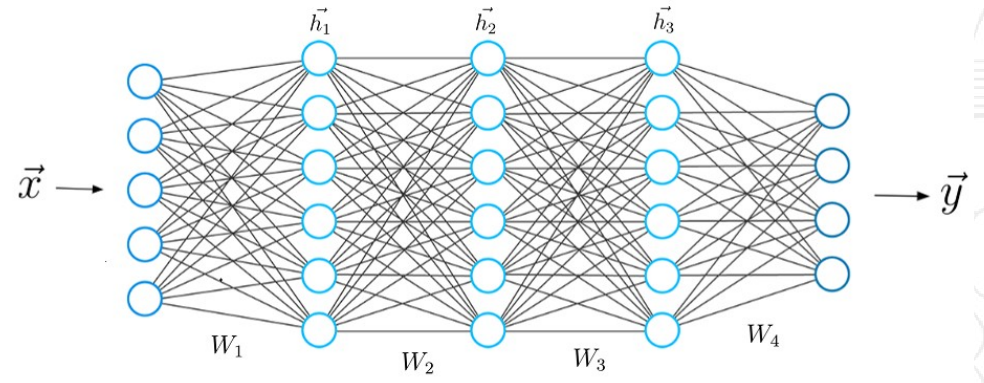
# Module 3

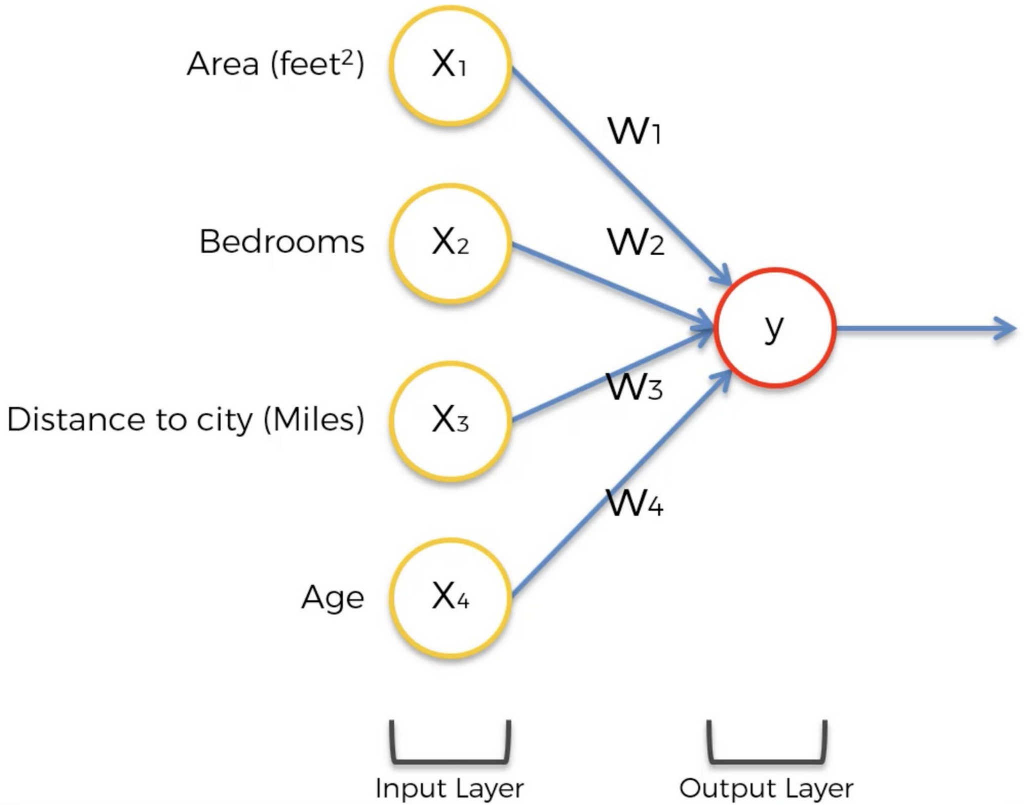
Deep Learning: Feed-Forward Architectures

# DL: Feed-Forward Architectures



# The Forward Pass

- Input nodes ▻ hidden nodes ▻ output nodes
- This is called the forward pass



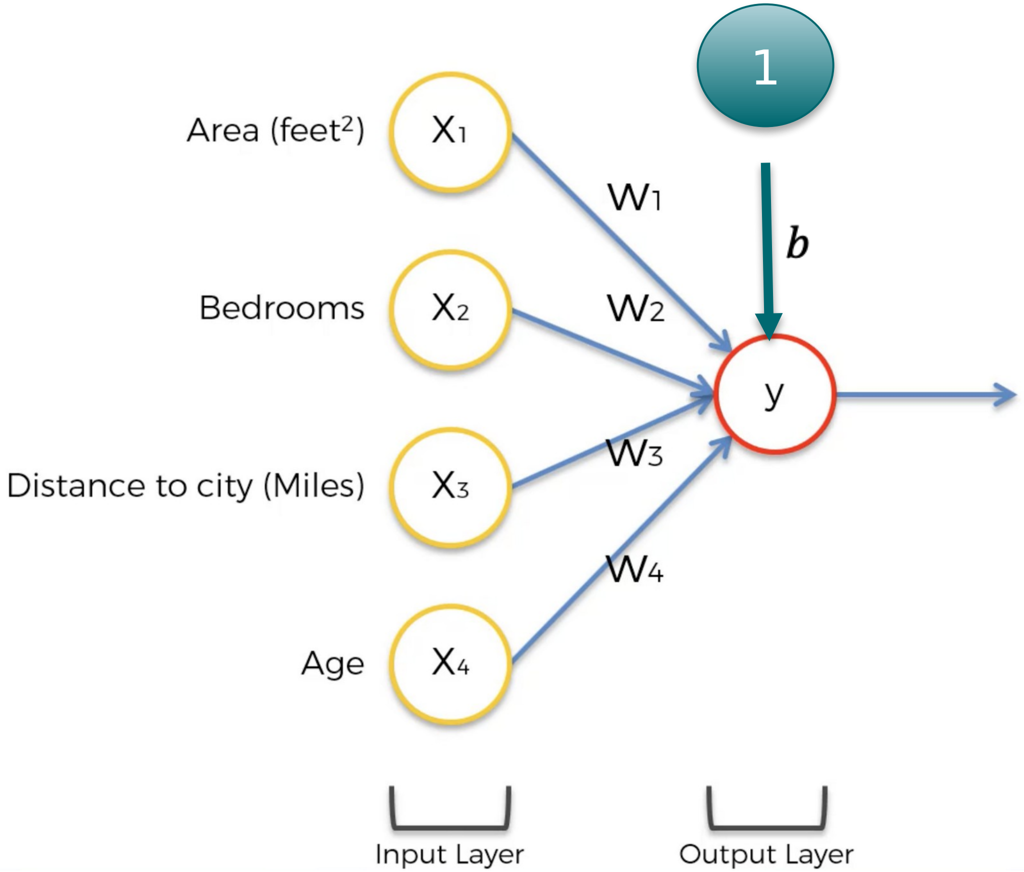

$$\Sigma \longrightarrow w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$$

$$\boldsymbol{w}^{\top}\mathbf{x}$$

$$f \longrightarrow f(\boldsymbol{w}^{\top}\mathbf{x})$$

# The Forward Pass

- Input nodes ⊳ <u>hidden nodes</u> ⊳ output nodes
- This is called the forward pass



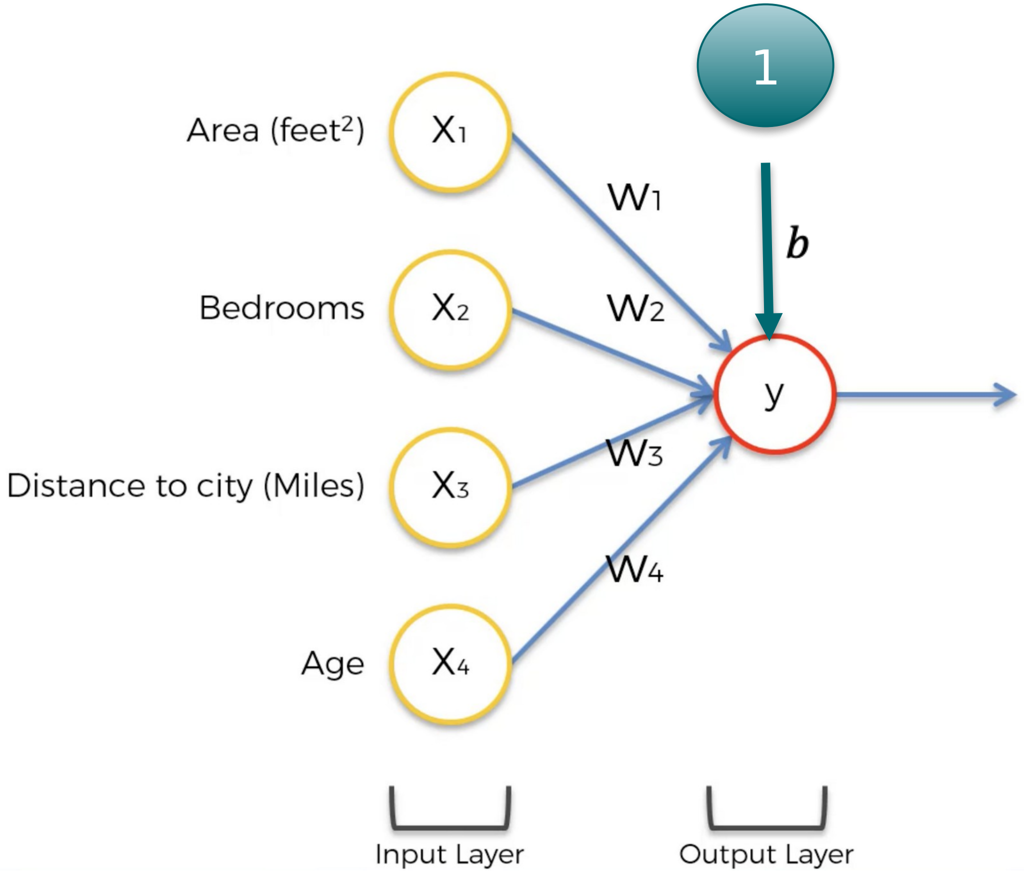

$$\Sigma \longrightarrow b + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$$

$$\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}$$

$$f \longrightarrow \boldsymbol{f}(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$$

# The Forward Pass

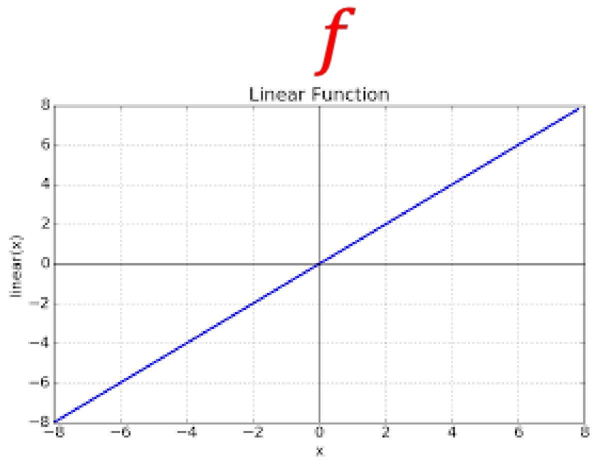
- Input nodes ⊳ hidden nodes ⊳ output nodes
- This is called the forward pass




$\Sigma \longrightarrow \quad 0 + .32x_1 + 0.3x_2 - 0.21x_3 + .002x_4$

$\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}$

$f \longrightarrow \quad \boldsymbol{f}(0 + .32x_1 + 0.3x_2 - 0.21x_3 + .002x_4)$



# Parameter Initializers

- Setting initial values of model's weights and biases before training
  - Zero initializations
    - Advantages: computational complexity,
    - Disadvantages: all neurons learning similar features (symmetry learning)
  - Random Initialization
    - Normal
    - Uniform
    - Advantages: random initialization avoiding symmetry,
    - Disadvantages: if values are too large/small, issues with sigmoid and tanh activations
  - Xavier Glorot (tanh, sigmoid), He (ReLu), Lecun (sigmoid, tanh normalized)
  - Orthogonal (RNNs)

$$\Sigma \longrightarrow 0 + .32x_1 + 0.3x_2 - 0.21x_3 + .002x_4$$

$$\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}$$

$$f \longrightarrow \boldsymbol{f}(0 + .32x_1 + 0.3x_2 - 0.21x_3 + .002x_4)$$

**Weights (kernels)**

$$\boldsymbol{w} = \begin{bmatrix} 0.320 \\ 0.300 \\ -0.210 \\ 0.002 \end{bmatrix}$$

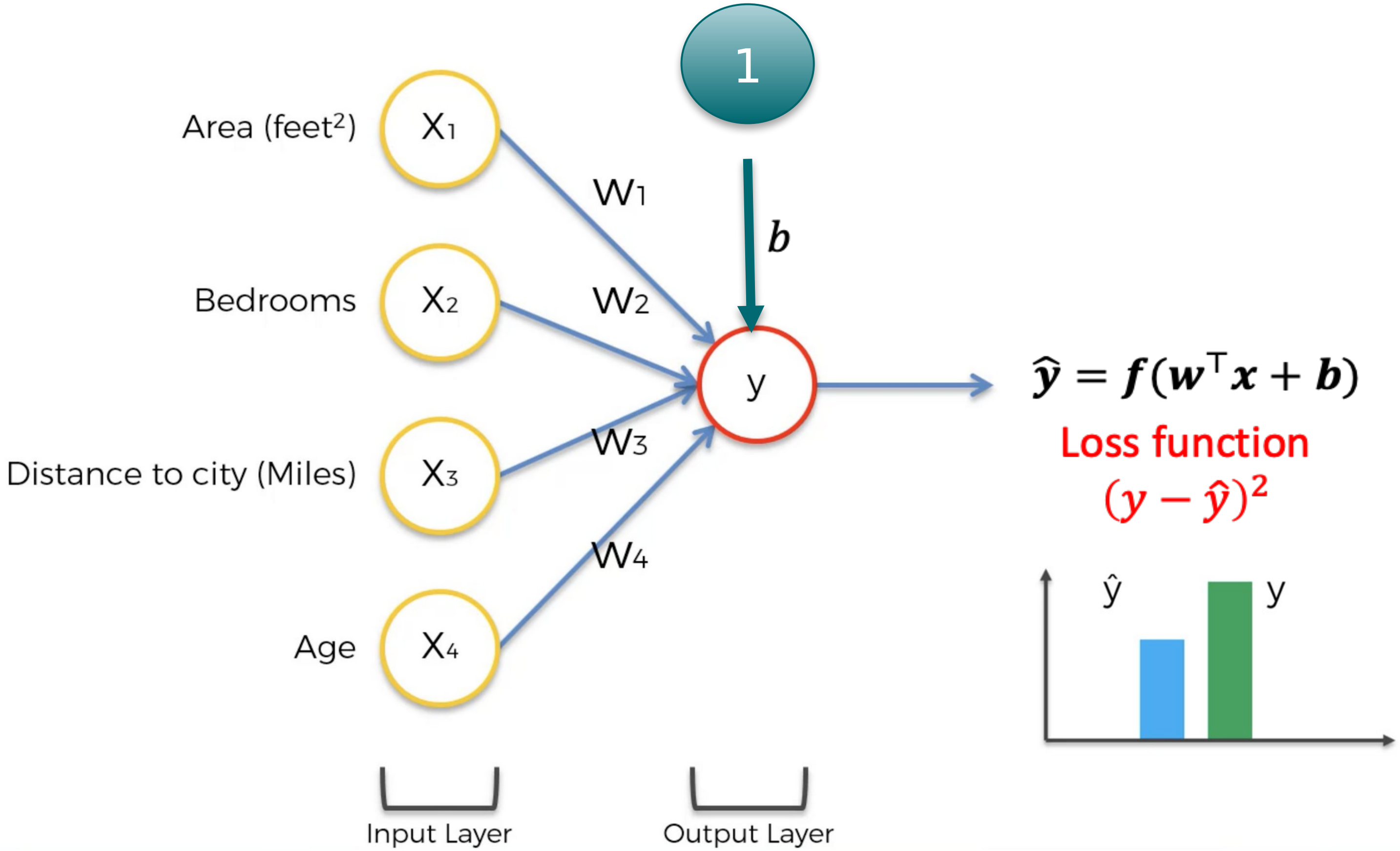
**Biases (intercepts)**

$$\boldsymbol{b} = [0]$$

# Python

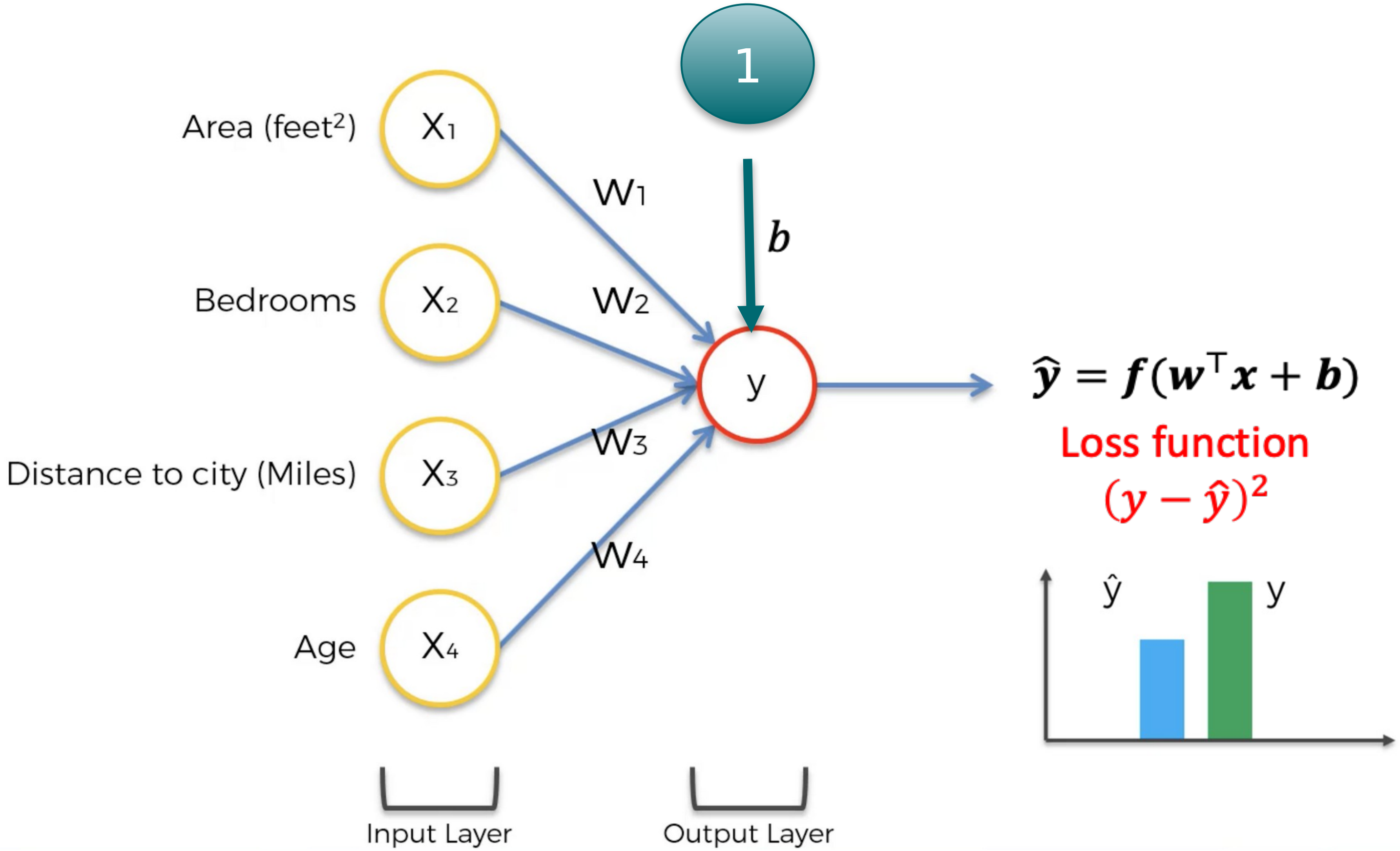
# The Forward Pass

- Input nodes ▻ hidden nodes ▻ output nodes
- This is called the forward pass



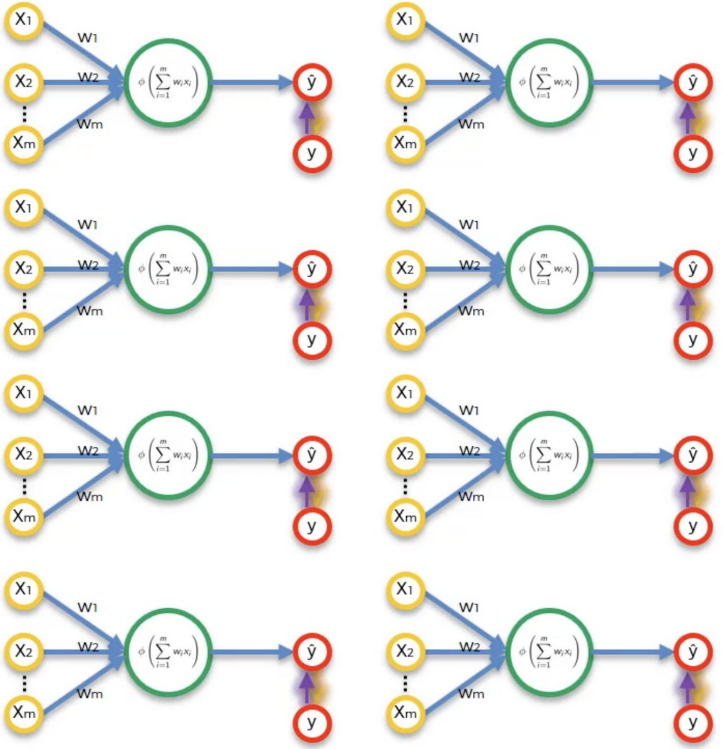
# The Forward Pass

- Input nodes ▻ hidden nodes ▻ output nodes
- This is called the forward pass



# The Forward Pass

- Input nodes ⊳ hidden nodes ⊳ output nodes
- This is called the forward pass

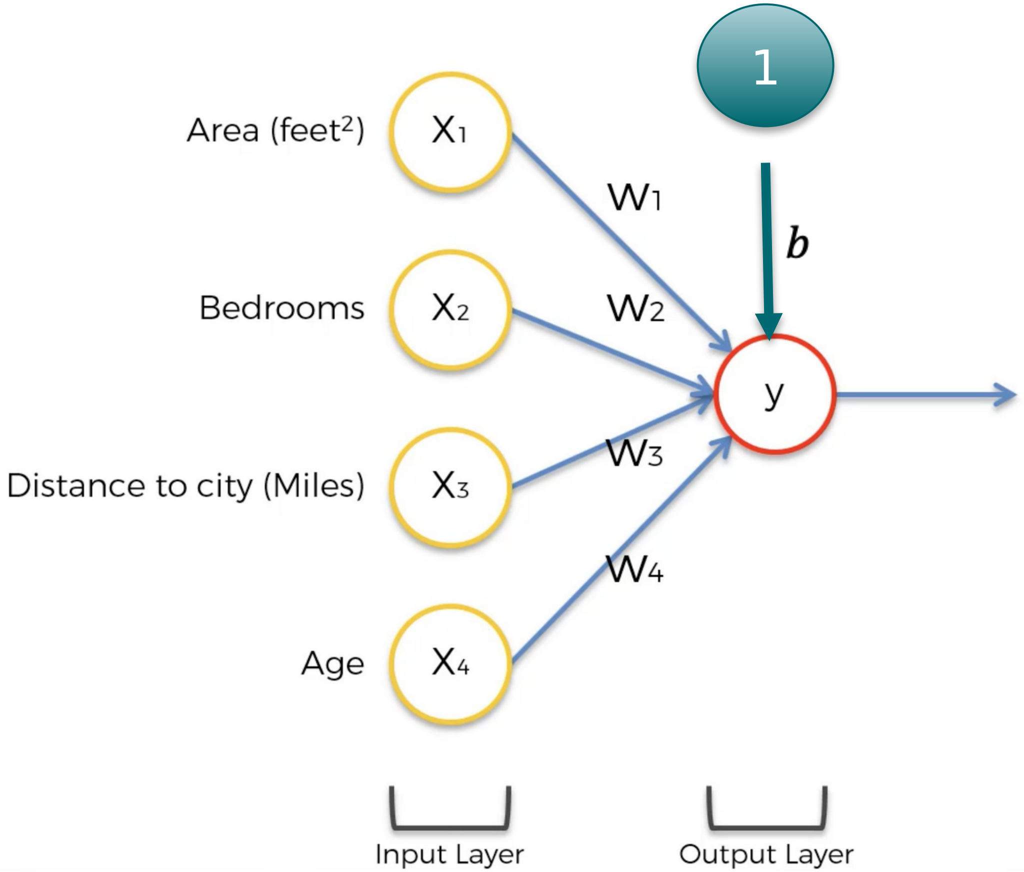



| | Bedrooms | Area | City_Distance | Age | Price |
|---|---|---|---|---|---|
| **0** | 1 | 26.184098 | 1286.68 | 67 | 96004.804557 |
| **1** | 1 | 34.866901 | 1855.25 | 30 | 92473.722570 |
| **2** | 1 | 36.980709 | 692.09 | 24 | 98112.519940 |
| **3** | 1 | 17.445723 | 1399.49 | 66 | 92118.326874 |
| **4** | 1 | 52.587646 | 84.65 | 3 | 98976.653176 |

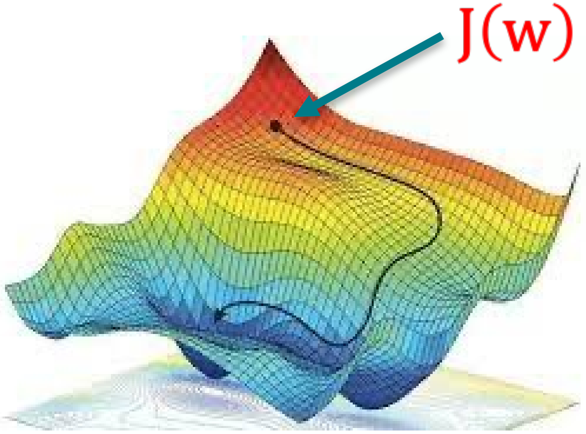

$$J(w) = \frac{\sum_{i=1}^{n}(y-\hat{y})^2}{n}$$

# The Forward Pass

- Input nodes ⊳ hidden nodes ⊳ output nodes
- This is called the forward pass



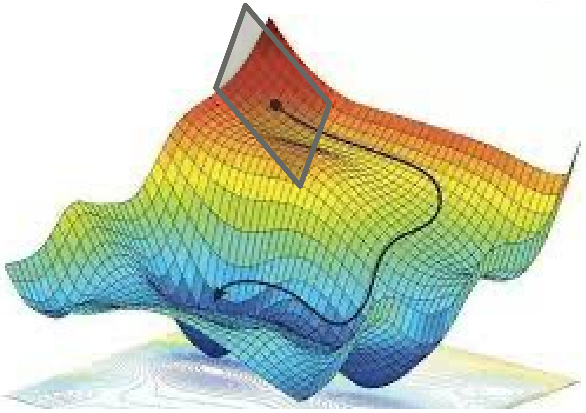

$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

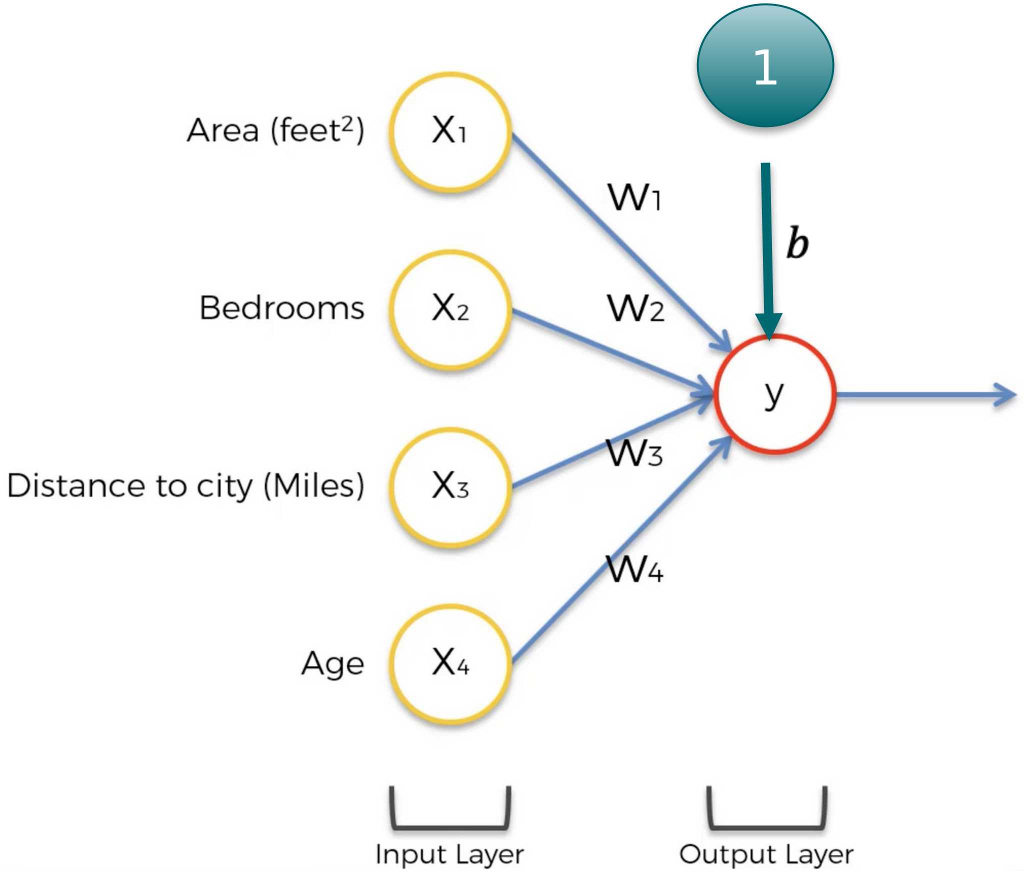



# The Forward Pass

- Input nodes ⊳ hidden nodes ⊳ output nodes
- This is called the forward pass

$$\nabla J(\mathbf{w}) = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - y)$$

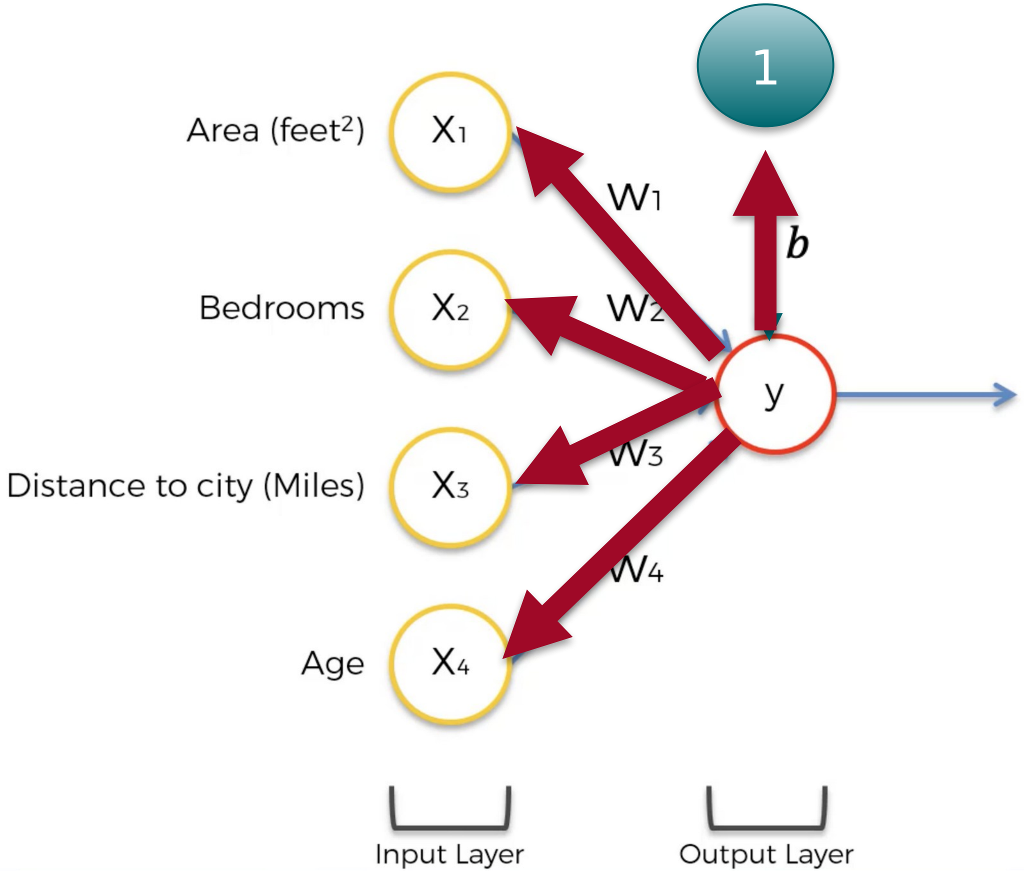




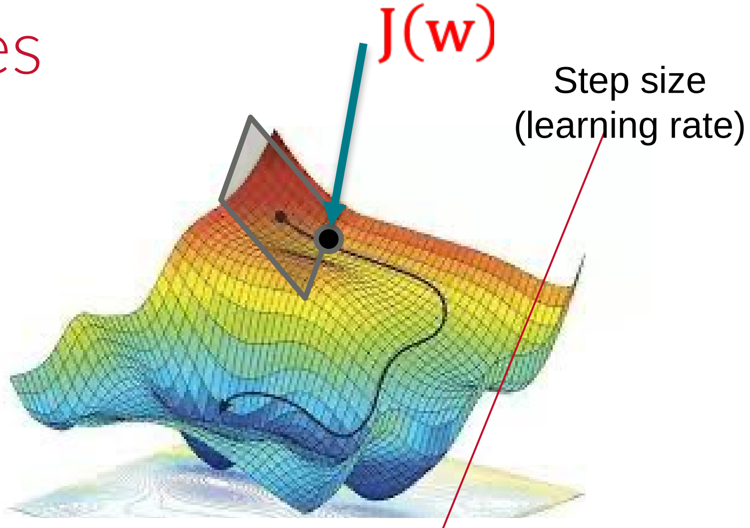

$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$$

# Backward Propagation

- Output nodes ⊳ hidden nodes ⊳ input nodes
- This is called the backward pass



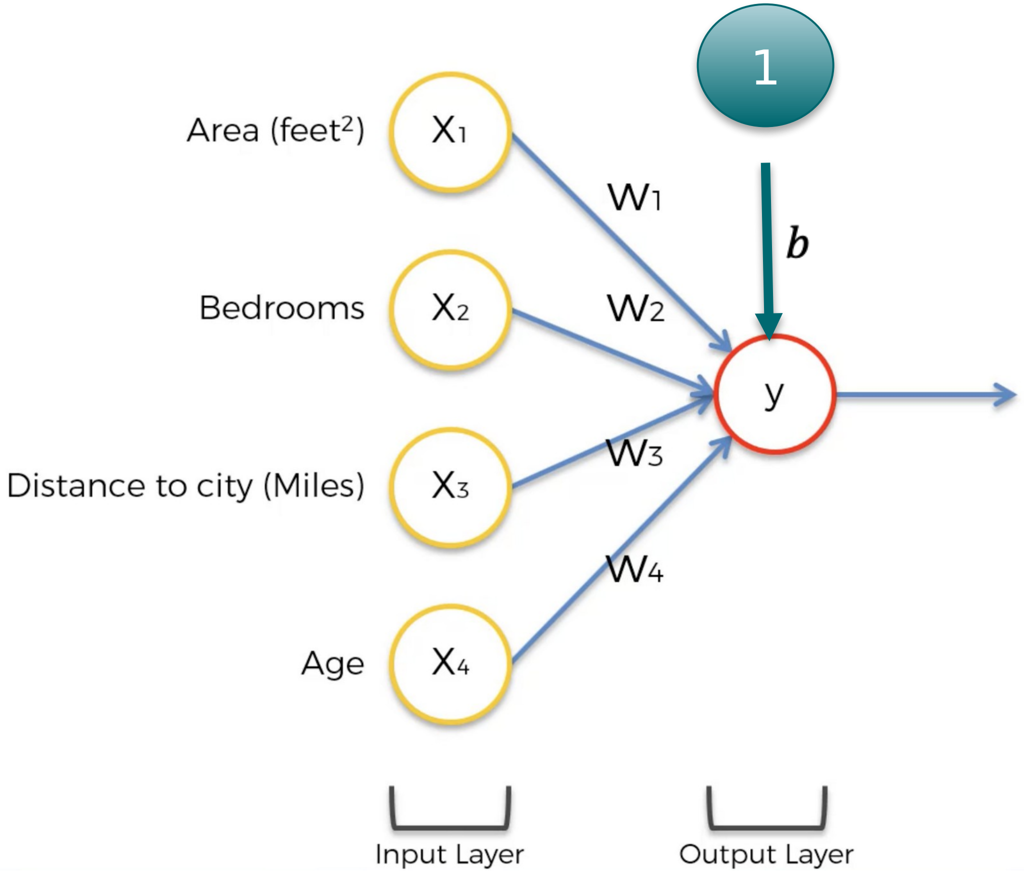

$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$$



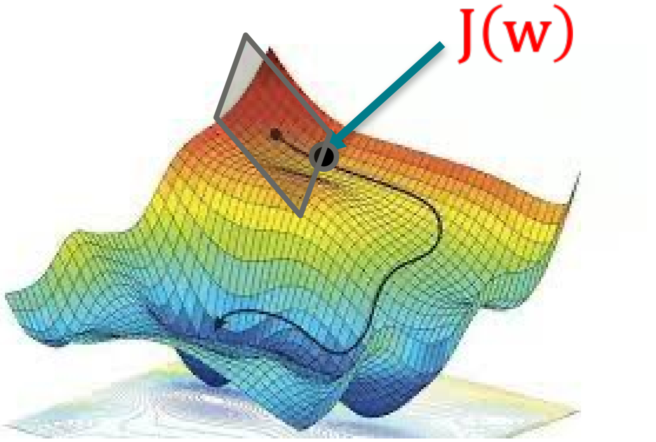

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$

# Forward Pass

- ❑ Input nodes ▻ hidden nodes ▻ output nodes
- ❑ This is called the forward pass

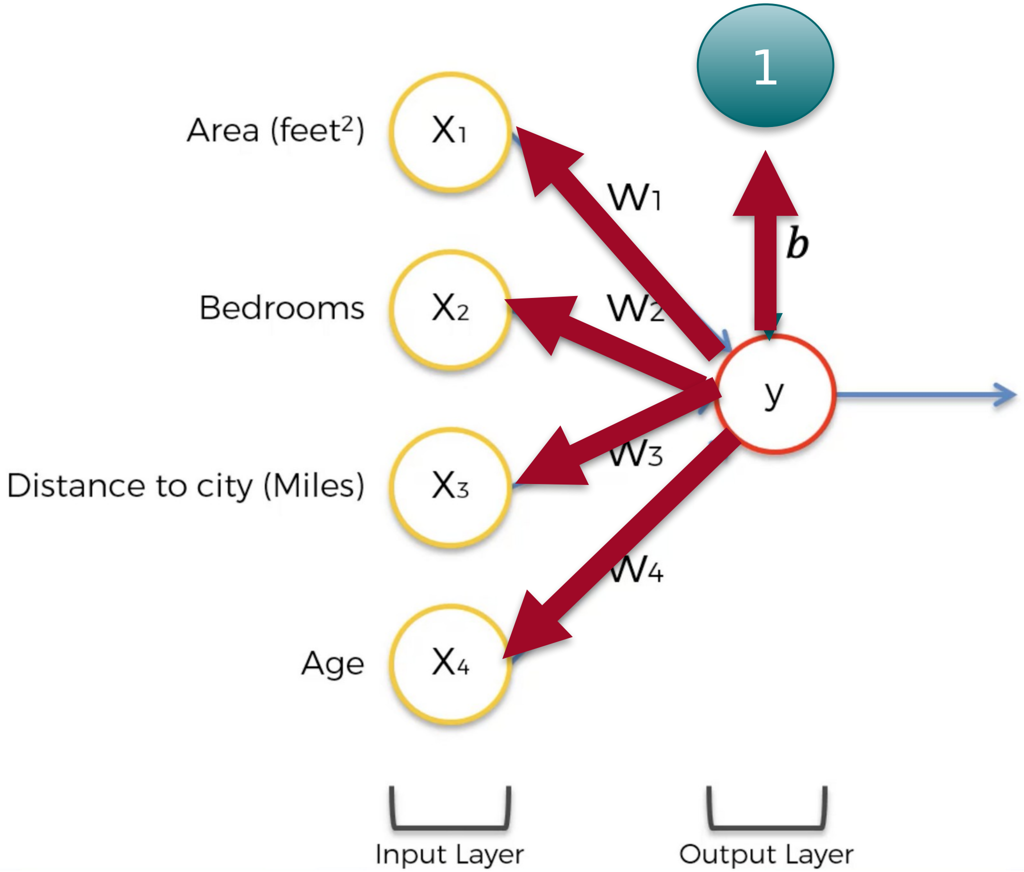



$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

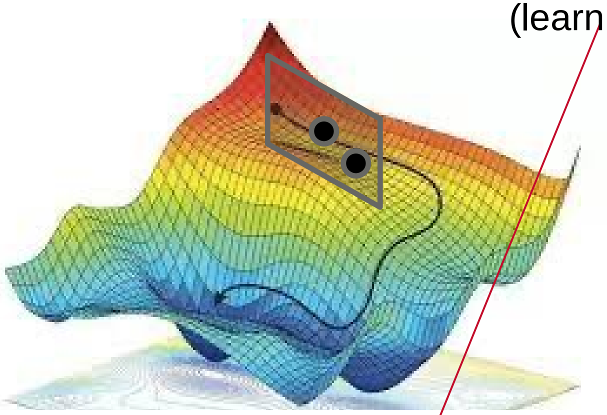



# Backward Pass

- Output nodes ▻ hidden nodes ▻ input nodes
- This is called the backward pass




$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

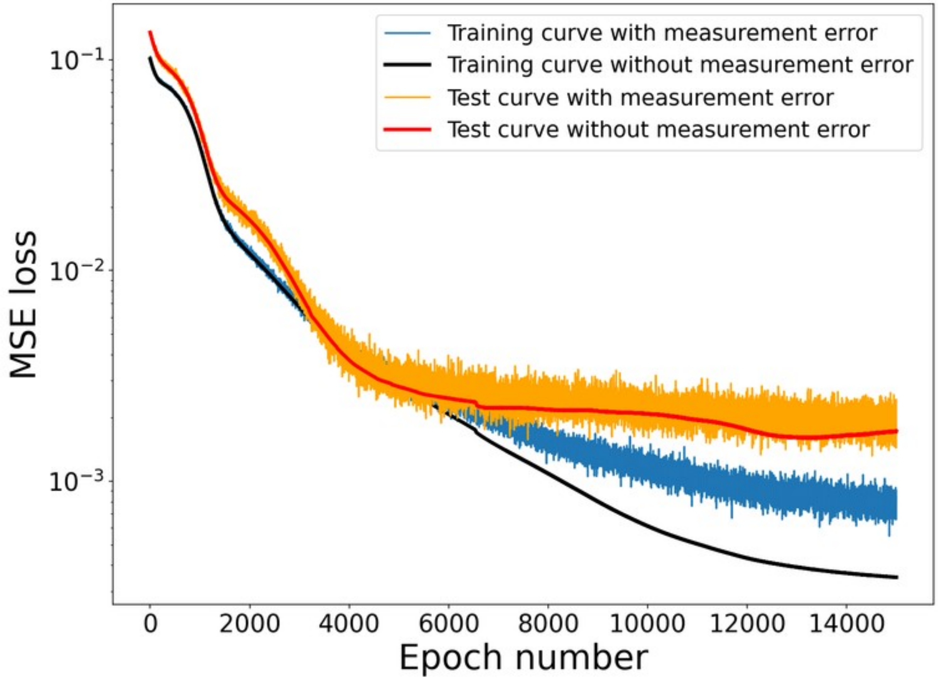


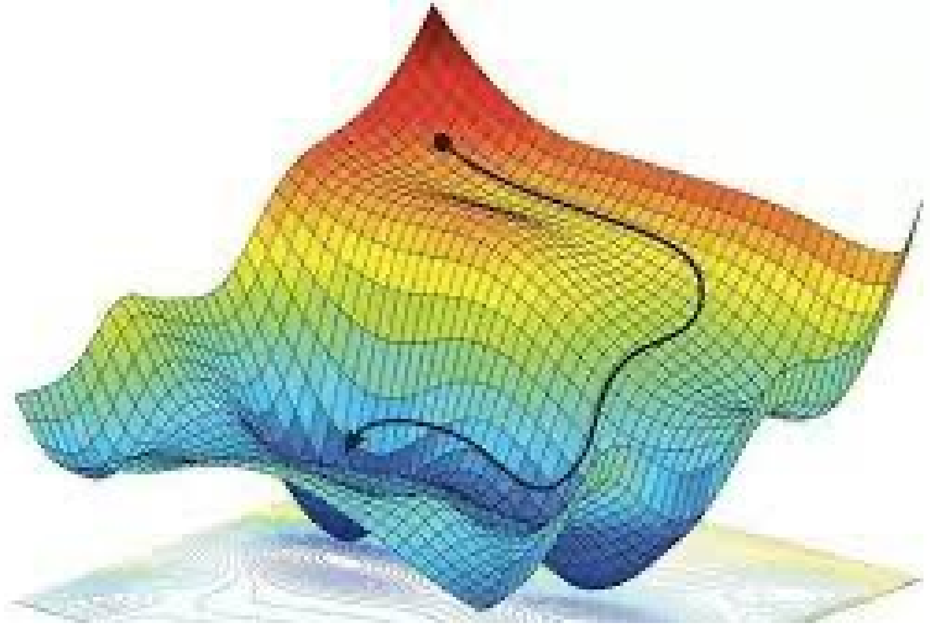
Step size
(learning rate)

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$

# Learning Curve

- Output nodes ▻ hidden nodes ▻ input nodes
- This is called the backward pass



slido

Please download and install the Slido app on all computers you use

**Consider the role of the learning rate in training a neural network. Which of the following statements best describes the effects of choosing a learning rate that is either too small or too large?**

☐ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

**Given a learning curve of the training set as given below, what is likely to be case in the training of this model.**

ⓘ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

**Given a learning curve of the training set as given below, what is likely to be case in the training of this model.**

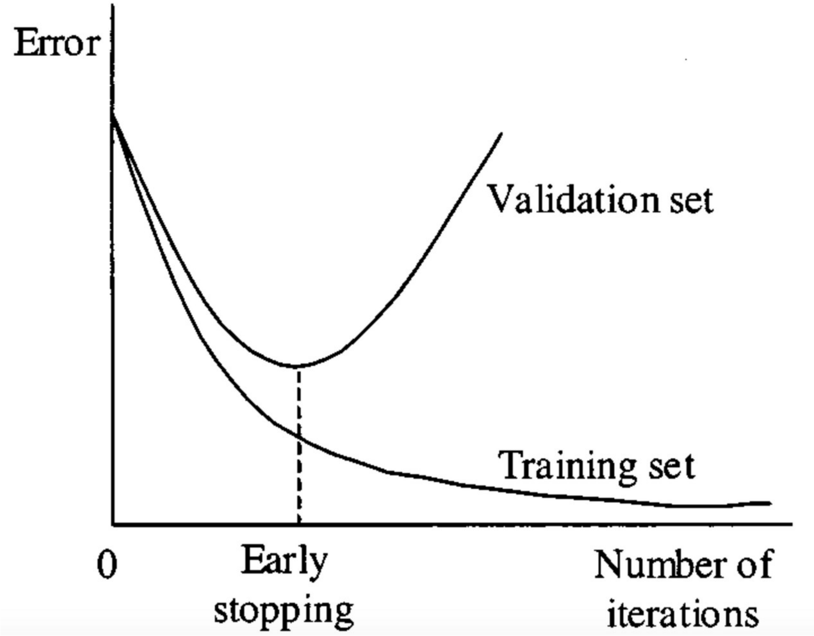
ⓘ Start presenting to display the poll results on this slide.

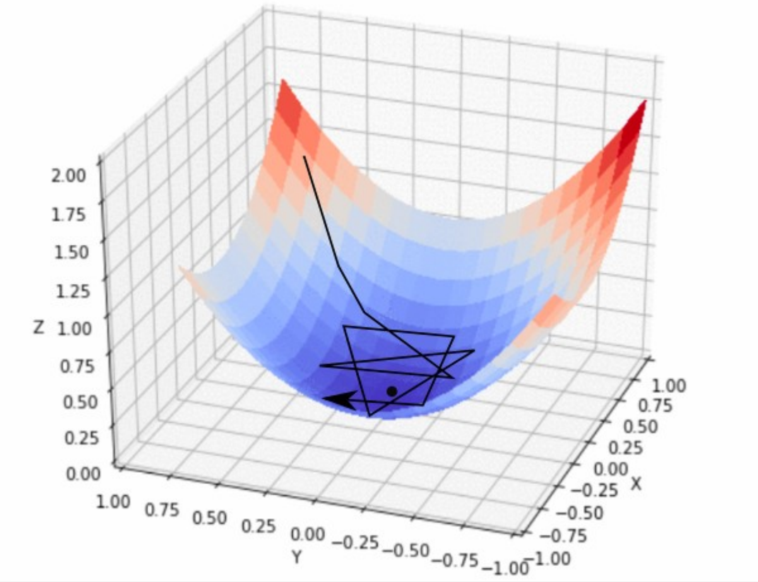
# Python

# Callbacks

- Special functions or methods that can be executed at specific points during the training
- Early stopping sops training when the validation loss stops improving

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$



# Python

# Optimizers (SGD vs Batch GD)

- We do not need to update the weights using ALL observations
- We still need to go over ALL observations (epoch)

Upd w's

| Row ID | Study Hrs | Sleep Hrs | Quiz | Exam |
|---|---|---|---|---|
| 1 | 12 | 6 | 78% | 93% |
| 2 | 22 | 6.5 | 24% | 68% |
| 3 | 115 | 4 | 100% | 95% |
| 4 | 31 | 9 | 67% | 75% |
| 5 | 0 | 10 | 58% | 51% |
| 6 | 5 | 8 | 78% | 60% |
| 7 | 92 | 6 | 82% | 89% |
| 8 | 57 | 8 | 91% | 97% |

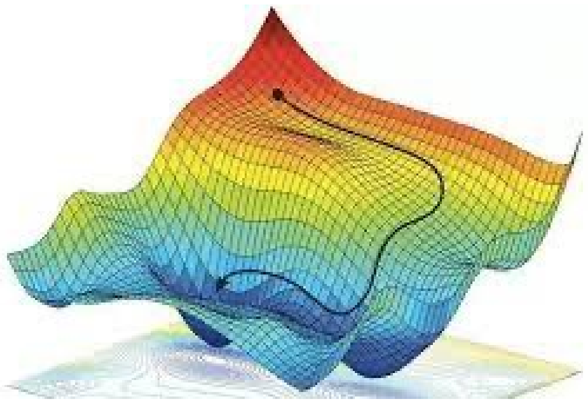
**Batch Gradient Descent**

Upd w's
Upd w's
Upd w's
Upd w's
Upd w's
Upd w's
Upd w's
Upd w's

| Row ID | Study Hrs | Sleep Hrs | Quiz | Exam |
|---|---|---|---|---|
| 1 | 12 | 6 | 78% | 93% |
| 2 | 22 | 6.5 | 24% | 68% |
| 3 | 115 | 4 | 100% | 95% |
| 4 | 31 | 9 | 67% | 75% |
| 5 | 0 | 10 | 58% | 51% |
| 6 | 5 | 8 | 78% | 60% |
| 7 | 92 | 6 | 82% | 89% |
| 8 | 57 | 8 | 91% | 97% |

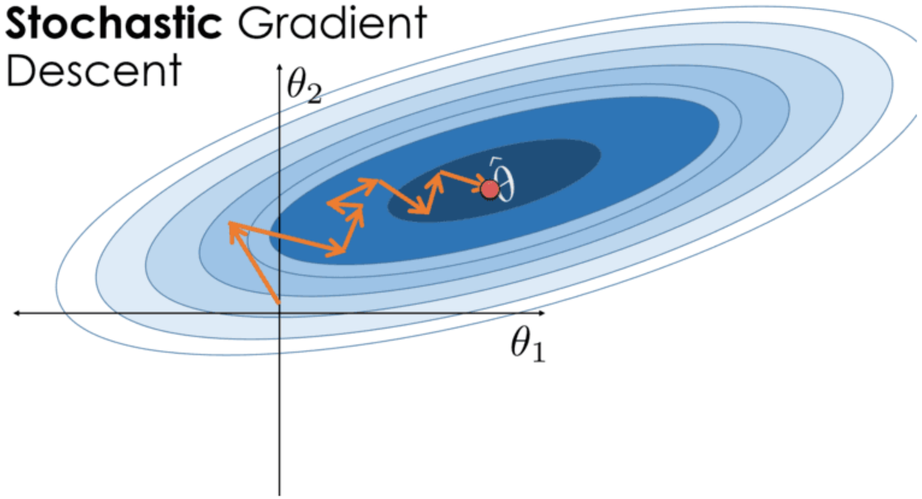
**Stochastic Gradient Descent**

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$

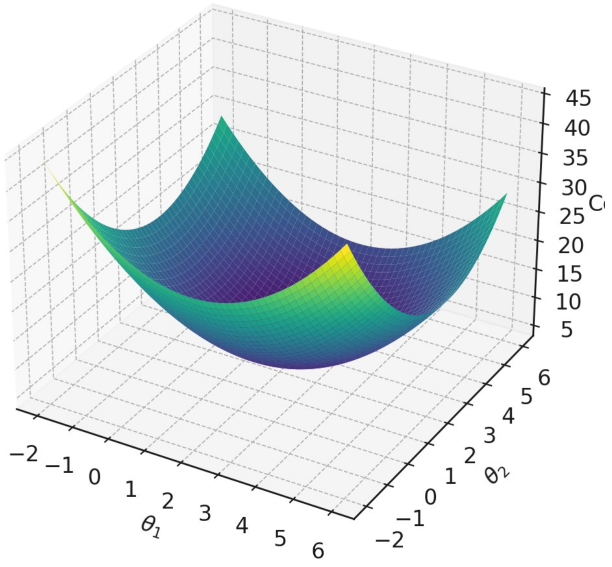


# Optimizers (SGD)

- We do not need to update the weights using ALL observations
- We still need to go over ALL observations (epoch)

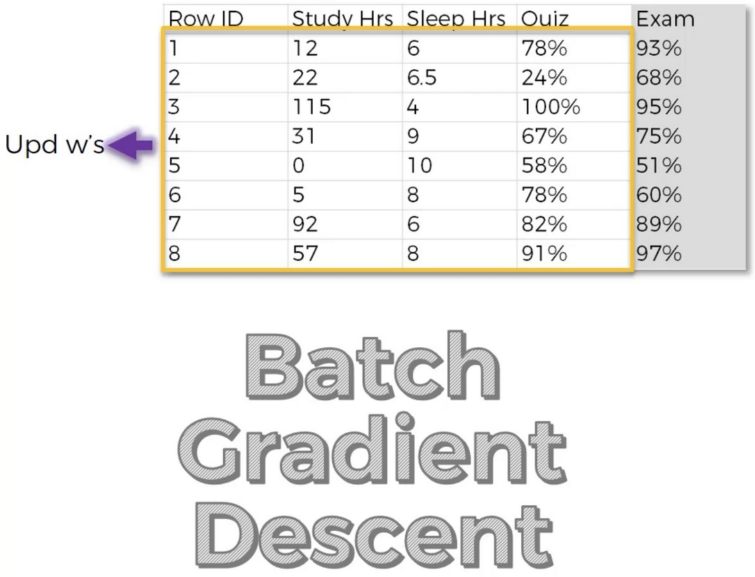



$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$



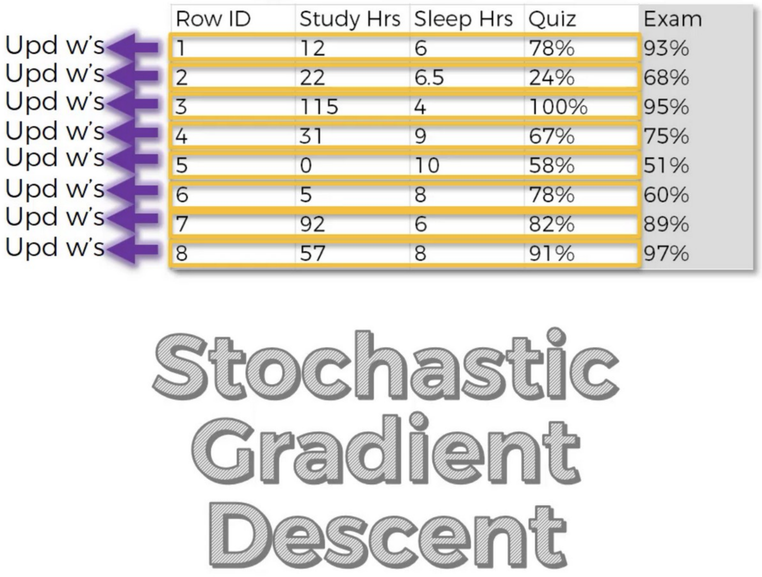
# Batch Gradient Descent

- We do not need to update the weights using ALL observations
- A mini-batch can do as well
- We still need to go over ALL observations (epoch)



| Row ID | Study Hrs | Sleep Hrs | Quiz | Exam |
|---|---|---|---|---|
| 1 | 12 | 6 | 78% | 93% |
| 2 | 22 | 6.5 | 24% | 68% |
| 3 | 115 | 4 | 100% | 95% |
| 4 | 31 | 9 | 67% | 75% |
| 5 | 0 | 10 | 58% | 51% |
| 6 | 5 | 8 | 78% | 60% |
| 7 | 92 | 6 | 82% | 89% |
| 8 | 57 | 8 | 91% | 97% |

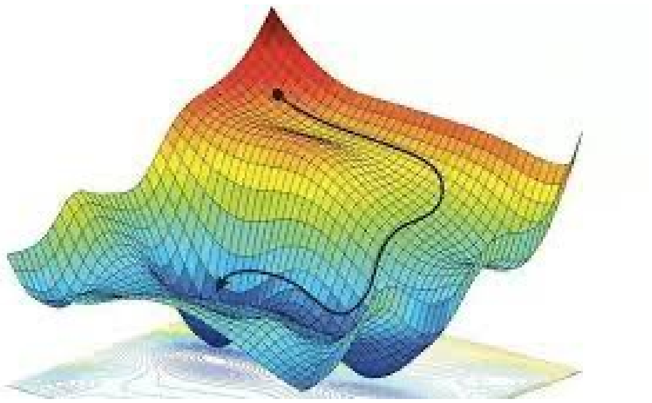




| Row ID | Study Hrs | Sleep Hrs | Quiz | Exam |
|---|---|---|---|---|
| 1 | 12 | 6 | 78% | 93% |
| 2 | 22 | 6.5 | 24% | 68% |
| 3 | 115 | 4 | 100% | 95% |
| 4 | 31 | 9 | 67% | 75% |
| 5 | 0 | 10 | 58% | 51% |
| 6 | 5 | 8 | 78% | 60% |
| 7 | 92 | 6 | 82% | 89% |
| 8 | 57 | 8 | 91% | 97% |

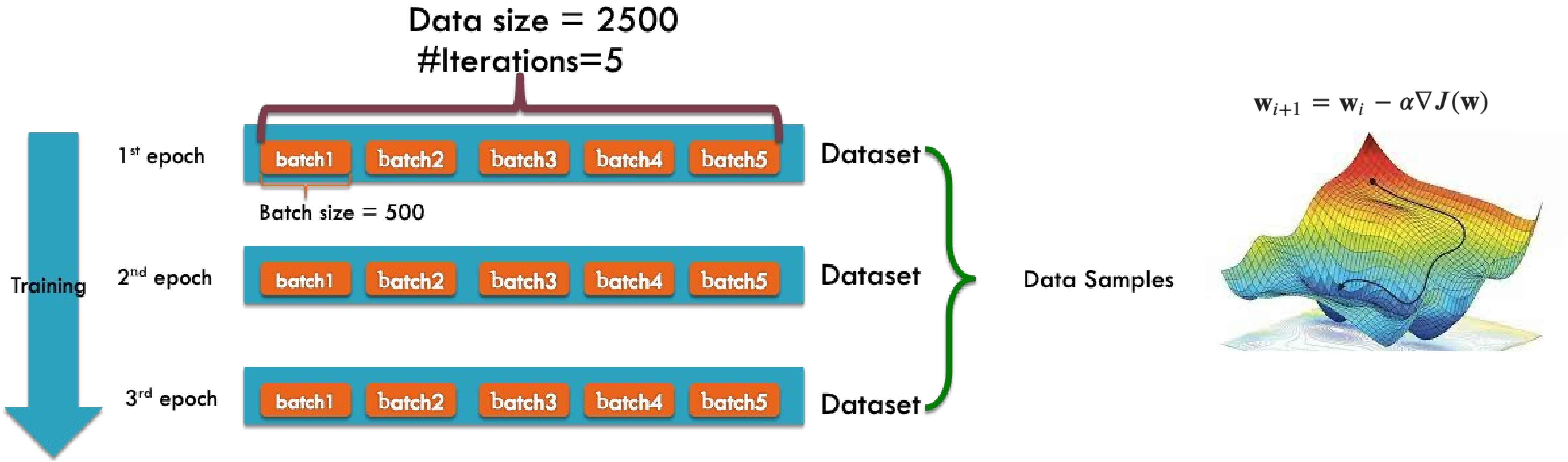


$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$



# Mini-Batch Gradient Descent

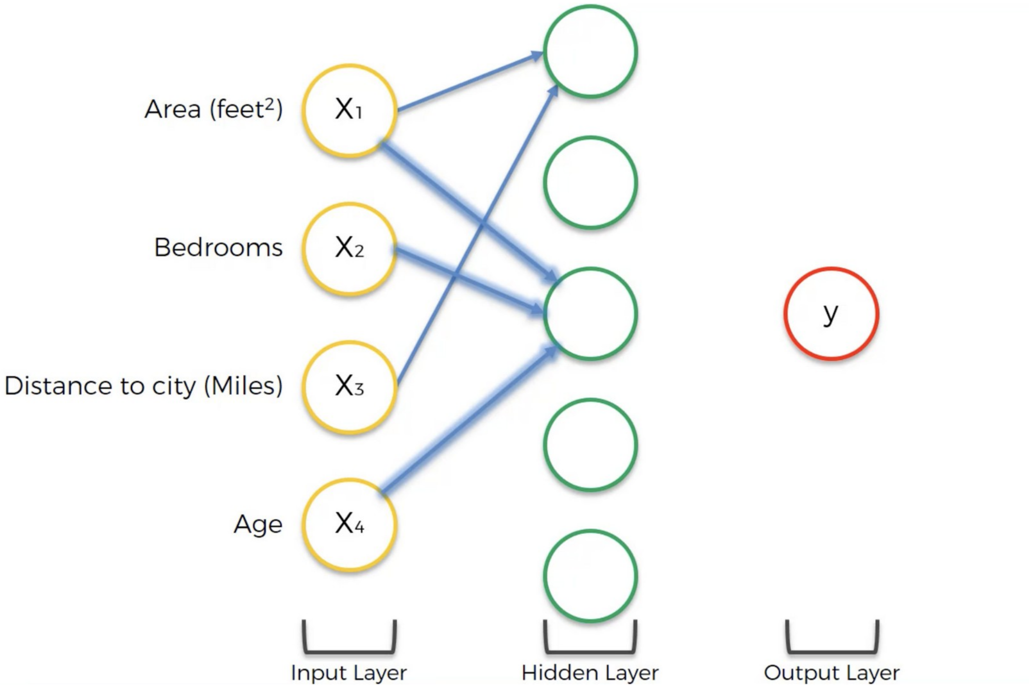
- We do not need to update the weights using ALL observations
- A mini-batch can do as well
- We still need to go over ALL observations (epoch)

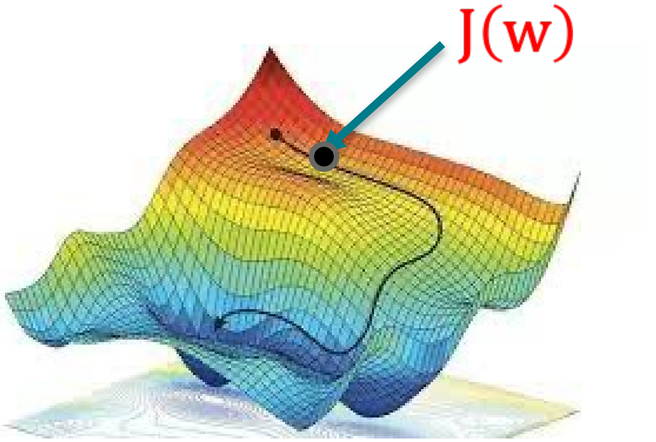


# Python

# Forward Pass

- Hidden layers measure higher-order interactions

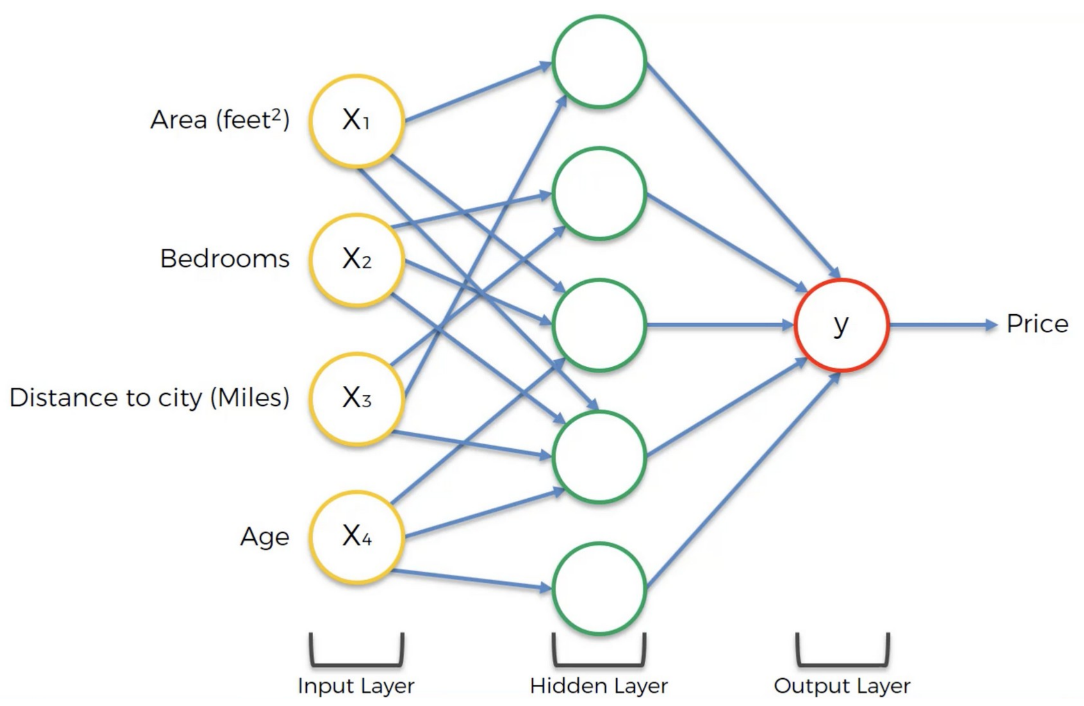


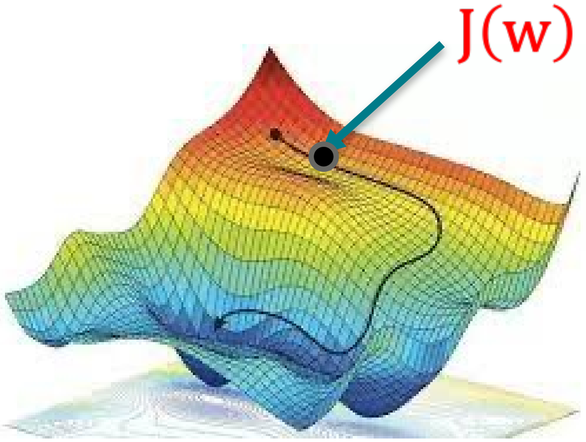

$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

# Forward Pass

- Hidden layers measure higher-order interactions

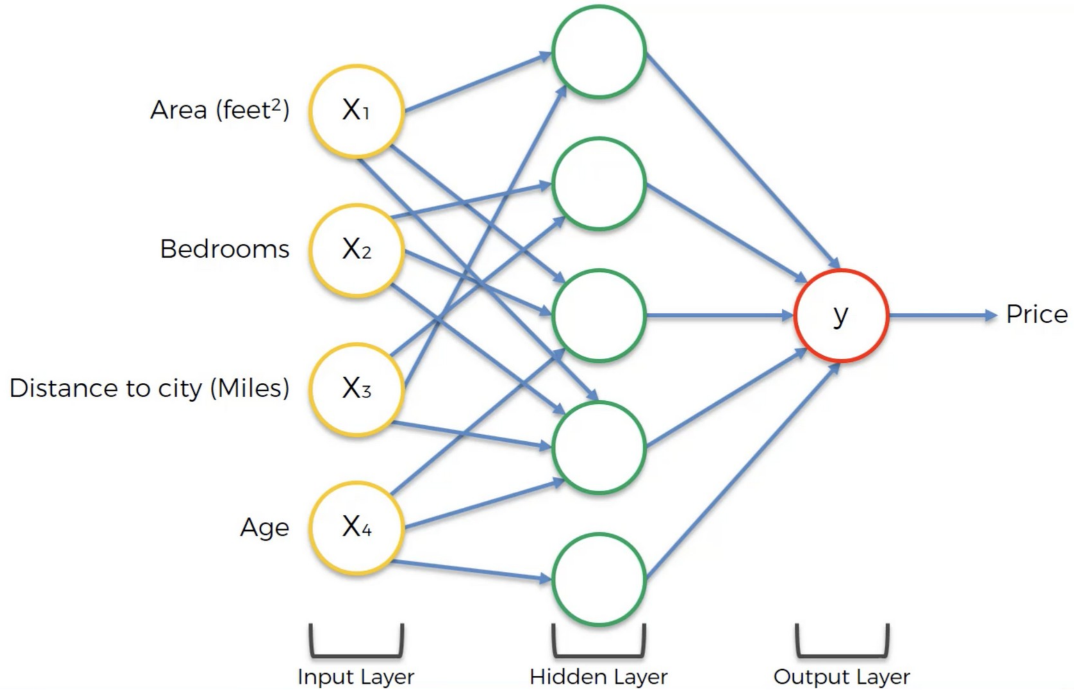



$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

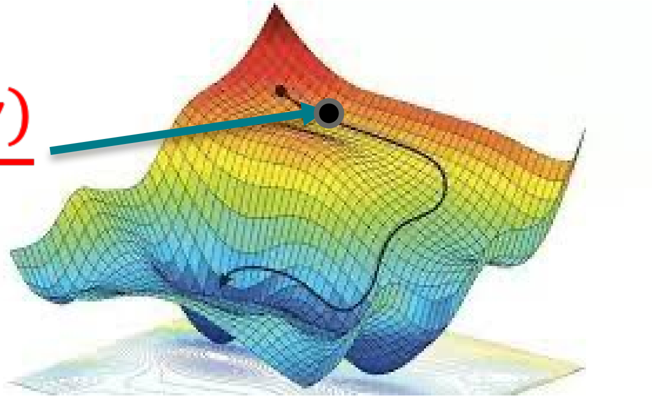



# Update

- How do we update with activations and hidden layers?



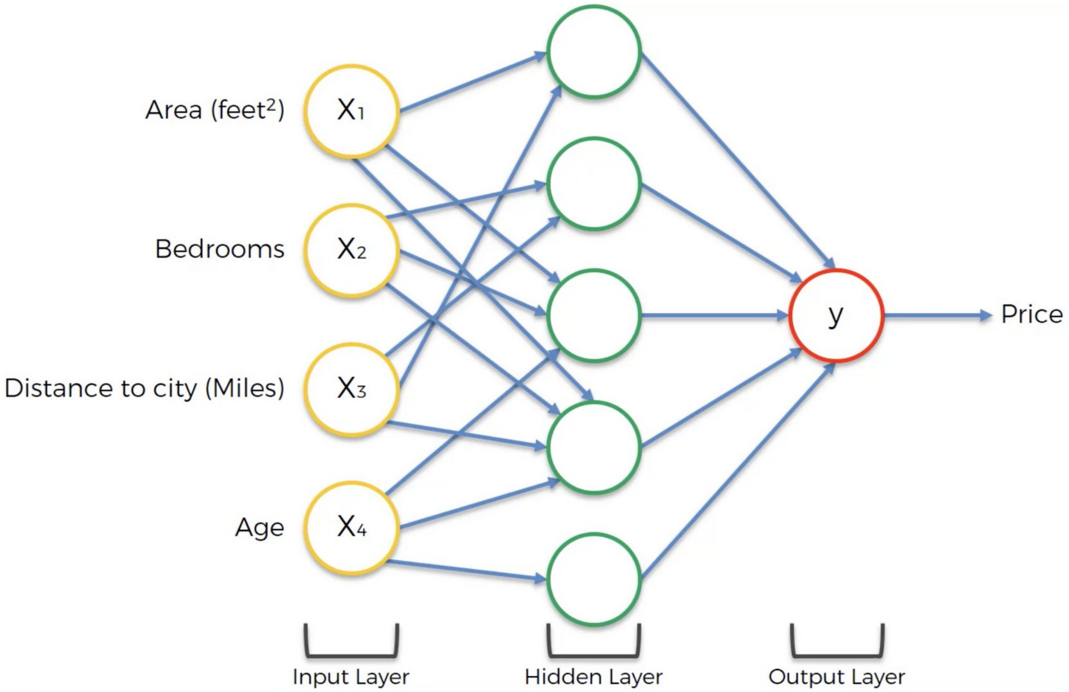

$$\nabla J(w) = \frac{\delta J(w)}{\delta w}$$



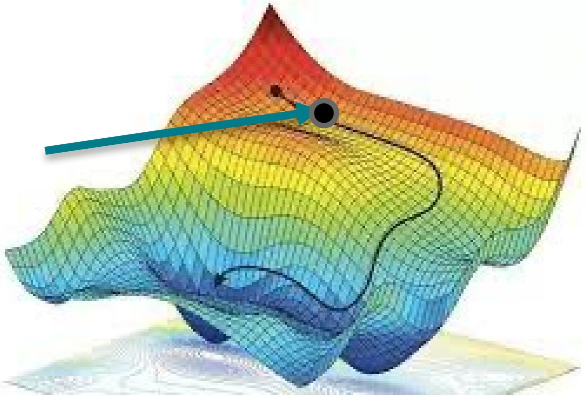
$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

# Update

- How do we update with activations and hidden layers?



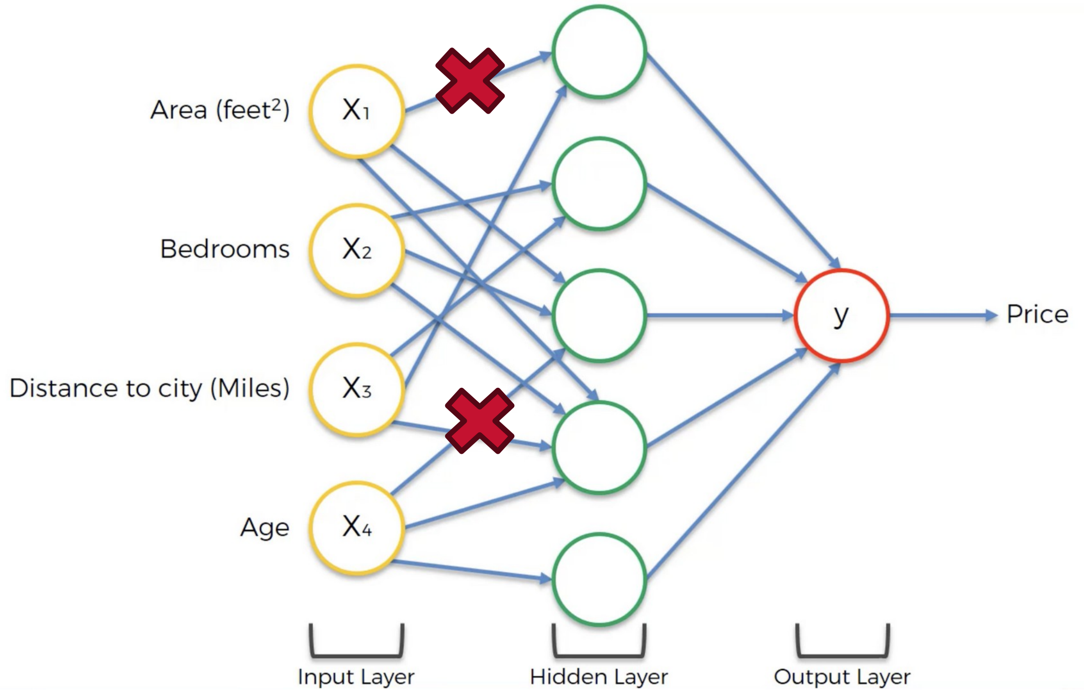

$$\nabla J(w) = \frac{\delta J(w)}{\delta f(z)} \frac{\delta f(z)}{\delta w}$$



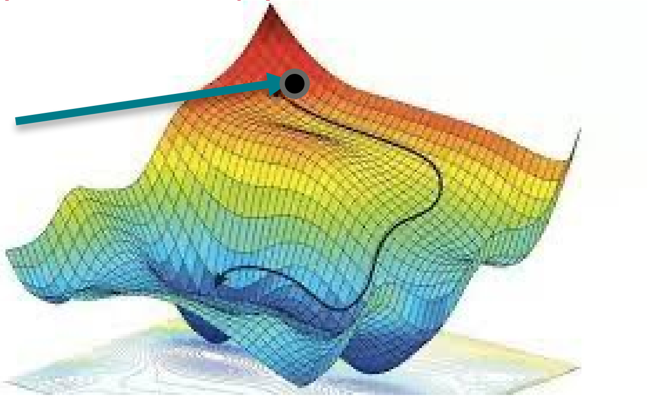
$$\hat{y} = f(w^{\top} x + b)$$

# Dropout Layers

- Regularization technique in neural networks that randomly sets a fraction of input units (neurons) to zero during training.



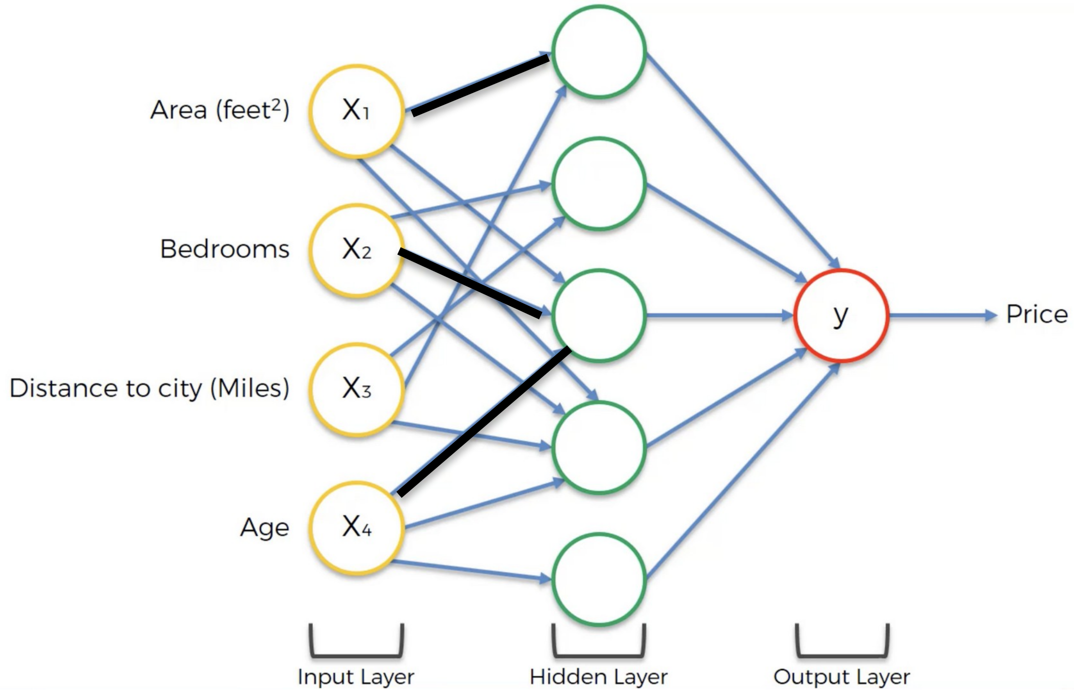

$$\nabla \mathrm{J}(\mathrm{w}) = \frac{\delta J(w)}{\delta f(z)} \frac{\delta f(z)}{\delta w}$$



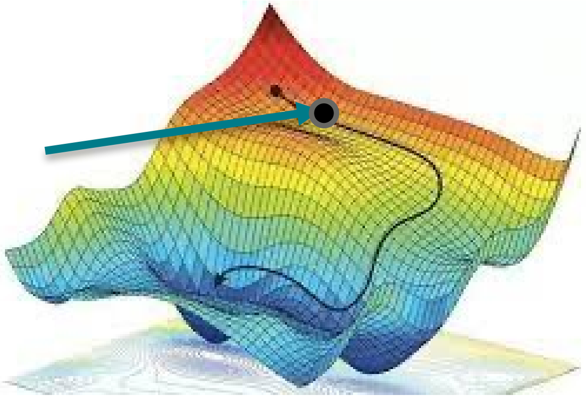
$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

# $L_1$ (Lasso) / $L_2$ (Ridge) Layer Regularization

- Prevents overfitting by adding constraints to a model's parameters.




$$\nabla J(w) = \frac{\delta J(w)}{\delta f(z)} \frac{\delta f(z)}{\delta w}$$

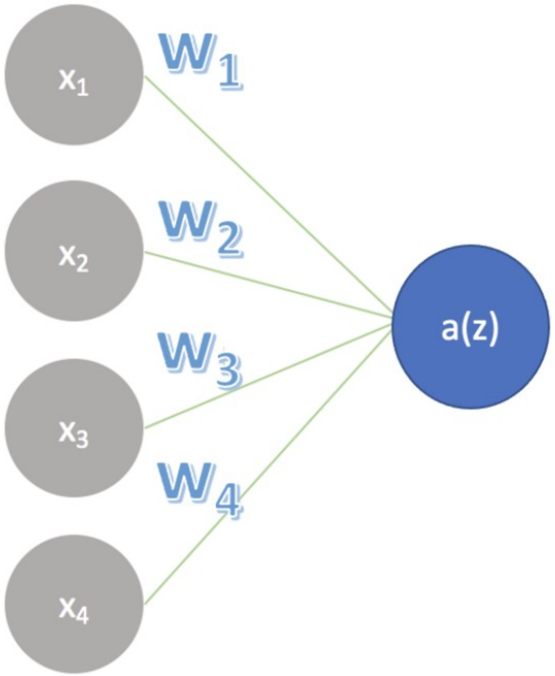


$$\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{w}^{\top}\boldsymbol{x} + \boldsymbol{b})$$

# Python

# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function

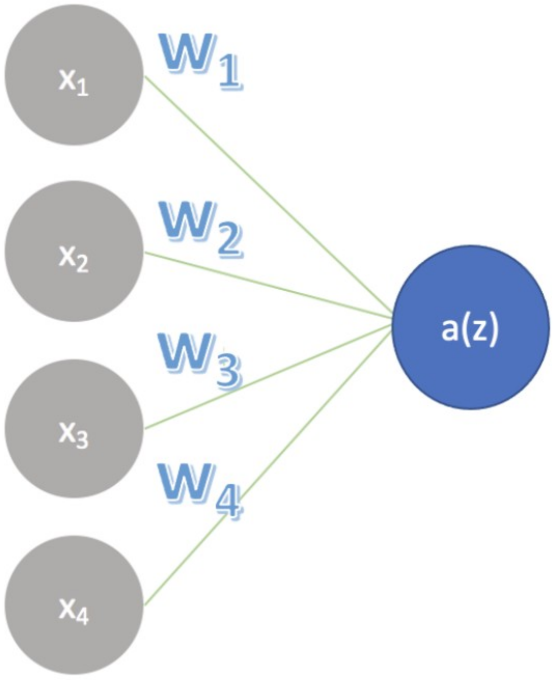



$$z = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$

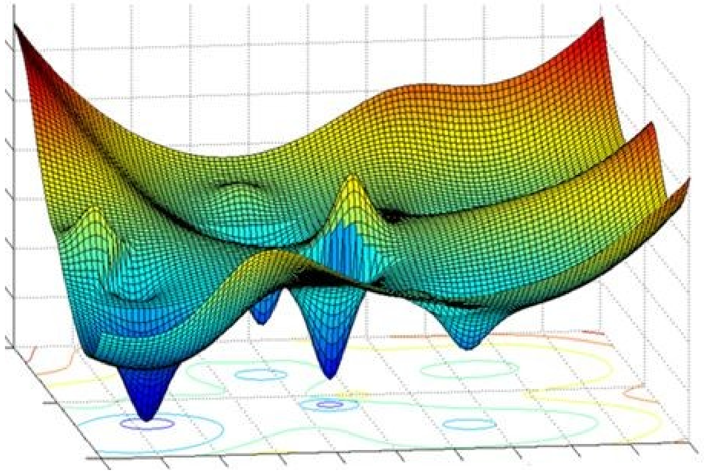
$$a(z) = \frac{1}{1+e^{-z}}$$

# Everything is Vectorized

- ☒ Suppose $y = \{0,1\}$
- ☐ Cost function is the binary cross-entropy
- ☐ Output layer has sigmoid activation function

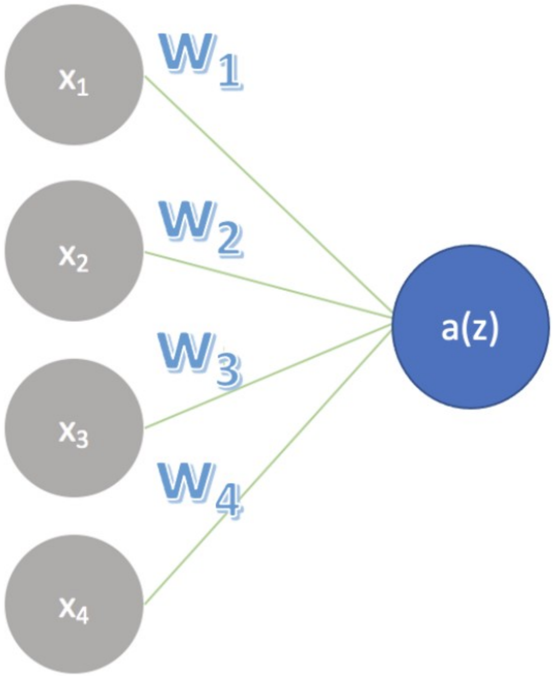



$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



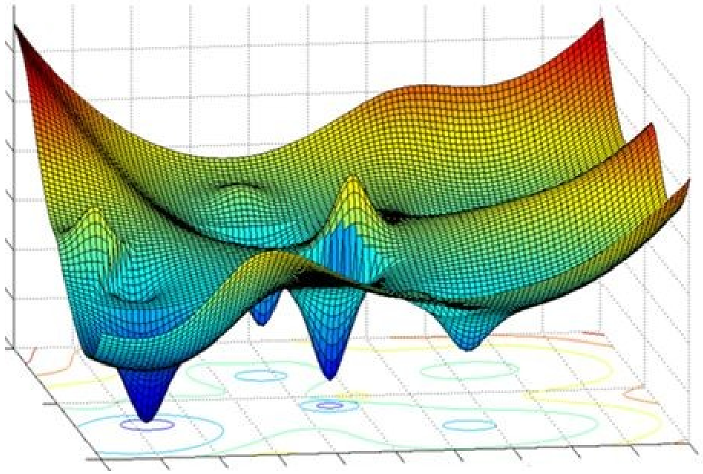
# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



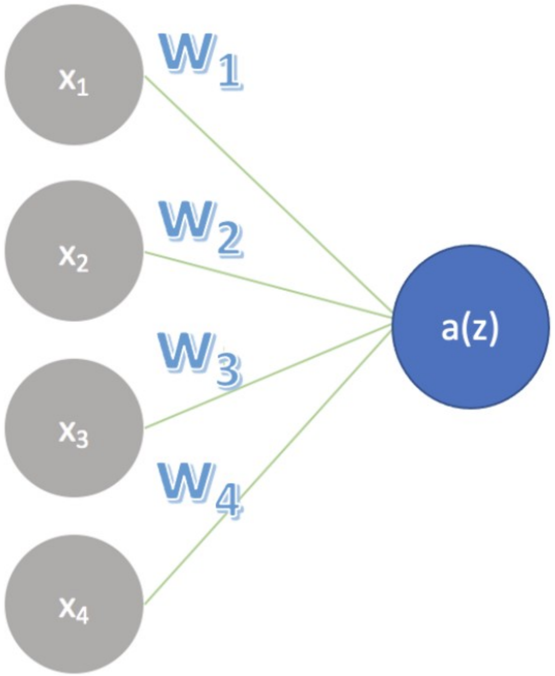

$$\frac{\delta J(\mathbf{w})}{\delta a(z)} = \frac{-\delta \sum_{i=1}^{n}(y_i \log(a) + (1 - y_i) log(1 - a))}{\delta a}$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



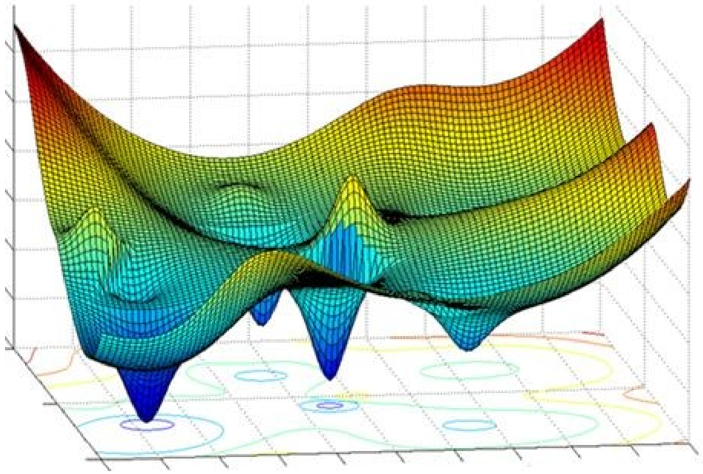
# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



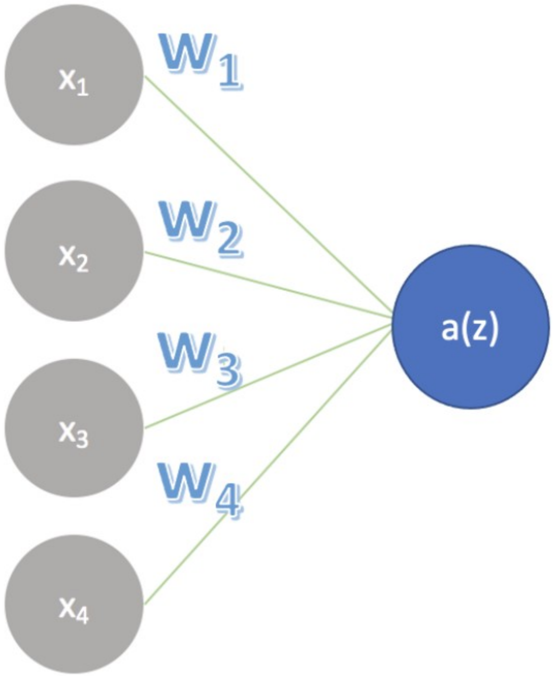

$$\frac{\delta J(\mathbf{w})}{\delta a(z)} = a - y$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



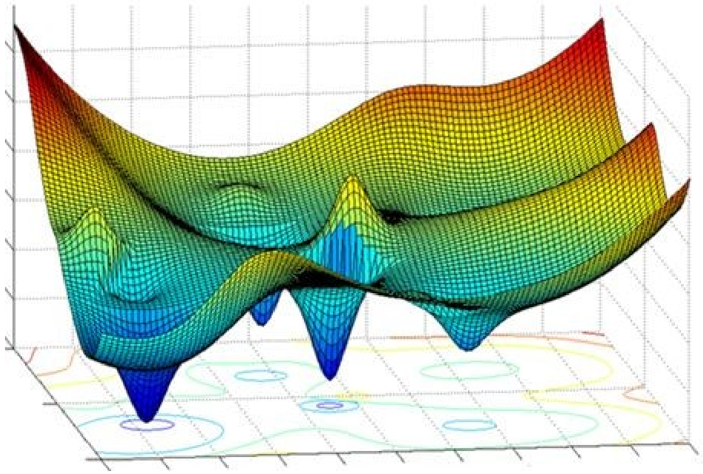
# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



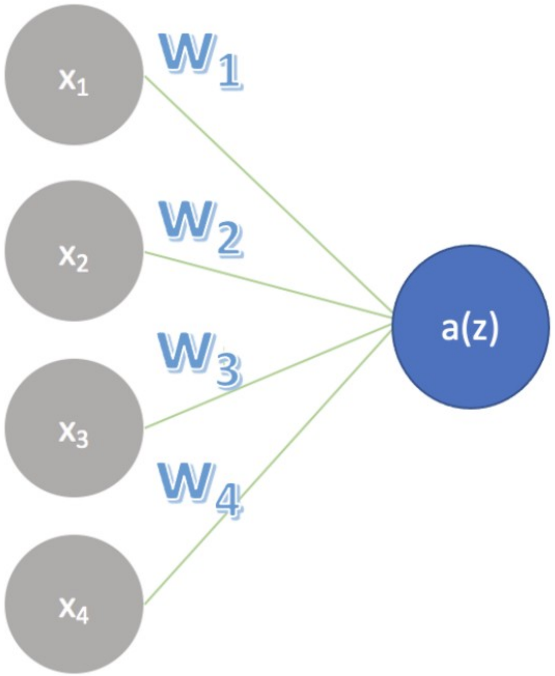

$$\frac{\delta a(z)}{\delta \mathbf{w}} = \frac{\delta \mathbf{w}^\top \mathbf{x} + b}{\delta \mathbf{w}} = \mathbf{x}^\top$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



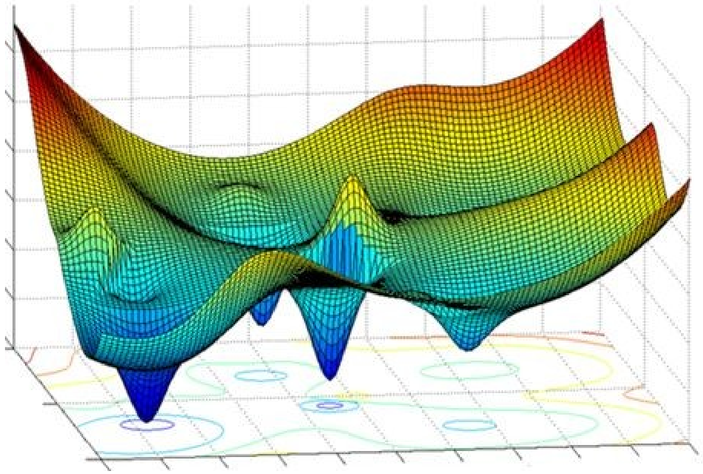
# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



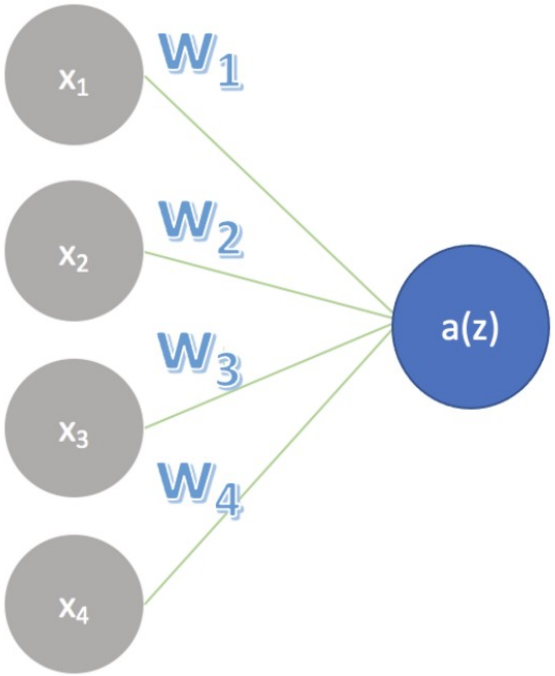

$$\frac{\delta J(\mathbf{w})}{\delta \mathbf{w}} = (a - y)\mathbf{x}^\top$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



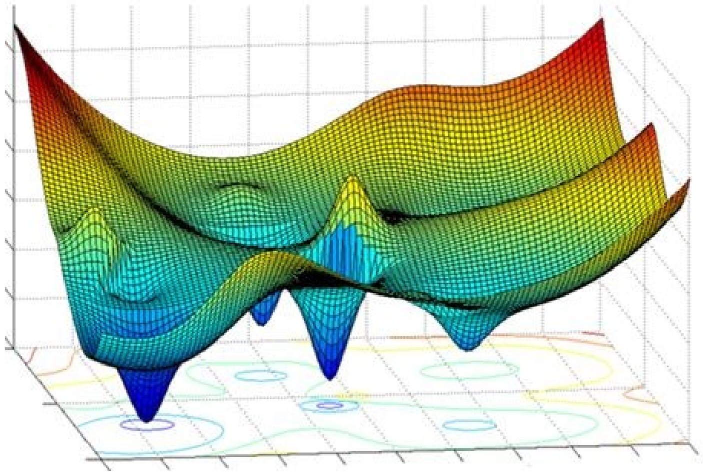
# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



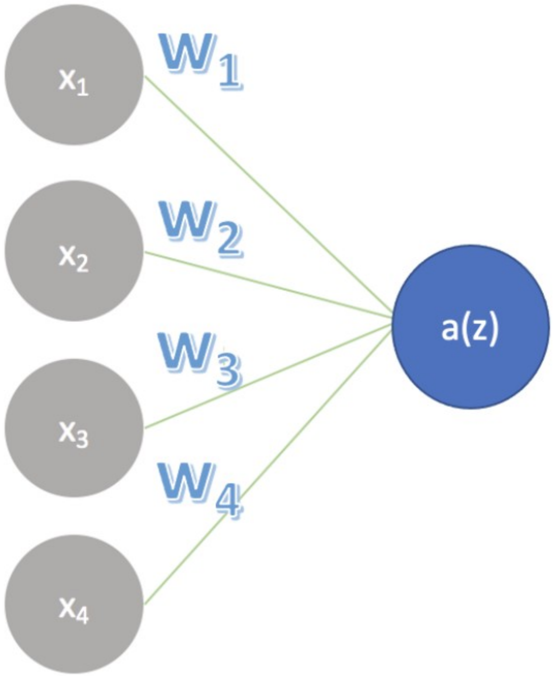

$$\nabla J(\mathbf{w}) = (a - y)\mathbf{x}^{\top}$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



# Everything is Vectorized

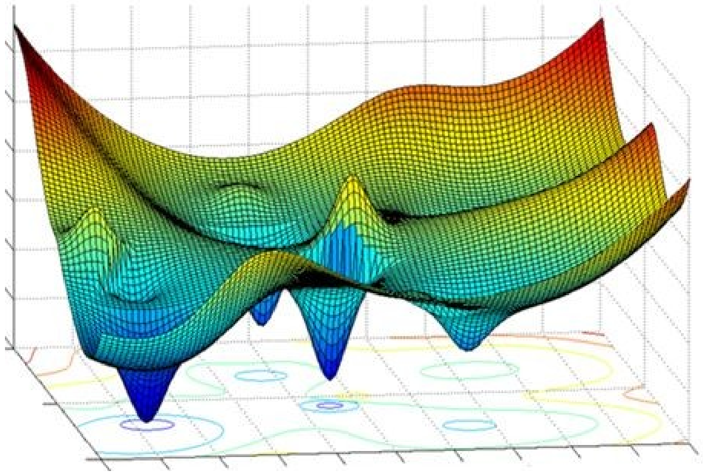
- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function




How weights are updated

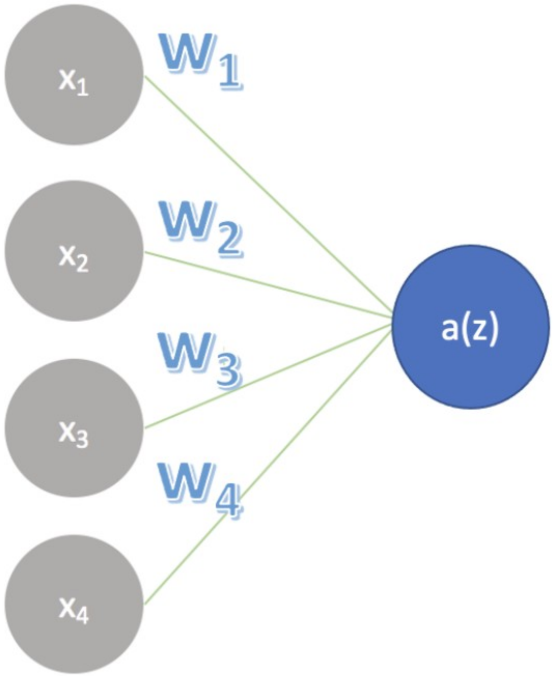
$$\nabla J(\mathbf{w}) = (A - Y)\mathbf{X}^{\top}$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



# Everything is Vectorized

- Suppose $y = \{0,1\}$
- Cost function is the binary cross-entropy
- Output layer has sigmoid activation function



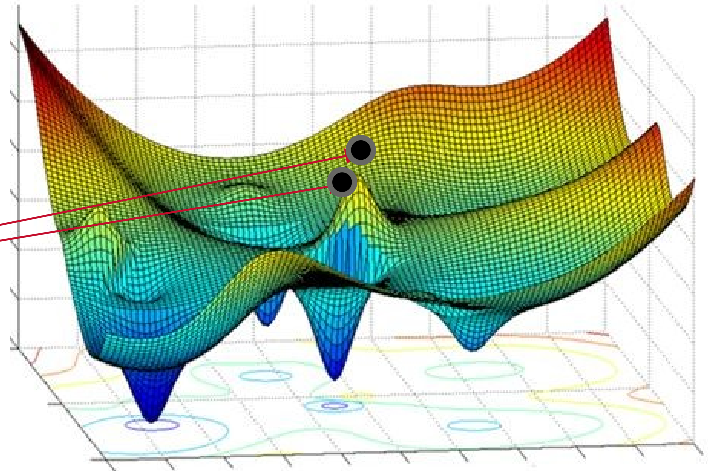

How biases are updated

$$\nabla J(\mathbf{w}) = (A - Y)\mathbf{1}^{\top}$$

All updates are done here

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla J(\mathbf{w})$$

$$\nabla J(\mathbf{w}) = \frac{\delta J(\mathbf{w})}{\delta a(z)} \frac{\delta a(z)}{\delta \mathbf{w}}$$



# Python