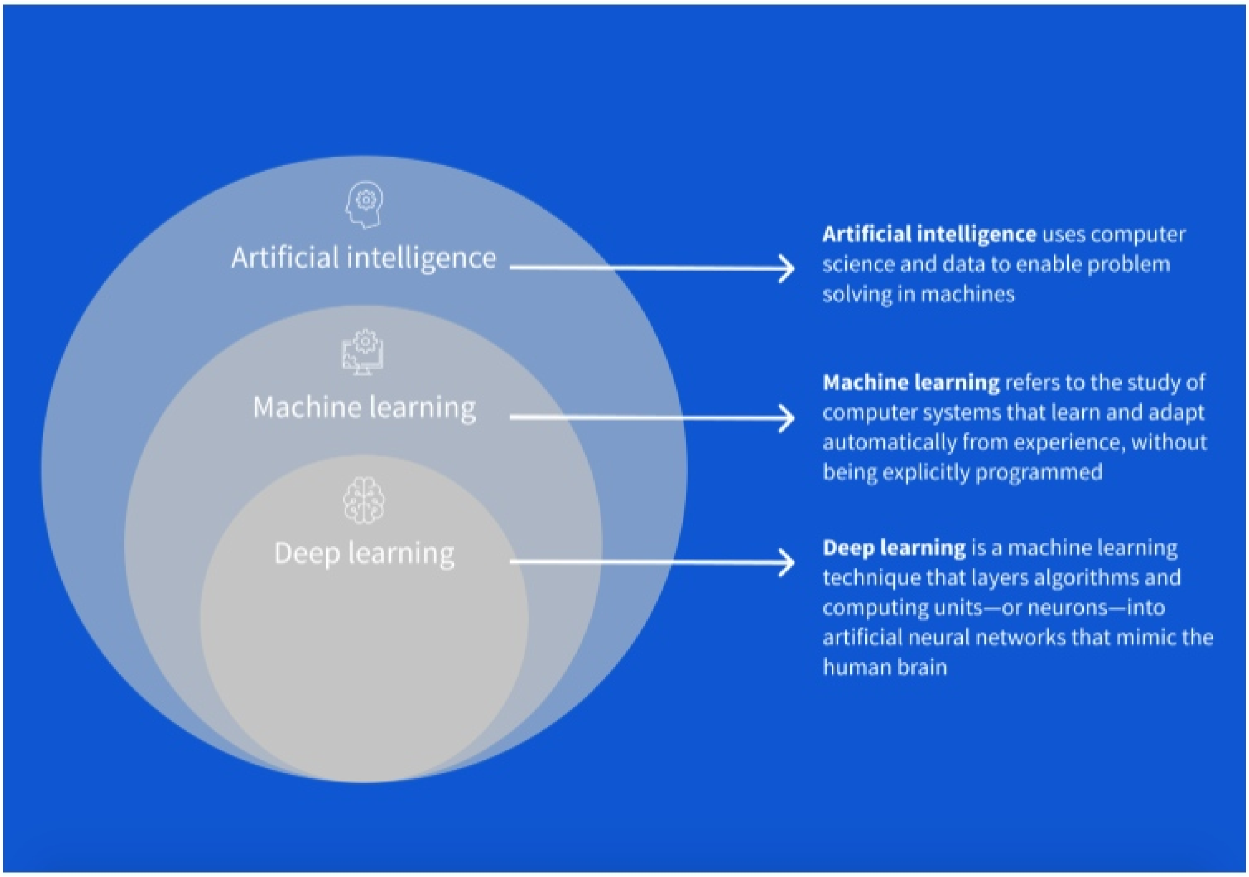
NEURAL NETWORKS AND DEEP LEARNING

Table of Contents

1. AI, ML, DL



Artificial intelligence

Machine learning

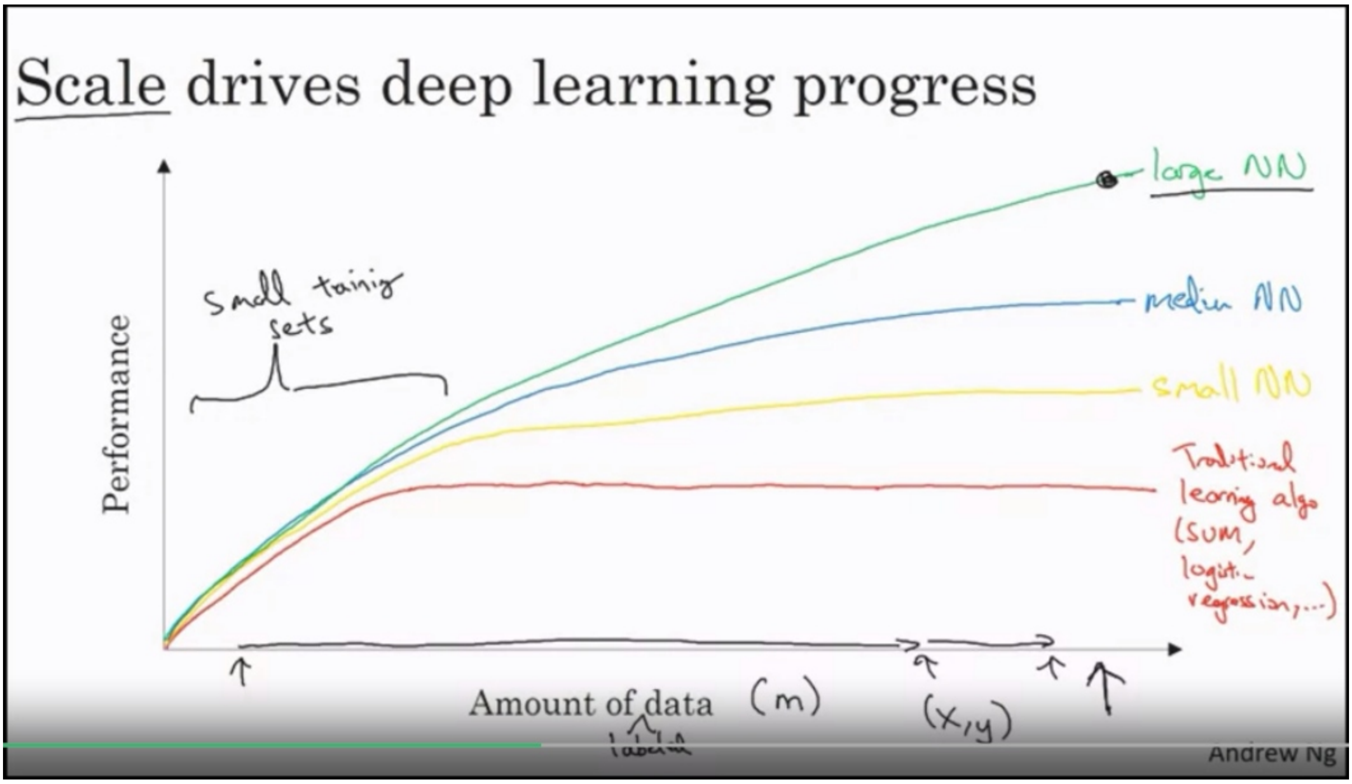
Deep learning

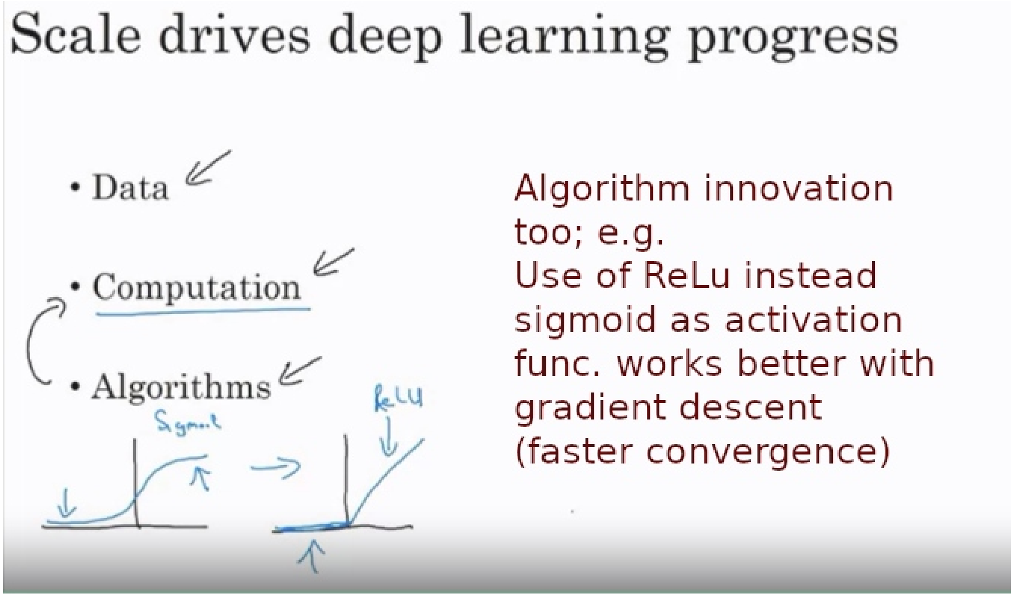
Artificial intelligence uses computer science and data to enable problem solving in machines

Machine learning refers to the study of computer systems that learn and adapt automatically from experience, without being explicitly programmed

Deep learning is a machine learning technique that layers algorithms and computing units—or neurons—into artificial neural networks that mimic the human brain

Scale drives deep learning progress

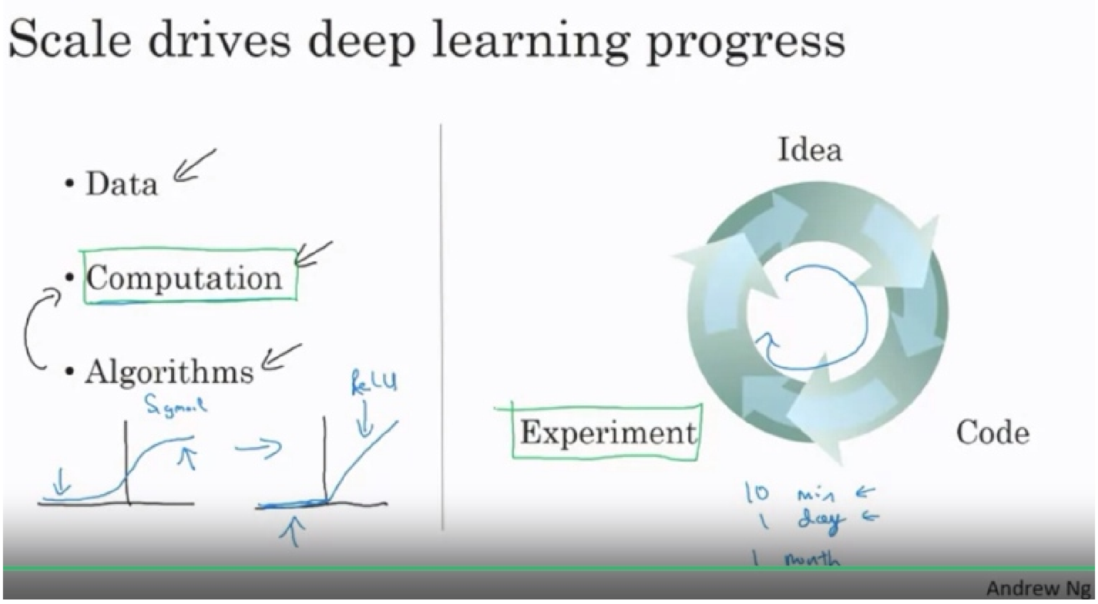




Scale drives deep learning progress

- Data
- Computation
- Algorithms

Algorithm innovation too; e.g.
Use of ReLu instead sigmoid as activation func. works better with gradient descent (faster convergence)



Scale drives deep learning progress

- Data
- Computation
- Algorithms

Idea

Experiment

Code

Andrew Ng

2. Interview with Geoffrey Hinton

I've seen you embroiled in debates about paradigms for AI*, and* whether there's been a paradigm shift for AI.
What are your, can you share your thoughts on that?

Yes, happily, so I think that in the early days, back in the 50s, people like von Neumann and Turing didn't believe in symbolic AI, they were far more inspired by the brain. Unfortunately, they both died much too young, and their voice wasn't heard. And in the early days of AI, people were completely convinced that the representations you need for intelligence were symbolic expressions of some kind. Sort of cleaned up logic, where you could do non-monotonic things, and not quite logic, but something like logic, and that the essence of intelligence was reasoning.

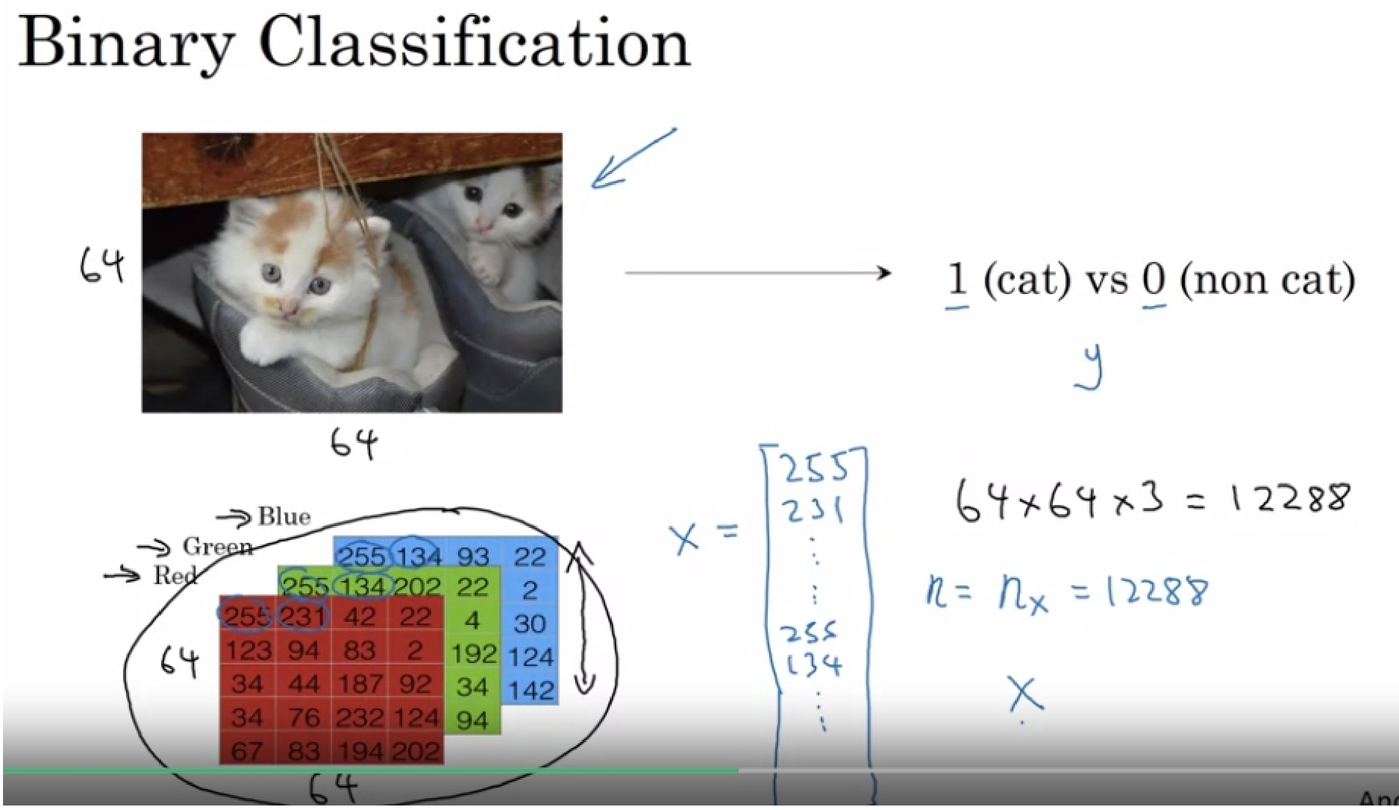
What's happened now is, there's a completely different view, which is that what a thought is, is just a great big vector of neural activity, so contrast that with a thought being a symbolic expression. And I think the people who thought that thoughts were symbolic expressions just made a huge mistake.

What comes in is a string of words, and what comes out is a string of words. And because of that, strings of words are the obvious way to represent things. So they thought what must be in between was a string of words, or something like a string of words. And I think what's in between is nothing like a string of words.
I think the idea that thoughts must be in some kind of language is as silly as the idea that understanding the layout of a spatial scene must be in pixels, pixels come in. And if we could, if we had a dot matrix printer attached to us, then pixels would come out, but what's in between isn't pixels.

And so I think thoughts are just these great big vectors, and that big vectors have causal powers. They cause other big vectors, and that's utterly unlike the standard AI view that thoughts are symbolic expressions.

3. Logistic Regression as Neural Network

Cat recognition in pictures:



Binary Classification

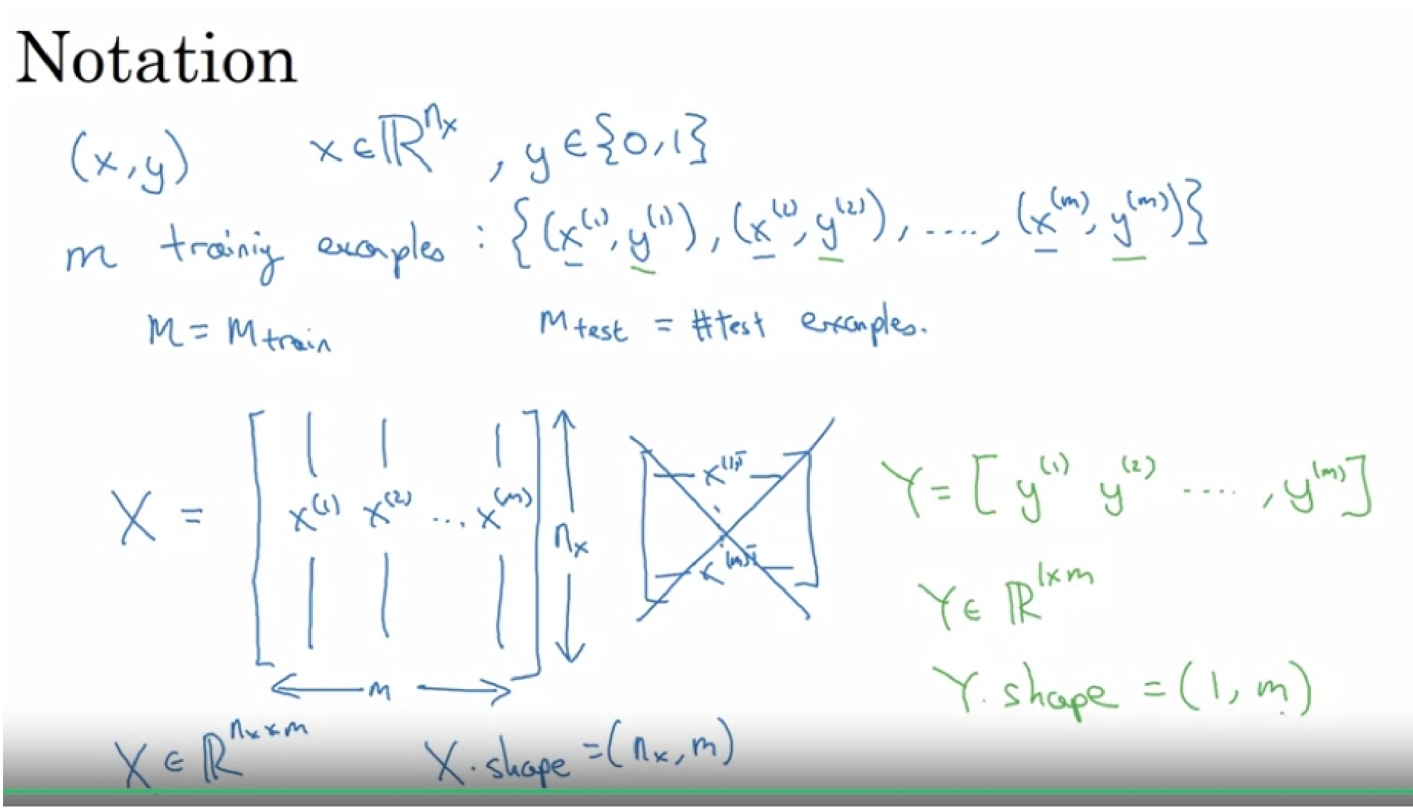
1 (cat) vs 0 (non cat)

y

$64 \times 64 \times 3 = 12288$

$n = n_x = 12288$

$X = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$



Notation

$(x, y) \qquad x \in \mathbb{R}^{n_x}, \; y \in \{0, 1\}$

m training examples : $\{(\underline{x}^{(1)}, \underline{y}^{(1)}), (\underline{x}^{(2)}, \underline{y}^{(2)}), \dots, (\underline{x}^{(m)}, \underline{y}^{(m)})\}$

$M = M_{train} \qquad M_{test} = \#$test examples.

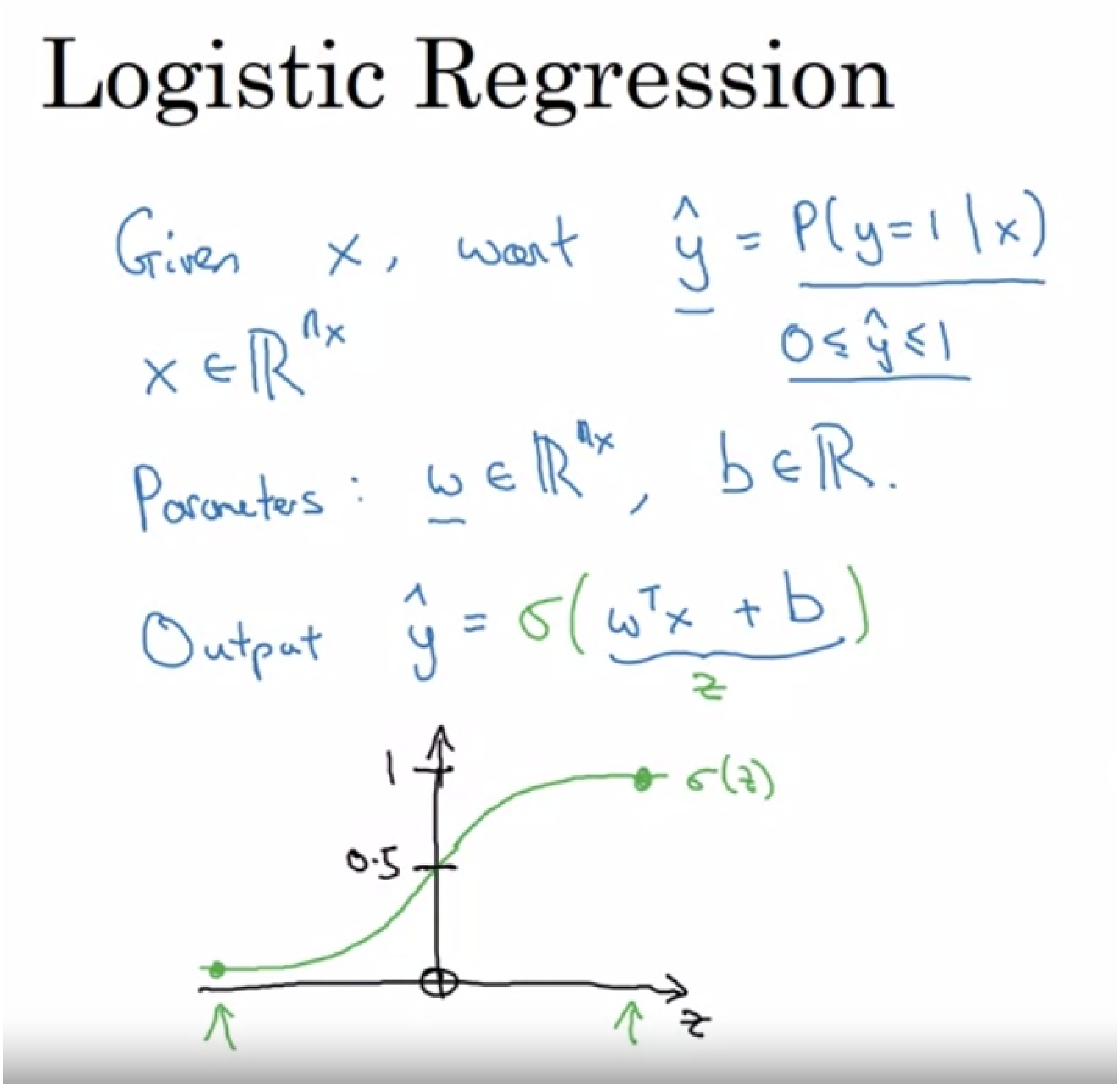
$$X = \underbrace{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}}_{\longleftrightarrow\; m \;\longleftrightarrow} \updownarrow n_x \qquad \cancel{\begin{bmatrix} - & x^{(1)T} & - \\ & \vdots & \\ - & x^{(m)T} & - \end{bmatrix}}$$

$Y = [\, y^{(1)} \; y^{(2)} \; \dots, y^{(m)} \,]$

$Y \in \mathbb{R}^{1 \times m}$

$Y.\text{shape} = (1, m)$

$X \in \mathbb{R}^{n_x \times m} \qquad X.\text{shape} = (n_x, m)$



Logistic Regression

Given x, want $\hat{y} = P(y=1 \mid x)$

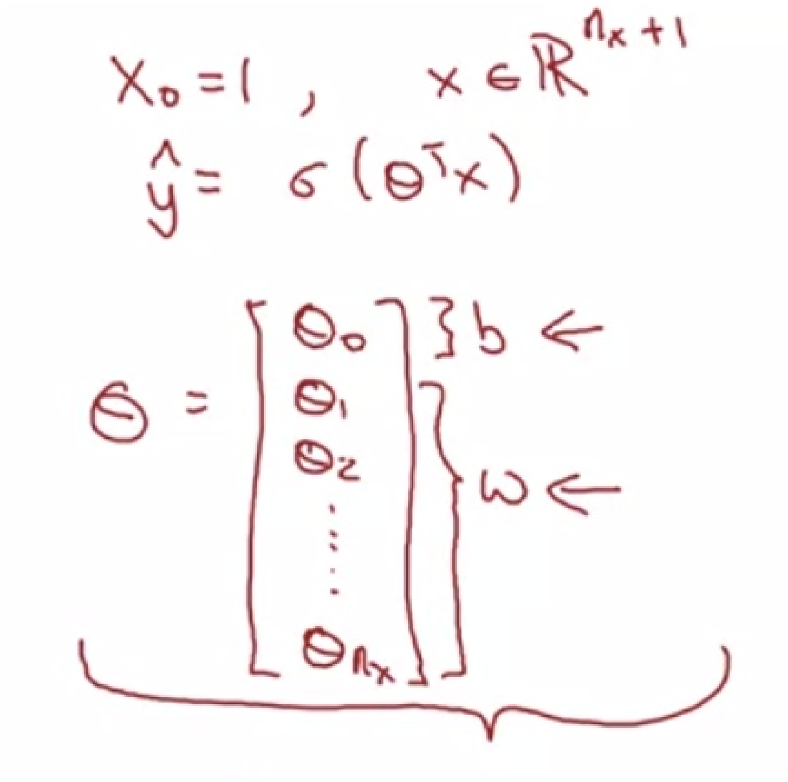
$x \in \mathbb{R}^{n_x}$

$0 \le \hat{y} \le 1$

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.

Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$

Notice the equivalence of notation of the coefficients (parameters) ω and b (the one we will use in this course) with the following notation used in the ML course with Andrew Ng:



$\widehat{y^{(i)}} = \sigma(\omega^T X^{(i)} + b)$ Probability of 1: having a cat in the picture (i)

$\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ The sigmoid function for the linear model

Note that when the algorithm has converged the probability predictions $\widehat{y^{(i)}}$ are transformed either to 0 or 1 according to their its value by using a threshold: normally $\widehat{y^{(i)}}$ maps to 1 if $\widehat{y^{(i)}} \geq 0.5$. Like that metrics like precision, recall, f1 can be used.

3.1. Cost Function

Of the algorithm for all training points:

$J(\omega,b)=\frac{1}{m}\sum_{i=1}^{m}L(\widehat{y^{(i)}},y^{(i)})$ Cost function for the whole data

$L(\widehat{y^{(i)}},y^{(i)})=-(y^{(i)}\log(\widehat{y^{(i)}})+(1-y^{(i)})\log(1-\widehat{y^{(i)}}))$ Logistic loss function (for a single observation)

$\log$ means natural logarithm

$0<L(\widehat{y^{(i)}},y^{(i)})<\infty$

$0\leq\widehat{y^{(i)}}\leq1$ is the prediction (probability) for observation (i). Therefore the $\log$ of the values above are always negative.

$y^{(i)}\in[0,1]$ is the true value of observation (i)

$L(\widehat{y^{(i)}},1)=-\log(\widehat{y^{(i)}})$ want $\widehat{y^{(i)}}$ close to 1 so loss tend to 0

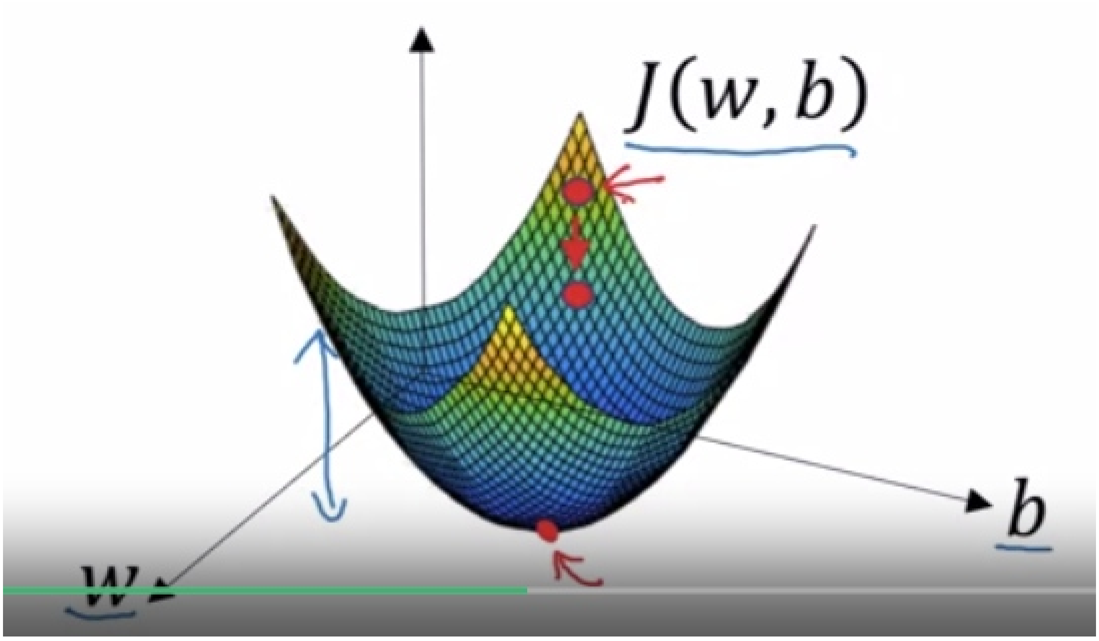
$L(\widehat{y^{(i)}},0)=-\log(1-\widehat{y^{(i)}})$ want $\widehat{y^{(i)}}$ close to 0 so loss tend to 0

$-\infty<\log(y)\leq0,0<y\leq1$

3.2. Gradient Descent

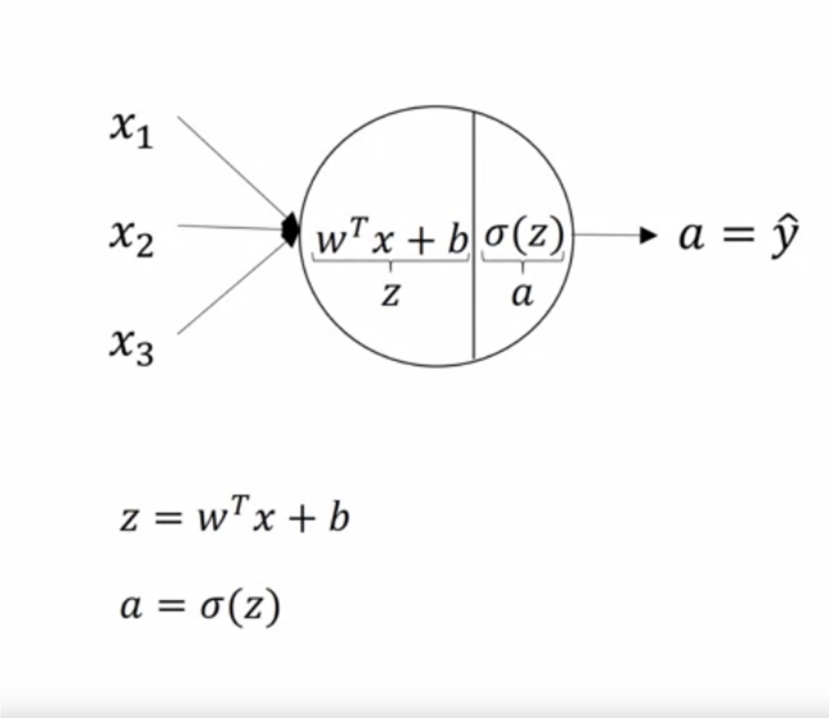
Find values ω,b that minimize $J(\omega,b)$

$J(\omega,b)$ **as described above is a function with a SINGLE MINIMUM,** no matter the dimensionality of ω



4. Shallow Neural Networks

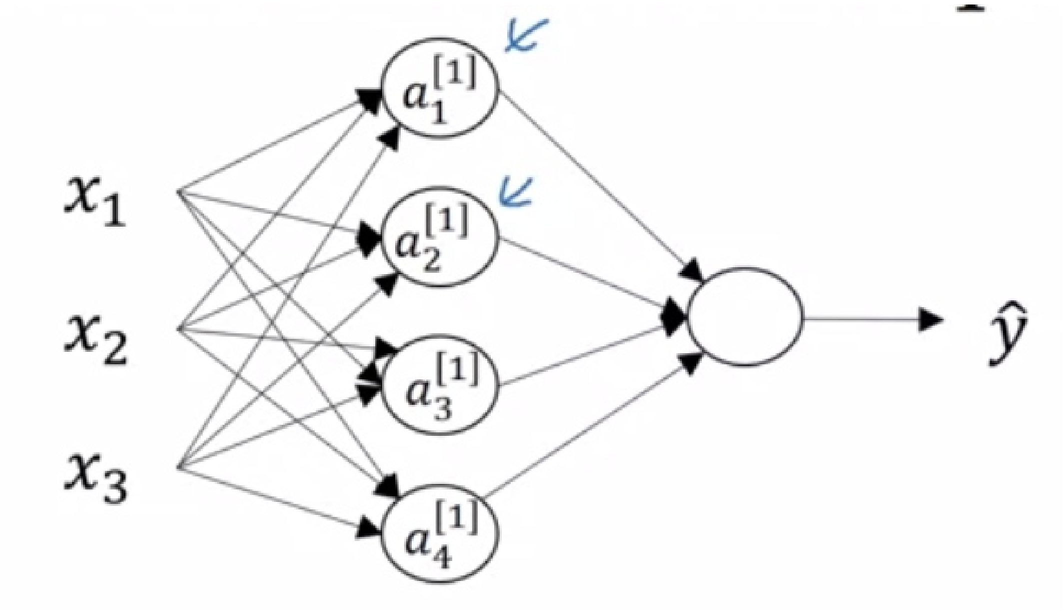
In logistic we regression we had:



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

$$a = \sigma(z)$$

4.1. Generalization to a Single Hidden-Layer Neural Network for a Single observation $\vec{x}$



Such a NN is composed of:

1. Input layer or layer 0: $\vec{a}^{[0]}=\vec{x}$
2. Hidden layer or layer 1: generally a matrix of the vectors:
    1. $\vec{a}_k^{[1]}$ with $1\leq k\leq 4$ in this example. Each vector has parameters $\vec{w}_k^{[1]}$ with $dim=(3,1)$ and $b_k^{[1]}$ a scalar.
3. Output layer or layer 2: $\vec{a}^{[2]}=\hat{y}$ a scalar

The square brackets in the notation above refer to the layer number

The sub-indices in the notation above refer to the node/neuron number in a layer

Hidden layers are the ones for which their values are not visible to the user.

A layer is composed of nodes. In this case the hidden layer has 4 nodes.

So for a **specific input observation** $\vec{x}=([x_1,x_2,x_3])^T$ we can write the values for the k^{th} node of layer 1 as:

$z_k^{[1]}=(\vec{w}_k^{[1]})^T(\vec{x})+b_k^{[1]}$ a scalar. $dim\left((\vec{w}_k^{[1]})^T(\vec{x})\right)=(1,3)(3,1)=(1,1)$

$z_k^{[1]}=(\vec{w}_k^{[1]})^T(\vec{a}^{[0]})+b_k^{[1]}$

with

$\vec{w}_k^{[1]}=([w_{k1}^{[1]},\ldots,w_{k3}^{[1]}])^T$ .With double sub-index:
$(k^{[1]}k^{[0]})=(current\, layer\, node\, ,index\, of\, previous\, layer\, output\, \vec{a}^{[0]})$

T denotes the transpose as required in the code for a matrix/vector multiplication. In the equation above the vector multiplication reduces to a inner product.

This can be written in vectorized form for all $z_k^{[1]}$:

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

with

$\vec{z}^{[1]}=([z_1^{[1]},\ldots z_4^{[1]}])^T \quad dim=(4,1)$

$\vec{b}^{[1]}=([b_1^{[1]},\ldots,b_4^{[1]}])^T \quad dim=(4,1)$

$\vec{x}=([x_1,x_2,x_3])^T \quad dim=(3,1)$

$W^{[1]}$ is a matrix with row vectors $\vec{w}_k^{[1]}$ and $1\leq k\leq 4$

$$W^{[1]}=\begin{bmatrix}(\vec{w}_1^{[1]})^T\\(\vec{w}_2^{[1]})^T\\(\vec{w}_3^{[1]})^T\\(\vec{w}_4^{[1]})^T\end{bmatrix} \qquad dim=(4,3)$$

with

$\vec{w}_k^{[1]}=([w_{k1}^{[1]},\ldots,w_{k3}^{[1]}])^T \quad dim=(3,1)$

Putting this together:

Output of layer 1:

$$\vec{z}^{[1]}=\begin{bmatrix}z_1^{[1]}\\z_2^{[1]}\\z_3^{[1]}\\z_4^{[1]}\end{bmatrix}=W^{[1]}\vec{x}+\vec{b}^{[1]}=\begin{bmatrix}(\vec{w}_1^{[1]})^T\\(\vec{w}_2^{[1]})^T\\(\vec{w}_3^{[1]})^T\\(\vec{w}_4^{[1]})^T\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}+\begin{bmatrix}b_1^{[1]}\\b_2^{[1]}\\b_3^{[1]}\\b_4^{[1]}\end{bmatrix} \qquad dim=(4,1)$$

Remember that $\vec{a}^{[0]}=\vec{x}$

And finally for layer 1:

$$\vec{a}^{[1]}=\sigma(\vec{z}^{[1]}) \qquad dim=(4,1)$$

Here the sigmoid function is applied elementwise on the vector $\vec{z}^{[1]}$. This function plays the role of a so-called activation function for each of the nodes of layer 1.

Following the same procedure for layer 2:

Output of layer 2:

$$\vec{z}^{[2]}=W^{[2]}\vec{a}^{[1]}+\vec{b}^{[2]} \qquad dim=(1,1)$$

$$\vec{a}^{[2]}=\sigma(\vec{z}^{[2]})=\hat{y} \qquad dim=(1,1)$$

Generalizing the procedure for layer 1 for one observation $\vec{a}^{[0]}=\vec{x}$, one can note that:

Output for layer l :

With $n^{[l]}=number\,of\,neurons\,\text{in}\,layer\,[l]$

$$dim(W^{[l]})=(n^{[l]},n^{[l-1]})$$

$$dim(\vec{b}^{[l]})=(n^{[l]},1)$$

$$dim(\vec{z}^{[l]})=(n^{[l]},1)$$

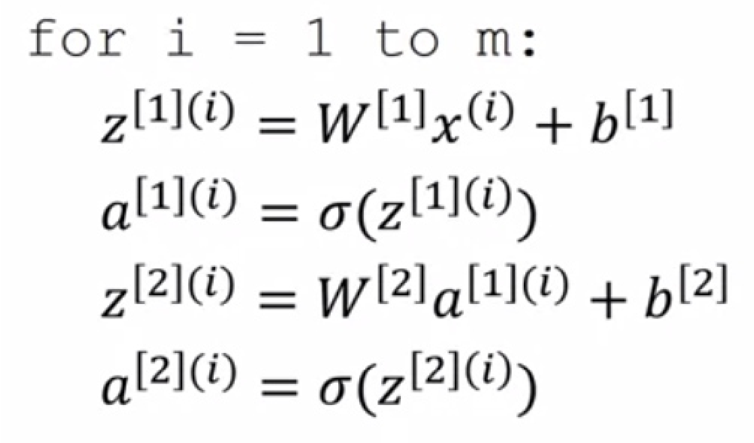
$$dim(\vec{a}^{[l]})=(n^{[l]},1)$$

So for $l=2$:

$$dim(W^{[2]})=(n^{[2]},n^{[1]})=(1,4)$$

$dim(\vec{z}^{[l]})=(1,1)$ a scalar

So with the equations for layer 1 and 2 we could train a NN model for all the observation in a data set by looping over through all observations:



```
for i = 1 to m:
    $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
    $a^{[1](i)} = \sigma(z^{[1](i)})$
    $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
    $a^{[2](i)} = \sigma(z^{[2](i)})$
```

NOTE: in the picture above $a^{[l](i)}$ **and** $b^{[l]}$ **are vectors**!

However this is computationally inefficient and we want to have it vectorized.

4.2. Shallow Neural Network Vectorization for all Observations

Let’s define a Matrix for all the observations:

$$X = A^{[0]} = [\vec{x}^{(1)}, \vec{x}^{(2)} \ldots, \vec{x}^{(m)}] \qquad dim = (n, m)$$

n : number of features

m : number of observations

with

$$\vec{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \ldots \\ x_n^{(i)} \end{bmatrix} \qquad 1 \le i \le m$$

4.2.a. Forward Propagation: Get predictions $\hat{\vec{y}}$

If one use analogies for the outputs of layers 1 and 2 but instead of having a single observation $\vec{x}^{(i)}$ we introduce the matrix , we will require also matrix-versions collecting the vectors $\vec{a}^{[l](i)}$ and $\vec{b}^{[l]}$:

For layer 1:

$$Z^{[1]}=W^{[1]}X+\vec{B}^{[1]}$$

$dim(Z^{[1]})=(\#of\ nodes\ \mathrm{in}\ layer[1],m)=(4,m)$ we will see why next:

with:

$$Z^{[1]}=[\vec{z}^{[1](1)},\vec{z}^{[1](2)},\dots,\vec{z}^{[1](m)}]$$

with:

$$\vec{z}^{[1](i)}=W^{[1]}\vec{x}^{(i)}+\vec{b}^{[1]}$$

- Superindices:
  - square brackets [ ]: layer index
  - round brackets: ( ): observation index

We know that:

- $W^{[1]}$ has $dim=(\#nodes\ \mathrm{in}\ layer\ 1,\#\ nodes\ \mathrm{in}\ layer\ 0)=(4,3)$
- $dim(\vec{b}^{[1]})=(\#\ of\ nodes\ \mathrm{in}\ layer[1],1)=(4,1)$
- $dim(X)=(n,m)=(3,m)$

So:

- $dim(W^{[1]}X)=(4,m)$
- Therefore it must hold $dim(B^{[1]})=(4,m)$ as well
- $B^{[1]}$ is just $\vec{b}^{[1]}$ replicated m-times along the horizontal axis and off all its elements are just $b^{[1]}$: $B^{[1]}=[\vec{b}^{[1]},\dots,\vec{b}^{[1]}]$ and $dim(B^{[1]})=(\#\ of\ nodes\ \mathrm{in}\ layer[1],m)=(4,m)$

It follows that:

$A^{[1]}=\sigma(Z^{[1]})$ and $dim(A^{[1]})=(\#of\ nodes\ \mathrm{in}\ layer[1],m)=(4,m)$

Following an analogy for layer 2 we have that:

Output for layer 2 :

$Z^{[2]}=W^{[2]}A^{[1]}+\vec{B}^{[2]}$ and

$dim(Z^{[2]})=(\# of\ nodes\ \text{in}\ layer[2],m)=(1,m)$

$A^{[2]}=\sigma(Z^{[2]})$ and $dim(A^{[2]})=(\# of\ nodes\ \text{in}\ layer[2],m)=(1,m)$

4.2.b. Summary Forward Propagation: Get predictions $\hat{\vec{y}}$

For all layers equations are symmetrical. From results for layers 1 and 2 one could derive a generalization for layer l .

Let’s introduce further notation for the nodes of the layers:

$n^{[l]}$: number of nodes in layer l

so:

$n=n^{[0]}$: number of nodes in layer 0 = number of features

Output for layer 1 :

$Z^{[1]}=W^{[1]}A^{[0]}+\vec{B}^{[1]}$ and $dim(Z^{[1]})=(n^{[1]},m)=(4,m)$

$A^{[1]}=\sigma(Z^{[1]})$ and $dim(A^{[1]})=(n^{[1]},m)=(4,m)$

with

$A^{[0]}=X=[\vec{x}^{(1)},\vec{x}^{(2)}\dots,\vec{x}^{(m)}]$ $\quad dim(X)=(n^{[0]},m)=(n,m)$

$dim(W^{[1]})=(n^{[1]},n^{[0]})=(4,3)$

$dim(B^{[1]})=(n^{[1]},m)=(4,m)$

Output for layer 2 :

$Z^{[2]}=W^{[2]}A^{[1]}+\vec{B}^{[2]}$ and $dim(Z^{[2]})=(n^{[2]},m)=(1,m)$

$\hat{\vec{y}}=A^{[2]}=\sigma(Z^{[2]})$ and $dim(A^{[2]})=(n^{[2]},m)=(1,m)$

with

$dim(W^{[2]})=(n^{[2]},n^{[1]})=(1,4)$

$dim(B^{[2]})=(n^{[2]},m)=(1,m)$

In the python code implementation of the sum in $Z^{[l]}=W^{[l]}A^{[l-1]}+\vec{B}^{[l]}$ one probably does not need to construct matrix or even a vector from the scalar values $b^{[l]}$. Python’s broadcasting mechanism can take care of this. It did in the implementation of logistic regression:

```
def propagate(w, b, X, y):
    """
    Implement the cost function and its gradient for the propagation explained
above

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1,
number of examples)
…

z = np.matmul(w.T,X) + b
```

4.2.c. Backward Propagation: Get Cost Function Gradient Vector

In this section we present the vectorized formulas for all observations for a binary classification with a one hidden layer NN.

The cost function remains the same as for logistic regression:

$$J=J\left(W^{[l]},b^{[l]}\right) \qquad 1\le i\le m$$

L : total (number of layers-1) because the input layer/ layer 0 is not counted and does not contribute to cost.

$$J=\frac{-1}{m}\sum_{i=1}^{m} Loss\left(\widehat{y^{(i)}},y^{(i)}\right)=\frac{-1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\widehat{y^{(i)}}\right)+\left(1-y^{(i)}\right)\log\left(1-\widehat{y^{(i)}}\right)\right) \quad \text{a scalar}$$

with $0\le\widehat{y^{(i)}}\le 1$ the probability of 1/true given by the activation sigmoid function.

The formulas for the back propagation are then:

Layer 2:

Gradient for layer 2 :

(i) $\frac{\partial J}{\partial Z^{[2]}} = dZ^{[2]} = A^{[2]} - \vec{y}$ and

$dim(dZ^{[2]}) = dim(Z^{[2]}) = (n^{[2]}, m) = (1, m)$

with

$\vec{y} = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$ $dim(A^{[2]}) = dim(\vec{y}) = (n^{[2]}, m) = (1, m)$

(ii) $\frac{\partial J}{\partial W^{[2]}} = dW^{[2]} = \frac{1}{m}(dZ^{[2]})(A^{[1]})^T$ and

$dim(dW^{[2]}) = dim(W^{[2]}) = (n^{[2]}, m)(m, n^{[1]}) = (n^{[2]}, n^{[1]}) = (1,4)$

(iii) $\frac{\partial J}{\partial b^{[2]}} = db^{[2]} = \frac{1}{m}\sum_{i=1}^{m} dZ^{[2]}$ and $dim(db^{[2]}) = dim(b^{[2]}) = 1$

Implement this in Python with: np.sum(dZ2, axis=1, keepdims=True)

Layer 1:

Gradient for layer 1 :

(i) $\frac{\partial J}{\partial Z^{[1]}} = dZ^{[1]} = (W^{[2]})^T (dZ^{[2]}) * \frac{\partial g^{[1]}(Z^{[1]})}{\partial Z^{[1]}}$

Important: $A * B$ denotes an **element-wise multiplication** of the matrices A and B

$dim(dZ^{[1]}) = dim(Z^{[1]}) = (n^{[1]}, n^{[2]})(n^{[2]}, m) = (n^{[1]}, m) = (4, m)$

Note that :

$$\frac{\partial g^{[1]}(Z^{[1]})}{\partial Z^{[1]}}$$

$g^{[l]}$ denotes the activation function of layer l .

In our case the sigmoid function for all layers and:

$dim(\frac{\partial \sigma^{[1]}}{\partial Z^{[1]}}) = dim(Z^{[1]}) = (n^{[1]}, m) = (4, m)$

(ii) $\frac{\partial J}{\partial W^{[1]}} = dW^{[1]} = \frac{1}{m}(dZ^{[1]})(X)^T$

$dim(dW^{[1]}) = dim(W^{[1]}) = (n^{[1]}, m)(m, n) = (n^{[1]}, n) = (4{,}3)$

(iii) $\frac{\partial J}{\partial b^{[1]}} = d\vec{b}^{[1]} = \frac{1}{m}\sum_{i=1}^{m} dZ^{[1]}$ since $dim(Z^{[1]}) = (n^{[1]}, m) = (4, m)$ it follows $dim(d\vec{b}^{[1]}) = (n^{[1]}, 1) = (4{,}1)$

Implement this in Python with: np.sum(dZ1, axis=1, keepdims=True)

4.3. Initialization of Parameters W and b in Neural Networks

In the logistic regression we initialized all parameters to zero and the the gradient descent started to work. This can be understood by viewing the LR as single layer, single-neuron NN (only input and output layers) with sigmoid activation function.

4.3.a. Different parameters for different neurons

This approach does not work in neural networks. Neural network parameters must be initialized to different values for different neurons/nodes in a layer; otherwise the functions represented by the neurons are identical. This means:

NN Requirements for Initialization Values of Model Parameters / Weights :	For a layer l and neurons with activation functions $a_p^{[l]}$, $a_q^{[l]}$ where p and q any different neurons in that layer, it **must hold that for any vectors of the matrix** $W^{[l]}$: $\vec{w}_p^{[l]} \neq \vec{w}_q^{[l]}$ Remember that such a vector looks like: $\vec{w}_k^{[l]} = ([w_{k1}^{[l]}, \dots, w_{kn^{[l-1]}}^{[l]}])^T$

where the elements are the parameters/weights for a given neuron k at a given layer l and $n^{[l-1]}$ is the number of neurons in the previous layer.

The bias $b^{[l]}$ (a scalar) can be initialized to zero as for the logistic regression.

4.3.b. Initialization values

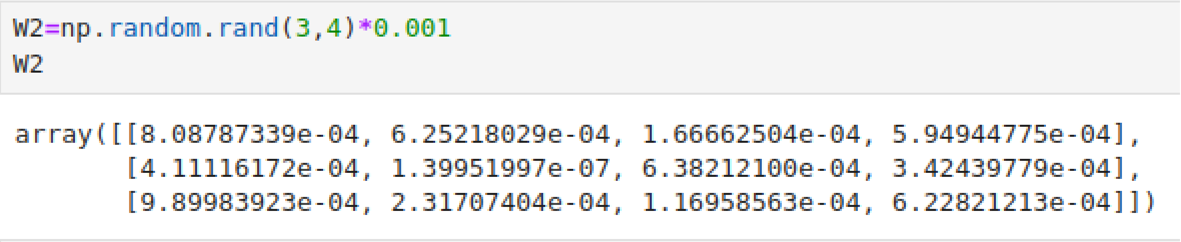
Initialization values depend on the type of activation function.

- *Sigmoid* and *tanh*: random real values close to zero. Choosing big init values implies derivative values that tends to zero (because of the flat shape of these function at extreme values). This drives the gradient descent to a nearly stagnation.

4.3.c. Code Implementation

Practically a whole matrix $W^{[l]}$ can be initialized to random values everywhere multiplied by a constant close to zero. This makes not only that the vectors of different neurons of a layer are different but also that the elements of such a vector are most probably different.

An example for layer 2:



```
W2=np.random.rand(3,4)*0.001
W2
```

```
array([[8.08787339e-04, 6.25218029e-04, 1.66662504e-04, 5.94944775e-04],
       [4.11116172e-04, 1.39951997e-07, 6.38212100e-04, 3.42439779e-04],
       [9.89983923e-04, 2.31707404e-04, 1.16958563e-04, 6.22821213e-04]])
```

The code above implies:

$l=2$

$n^{[2]}=3$

$n^{[1]}=4$

4.4. Activation Functions

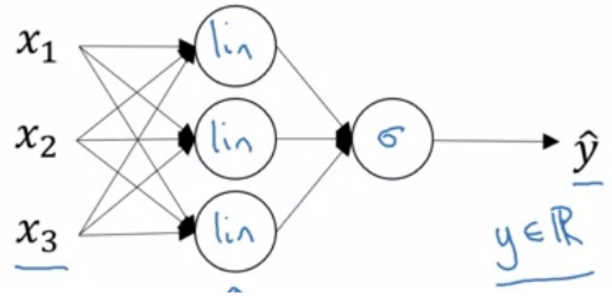
Facts:

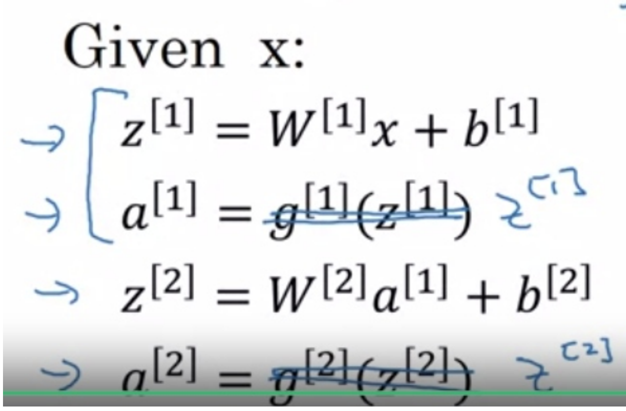
1. In neural networks the activation functions of hidden layers must be non-linear
2. In the output layer the activation function can be linear; for example for a linear regression model
3. The sigmoid function is nowadays not any more used as activation function for the hidden layers. Only for the output layer for binary classifications like logistic regression.
4. Normally a layer uses the same activation functions for all its units/nodes.

4.4.a. Non-Linearity of Activation Function in Hidden Layers

Using linear functions or no functions (identity function) in the hidden layers creates at the end a linear model, which is in principle not what we want with a neural network. Example:

Let us the identity function as activation function for the layers 1 and 2:

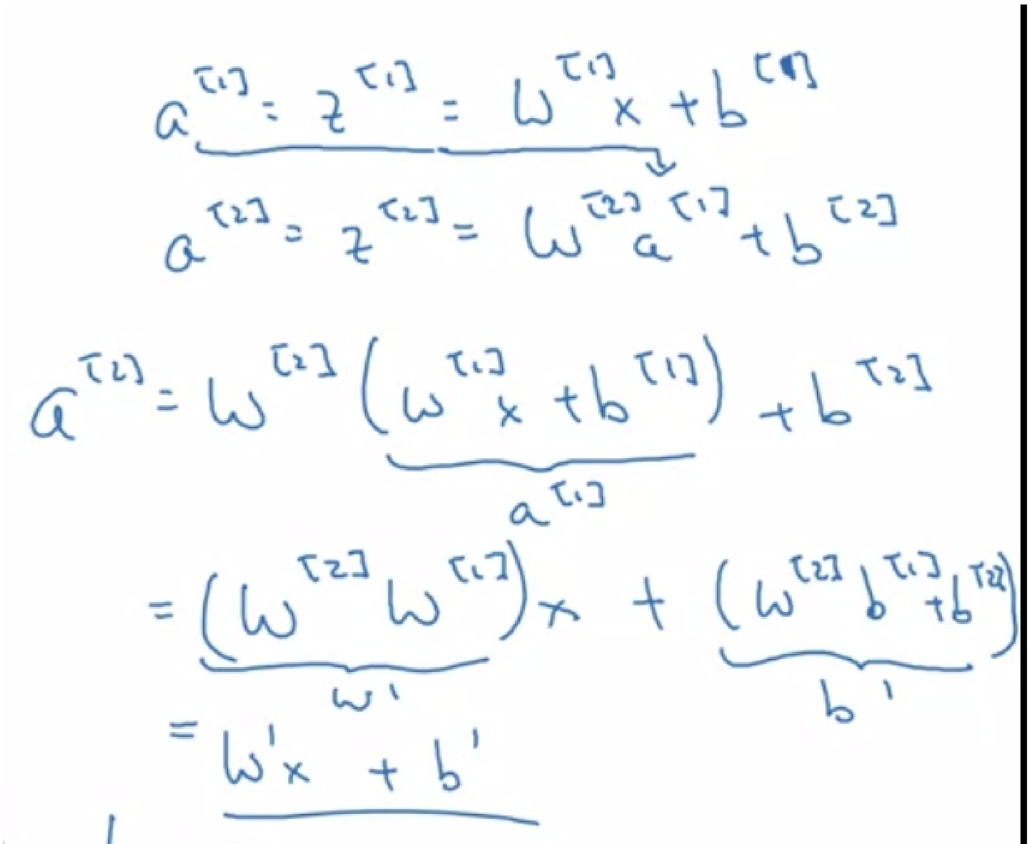



Given x:

$$\begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[1]} = \cancel{g^{[1]}(z^{[1]})}\ z^{[1]} \end{cases}$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \cancel{g^{[2]}(z^{[2]})}\ z^{[2]}$$



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

$$a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = W^{[2]}\underbrace{\left(W^{[1]}x + b^{[1]}\right)}_{a^{[1]}} + b^{[2]}$$

$$= \underbrace{\left(W^{[2]}W^{[1]}\right)}_{W'}x + \underbrace{\left(W^{[2]}b^{[1]} + b^{[2]}\right)}_{b'}$$

$$= \underline{W'x + b'}$$

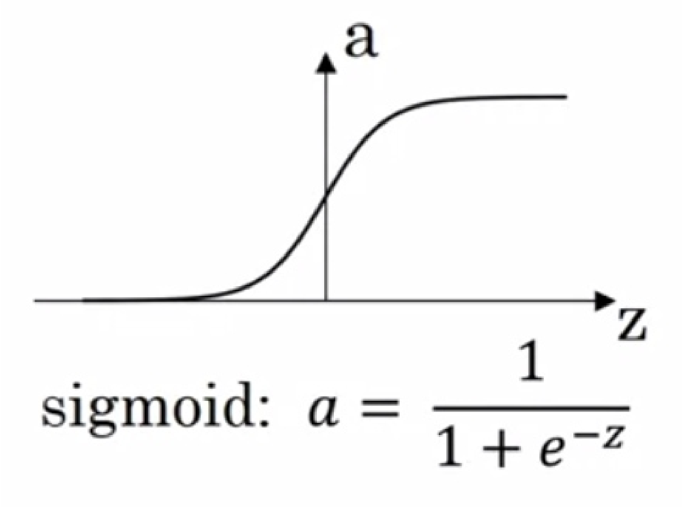
In the equations above we see the predictions have a clear linear form.

If we had used no function for layer 1 and sigmoid for layer 2 we had built a logistic regression with a most likely useless hidden layer.

4.4.b. Common Activation Functions

4.4.b.1. Sigmoid

$l=2$



$$0<g(z)=a<1$$

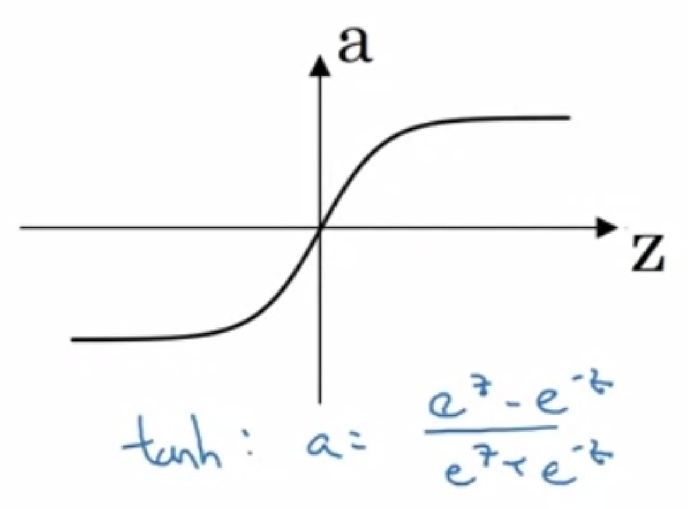
Derivative:

$$g'(z)=\frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)=a(1-a)$$

Disadvantages:

- not centered a zero but at 0.5 what can pose some problems in gradient descent
- Derivative tends to zero for large values (absolute values) which makes gradient descent to converge slowly

4.4.b.2. Hyperbolic Tangent tanh



$$-1 < g(z) = a < 1$$

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

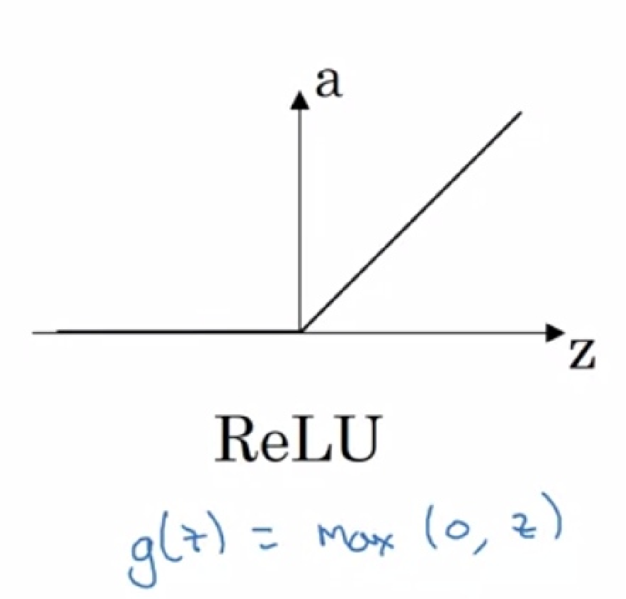
Derivative:

$a' = g'(z) = 1 - (\tanh(z))^2 = 1 - \tanh^2(z) = 1 - a^2$

Facts:

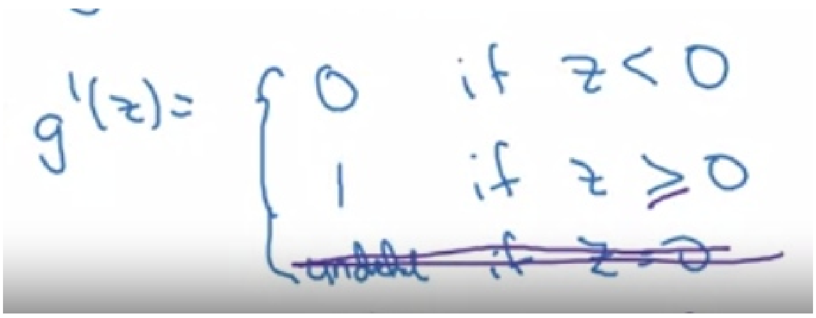
- Centered at zero: better for gradient descent than sigmoid
- Same shape of sigmoid for the derivative: slow convergence

4.4.b.3. ReLU: Rectified Linear Unit



$$g(z) = \max(0, z)$$

Derivative:



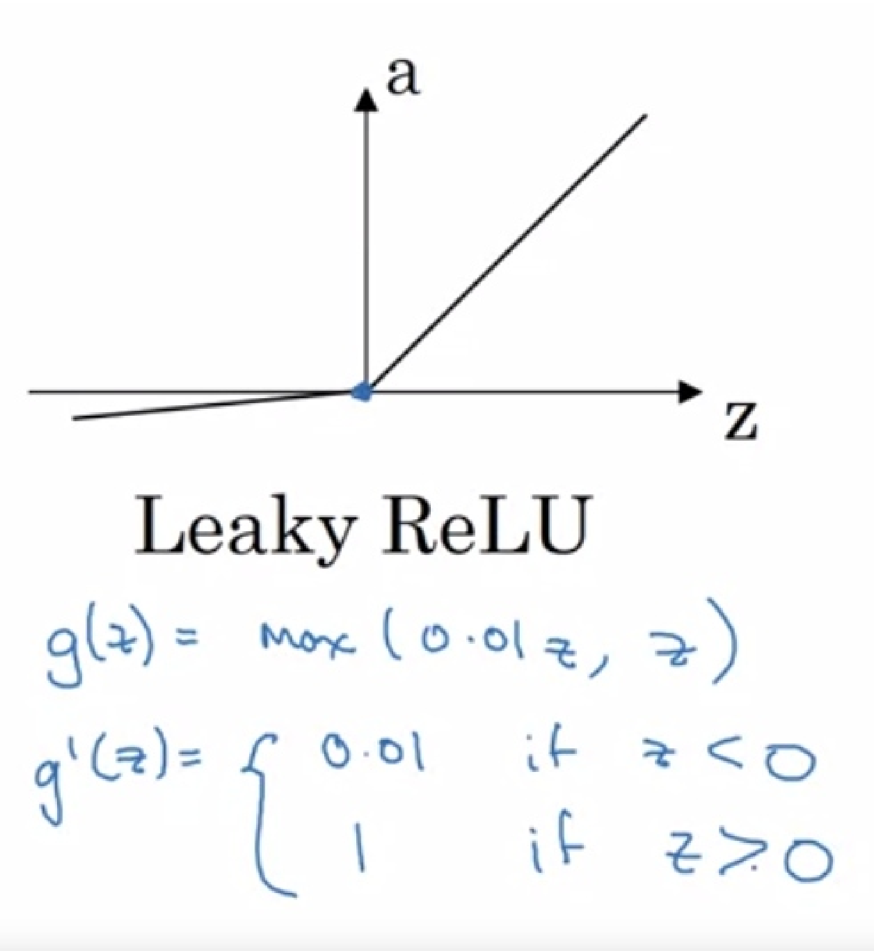
$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \\ \cancel{\text{undef}} & \cancel{\text{if } z = 0} \end{cases}$$

Facts:

- Non-differentiable at zero, which can be targeted in code as shown above in the expression for the derivative.
- 2 constant values of the derivative but one is 0. This can slow down gradient descent.
- Most widely used nowadays

4.4.b.4. Leaky ReLU

$l=2$



$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

There is nothing special about the value 0.01 for negative z. It can be another “small” positive number.

Facts:

- Not as widely used as ReLU
- Should be better than ReLU because of non-zero derivatives

5. Deep Neural Networks

The reasoning used for the shallow NN can be generalized for NN with more hidden layers.

NOTE: In this course when a L-NN is referred it means the NN has L+1 layers including the input layer. This is convenient because the input layer index is not used in the FWD and BWD propagation excpet for the X term in FWD.

5.1. Recapitulation: What is this all about?

5.1.a. Aim

We want to find the best parameters $W^{[l]}, b^{[l]}$ for our model for all layers with $l > 0$

5.1.b. Method

We use the gradient descent algorithm for that:

1. Initialize parameters $W^{[l]}, b^{[l]}$
2. FWD propagation: Calculate predictions for all observations in training set for current values of parameters.
3. Compute cost: metric on how good our predictions are
4. BWD propagation: Compute gradient vector of cost function in the space $J(W^{[l]}, b^{[l]})$ and so get updated values for $W^{[l]}, b^{[l]}$. From here we iterate: back to point 2. of this list. And so on for the specified number of iterations or until some condition is fulfilled.

5.2. Generalization FWD and BWD Propagation

In the following generalization, we assume that all neurons/nodes/units in a given layer l share the same activation function $g^{[l]}(Z^{[l]})$. However different layers can have different activation functions.

We assume the cost function has the form from the logistic regression:

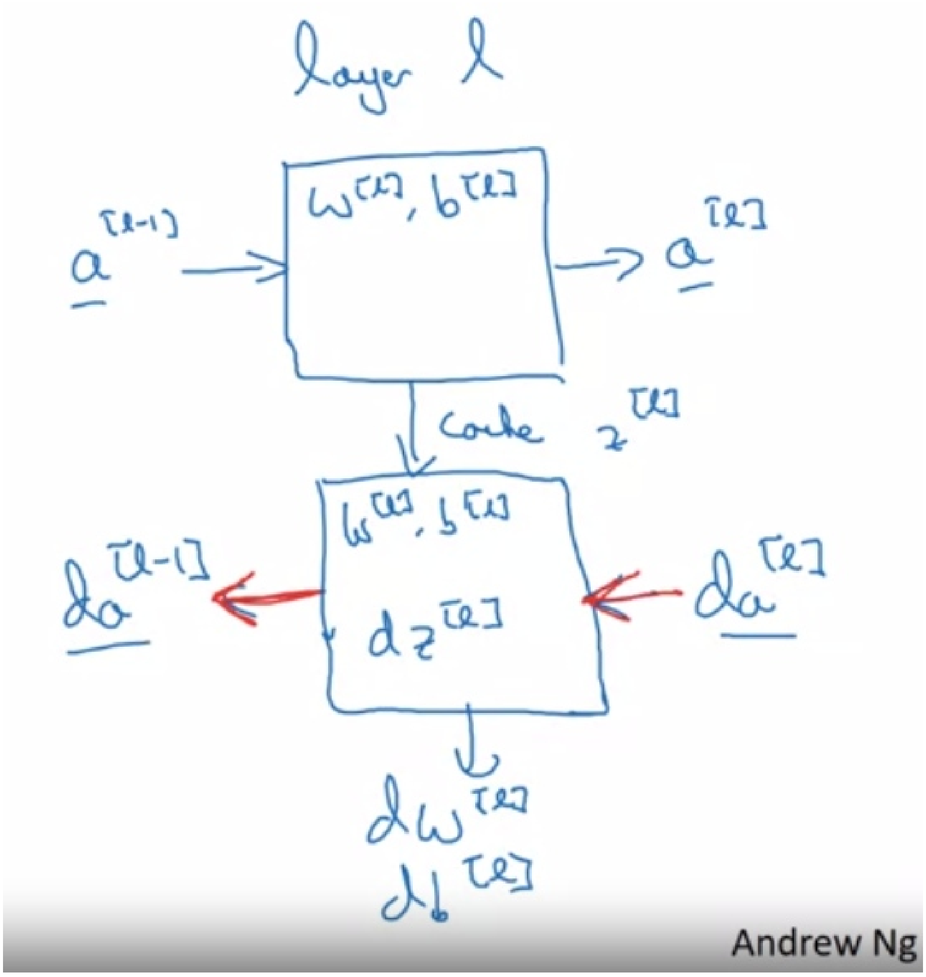
$$J=\frac{-1}{m}\sum_{i=1}^{m} Loss(\widehat{y^{(i)}}, y^{(i)})=\frac{-1}{m}\sum_{i=1}^{m}(y^{(i)}\log(\widehat{y^{(i)}})+(1-y^{(i)})\log(1-\widehat{y^{(i)}}))$$

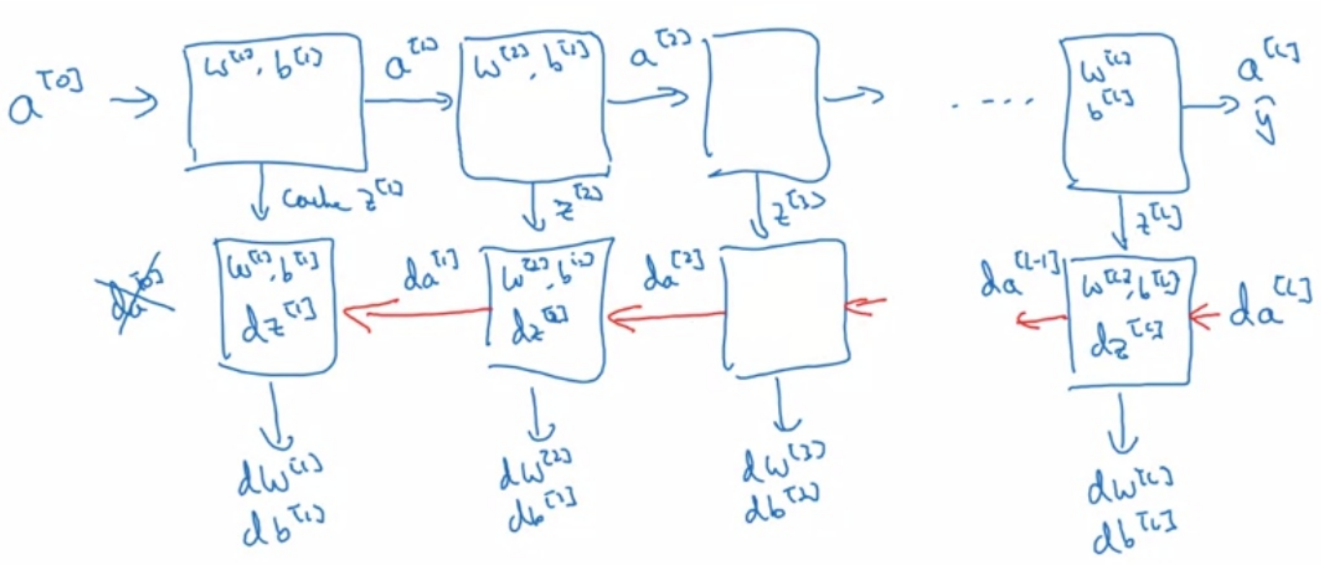
where $\hat{y}^{(i)}$ is the probability prediction for 1/Positive for observation m .

5.3. No Vectorization for layers Possible. Caching

Because layers are dependent of output calculations of adjacent layers both in FWD and BWD propagations, it is not possible to vectorize (parallelize) this process. A code loop must be implemented for going through layers.

A caching mechanism is useful to store values needed later. Specifically we store the $Z^{[l+1]}$ calculated during FWD propagation which are required by the BWD propagation for layer : $dZ^{[l]}=dZ^{[l]}(dZ^{[l+1]})$.





$a^{[0]}$
$W^{[1]}, b^{[1]}$
$a^{[1]}$
$W^{[2]}, b^{[2]}$
$a^{[2]}$
. . . .
$W^{[L]}$
$b^{[L]}$
$a^{[L]}$
$\hat{y}$
cache $z^{[1]}$
$z^{[2]}$
$z^{[3]}$
$z^{[L]}$
~~$da^{[0]}$~~
$W^{[1]}, b^{[1]}$
$dz^{[1]}$
$da^{[1]}$
$W^{[2]}, b^{[2]}$
$dz^{[2]}$
$da^{[2]}$
$da^{[L-1]}$
$W^{[L]}, b^{[L]}$
$dz^{[L]}$
$da^{[L]}$
$dW^{[1]}$
$db^{[1]}$
$dW^{[2]}$
$db^{[2]}$
$dW^{[3]}$
$db^{[3]}$
$dW^{[L]}$
$db^{[L]}$

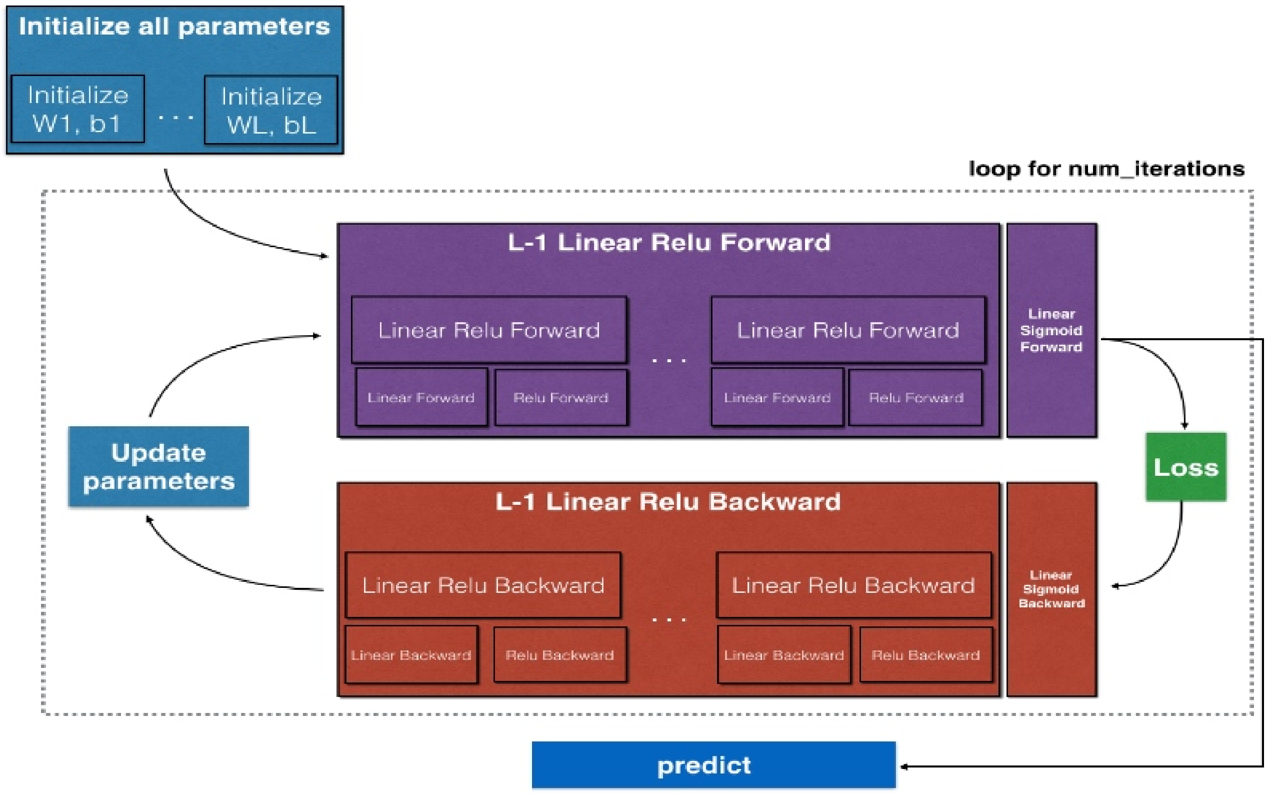


Figure 1

5.3.a. FWD propagation

Vectorized generalization for computation of output of a layer in an iteration for all observations in training set:

Output for layer $l=0$:	$A^{[0]}=X=[\vec{x}^{(1)},\vec{x}^{(2)}...,\vec{x}^{(m)}]$ $dim(X)=(n^{[0]},m)=(n,m)$ Here no computations are required!
Output for layer $l>0$:	(i) $Z^{[l]}=W^{[l]}A^{[l-1]}+B^{[l]}$ and $dim(Z^{[l]})=(n^{[l]},m)$

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

the activation function is applied elementwise on all elements of the matrix

with

$$dim(W^{[l]})=(n^{[l]},n^{[l-1]})$$

$$dim(B^{[l]})=(n^{[l]},m)$$

For the somewhat special case $l=1$ it is useful to remember:

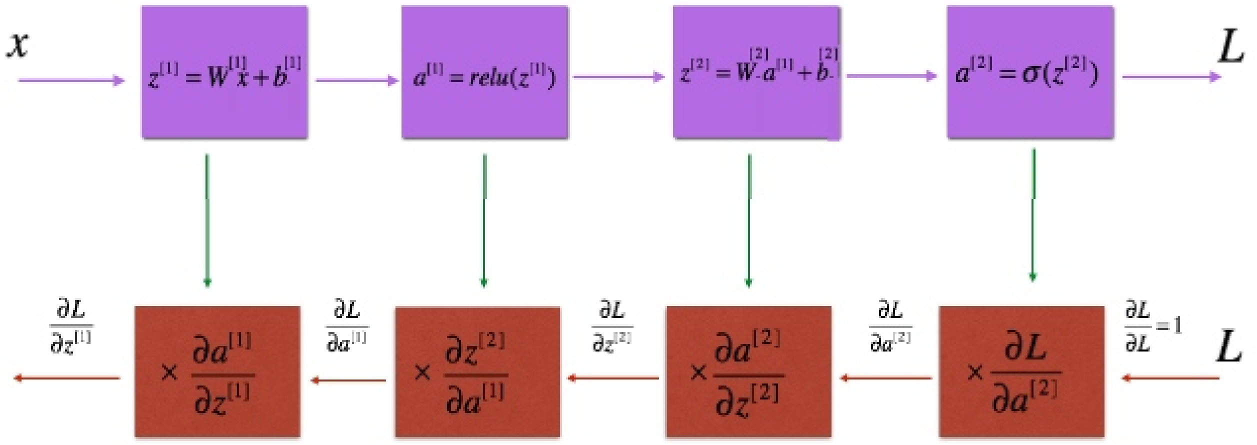
$$A^{[0]}=X=[\vec{x}^{(1)},\vec{x}^{(2)}\ldots,\vec{x}^{(m)}] \quad dim(X)=(n^{[0]},m)=(n,m)$$

Code implementation in Python:

Instead of explicitly expanding the vector $\vec{b}^{[l]}$ with $dim(\vec{b}^{[l]})=(n^{[l]},1)$ to the matrix $B^{[l]}$ we just use Pythons broadcasting to add a vector to a matrix like this:

```
Z1 = np.matmul(W1, X) + b1
```

5.3.b. BWD propagation


x
$z^{[1]}=W^{[1]}x+b^{[1]}$
$a^{[1]}=relu(z^{[1]})$
$z^{[2]}=W^{[2]}a^{[1]}+b^{[2]}$
$a^{[2]}=\sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times\frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times\frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times\frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times\frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L}=1$
L

r:

NOTE: In the figure above L is used to denote both, the LOSS and the number of layers. The big letters to the right mean the LOSS as well as the numerator of the partial derivatives!

Noting that in general, by the chain rule:

$$\frac{\partial J}{\partial Z^{[l]}} := dZ^{[l]} = \frac{\partial J}{\partial A^{[l]}} * \frac{\partial A^{[l]}(Z^{[l]})}{\partial Z^{[l]}}$$ and

$A^{[l]}(Z^{[l]}) = g^{[l]}(Z^{[l]})$ is the activation function of layer l

$l=L$:

As for FWD propagation, there is a singular starting point after which we can generalize the calculations:

This singularity is at $l=L$, the output layer of the NN. In this case we have:

(0) $\frac{\partial J}{\partial Z^{[L]}} = dZ^{[L]} = A^{[L]} - Y$ and $dim(dZ^{[L]}) = (n^{[L]}, m)$

in the general way Y can be a matrix, not necessarily a vector (case $n^{[l]}=1$).

Which can be easily calculated.

However in our implementation we calculate the $dZ^{[l]}$ values over explicit calculation of the $dA^{[l]}$ values when computing for layer l

$dZ^{[l]}=dA^{[l]}*\frac{\partial\,\sigma(Z^{[l]})}{\partial\,Z^{[l]}}$ generally so that:

$$dA^{[L]}=\frac{dZ^{[L]}}{\frac{\partial\,\sigma(Z^{[L]})}{\partial\,Z^{[L]}}}$$

and for the sigmoid function: $\frac{\partial\,\sigma(Z^{[l]})}{\partial\,Z^{[l]}}=A^{[l]}(1-A^{[l]})$

(i) $dA^{[L]}=\frac{A^{[L]}-Y}{A^{[L]}(1-A^{[L]})}=\frac{-Y}{A^{[L]}}+\frac{1-Y}{1-A^{[L]}}$ and $dim(dA^{[L]})=(n^{[L]},m)$

with $n^{[l]}=1$ we have that $dA^{[L]}$ is a row vector of the values $\frac{-y^{(i)}}{a^{[L](i)}}+\frac{1-y^{(i)}}{1-a^{[L](i)}}$

where $a^{[L](i)}=\sigma(z^{(i)})$ and $1\leq i\leq m$

Gradient for layer $1\leq l\leq L-1$:

From this point on, we can have generalized equations for the layer $1\leq l\leq L-1$

(ii) $dZ^{[l]}=(W^{[l+1]})^{T}(dZ^{[l+1]})*\frac{\partial\,g^{[l]}(Z^{[l]})}{\partial\,Z^{[l]}}$ equation with terms across layers!

$dZ^{[l]}=dA^{[l]}*\frac{\partial\,g^{[l]}(Z^{[l]})}{\partial\,Z^{[l]}}$ with $dA^{[l]}=\frac{\partial\,J}{\partial\,A^{[l]}}=(W^{[l+1]})^{T}(dZ^{[l+1]})$

Important: $A*B$ denotes an **element-wise multiplication** of the matrices A and B

$$dim(dZ^{[l]})=dim(Z^{[l]})=(n^{[l]},n^{[l+1]})(n^{[l+1]},m)=(n^{[l]},m)$$

$$dim(\frac{\partial g^{[l]}}{\partial Z^{[l]}})=dim(Z^{[l]})=(n^{[l]},m)$$

(iii) $\frac{\partial J}{\partial W^{[l]}}=dW^{[l]}=\frac{1}{m}(dZ^{[l]})(A^{[l-1]})^T$ equation with terms across layers!

$$dim(dW^{[l]})=dim(W^{[l]})=(n^{[l]},m)(m,n^{[l-1]})=(n^{[l]},n^{[l-1]})$$

(iv) $\frac{\partial J}{\partial b^{[l]}}=\vec{db}^{[l]}=\frac{1}{m}\sum_{i=1}^{m}dZ^{[l]}$ No terms across layers!

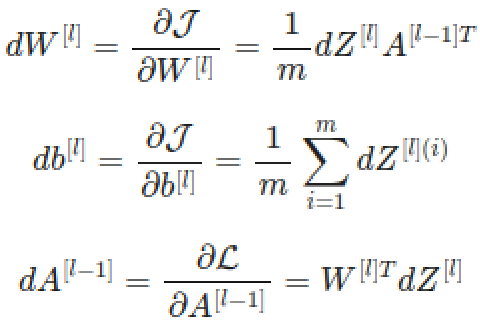
since $dim(Z^{[l]})=(n^{[l]},m)$ it follows $dim(\vec{db}^{[l]})=(n^{[l]},1)$

Implement this in Python with:

```
np.sum(dZ1, axis=1, keepdims=True)
```

Code implementation in Python:

In the code implementation we calculate and store the $dA^{[l]}$ values explicitly because for getting $dZ^{[l]}$ for a given layer we need the value $dA^{[l]}$ which depends exclusively on parameters of the following layer (see eq. (ii) above or picture below).



$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

Initializing of BWD propagation at output layer:

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 – AL))
```

Function for getting gradients for layer $1 \leq l \leq L-1$

```
def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

5.4. Parameters and Hyperparameters

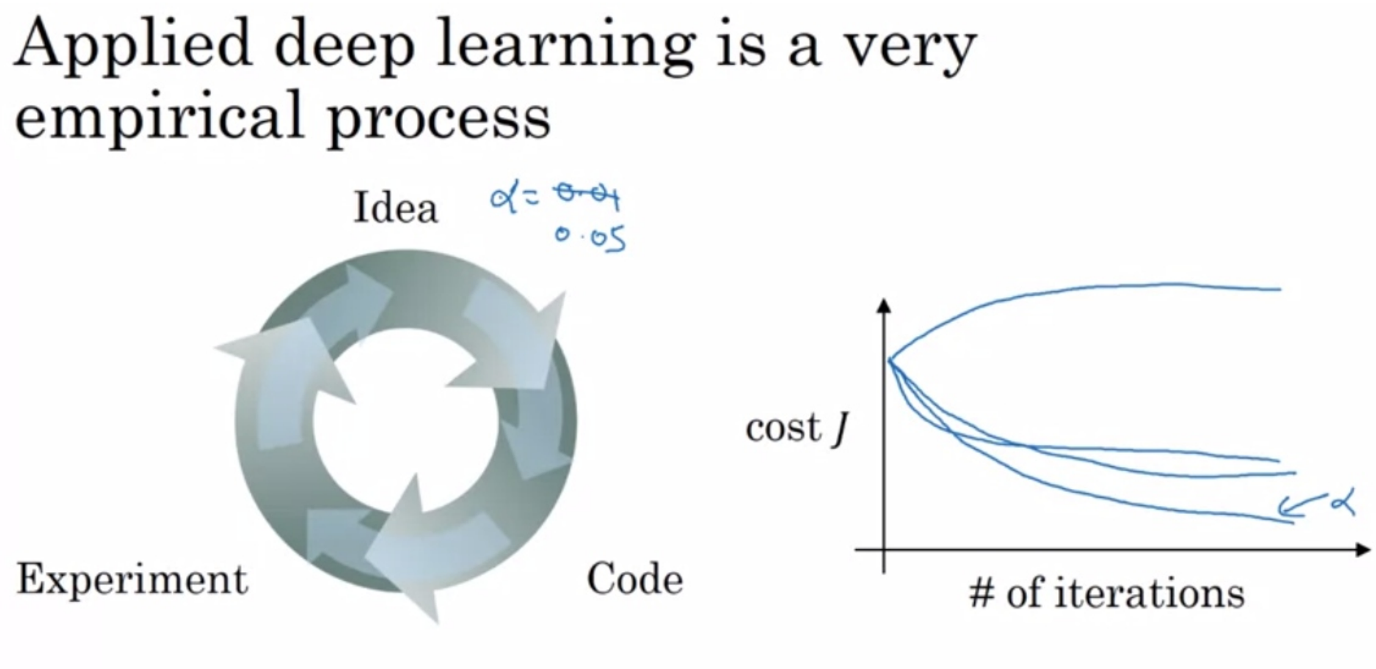
Strictly speaking the only parameters of a NN are the scalars $W^{[l]}, \vec{b}^{[l]}$ for $1 \leq l \leq L$

Hyperparameters

By now we have used:

1. α : Learning rate for parameter update in an iteration
2. Number of algorithm iterations
3. L : Number of layers (hidden layers plus output layer)
4. $n^{[l]}$ Number of units in layer l
5. $g^{[l]}$: activation function for layer l

However there are more hyperparameters.

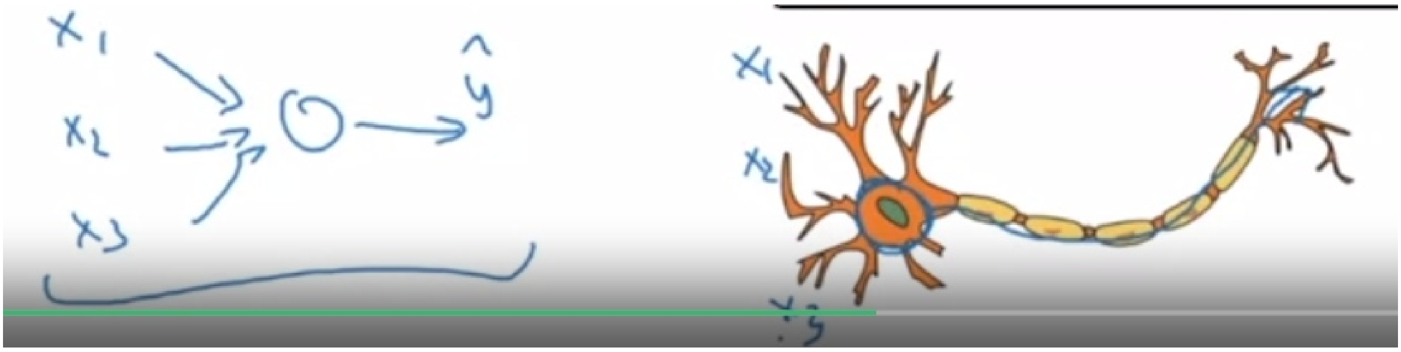


Applied deep learning is a very empirical process

5.5. Do Neural Networks Resemble Mammals' Brain?

NO

5.5.a. Poor Analogy: Biological Neuron Vs. DL-Unit

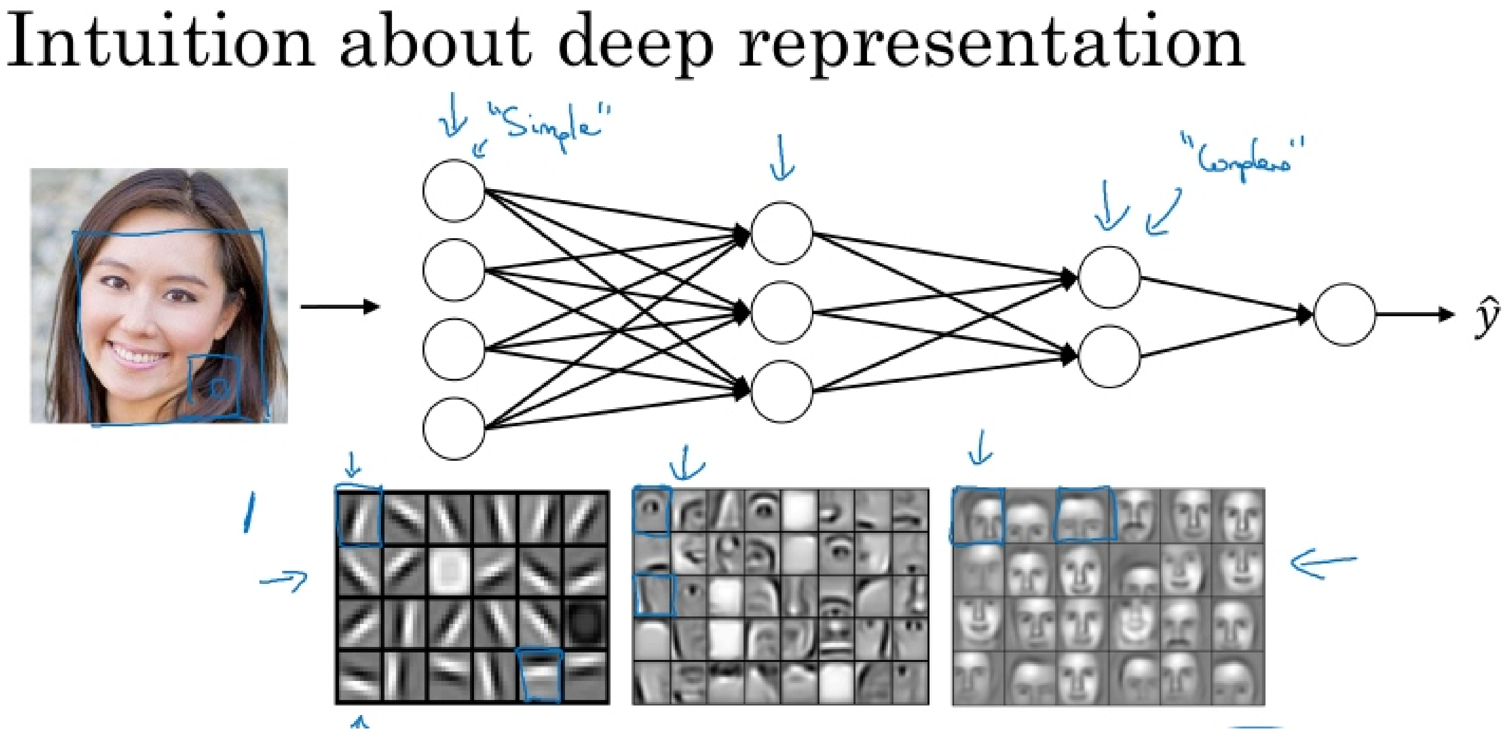


5.6. Why Neural Networks Have to be Deep?

Deeper networks have the POTENTIAL to model phenomena better than shallower models.

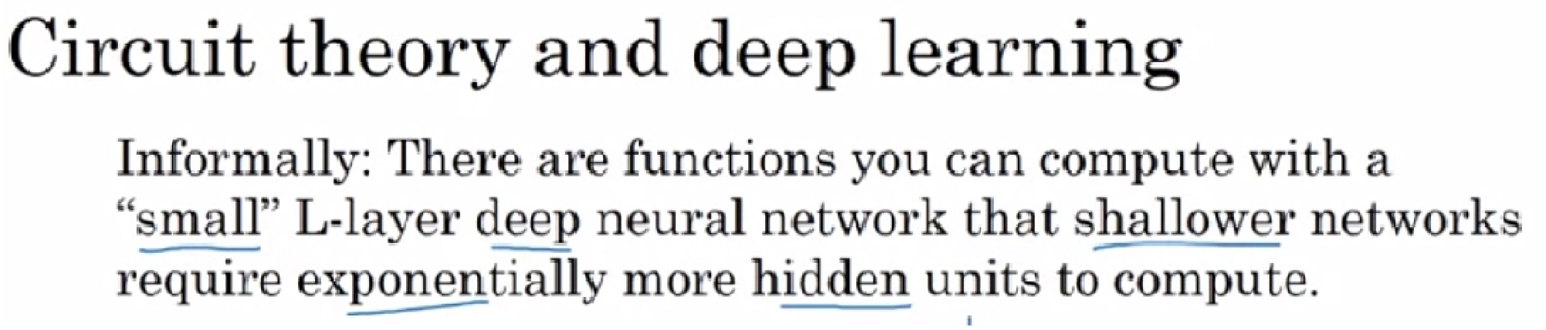
However always start with simple shallow models to see where you are standing. Do not overshoot with the depth of NN. Be reasonable!

5.6.a. How NN Work: Intuition



Intuition about deep representation

It seems reasonable to have a monotonically decreasing number of units with increasing layer number. Early layers take care of a lot of simple features while deeper layers take care of less but more complex features. Finally the output layer must be something understandable by humans.



Circuit theory and deep learning

Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.