Machine Learning 101

Source: https://xkcd.com/

Brad Quinton, Scott Chin

Implementation and Evaluation of Neural Networks

Deep Learning

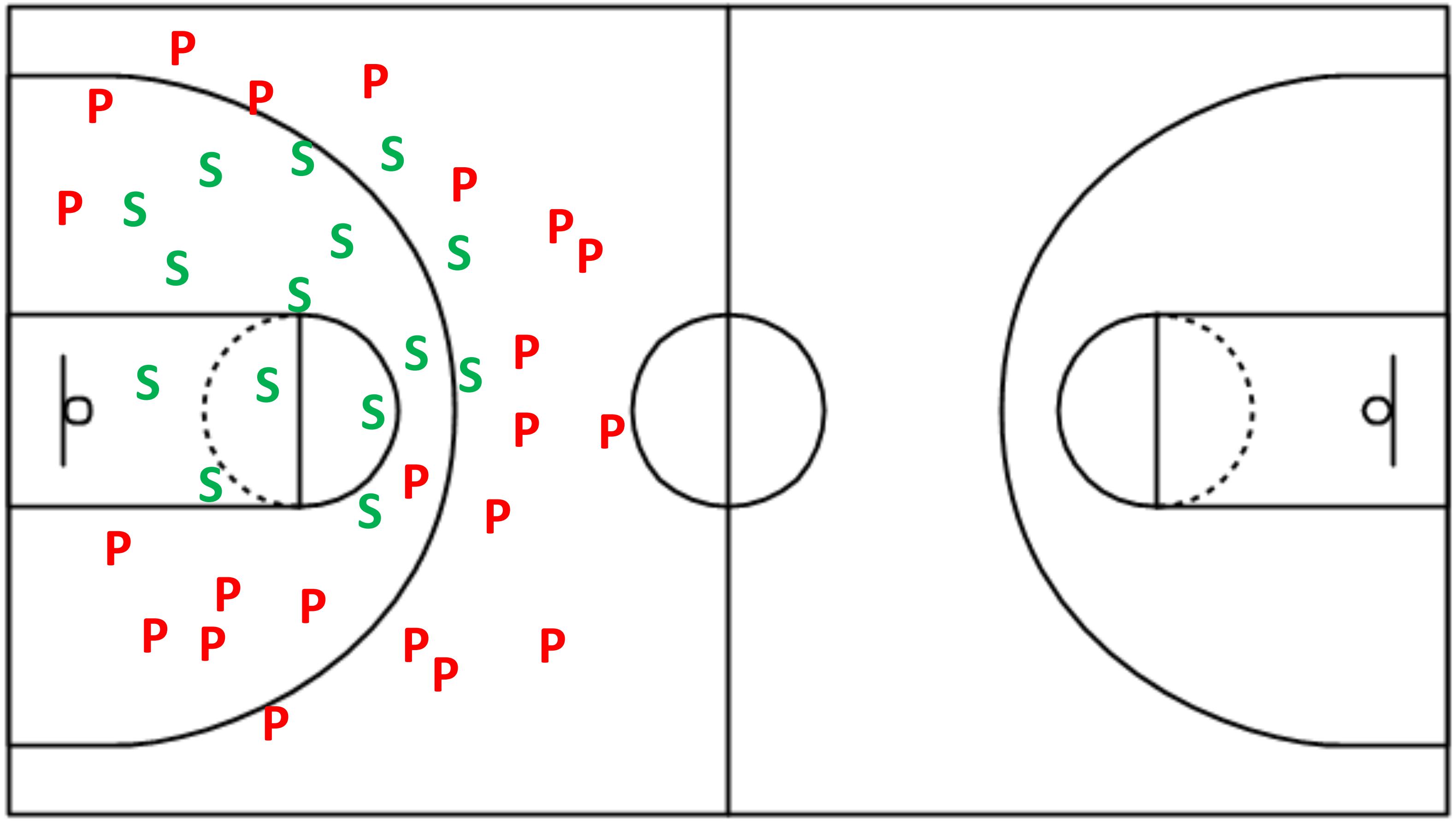
Brad Quinton, Scott Chin

Case Study #2: Where we left off…

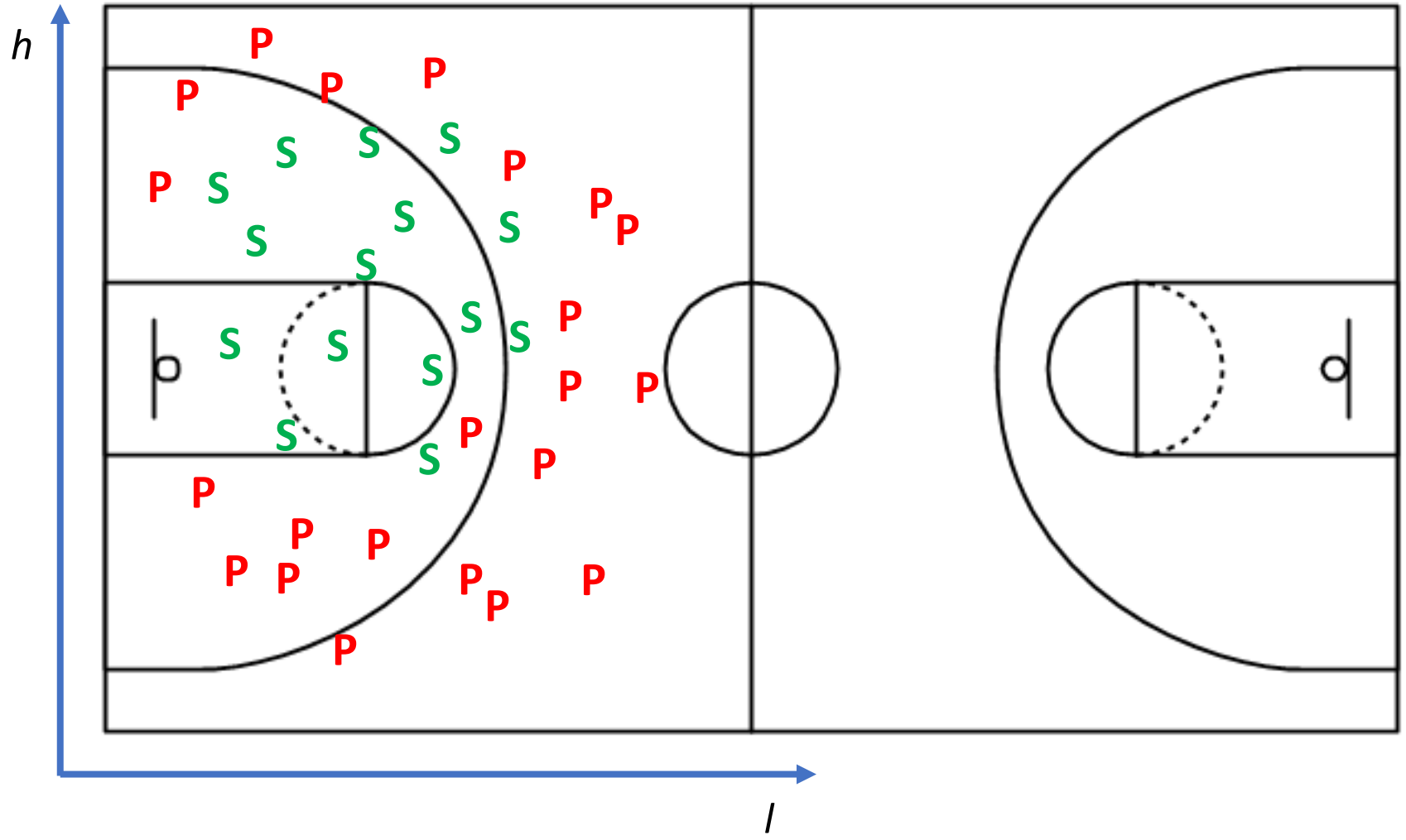
- In the last lecture we were given the task of creating an AI to tell basketball players if they should “shoot” or “pass” when they have the ball on offense

- After understanding the problem (*if you shoot you might miss and lose possession*), and locating a source of labelled data (*post-game video analysis*), we took a quick look at some of the data

Case Study #2: Take a Look at the Data

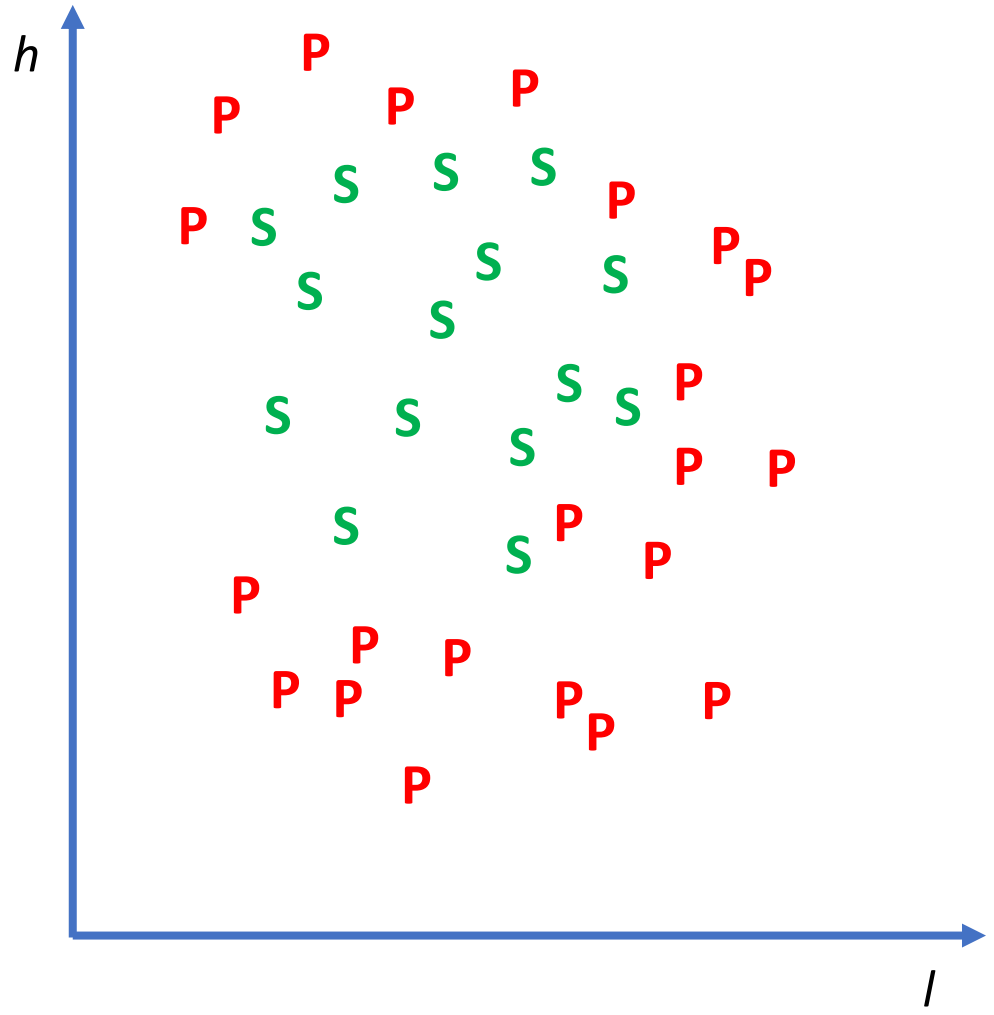


Case Study #2: Take a Look at the Data



Brad Quinton, Scott Chin

Case Study #2: Take a Look at the Data



- The data has a clear pattern, but it is non-linear….

Case Study #2: Where we left off…

- From a quick look at two features of the the input data, we see that Logistic Regression is unlikely to work well

- Instead, we convinced ourselves that by combining simple Logistic Regression units together using a computation graph we could create a *Neural Network* that could **learn complex functions**

- But, to make progress we need a more systematic way to implement efficient and “tunable” Neural Networks

Efficient Implementation of Neural Networks

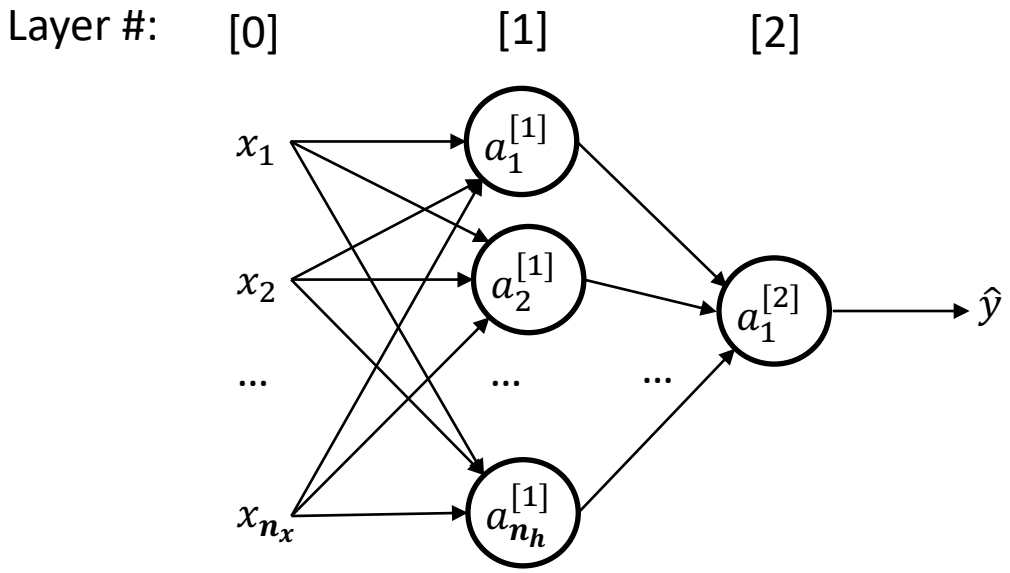
- Just like with Logistic Regression, the first key to efficient implementations of Neural Networks is using **vectorization** to “group” operations together so they can be efficiently mapped to specialized hardware

- The second key is to plan the algorithm to **avoid re-calculating** values that are used repeatedly

- Finally, we need to do all this while remembering that the number of layers in the NN, and the number of neurons in each layer are tunable “**hyperparameters**” that will need to be easily modified without having to re-work on implementation…

Neural Network Vectorization

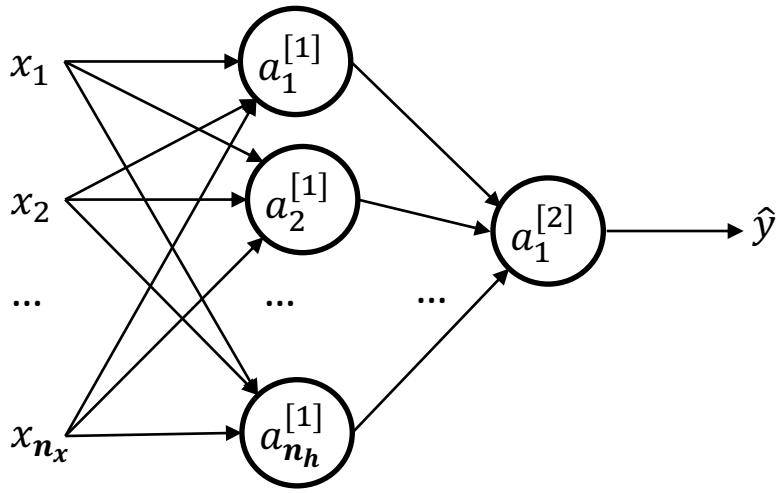
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



Neural Network Vectorization

- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :

Layer #: [0] [1] [2]

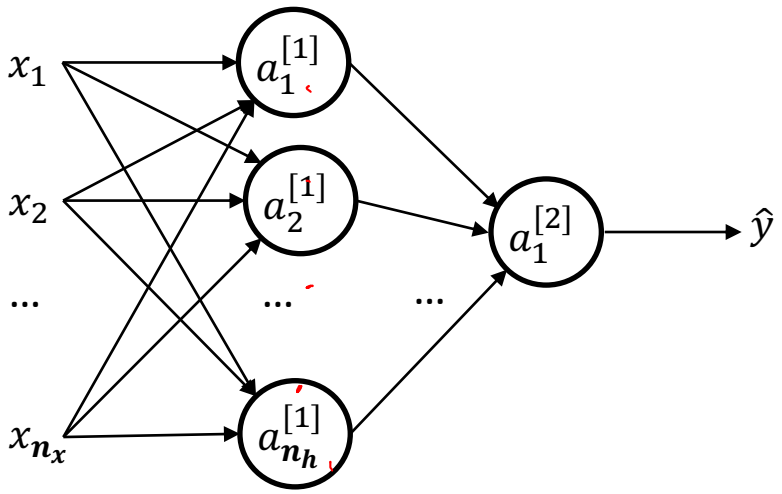


- $\boldsymbol{n_x}$ is the number of input features

Neural Network Vectorization

- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :

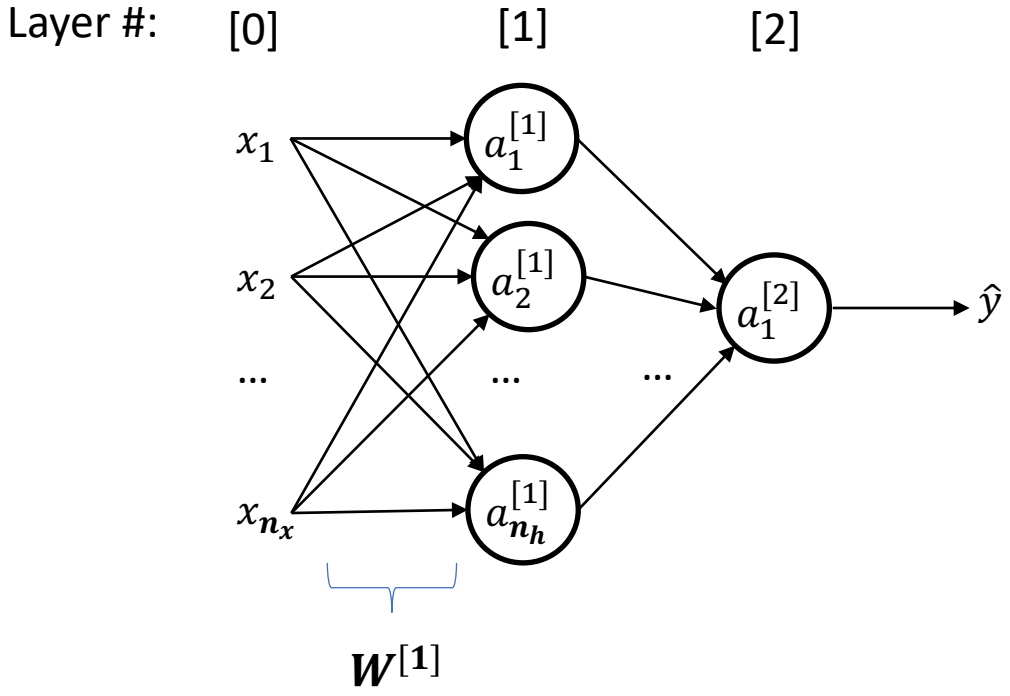
Layer #: [0] [1] [2]



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].

Neural Network Vectorization

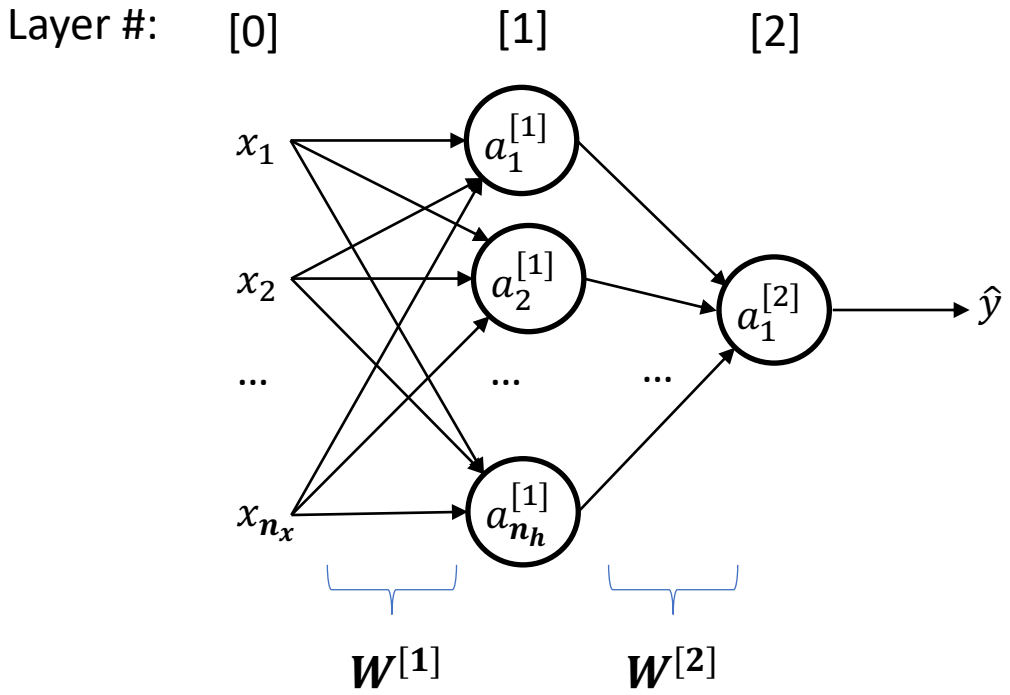
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].
- $W^{[1]}$ is a matrix of all the parameters in layer [1]. Shape (n_h, n_x).

Neural Network Vectorization

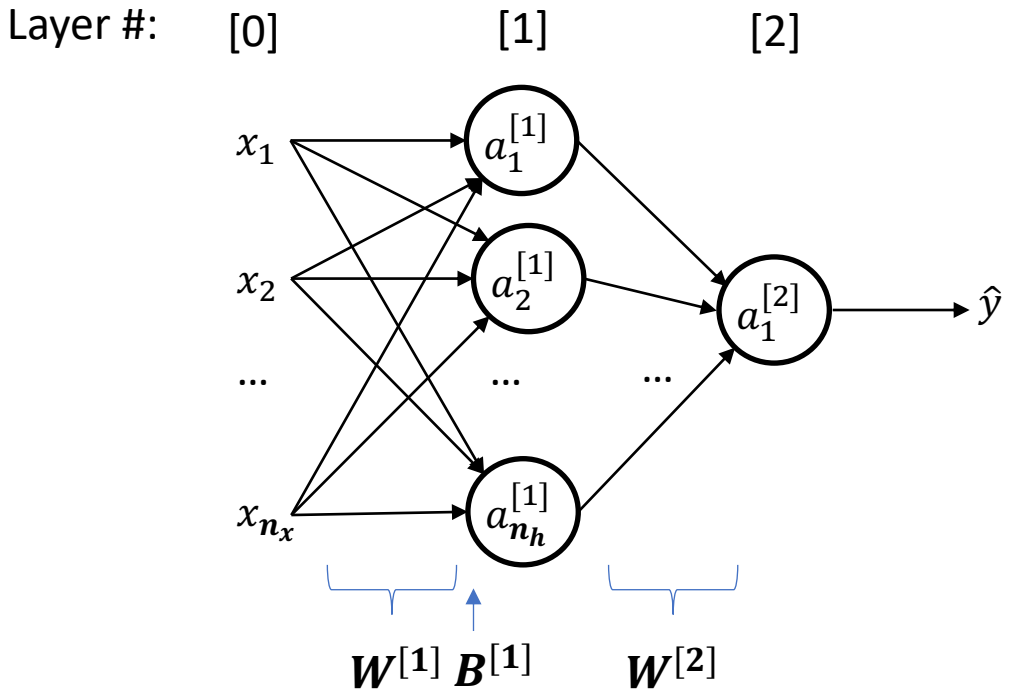
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].
- $W^{[1]}$ is a matrix of all the parameters in layer [1]. Shape (n_h, n_x).
- $W^{[2]}$ is a matrix of all the parameters in layer [2]. Shape (1, n_h).

Neural Network Vectorization

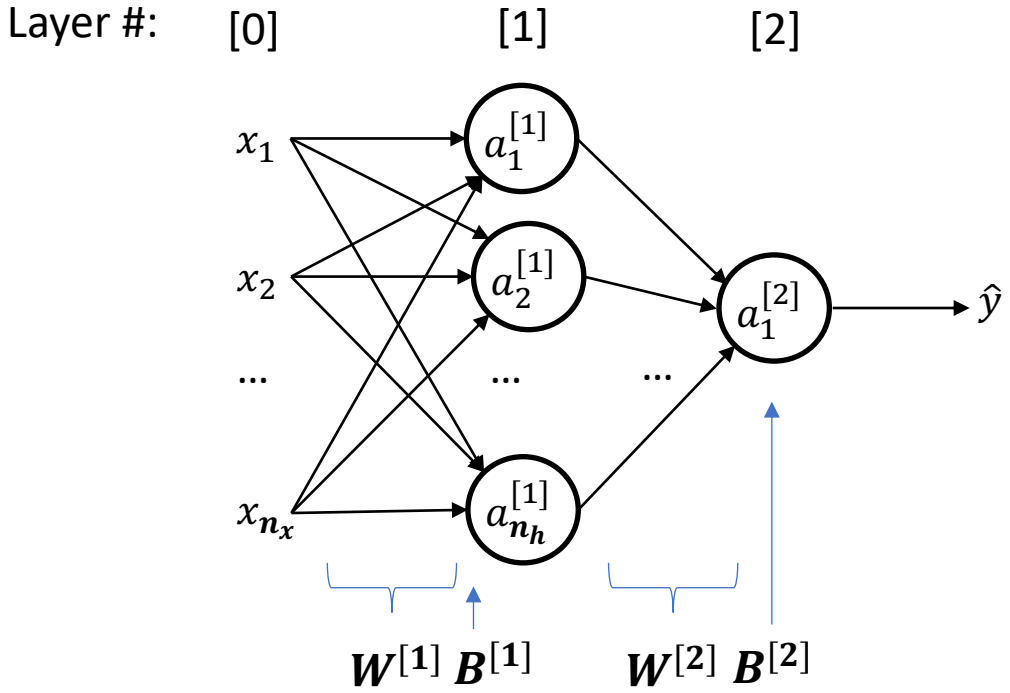
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- $\boldsymbol{n_x}$ is the number of input features
- $\boldsymbol{n_h}$ is the number of hidden units in layer [1].
- $\boldsymbol{W^{[1]}}$ is a matrix of all the parameters in layer [1]. Shape ($\boldsymbol{n_h}$, $\boldsymbol{n_x}$).
- $\boldsymbol{W^{[2]}}$ is a matrix of all the parameters in layer [2]. Shape (1, $\boldsymbol{n_h}$).
- $\boldsymbol{B^{[1]}}$ is a vector of the bias parameters in layer [1]. Shape ($\boldsymbol{n_h}$, 1).

Neural Network Vectorization

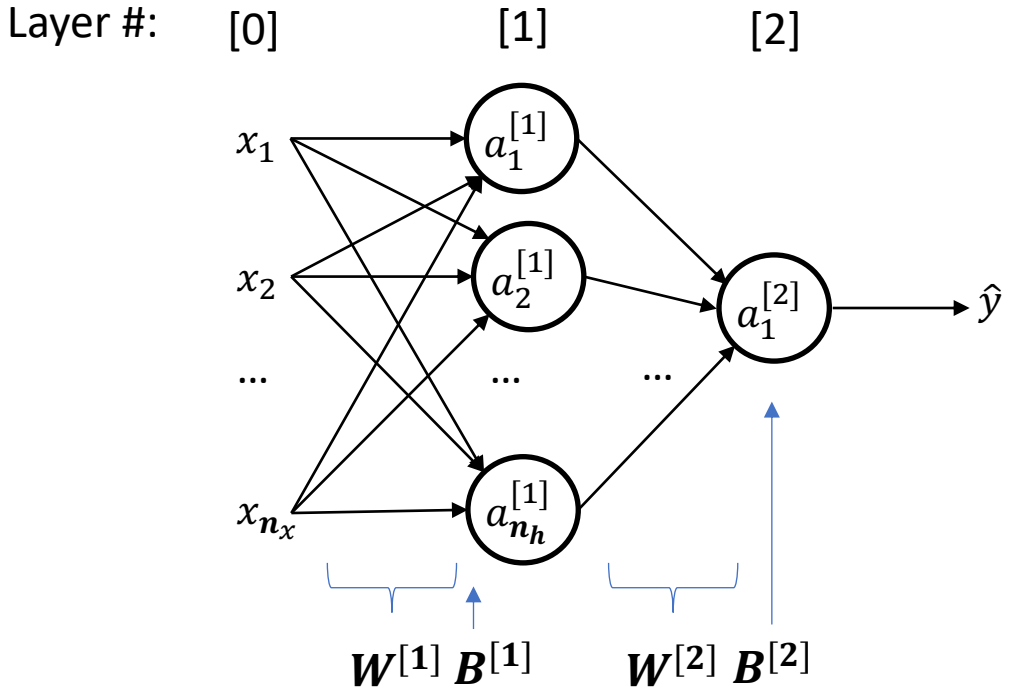
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].
- $W^{[1]}$ is a matrix of all the parameters in layer [1]. Shape (n_h, n_x).
- $W^{[2]}$ is a matrix of all the parameters in layer [2]. Shape (1, n_h).
- $B^{[1]}$ is a vector of the bias parameters in layer [1]. Shape (n_h, 1).
- $B^{[2]}$ is a vector of the bias parameters in layer [2]. Shape (1, 1).

Neural Network Vectorization

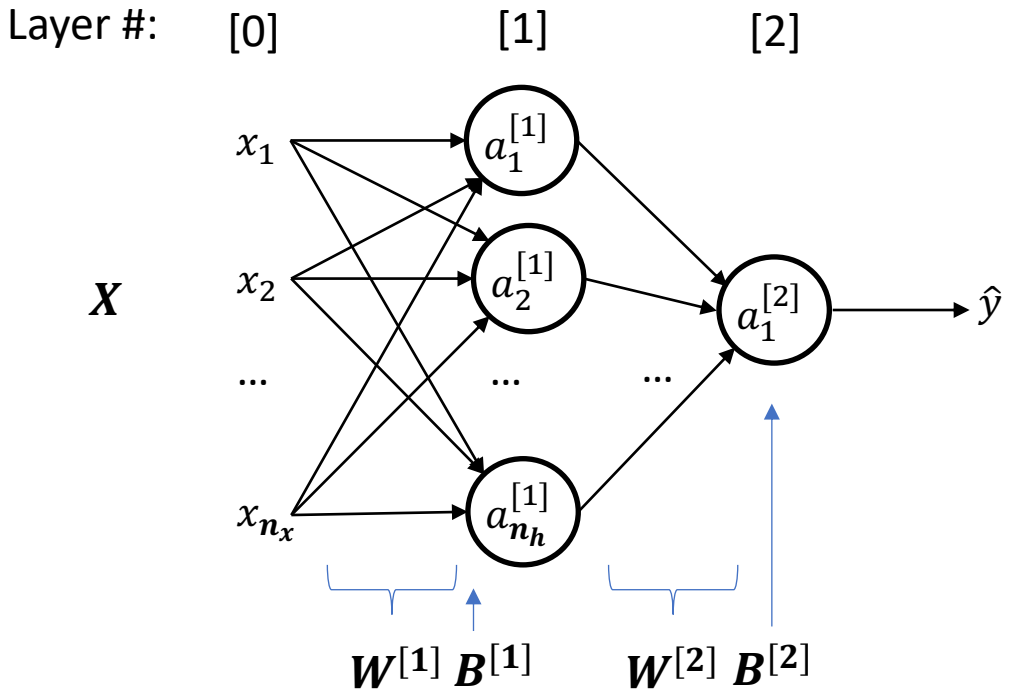
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].
- $W^{[1]}$ is a matrix of all the parameters in layer [1]. Shape (n_h, n_x).
- $W^{[2]}$ is a matrix of all the parameters in layer [2]. Shape (1, n_h).
- $B^{[1]}$ is a vector of the bias parameters in layer [1]. Shape (n_h, 1).
- $B^{[2]}$ is a vector of the bias parameters in layer [2]. Shape (1, 1).
- m is the number of examples in the training data set

Neural Network Vectorization

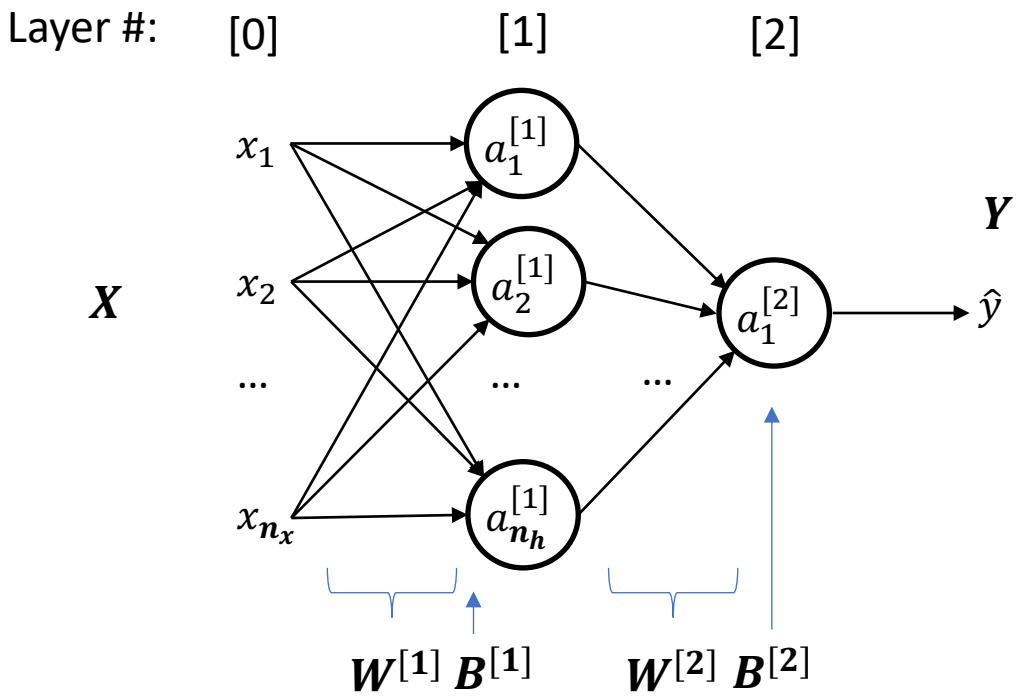
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- $\boldsymbol{n_x}$ is the number of input features
- $\boldsymbol{n_h}$ is the number of hidden units in layer [1].
- $\boldsymbol{W^{[1]}}$ is a matrix of all the parameters in layer [1]. Shape ($\boldsymbol{n_h}$, $\boldsymbol{n_x}$).
- $\boldsymbol{W^{[2]}}$ is a matrix of all the parameters in layer [2]. Shape (1, $\boldsymbol{n_h}$).
- $\boldsymbol{B^{[1]}}$ is a vector of the bias parameters in layer [1]. Shape ($\boldsymbol{n_h}$, 1).
- $\boldsymbol{B^{[2]}}$ is a vector of the bias parameters in layer [2]. Shape (1, 1).
- $\boldsymbol{m}$ is the number of examples in the training data set
- $\boldsymbol{X}$ is a matrix of all the input features for all examples in the training data set. Shape ($\boldsymbol{n_x}, \boldsymbol{m}$)

Neural Network Vectorization

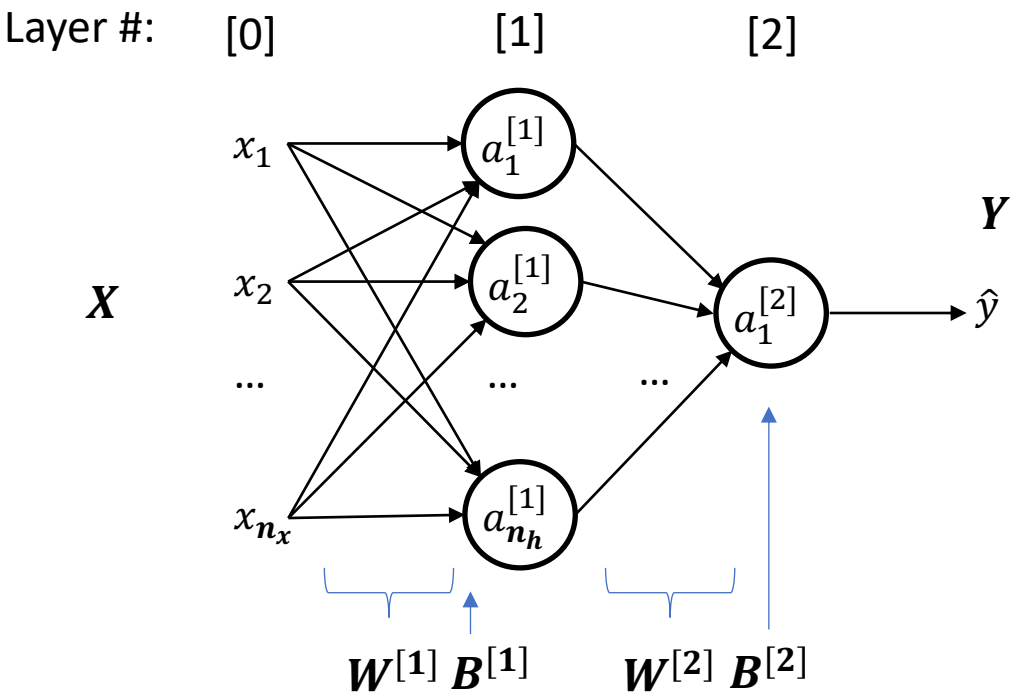
- Lets workout how we can vectorize with 1 hidden layer (and single output, $\hat{y}$) :



- n_x is the number of input features
- n_h is the number of hidden units in layer [1].
- $W^{[1]}$ is a matrix of all the parameters in layer [1]. Shape (n_h, n_x).
- $W^{[2]}$ is a matrix of all the parameters in layer [2]. Shape (1, n_h).
- $B^{[1]}$ is a vector of the bias parameters in layer [1]. Shape (n_h, 1).
- $B^{[2]}$ is a vector of the bias parameters in layer [2]. Shape (1, 1).
- m is the number of examples in the training data set
- X is a matrix of all the input features for all examples in the training data set. Shape (n_x, m)
- Y is the labels for all the examples in the training data set. Shape $(1, m)$

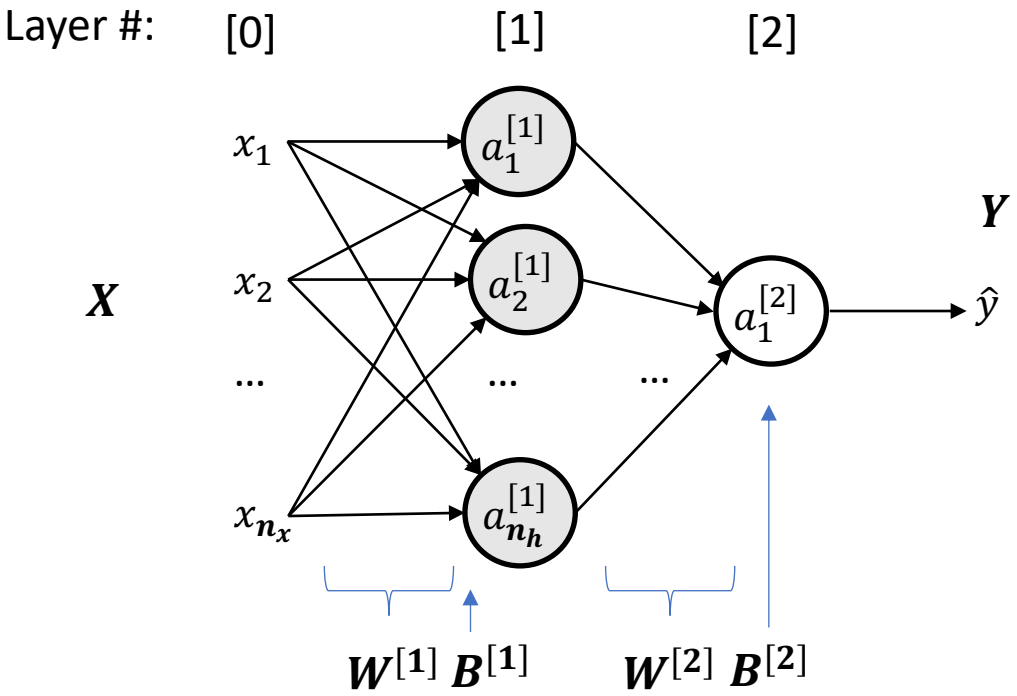
Forward Propagation

- We can workout the activation of each layer:



Forward Propagation

- We can workout the activation of each layer:

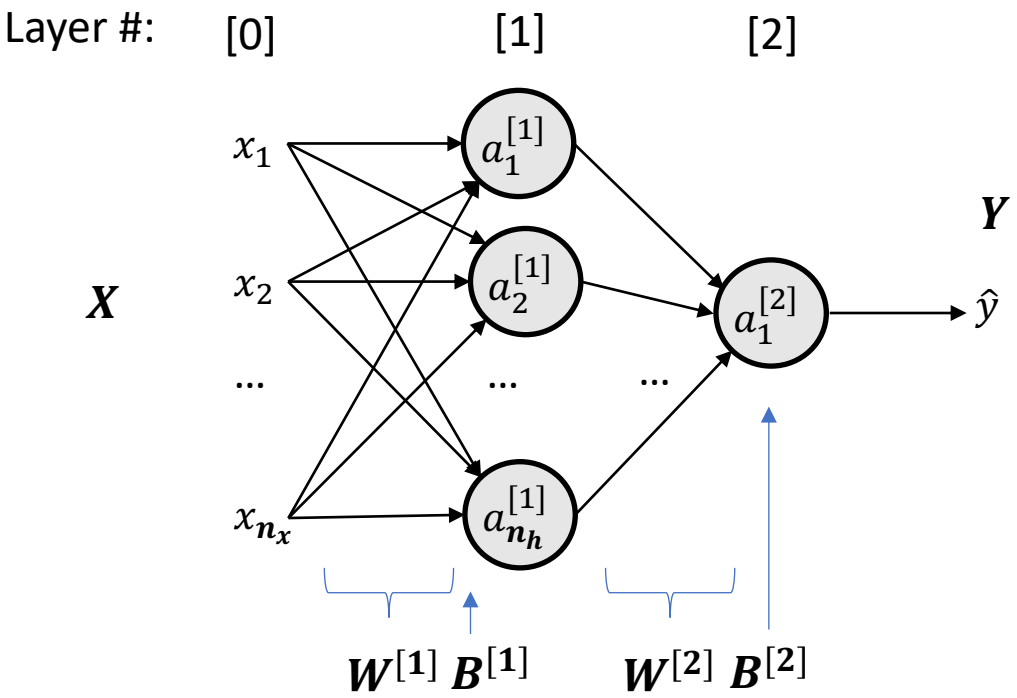


- If we consider a single example, i:

$$z^{[1](i)} = W^{[1]}x^{(i)} + B^{[1]}$$
$$a^{[1](i)} = g\left(z^{[1](i)}\right), \text{ where } g(z) = \tanh(z)$$

Forward Propagation

- We can workout the activation of each layer:



- If we consider a single example, i:

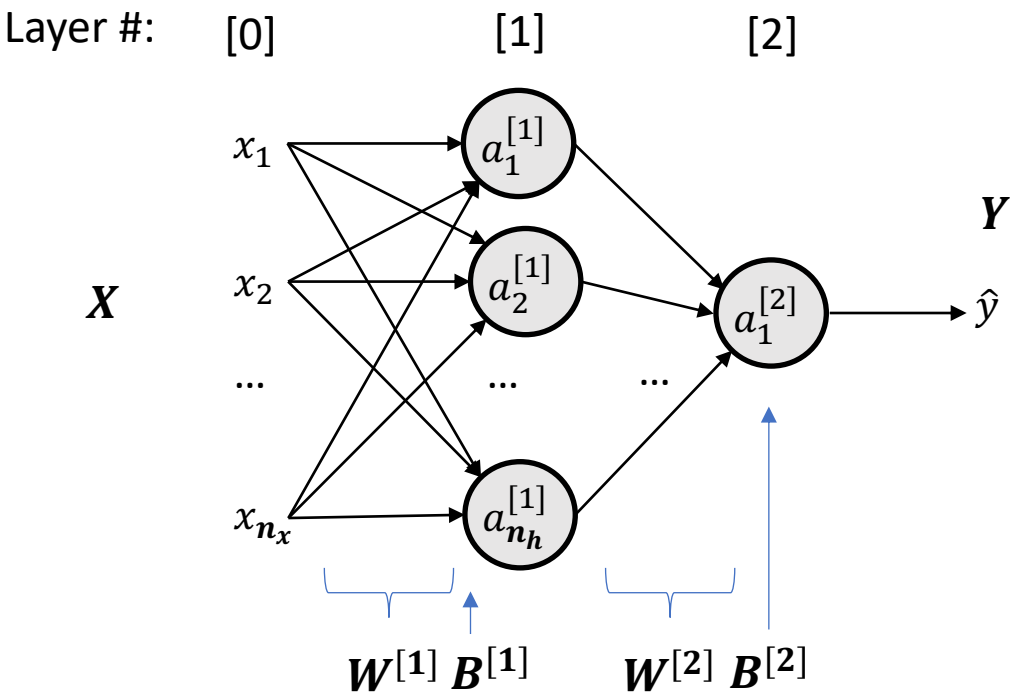
$$z^{[1](i)} = W^{[1]}x^{(i)} + B^{[1]}$$
$$a^{[1](i)} = g\left(z^{[1](i)}\right), \text{ where } g(z) = \tanh(z)$$

- And:

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + B^{[1]}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$

Forward Propagation

- We can workout the activation of each layer:



- For all m examples:

$$Z^{[1]} = W^{[1]}X + B^{[1]}$$
$$A^{[1]} = g\left(Z^{[1]}\right), \text{where } g(Z) = \tanh(Z)$$

- And:

$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[1]}$$
$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]})$$

Backpropagation

- Recall our 6 key backpropagation equations from last lecture

(1) $$\frac{\partial L}{\partial z_1^{[2]}} = (\hat{y} - y)$$

(2) $$\frac{\partial L}{\partial w_{1,1}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \cdot a_1^{[1]}$$

(3) $$\frac{\partial L}{\partial b_1^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}}$$

(4) $$\frac{\partial L}{\partial z_1^{[1]}} = w_1^{[2]} \cdot \frac{\partial L}{\partial z_1^{[2]}} \cdot \mathrm{g}'(z_1^{[1]})$$

(5) $$\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \cdot x_1$$

(6) $$\frac{\partial L}{\partial b_1^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}}$$

Backpropagation

- Recall our 6 key backpropagation equations from last lecture

- When we are implementing backprop, the $\frac{\partial L}{\partial z}$ notation style becomes cumbersome, so lets assume we are always referring to ∂L and use, for example dz to mean $\frac{\partial L}{\partial z}$

(1) $$\frac{\partial L}{\partial z_1^{[2]}} = (\hat{y} - y)$$

(2) $$\frac{\partial L}{\partial w_{1,1}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \cdot a_1^{[1]}$$

(3) $$\frac{\partial L}{\partial b_1^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}}$$

(4) $$\frac{\partial L}{\partial z_1^{[1]}} = w_1^{[2]} \cdot \frac{\partial L}{\partial z_1^{[2]}} \cdot \mathrm{g}'(z_1^{[1]})$$

(5) $$\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \cdot x_1$$

(6) $$\frac{\partial L}{\partial b_1^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}}$$

Backpropagation

- Recall our 6 key backpropagation equations from last lecture

- When we are implementing backprop, the $\frac{\partial L}{\partial z}$ notation style becomes cumbersome, so lets assume we are always referring to ∂L and use, for example dz to mean $\frac{\partial L}{\partial z}$

(1) $\frac{\partial L}{\partial z_1^{[2]}} = (\hat{y} - y)$ ➡ $dz_1^{[2]} = (\hat{y} - y)$

(2) $\frac{\partial L}{\partial w_{1,1}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \cdot a_1^{[1]}$ ➡ $dw_{1,1}^{[2]} = dz_1^{[2]} a_1^{[1]}$

(3) $\frac{\partial L}{\partial b_1^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}}$ ➡ $db_1^{[2]} = dz_1^{[2]}$

(4) $\frac{\partial L}{\partial z_1^{[1]}} = w_1^{[2]} \cdot \frac{\partial L}{\partial z_1^{[2]}} \cdot \mathrm{g}'(z_1^{[1]})$ ➡ $dz_1^{[1]} = w_1^{[2]} dz_1^{[2]} \mathrm{g}'(z_1^{[1]})$

(5) $\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \cdot x_1$ ➡ $dw_{1,1}^{[1]} = dz_1^{[1]} x_1$

(6) $\frac{\partial L}{\partial b_1^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}}$ ➡ $db_1^{[1]} = dz_1^{[1]}$

Backpropagation - Vectorization

Element-wise

(1) $dz_1^{[2]} = (\hat{y} - y)$

(2) $dw_{1,1}^{[2]} = dz_1^{[2]} a_1^{[1]}$

(3) $db_1^{[2]} = dz_1^{[2]}$

(4) $dz_1^{[1]} = w_1^{[2]} dz_1^{[2]} \mathrm{g}'(z_1^{[1]})$

(5) $dw_{1,1}^{[1]} = dz_1^{[1]} x_1$

(6) $db_1^{[1]} = dz_1^{[1]}$

Brad Quinton, Scott Chin

Backpropagation - Vectorization

	Element-wise	Vectorized - single sample, i
(1)	$dz_1^{[2]} = (\hat{y} - y)$	$dz^{[2](i)} = (\hat{y}^{(i)} - y^{(i)})$
(2)	$dw_{1,1}^{[2]} = dz_1^{[2]} a_1^{[1]}$	$dW^{[2](i)} = dz^{[2](i)} a^{[1](i)T}$
(3)	$db_1^{[2]} = dz_1^{[2]}$	$dB^{[2](i)} = dz^{[2](i)}$
(4)	$dz_1^{[1]} = w_1^{[2]} dz_1^{[2]} \mathrm{g}'(z_1^{[1]})$	$dz^{[1](i)} = W^{[2](i)T} dz^{[2](i)} * \mathrm{g}'(z^{[1(i)]})$
(5)	$dw_{1,1}^{[1]} = dz_1^{[1]} x_1$	$dW^{[1](i)} = dz^{[1](i)} x^{(i)T}$
(6)	$db_1^{[1]} = dz_1^{[1]}$	$dB^{[1](i)} = dz^{[1](i)}$

Brad Quinton, Scott Chin

Backpropagation - Vectorization

	Element-wise	Vectorized - single sample, i	Vectorize for all m samples
(1)	$dz_1^{[2]} = (\hat{y} - y)$	$dz^{[2](i)} = (\hat{y}^{(i)} - y^{(i)})$	$dZ^{[2]} = (\hat{Y} - Y)$
(2)	$dw_{1,1}^{[2]} = dz_1^{[2]} a_1^{[1]}$	$dW^{[2](i)} = dz^{[2](i)} a^{[1](i)T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
(3)	$db_1^{[2]} = dz_1^{[2]}$	$dB^{[2](i)} = dz^{[2](i)}$	$dB^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$
(4)	$dz_1^{[1]} = w_1^{[2]} dz_1^{[2]} \mathrm{g}'(z_1^{[1]})$	$dz^{[1](i)} = W^{[2](i)T} dz^{[2](i)} * \mathrm{g}'(z^{[1(i)]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \mathrm{g}'(Z^{[1]})$
(5)	$dw_{1,1}^{[1]} = dz_1^{[1]} x_1$	$dW^{[1](i)} = dz^{[1](i)} x^{(i)T}$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
(6)	$db_1^{[1]} = dz_1^{[1]}$	$dB^{[1](i)} = dz^{[1](i)}$	$dB^{[1]} = \frac{1}{m} \sum_{rows} dZ^{[1]}$

Brad Quinton, Scott Chin

Backpropagation - Vectorization

	Element-wise	Vectorized - single sample, i	Vectorize for all m samples
(1)	$dz_1^{[2]} = (\hat{y} - y)$	$dz^{[2](i)} = (\hat{y}^{(i)} - y^{(i)})$	$dZ^{[2]} = (\hat{Y} - Y)$ $(1, m)$ $(1, m)$ $(1, m)$
(2)	$dw_{1,1}^{[2]} = dz_1^{[2]} a_1^{[1]}$	$dW^{[2](i)} = dz^{[2](i)} a^{[1](i)T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$ $(1, n_h)$ $(1, m)$ (m, n_h)
(3)	$db_1^{[2]} = dz_1^{[2]}$	$dB^{[2](i)} = dz^{[2](i)}$	$dB^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$ $(1,1)$ $(1, m)$
(4)	$dz_1^{[1]} = w_1^{[2]} dz_1^{[2]} \mathrm{g}'(z_1^{[1]})$	$dz^{[1](i)} = W^{[2](i)T} dz^{[2](i)} * \mathrm{g}'(z^{[1(i)]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \mathrm{g}'(Z^{[1]})$ (n_h, m) $(n_h, 1)$ $(1, m)$
(5)	$dw_{1,1}^{[1]} = dz_1^{[1]} x_1$	$dW^{[1](i)} = dz^{[1](i)} x^{(i)T}$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$ (n_h, n_x) (n_h, m) (m, n_x)
(6)	$db_1^{[1]} = dz_1^{[1]}$	$dB^{[1](i)} = dz^{[1](i)}$	$dB^{[1]} = \frac{1}{m} \sum_{rows} dZ^{[1]}$ $(n_h, 1)$ (n_h, m)

Brad Quinton, Scott Chin

Parameter Update - Vectorization

Element-wise

$$w_{1,1}^{[1]} = w_{1,1}^{[1]} - \alpha \cdot dw_{1,1}^{[1]}$$

$$b_1^{[1]} = b_1^{[1]} - \alpha \cdot db_1^{[1]}$$

$$w_{1,1}^{[2]} = w_{1,1}^{[2]} - \alpha \cdot dw_{1,1}^{[2]}$$

$$b_1^{[2]} = b_1^{[2]} - \alpha \cdot db_1^{[2]}$$

Vectorized - single sample, i

$$W^{[1](i)} = W^{[1](i)} - \alpha \cdot dW^{[1](i)}$$

$$B^{[1](i)} = B^{[1](i)} - \alpha \cdot dB^{[1](i)}$$

$$W^{[2](i)} = W^{[2](i)} - \alpha \cdot dW^{[2](i)}$$

$$B^{[2](i)} = B^{[2](i)} - \alpha \cdot dB^{[2](i)}$$

Vectorize for all m samples

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$$

$$B^{[1]} = B^{[1]} - \alpha \cdot dB^{[1]}$$

$$W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]}$$

$$B^{[2]} = B^{[2]} - \alpha \cdot dB^{[2]}$$

Vectorized Backpropagation

1. Make an initial prediction

$$Z^{[1]} = W^{[1]}X + B^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[1]}$$
$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]})$$

2. Determine Partial Derivatives

$$dZ^{[2]} = (\hat{Y} - Y)$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$dB^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]})$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$
$$dB^{[1]} = \frac{1}{m} \sum_{rows} dZ^{[1]}$$

3. Update parameters

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$$
$$B^{[1]} = B^{[1]} - \alpha \cdot dB^{[1]}$$
$$W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]}$$
$$B^{[2]} = B^{[2]} - \alpha \cdot dB^{[2]}$$

4. Repeat until J < target

Vectorized Backpropagation

0. Initialize the Parameters

?

1. Make an initial prediction

$$Z^{[1]} = W^{[1]}X + B^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[1]}$$
$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]})$$

2. Determine Partial Derivatives

$$dZ^{[2]} = (\hat{Y} - Y)$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$dB^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]})$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$
$$dB^{[1]} = \frac{1}{m} \sum_{rows} dZ^{[1]}$$

3. Update parameters

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$$
$$B^{[1]} = B^{[1]} - \alpha \cdot dB^{[1]}$$
$$W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]}$$
$$B^{[2]} = B^{[2]} - \alpha \cdot dB^{[2]}$$

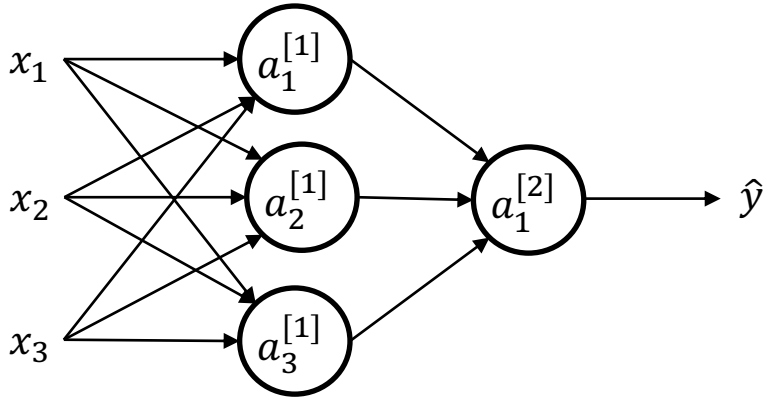
4. Repeat until J < target

Initializing Parameters in Neural Networks

- For Logistic Regression we were able to initialize the parameters to any starting value
- For convenience, we tended to simply initialize all parameters to 0 to get started
- However, this does **NOT** work with Neural Networks, why?

Initializing Parameters in NNs

- Let's work through a simple example:

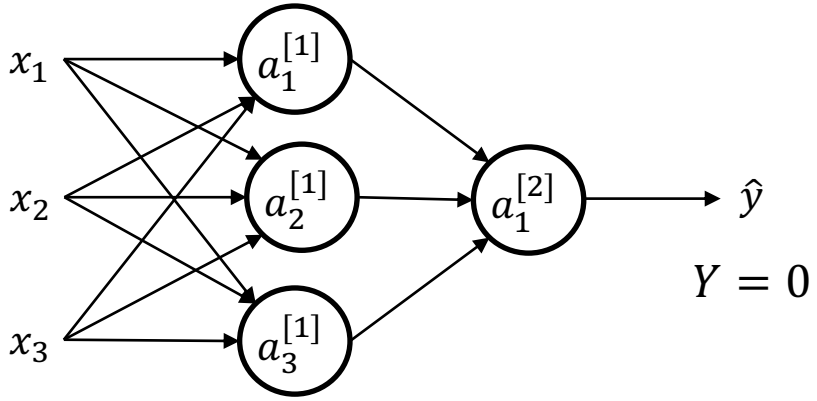


$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$



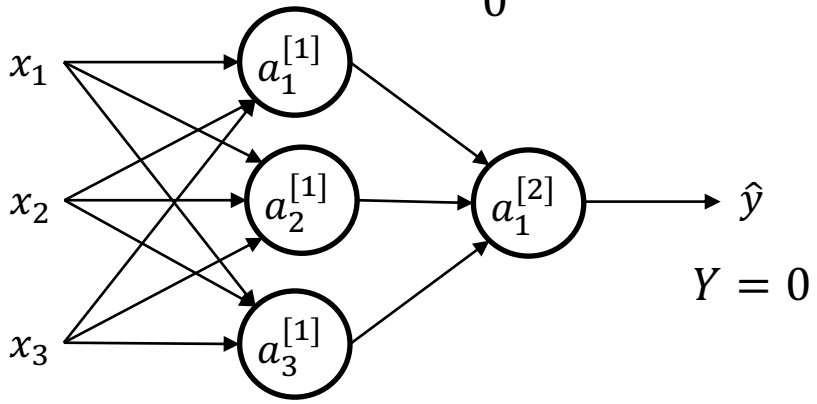
$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$



$Y = 0$

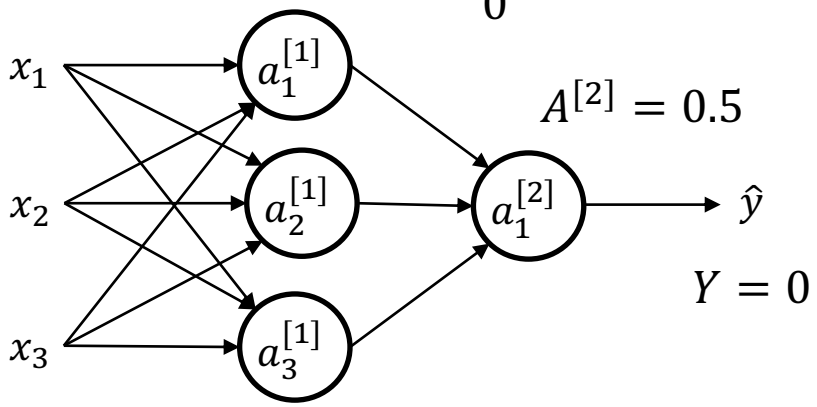
$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$



$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

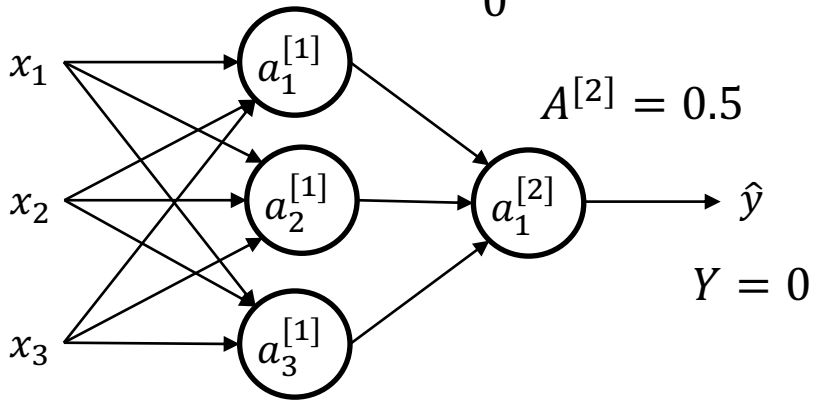
Initializing Parameters in NNs

- Let's work through a simple example:

$$dZ^{[2]} = \hat{Y} - Y = 0.5$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$



$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

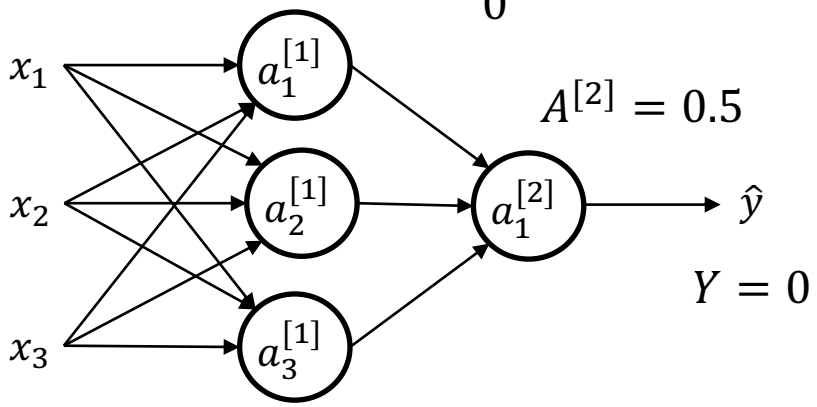
- Let's work through a simple example:

$$dZ^{[2]} = \hat{Y} - Y = 0.5$$

$$dW^{[2]} = dZ^{[2]} A^{[1]T} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$



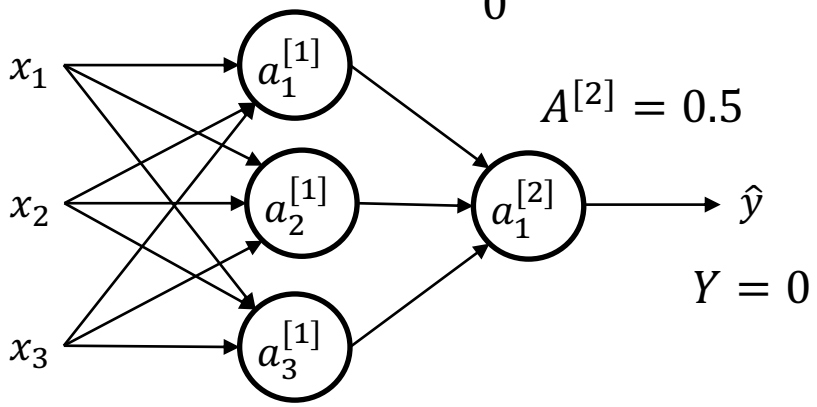
$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$



$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$dZ^{[2]} = \hat{Y} - Y = 0.5$$

$$dW^{[2]} = dZ^{[2]} A^{[1]T} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

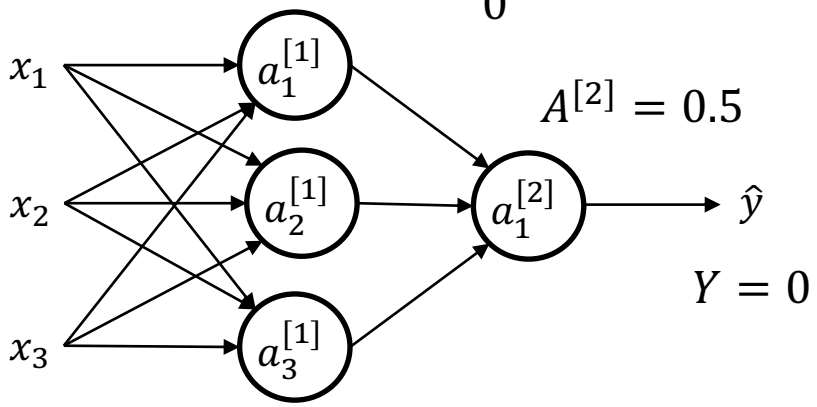
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \mathrm{g}'(Z^{[1]}) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$



$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$dZ^{[2]} = \hat{Y} - Y = 0.5$$

$$dW^{[2]} = dZ^{[2]} A^{[1]T} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \mathrm{g}'(Z^{[1]}) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

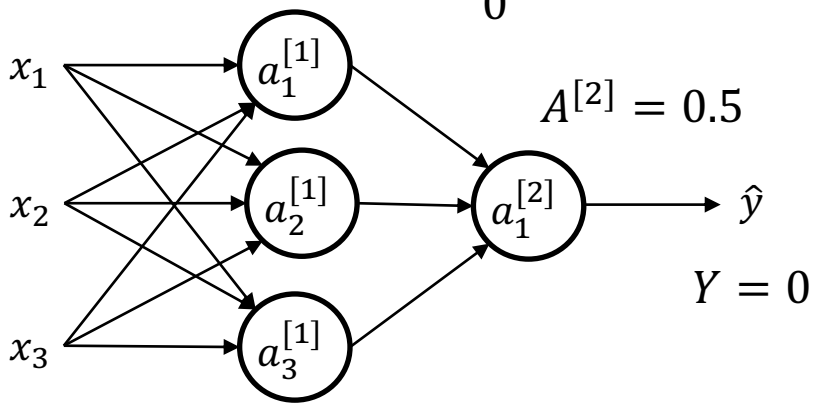
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- Let's work through a simple example:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

$$A^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$



$$W^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad W^{[2]} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$dZ^{[2]} = \hat{Y} - Y = 0.5$$

$$dW^{[2]} = dZ^{[2]} A^{[1]T} = \begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]}) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

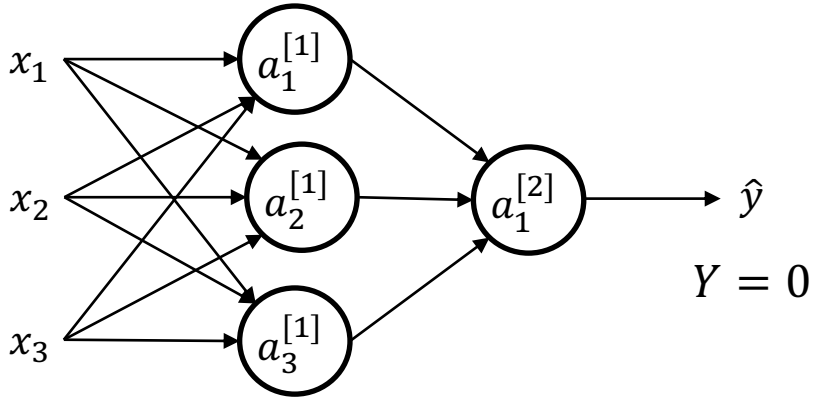
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Initializing Parameters in NNs

- What if we use a uniform non-zero value?:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

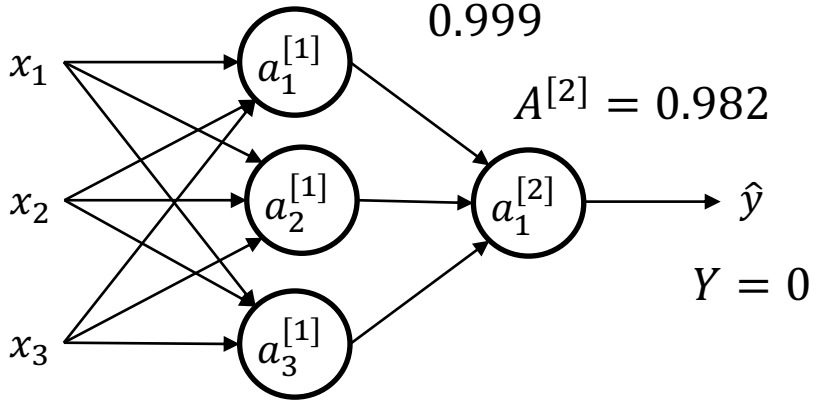


$$W^{[1]} = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \qquad W^{[2]} = \begin{matrix} 1 & 1 & 1 \end{matrix}$$

Initializing Parameters in NNs

- What if we use a uniform non-zero value?:

$$X = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$



$$A^{[1]} = \begin{matrix} 0.999 \\ 0.999 \\ 0.999 \end{matrix}$$

$$A^{[2]} = 0.982$$

$$Y = 0$$

$$W^{[1]} = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

$$W^{[2]} = \begin{matrix} 1 & 1 & 1 \end{matrix}$$

$$dZ^{[2]} = \hat{Y} - Y = 0.982 - 1$$

$$dW^{[2]} = dZ^{[2]} A^{[1]T} = \begin{matrix} 0.982 & 0.982 & 0.982 \end{matrix}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]}) = \begin{matrix} 3.266e-6 \\ 3.266e-6 \\ 3.266e-6 \end{matrix}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T = \begin{matrix} 3.266e-6 & 6.533e-6 & 9.799e-6 \\ 3.266e-6 & 6.533e-6 & 9.799e-6 \\ 3.266e-6 & 6.533e-6 & 9.799e-6 \end{matrix}$$

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]} = ?$$

Final Vectorized Backpropagation

0. Initialize the Parameters

$$W^{[1]}, W^{[2]}, B^{[1]}, B^{[2]} = rand()$$

1. Make an initial prediction

$$Z^{[1]} = W^{[1]}X + B^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[1]}$$
$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]})$$

2. Determine Partial Derivatives

$$dZ^{[2]} = (\hat{Y} - Y)$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$dB^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]})$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$
$$dB^{[1]} = \frac{1}{m} \sum_{rows} dZ^{[1]}$$

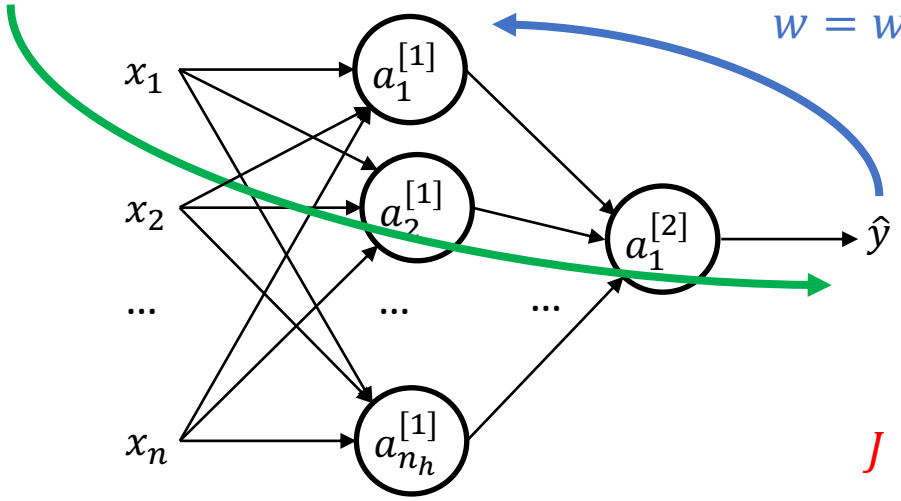
3. Update parameters

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$$
$$B^{[1]} = B^{[1]} - \alpha \cdot dB^{[1]}$$
$$W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]}$$
$$B^{[2]} = B^{[2]} - \alpha \cdot dB^{[2]}$$

4. Repeat until J < target

Last Lecture: Go Forwards, then Backwards…

Step 1: Calculate $\hat{y}$ using computation graph.



Step 3: Update *each* parameter (Using the partial derivative of cost).

$$w = w - \alpha \frac{\partial J}{\partial w}; b = b - \alpha \frac{\partial J}{\partial b}$$

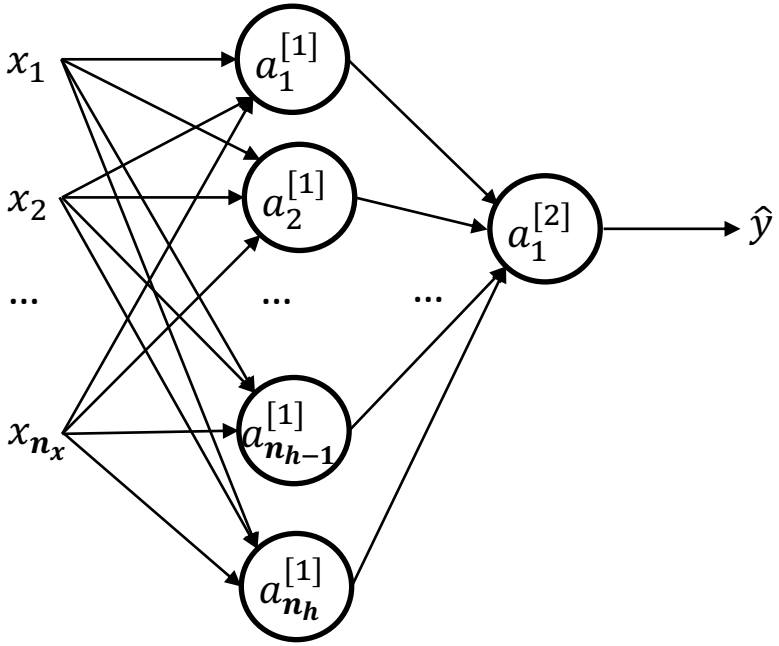
Step 2: Determine the loss.

$$J = -\frac{1}{m}\left(\sum_{i=1}^{m} y^i \log\left(\hat{y}^{(i)}\right) + \sum_{i=1}^{m}\left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right)$$

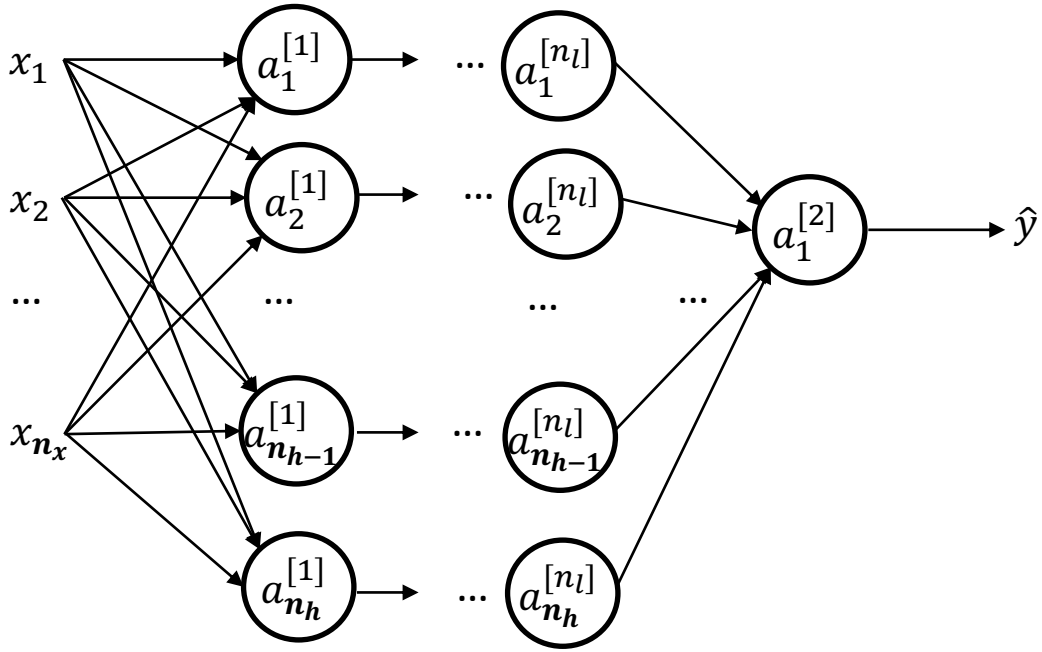
Step 4: Repeat until J < target.

How Many Hidden Units/Layers?

- As we have defined it, the Neural Network architecture is extremely flexible. It is possible to define any number of hidden layers and any number of units/layer and everything we have derived will workout



$n_h > n_x$



$n_h > n_x$ & large n_l

$etc.$

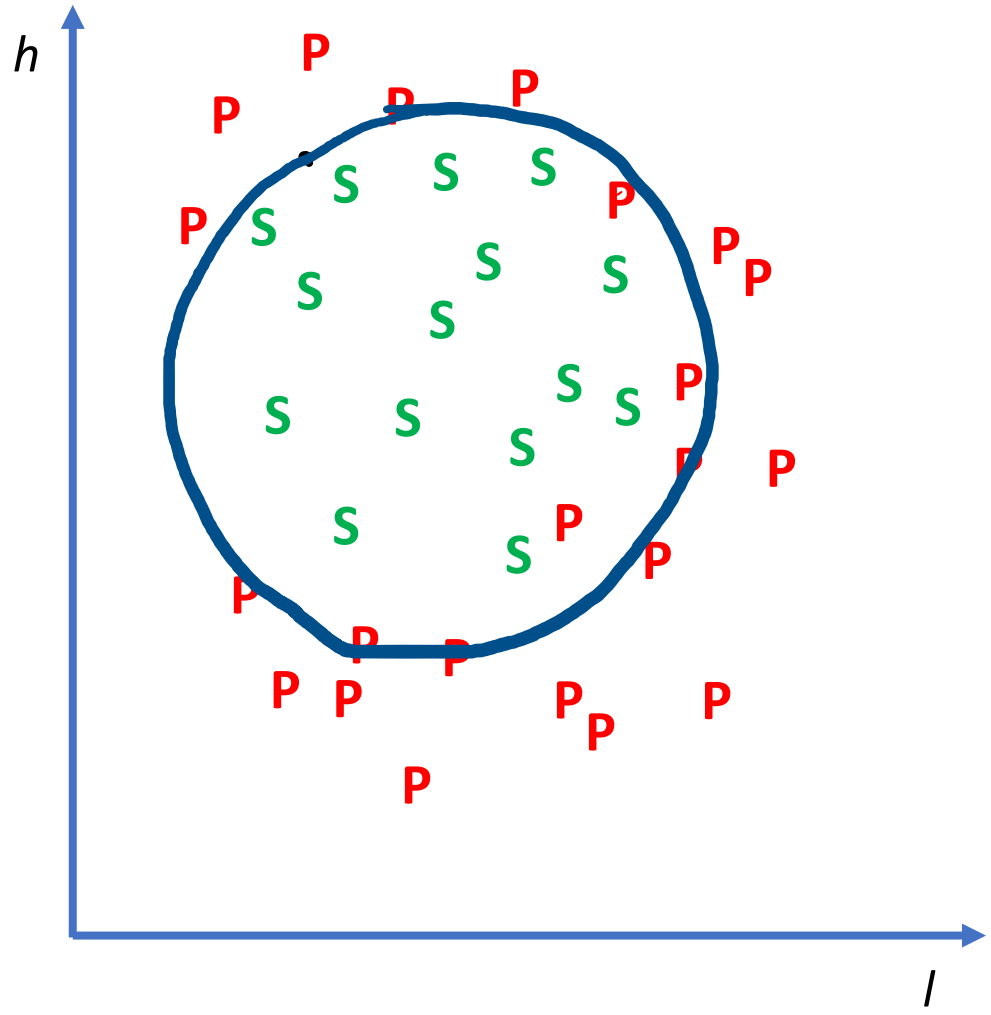
How Many Hidden Units/Layers?

- However, more units are **NOT** necessarily better!
- There is a cost in terms of training and deployment computing resources/time
- Further, extra units contribute to the problem of **overfit...**

Overfit In Neural Networks

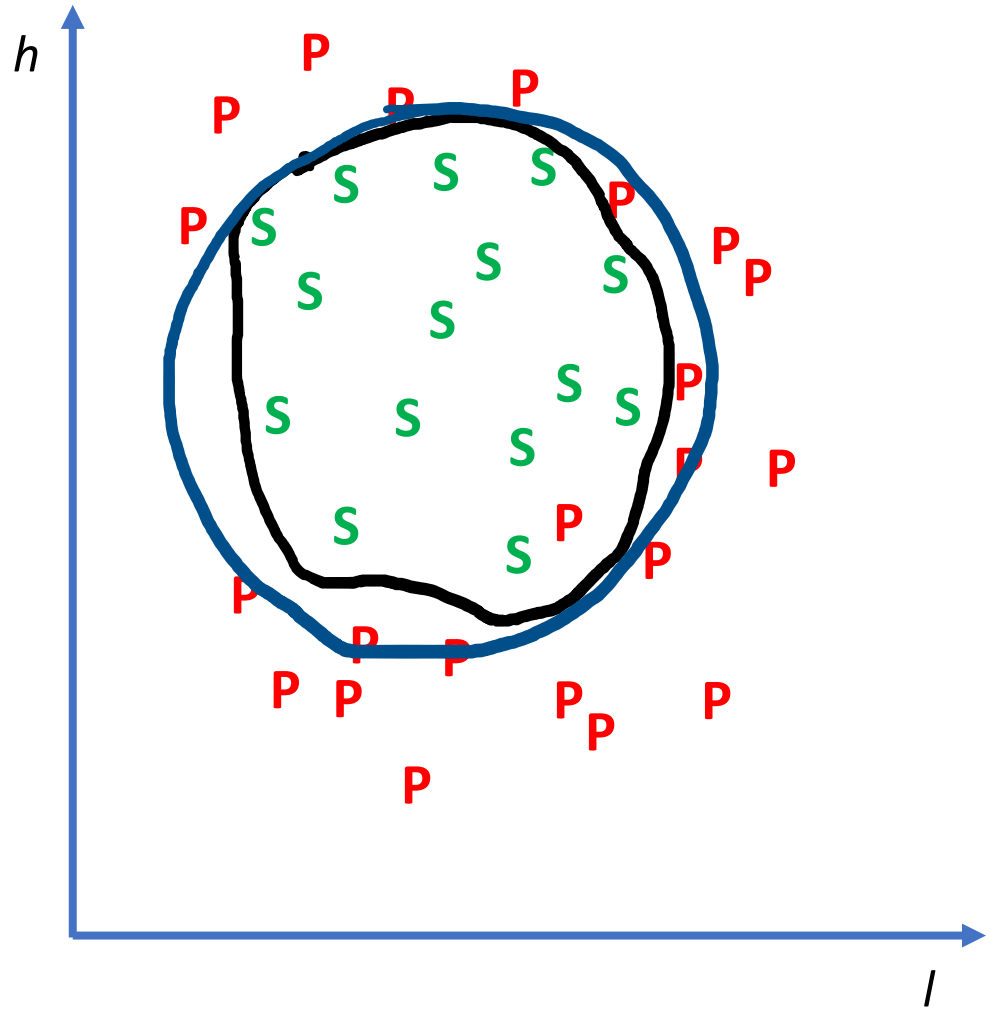
- Neural Networks can have A LOT of flexibility (lots of units, lots of layers, lots of parameters)
- This can be very powerful when the underlying relationship is very complex and/or there are a lot of input features to consider
- However, it also leads to the critical issue of **overfit**
- In a nutshell: “It is possible the Neural Network finds an approximate function that ‘fits’ the training data, but does not generalize to new data”

Overfit In Neural Networks



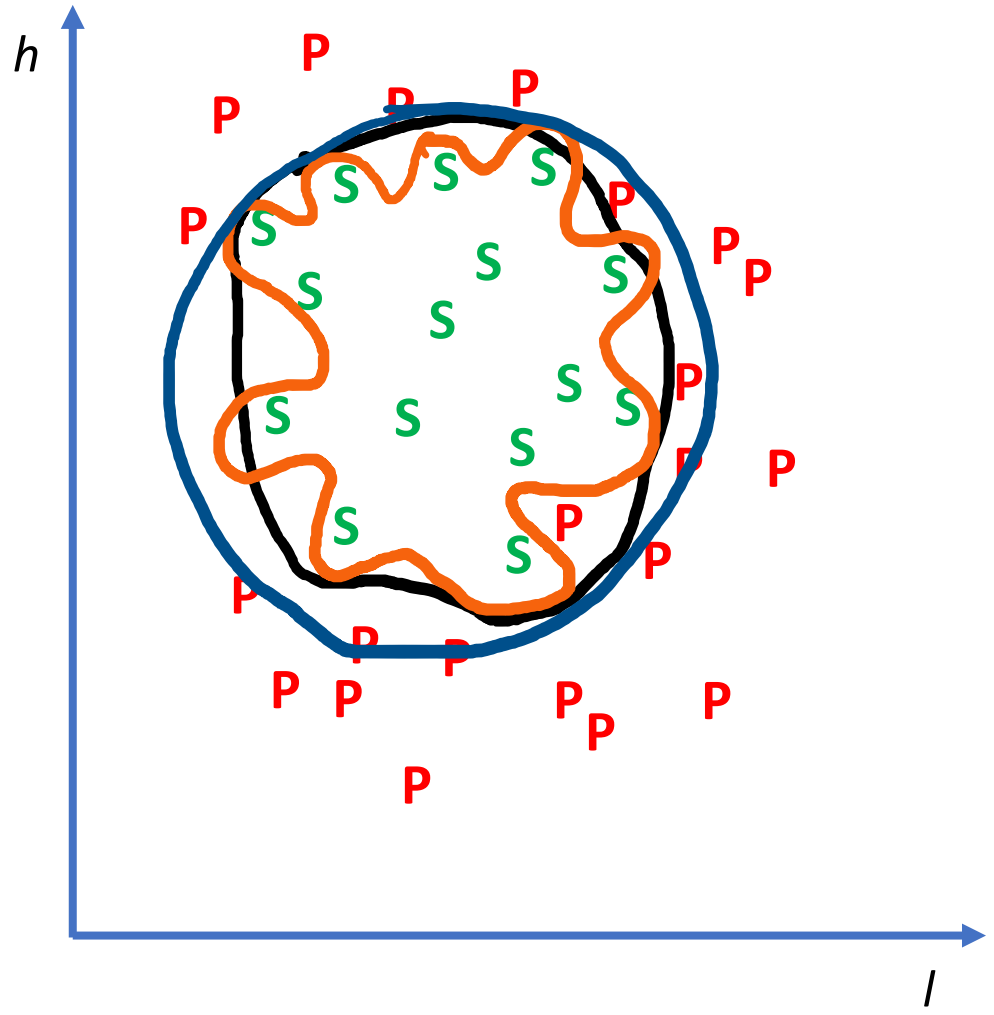
Brad Quinton, Scott Chin

Overfit In Neural Networks



Brad Quinton, Scott Chin

Overfit In Neural Networks



- Looking at just the training data set, it is not clear which is the “best” answer.
- Remember the ”best” answer is the one that is to most accurate on **NEW DATA**
- A learned solution that track too close to the training data risks missing the “big picture” and simply memorizing training data

Overfit In Neural Networks

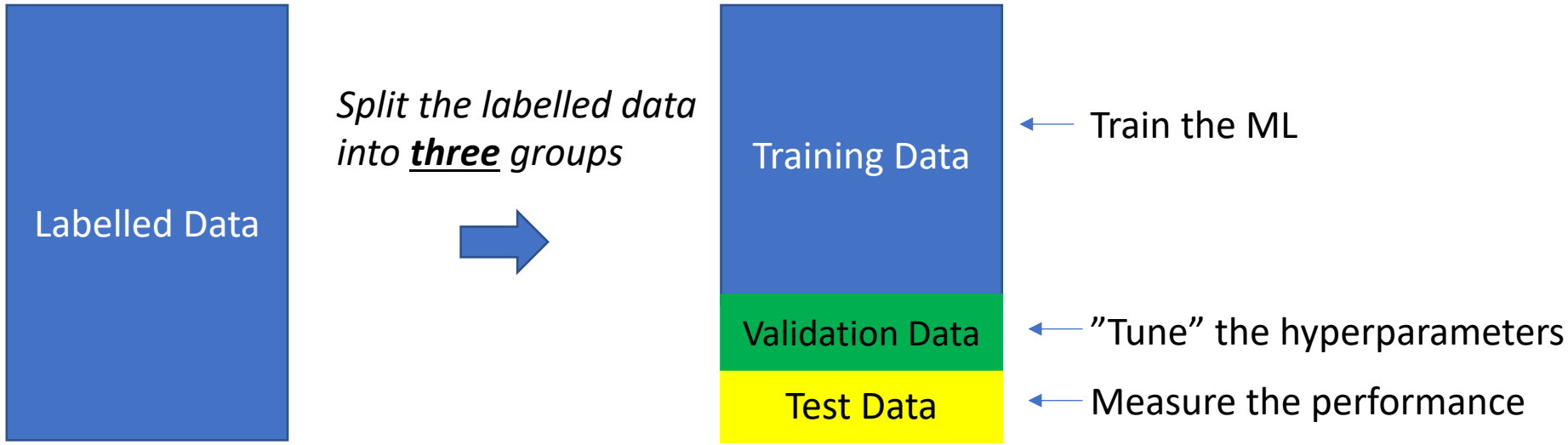
- Because they effect the overall performance of the AI, the number of hidden layers and the number of units/layer become **hyperparameters** that need to be ”tuned” to achieve optimal performance

- It is important to be able to measure the effect of different hyperparameter selections independently of the evaluation of the performance of the network

Validation Data Set

- Recall that to evaluated Logistic Regression we split our labelled data into two set: test and training

- Because of the problem of overfit, for NNs we need 3 data sets



Validation Data Set

- Why do we need the third data set? Because otherwise we lose the ability to have measure our expected performance

- Consider this scenario:

1. We achieve 98% training accuracy and 87% validation accuracy.
2. Potentially we are “overfit”, so we adjust the NN architecture.
3. We then achieve 97% training accuracy and 93% validation accuracy.
4. Better, so we adjust the architecture again.
5. Now we achieve 97% training accuracy, and 95% validation accuracy.
6. Is it correct to report 95% as the expected accuracy of our AI?

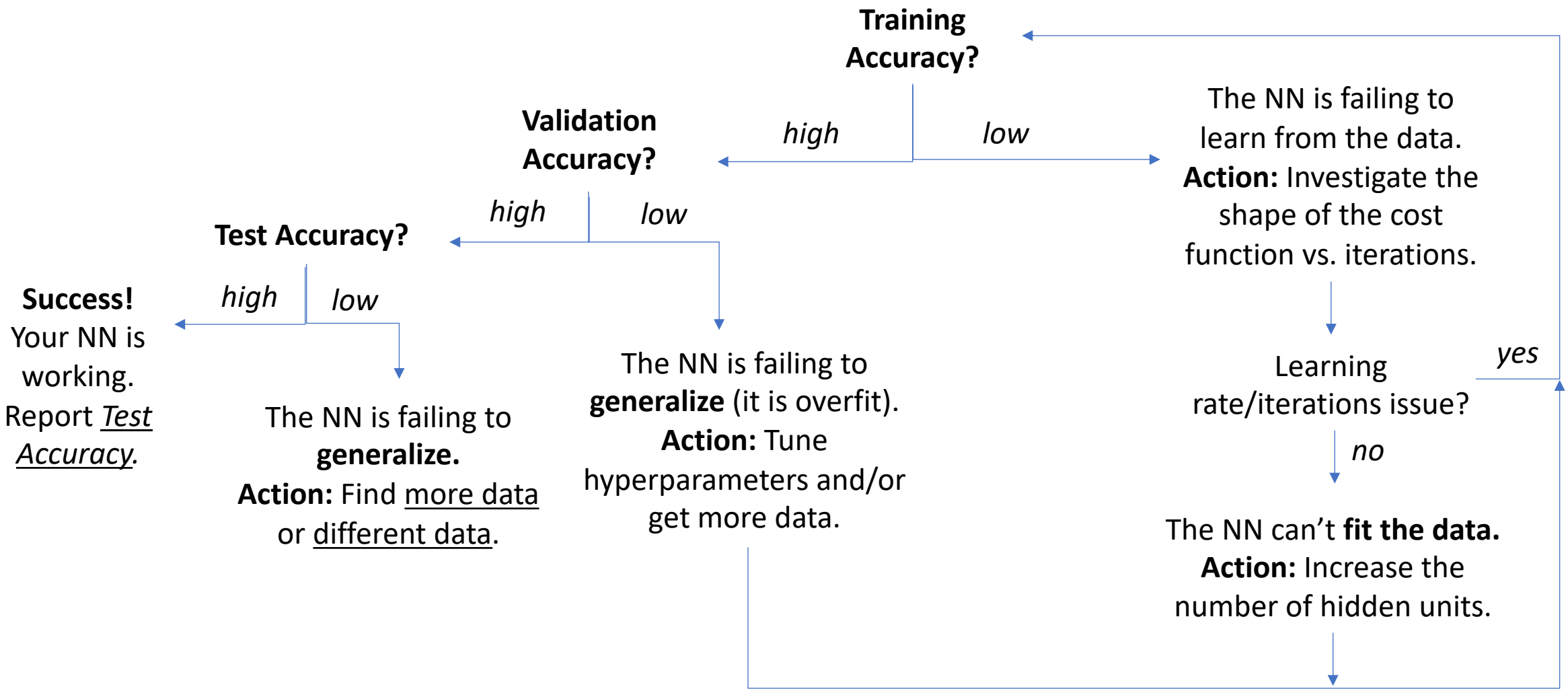
Validation Data Set

- The answer is: **“No!”** We are adjusting the NN architecture specifically to improve our validation accuracy, so there is a danger we are simply fitting our hyperparameters to the validation set

- So the validation data set gives us data that was not used to train to NN, but can be used to tune the hyperparameters

- The test data set then gives us our independent reference to measure the performance of the AI

Neural Network ML Decision Tree



Brad Quinton, Scott Chin

Why is this Exciting?

- We can now build, train and evaluate Neural Networks, *so what?*

- The exciting thing is that we now have a system that can learn complex, non-linear relationships between an arbitrary number of features across an arbitrary number of examples

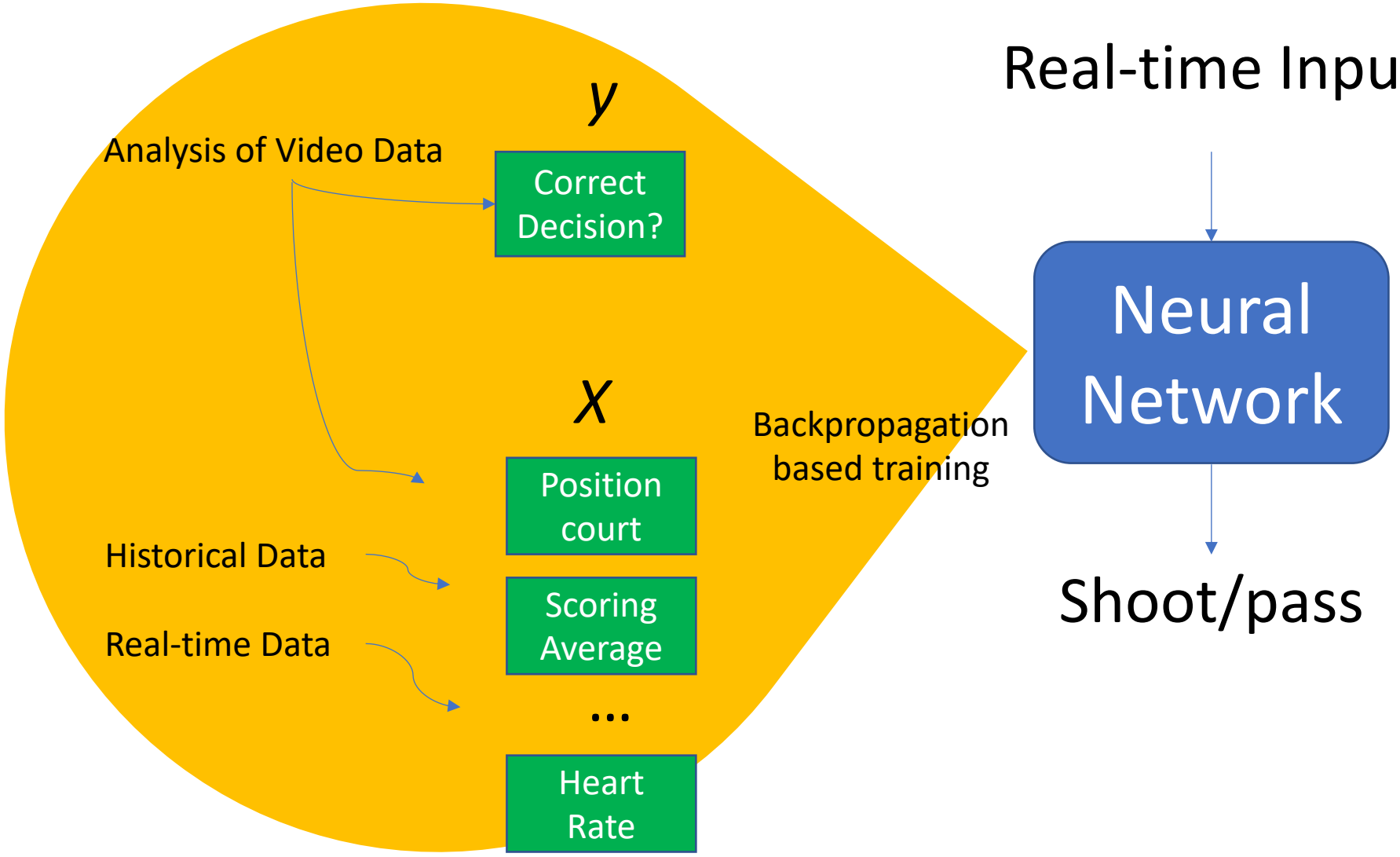
- In the era of "Big Data" this is very compelling. We often have a lot of data. Some of features of the data may or may not be relevant, other features will may be important is non-obvious ways… but with NNs we do not need to determine this ahead of time to get value from the data!

Back to our Case Study….

- Not only can we deal with the initial problem of fitting non-linear relationships, but now we can easily add almost any data we want to make help make a more accurate prediction

- We can start to think: “What might matter?”
  - Position on the court?
  - Average shooting accuracy *this season*?
  - Average shooting accuracy *this game*?
  - Average shooting accuracy against *this team*?
  - Current heart rate?
  - Height, weight, wingspan, time of day, … ?

Case Study: Predicting the Right Action



Brad Quinton, Scott Chin

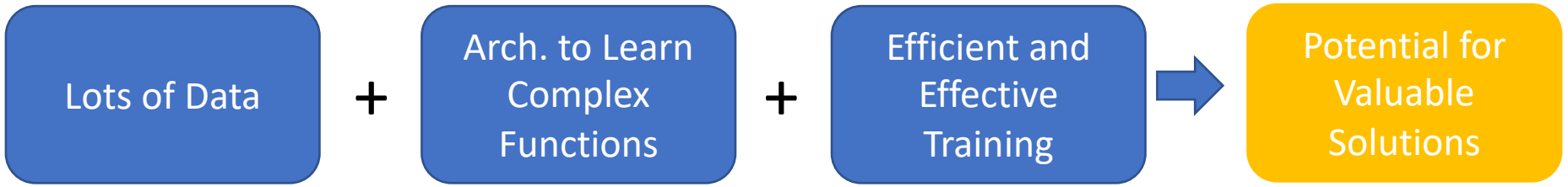
Case Study #2: Second Meeting as Director of AI

- You: *“Hey Coach. Great news! I have a framework that can learn almost to relate almost any data we can get to the question of Pass or Shoot!”*
- Coach: *“Really? I was mostly just toying with you because you are new… I didn’t think it was actually possible. Wow.”*
- You: *“Sure, Neural Networks are great at learning data relationships. Even ones that are far to complex for humans to understand.”*
- Coach: *“That’s great. Why don’t you build something and we can think of how to test it…”*

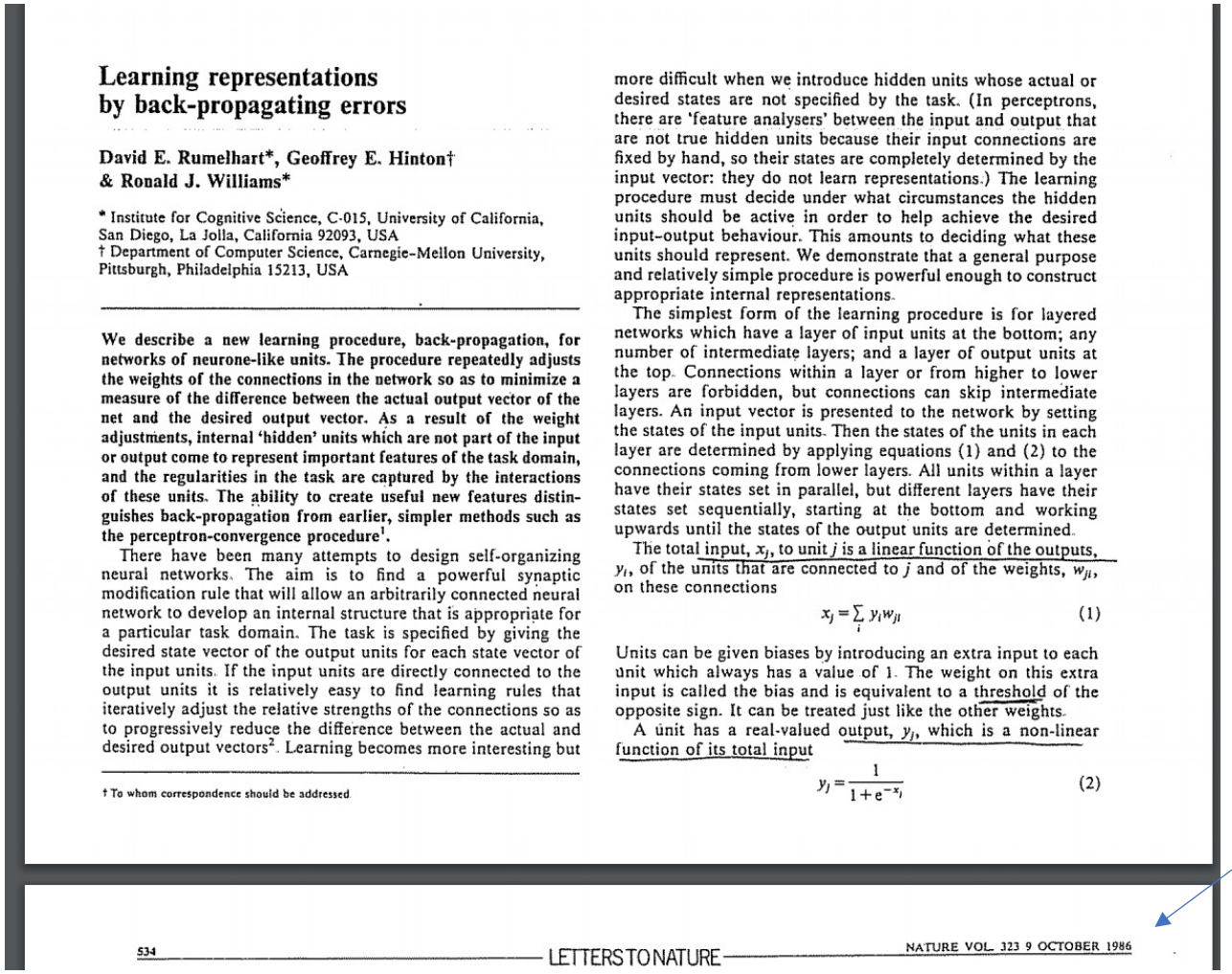
Historical Context

- Now that we understand how backpropagation, it is worth thinking a bit about the historical context

- As we discussed previously, Deep Learning is possible because backpropagation continues to be efficient and effective even for very large Neural Networks



Original Paper On Backpropagation



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

1986!

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Interesting Article/Quotes About Backprop

“The breakthrough of backpropagation was the next step: the algorithm uses partial derivatives to apportion "blame" for the wrong output among the neuron's inputs. ….

…backpropagation radically expanded the scope of trainable neural networks. People were no longer limited to simple networks with one or two layers. They could build networks with five, ten, or fifty layers, and these networks could have arbitrarily complex internal structures.”

https://arstechnica.com/science/2019/12/how-neural-networks-work-and-why-theyve-become-a-big-business/

Interesting Video About IBM Watson

- Nova: “Smartest Machine on Earth - Watson the Computer Versus Jeopardy”
- https://youtu.be/4svcCJJ6ciw

- Discussion of ML 18:00 to ~21:00

- Why was ML needed for Jeopardy?

Key Take-Aways

- Vectorization of the backpropagation algorithm enables efficient training of Neural Networks
- Because of Neural Networks have a lot of flexibility to build complex functions they are proven to overfit
- It is important to use a validation and test data set when tuning NN hyperparameters
- Neural Networks are well suited to processing complex and diverse input data