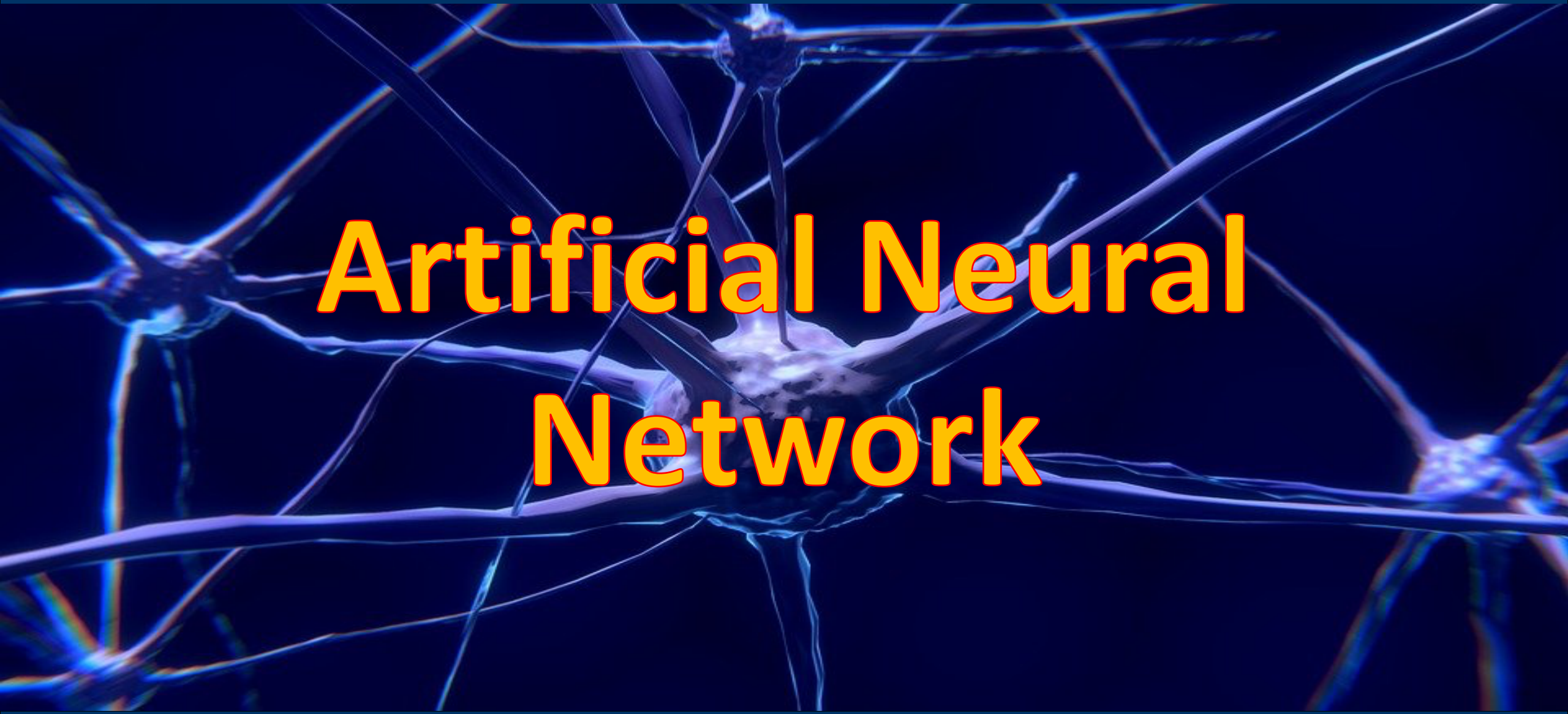
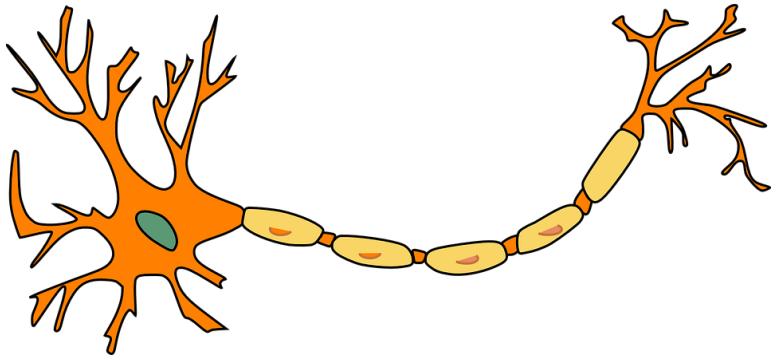
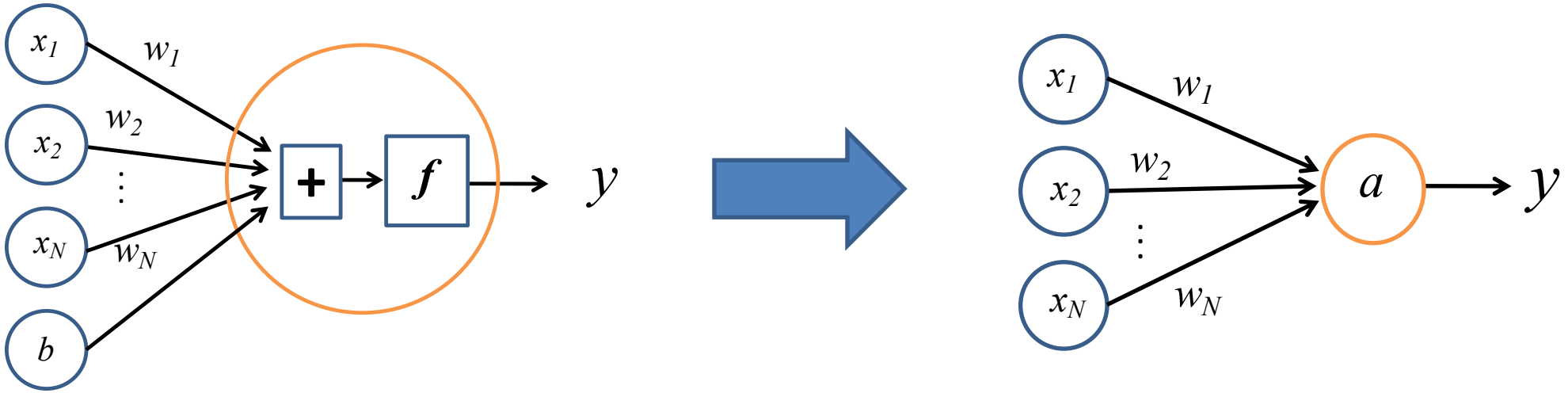
# Advanced Machine Learning and Deep Learning

**Mohammad Pourhomayoun**
Assistant Professor
Computer Science Department
California State University, Los Angeles


Artificial Neural
Network

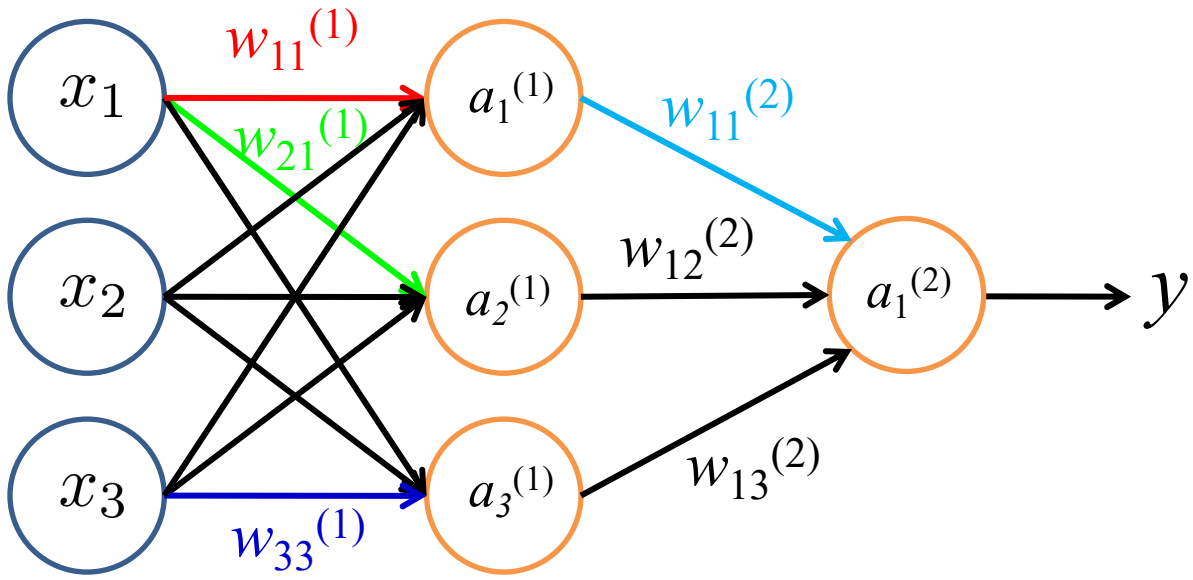
# First Step in ANN: Simulating a Neuron



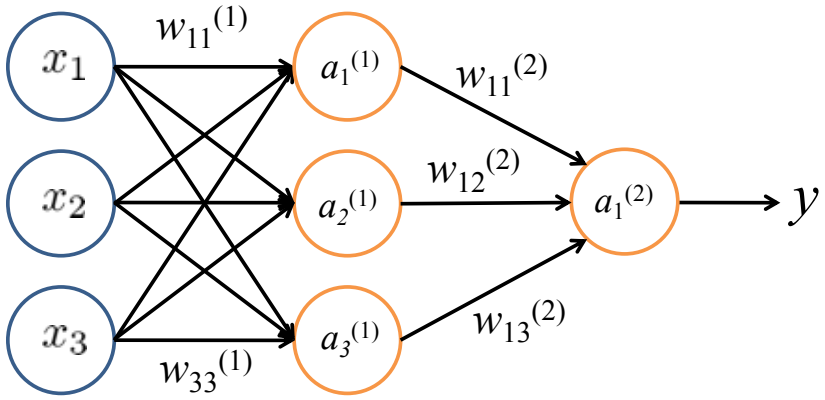


# Notations

- $a_i^{(l)}$ = "***output of activation function***" of unit $i$ in hidden layer $l$.
- $w_{ij}^{(l)}$ = "***weight***" of path from unit $j$ in layer $l$-$1$ to unit $i$ in layer $l$
- $W^{(l)}$ = "***matrix of weights***" from layer $l$-$1$ to layer $l$ .
- $b^{(l)}$ = "***Bias***" in layer $l$ .



# Forward propagation: Vectorized implementation




$$a_1^{(1)} = g\left(w_{10}^{(1)} b^{(1)} + w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3\right)$$

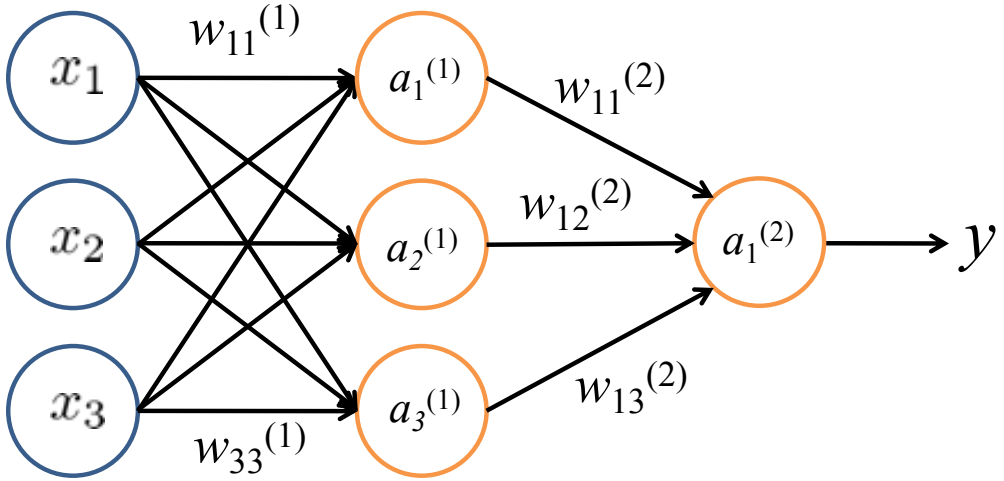
$$a_2^{(1)} = g\left(w_{20}^{(1)} b^{(1)} + w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3\right)$$

$$a_3^{(1)} = g\left(w_{30}^{(1)} b^{(1)} + w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3\right)$$

$$\boldsymbol{a}^{(1)} = \begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{pmatrix} \qquad W^{(1)} = \begin{bmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{30}^{(1)} & w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} b \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\boldsymbol{a}^{(1)} = g(\boldsymbol{W}^{(1)}\boldsymbol{x})$$

# Forward propagation: Vectorized implementation



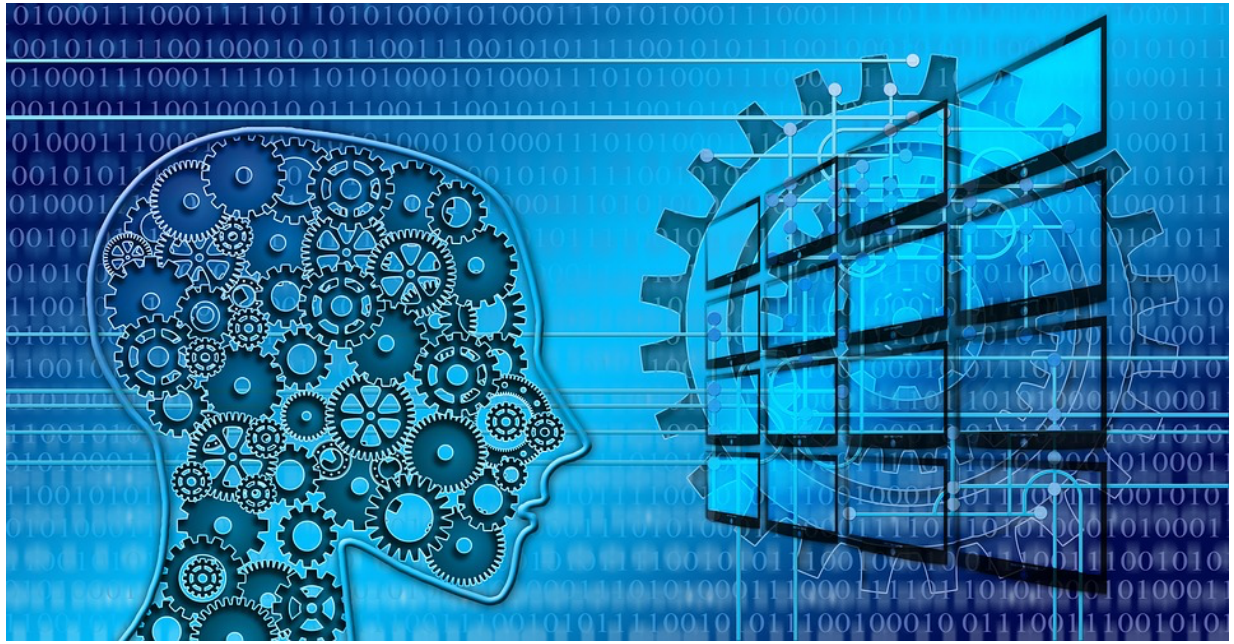

*Similarly,*

$$\boldsymbol{a}^{(1)} = g(\boldsymbol{W}^{(1)}\boldsymbol{x})$$

$$y = \boldsymbol{a}^{(2)} = g(\boldsymbol{W}^{(2)}\,\boldsymbol{a}^{(1)})$$

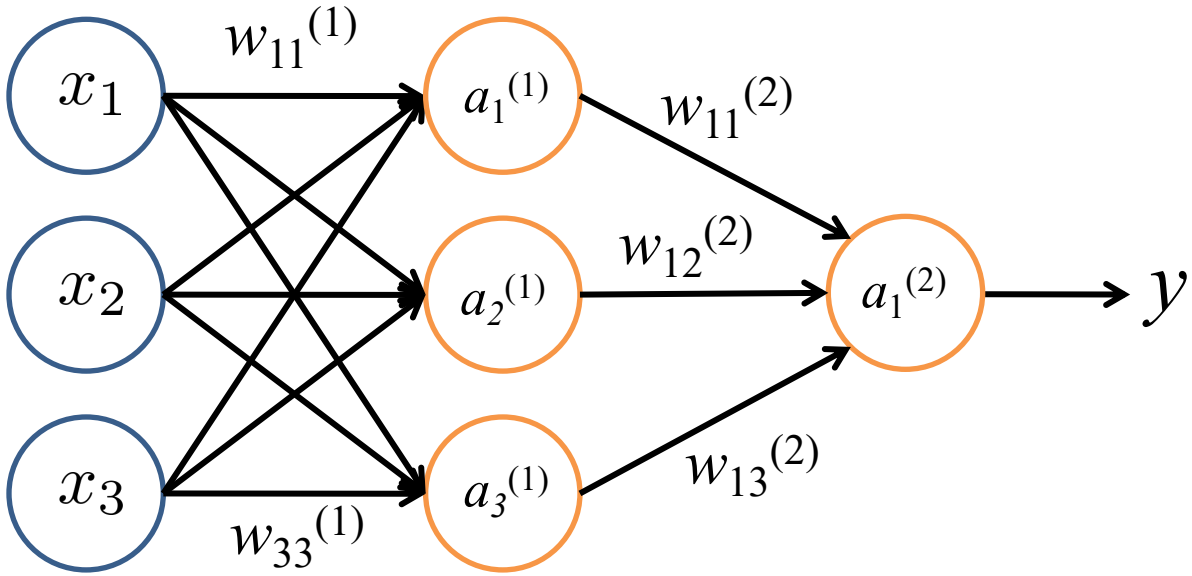
Note: in last equation, we have to add $a_0^{(1)} = b^{(2)}$ to take into account the bias of second layer.

# Training an ANN Model

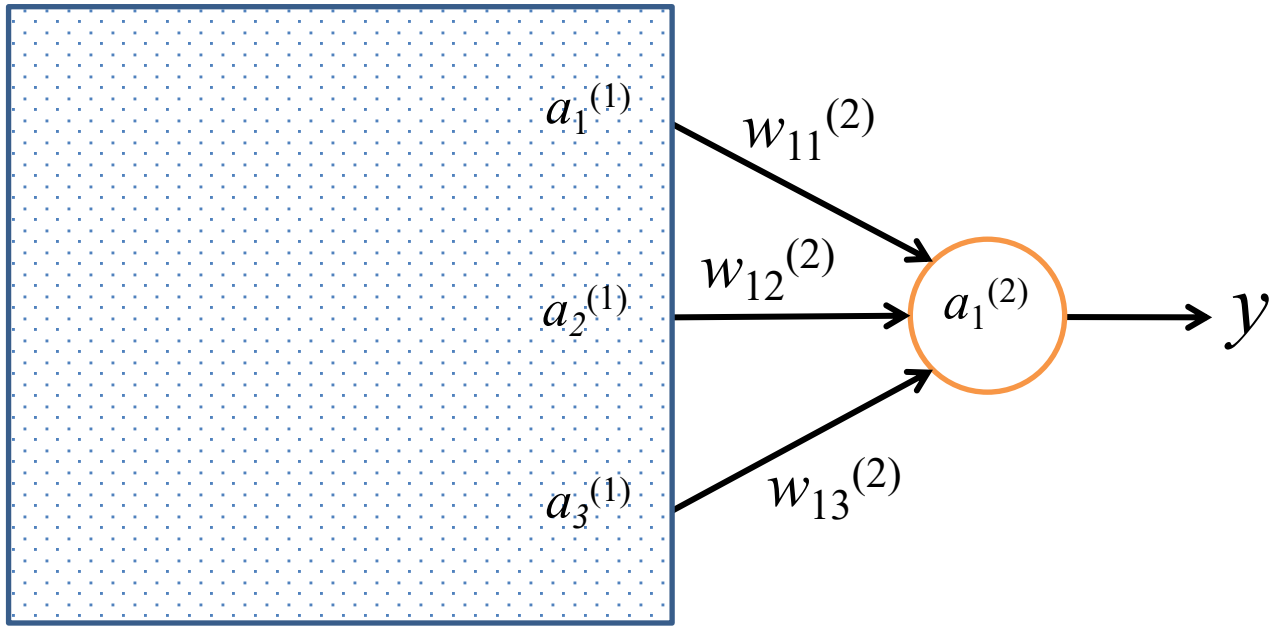


# A Very Cool Interpretation of ANN Model!

- In the following ANN structure, if we forget about the first layers, and just focus on the **last layer**, how does it look like?



# A Very Cool Interpretation of ANN Model!



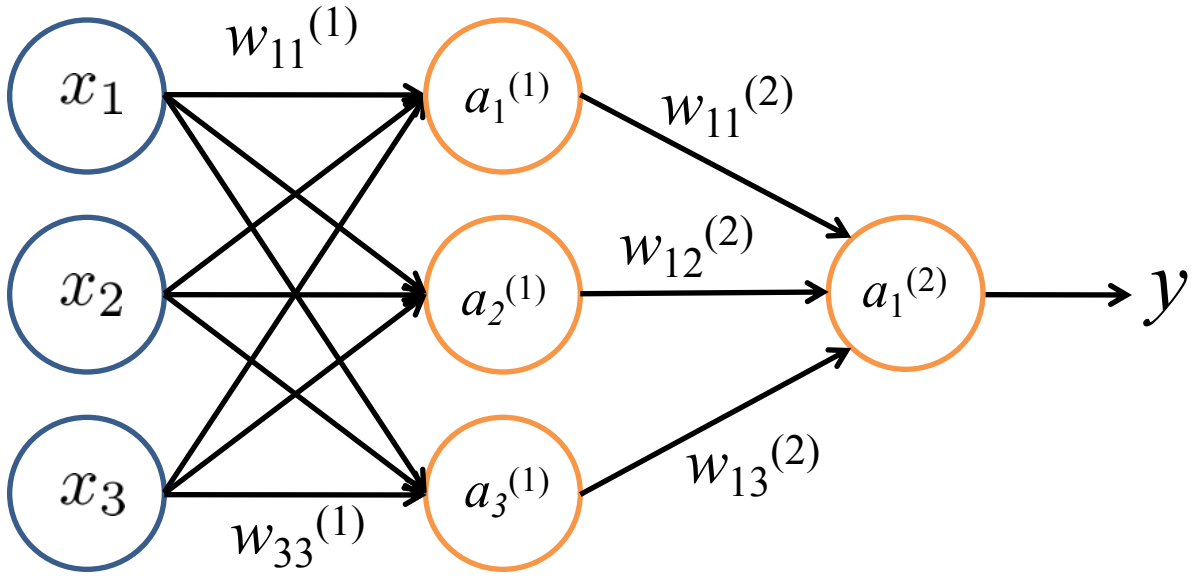

$$y = g\left(w_{10}^{(2)} b^{(2)} + w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + w_{13}^{(2)} a_3^{(1)}\right)$$

What does it look like if we put previous layers in a black box?

**Answer: Logistic Regression!!!!**

# A Very Cool Interpretation of ANN Model!

- In fact, **ANN** acts like **logistic/Linear regression**. The important difference is that, rather than just using raw input features, ANN processes the input features, finds the best combination of them as new features (**learns to create its own features**), and eventually uses them for classification!



# Training for ANN

- Similar to any other Machine Learning algorithm, We can **Train** an Artificial Neural Network based on a "***Training Data***".

- Artificial Neural Network "***learns***" from historical data, just like children learn to recognize a dog from previous examples of dogs, by extracting and recognizing the pattern.

# Training for ANN

- **In Training Stage, an ANN tries to find the best wiring structure among neurons. Particularly, it tries to find the best connection WEIGHTS based on the training dataset.**

- To do that, we first need to define a ***cost function (error function)*** that represents the ERROR! Then, try to **minimize the cost function** on the Training Dataset, Solve the minimization problem in terms of **Weights** to find the best weights.

# Terminology

Training Set: $\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$

$m$ = Total number of training samples

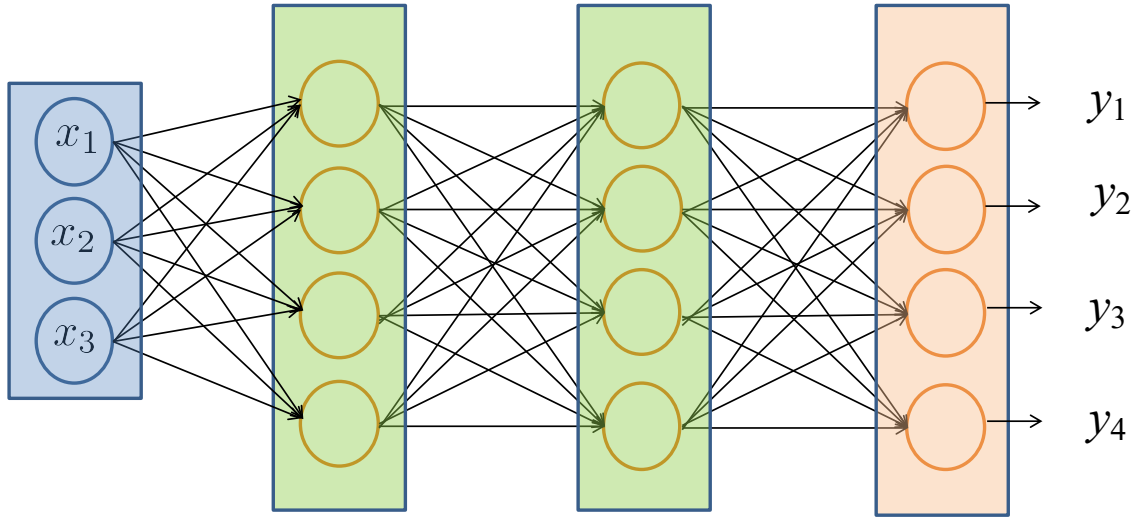
$K$ = Number of outputs for the network

$a_i^{(l)}$ = *output of activation function* of unit $i$ in layer $l$.

$L$ = Total number of Layers including input and output layers.

$S_l$ = Number of units in layer $l$.

$w_{ij}^{(l)}$ = "*weight*" of path <u>from</u> unit $j$ in layer $l$-$1$ <u>to</u> unit $i$ in layer $l$

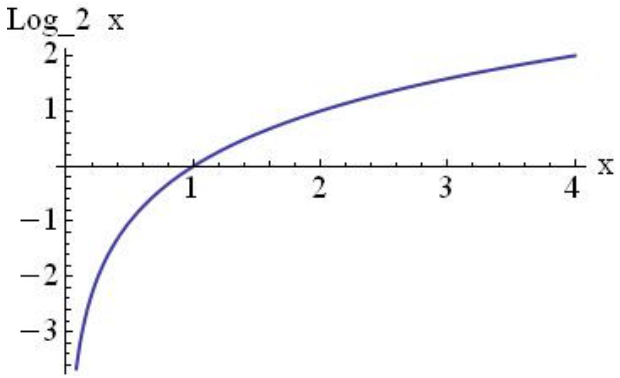


# The Cost Function (Error Function)

**The cost function for Logistic regression:**

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

- $y$ is the actual output
- $h$ is the predicted output



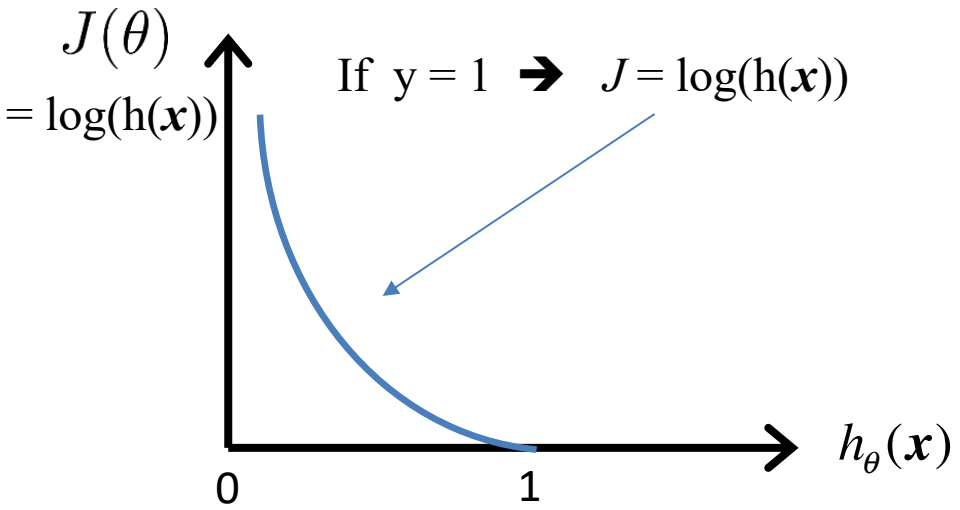
# *Optional*

- **Why does this Cost function work?**

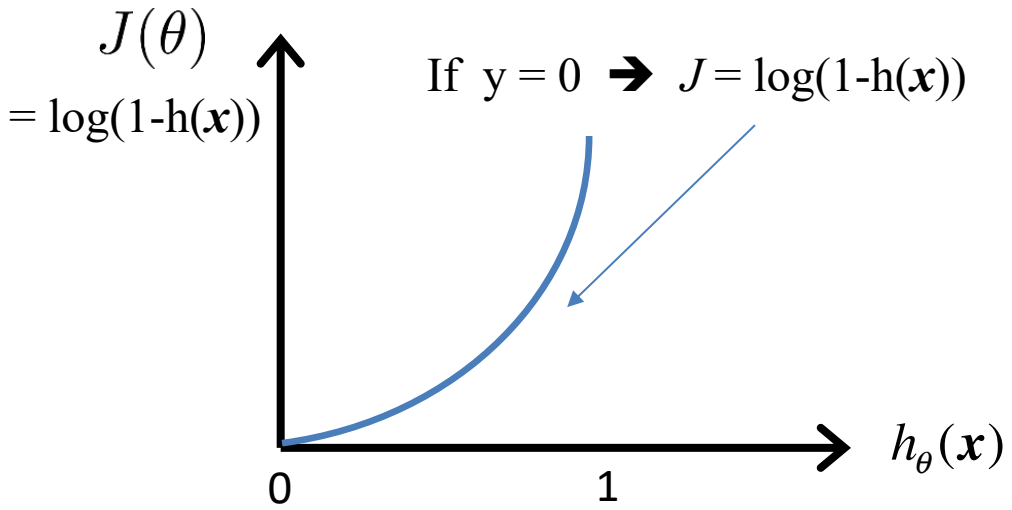
$$J(\boldsymbol{\theta}) = J(\theta_0, \theta_1, \ldots, \theta_n)$$

Note: $y = 0$ or $1$ always

$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log h_\theta(\boldsymbol{x}^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(\boldsymbol{x}^{(i)}))\right]$$




- when y=1, and h(x) is close to 1, the cost will be very small, and when y=1, and h(x) is close to 0, the cost will be too high.




- when y=0, and h(x) is close to 0, the cost will be very small, and when y=0, and h(x) is close to 1, the cost will be too high.

# The Cost Function (Error Function)

**The cost function for Logistic regression:**

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

**Similarly, the cost function for Neural network:**

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_W(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_W(x^{(i)}))_k)\right]$$

where,

$y^{(i)}$ : The actual label for $i^{th}$ data sample in the Training Set (y_train).

$h_W(x^{(i)})$ : The predicted label for $i^{th}$ data sample $x^{(i)}$ in the Training Set (y_predict).

$m$ = Total number of training samples

$K$ = Number of outputs for the network

# The Cost Function (Error Function)

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_W(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_W(x^{(i)}))_k)\right]$$

- The **cost function $J(W)$** represents the error between ***our prediction*** and the ***actual values (ground truth)***.
- Now, we try to **minimize the cost function $J(W)$** on the **Training Dataset,** in **terms of Weights $W$**, to find the $W$ corresponding to the minimum error.

$$\min_W J(W)$$

- *Thus, It will give us* the best weights that minimizes the prediction error!

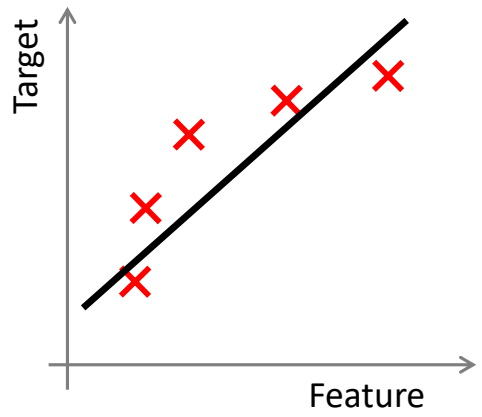
# Review: The Problem of Overfitting

- **Overfitting** happens when the predictive model (classification model or regression model) **fits too much** with the **training samples** so that it starts capturing, learning, and representing the **noise and randomness or outlier samples** of the training dataset.

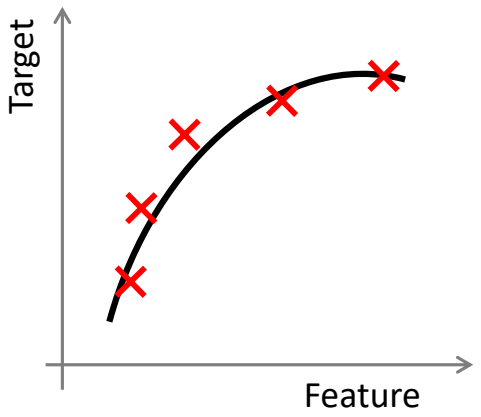
- Overfitting provides excellent accuracy for training data, but poor results for future data samples (testing set)!

# Review: The Problem of Overfitting

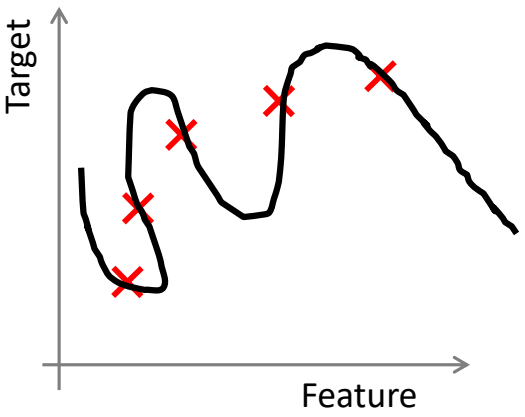
- **Overfitting** occurs when a model is excessively complex. The two main reasons that makes a model too complex are:
  1. having **too many features**.
  2. having a **complex model with very high order**.




**Under-fit**
**(High Bias)**




**Ideal fit**




**Over-fit**
**(High Variance)**

# Review: Addressing the Overfitting Problem by Regularization

- **Approach #2**: **Regularization**:
  - Keep all features, but reduce the magnitude/values of parameters of the model ($\theta_j$ ) to simplify the model.

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_1^2 + \theta_5 x_2^2 + \theta_6 x_2 x_3 + \theta_6 x_2 x_3^2 + \ldots$$

$$\rightarrow \quad \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_1^2 + \theta_6 x_2 x_3$$

# Regularization

**Similar to Logistic Regression, we can add a new term to the cost function to force the weights to be small, and make the weight matrix sparse to avoid overfitting:**

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_W(x^{(i)}))_k + (1- y_k^{(i)}) \log(1 - (h_W(x^{(i)}))_k)\right]$$

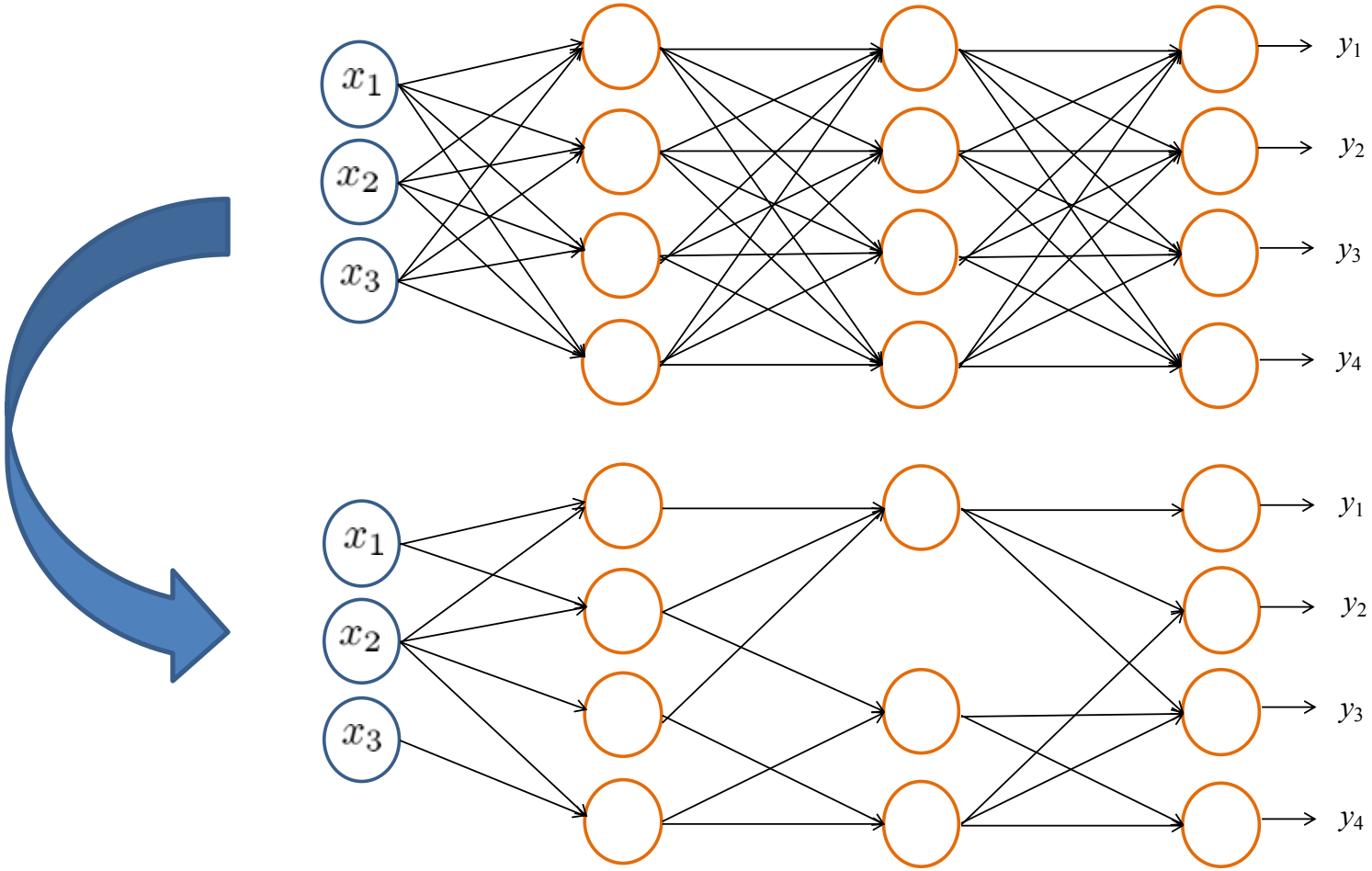
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (w_{ij}^{(l)})^2$$

**By adding this term (called Regularization) to the minimization problem, we not only minimize the cost, but at the same time we force the weights to be small and sparse to avoid overfitting.**

$$\min_W J(W)$$

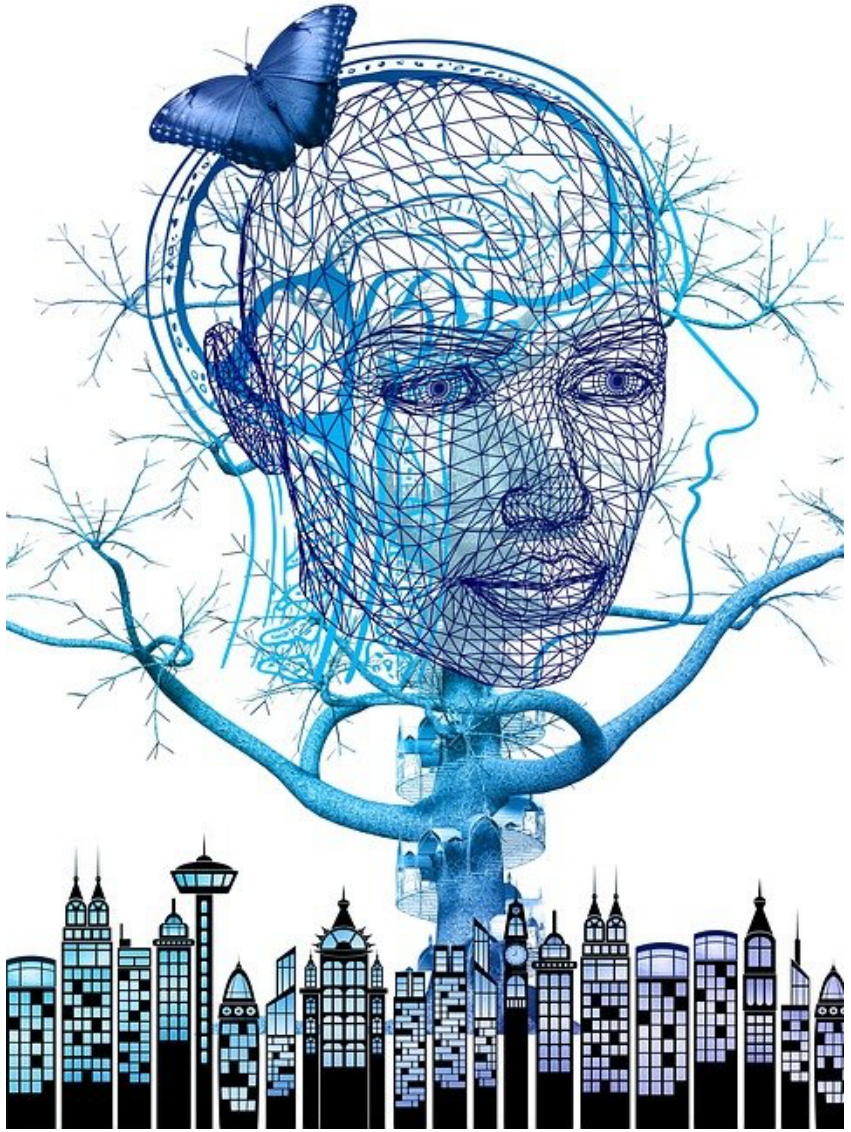
# Regularization

- Regularization makes the Neural Network sparse:



# Regularization

- Believe or not, Your brain does regularization too!
- It is called **Synaptic Pruning.**
  - www.youtube.com/watch?v=bbLP-as1ABk



# The Cost Function (Error Function)

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_W(x^{(i)}))_k + (1- y_k^{(i)}) \log(1 - (h_W(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (w_{ij}^{(l)})^2$$

$$\min_W J(W)$$

This gives us the best weights for all layers of the Neural Network that minimizes the prediction error.

where,

$y^{(i)}$ : The actual label for $i^{\text{th}}$ data sample in the Training Set (y_train).

$h_W(x^{(i)})$ : The predicted label for $i^{\text{th}}$ data sample $x^{(i)}$ in the Training Set (y_predict).

$w_{ij}^{(l)}$ = "*weight*" of path from unit $j$ in layer $l$-$1$ to unit $i$ in layer $l$

# The Cost Function (Error Function)

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_W(x^{(i)}))_k + (1- y_k^{(i)}) \log(1 - (h_W(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (w_{ij}^{(l)})^2$$

$$\min_W J(W)$$

We can use Gradient Descent Algorithm to minimize the Cost Function in terms of "weights", by calculating the partial derivatives: $\frac{\partial}{\partial w_{ij}^{(l)}} J(W)$

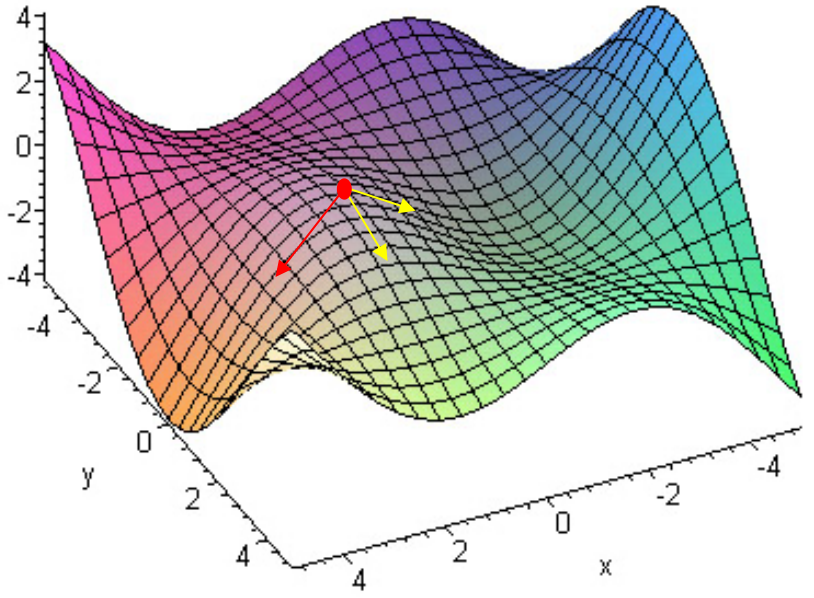
# Review: Gradient Descent Algorithm*

- Gradient Descent is an **iterative** optimization approach that tries to minimize a function.

- The **gradient** is a generalization of the usual concept of **derivative** for functions of **several variables**.

- The **gradient represents the slope of the tangent** of the graph of the function.

- The **negative gradient** points in the **direction** of the **greatest** rate of reduction of the function, and its **magnitude** is the **slope** of the graph in that direction.



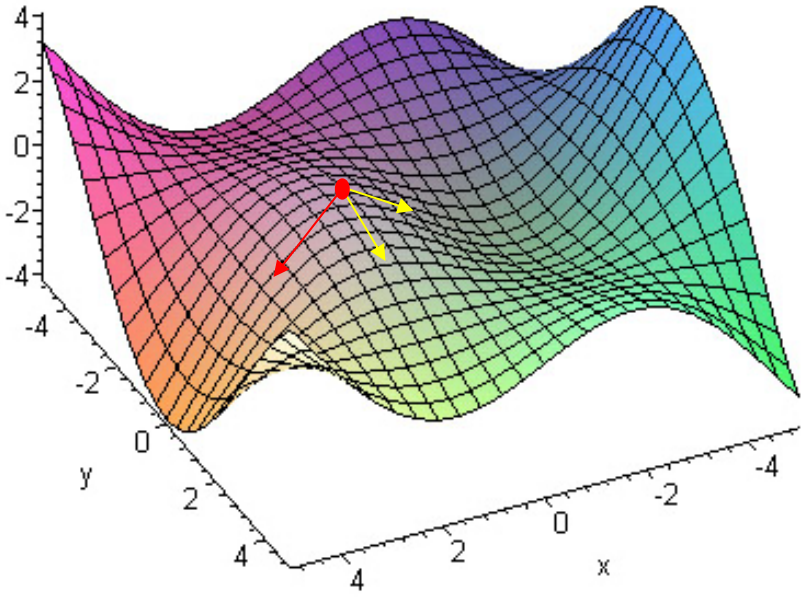

* Reference: Andrew Ng, Machine Learning, Stanford University.

# Gradient Descent Algorithm*

- Gradient Descent is an iterative optimization approach that tries to minimize a function.

- The **negative** direction of the gradient at the current point is the **Steepest Direction** at the moment.

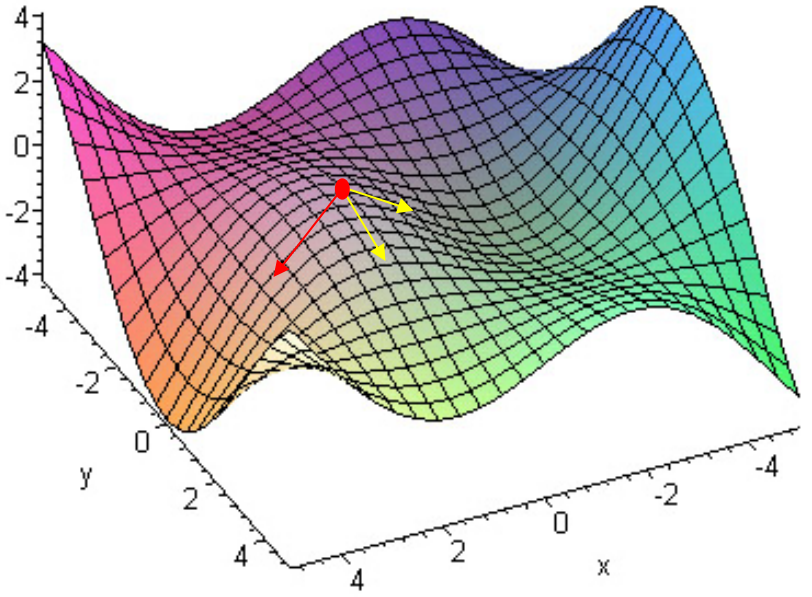
- In other word, a differentiable function ***F*** **decreases fastest** if one moves in the direction of the **negative gradient** of ***F*** at point ***x***.

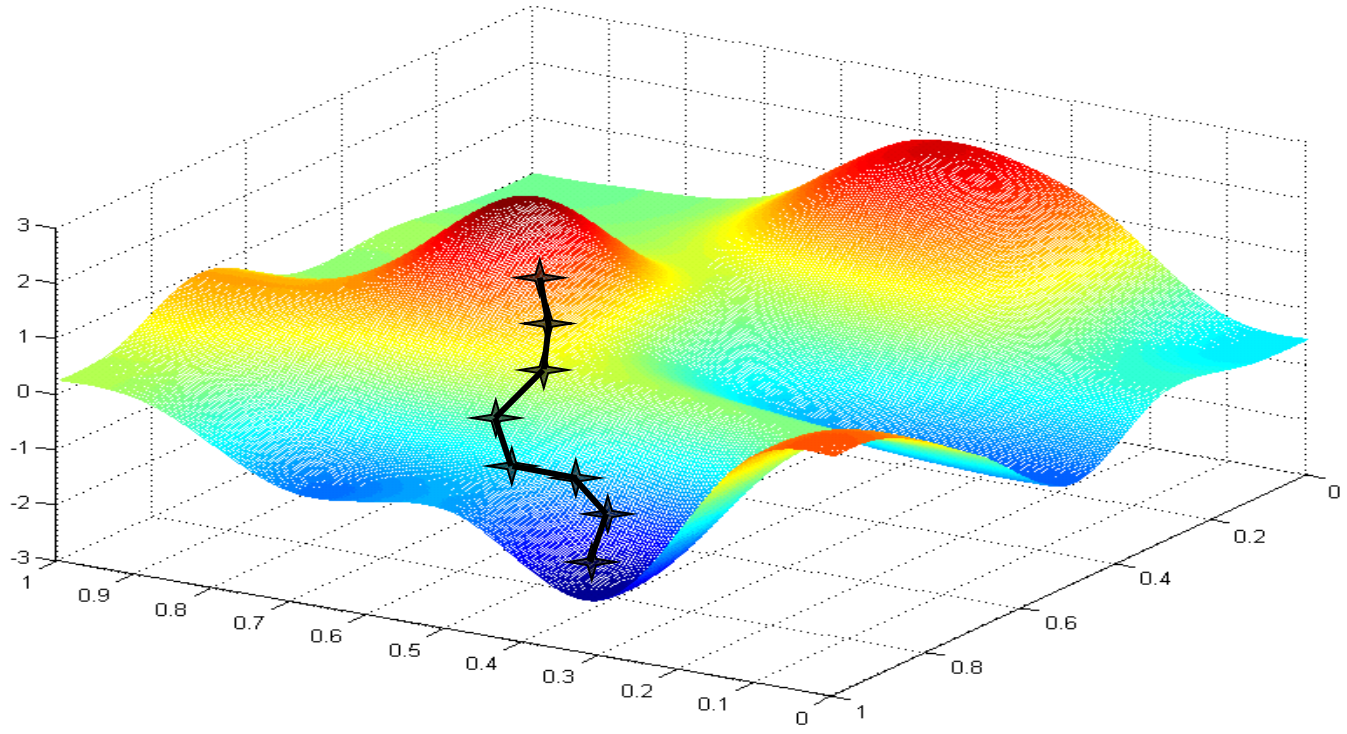


# Gradient Descent Algorithm*

- Gradient Descent is an iterative optimization approach that tries to minimize a function.

- To minimize a function, Gradient Descent starts with an initial set of parameter values, and then iteratively takes steps with a predefined rate in the **negative direction of the function gradient** at the current point (The Steepest Direction).






* Reference: Andrew Ng, Machine Learning, Stanford University.

*Thank You!*

**Questions?**