Week 1: Introduction

What is a neural network?

Supervised Learning with Neural Networks

In supervised learning, you have some input x and some output y. The goal is to learn a mapping $x \rightarrow y$.

Possibly, the single most lucrative (but not the most inspiring) application of deep learning today is online advertising. Using information about the ad combined with information about the user as input, neural networks have gotten very good at predicting whether or not you click on an ad. Because the ability to show you ads that you're more likely to click on has a *direct impact on the bottom line of some of the very large online advertising companies*.

Here are some more areas in which deep learning has had a huge impact:

- **Computer vision** the recognition and classification of objects in photos and videos.
- **Speech Recognition** converting speech in audio files into transcribed text.
- **Machine translation** translating one natural language to another.
- **Autonomous driving**

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

[https://postimg.cc/image/r2br1relb/]

A lot of the value generation from using neural networks have come from intelligently choosing our x and y and learning a mapping.

We tend to use different architectures for different types of data. For example, **convolutional neural networks** (CNNs) are very common for *image data*, while **recurrent neural networks** (RNNs) are very common for *sequence data* (such as text). Some data, such as radar data from autonomous vehicles, don't neatly fit into any particularly category and so we typical use a complex/hybrid network architecture.

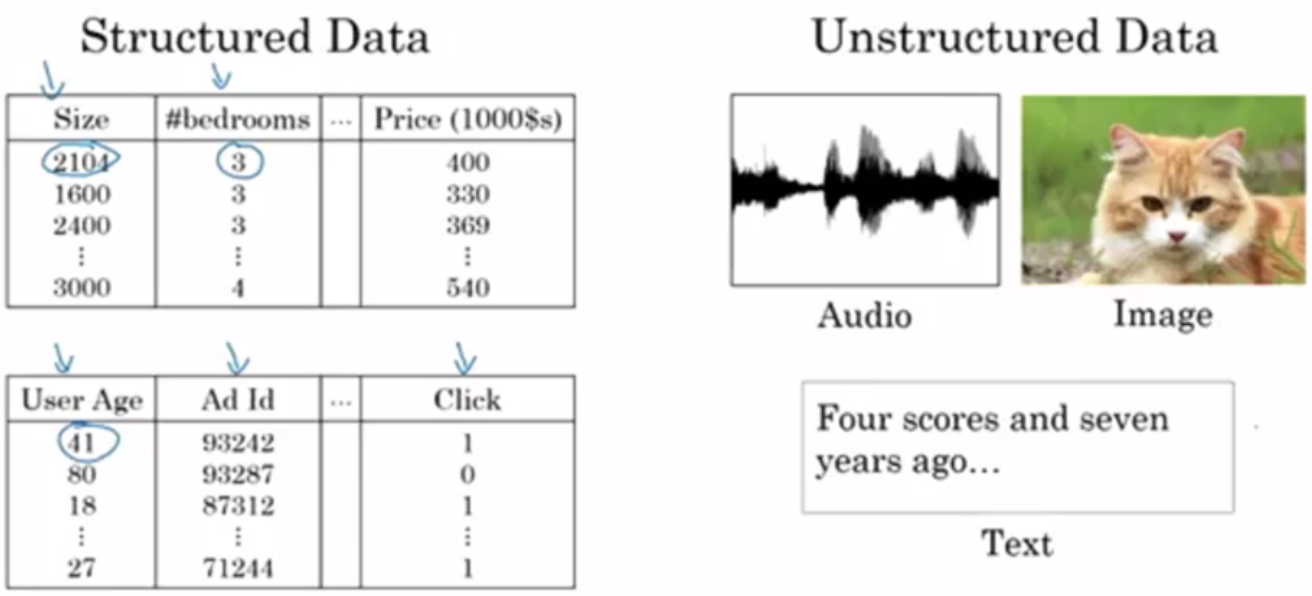
Structured vs. Unstructured Data

You can think of **structured data** as essentially meaning *databases of data*. It is data that is highly *structured*, typically with multiple, well-defined attributes for each piece of data. For example, in housing price prediction, you might have a database where the columns tells you the size and the number of bedrooms. Or in predicting whether or not a user will click on an ad,

you might have information about the user, such as the age, some information about the ad, and then labels why that you're trying to predict.

In contrast, **unstructured data** refers to things like audio, raw audio, or images. Here the features might be the pixel values in an image or the individual words in a piece of text. Historically, it has been much harder for computers to make sense of unstructured data compared to structured data. In contrast, the human race has evolved to be very good at understanding audio cues as well as images. *People are really good at interpreting unstructured data*. And so one of the most exciting things about the rise of neural networks is that, thanks to deep learning, thanks to neural networks, computers are now much better at interpreting unstructured data as compared to just a few years ago. This creates opportunities for many new exciting applications that use speech recognition, image recognition, and natural language processing on text.

Because people have a natural empathy to understanding unstructured data, you might hear about neural network successes on unstructured data more in the media because it's just cool when the neural network recognizes a cat. We all like that, and we all know what that means. But it turns out that a lot of short term economic value that neural networks are creating has also been on structured data, such as much better advertising systems, much better profit recommendations, and just a much better ability to process the giant databases that many companies have to make accurate predictions from them.



[https://postimg.cc/image/5ty2jrutb/]

Why is Deep Learning taking off?

If the basic technical details surrounding deep learning have been around for decades, why are they just taking off now?

First and foremost, the massive amount of (labeled) data we have been generating for the past couple of decades (in part because of the 'digitization' of our society).

It turns out, that large, complex neural networks can take advantage of these huge data stores. Thus, we often say *scale* has been driving progress with deep learning, where scale means the size of the data, the size/complexity of the neural network, and the growth in computation.

The interplay between these 'scales' is apparent when you consider that many of the algorithmic advances of neural networks have come from making them more computational efficient.

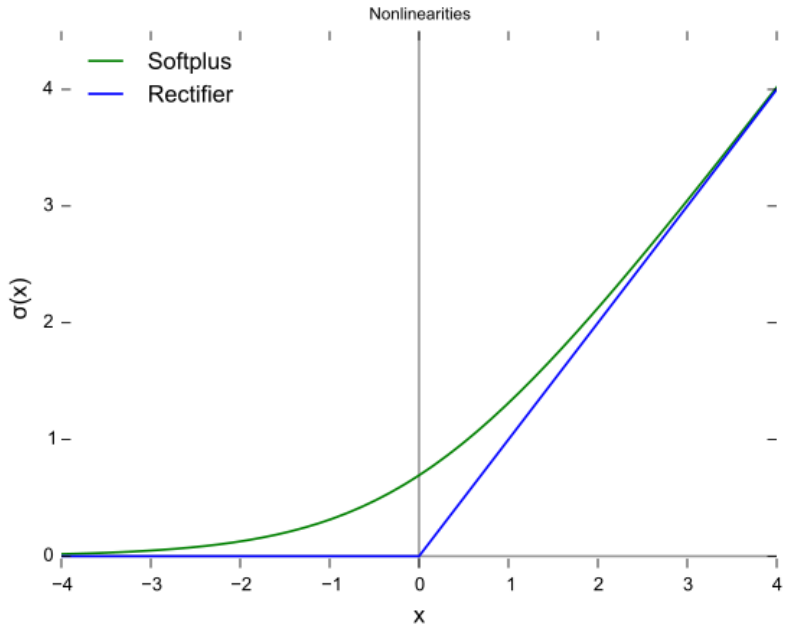
Algorithmic Advances: ReLu

One of the huge breakthroughs in neural networks has been the seemingly simple switch from the **sigmoid** activation function to the **rectified linear** (ReLu) activation function.

One of the problems with using **sigmoid** functions is that its gradients approach 0 as input to the sigmoid function approaches and $+\infty$ and $-\infty$. In this case, the updates to the parameters become very small and our learning slows

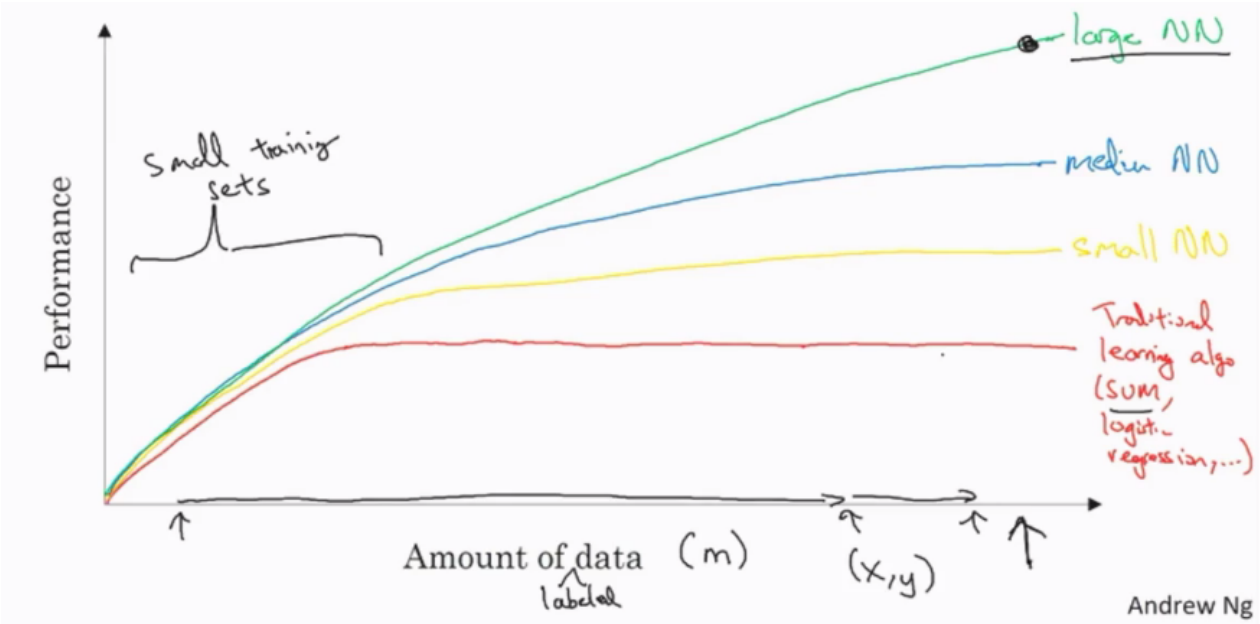
dramatically.

With ReLu units, our gradient is equal to 1 for all positive inputs. This makes learning with gradient descent much faster. See here [https://en.wikipedia.org/wiki/Rectifier_(neural_networks)] for more information on ReLu's.



Scale Advances

With smaller training sets, the relative ordering of the algorithms is actually not very well defined so if you don't have a lot of training data it is often up to your skill at hand engineering features that determines the performance. For small training sets, it's quite possible that if someone training an SVM is more motivated to hand engineer features they will outperform a powerful neural network architecture.



[https://postimg.cc/image/t6w42u0sf/]

However, for very large training sets, *we consistently see large neural networks dominating the other approaches*.

Week 2

Week 2: Neural networks basics

Binary Classification

First, some notation,

- n is the number of data attributes, or *features*
- m is the number of input examples in our dataset (sometimes we write m_{train}, m_{test} to be more explicit).
- our data is represented as input, output pairs, $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$ where $x \in \mathbb{R}^n$, $y \in 0, 1$
- X is our design matrix, which is simply columns of our input vectors $x^{(i)}$, thus it has dimensions of n x m .
- $Y = [y^{(1)}, \ldots, y^{(m)}]$, and is thus a 1 x m matrix.

> Note, this is different from many other courses which represent the design matrix, X as rows of transposed input vectors and the output vector Y as a m x 1 column vector. The above convention turns out to be easier to implement.

When programming neural networks, implementation details become extremely important (*e.g.* vectorization in place of for loops).

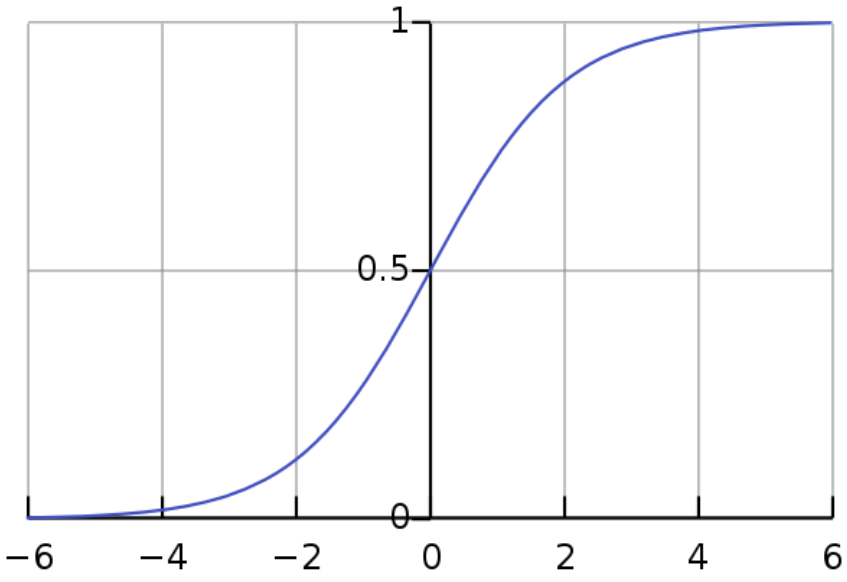
We are going to introduce many of the key concepts of neural networks using **logistic regression**, as this will make them easier to understand. Logistic regression is an algorithm for **binary classification**. In binary classification, we have an input (*e.g.* an image) that we want to classifying as belonging to one of two classes.

Logistic Regression (Crash course)

Given an input feature vector x (perhaps corresponding to an images flattened pixel data), we want $\hat{y}$, the probability of the input examples class, $\hat{y} = P(y = 1|x)$

> If x is a picture, we want the chance that this is a picture of a cat, $\hat{y}$.

The parameters of our model are $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$. Our output is $\hat{y} = \sigma(w^T x + b)$ were σ is the **sigmoid function**.



The formula for the sigmoid function is given by: $\sigma(z) = \frac{1}{1+e^{-z}}$ where $z = w^T x + b$. We notice a few things:

- If z is very large, e^{-z} will be close to 0, and so $\sigma(z)$ is very close to 1.
- If z is very small, e^{-z} will grow very large, and so $\sigma(z)$ is very close to 0.

> It helps to look at the plot $y = e^{-x}$

Thus, logistic regression attempts to learn parameters which will classify images based on their probability of belonging to one class or the other. The classification decision is decided by applying the sigmoid function to $w^T x + b$.

> Note, with neural networks, it is easier to keep the weights w and the biases b separate. Another notation involves adding an extra parameters (w_0 which plays the role of the bias.

Loss function

Our prediction for a given example $x^{(i)}$ is $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$.

We chose **loss function**, $\ell(\hat{y}, y) = -(y\ log\ \hat{y} + (1-y)\ log(1-\hat{y}))$.

We note that:

- If $y = 1$, then the loss function is $\ell(\hat{y}, y) = -log\ \hat{y}$. Thus, the loss approaches zero as $\hat{y}$ approaches 1.
- If $y = 0$, then the loss function is $\ell(\hat{y}, y) = -log\ (1-\hat{y})$. Thus, the loss approaches zero as $\hat{y}$ approaches 0.

> Note, while ℓ_2 loss is taught in many courses and seems like an appropriate choice, it is non-convex and so we cannot use gradient descent to optimize it.
>
> An optional video is given further justifying the use of this loss function. Watch it and add notes here!

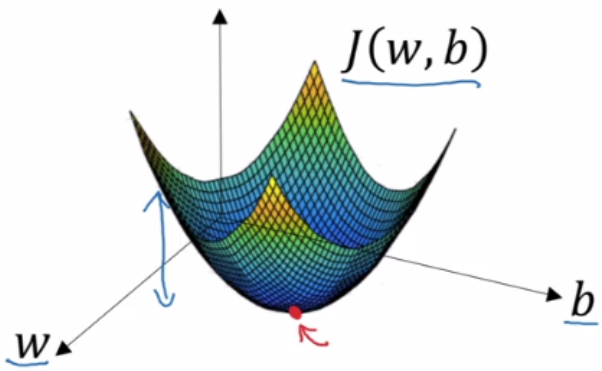
Note that the **loss function** measures how well we are doing on a *single example*. We now define a **cost function**, which captures how well we are doing on the entire dataset:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\ log\ \hat{y}^{(i)} + (1-y^{(i)})\ log(1-\hat{y}^{(i)}))$$

> Note that this notation is somewhat unique, typically the cost/loss functions are just interchangeable terms. However in this course, we will define the **loss function** as computing the error for a single training example and the **cost function** as the average of the loss functions of the entire training set.

Gradient Descent

We want to find w, b which minimize $J(w,b)$. We can plot the **cost function** with w and b as our horizontal axes:



[https://postimg.cc/image/a1suswm2n/]

> In practice, w typically has many more dimensions.

Thus, the cost function $J(w,b)$ can be thought of as a surface, were the height of the surface above the horizontal axes is its value. We want to find the values of our parameters w, b at the lowest point of this surface, the point at which the average loss is at its minimum.

Gradient Descent Algorithm

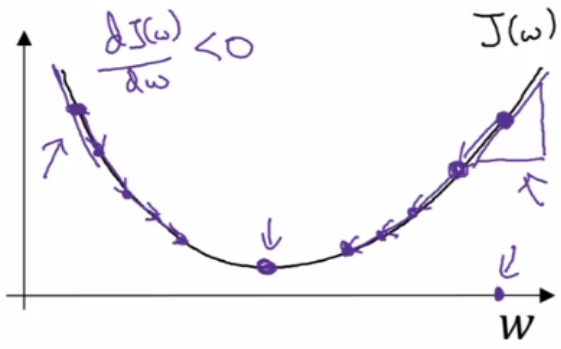
Initialize w, b to some random values

> because this cost function is convex, it doesn't matter what values we use to initialize, 0 is usually chosen for logistic regression.

Repeat

1. $w := w - \alpha \frac{dJ(w)}{dw}$
2. $b := b - \alpha \frac{dJ(w)}{db}$

> α is our learning rate, it controls how big a step we take on each iteration. In some notations, we use ∂ to denote the partial derivative of a function with 2 or more variables, and d to denote the derivative of a function of only 1 variable.



[https://postimg.cc/image/y5jmh99pb/]

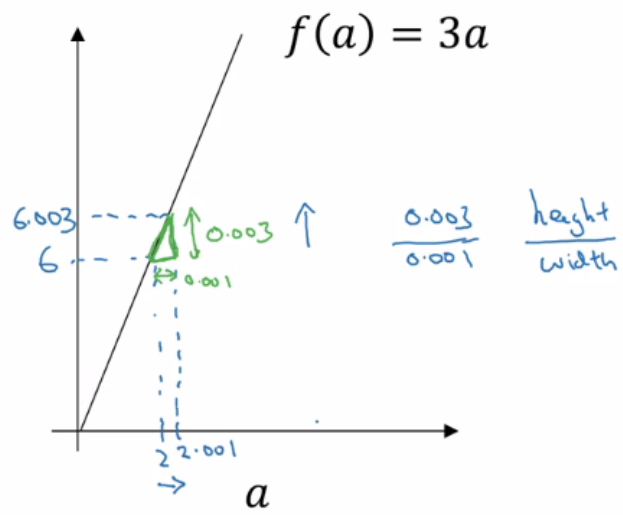
When implementing gradient descent in code, we will use the variable dw to represent $\frac{dJ(w,b)}{dw}$ (this size of the step for w and db to represent $\frac{dJ(w,b)}{db}$ (the size of the step for b .

(ASIDE) Calculus Review

Intuition about derivatives

Linear Function Example

Take the function $f(a) = 3a$. Then $f(a) = 6$ when $a = 2$. If we were to give a a tiny nudge, say to $a = 2.001$, what happens to $f(a)$?



[https://postimg.cc/image/4qdy86sa7/]

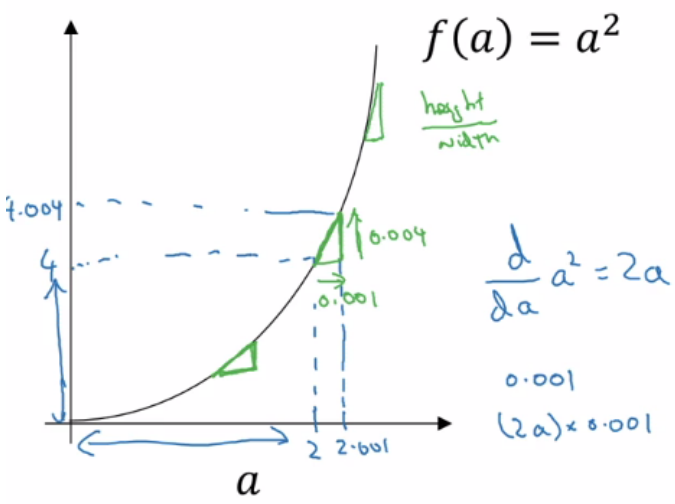
Then $f(a) = 6.003$, but more importantly if we inspect the triangle formed by performing the nudge, we can get the slope of the function between a and $a + 0.001$ as the $\frac{height}{width} = 3$.

Thus, the **derivative** (or slope) of $f(a)$ *w.r.t* a is 3 . We say that $\frac{df(a)}{da} = 3$ or $\frac{d}{da} f(a) = 3$

> Add my calculus notes here! Link to BlueBrown videos.

Non-Linear Function Example

Take the function $f(a) = a^2$. Then $f(a) = 4$ when $a = 2$. If we were to give a a tiny nudge, say to $a = 2.001$, what happens to $f(a)$?



[https://postimg.cc/image/89zvy3pvz/]

Then $f(a) = 4.004$, but more importantly if we inspect the triangle formed by performing the nudge, we can get the slope of the function between a and $a + 0.001$ as the $\frac{height}{width} = 4$.

In a similar way, we can perform this analysis for any point a on the plot, and we will see that slope of $f(a)$ at some point a is equal to $2a$.

Thus, the **derivative** (or slope) of $f(a)$ *w.r.t* a is $2a$. We say that $\frac{df(a)}{da} = 2a$ or $\frac{d}{da} f(a) = 2a$.

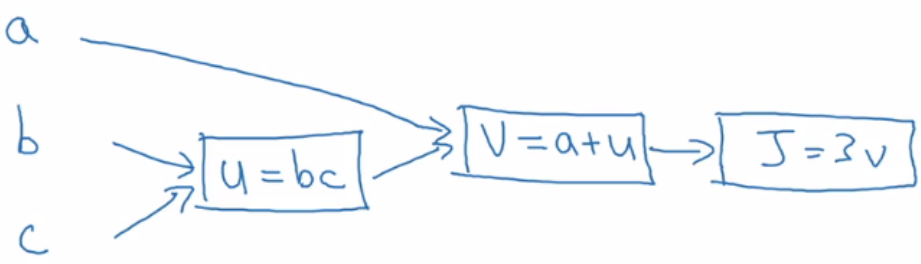
Computation Graph

A **computation graph** organizes a series of computations into left-to-right and right-to-left passes. Lets build the intuition behind a computation graph.

Say we are trying to compute a function $J(a, b, c) = 3(a + bc)$. This computation of this function actually has three discrete steps:

- compute $u = bc$
- compute $v = a + u$
- compute J = $3v$

We can draw this computation in a graph:



[https://postimg.cc/image/r2br1l6tr/]

The computation graph is useful when you have some variable or output variable that you want to optimize (J in this case, in logistic regression it would be our *cost function output*). A *forward pass* through the graph is represented by *left-to-right*

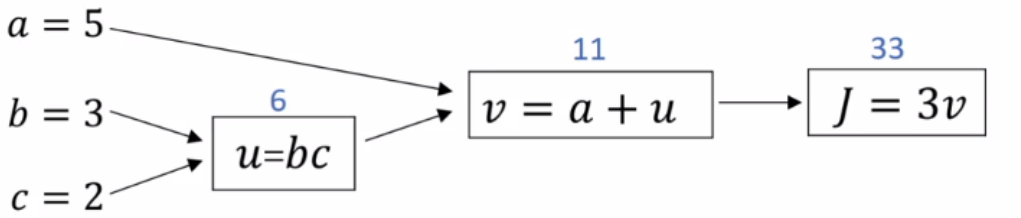
arrows (as drawn above) and a *backwards pass* is represented by *right-to-left* arrows.

A backwards pass is a natural way to represent the computation of our derivatives.

Derivatives with a computation graph

Lets take a look at our computation graph, and see how we can use it to compute the partial derivatives of J i.e., lets carry out backpropogation on this computation graph by hand.

> Informally, you can think of this as asking: "If we were to change the value of v slightly, how would J change?"



[https://postimg.cc/image/iwtp3evfj/]

First, we use our informal way of computing derivatives, and note that a small change to v results in a change to J of 3X that small change, and so $\frac{dJ}{dv} = 3$. This represents one step in our backward pass, the first step in backpropagation.

Now let's look at another example. What is $\frac{dJ}{da}$?

We compute the $\frac{dJ}{da}$ from the second node in the computation graph by noting that a small change to a results in a change to J of 3X that small change, and so $\frac{dJ}{da} = 3$. This represents our second step in our backpropagation.

One way to break this down is to say that by changing a, we change v, the magnitude of this change is $\frac{dv}{da}$. Through this change in v, we also change J, and the magnitude of the change is $\frac{dJ}{dv}$. To capture this more generally, we use the **chain rule** from calculus, informally:

$$\text{if } a \to v \to J, \text{ then } \frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da}$$

> Here, just take $\to$ to mean 'effects'. A formal definition of the chain rule can be found here [https://en.wikipedia.org/wiki/Chain_rule].

The amount J changes when you when you nudge a is the product of the amount J changes when you nudge v multiplied by the amount v changes when you nudge a .

> **Implementation note**: When writing code to implement backpropagation, there is typically a single output variable you want to optimize, $dvar$, (the value of the cost function). We will follow to notation of calling this variable $dvar$.

If we continue performing backpropagation steps, we can determine the individual contribution a change to the input variables has on the output variable. For example,

$$\frac{dJ}{db} = \frac{dJ}{du}\frac{du}{db} = (3)(2) = 6$$

The key take away from this video is that when computing derivatives to determine the contribution of input variables to change in an output variable, the most efficient way to do so is through a right to left pass through a computation graph. In particular, we'll first compute the derivative with respect to the output of the left-most node in a backward pass, which becomes useful for computing the derivative with respect to the next node and so forth. The **chain rule** makes the computation of these derivatives tractable.

Logistic Regression Gradient Descent

Logistic regression recap:

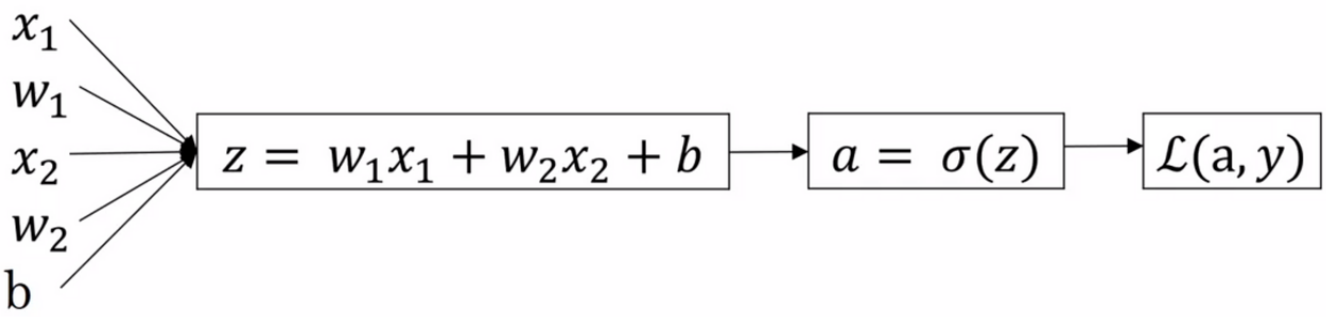
$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$\ell(a, y) = -(ylog(a) + (1 - y)log(1 - a))$$

> ℓ is our loss for a single example, and $\hat{y}$ are our predictions.

For this example, lets assume we have only two features: x_1, x_2 . Our computation graph is thus:



[https://postimg.cc/image/le5gaphwv/]

Our goal is to modify the parameters to minimize the loss ℓ . This translates to computing derivatives $w.r.t$ the loss function. Following our generic example above, we can compute all the relevant derivatives using the chain rule. The first two passes are computed by the following derivatives:

1. $\frac{d\ell(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$
2. $\frac{d\ell(a,y)}{dz} = \frac{d\ell(a,y)}{da} \cdot \frac{da}{dz} = a - y$

> Note: You should prove these to yourself.
>
> **Implementation note**, we use dx as a shorthand for $\frac{d\ell(\hat{y},y)}{dx}$ for some variable x when implementing this in code.

Recall that the final step is to determine the derivatives of the loss function $w.r.t$ to the parameters.

- $\frac{d\ell(a,y)}{dw_1} = x_1 \cdot \frac{d\ell(a,y)}{dz}$
- $\frac{d\ell(a,y)}{dw_2} = x_2 \cdot \frac{d\ell(a,y)}{dz}$

One step of gradient descent would perform the updates:

- $w_1 := w_1 - \alpha \frac{d\ell(a,y)}{dw_1}$
- $w_2 := w_2 - \alpha \frac{d\ell(a,y)}{dw_2}$
- $b := b - \alpha \frac{d\ell(a,y)}{db}$

Extending to m examples

Lets first remind ourself of the logistic regression **cost** function:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)}\ log\ \hat{y}^{(i)} + (1 - y^{(i)})\ log(1 - \hat{y}^{(i)}))$$

Where,

$$\hat{y} = a = \sigma(z) = \sigma(w^T x^{(i)} + b)$$

In the example above for a single training example, we showed that to perform a gradient step we first need to compute the derivatives $\frac{d\ell(a,y)}{dw_1}$, $\frac{d\ell(a,y)}{dw_2}$, $\frac{d\ell(a,y)}{db}$. For m examples, these are computed as follows:

- $\frac{\partial \ell(a,y)}{\partial dw_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1}\ell(\hat{y}^{(i)}, y^{(i)})$
- $\frac{\partial \ell(a,y)}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_2}\ell(\hat{y}^{(i)}, y^{(i)})$
- $\frac{\partial \ell(a,y)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial b}\ell(\hat{y}^{(i)}, y^{(i)})$

We have already shown on the previous slide how to compute $\frac{\partial}{\partial w_1}\ell(\hat{y}^{(i)}, y^{(i)})$, $\frac{\partial}{\partial w_2}\ell(\hat{y}^{(i)}, y^{(i)})$ and $\frac{\partial}{\partial b}\ell(\hat{y}^{(i)}, y^{(i)})$. Gradient descent for m examples essentially involves computing these derivatives for each input example $x^{(i)}$ and averaging the result before performing the gradient step. Concretely, the pseudo-code for gradient descent on m examples of $n = 2$ features follows:

ALGO

Initialize $J = 0; dw_1 = 0; dw_2 = 0; db = 0$

for $i = 1$ to m:

- $z^{(i)} = w^T x^{(i)}$
- $a^{(i)} = \sigma(z^{(i)})$
- $J += -[y^{(i)} log(a^{(i)}) + (1 - y^{(i)}) log(1 - a^{(i)})]$
- $dz^{(i)} = a^{(i)} - y^{(i)}$
- for $j = 1$ to n
- $dw_j += x_j^{(i)} dz^{(i)}$
- $dw_j += x_j^{(i)} dz^{(i)}$
- $db += dz^{(i)}$

$J /= m;\ dw_1 /= m;\ dw_2 /= m;\ db /= m$

In plain english, for each training example, we use the sigmoid function to compute its activation, accumulate a loss for that example based on the current parameters, compute the derivative of the current cost function $w.r.t$ the activation function, and update our parameters and bias. Finally we take the average of our cost function and our gradients.

Finally, we use our derivatives to update our parameters,

- $w_1 := w_1 - \alpha \cdot dw_1$
- $w_2 := w_2 - \alpha \cdot dw_2$
- $b := b - \alpha \cdot db$

This constitutes **one step** of gradient descent.

The main problem with this implementation is the nested for loops. For deep learning, which requires very large training sets, *explicit for loops* will make our implementation very slow. Vectorizing this algorithm will greatly speed up our algorithms running time.

Vectorization

Vectorization is basically the art of getting rid of explicit for loops. In practice, deep learning requires large datasets (at least to obtain high performance). Explicit for loops lead to computational overhead that significantly slows down the training process.

The main reason vectorization makes such a dramatic difference is that it allows us to take advantage of **parallelization**. The rule of thumb to remember is: *whenever possible, avoid explicit for-loops*.

> In a toy example were n_x is 10^6, and $w, x^{(i)}$ are random values, vectorization leads to an approximately 300X speed up to compute all $z^{(i)}$

Lets take a look at some explicit examples:

- Multiply a **matrix** by a **vector**, e.g., $u = Av$.

So, $u_i = \sum_j A_{ij} v_j$. Instead of using for nested loops, use: `u = np.dot(A,v)`

- Apply exponential operation on every element of a matrix/vector v.

Again, use libraries such as `numpy` to perform this with a single operation, e.g., `u = np.exp(v)`

> This example applies to almost all operations, `np.log(v)` , `np.abs(v)` , `np.max(v)` , etc...

Example: Vectorization of Logistic Regression

Forward pass

Lets first review the forward pass of logistic regression for m examples:

$$z^{(1)} = w^T x^{(1)} + b; a^{(1)} = \sigma(z^1), \ldots, z^{(m)} = w^T x^{(m)} + b; a^{(m)} = \sigma(z^m)$$

In logistic regression, we need to compute $z^{(i)} = w^T x^{(i)} + b$ for each input example $x^{(i)}$. Instead of using a for loop over each i in range (m) we can use a vectorized implementation to compute z directly.

Our vectors are of the dimensions: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}^{n_x}$, $x \in \mathbb{R}^{n_x}$.

Our parameter vector, bias vector, and design matrix are,

$$w = \begin{bmatrix} w_1 \\ \ldots \\ w_{n_x} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \ldots \\ b_{n_x} \end{bmatrix}, X = \begin{bmatrix} x_1^{(1)} & \ldots & x_1^{(m)} \\ \ldots & & \\ x_{n_x}^{(1)} & & \end{bmatrix}$$

So, $w^T \cdot X + b = w^T x^{(i)} + b$ (for all i). Thus we can compute all $w^T x^{(i)}$ in one operation if we vectorize!

In numpy code:

```
Z = np.dot(w.T, X) + b
```

> Note, $+b$ will perform element-wise addition in python, and is an example of **broadcasting**.

Where Z is a row vector $[z^{(1)}, \ldots, z^{(m)}]$.

Backward pass

Recall, for the gradient computation, we computed the following derivatives:

$$dz^{(1)} = a^{(1)} - y^{(1)} \ldots dz^{(m)} = a^{(m)} - y^{(m)}$$

We define a row vector,

$$dZ = [dz^{(1)}, \ldots, dz^{(m)}] .$$

From which it is trivial to see that,

$dZ = A - Y$, where $A = [a^{(1)}, \ldots, a^{(m)}]$ and $Y = [y^{(1)}, \ldots, y^{(m)}]$.

> This is an element-wise subtraction, $a^{(1)} - y^{(1)}, \ldots, a^{(m)} - y^{(m)}$ that produces a m length row vector.

We can then compute our *average* derivatives of the cost function $w.r.t$ to the parameters in two lines of codes,

```
db = 1/m * np.sum(dZ)
```

```
dw = 1/m * np.dot(X, dZ.T)
```

Finally, we compare our non-vectorized approach to linear regression vs our vectorized approaches

Non-vectorized Approach	Vectorized Approach
$J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$ for i = 1 to m: $z^{(i)} = w^T x^{(i)} + b$ $a^{(i)} = \sigma(z^{(i)})$ $J\ +\!= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$ $dz^{(i)} = a^{(i)} - y^{(i)}$ $dw_1\ +\!= x_1^{(i)} dz^{(i)}$ $dw_2\ +\!= x_2^{(i)} dz^{(i)}$ $db\ +\!= dz^{(i)}$ $J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m$ $db = db/m$ [https://postimg.cc/image/shdbqdksf/]	for iter in range(1000): $Z = w^T X + b$ $= np.dot(w.T, X) + b$ $A = \sigma(Z)$ $dZ = A - Y$ $dw = \frac{1}{m} X dZ^T$ $db = \frac{1}{m} np.sum(dZ)$ $w := w - \alpha dw$ $b := b - \alpha db$ [https://postimg.cc/image/u96al9wfj/]
Two for loops, one over the training examples $x^{(i)}$ and a second over the features $x_j^{(i)}$. We have omitted the outermost loop that iterates over gradient steps.	Note that, we still need a single for loop to iterate over each gradient step (regardless if we are using stochastic or mini-batch gradient descent) even in our vectorized approach.

Broadcasting

Lets motivate the usefulness of **broadcasting** with an example. Lets say you wanted to get the percent of total calories from carbs, proteins, and fats for multiple foods.

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

[https://postimg.cc/image/le5gapa73/]

Can we do this without an explicit for loop?

Set this matrix to a `(3,4)` numpy matrix `A`.

```
import numy as np

# some numpy array of shape (3,4)
A = np.array([
  [...],
  [...],
  [...]
  ])
```

```
cal = A.sum(axis=0) # get column-wise sums
percentage = 100 * A / cal.reshape(1,4) # get percentage of total calories
```

So, we took a `(3,4)` matrix `A` and divided it by a `(1,4)` matrix `cal`. This is an example of broadcasting.

The general principle of broadcast can be summed up as follows:

- $(m, n)\ [+ \text{ OR - OR * OR } /]\ (1, n) \Rightarrow (m, n)\ [+ \text{ OR - OR * OR } /]\ (m \text{ copies}, n)$
- $(m, n)\ [+ \text{ OR - OR * OR } /]\ (m, 1) \Rightarrow (m, n)\ [+ \text{ OR - OR * OR } /]\ (m, n \text{ copies})$

Where $(m, n), (1, n)$ are matrices, and the operations are performed *element-wise* after broadcasting.

More broadcasting examples

Addition

Example 1: $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 == \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$

Example 2: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} == \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$

Example 3: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} == \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 206 \end{bmatrix}$

(AISDE) A note on python/numpy vectors

The great flexibility of the python language paired with the numpy library is both a strength and a weakness. It is a strength because of the great expressivity of the pair, but with this comes the opportunity to intro strange, hard-to-catch bugs if you aren't familiar with the intricacies of numpy and in particular broadcasting.

Here are a couple of tips and tricks to minimize the number of these bugs:

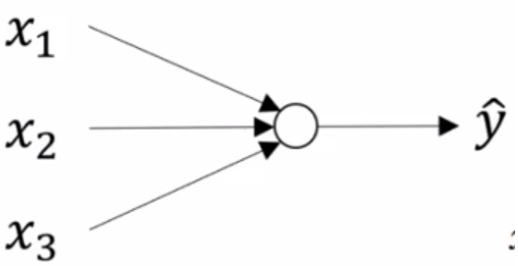
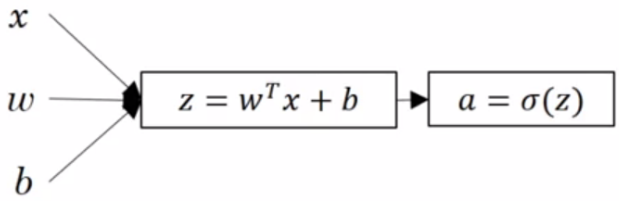
- Creating a random array: `a = np.random.randn(5)`
- Arrays of shape `(x, )` are known as **rank 1 array**. They have some nonintuitive properties and don't consistently behave like either a column vector or a row vector. Let `b` be a rank 1 array.
- `b.T == b`
- `np.dot(b, b.T)` is a real number, *not the outer product as you might expect*.
- Thus, in this class at least, using rank 1 tensors with an unspecified dimension length is not generally advised. *Always specify both dimensions*.
- If you know the size that your numpy arrays should be in advance, its often useful to throw in a python assertion to help catch strange bugs before they happen:
- `assert(a.shape == (5,1))`
- Additionally, the reshape function runs in linear time and is thus very cheap to call, use it freely!
- `a = a.reshape((5,1))`

Week 3

Week 3: Shallow neural networks

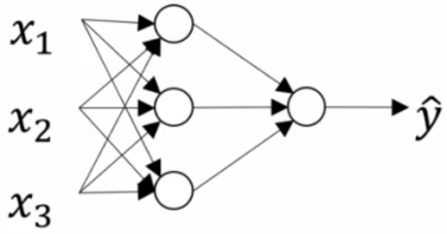
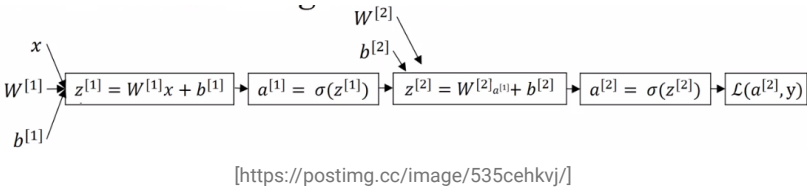
Neural network overview

Up until this point, we have used logistic regression as a stand-in for neural networks. The "network" we have been describing looked like:

Network	Computation Graph
	
[https://postimg.cc/image/jmchfu8un/]	[https://postimg.cc/image/cj4m09dpr/]

> a and $\hat{y}$ are used interchangeably

A neural network looks something like this:

Network	Computation Graph
	
[https://postimg.cc/image/c6d7u4dgf/]	[https://postimg.cc/image/535cehkvj/]

> We typically don't distinguish between z and a when talking about neural networks, one neuron = one activation = one a like calculation.

We will introduce the notation of superscripting values with $^{[l]}$, where l refers to the layer of the neural network that we are talking about.

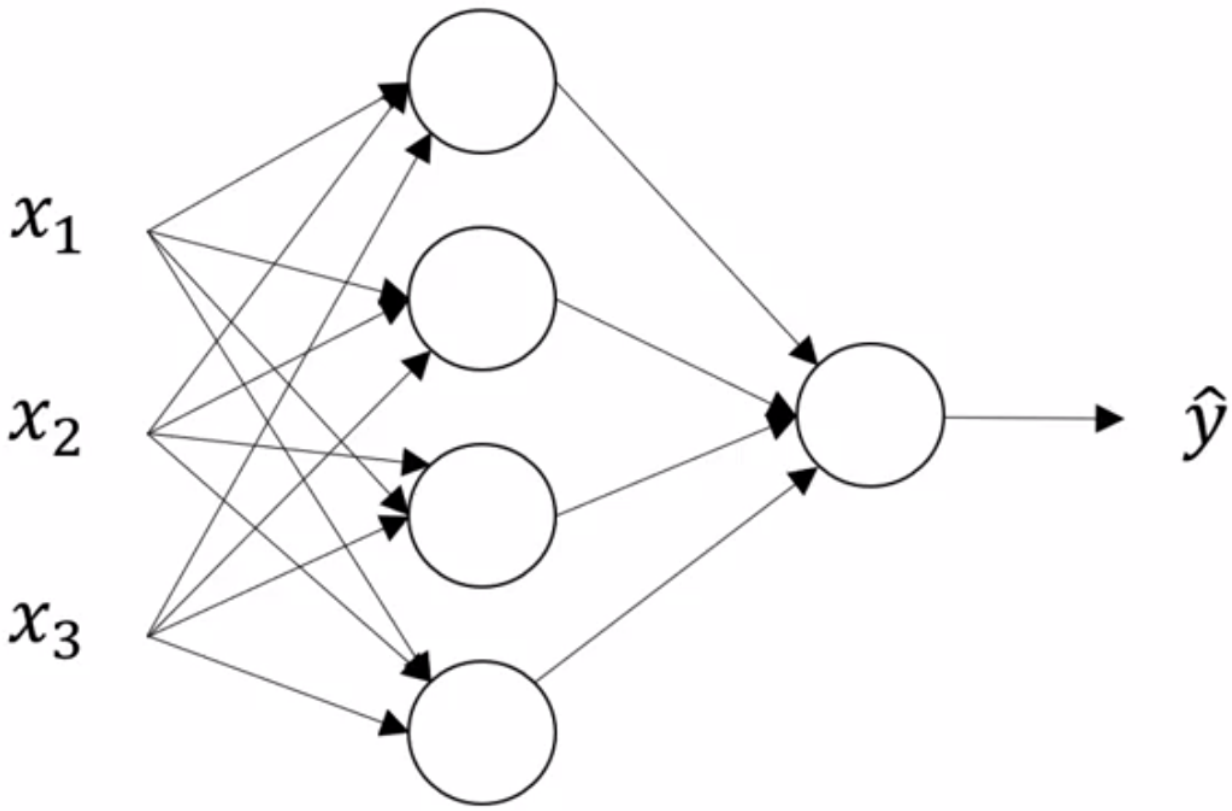
> Not to be confused with $^{(i)}$ which we use to refer to a single input example i .

The key intuition is that neural networks stack activations of inputs multiplied by their weights.

Similar to the 'backwards' step that we discussed for logistic regression, we will explore the backwards steps that makes learning in a neural network possible.

Neural network Representation

This is the canonical representation of a neural network



[https://postimg.cc/image/4qdy8babj/]

On the left, we have the **input features** stacked vertically. This constitutes our **input layer**. The final layer, is called the **output layer** and it is responsible for generating the predicted value $\hat{y}$. Any layer in between these two layers is known as a **hidden layer**. This name derives from the fact that the *true values* of these hidden units is not observed in the training set.

> The hidden layers and output layers have parameters associated with them. These parameters are denoted $W^{[l]}$ and $b^{[l]}$ for layer l .

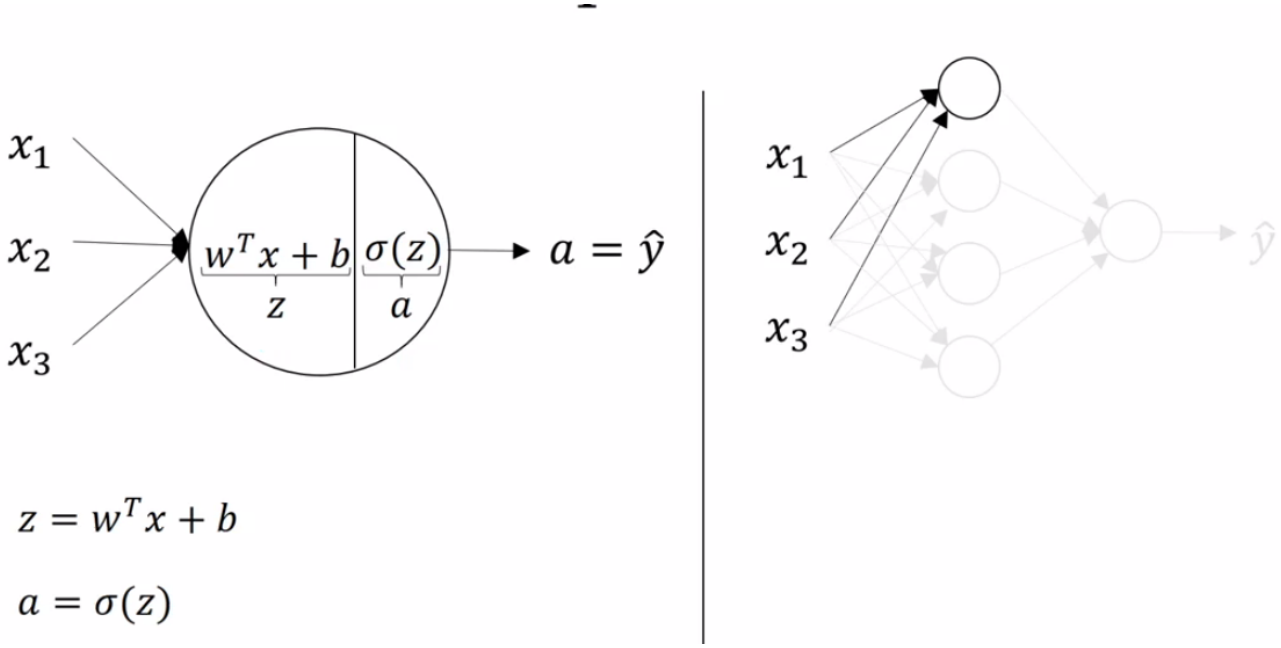
Previously, we were referring to our input examples as $x^{(i)}$ and organizing them in a design matrix X . With neural networks, we will introduce the convention of denoting output values of a layer l, as a column vector $a^{[l]}$, where a stands for *activation*. You can also think of these as the values a layer l passes on to the next layer.

Another note: the network shown above is a *2-layer* neural network. We typically do not count the input layer. In light of this, we usually denote the input layer as $l = 0$.

Computing a Neural Networks Output

We will use the example of a single hidden layer neural network to demonstrate the forward propagation of inputs through the network leading to the networks output.

We can think of each unit in the neural network as performing two steps, the *multiplication of inputs by weights and the addition of a bias*, and the *activation of the resulting value*



[https://postimg.cc/image/i8kuk2z67/]

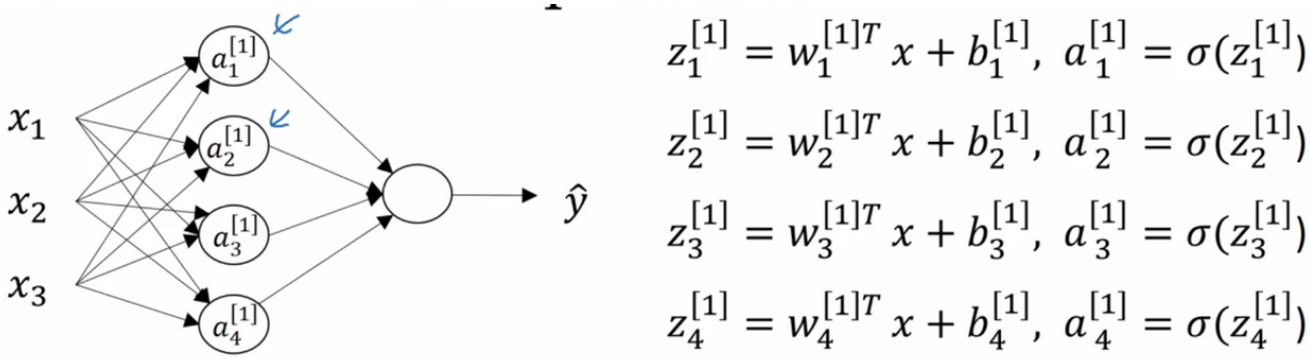
> Recall, that we will use a superscript, $^{[l]}$ to denote values belonging to the $l-th$ layer.

So, the j^{th} node of the l^{th} layer performs the computation

$$a_j^{[l]} = \sigma(w_j^{[l]^T} a^{[l-1]} + b_j^{[l]})$$

> Where $a^{[l-1]}$ is the activation values from the precious layer.

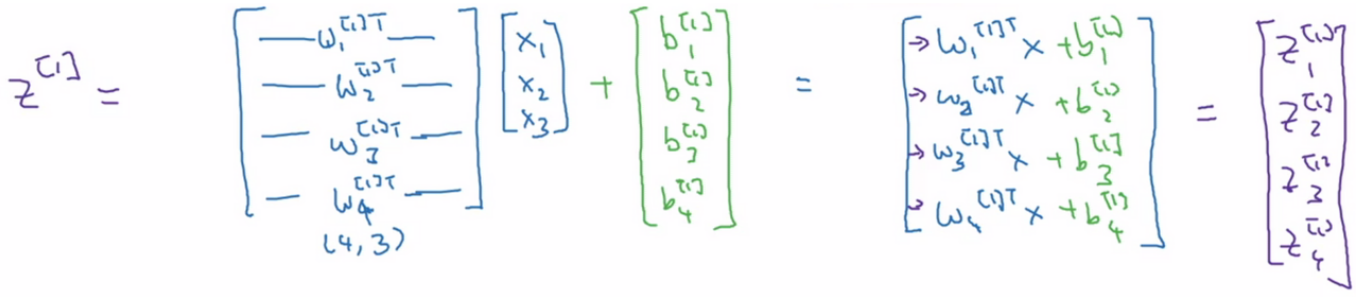
for some input x. With this notation, we can draw our neural network as follows:



[https://postimg.cc/image/f0gd7lxn3/]

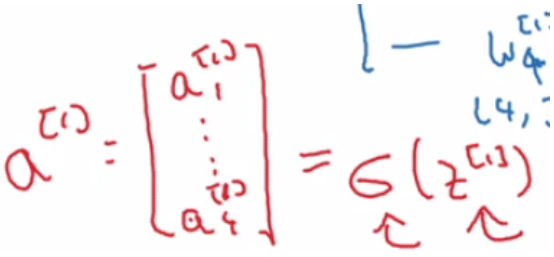
In order to easily vectorize the computations we need to perform, we designate a matrix $W^{[l]}$ for each layer l, which has dimensions *(number of units in current layer X number of units in previous layer)*

We can vectorize the computation of $z^{[l]}$ as follows:



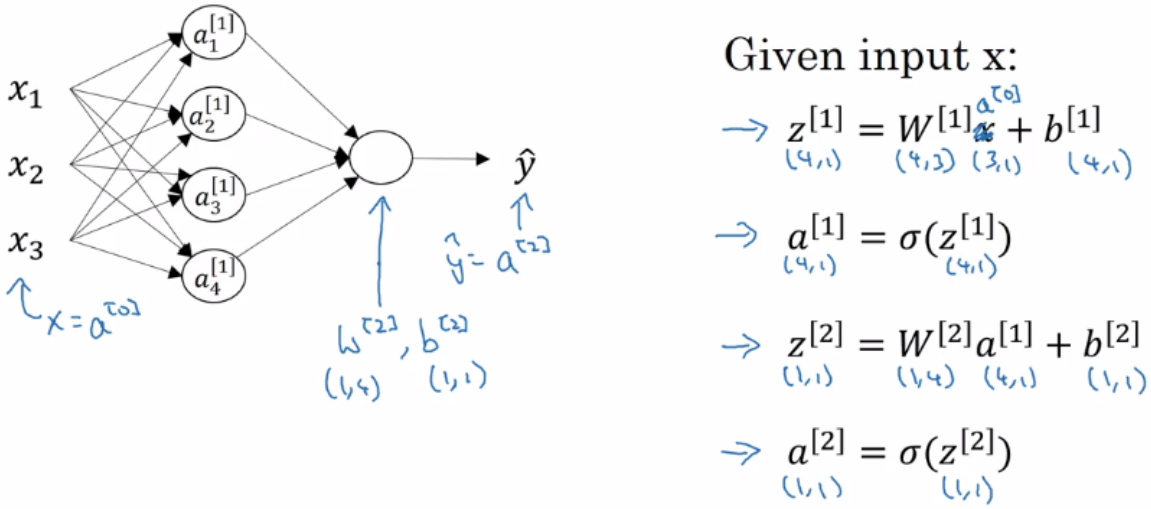
[https://postimg.cc/image/66pgpz08f/]

And the computation of $a^{[l]}$ just becomes the element-wise application of the sigmoid function:



[https://postimg.cc/image/n78cymaov/]

We can put it all together for our two layer neural network, and outline all the computations using our new notation:



[https://postimg.cc/image/h50q8noz3/]

Vectorizing across multiple examples

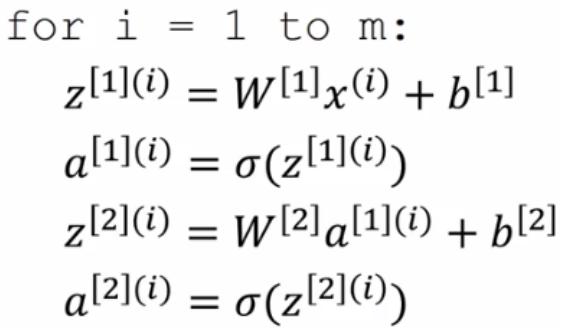
In the last video, we saw how to compute the prediction for a neural network with a single input example. In this video, we introduce a vectorized approach to compute predictions for many input examples.

We have seen how to take a single input example x and compute $a^{[2]} = \hat{y}$ for a 2-layered neural network. If we have m training examples, we can used a vectorized approach to compute all m predictions.

First, lets introduce a new notation. The activation values of layer l for input example i is:

$$a^{[l](i)}$$

The m predictions our 2-layered are therefore computed in the following way:



[https://postimg.cc/image/i7awr5acf/]

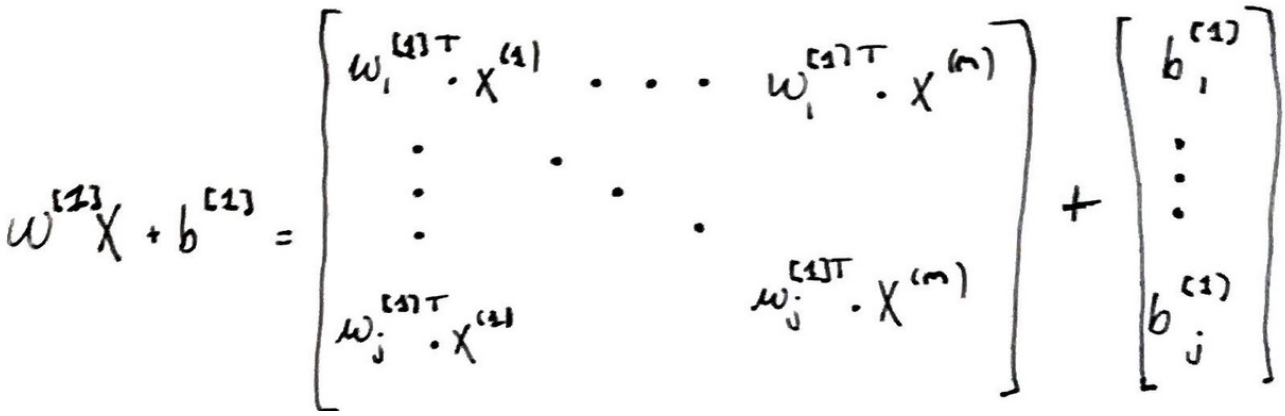
Recall that X is a (n_x, m) design matrix, where each column is a single input example and $W^{[l]}$ is a matrix where each row is the transpose of the parameter column vector for layer l.

Thus, we can now compute the activation of a layer in the neural network for all training examples:

$$Z^{[l]} = W^{[l]}X + b^{[l]}$$

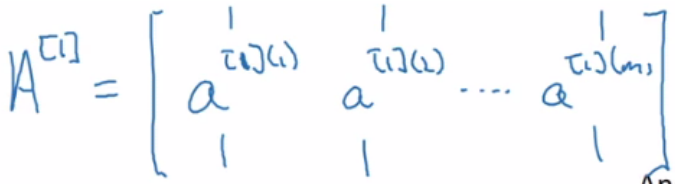
$$A^{[l]} = sign(Z^{[l]})$$

As an example, the result of a matrix multiplication of $W^{[1]}$ by X is a matrix with dimensions (j, m) where j is the number of units in layer 1 and m is the number of input examples



[https://postimg.cc/image/un7mkfljj/]

$A^{[l]}$ is therefore a matrix of dimensions (size of layer l X m). The top-leftmost value is the activation for the first unit in the layer l for the first input example i, and the bottom-rightmost value is the activation for the last unit in the layer l for the last input example m .



[https://postimg.cc/image/o9ijh617z/]

Activation Functions

So far, we have been using the **sigmoid** activation function

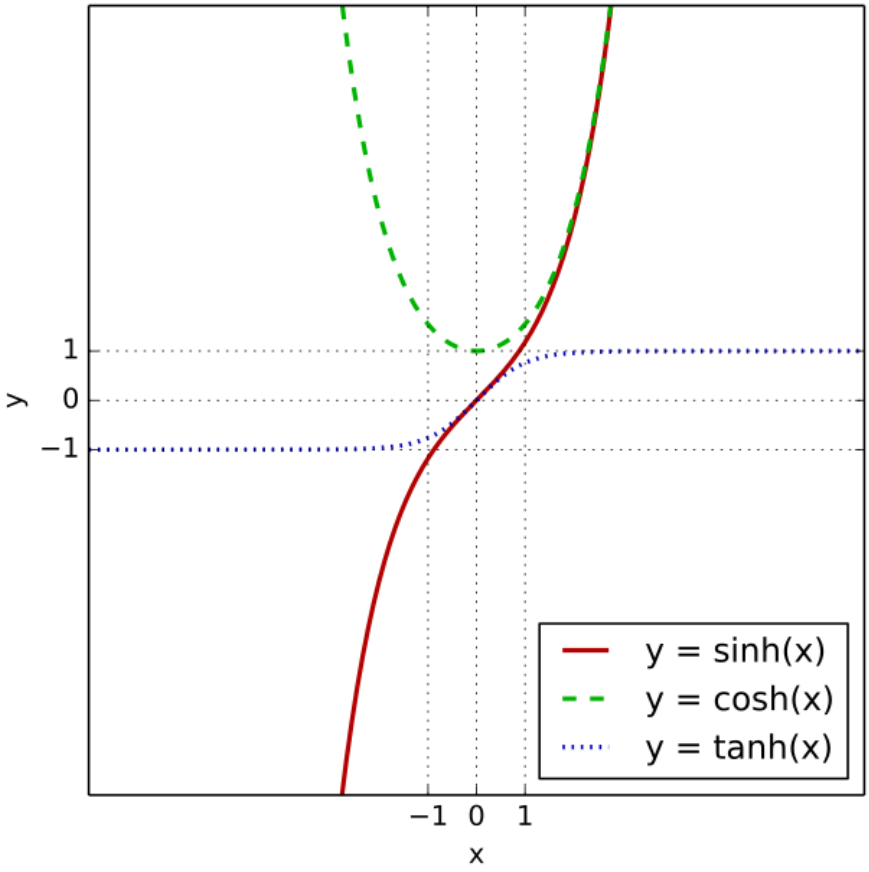
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

It turns out there are much better options.

Tanh

The **hyperbolic tangent function** is a non-linear activation function that almost always works better than the sigmoid function.

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



> The tanh function is really just a shift of the sigmoid function so that it crosses through the origin.

The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.

The single exception of sigmoid outperforming tanh is when its used in the ouput layer. In this case, it can be more desirable to scale our outputs from 0 to 1 (particularly in classification, when we want to output the probability that something belongs to a certain class). Indeed, we often mix activation functions in neural networks, and denote them:

$$g^{[p]}(z)$$

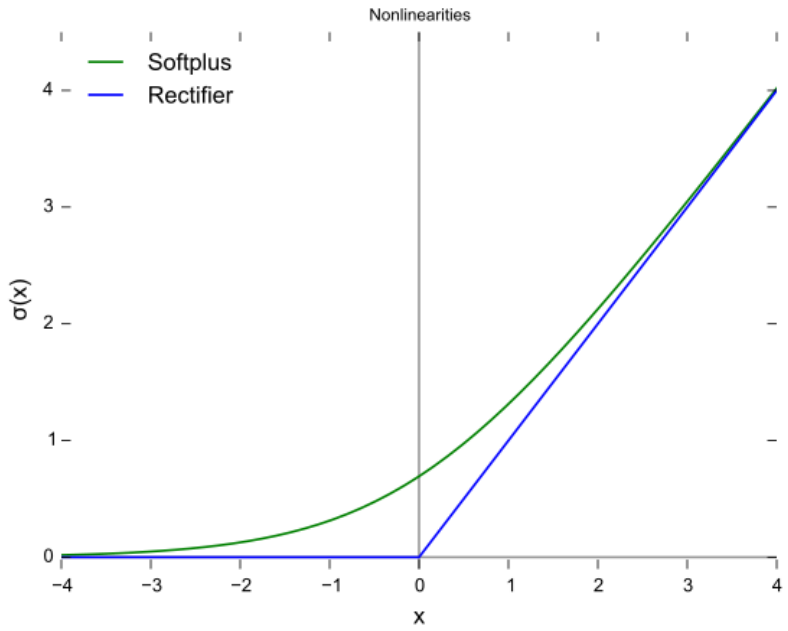
Where p is the p^{th} activation function.

If z is either very large, or very small, the derivative of both the tanh and sigmoid functions becomes very small, and this can slow down learning.

ReLu

The **rectified linear unit** activation function solves the disappearing gradient problem faced by tanh and sigmoid activation functions. In practice, it also leads to faster learning.

$$ReLu(z) = max(0, z)$$



> Note: the derivative at exactly 0 is not well-defined. In practice, we can simply set it to 0 or 1 (it matters little, due to the unlikeliness of a floating point number to ever be $0.0000\ldots$ exactly).

One disadvantage of ReLu is that the derivative is equal to 0 when z is negative. **Leaky ReLu**'s aim to solve this problem with a slight negative slope for values of $z < 0$.

$$ReLu(z) = max(0.01 * z, z)$$

> Image sourced from here [http://lamda.nju.edu.cn/weixs/project/CNNTricks/imgs/leaky.png].

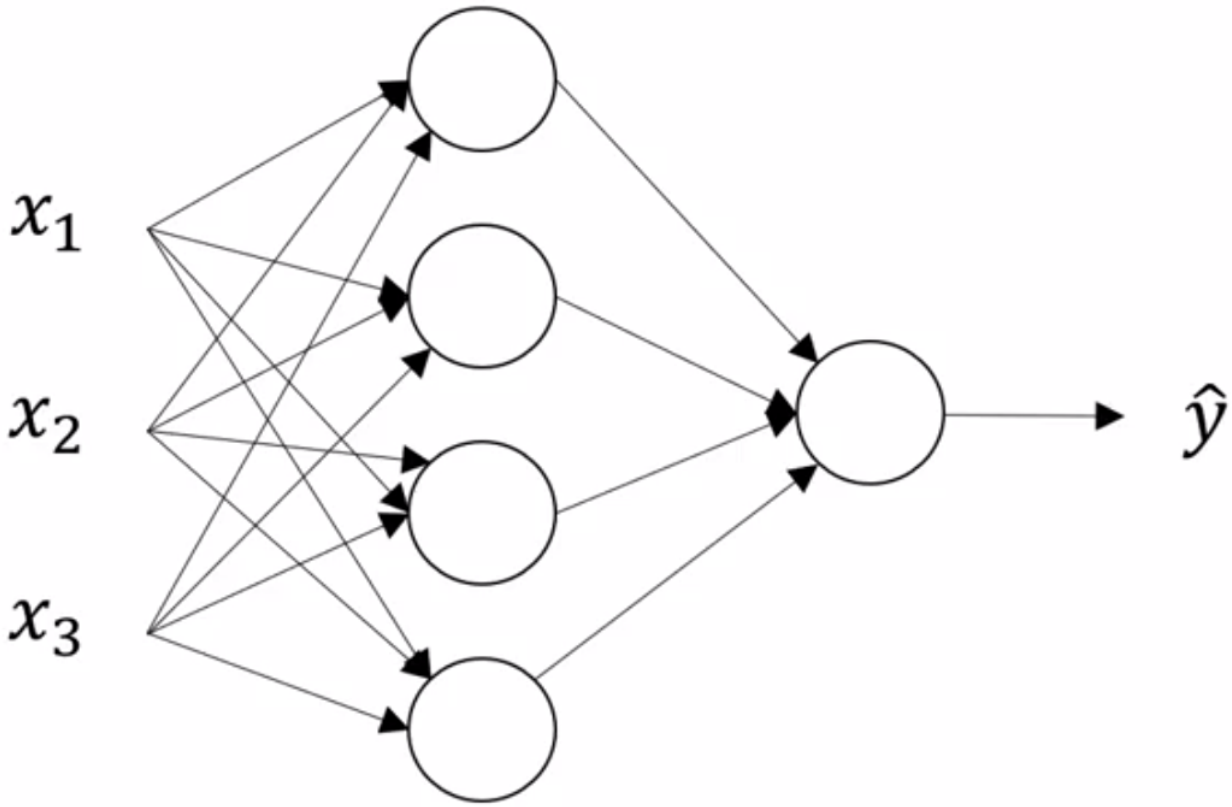
Sometimes, the 0.01 value is treated as an adaptive parameter of the learning algorithm. Leaky ReLu's solve a more general problem of "dead neurons [https://www.quora.com/What-is-the-definition-of-a-dead-neuron-in-Artificial-Neural-Networks?share=1]". However, it is not used as much in practice.

Rules of thumb for choosing activations functions

- *If your output is a 0/1 value*, i.e., you are performing binary classification, the sigmoid activation is a natural choice for the output layer.
- *For all other units*, ReLu's is increasingly the default choice of activation function.

Why do you need non-linear activation functions?

We could imagine using some **linear** activation function, $g(z) = z$ in place of the **non-linear** activation functions we have been using so far. Why is this a bad idea? Lets illustrate out explanation using our simple neural networks



[https://postimg.cc/image/4qdy8babj/]

For this linear activation function, the activations of our simple network become:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = z^{[1]}$$

$$z^{[2]} = W^{[2]}x + b^{[2]}$$

$$a^{[2]} = z^{[2]}$$

From which we can show that,

$$a^{[2]} = (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]})$$

$$a^{[2]} = W'x + b', \text{ where } W' = W^{[2]}W^{[1]} \text{ and } b' = W^{[2]}b^{[1]} + b^{[2]}$$

Therefore, in the case of a *linear activation function*, the neural network is outputting a *linear function of the inputs*, no matter how many hidden layers!

Exceptions

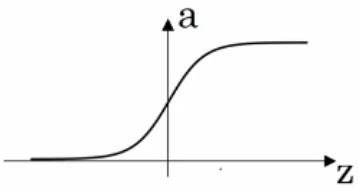
There are (maybe) two cases in which you may actually want to use a linear activation function.

1. The output layer of a network used to perform regression, where we want $\hat{y}$ to be a real-valued number, $\hat{y} \in \mathbb{R}$
2. Extremely specific cases pertaining to compression.

Derivatives of activation functions

When performing back-propogation on a network, we need to compute the derivatives of the activation functions. Lets take a look at our activation functions and their derivatives

Sigmoid



$$g(z) = \frac{1}{1+e^{-z}}$$

[https://postimg.cc/image/535ceiv67/]

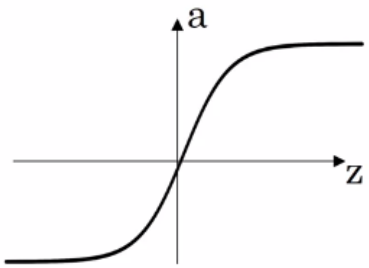
The deriviative of $g(z)$, $g(z)'$ is:

$$\frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}(1 - \frac{1}{1+e^{-z}}) = g(z)(1-g(z)) = a(1-a)$$

> We can sanity check this by inputting very large, or very small values of z into our derivative formula and inspecting the size of the outputs.

Notice that if we have already computed the value of a, we can very cheaply compute the value of $g(z)'$.

Tanh



$$g(z) = \tanh(z)$$

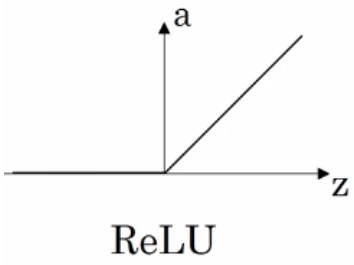
[https://postimg.cc/image/i7awr82nj/]

The deriviative of $g(z)$, $g(z)'$ is:

$$\frac{d}{dz}g(z) = 1 - (tanh(z))^z$$

> Again, we can sanity check this inspecting that the outputs for different values of z match our intuition about the activation function.

ReLu



[https://postimg.cc/image/iwtp3jswf/]

The derivative of $g(z)$, $g(z)'$ is:

$$\frac{d}{dz}g(z) = 0 \text{ if } z < 0; 1 \text{ if } z > 0; \text{ undefined if } z = 0$$

> If $z = 0$, we typically default to setting $g(z)$ to either 0 or 1 . In practice this matters little.

Gradient descent for Neural Networks

Lets implement gradient descent for our simple 2-layer neural network.

Recall, our parameters are: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$. We have number of features, $n_x = n^{[0]}$, number of hidden units $n^{[1]}$, and $n^{[2]}$ output units.

Thus our dimensions:

- $W^{[1]} : (n^{[1]}, n^{[0]})$
- $b^{[1]} : (n^{[1]}, 1)$
- $W^{[2]} : (n^{[2]}, n^{[1]})$
- $b^{[2]} : (n^{[2]}, 1)$

Our cost function is: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}, y)$

> We are assuming binary classification.

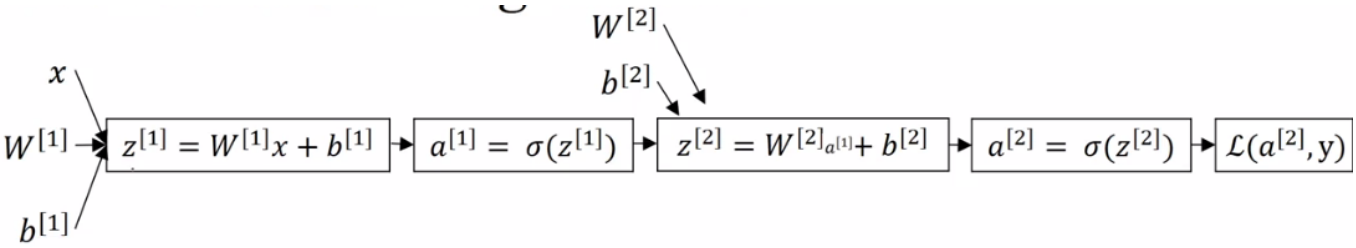
Gradient Descent sketch

1. Initialize parameters *randomly*
2. Repeat:
   - compute predictions $\hat{y}^{(i)}$ for $i = 1, \ldots, m$
   - $dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}, db^{[1]} = \frac{\partial J}{\partial b^{[1]}}, \ldots$
   - $W^{[1]} = W^{[1]} - \alpha dW^{[1]}, \ldots$
   - $b^{[1]} = b^{[1]} - \alpha db^{[1]}, \ldots$

The key to gradient descent is to computation of the derivatives, $\frac{\partial J}{\partial W^{[l]}}$ and $\frac{\partial J}{\partial b^{[l]}}$ for all layers l .

Formulas for computing derivatives

We are going to simply present the formulas you need, and defer their explanation to the next video. Recall the computation graph for our 2-layered neural network:



[https://postimg.cc/image/535cehkvj/]|

And the vectorized implementation of our computations in our **forward propagation**

1.

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

2.

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

3.

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

4.

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

> Where $g^{[2]}$ would likely be the sigmoid function if we are doing binary classification.

Now we list the computations for our **backward propagation**

1.

$$dZ^{[2]} = A^{[2]} - Y$$

2.

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

> Transpose of A accounts for the fact that W is composed of transposed column vectors of parameters.

3.

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

> Where $Y = [y^{(1)}, \dots, y^{[m]}]$. The `keepdims` arguments prevents numpy from returning a rank 1 array, $(n,)$

4.

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} \odot g(Z)'(Z^{[1]})$$

> Where $\odot$ is the element-wise product. Note: this is a collapse of dZ and dA computations.

5.

$$dW[1] = \frac{1}{m} = dZ^{[1]} X^T$$

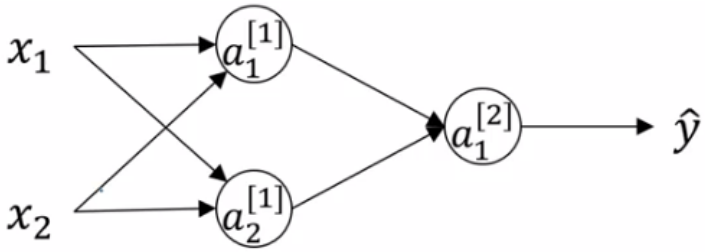
6.

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

Random Initialization

When you train your neural network, it is important to initialize your parameters *randomly*. With logistic regression, we were able to initialize our weights to *zero* because the cost function was convex. We will see that this *will not work* with neural networks.

Lets take the following network as example:



[https://postimg.cc/image/aslkya3r3/]

Lets say we initialize our parameters as follows:

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, W^{[2]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[2]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

> It turns out that initializing the bias b with zeros is OK.

The problem with this initialization is that for any input examples i, j,

$$a_i^{[1]} == a_j^{[1]}$$

Similarly,

$$dz_i^{[1]} == dz_j^{[1]}$$

Thus, $dW^{[1]}$ will be some matrix $\begin{bmatrix} u & v \\ u & v \end{bmatrix}$ and all updates to the parameters $W^{[1]}$ will be identical.

> Note we are referring to our single hidden layer $^{[1]}$ but this would apply to any hidden layer of any fully-connected network, no matter how large.

Using a *proof by induction*, it is actually possible to prove that after any number of rounds of training the two hidden units are still computing *identical functions*. This is often called the **symmetry breaking problem**.

The solution to this problem, is to initialize parameters *randomly*. Heres an example on how to do that with numpy:

- $W^{[1]}$ = `np.random.rand(2,2) * 0.01`
- $W^{[2]}$ = `np.random.rand(1,2) * 0.01`
- ...

> This will generate small, gaussian random values.

- $b^{[1]}$ = `np.zeros((2,1))`
- $b^{[2]}$ = `0`
- ...

> In next weeks material, we will talk about how and when you might choose a different factor than 0.01 for initialization.

It turns out the b does not have this symmetry breaking problem, because as long as the hidden units are computing different functions, the network will converge on different values of b, and so it is fine to initialize it to zeros.

Why do we initialize to small values?

For a *sigmoid-like* activation function, large parameter weights (positive or negative) will make it more likely that z is very large (positive or negative) and thus dz will approach 0, *slowing down learning dramatically*.

> Note this is a less of an issue when using ReLu's, however many classification problems use sigmoid activations in their output layer.

Week 4

Week 4: Deep Neural Networks

What is a deep neural network?

A deep neural network is simply a network with more than 1 hidden layer. Compared to logistic regression or a simple neural network with one hidden layer (which are considered **shallow** models) we say that a neural network with many hidden layers is a **deep** model, hence the terms *deep learning / deep neural networks*.

> Shallow Vs. deep is a matter of degree, the more hidden layers, the deeper the model.

Over the years, the machine learning and AI community has realized that deep networks are excellent function approximators, and are able to learn incredibly complex functions to map inputs to outputs.

> Note that is difficult to know in advance how deep a neural network needs to be to learn an effective mapping.

Notation

Lets go over the notation we will need using an example network

- We will use L to denote the number of layers in the network (in this network $L = 4$)
- $n^{[l]}$ denotes the number of units in layers l (for example, in this network $n^{[1]} = 5$)
- $a^{[l]}$ denotes the activations in layer l
- $W^{[l]}$, $b^{[l]}$ denotes the weights and biases for $z^{[l]}$
- $x = a^{[0]}$ and $\hat{y} = a^{[L]}$

Forward Propagation in a Deep Network

Forward propogation in a deep network just extends what we have already seen for forward propogation in a neural network by some number of layers. More specifically, for each layer l we perform the computations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

> Note that the above implementation is vectorized across all training examples. Matrices $A^{[l]}$ and $Z^{[l]}$ stacked column vectors pertaining to a single input example for layer l.

Finally, our predictions (the results of our output layer) are:

$$\hat{Y} = g(Z^{[L]}) = A^{[L]}$$

> Note that this solution is not completely vectorized, we still need an explicit for loop over our layers $l = 0, 1, \ldots, L$

Getting your matrix dimensions right

When implementing a neural network, it is extremely important that we ensure our matrix dimensions "line up". A simple debugging tool for neural networks then, is pen and paper!

For a l-layered neural network, our dimensions are as follows:

- $W^{[l]} : (n^{[l]}, n^{[l-1]})$
- $b^{[l]} : (n^{[l]}, 1)$
- $Z^{[l]}, A^{[l]} : (n^{[l]}, m)$
- $A^{[0]} = X : (n^{[0]}, m)$

Where $n^{[l]}$ is the number of units in layer l.

> See this [https://www.coursera.org/learn/neural-networks-deep-learning/lecture/rz9xJ/why-deep-representations] video for a derivation of these dimensions.

When implementing backpropagation, the dimensions are the same, i.e., the dimensions of W, b, A and Z are the same as dW, db, ...

Why deep representations?

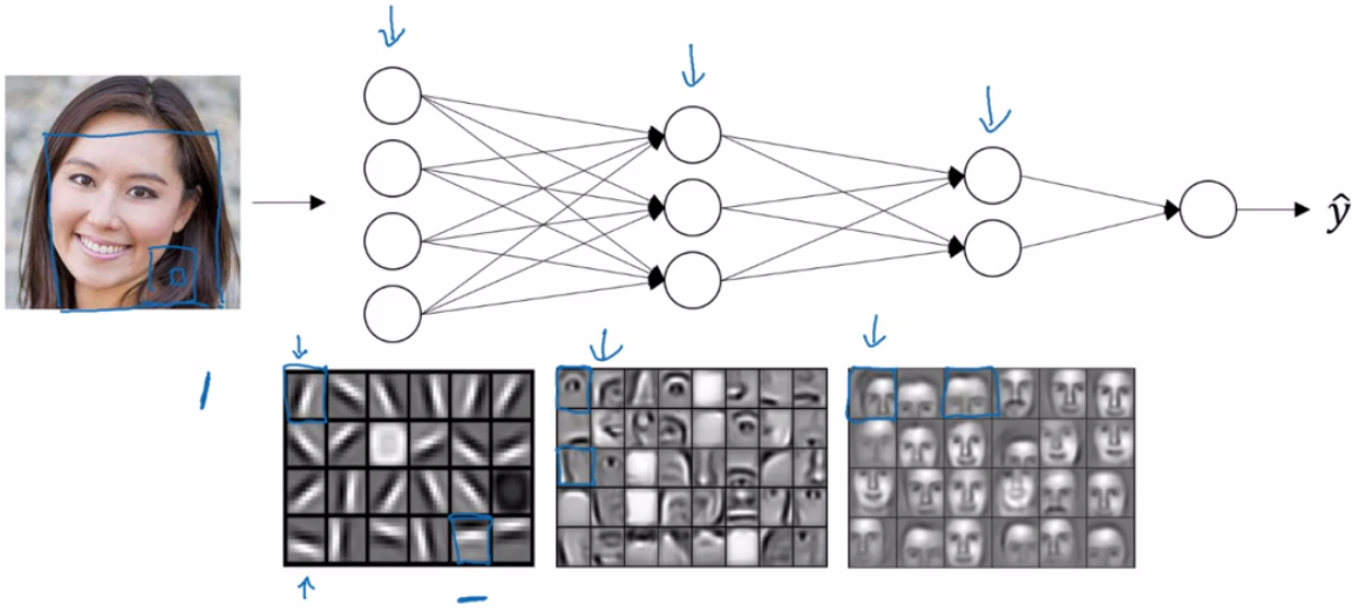
Lets train to gain some intuition behind the success of deep representation for certain problem domains.

What is a deep network computing?

Lets take the example of image recognition. Perhaps you input a picture of a face, then you can think of the first layer of the neural network as an "edge detector".

The next layer can use the outputs from the previous layer, which can roughly be thought of as detected edges, and "group" them in order to detect parts of faces. Each neuron may become tuned to detect different parts of faces.

Finally, the output layer uses the output of the previous layer, detected features of a face, and compose them together to recognize a whole face.



[https://postimg.cc/image/slpz8zvlr/]

> The main intuition is that earlier layers detect "simpler" structures, and pass this information onto the next layer which can use it to detect increasingly complex structures.

These general idea applies to other examples than just computer vision tasks (e.g., audio). Moreover, there is an analogy between deep representations in neural networks and how the brain works, however it can be dangerous to push these

analogies too far.

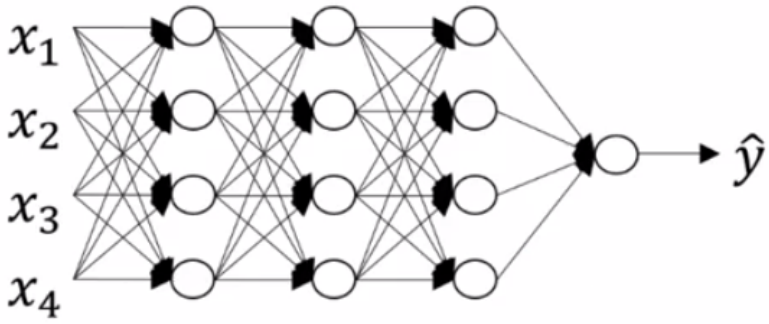
Circuit theory and deep learning

Circuit theory also provides us with a possible explanation as to why deep networks work so well for some tasks. Informally, there are function you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

> Check out this [https://www.coursera.org/learn/neural-networks-deep-learning/lecture/rz9xJ/why-deep-representations] video starting at 5:36 for a deeper explanation of this.

Building blocks of deep neural networks

Lets take a more holistic approach and talk about all the building blocks of deep neural networks. Here is a deep neural network with a few hidden layers



[https://postimg.cc/image/fgkkwtgun/]

Lets pick one layer, l and look at the computations involved.

For this layer l, we have parameters $W^{[l]}$ and $b^{[l]}$. Our two major computation steps through this layer are:

Forward Propagation

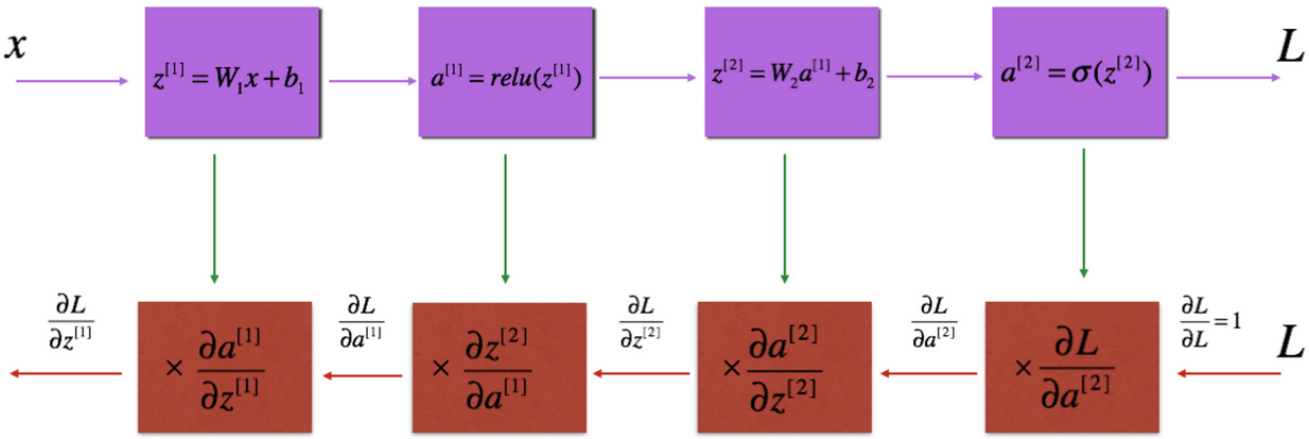
- Input: $a^{[l-1]}$
- Output: $a^{[l]}$
- Linear function: $z^{[l]} = W^{[l]}a^{[l-1]+b^{[l]}}$
- Activation function: $a^{[l]} = g^{[l]}(z^{[l]})$

Because $z^{[l]}, W^{[l]} and b^{[l]}$ are used in then backpropagation steps, it helps to cache theses values during forward propagation.

Backwards Propagation

- Input: $da^{[l]}, cache(z^{[l]})$
- Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

The key insight, is that for every computation in forward propagation there is a corresponding computation in backwards propagation



[https://postimg.cc/image/ct3ctjjnz/]

So one iteration of training with a neural network involves feeding our inputs into the network ($a^{[0]}$), performing forward propagation computing $\hat{y}$, and using it to compute the loss and perform backpropagation through the network. This will produce all the derivatives of the parameters w.r.t the loss that we need to update the parameters for gradient descent.

Parameters vs hyperparameters

The **parameters** of your model are the *adaptive* values, W and b which are *learned* during training via gradient descent.

In contrast, **hyperparameters** are set before training and can be viewed as the "settings" of the learning algorithms. They have a direct effect on the eventual value of the parameters.

Examples include:

- number of iterations
- learning rate
- number of hidden layers L
- number of hidden units $n^{[1]}, n^{[2]}, \ldots$
- choice of activation function

> the learning rate is sometimes called a parameter. We will follow the convetion of calling it a hyperparameter.

It can be difficult to know the optimal hyperparameters in advance. Often, we start by simply trying out many values to see what works best, this allows us to build our intuition about the best hyperparameters to use. We will defer a deep discussion on how to choose hyperparameters to the next course.

What does this all have to do with the brain?

At the risk of giving away the punch line, *not a whole lot*.

The most important mathematical components of a neural networks: *forward propagation* and *backwards propagation* are rather complex, and it has been difficult to convey the intuition behind these methods. As a result, the phrase, "it's like the brain" has become an easy, but dramatically oversimplified explanation. It also helps that this explanation has caught the publics imagination.

There is a loose analogy to be drawn from a biological neuron and the neurons in our artificial neural networks. Both take inputs (derived from other neurons) process the information and propagate a signal forward.

However, even today neuroscientists don't fully understand what a neuron is doing when it receives and propagates a signal. Indeed, we have no idea on whether the biological brain is performing some algorithmic processes similar to those performed by an ANN.

Deep learning is an excellent method for complex function approximation, i.e., learning mappings from inputs x to outputs y. However we should be very wary about pushing the, "its like a brain!" analogy too far.