Intro to Neural Networks

Foundations of Deep Learning

BOSTON UNIVERSITY
MACHINE INTELLIGENCE
COMMUNITY

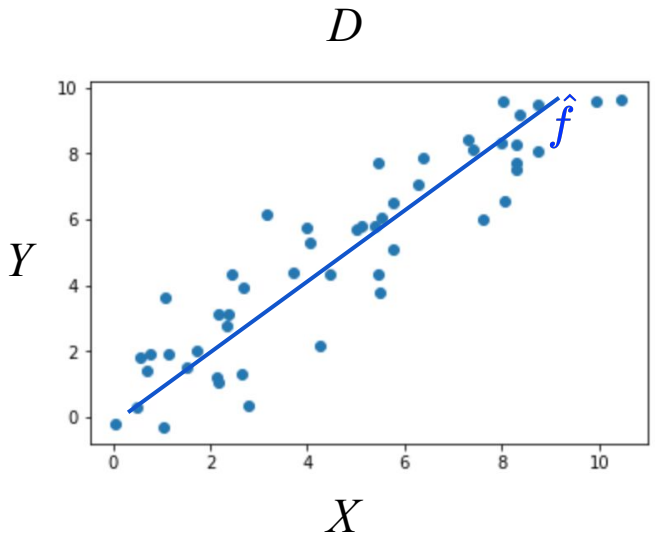
Presenter: Julius
09/11/2019

Building up to Neural Networks

Regression

What is learning?

The approximation of some unknown function f based on some data D.



$$f : X \rightarrow Y$$

$$\hat{f} = \theta_0 + \theta_1 x$$

How do we set the parameters?
How do we know what assumptions to make?

Dataset

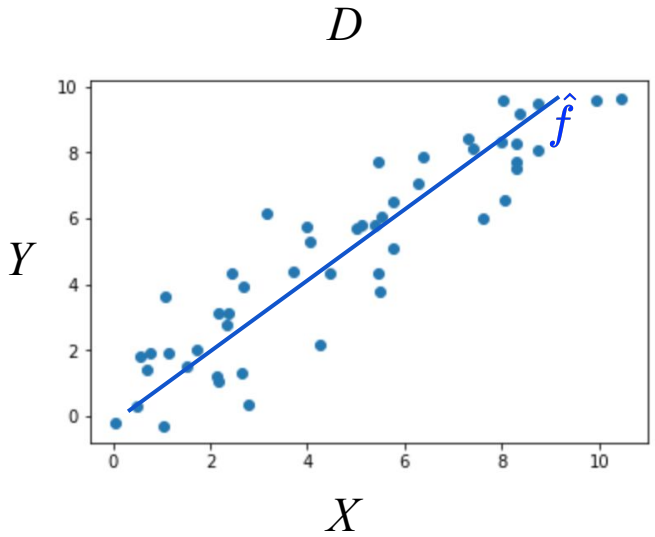
$$X_i = \begin{bmatrix} feature_1 \\ feature_2 \\ \vdots \\ feature_n \end{bmatrix}$$

$$Y_i = label_1$$

$$X = \overbrace{\begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array}}^{m_{examples}} \Bigg\} n_{features}$$

$$Y = \overbrace{\begin{array}{|c|c|c|}\hline & & \\ \hline \end{array}}^{m_{examples}}$$

Linear Regression



One dimensional input

$$y = mx + b$$

Multi dimensional input

$$y = w_1 x_1 + \cdots + w_n x_n + b$$

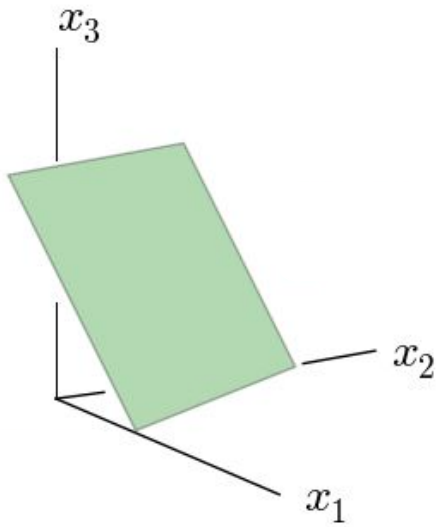
Linear Regression example

Say you want to predict house prices given some features about the house

$$x_1 = 5(bedrooms), x_2 = 3(bathrooms), x_3 = 3000(squarefeet)$$

Then the house price could be:

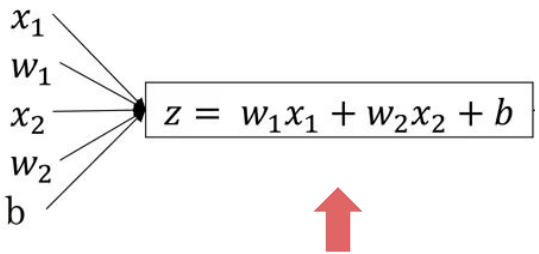
$$z = 5w_1 + 3w_2 + 3000w_3 + b$$



Logistic Regression (1)

1. **Has a linear combination / made up of weights and a bias**

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$



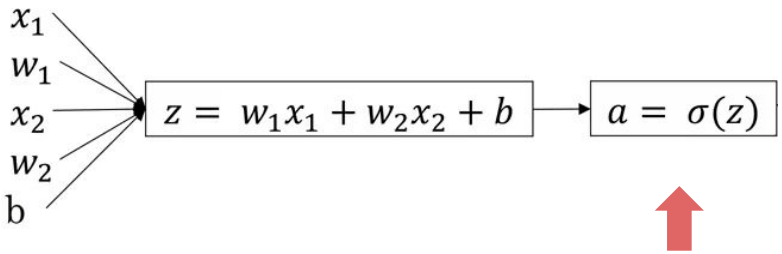
Logistic Regression (2)

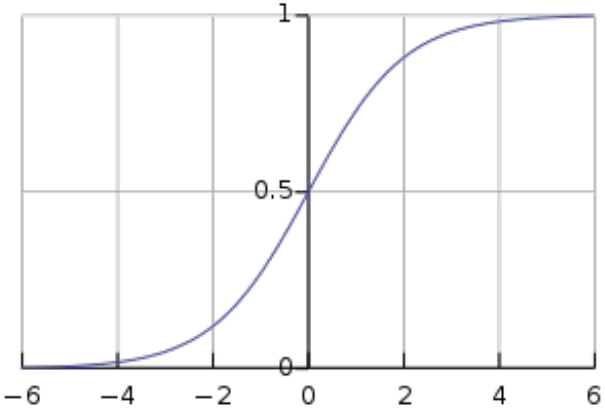
1. **Has a linear combination / made up of weights and a bias**

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

2. **Is fed through an activation function (such as a sigmoid function)**

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





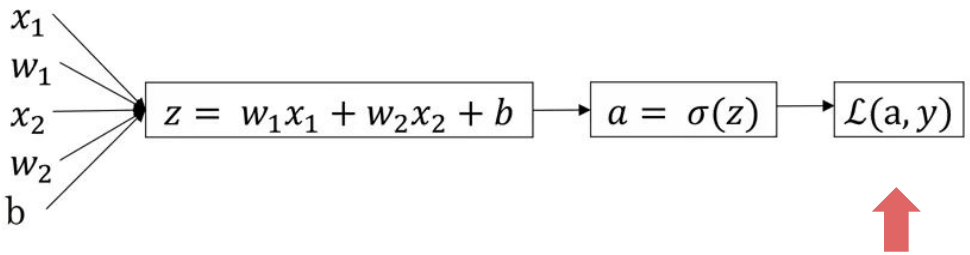
Logistic Regression (3)

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

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

3. The output is used to measure the difference from the actual data.

$$L(a, y) = y - a$$



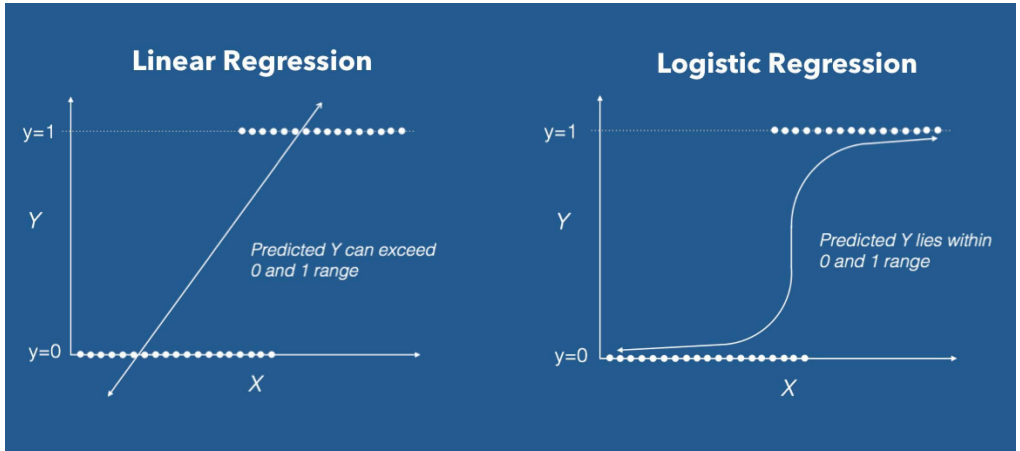
Logistic Regression (example)

Now say we want to determine if we can afford the house or not.

In this case we would use a sigmoid function to output values between 0 or 1.

This can be a metric of how affordable the house is.

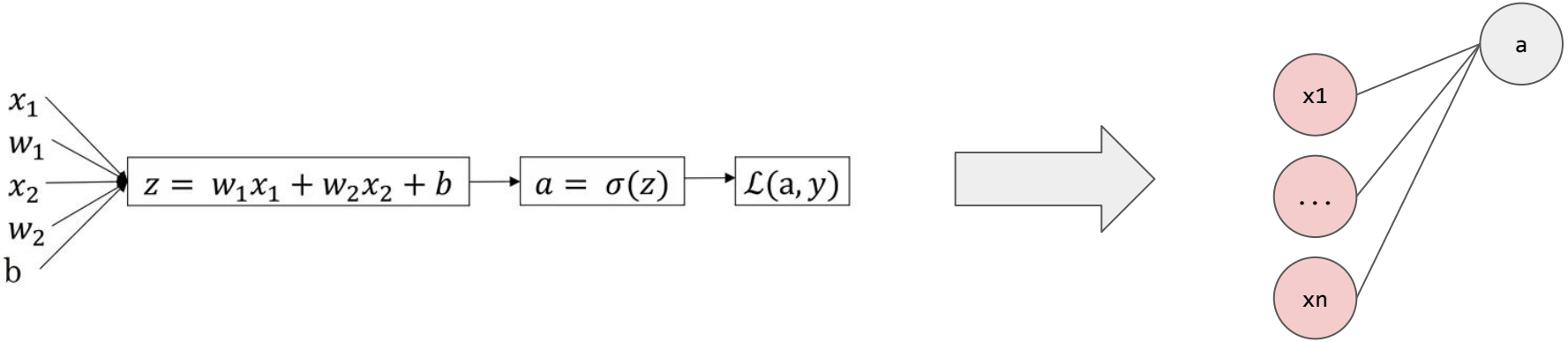
$$a = y = \sigma(z)$$



Logistic Regression to Neural Networks (1)

Single logistic regression

$$a = \sigma(w_1 x_2 + \cdots + w_n x_n + b)$$



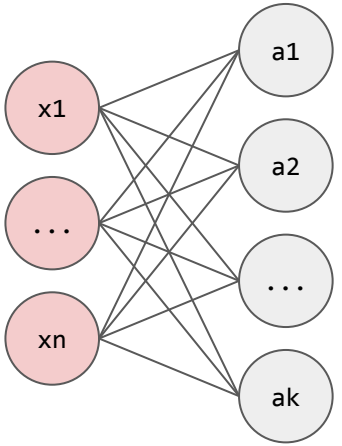
Expanded representation

Compact representation

Logistic Regression to Neural Networks (2)

Multiple logistic regressions

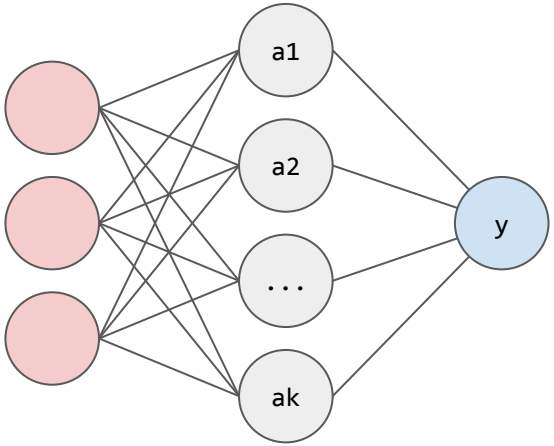
$$a_1 = \sigma(w_1^{(1)} x_1 + \cdots + w_n^{(1)} x_n + b^{(1)})$$
$$a_2 = \sigma(w_1^{(2)} x_1 + \cdots + w_n^{(2)} x_n + b^{(2)})$$
$$\vdots$$
$$a_k = \sigma(w_1^{(k)} x_1 + \cdots + w_n^{(k)} x_n + b^{(k)})$$



Logistic Regression to Neural Networks (2)

Logistic regression on the outputs of other logistic regressions

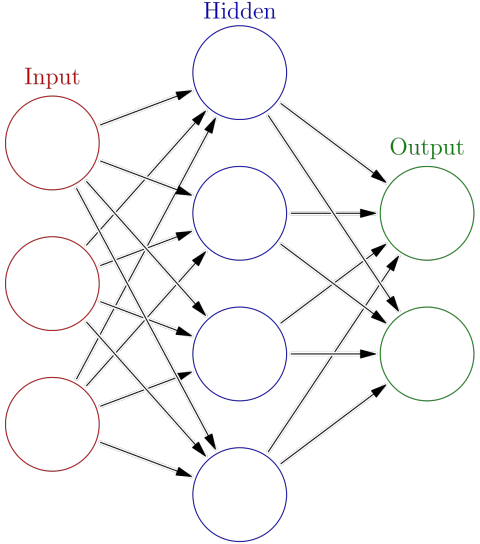
$$\hat{y} = \sigma(w_1^{(y)} a_1 + \cdots + w_k^{(y)} a_k + b^{(y)})$$



Shallow Neural Networks

The foundation of new AI

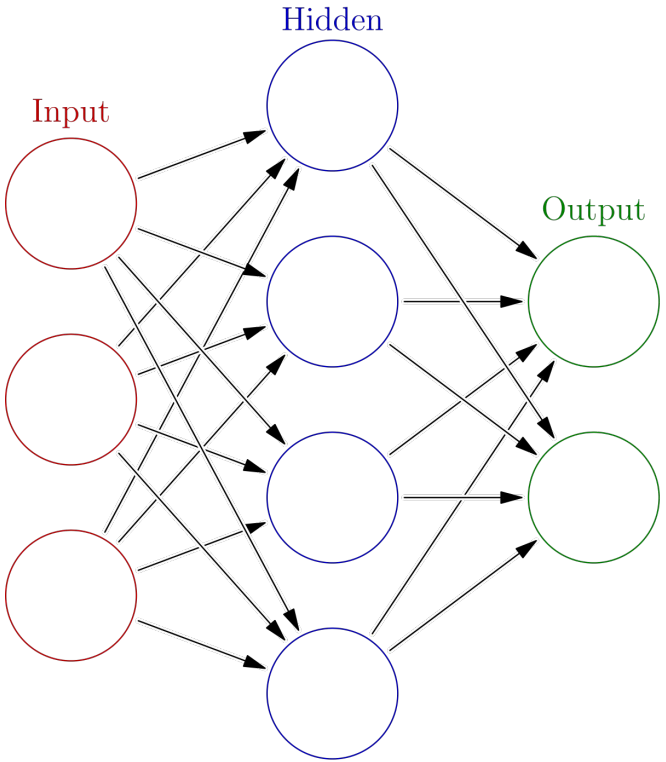
Neural Network



1. A neural network is a bunch of activations in layers
    a. The middle layers are called a hidden layer
2. Each activation takes in inputs from the previous layer
    a. The hidden layer is built from the input layer
    b. The output layer is built from the hidden layer
3. Adding more layers means we can learn more complicated functions

Activations (Neuron)

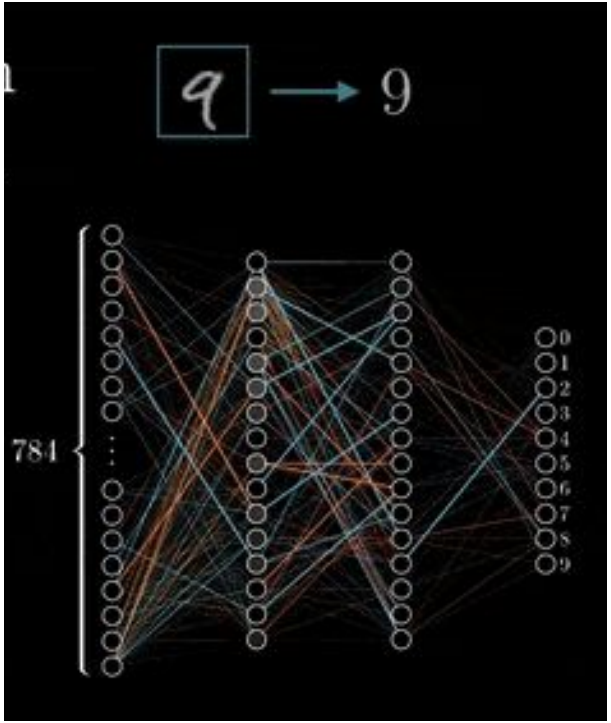
1. **Affine function**
   a. $z = w_1 a_1^{[l-1]} + \cdots + w_n a_n^{[l-1]} + b$
   b. Gathers and weights features
2. **Activation**
   a. $a = g(z)$
      i. $g(x)$ is an activation function
   b. Acts as a significance threshold



Where are the weights in this diagram?

Three steps to training a neural network

1. Forward propagation
    a. Push example through the network to get a predicted output
2. Compute the cost
    a. Calculate difference between predicted output and actual data
3. Backward propagation
    a. Push back the derivative of the error and apply to each weight, such that next time it will result in a lower error



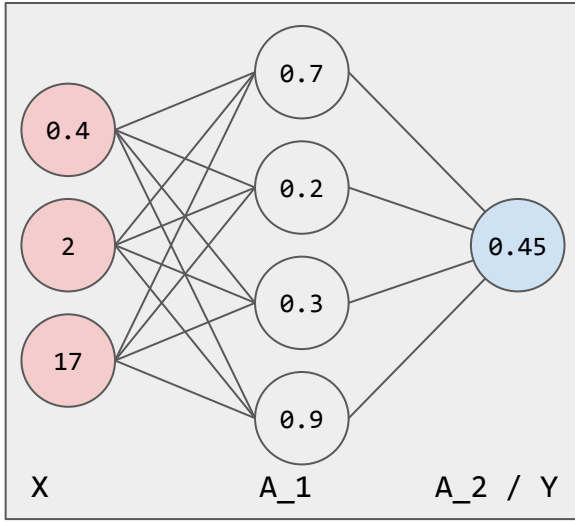
Shallow Neural Networks (example)

Say you want to predict if it is worth going to class or not

$$X = \begin{bmatrix} \text{difficulty of material} \\ \text{number of assignments due} \\ \text{days until midterm} \end{bmatrix} \qquad Y = \begin{bmatrix} \text{probability of going to class} \end{bmatrix}$$

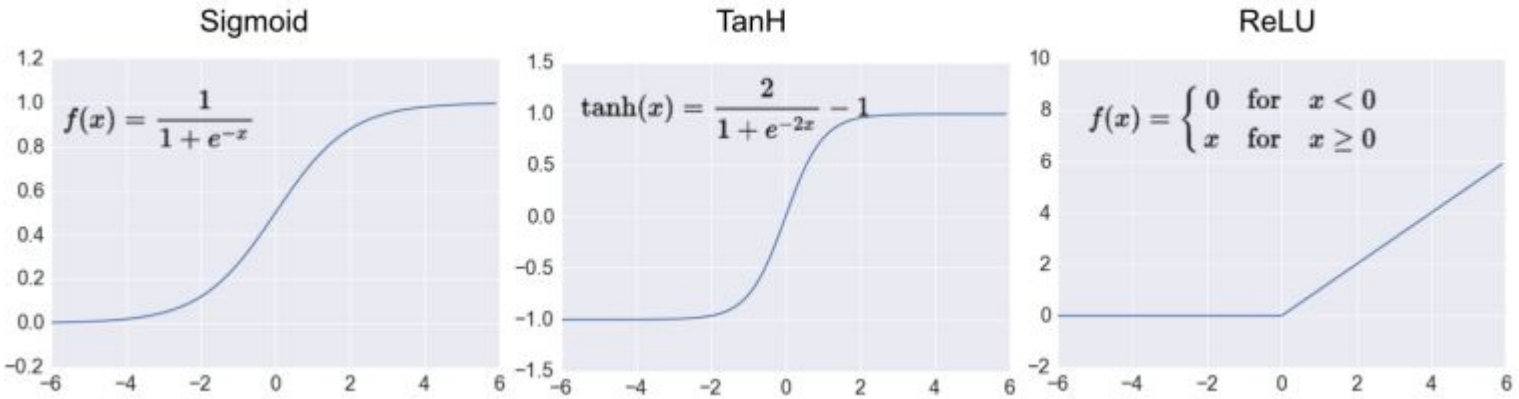
$$a_i^{[1]} = g(z) = g(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

$$a^{[2]} = g(w_1 a_1^{[1]} + w_2 a_2^{[1]} + w_3 a_3^{[1]} + w_4 a_4^{[1]} + b)$$



Activation Functions

1. **Sigmoid: output is between 0,1**
2. **Tanh: output is between -1,1**
3. **ReLu: output is positive real numbers**

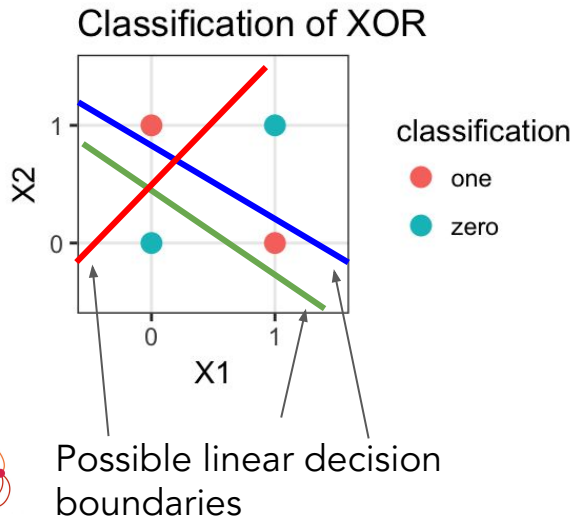


Why do you need nonlinear activation functions?

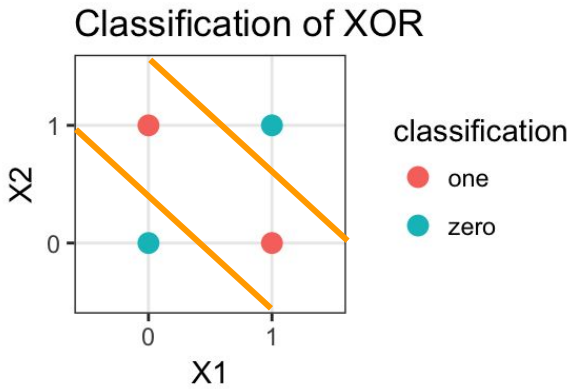
XOR Problem

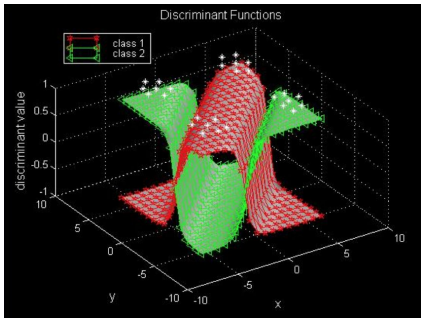
There is no way to correctly classify all inputs with a linear decision boundary

Linear Classifier



Two Layer Neural Network w/ Sigmoid Activation





Linear Activation Functions

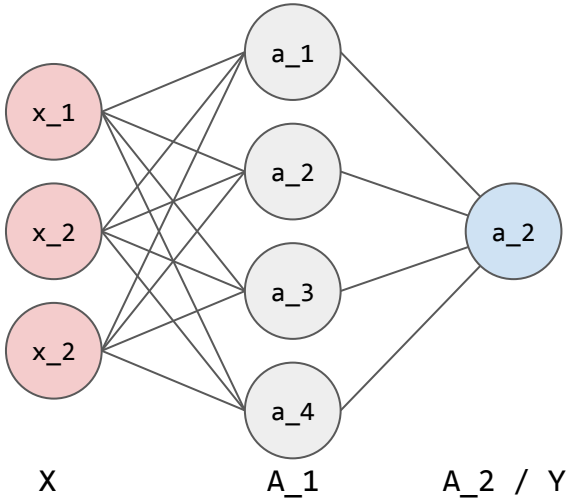
If we removed the activation function from our model that can be called a linear activation function.

$$a_1 = w_1 x + b_1 \qquad a_2 = w_2 a_1 + b_2$$

$$\begin{aligned} a_2 &= w_2 (w_1 x + b_1) + b_2 \\ &= w_2 w_1 x + w_2 b_1 + b_2 \\ &= wx + b \end{aligned}$$

Forward Propagation

1. **We've seen neural nets with one pass through, however usually, we have multiple samples.**
2. **One way is to do this iteratively**




```
for each sample i:
     for each layer l:
          for each activation j:
```

$$a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$$

```
finally, use the final activations to compute
the cost
```

Computing the cost

In order to train our neural network, we need some way to tell us how far off its estimate was from the actual value.

We define the cost function, $J(\hat{y}, y)$ **as the sum of losses,** $\sum_{i=0}^{m} L(\hat{y}, y)$

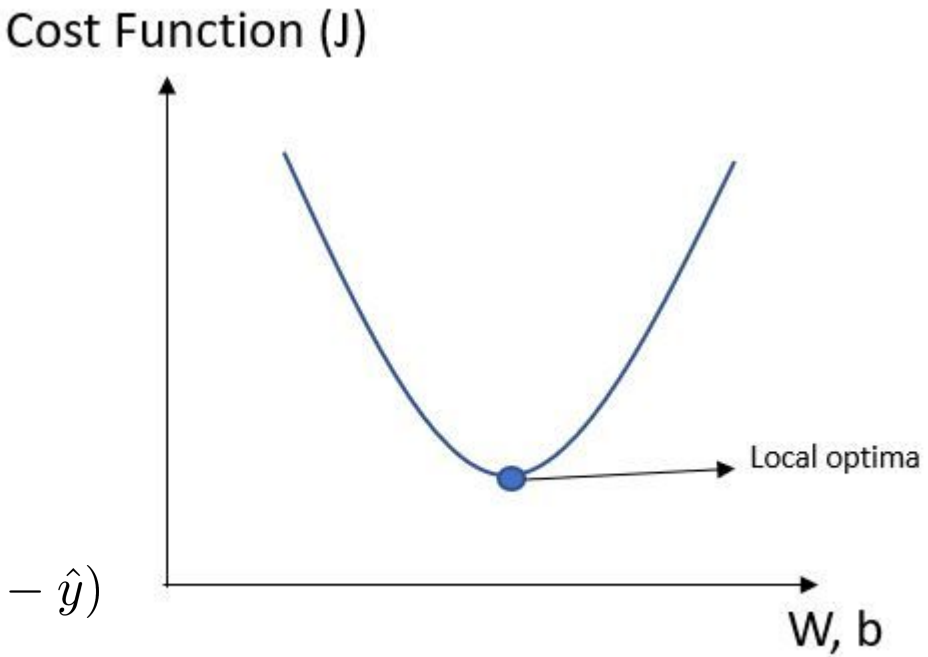
a. Loss = Error for a single training example
b. Cost = Sum of all Losses

Example cost functions

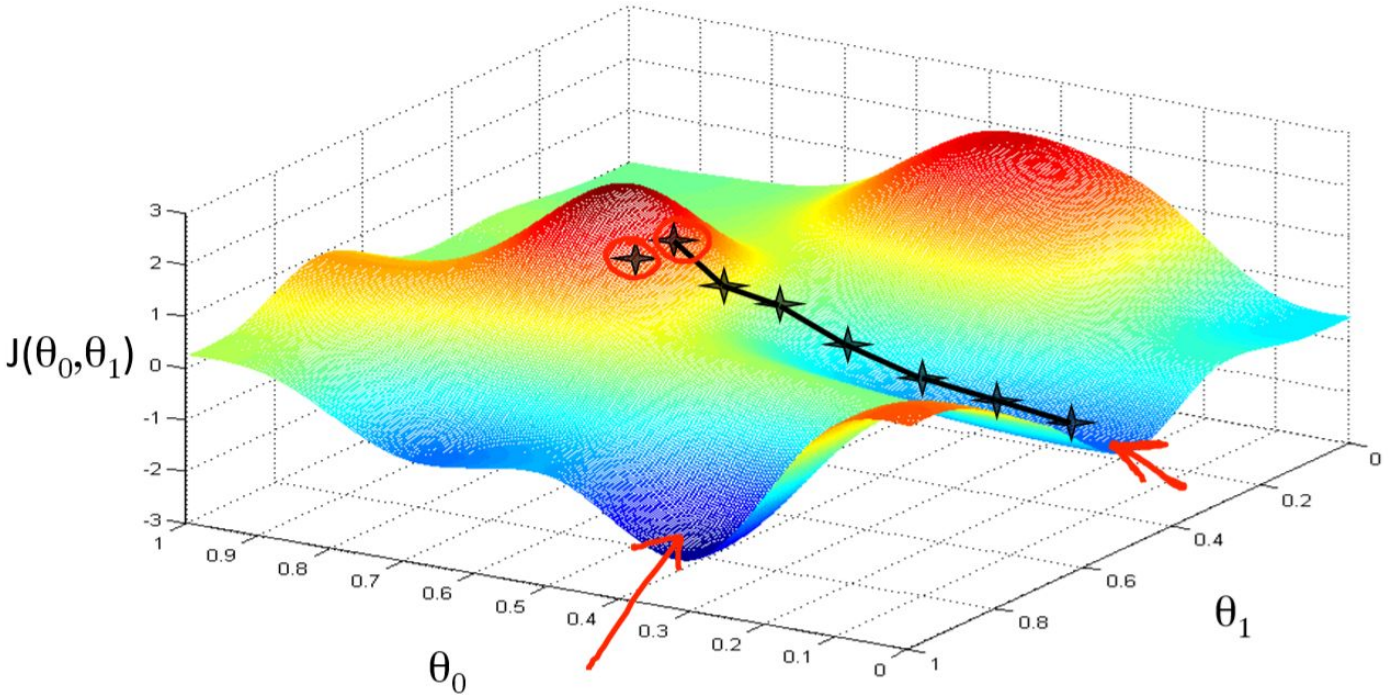
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

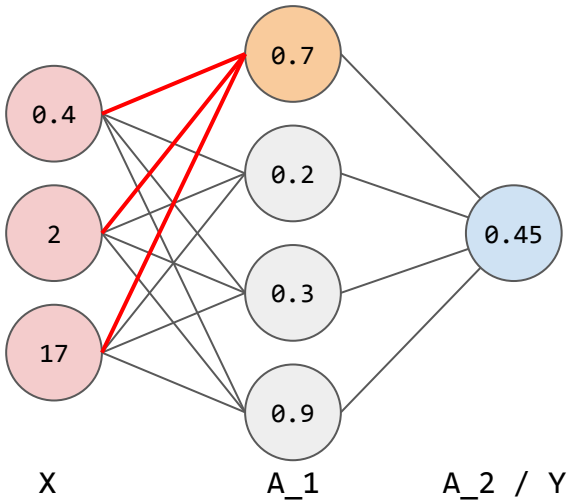
$$L(\hat{y}, y) = -y \log \hat{y} - (1-y)\log(1-\hat{y})$$



Cost function for gradient descent



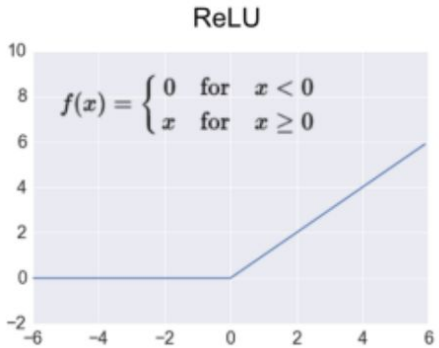
Forward Propagation (example)



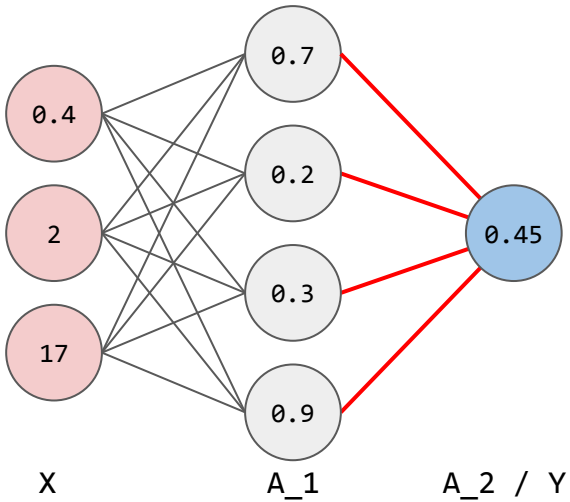
$$z_1^{[1]} = w_1^{[1]} x_1 + w_1^{[1]} x_2 + w_3^{[1]} x_3 + b$$

$$= 0.5 * 0.4 + 0.1 \times 2 + 0.0058 \times 17 + 0.2 = 0.7$$

$$a_1^{[1]} = g(0.7) = \mathrm{ReLU}(0.7) = 0.7$$



Forward Propagation (example)

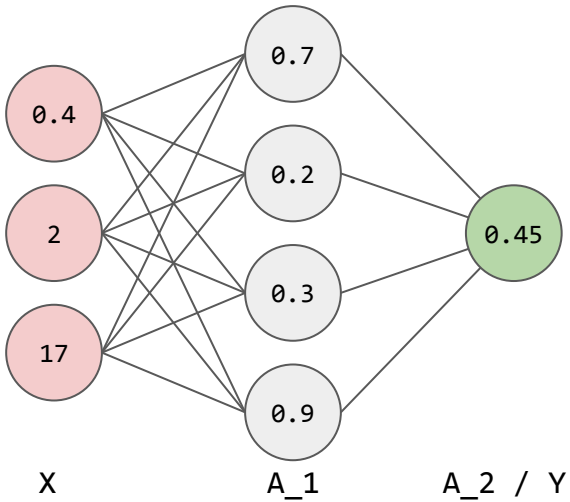


$$z_1^{[1]} = w_1^{[1]} x_1 + w_1^{[1]} x_2 + w_3^{[1]} x_3 + b$$
$$= 0.5 * 0.4 + 0.1 \times 2 + 0.0058 \times 17 + 0.2 = 0.7$$

$$a_1^{[1]} = g(0.7) = \text{ReLU}(0.7) = 0.7$$

$$z_1^{[2]} = w_1^{[2]} a_1^{[1]} + w_2^{[2]} a_2^{[1]} + w_3^{[2]} a_3^{[1]} + w_4^{[2]} a_4^{[1]} + b$$

Forward Propagation (example)



$$z_1^{[1]} = w_1^{[1]} x_1 + w_1^{[1]} x_2 + w_3^{[1]} x_3 + b$$
$$= 0.5 * 0.4 + 0.1 \times 2 + 0.0058 \times 17 + 0.2 = 0.7$$

$$a_1^{[1]} = g(0.7) = \text{ReLU}(0.7) = 0.7$$

$$z_1^{[2]} = w_1^{[2]} a_1^{[1]} + w_2^{[2]} a_2^{[1]} + w_3^{[2]} a_3^{[1]} + w_4^{[2]} a_4^{[1]} + b$$

$$\hat{y} = a_1^{[2]} = g(z_1^{[2]}) = 0.45$$

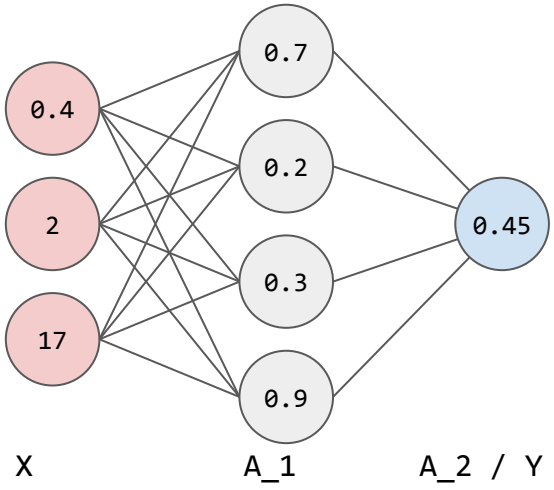
$$y = 1$$

$$L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

$$-(1) \log (0.45) - (1 - (1)) \log(1 - (0.45)) = 0.798508$$

Back to the example

Say you want to predict if it is worth going to class or not



$$\hat{y} = a_1^{[2]} = g(z_1^{[2]}) = 0.45$$

```
if y > 0.5:
    goto_class()
else:
    skip()
```

But it turns out midterm material was covered...

$$y = 1$$

Here is his mistake in numerical form

$$L(\hat{y}, y) = 0.798508$$

Training a network with Calculus

Gradient descent

Terminology for later - Parameters vs Hyperparameters

Parameters are values that are learned through training or backpropagation. (E.g. W, b)

Hyperparameters are values that are set manually without a learning method. (E.g. the learning rate, α)

Cost Function: revisited

The cost function is the sum of losses for all training examples

- combined error for all training examples.

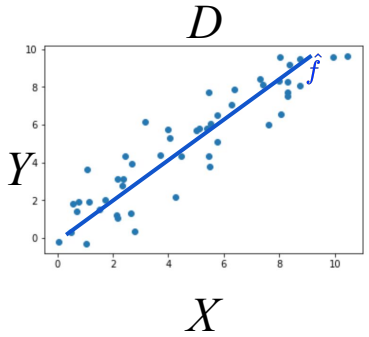
$$J(\hat{y}, y) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

This is the general form of writing the cost function

$$J(\theta) = \text{Cost Function}$$

Learning the weights

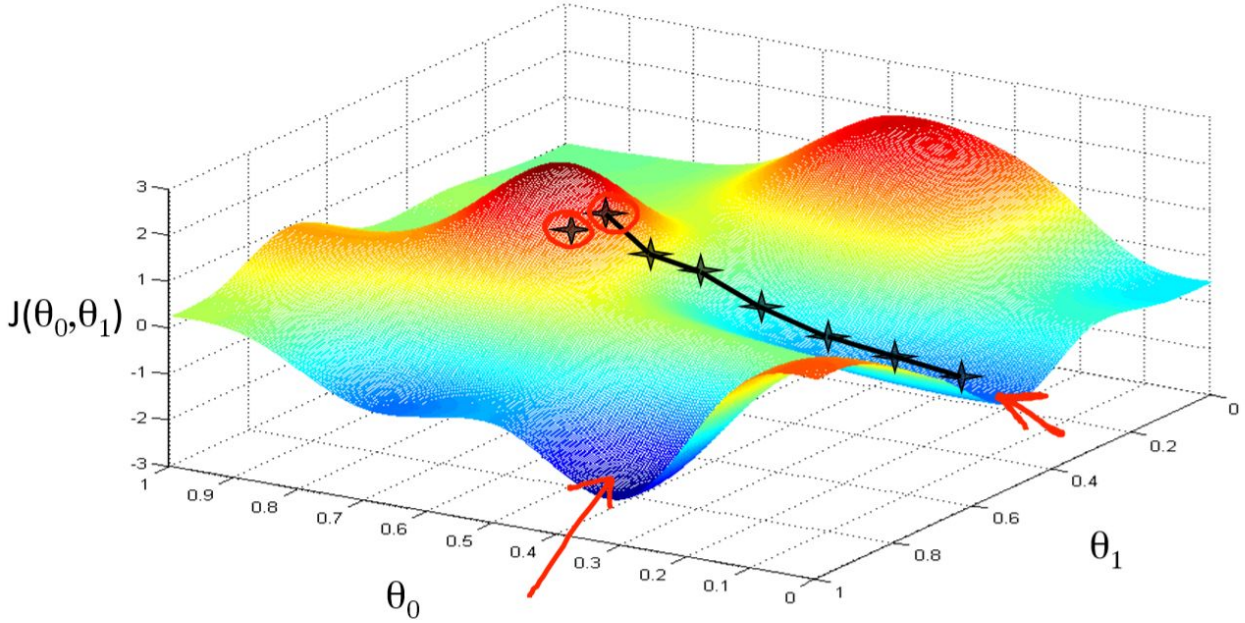
1. **Goal: we want to change all of the weights so that our predictions fit the data better**



2. **Method: the error between our predicted values and actual values tells us how much we need to change each individual weight (increase or decrease)**
   a. **By calculating the derivative from the cost function we get the numerical value of change.**
   b. **The derivative is equivalent to the slope that that point, and tells us where the error is minimized**

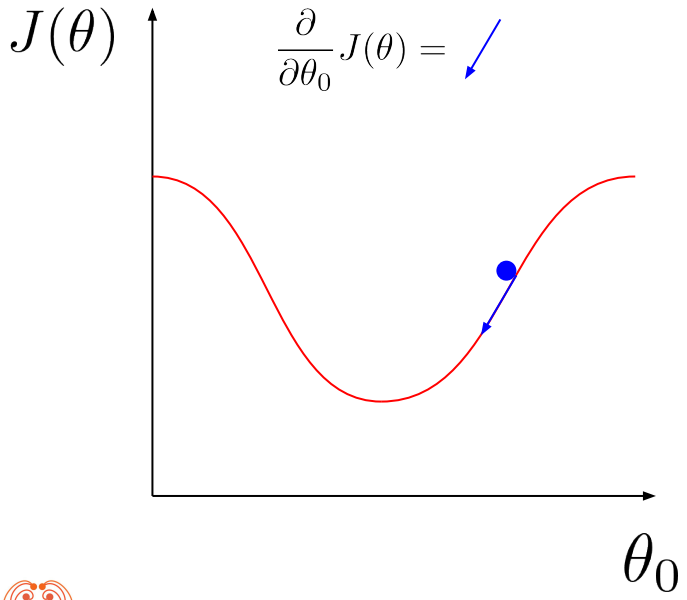
Gradient descent

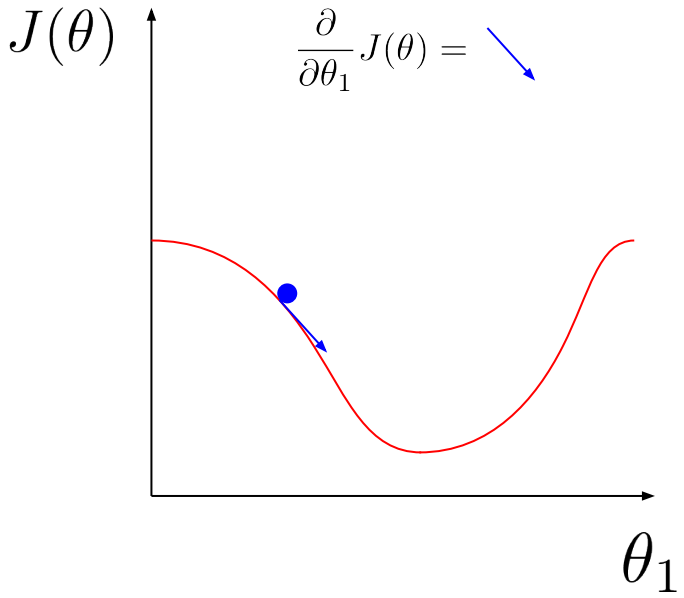
We find the direction of steepest slope, then take one step in that direction.



Gradient descent

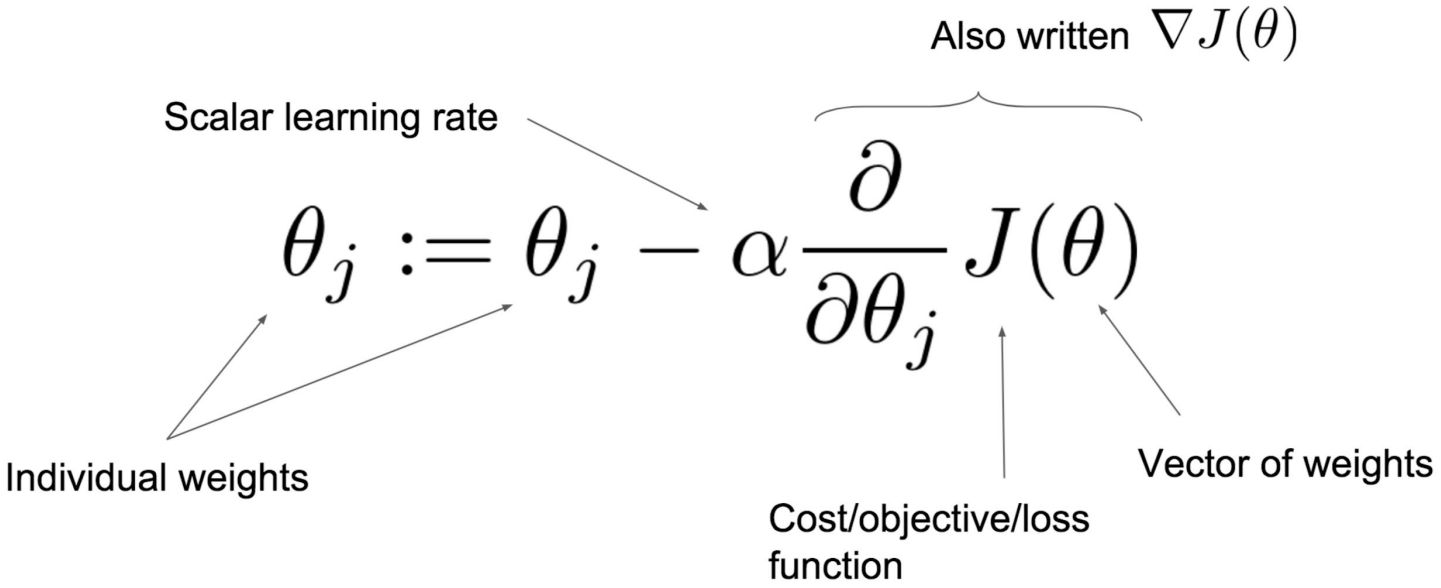
Finding the slope for each individual parameter





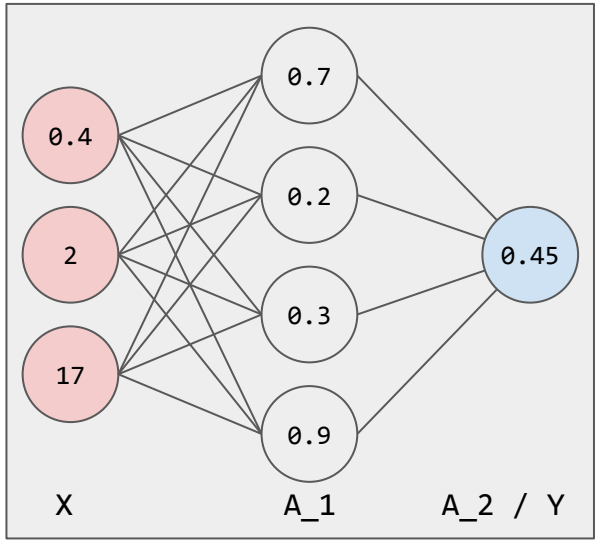
Gradient descent

To find the slope, we compute the derivative of the cost (gradient) with respect to a single parameter.



Gradient descent for Neural Networks

Goal: update the **weights** and **biases** such that the **cost function** will output a smaller value
(i.e. the difference actual and predicted values will be minimized)



Repeat for many iterations (training steps):

Compute forward pass and the cost function, **J**

$$W^{[1]} := W^{[1]} - \alpha \frac{\partial J}{\partial W^{[1]}}$$

$$b^{[1]} := b^{[1]} - \alpha \frac{\partial J}{\partial b^{[1]}}$$

$$W^{[2]} := W^{[2]} - \alpha \frac{\partial J}{\partial W^{[2]}}$$

$$b^{[2]} := b^{[2]} - \alpha \frac{\partial J}{\partial b^{[2]}}$$

Chain Rule

1. To get the direction that a parameter must change in order reduce prediction error, we use the **chain rule** from calculus.

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

$$\frac{\partial J}{\partial W^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial W^{[2]}}$$

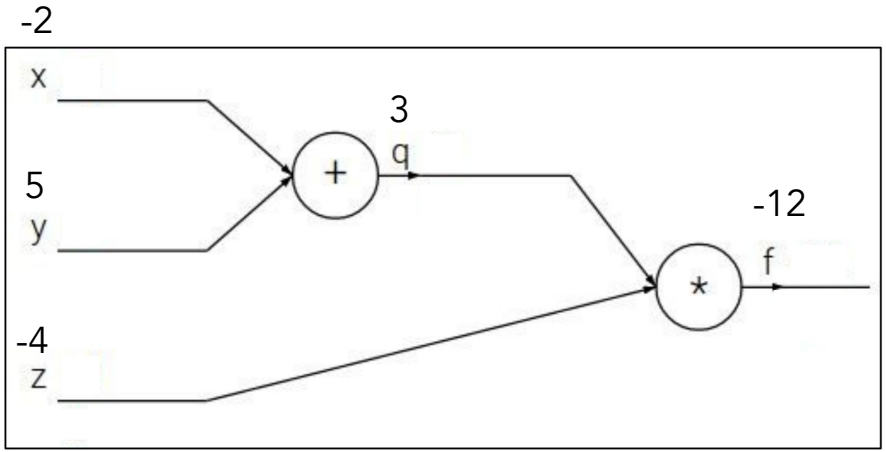
$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial W^{[1]}}$$

Chain Rule example

$$f(x, y, z) = (x + y)z$$

Eg. x = -2, y = 5, z = -4

$$q = x + y$$
$$f = qz$$



Chain Rule example

$$f(x, y, z) = (x + y)z$$

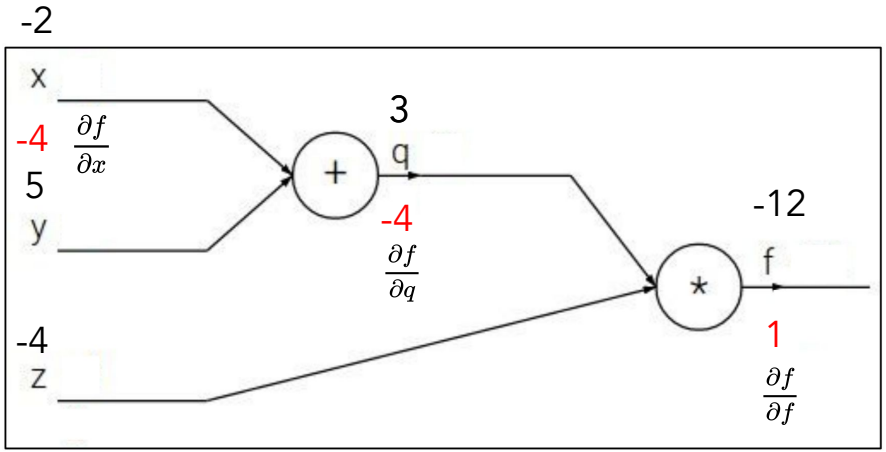
Eg. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

$$= z \times 1 = (-4) \times 1 = -4$$



$$J(\hat{y}, y) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

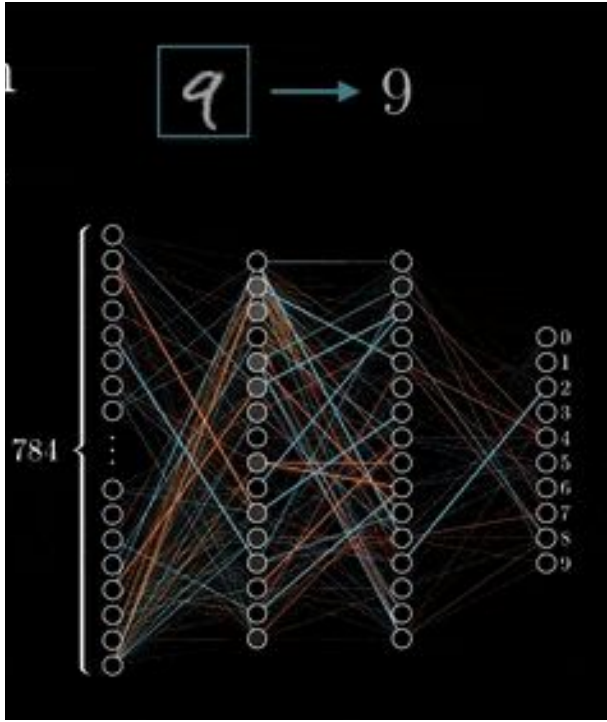
$$L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

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

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

$$a^{[1]} = g^{[1]}(z^{[1]})$$

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



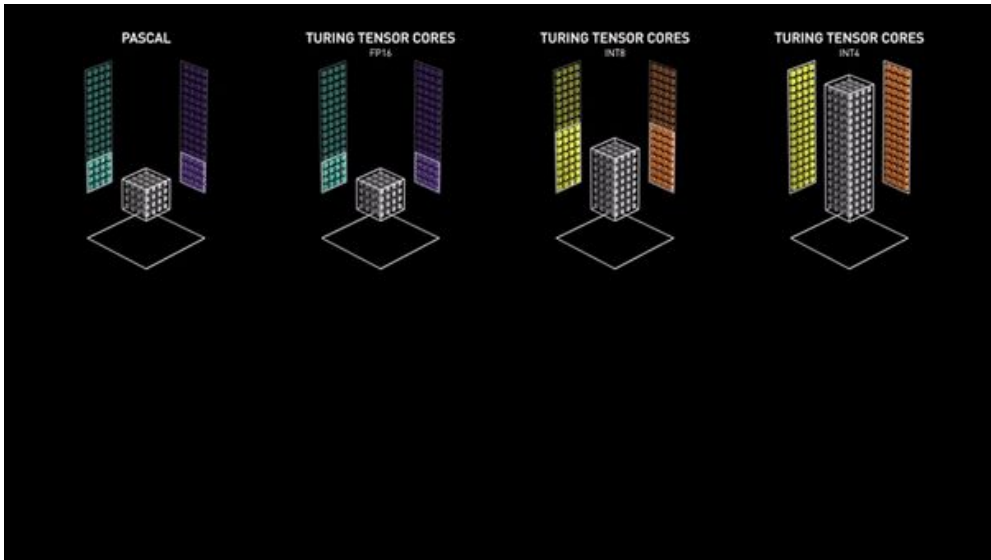
Speeding up with Linear Algebra

Vectors and matrices

Vectorization

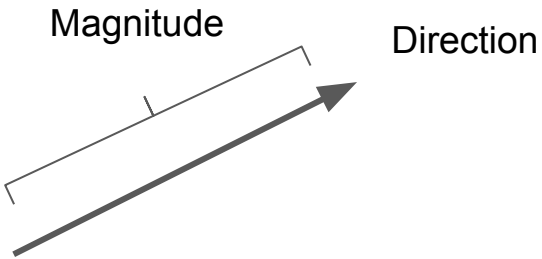
1. Recent models require massive amounts of computational power
2. Therefore obtaining the most efficiency out of your model is crucial
3. Linear algebra libraries are becoming more efficient, and neural networks can benefit from this outcome

- Matrix multiplication in Numpy
- Tensor operations in TensorFlow



Vectors and Matrices

Vector



Feature vector

$$x = \begin{bmatrix} 0.4 \\ 2 \\ 17 \end{bmatrix}$$

Matrix

Weight Matrix

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

rows

columns

Vectors and Matrices

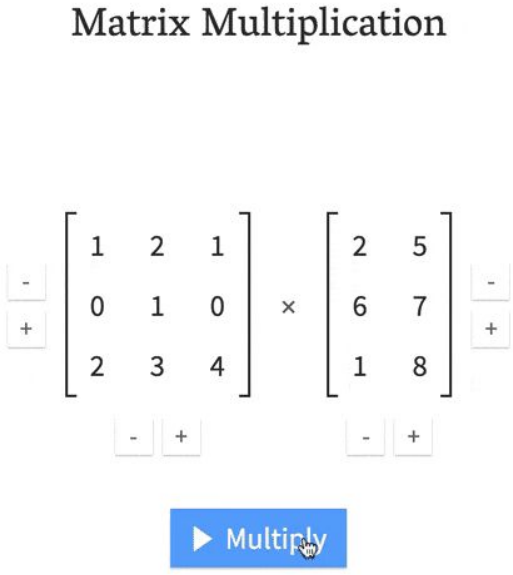
Inner product (dot product)

$$v \cdot w = v^T w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix} = 1 \times 4 + 2 \times 1 + 3 \times 3 = 15$$

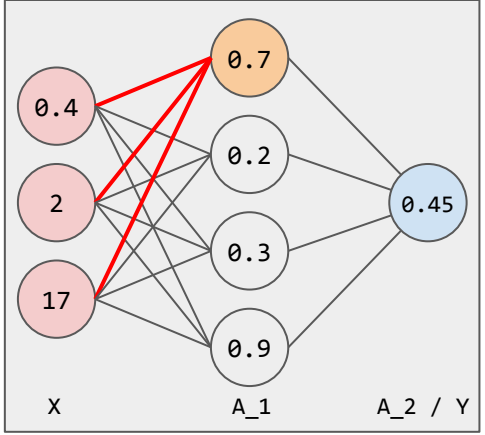
Matrix vector multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

3x3 3x1 = 3x1



Vectorizing within one activation



$$z^{[l]} = \left(w_1^{[l]} a_1^{[l-1]} + \ldots + w_n^{[l]} a_n^{[l-1]} + b^{[l]} \right)$$

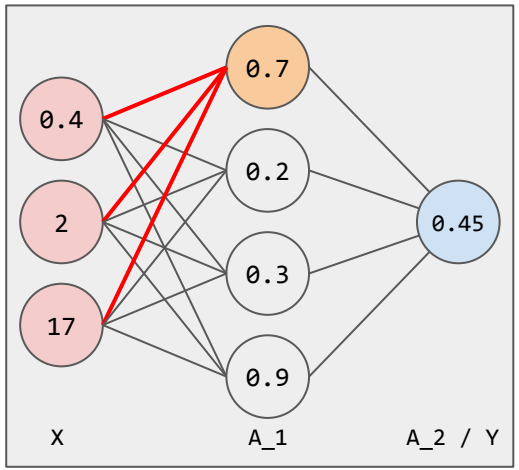
$$= \begin{bmatrix} w_1^{[l]} \\ \vdots \\ w_n^{[l]} \end{bmatrix}^T \begin{bmatrix} a_1^{[l-1]} \\ \vdots \\ a_n^{[l-1]} \end{bmatrix} + b^{[l]} = w^{[l]T} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) = g^{[l]}(w^{[l]T} a^{[l-1]} + b^{[l]})$$

Vectorizing within one activation (example)

$$z^{[l]} = \left(w_1^{[l]} a_1^{[l-1]} + \ldots + w_n^{[l]} a_n^{[l-1]} + b^{[l]}\right) = w^{[l]T} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) = g^{[l]}(w^{[l]T} a^{[l-1]} + b^{[l]})$$

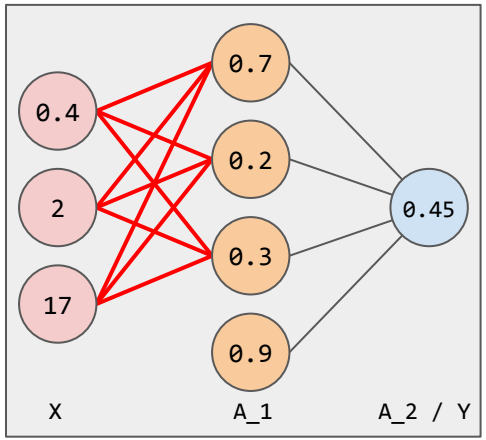


$$z_1^{[1]} = w_1^{[1]} x_1 + w_1^{[1]} x_2 + w_3^{[1]} x_3 + b$$

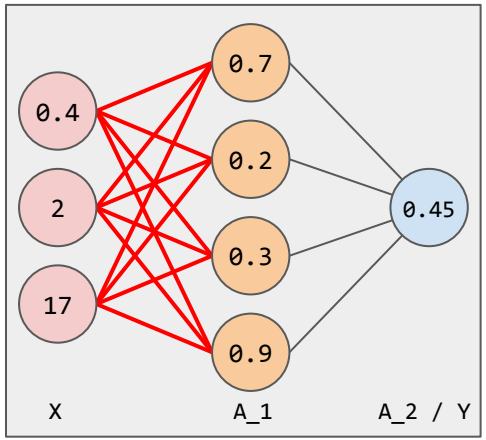
$$= \underbrace{0.5 * 0.4 + 0.1 \times 2 + 0.0058 \times 17}_{\begin{bmatrix} 0.5 \\ 0.1 \\ 0.0058 \end{bmatrix}^T \begin{bmatrix} 0.4 \\ 2 \\ 17 \end{bmatrix}} + 0.2 = 0.7$$

$$a_1^{[1]} = g(0.7) = \mathrm{ReLU}(0.7) = 0.7$$

Vectorizing across all activations

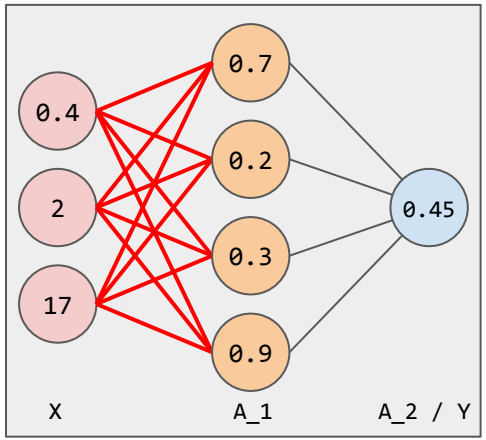


Vectorizing across all activations



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = g^{[1]}(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = g^{[1]}(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = g^{[1]}(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = g^{[1]}(z_3^{[1]})$$

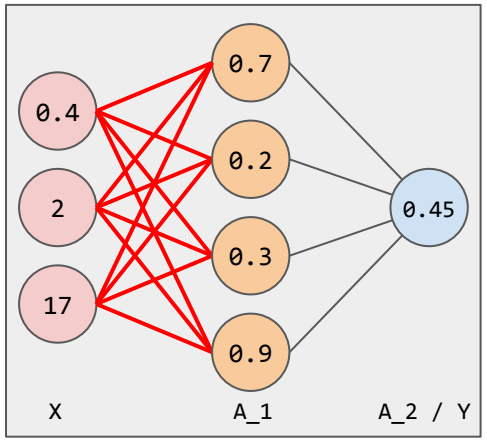
Vectorizing across all activations



$$
\begin{aligned}
z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]}, & a_1^{[1]} &= g^{[1]}(z_1^{[1]}) \\
z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]}, & a_2^{[1]} &= g^{[1]}(z_2^{[1]}) \\
z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]}, & a_3^{[1]} &= g^{[1]}(z_3^{[1]}) \\
z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]}, & a_4^{[1]} &= g^{[1]}(z_3^{[1]})
\end{aligned}
$$

$$
z^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_3^{[1]} \end{bmatrix}
$$

Vectorizing across all activations



$$z^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_3^{[1]} \end{bmatrix}$$

$$z^{[l](i)} = W^{[l]} a^{[l-1](i)} + b^{[l]}, \quad a^{[l](i)} = g^{[l]}(z^{[l](i)})$$

```
for i = 1 to m:
```

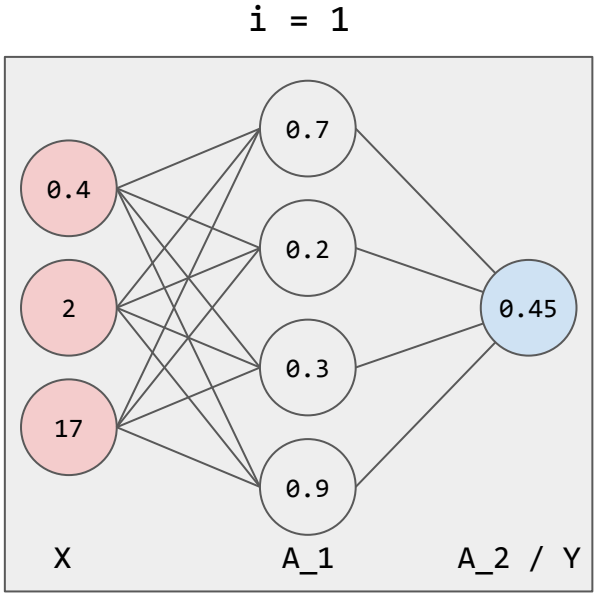
$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

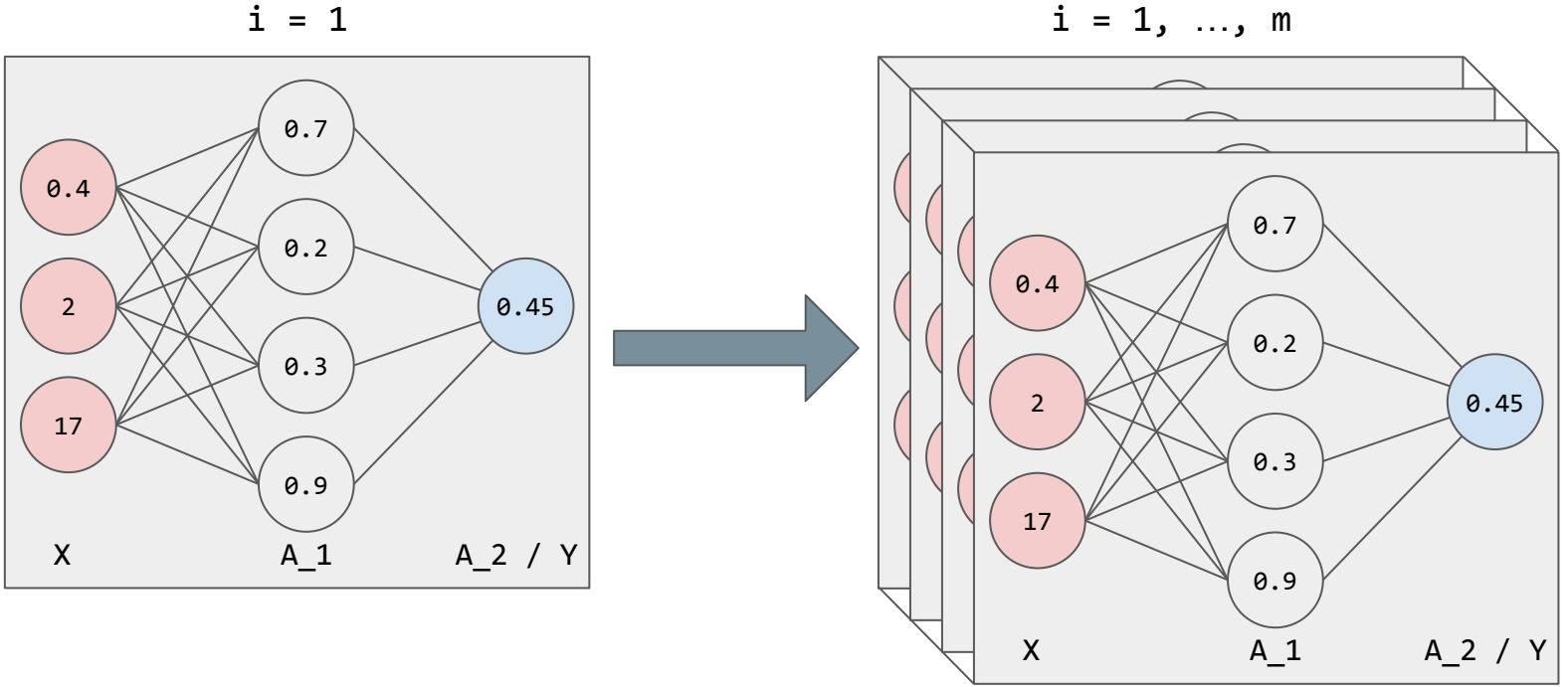
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Single Example Neural Network

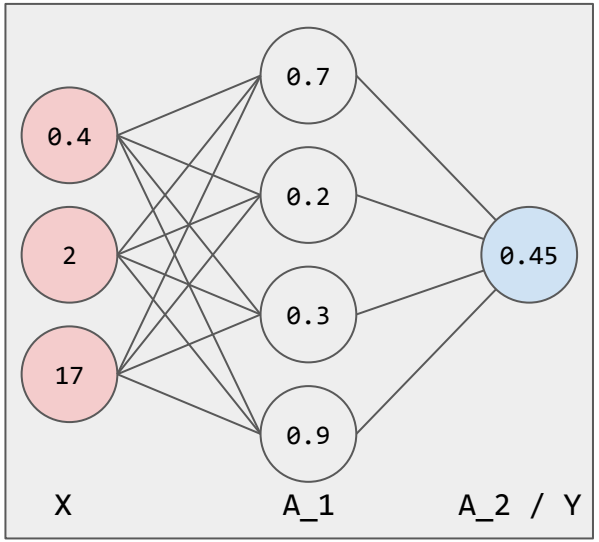


Vectorizing across multiple examples



Vectorizing across multiple examples

Algorithm: forward propagate through each example



for i = 1 to m:

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

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

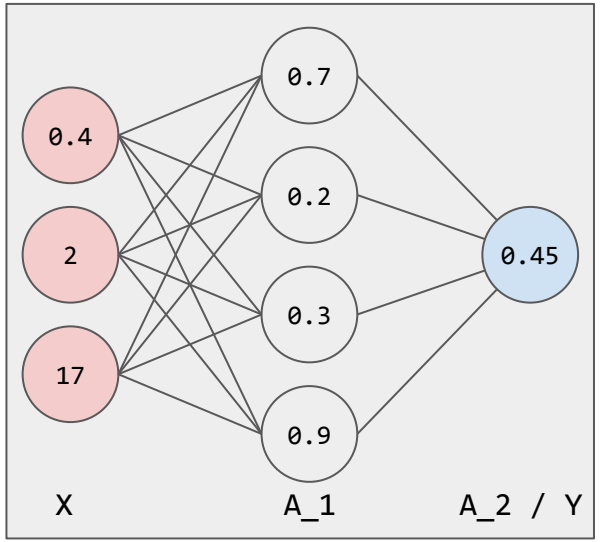
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$$J(\hat{y}, y) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Vectorizing across multiple examples

Algorithm: forward propagate through each example

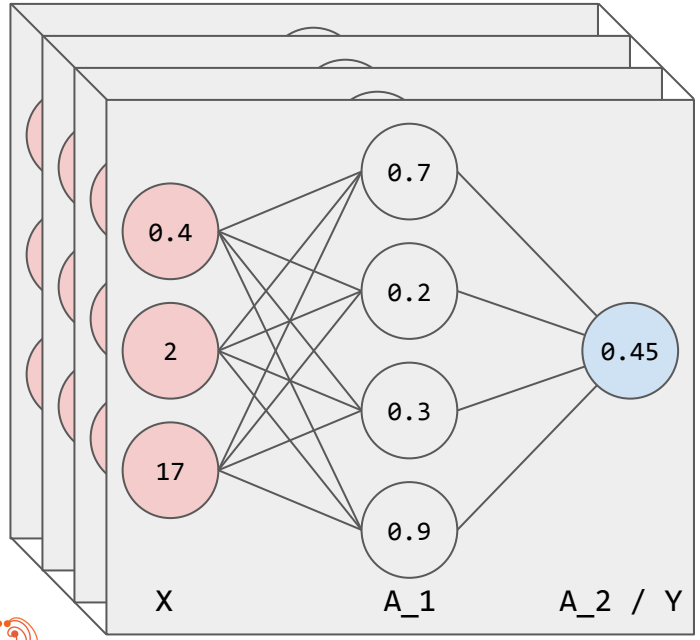


$$z^{[l](i)} = W^{[l]} a^{[l-1](i)} + b^{[l]}, \quad a^{[l](i)} = g^{[l]}(z^{[l](i)})$$

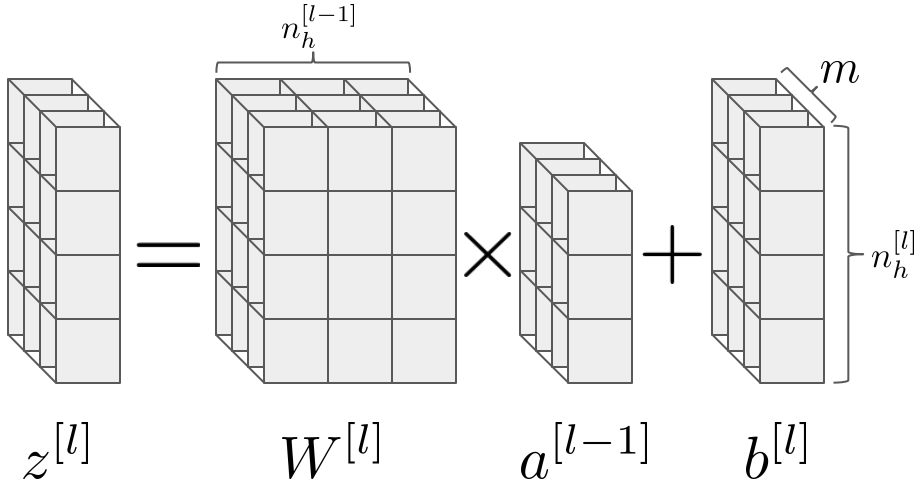
$$\underbrace{\begin{bmatrix} \\ \\ \\ \end{bmatrix}}_{z^{[l]}} = \underbrace{\overbrace{\begin{bmatrix} & & \\ & & \\ & & \\ & & \end{bmatrix}}^{n_h^{[l-1]}}}_{W^{[l]}} \times \underbrace{\begin{bmatrix} \\ \\ \end{bmatrix}}_{a^{[l-1]}} + \underbrace{\begin{bmatrix} \\ \\ \\ \end{bmatrix}}_{b^{[l]}} \; n_h^{[l]}$$

Vectorizing across multiple examples

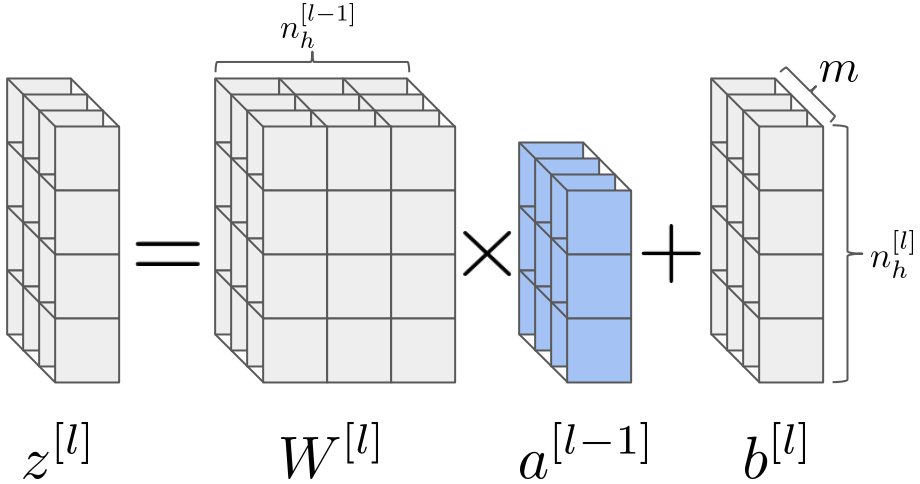
Vectorized algorithm: forward propagate through multiple examples in parallel



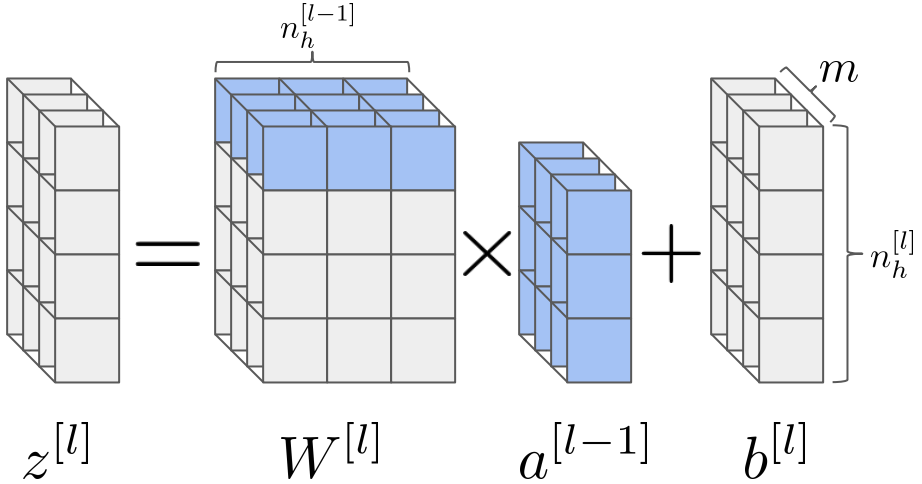
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

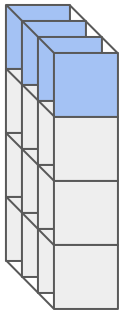


$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$



$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

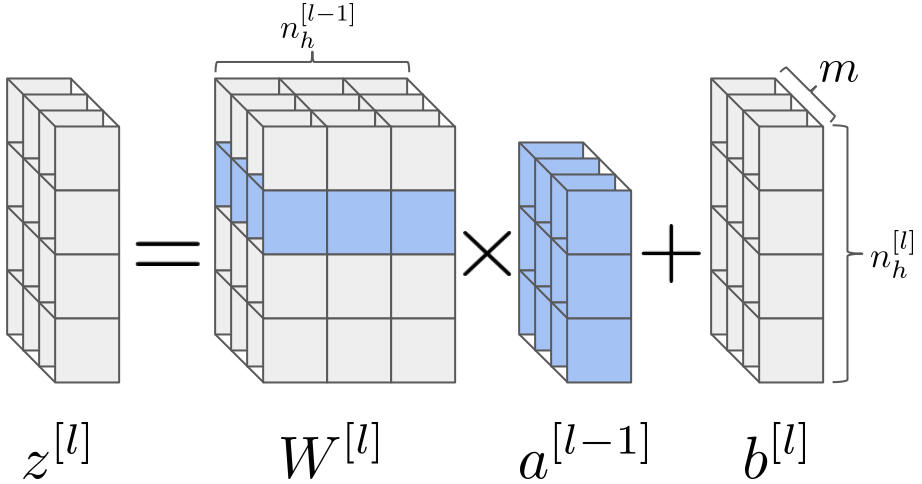


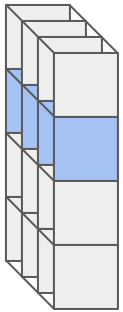


Result Tensor

Affine Function Tensor Operations

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

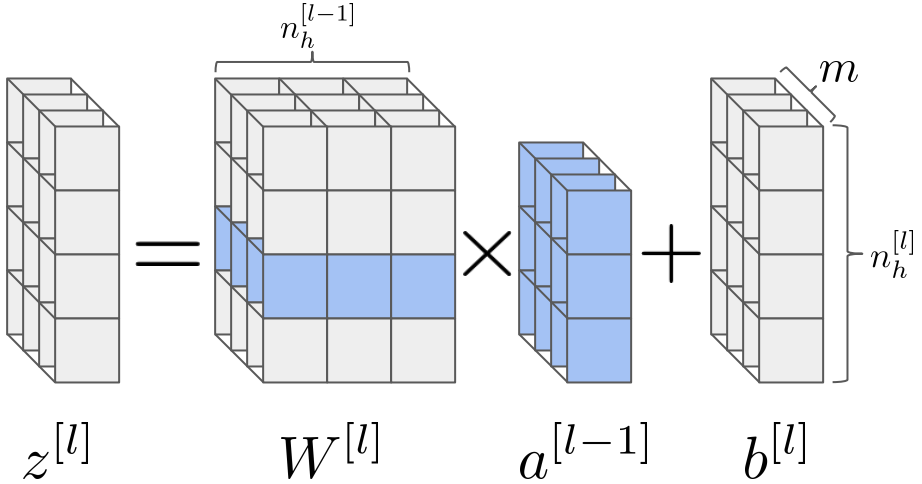


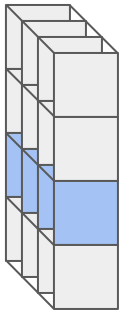


Result Tensor

Affine Function Tensor Operations

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

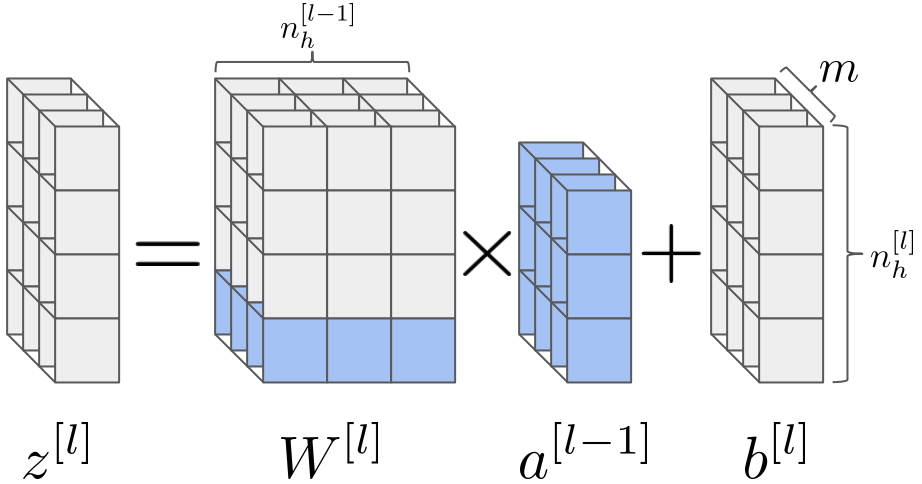


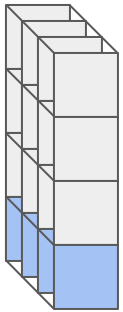


Result Tensor

Affine Function Tensor Operations

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

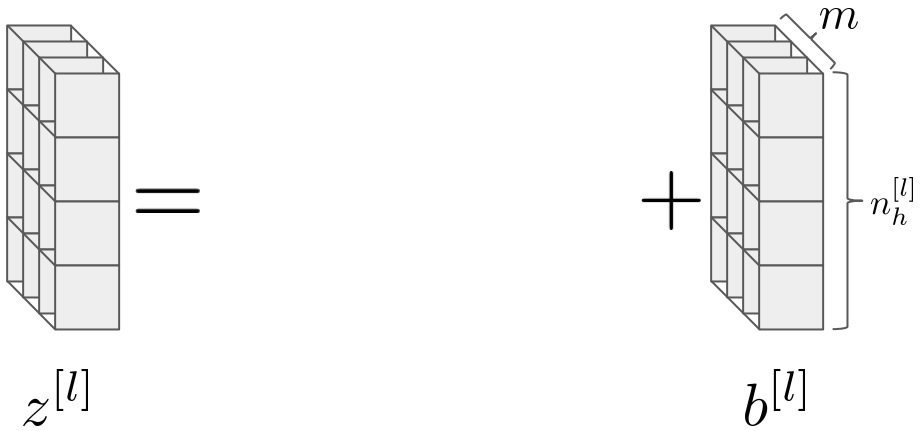


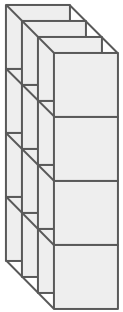


Result Tensor

Affine Function Tensor Operations

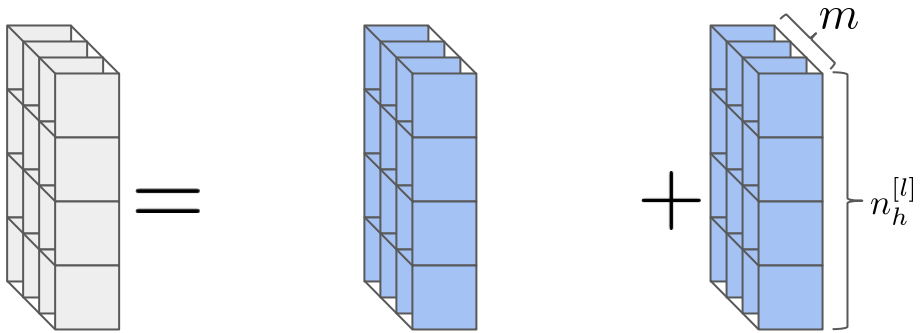
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$





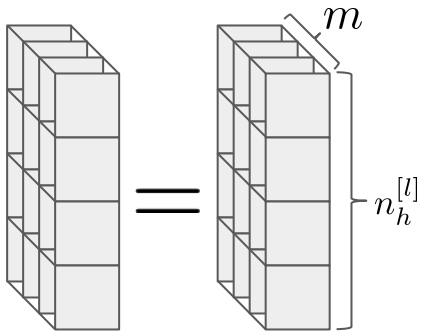
Result Tensor

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

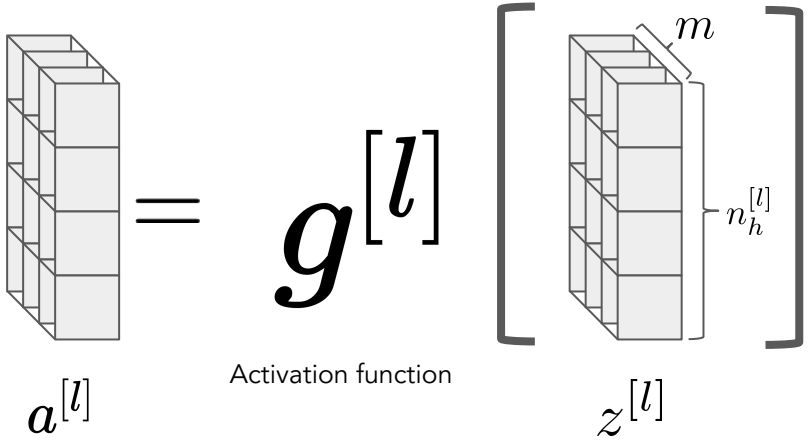


$z^{[l]}$ Result Tensor $b^{[l]}$

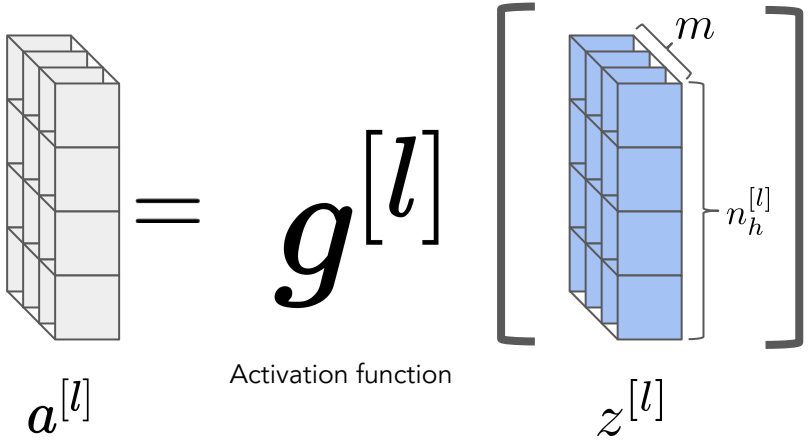
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$



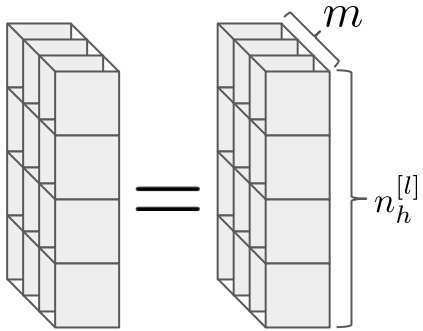
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$



$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$



$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$



$a^{[l]}$ Result Tensor

Vectorizing Gradient Descent

Going from: Iteratively changing each weight by the derivative of the cost function

To: In parallel, simultaneously changing each weight by the derivative of the cost function

Vectorizing Gradient Descent

Going from: Iteratively changing each weight by the derivative of the cost function

To: In parallel, simultaneously changing each weight by the derivative of the cost function

Previously, $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$

Now, W is the weight matrix for layer l

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial}{\partial W^{[l]}} J(\theta)$$

Note similarly we do this for the bias vector as well

Recap: Gradient descent for Neural Networks

Repeat for many iterations (training steps):

Compute forward pass and the cost function, **J**

$$W^{[1]} := W^{[1]} - \alpha \frac{\partial J}{\partial W^{[1]}}$$

$$b^{[1]} := b^{[1]} - \alpha \frac{\partial J}{\partial b^{[1]}}$$

$$W^{[2]} := W^{[2]} - \alpha \frac{\partial J}{\partial W^{[2]}}$$

$$b^{[2]} := b^{[2]} - \alpha \frac{\partial J}{\partial b^{[2]}}$$

Derivatives for a simple Neural Network

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

Note: $dZ^{[1]} \rightarrow \frac{\partial J}{\partial Z^{[1]}}$

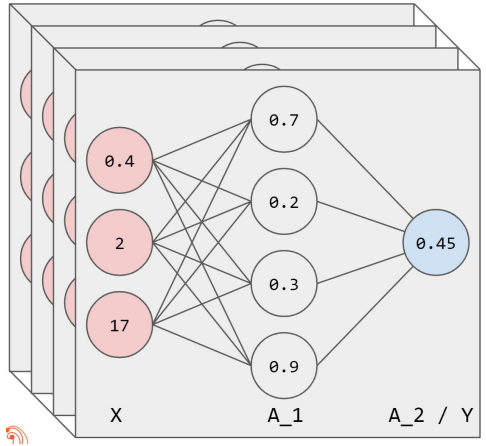
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$$

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

$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} * g^{[1]\prime}(\mathrm{Z}^{[1]})$$

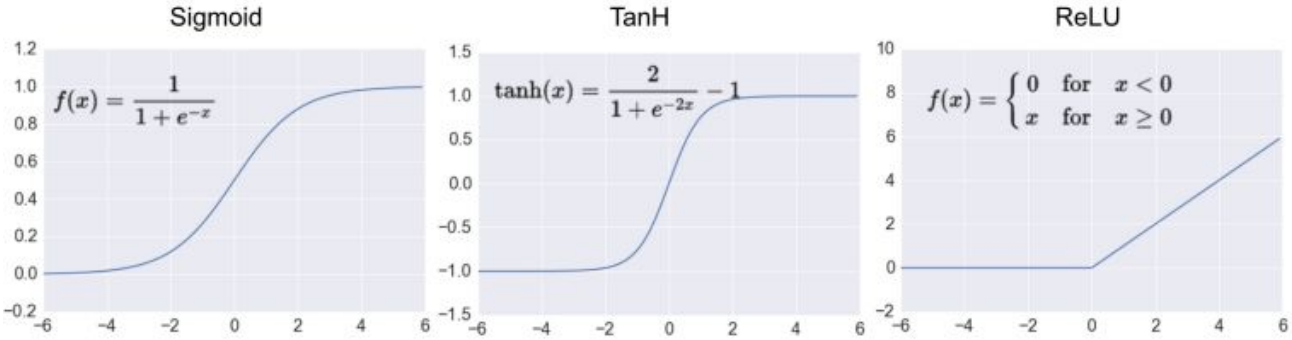
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

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



Derivatives for activation functions

$$g'(x) = \begin{cases} \dfrac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x)) \\ \dfrac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \\ \dfrac{d}{dx}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases} \end{cases}$$



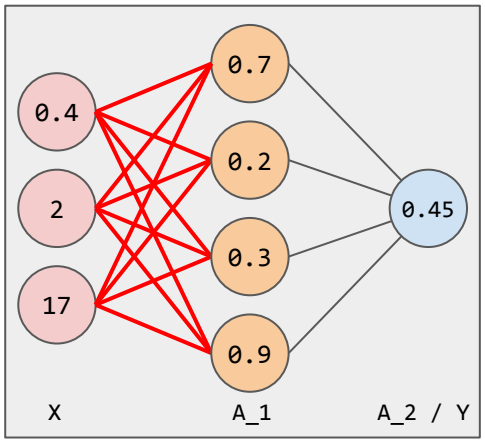
Deep Neural Networks

The general case

Abstracting Layers

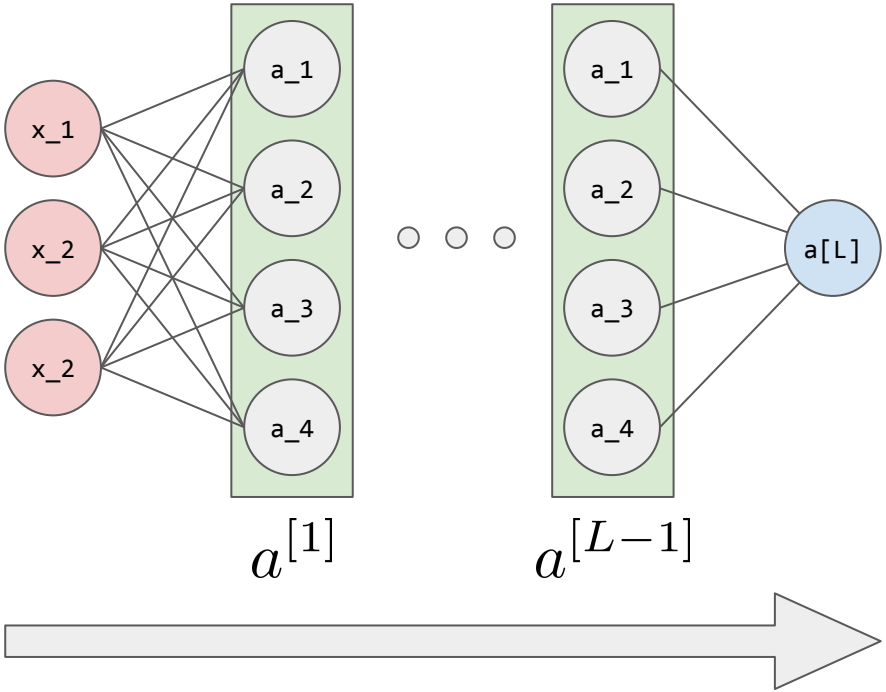
A layer is an abstract generalization of a transformation function on the previous activations.

$$a^{[l]} = f^{[l]}(a^{[l-1]})$$



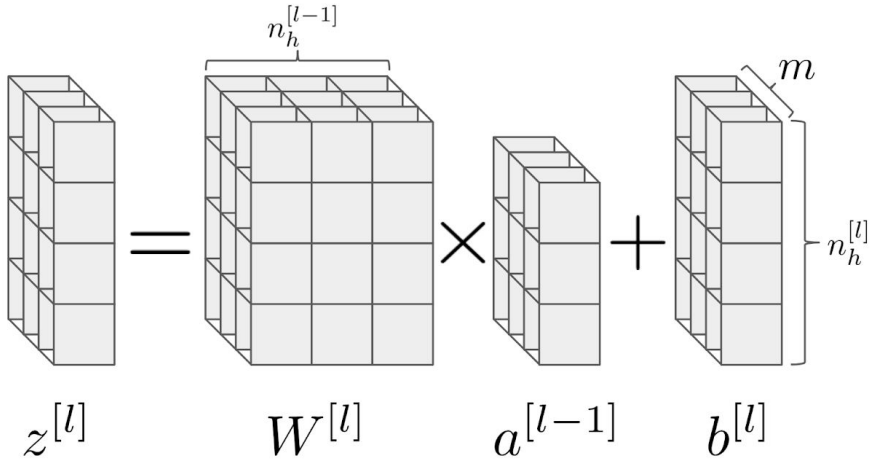
$$a^{[1]} = f^{[1]}(\mathbf{x})$$

Forward Propagation

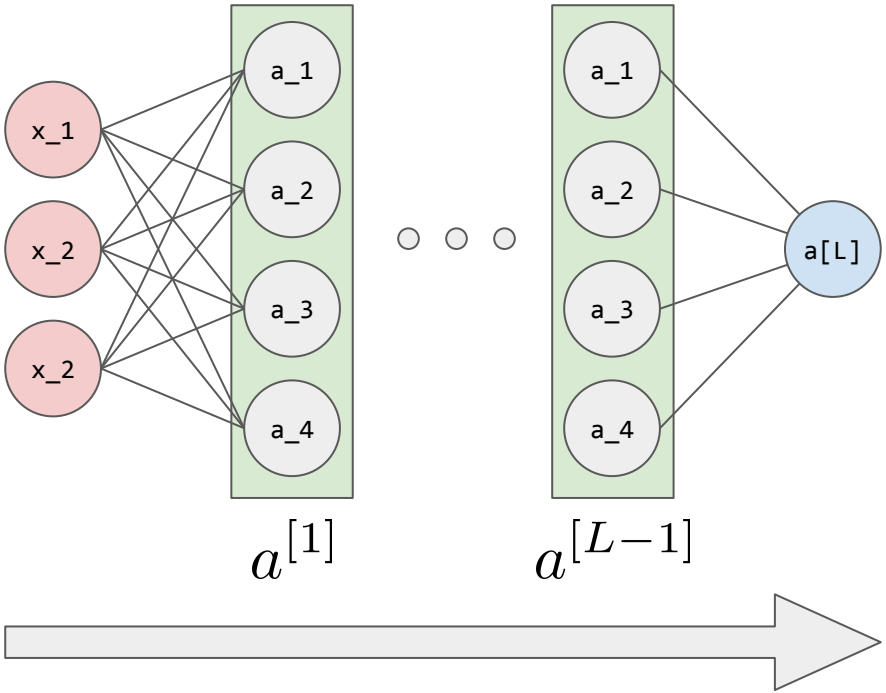


Repeat L-1 times

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

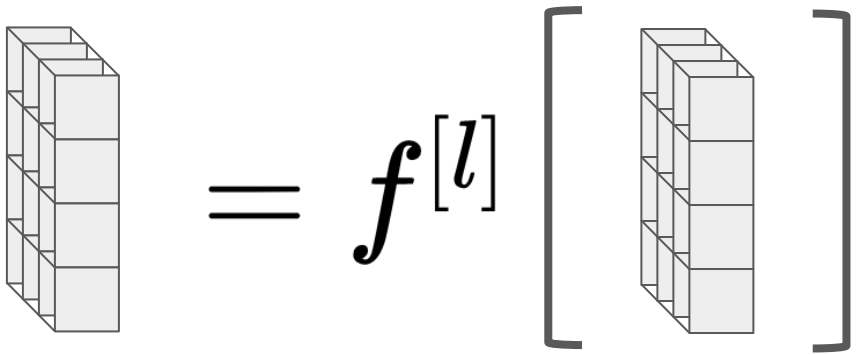


Layered Forward Propagation



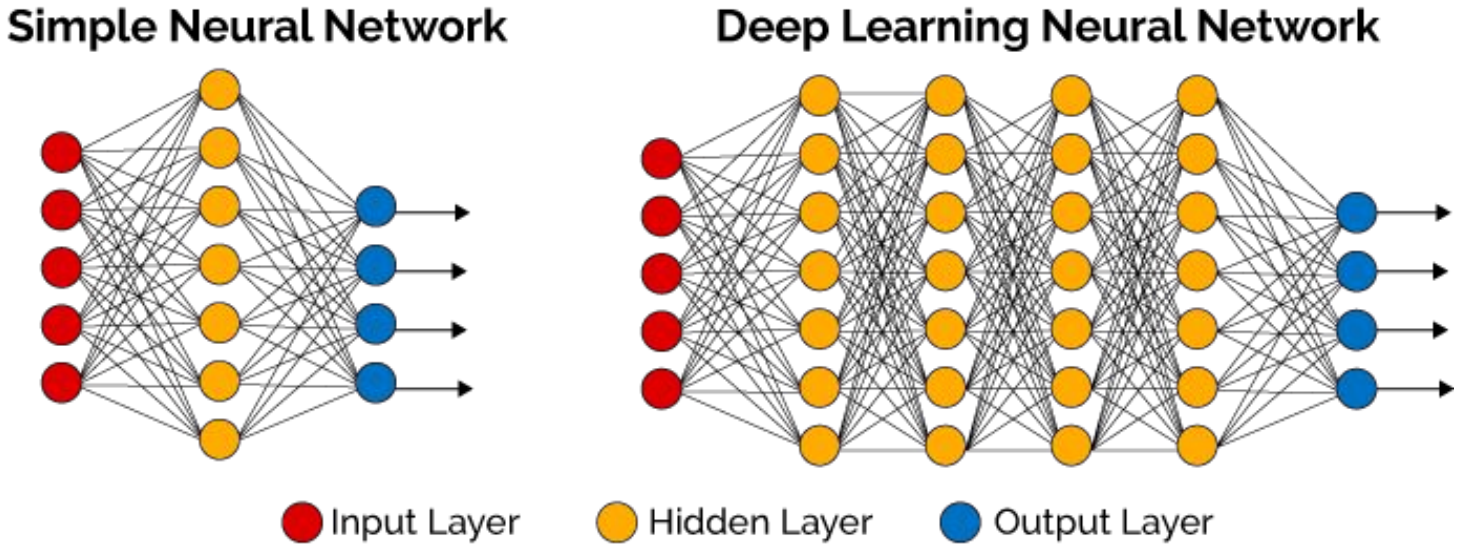
Repeat L-1 times

$$a^{[l]} = f^{[l]}(a^{[l-1]})$$



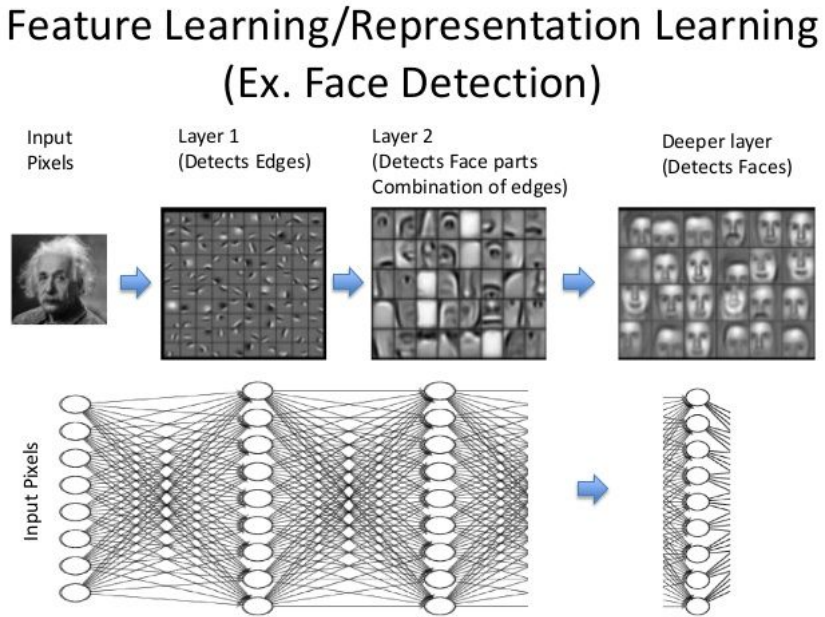
Deep Neural Networks

1. **Just a neural network with more layers**
2. **The number of layers is notated as L**

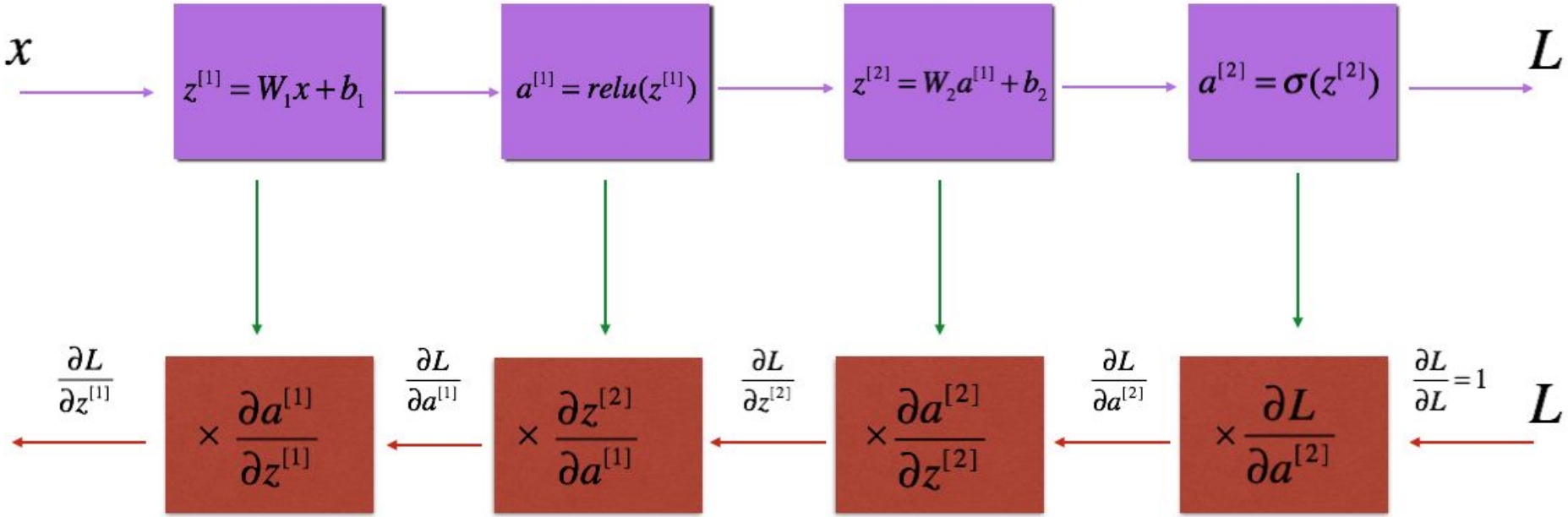


Why do we need more layers?

1. **Lower layers learn more basic features of the input (edge detection)**
2. **Deeper layers build off of previously learned features to conceptualize more complex features (face detection)**
3. **Adding more layers allows the network to approximate more complex functions**



Backward Propagation



Backward Propagation

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

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

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

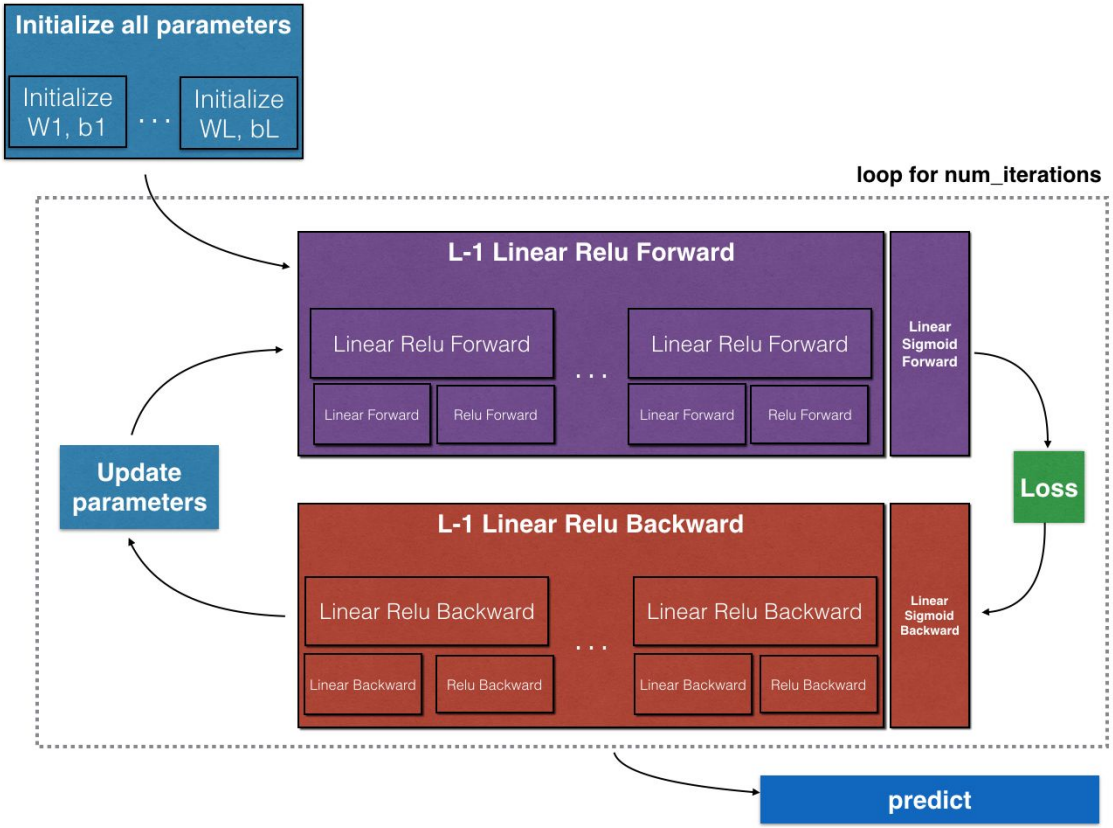
$$dZ^{[l]} = W^{[l+1]T} dZ^{[l+1]} * g^{[l]'}(Z^{[l]})$$

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

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

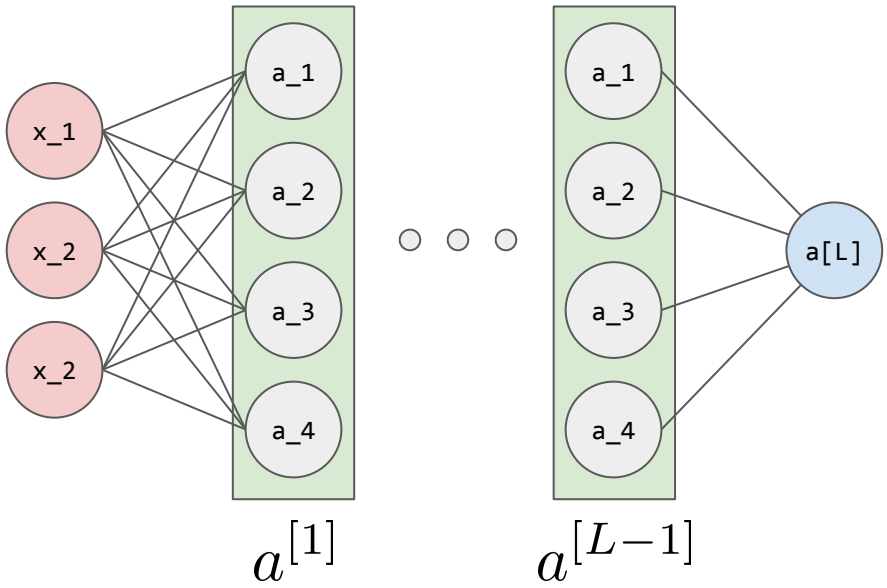
```
for l = L-1 to 1
```

Building blocks of deep neural networks



Summary

1. **Forward propagate**
2. **Compute the cost**
3. **Back propagate**
4. **Update weights**



References and Further Reading

1. Deep Learning Specialization by Andrew Ng: https://www.deeplearning.ai/
2. Open AI five: https://blog.openai.com/openai-five/
3. AlphaGo Zero: https://deepmind.com/blog/alphago-zero-learning-scratch/
4. Google Colab, Free Cloud GPUs: https://colab.research.google.com
5. BU MIC Youtube Channel: https://www.youtube.com/channel/UCWZLArtz3qv7bVaughcI1bA
6. Feedback form for this workshop: https://goo.gl/forms/i9dNSezmGjFKbsXD3
7. Deep Learning Applications: Forbes Deep Learning Applications
8. Deep Learning vs. ML: https://www.zendesk.com/blog/machine-learning-and-deep-learning/
9. AlexNet: https://en.wikipedia.org/wiki/AlexNet