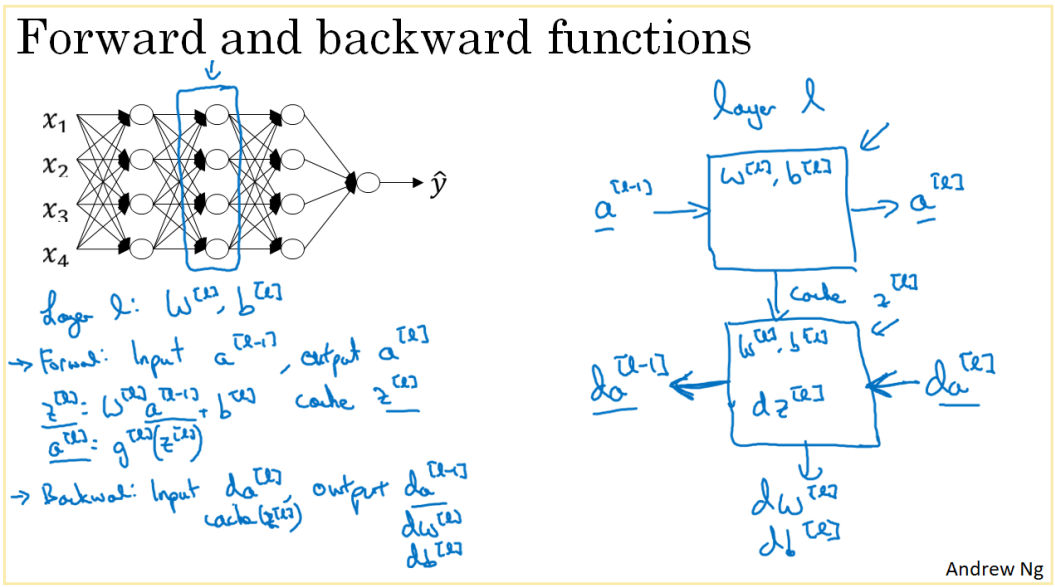
The other piece of intuition that can help us understand the power of deep NN is based on circuit theory:



If we wanted to construct that function y, it would be much more cheaper to do it with a deep small network rather than with a big shallow one. This means we can achieve more complex functions with deep neural networks.

4.3. Building blocks of Deep NN

Until now, we have already seen the basic building blocks of forward propagation and backpropagation, the key components we need to implement a deep NN. Let's see how to put these components together to build a deep net.



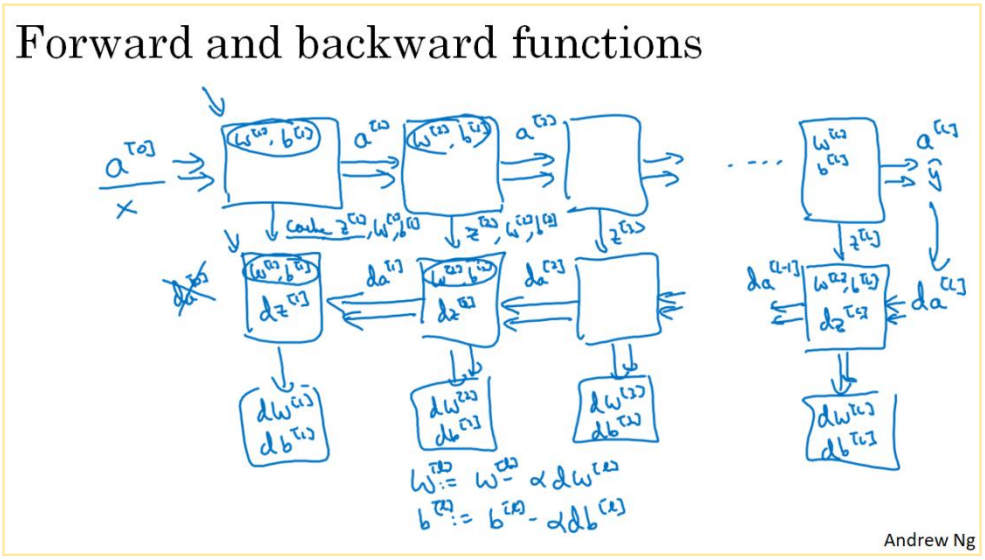
On the left we have, for a given layer l, the computations we need to make to do both forward propagation and backpropagation. In particular, we have named the inputs and outputs of both processes.

On the right we have put these ideas in a diagram shape. For example, to compute forward propagation, we need the activation of layer $l-1$ as an input, we'll use $w^{[l]}$

and $b^{[l]}$ in the calculation and return $a^{[l]}$. We will also output a cache which contains $z^{[l]}$.

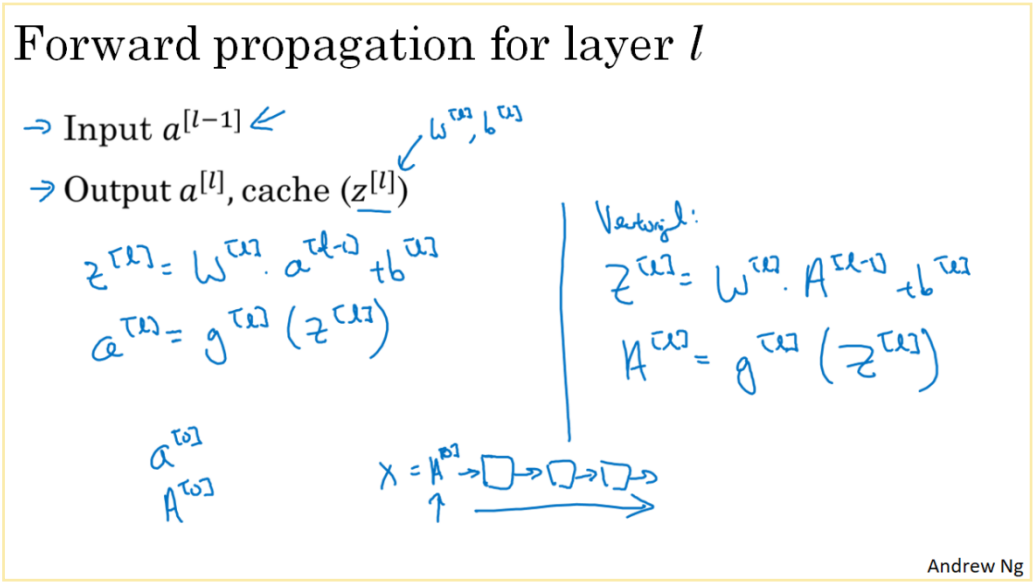
To compute backpropagation, we'll take $da^{[l]}$ as an input and output $w^{[l]}$, $db^{[l]}$ and $da^{[l-1]}$, computing $w^{[l]}$, $b^{[l]}$, $dz^{[l]}$ in the process.

So if we are able to create these functions, the basic coimputation of the neural network will be as follows:

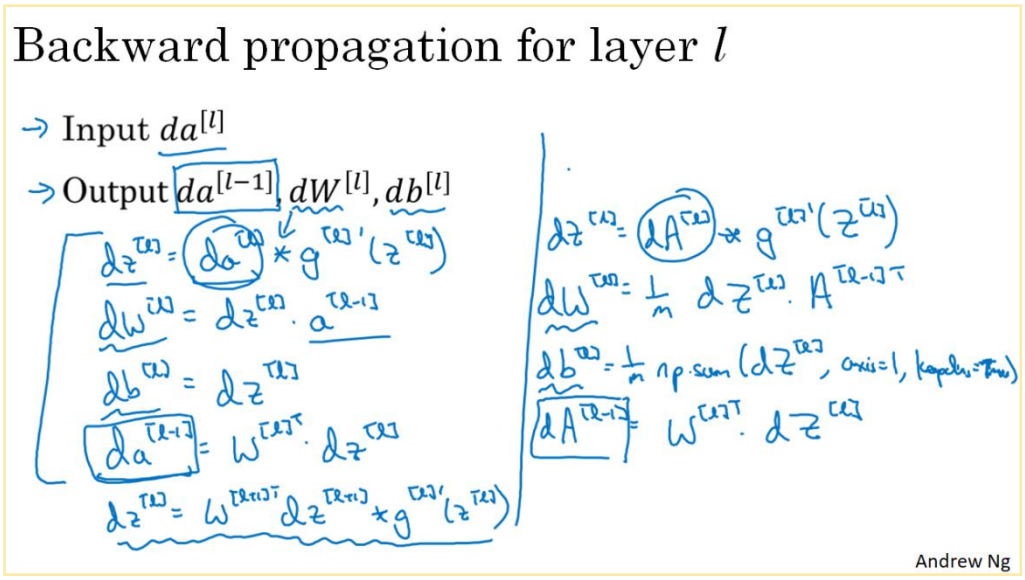


The image represents a single iteration of gradient descent. Along the forward and backpropagation process we compute the necessary things to update the weights.

Let's see how we can **implement** these steps:

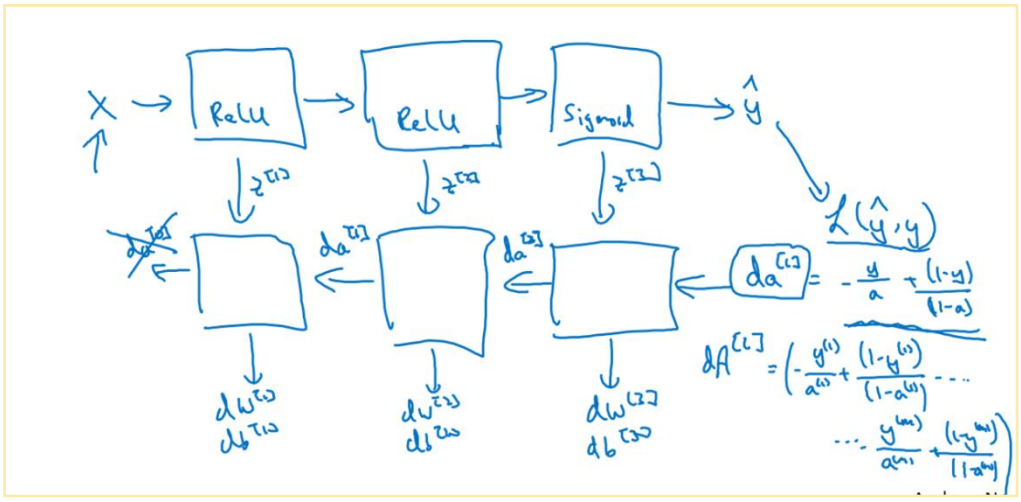


On the left we have the calculation for a given layer l, and on the right we have the vectorized version.



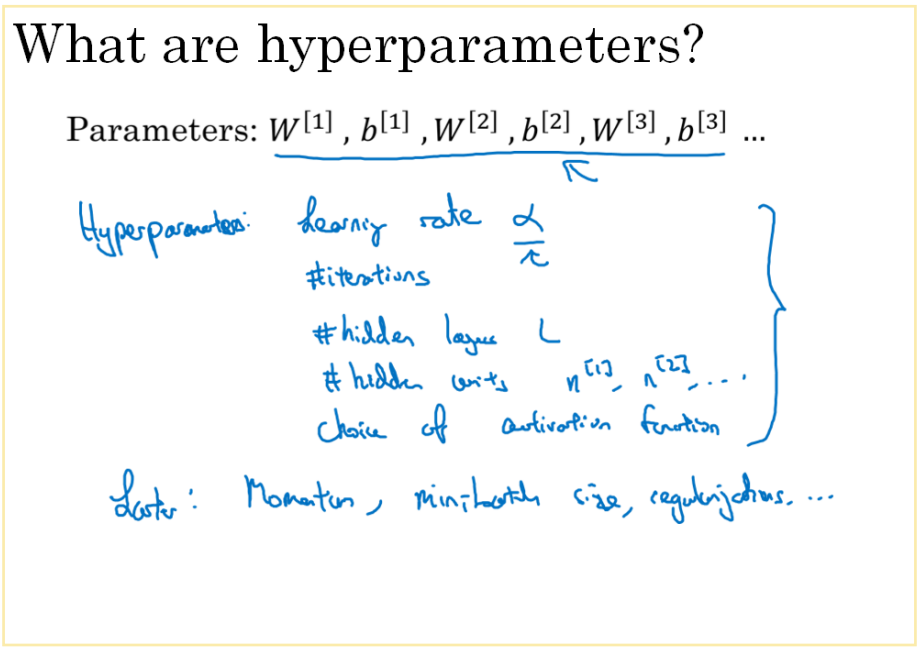
"*" denotes element-wise product.

As a summary:



We feed X into the net, compute the activations on each layer and store the cache, until we get y hat. At that moment the forward propagation process ends. We compute the loss at that iteration and then, start the backpropagation process by calculating dA[L]. We complete it and get the dW and db vectors for each layer, which will allow us to update the weights and start again.

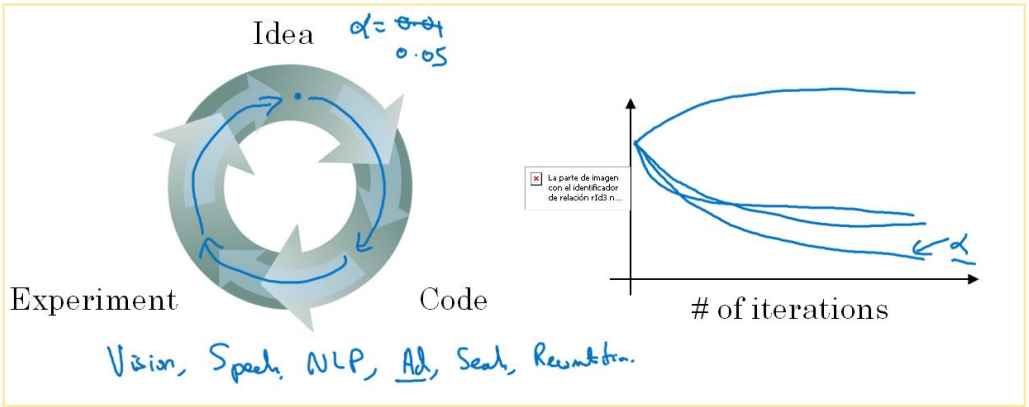
4.4. Parameters and hyperparameters



Parameters are the weights learned in the NN. **Hyperparameters** are the things we need to tell to our learning algorithm, such as the ones listed above. Later in the courses we'll see even more hyperparameters: momentum, mini-batch size, regularization parameters, etc…

Hyperparameters are named so because they control de final values of the model parameters.

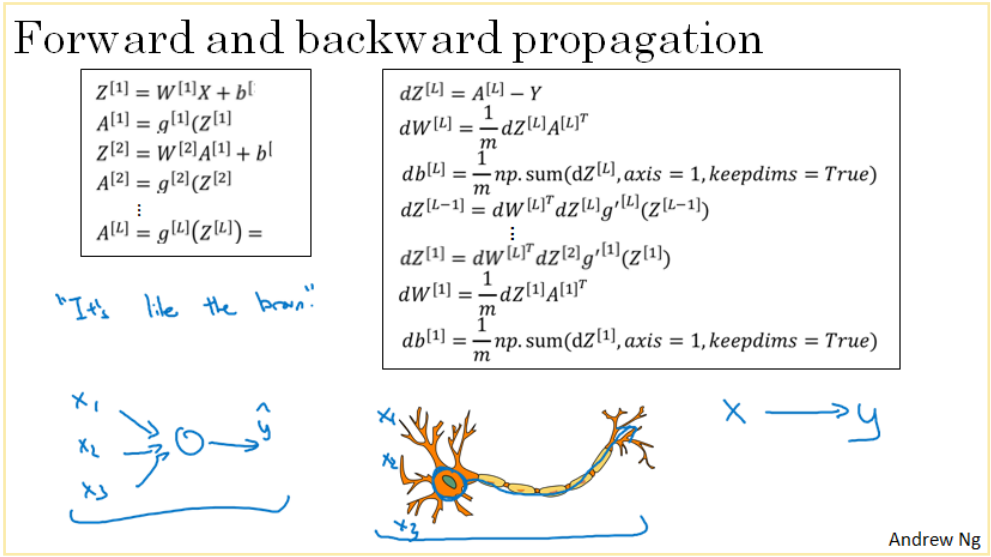
Applying deep learning is a very empirical process:



That is, we'll need to try combinations and see which one performs best.

4.5. What does this have to do with the brain?

Not a whole lot.



Some people establish the analogy that a neuron receives electric signals from other neurons and if the neuron fires, ir sends a pulse of electricity. This may be similar to what a hidden unit does, but be careful. At this day, we don't even really know how biologic neurons work.