Neural Network Introduction

Harpreet (Spring 2024)

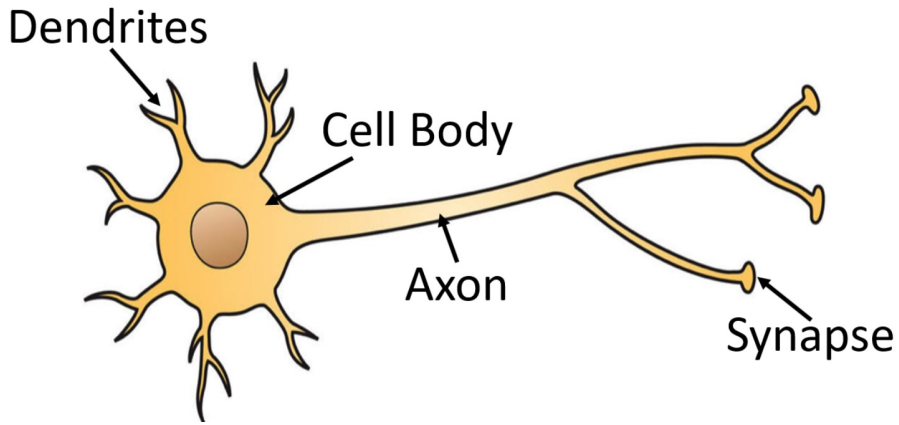
Artificial Neural Network (Feed Forward neural Network

Biological Neural Networks

Biological neurons refer to a collection of interconnected nerve cells organized in layers, which communicate with one another when specific conditions are met.

A biological neuron consists of three primary components:

- Dendrites
- Soma (or cell body)
- Axon



Dendrites receive signals from other neurons.

The soma performs the summation of incoming signals.

Once sufficient input is received, the neuron fires by transmitting a signal over its axon to other cells

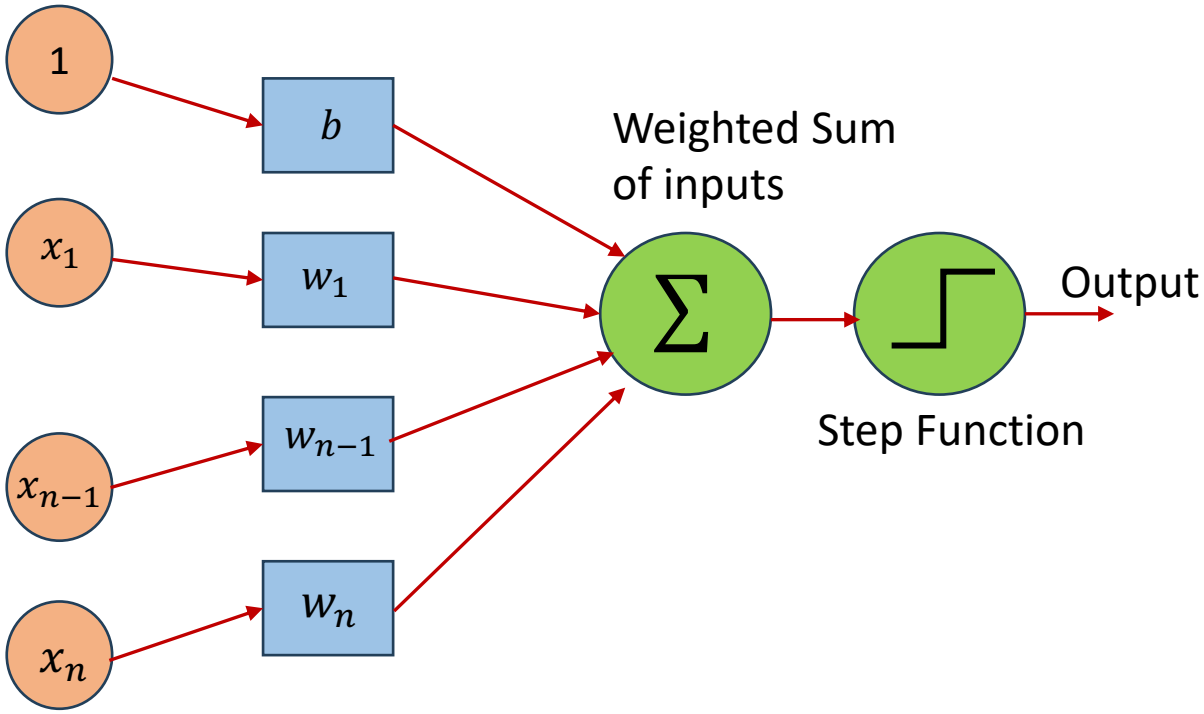
Artificial Neurons

The properties of biological neurons inspire the processing elements in ANN in several ways:

- The processing element receives multiple signals.
- These signals can be adjusted by weights at the receiving connection points.
- The processing element calculates the weighted sum of the inputs.
- When the conditions are appropriate (sufficient input), the neuron transmits a single output.
- The output from a particular neuron can be connected to many other neurons

Perceptron

The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.

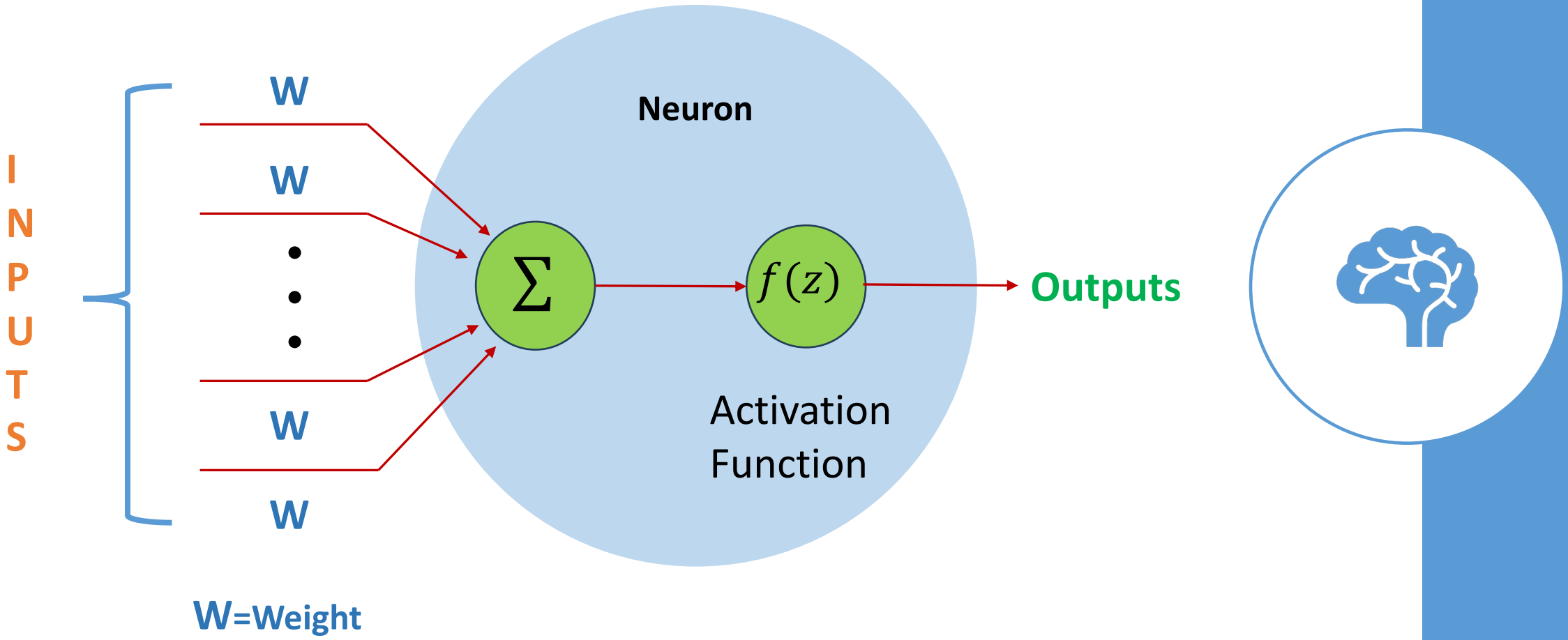


Linear threshold units can be used for simple binary classification.

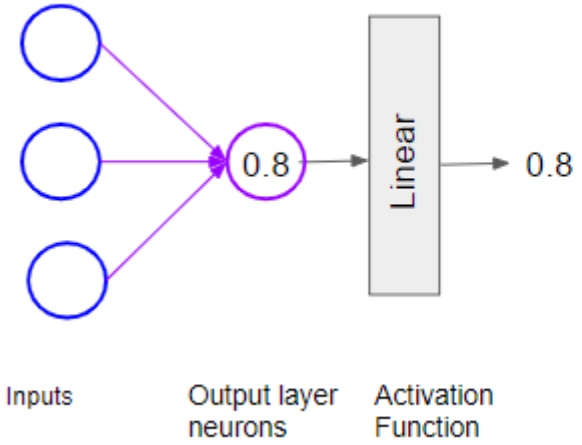
It computes a linear combination of the inputs and if the results exceeds a threshold, it outputs the positive class or else outputs the negative class.

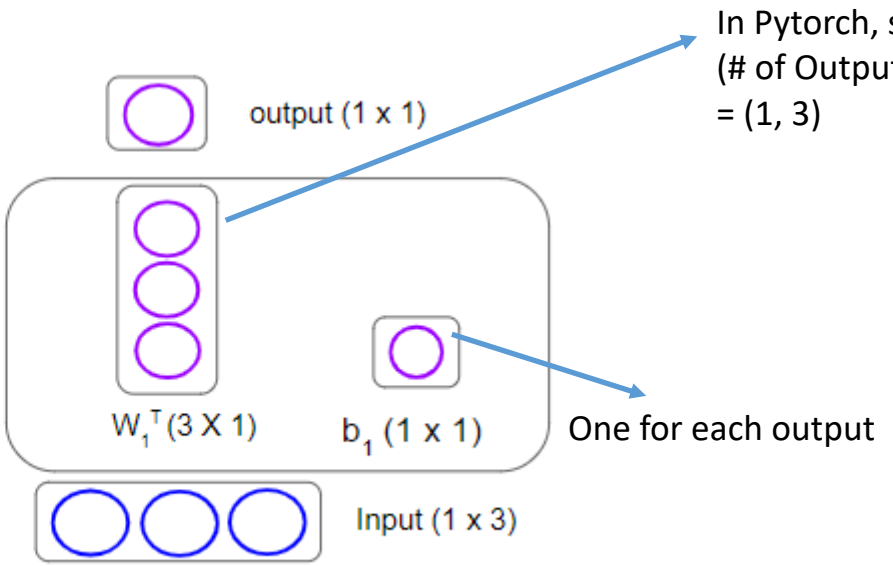
$$f(x) = \begin{cases} 1 & wx + b > 0 \\ 0 & otherwise \end{cases}$$

Artificial Neuron



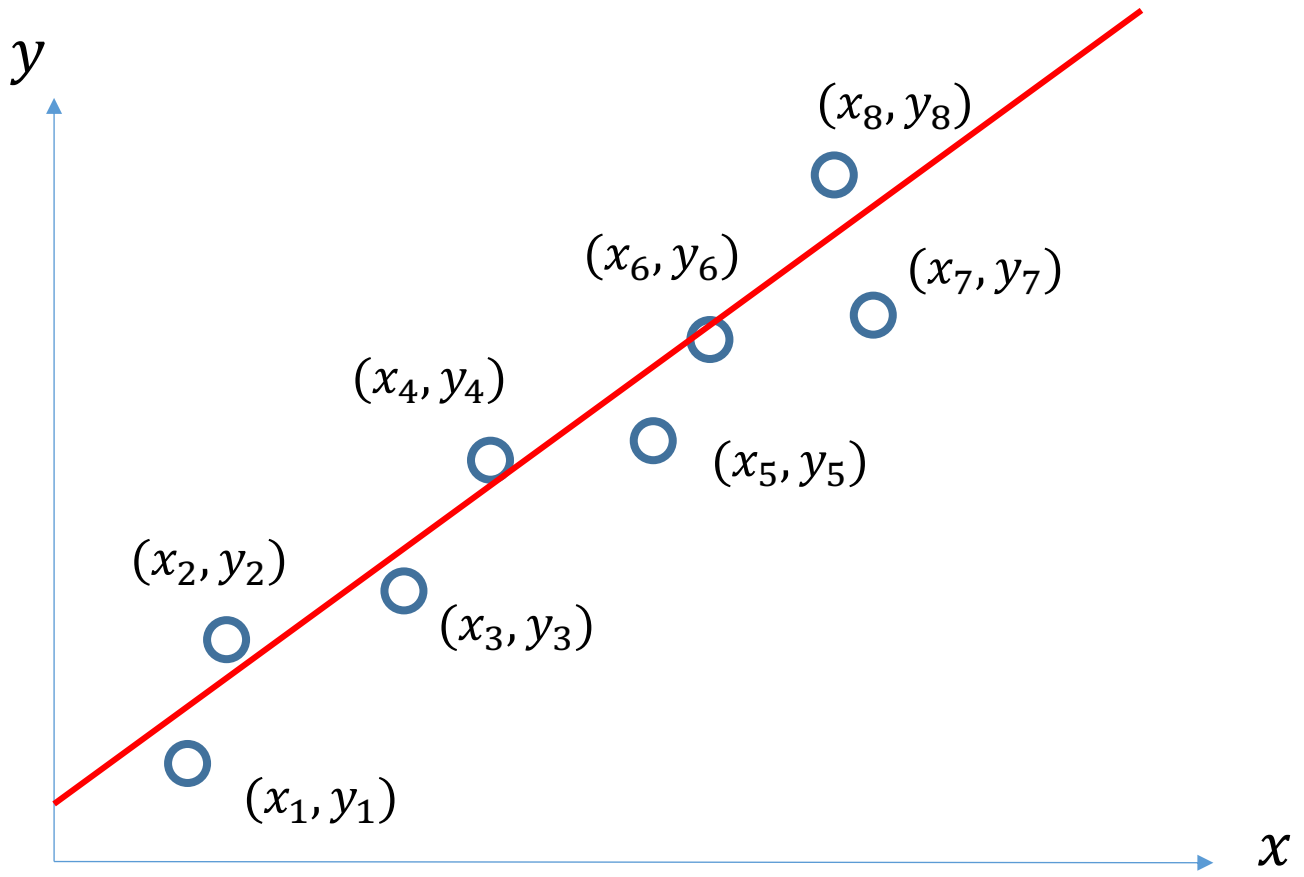
Linear Regression as a Single-layer Neural Network

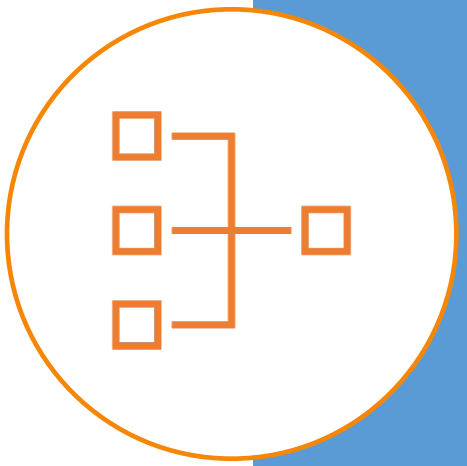




- Number of Neurons in Output Layer: One
- Activation Function for output Layer : Linear (None)
- Prediction $\hat{y} = b + \hat{w}_1 x_1 + \hat{w}_2 x_2 + \cdots + \hat{w}_n x_n$
- Loss Function: Mean Squared Error : $\sum(\hat{y}_i - y_i)^2$

Linear Regression – MSE Loss

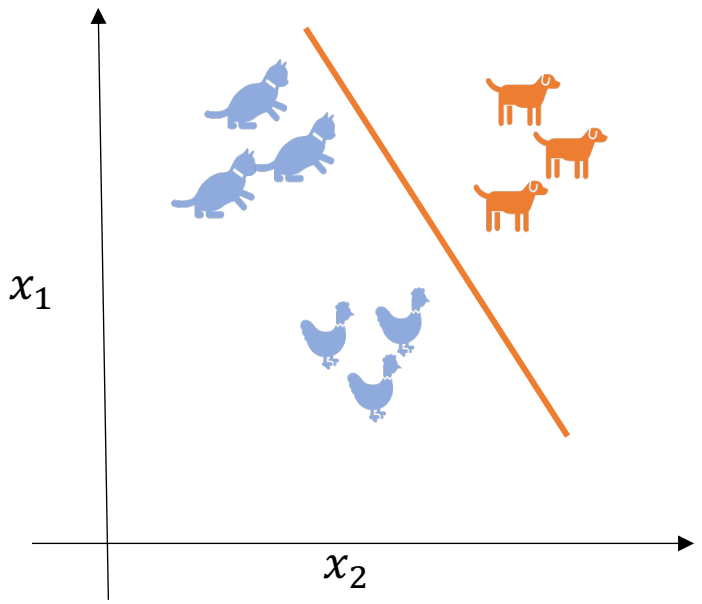




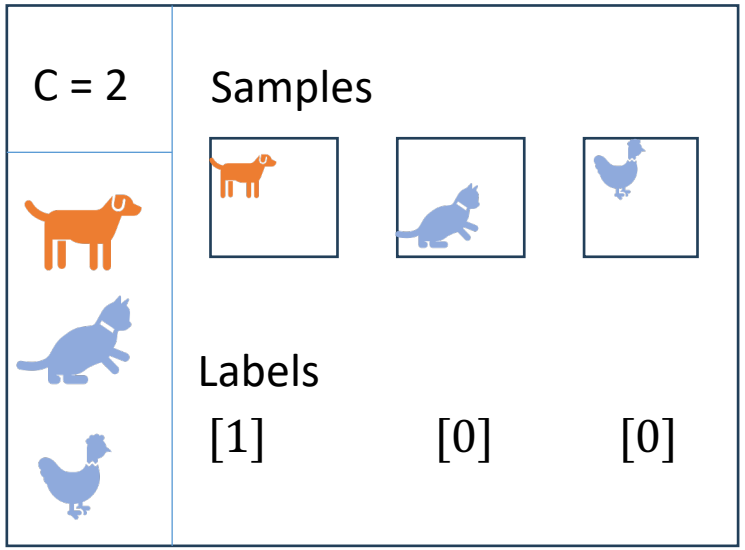
Model: $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x$ $\quad J(\theta) = \sum_{i=1}^{i=8} (\hat{y}_i - y_i)^2$

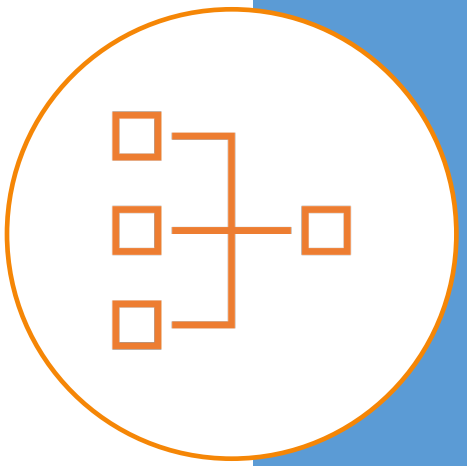
Binary Classification (pick one from two labels)

$$\hat{y} = b + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$



Dog or No Dog





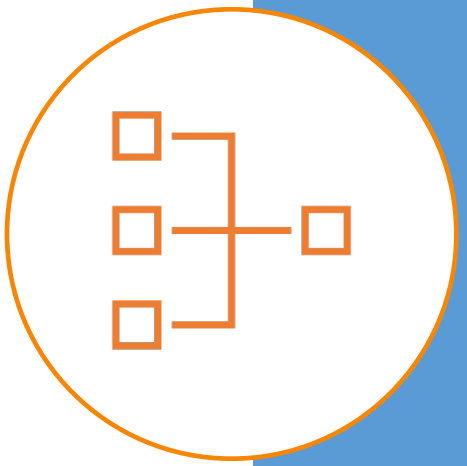
Math (Logistic Regression)

p: probability of class “1”

Need to relate p to predictors with a function that guarantees $0 \leq p \leq 1$

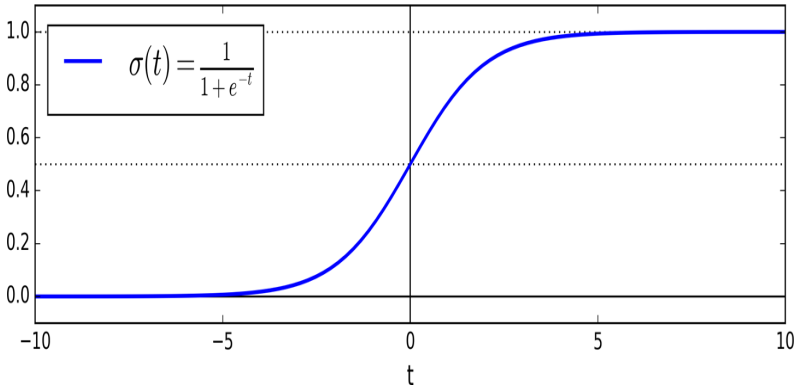
The standard linear function does not

Fix: logistic response function (logit/sigmoid)

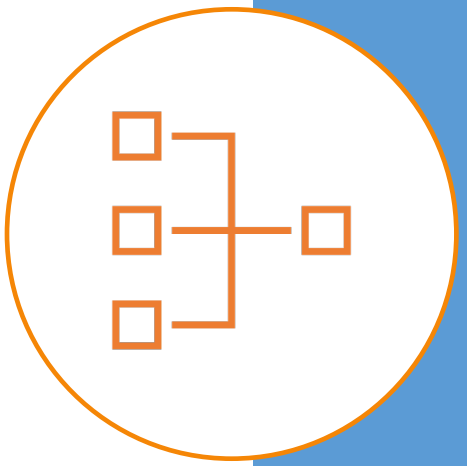


- $\hat{z} = \hat{b} + \hat{w}_1 x_1 + \hat{w}_2 x_2 + \cdots + \hat{w}_n x_n$
- $\hat{p} = \sigma(\hat{z}) = \frac{e^{\hat{z}}}{1+e^{\hat{z}}} = \frac{1}{1+e^{-\hat{z}}}$

Logit (Sigmoid) Function



Logit (sigmoid) function is a function that returns the result between 0 and 1.



Prediction $\hat{y} = \begin{cases} 0 \ \ if \ \hat{p} < threshold \\ 1 \ \ if \ \hat{p} \geq threshold \end{cases}$, typically, $threshold$ = 0.5

$\hat{p} < 0.5$, $when \ z < 0$

$\hat{p} \geq 0.5 \ when \ z \geq 0$

This means Logistic Regression predicts 1 if z is positive and 0 if it is negative

Binary Cross Entropy Loss/Logistic Loss function

Loss function single training instance :

$$c(\theta) = \begin{cases} -\log(\hat{p}) & if\ y = 1 \\ -\log(1-\hat{p}) & if\ y = 0 \end{cases}$$

Intuition

$c(\theta) = 0\ , if\ \hat{p} = y,$

$c(\theta) \to \infty$, if $y = 1\ and\ \hat{p} \to 0$

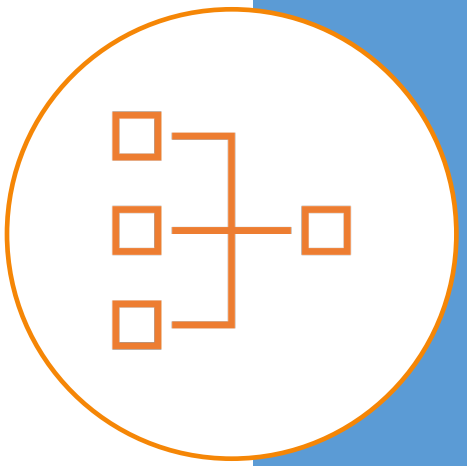
$c(\theta) \to \infty$, if $y = 0\ and\ \hat{p} \to 1$

Alternatively:

$$c(\theta) = -y\log(\hat{p}) - (1-y)\log(1-\hat{p})$$

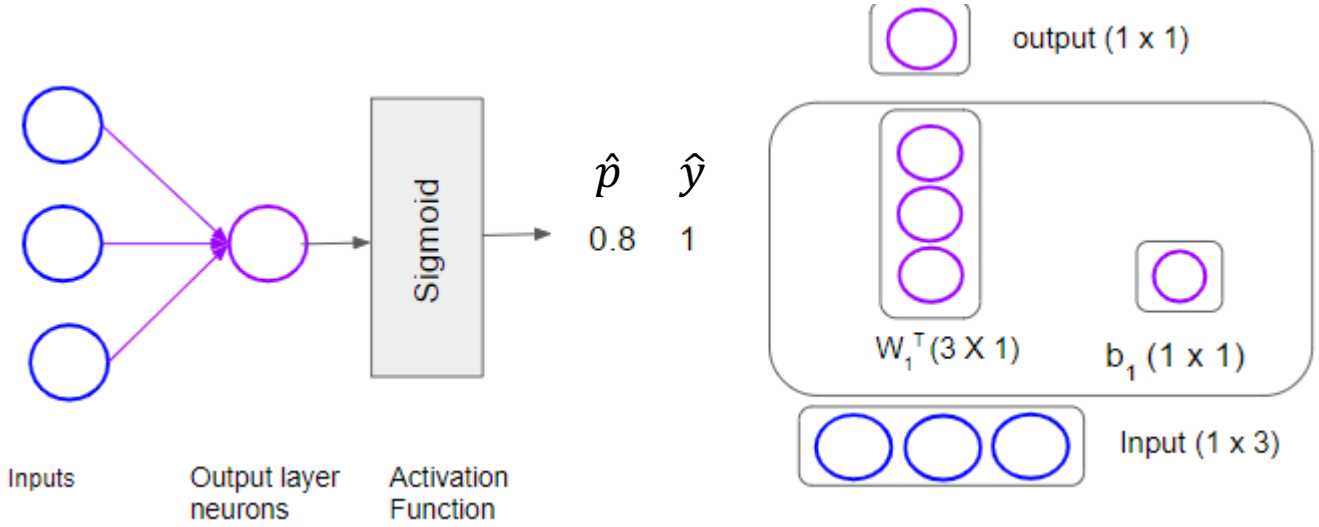
The cost function over the whole triaging set:

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



Intuition: Minimizing the loss will maximize the probability of the true label (class)

Logistic Regression as a Single-layer Neural Network



Number of Neurons in Output Layer: One

Activation Function for output Layer : Sigmoid

Prediction $\hat{y} = \begin{cases} 0 \ if \ \hat{z} < 0 \\ 1 \ if \ \hat{z} \geq 0 \end{cases} = \begin{cases} 0 \ if \ \hat{p} < 0.5 \\ 1 \ if \ \hat{p} \geq 0.5 \end{cases}$

Loss Function : Binary Cross Entropy Loss (Logistic Loss Function)

Loss Calculations for Logistic Regression

Let's assume the following weights and bias for the model:

- Weights: $w_1 = 0.2, w_2 = 0.5, w_3 = -0.3$
- Bias: $b = -0.1$
- Input features: $x_1 = 1, x_2 = 2, x_3 = 0$

The logit (z) is calculated as:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b$$

$$z = (0.2 \cdot 1) + (0.5 \cdot 2) + (-0.3 \cdot 0) + (-0.1) = 0.2 + 1.0 + 0.0 - 0.1 = 1.1$$

Using the sigmoid function, we convert the logit to a probability ($\hat{p}$):

$$\hat{p} = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-1.1}} \approx 0.75$$

Assuming the actual label for the email is "Not Spam" ($y = 0$), we can calculate the Binary Cross-Entropy Loss for this observation using the formula:

$$L(y, \hat{p}) = -[y \cdot \log(\hat{p}) + (1 - y) \cdot \log(1 - \hat{p})]$$

$$L(0, 0.75) = -[0 \cdot \log(0.75) + (1 - 0) \cdot \log(1 - 0.75)] \approx 0.2877$$

Loss Calculations for Logistic Regression

Imagine we have three emails, where:

- Email 1 is Spam (y=1), and the model predicts it as Spam with a probability of 0.9 ($\hat{p}_1 = 0.9$).
- Email 2 is Not Spam (y=0), but the model incorrectly identifies it as Spam with a probability of 0.3 ($\hat{p}_2 = 0.3$).
- Email 3 is Spam (y=1), and the model is less sure, giving a probability of 0.6 ($\hat{p}_3 = 0.6$).

The Binary Cross-Entropy Loss for these predictions would be:

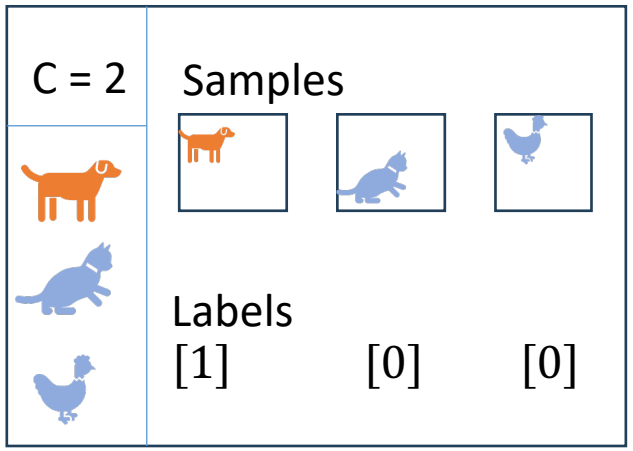
$$L = -\frac{1}{3}\left[\log(0.9) + \log(1 - 0.3) + \log(0.6)\right]$$

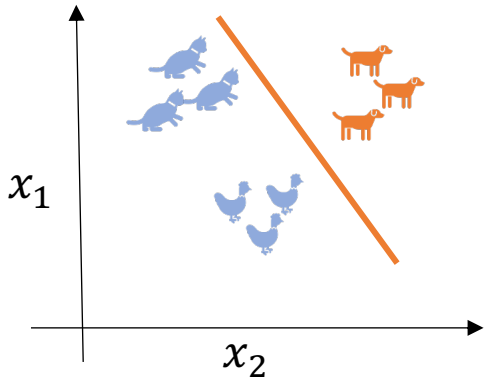
Intuition: Minimizing the loss will maximize the probability of the true label (class)

Multiclass Classification (Softmax Regression) (pick one from more than two labels)

Binary Classification

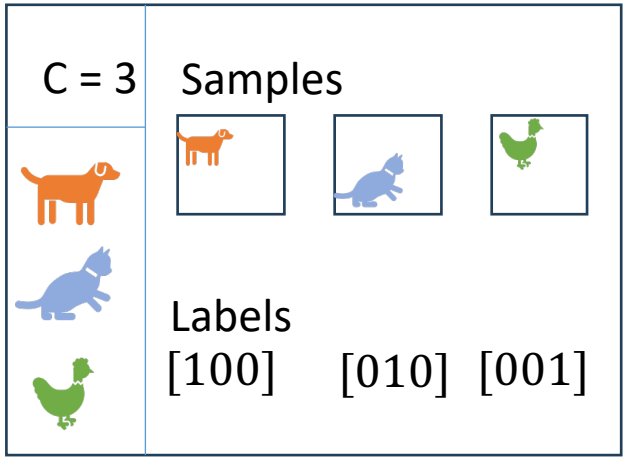
Dog or No Dog

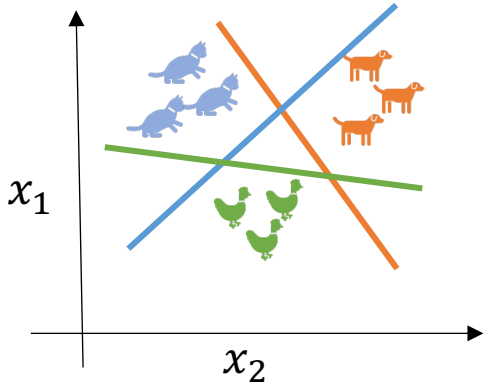




Multi-class Classification

Dog or Cat or Hen





Multiclass Classification (Softmax Regression) (pick one from more than two labels)

Generalization of Logistic regression:

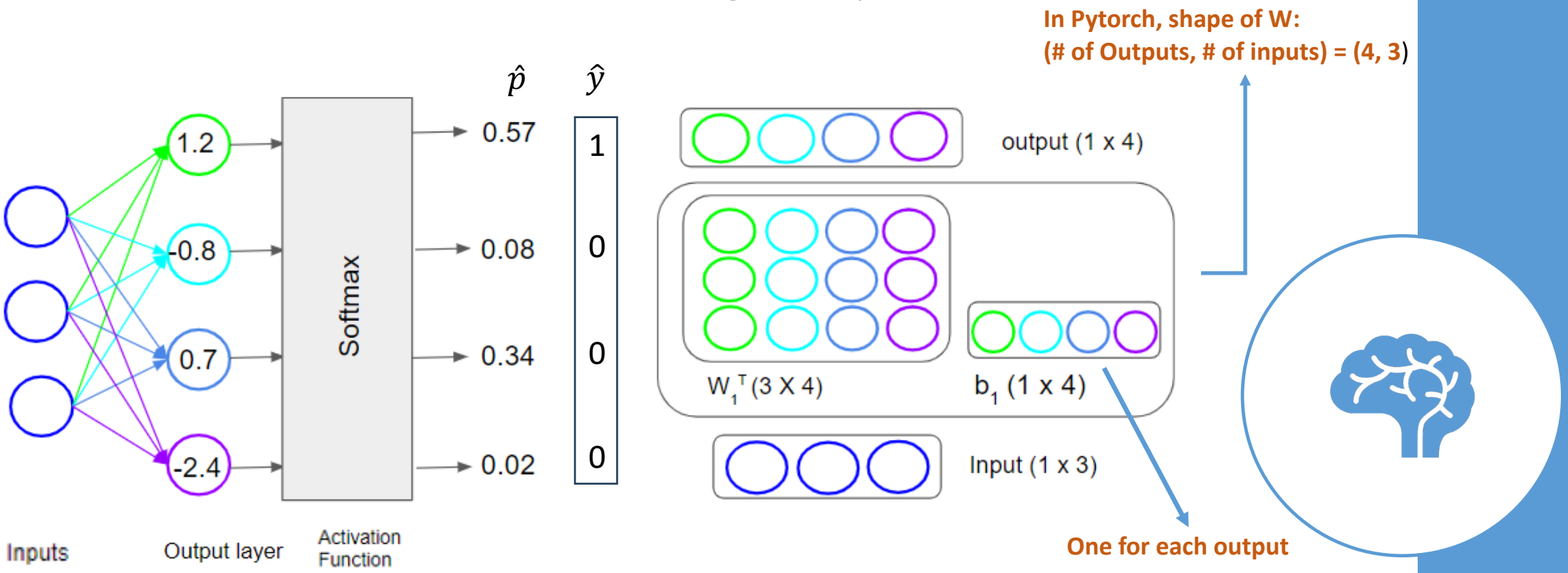
$\hat{z}_k(x) = \hat{b}^k + \hat{w}_1^k x_1 + \cdots + \hat{w}_n^k x_n$, each class has set of weights and bias.

Softmax Function:

$$\hat{p}_k = \frac{e^{z_k(x)}}{\sum_{j=1}^{k} e^{z_j(x)}}, for\ k = 2:\ p_1 = \frac{e^{z_1(x)}}{e^{z_1(x)} + e^{z_2(x)}}, p_2 = \frac{e^{z_2(x)}}{e^{z_1(x)} + e^{z_2(x)}}$$

$$\hat{y} = \underset{k}{argmax}\ \hat{z}_k(x) = \underset{k}{argmax}\ \hat{p}_k(x)$$

Multiclass Classification as a Single-layer Neural Network



Number of Neurons in Output Layer: Number of Classes

Activation Function for output Layer : Softmax Function

Prediction $\hat{y} = \underset{k}{argmax}\, z_k(x)$ $(Class\ which\ has\ the\ maximum\ probbaility\ value\ or\ z_k\ value)$

Loss Function : Cross Entropy Loss

Loss Calculations for MultiClass Classification

Let us assume we have to classify stack exchange post. Each post can have only one category – (1) C, (2) java, and (3) Python posts. We will also assume that there are three inputs (features).

Let's assume the following weights and bias for the model:

- Weights: $w_{1C} = 0.2, w_{2C} = 0.8, w_{3C} = -0.5$ for class C
- Weights: $w_{1Java} = 0.5, w_{2Java} = -0.4, w_{3Java} = 0.3$ for class Java
- Weights: $w_{1Python} = -0.1, w_{2Python} = 0.2, w_{3Python} = 0.7$ for class Python
- Bias: $b_C = -0.1, b_Java = 0.2, b_Python = 0.1$

Input features for a Stack Exchange post:

- $x_1 = 1, x_2 = 2, x_3 = 0$

The logits (z) for each class are calculated as:

- $z_C = w_{1C} \cdot x_1 + w_{2C} \cdot x_2 + w_{3C} \cdot x_3 + b_C$
- $z_{Java} = w_{1Java} \cdot x_1 + w_{2Java} \cdot x_2 + w_{3Java} \cdot x_3 + b_Java$
- $z_{Python} = w_{1Python} \cdot x_1 + w_{2Python} \cdot x_2 + w_{3Python} \cdot x_3 + b_Python$

Loss Calculations for MultiClass Classification

convert these logits to probabilities using the softmax function:

$$\hat{p}_C = \frac{e^{z_C}}{e^{z_C} + e^{z_{Java}} + e^{z_{Python}}}$$
$$\hat{p}_{Java} = \frac{e^{z_{Java}}}{e^{z_C} + e^{z_{Java}} + e^{z_{Python}}}$$
$$\hat{p}_{Python} = \frac{e^{z_{Python}}}{e^{z_C} + e^{z_{Java}} + e^{z_{Python}}}$$

Now, assuming the one-hot encoded true label is $y = [0, 0, 1]$ for Python

$$L(y, \hat{p}) = -[y_C \cdot \log(\hat{p}_C) + y_{Java} \cdot \log(\hat{p}_{Java}) + y_{Python} \cdot \log(\hat{p}_{Python})] = -\log(\hat{p}_{Python})$$

Loss Calculations for MultiClass Classification

Imagine we have probabilities for three Stack Exchange posts as follows:

- Post 1: Probability of being C, Java, Python: $\hat{p}_1 = [0.2, 0.7, 0.1]$
- Post 2: Probability of being C, Java, Python: $\hat{p}_2 = [0.1, 0.8, 0.1]$
- Post 3: Probability of being C, Java, Python: $\hat{p}_3 = [0.2, 0.2, 0.6]$

And the true classes for these posts are:

- Post 1: C (one-hot encoded as $[1, 0, 0]$)
- Post 2: Java (one-hot encoded as $[0, 1, 0]$)
- Post 3: Python (one-hot encoded as $[0, 0, 1]$)

What is the predicted class for each post?

- Java [0, 1,0]
- Java [0, 1,0]
- Python [0,0,1]

The Cross-Entropy Loss for these predictions would be:

$$L = -\frac{1}{3}\left[\log(0.2) + \log(0.8) + \log(0.6)\right]$$

Predictions - MultiClass

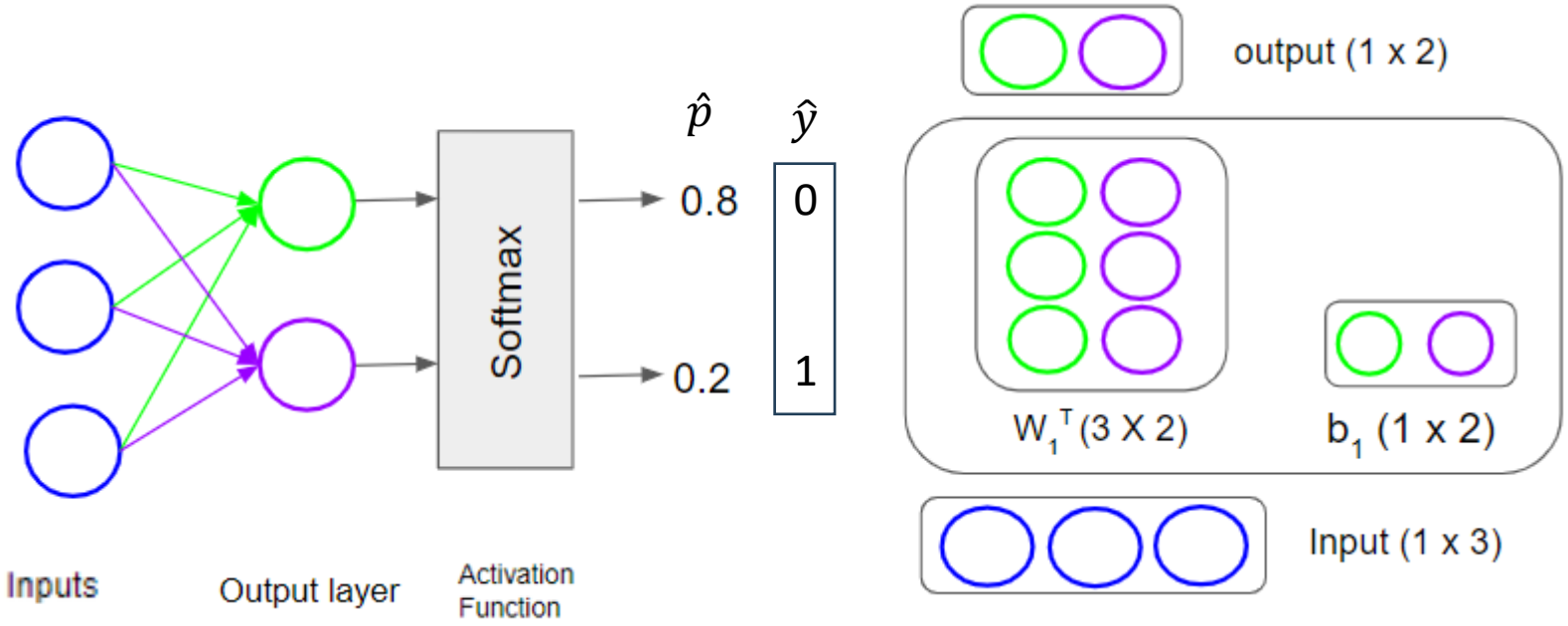
The logits for three different Stack Exchange posts, corresponding to the classes C, Java, and Python, are generated as follows:

- **Post 1 Logits**: [0.4967, -0.1383, 0.6477]
- **Post 2 Logits**: [1.5230, -0.2342, -0.2341]
- **Post 3 Logits**: [1.5792, 0.7674, -0.4695]

Can we get the predicted class labels based on this information? If Yes, what are the predicted labels?

- Python
- C
- C

Binary Classification with Softmax and Cross Entropy Loss function



Number of Neurons in Output Layer: Two

Activation Function for output Layer : Softmax Function

Prediction $\hat{y} = \underset{k}{argmax}\ z_k(x)$ $(Class\ which\ has\ the\ maximum\ probbaility\ value\ or\ z_k\ value)$

Loss Function : Cross Entropy Loss

Negative Log Likelihood Loss/Cross Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{\mathrm{p}}_{\mathrm{k}}^{(\mathrm{i})})$$

For two classes:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y_1^{(i)} log\left(\hat{\mathrm{p}}_1^{(\mathrm{i})}\right) + \mathrm{y}_2^{(\mathrm{i})}\log(\hat{\mathrm{p}}_2^{(\mathrm{i})})$$

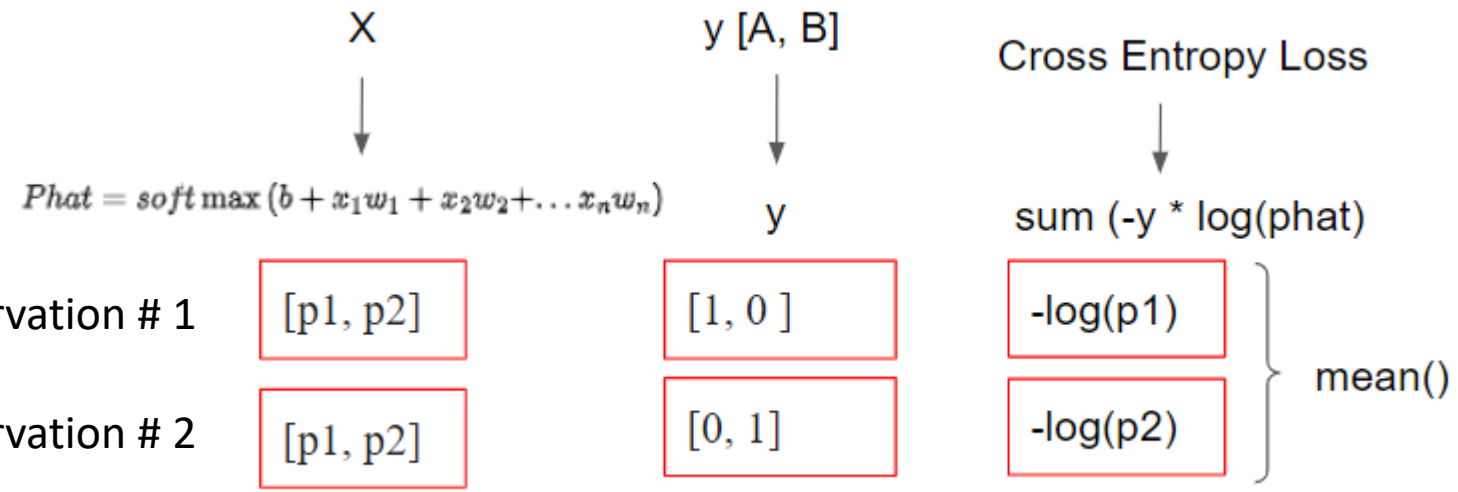
$$y_2 = 1 - y_1$$
$$p_2 = 1 - p_1$$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log\left(\hat{p}^{(i)}\right) + (1 - y^{(i)}) \log\left(1 - \hat{p}^{(i)}\right)\right]$$ Binary Cross Entropy Loss

Negative Log Likelihood Loss/Cross Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

Example for two observations, m = 2 and two classes, k = 2

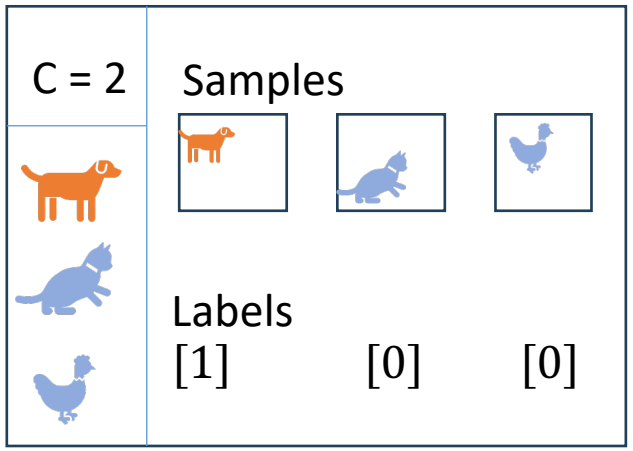


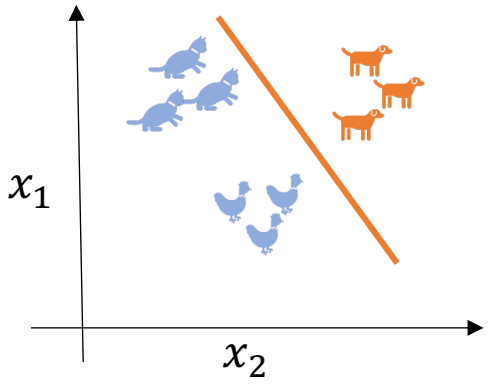
Intuition: Minimizing the loss will maximize the probability of the true label (class)

Multilabel Classification

Binary Classification

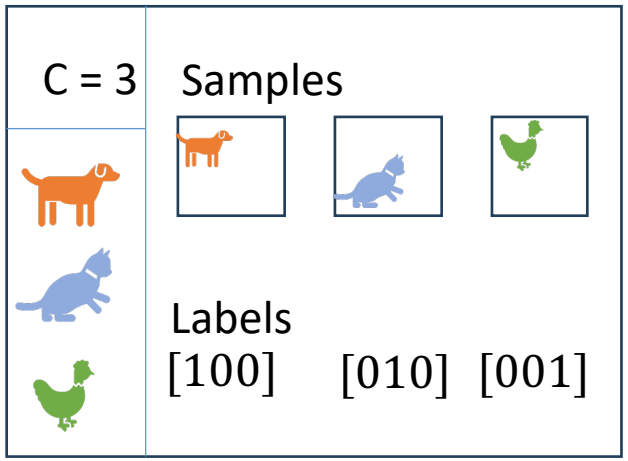
Dog or No Dog

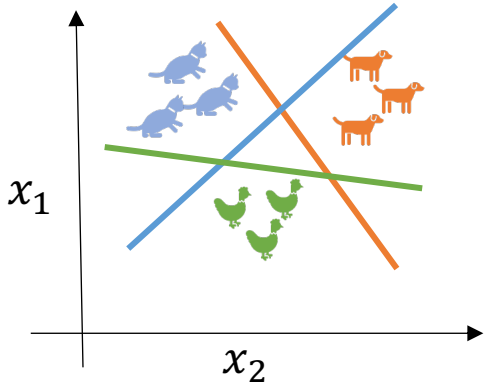




Multi-class Classification

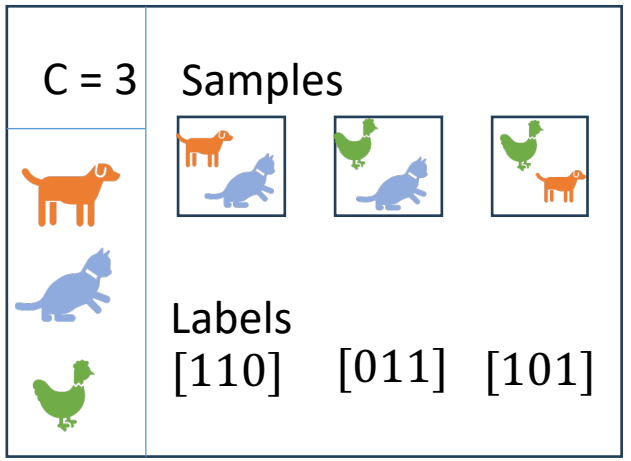
Dog or Cat or Hen

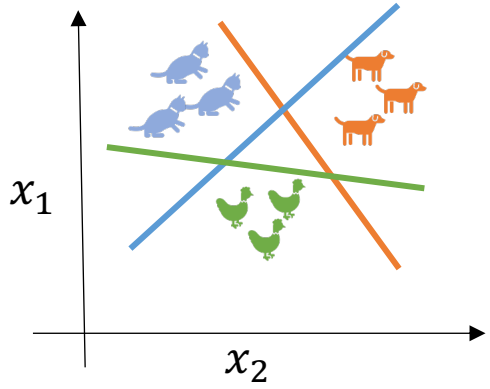




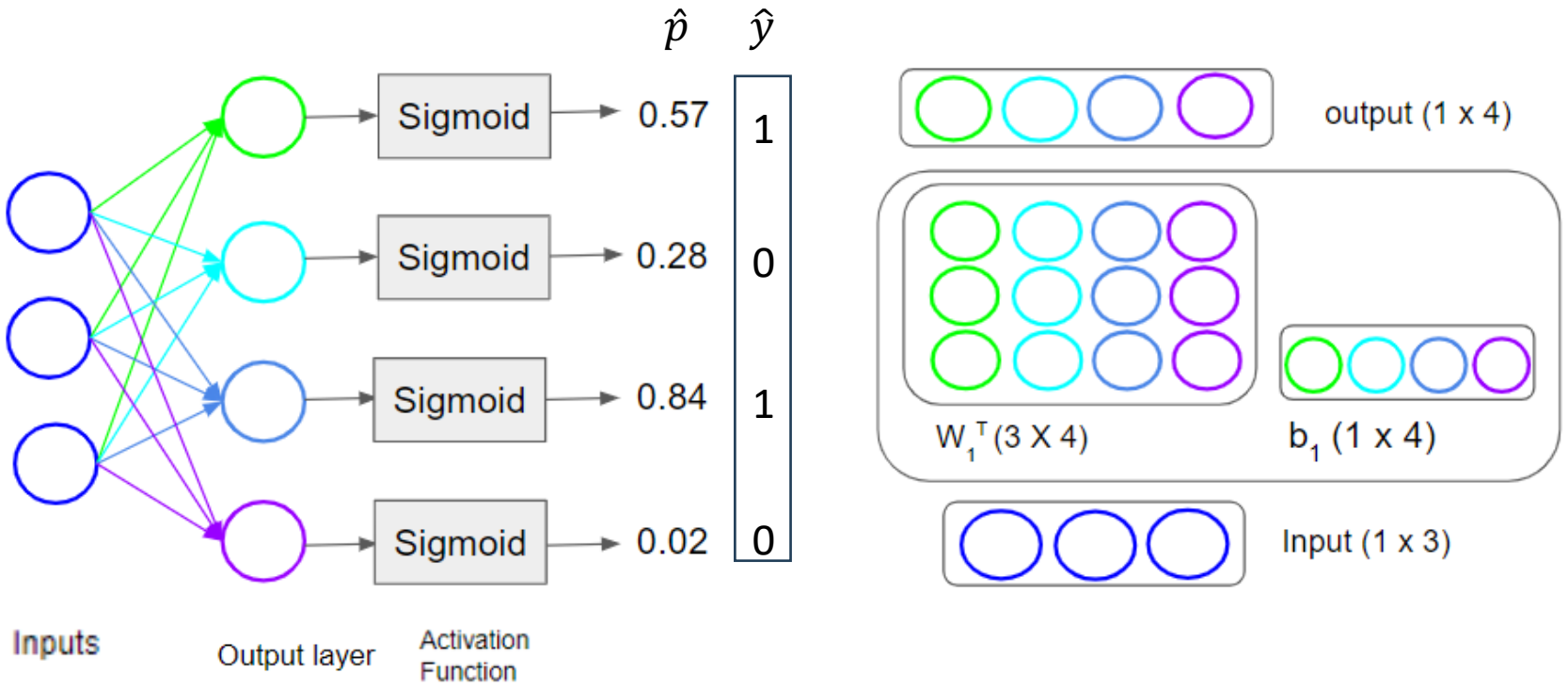
Multi-label Classification

Dog or Cat or Hen





Multilabel Classification as a Single-layer Neural Network



Number of Neurons in Output Layer: Number of Classes

Activation Function for output Layer : Sigmoid

Prediction $\hat{y} = \begin{cases} 0 \ if \ \hat{z} < 0 \\ 1 \ if \ \hat{z} \geq 0 \end{cases} = \begin{cases} 0 \ if \ \hat{p} < 0.5 \\ 1 \ if \ \hat{p} \geq 0.5 \end{cases}$

Loss Function : Binary Cross Entropy Loss (Logistic Loss Function)

Loss Calculations for MultiLabel Classification

Let us assume we have to classify stack exchange post. Each post can have only one category – (1) C, (2) java, and (3) Python posts. We will also assume that there are three inputs (features).

Let's assume the following weights and bias for the model:

- Weights: $w_{1C} = 0.2, w_{2C} = 0.8, w_{3C} = -0.5$ for class C
- Weights: $w_{1Java} = 0.5, w_{2Java} = -0.4, w_{3Java} = 0.3$ for class Java
- Weights: $w_{1Python} = -0.1, w_{2Python} = 0.2, w_{3Python} = 0.7$ for class Python
- Bias: $b_C = -0.1, b_Java = 0.2, b_Python = 0.1$

Input features for a Stack Exchange post:

- $x_1 = 1, x_2 = 2, x_3 = 0$

The logits (z) for each class are calculated as:

- $z_C = w_{1C} \cdot x_1 + w_{2C} \cdot x_2 + w_{3C} \cdot x_3 + b_C$
- $z_{Java} = w_{1Java} \cdot x_1 + w_{2Java} \cdot x_2 + w_{3Java} \cdot x_3 + b_Java$
- $z_{Python} = w_{1Python} \cdot x_1 + w_{2Python} \cdot x_2 + w_{3Python} \cdot x_3 + b_Python$

Loss Calculations for Multi-Label Classification

Then, we convert these logits to probabilities using the sigmoid function:

- $\hat{p}_C = \frac{1}{1+e^{-z_C}}$
- $\hat{p}_{Java} = \frac{1}{1+e^{-z_{Java}}}$
- $\hat{p}_{Python} = \frac{1}{1+e^{-z_{Python}}}$

Assuming the post is related to both C and Python, the true labels vector would be $y = [1, 0, 1]$.

- $L(y, \hat{p}) = -[y_C \cdot \log(\hat{p}_C) + (1 - y_C) \cdot \log(1 - \hat{p}_C) + y_{Java} \cdot \log(\hat{p}_{Java}) + (1 - y_{Java}) \cdot \log(1 - \hat{p}_{Java}) + y_{Python} \cdot \log(\hat{p}_{Python}) + (1 - y_{Python}) \cdot \log(1 - \hat{p}_{Python})]$

$= -[\log(\hat{p}_C) + \log(1 - \hat{p}_{Java}) + \log(\hat{p}_{Python})]$

This accounts for the fact that the post is labeled as C and Python (thus we want to maximize the probabilities $\hat{p}_C$ and $\hat{p}_{Python}$), and not labeled as Java (thus we want to maximize the probability of the negative class for Java, $1 - \hat{p}_{Java}$).

Predictions – Multi-Label

Each list corresponds to the probabilities that the respective post is related to C, Java, and Python, in that order.

- **Post 1 Probabilities**: [0.85, 0.15, 0.65]
- **Post 2 Probabilities**: [0.25, 0.70, 0.40]
- **Post 3 Probabilities**: [0.30, 0.45, 0.80]

Can we get the predicted class labels based on this information? If Yes, what are the predicted labels?

If we assume a threshold of 0.5, the predicted labels are :

Post1: [C, Python]
Post2: [Java]
Post3: [Python]

Predictions – Multi-Label

Each list corresponds to the probabilities that the respective post is related to C, Java, and Python, in that order.

- **Post 1 Probabilities**: [0.85, 0.15, 0.65]
- **Post 2 Probabilities**: [0.25, 0.70, 0.40]
- **Post 3 Probabilities**: [0.30, 0.45, 0.80]

Can we get the predicted class labels based on this information? If Yes, what are the predicted labels?

If we assume a threshold of 0.5, the predicted labels are :

Post1: [C, Python]
Post2: [Java]
Post3: [Python]

Predictions – Multi-Label

Here are sets of logits for three Stack Exchange posts corresponding to the probabilities for C, Java, and Python

- **Post 1 Logits**: [2.0, -1.0, 0.5]
- **Post 2 Logits**: [0.5, 1.5, 0.2]
- **Post 3 Logits**: [-0.5, 0.3, 2.2]

Can we get the predicted class labels based on this information? If Yes, what are the predicted labels?

A threshold of 0.5 for prob corresponds to a threshold of 0 for logits.

When $z = 0$, the sigmoid function simplifies to:

$$\sigma(0) = \frac{1}{1 + e^0} = \frac{1}{2} = 0.5$$

The predicted labels are :
Post1: [C, Python]
Post2: [C, Java, Python]
Post3: [Java, Python]

Linear Classifiers

Predictions in all the previous models can be made based on:

$\hat{z}_k(x) = \hat{b}^k + \hat{w}_1^k x_1 + \cdots + \hat{w}_n^k x_n$

Output equation given x_1 and x_2, is the equation of a line

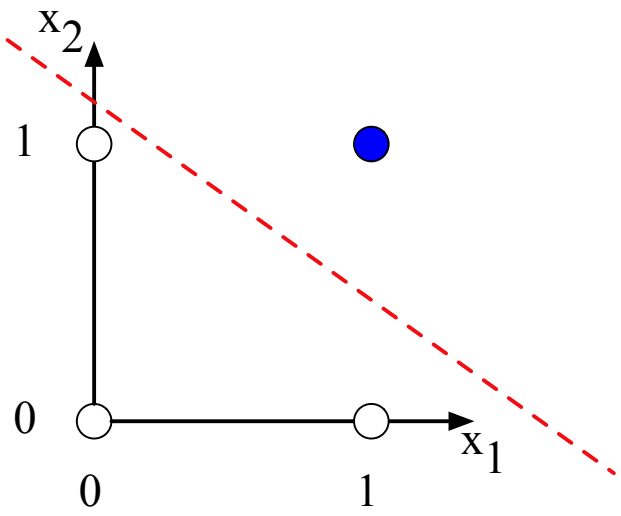
$w_1 x_1 + w_2 x_2 + b = 0$

(in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)

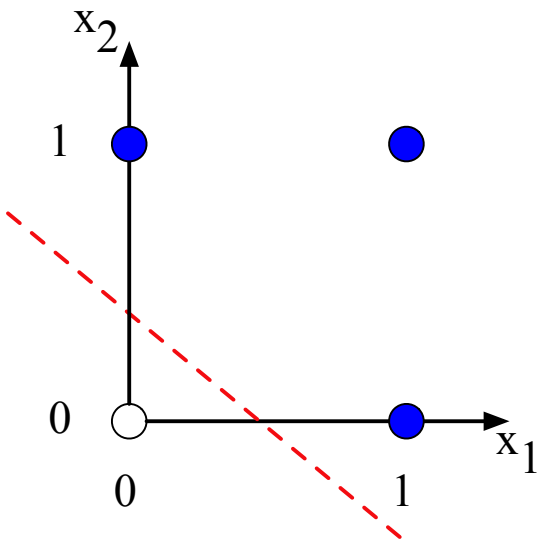
This line acts as a decision boundary

- 0 if input is on one side of the line
- 1 if on the other side of the line

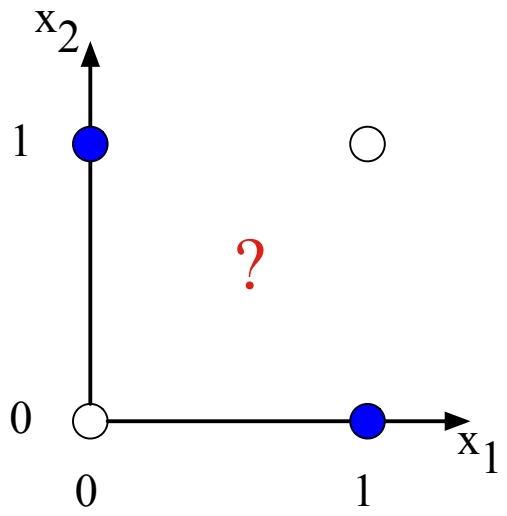
Decision Boundaries



a) x_1 AND x_2



b) x_1 OR x_2

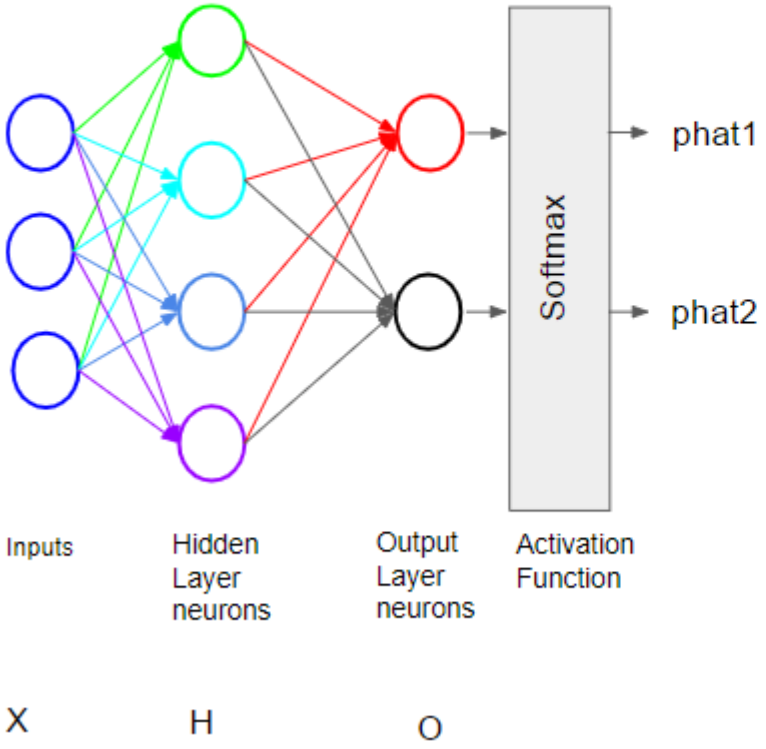


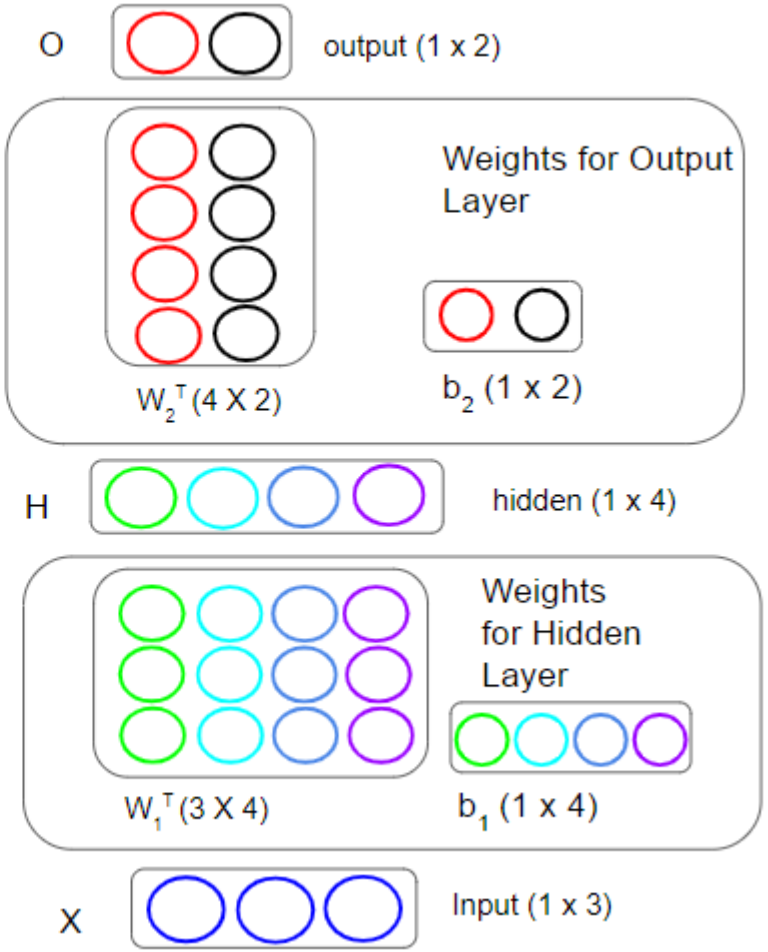
c) x_1 XOR x_2

XOR is not a **linearly separable** function!

Slide Credit: https://web.stanford.edu/~jurafsky/slp3/

Multilayer Perceptron





Multilayer Perceptron

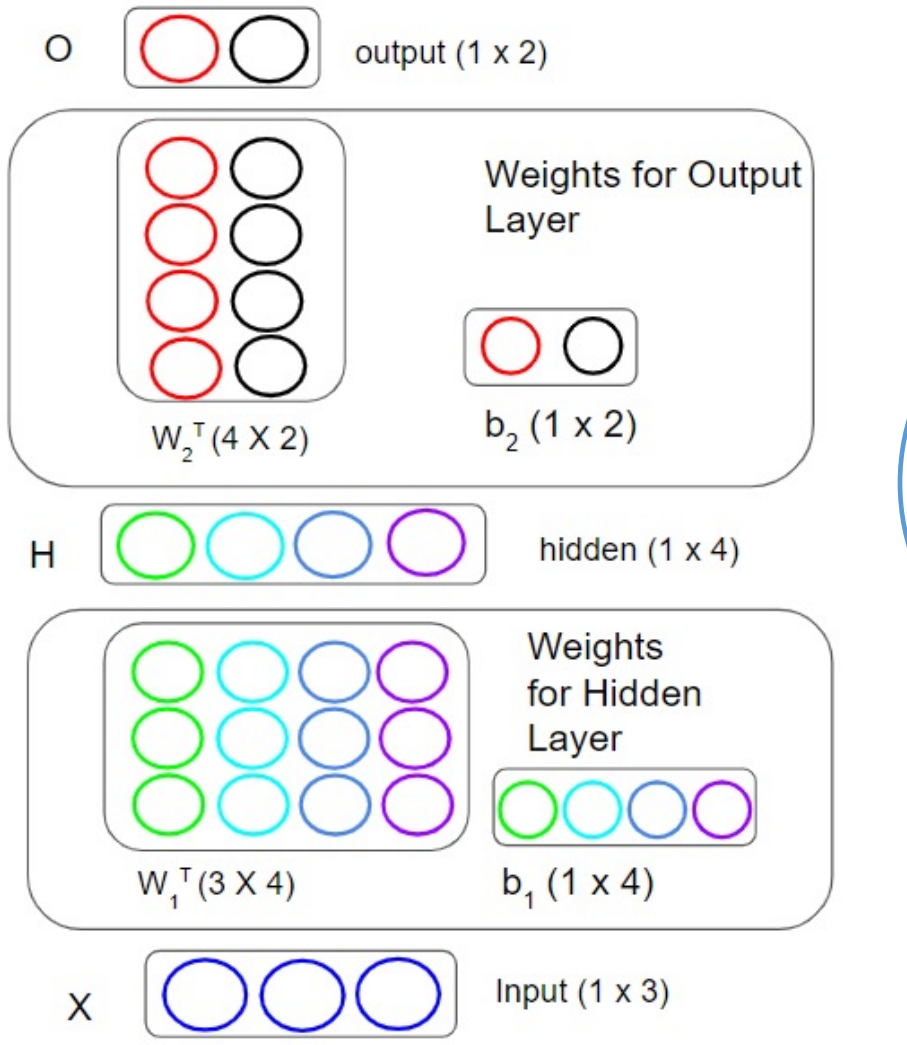
$$H = XW_1^T + b_1$$
$$O = HW_2^T + b_2$$

$$O = (XW_1^T + b_1)W_2^T + b_2$$
$$O = XW_1^T W_2^T + b_1 W_2^T + b_2$$
$$O = XW + b$$

$$W = W_1^T W_2^T$$
$$b = b_1 W_2^T + b_2$$

Apply Non-Linear Function to Hidden Layers

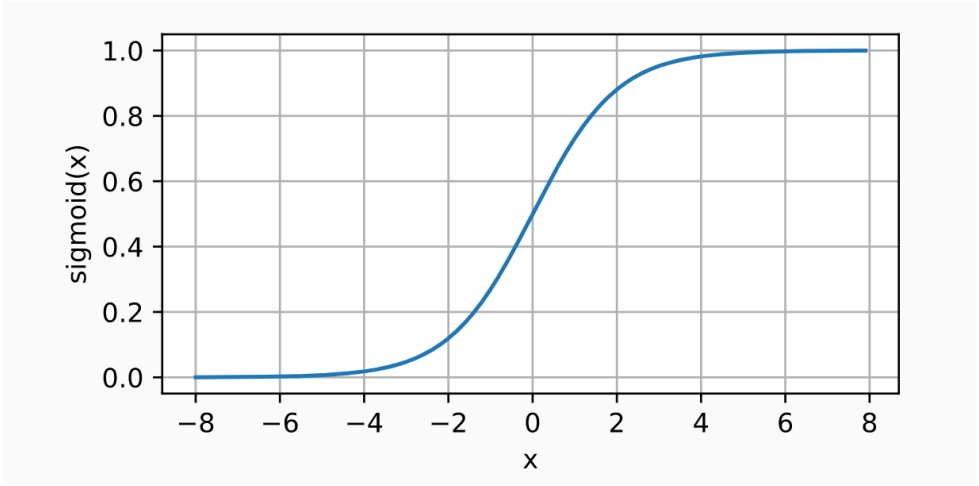
$$H = \sigma(XW_1^T + b_1)$$
$$O = HW_2^T + b_2$$



Sigmoid Activation

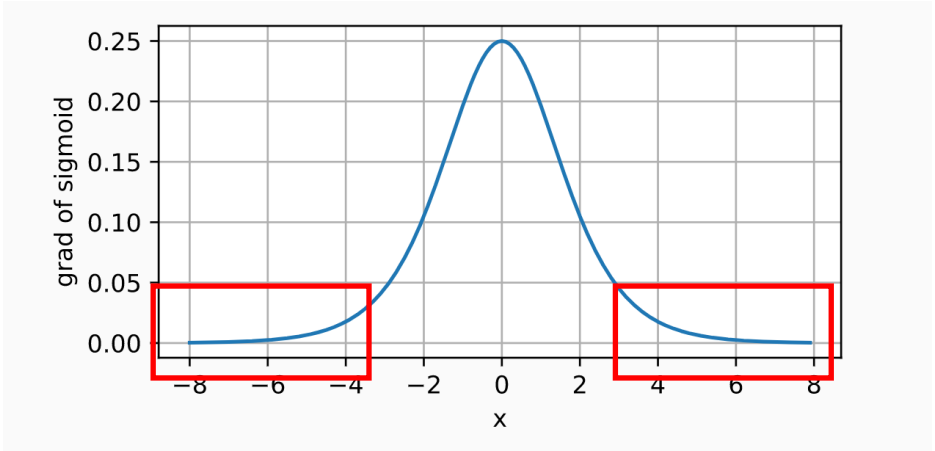
$$sigmoid(x) = \frac{1}{1 + \exp(-x)}$$

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2}$$



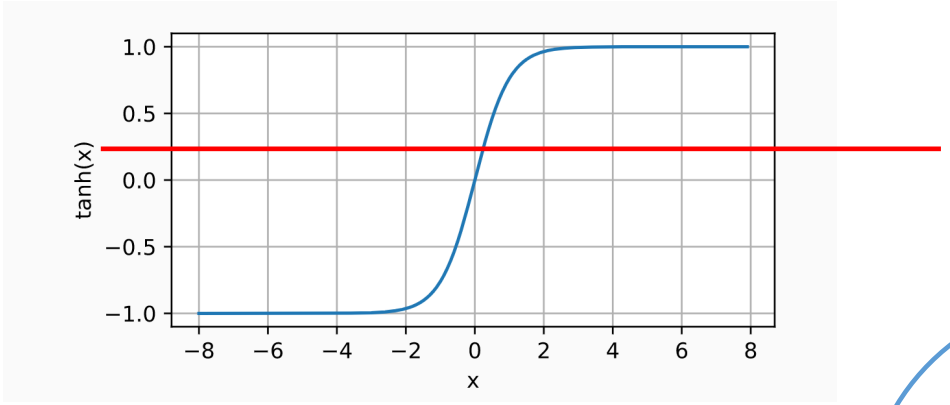
$$\frac{d}{dx} sigmoid(x) = \frac{\exp(-x)}{(1 + \exp(-x)^2}$$

$$= sigmoid(x)(1 - sigmoid(x))$$



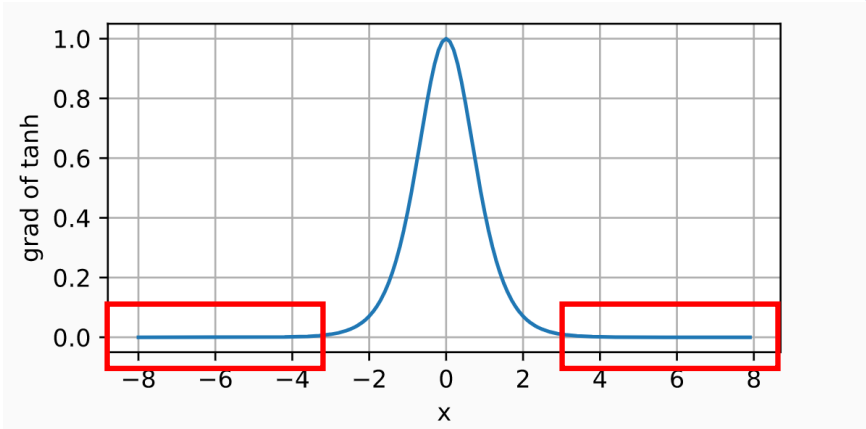
Tanh Activation

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$



- Output value ranges from –1 to 1 (instead of 0 to 1 in the case of the sigmoid function).
- That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

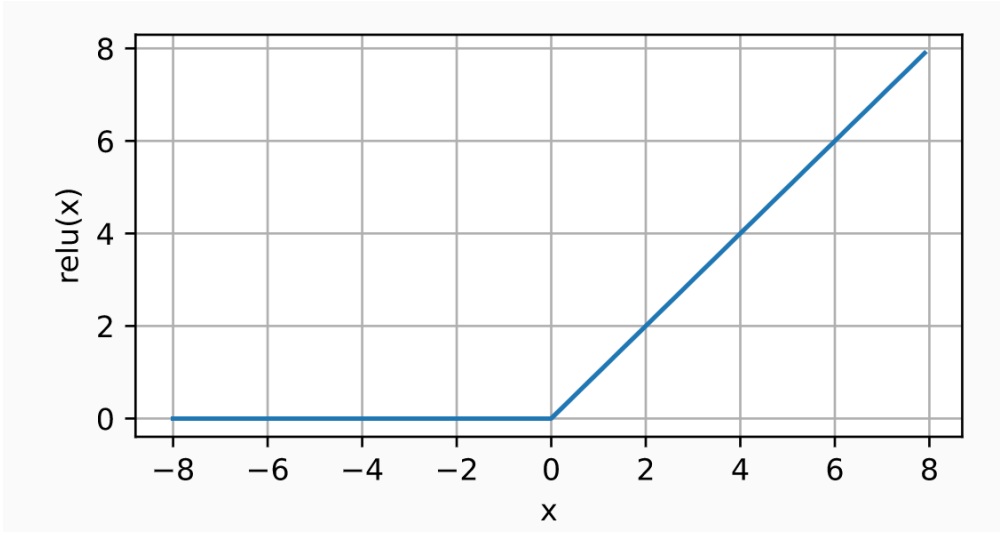
ReLU Activation

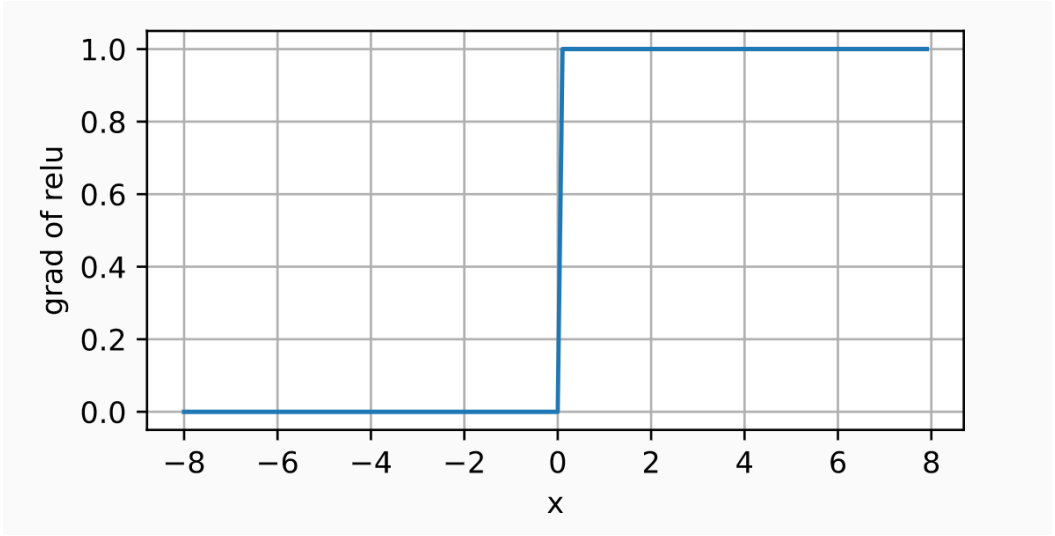
$$ReLU(x) = \max(x, 0)$$

It works very well and has the advantage of being fast to compute, so it has become the default.

$$\frac{d}{dx} ReLU(x) = \begin{cases} 0, if\ x < 0 \\ 1, if\ x \geq 0 \end{cases}$$

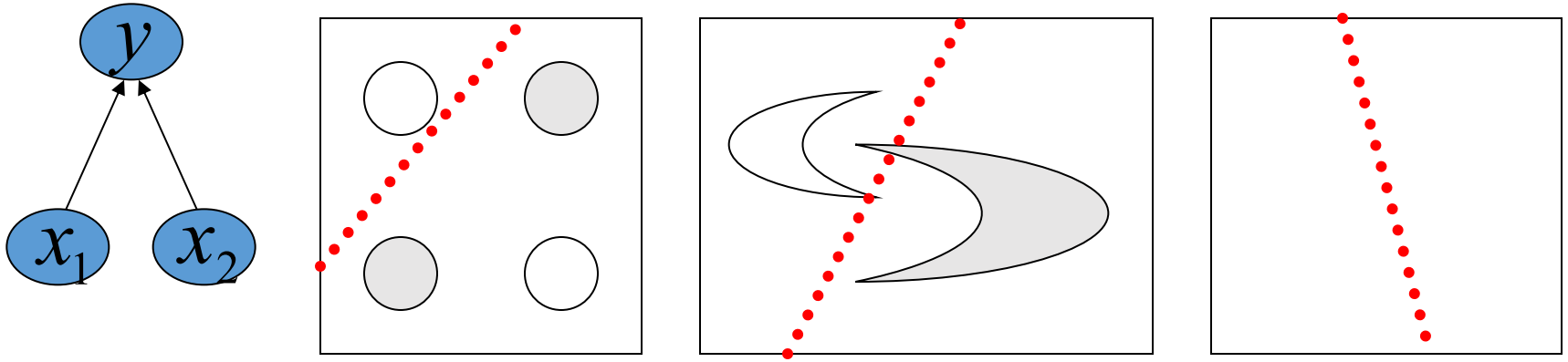
Note: derivative is not defined at $x = 0$





Decision Boundary

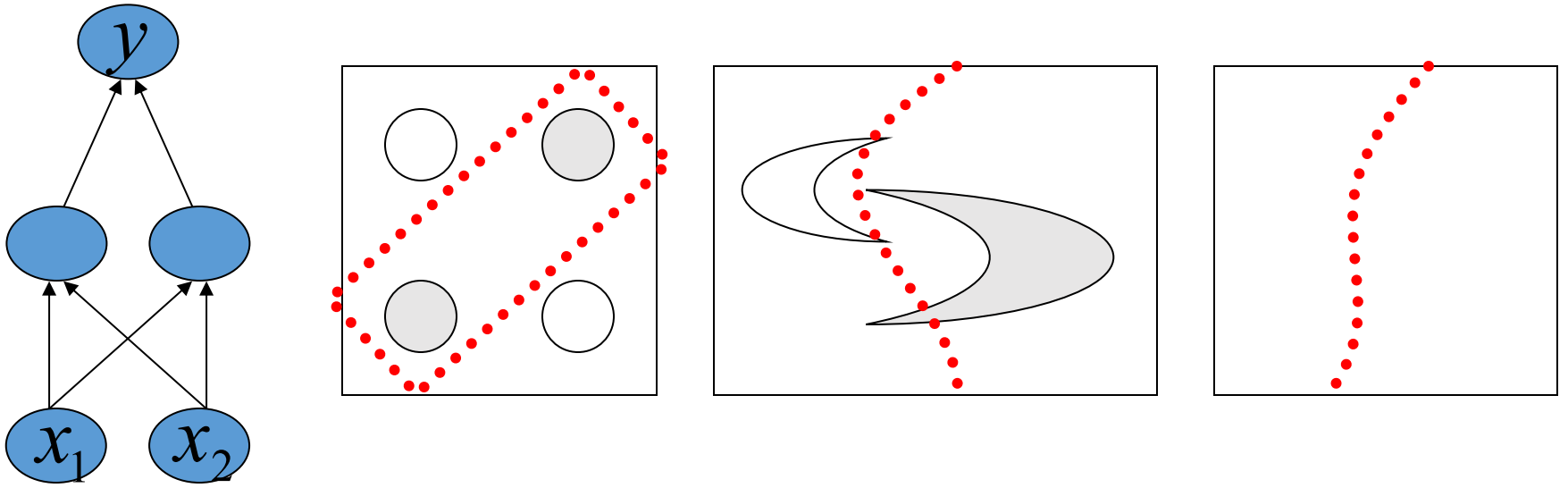
- 0 hidden layers: linear classifier
  - Hyperplanes



Slide Credit: © Eric Xing @ CMU, 2006-2011

Decision Boundary

- 1 hidden layer with nonlinear activation
  - Boundary of convex region (open or closed)

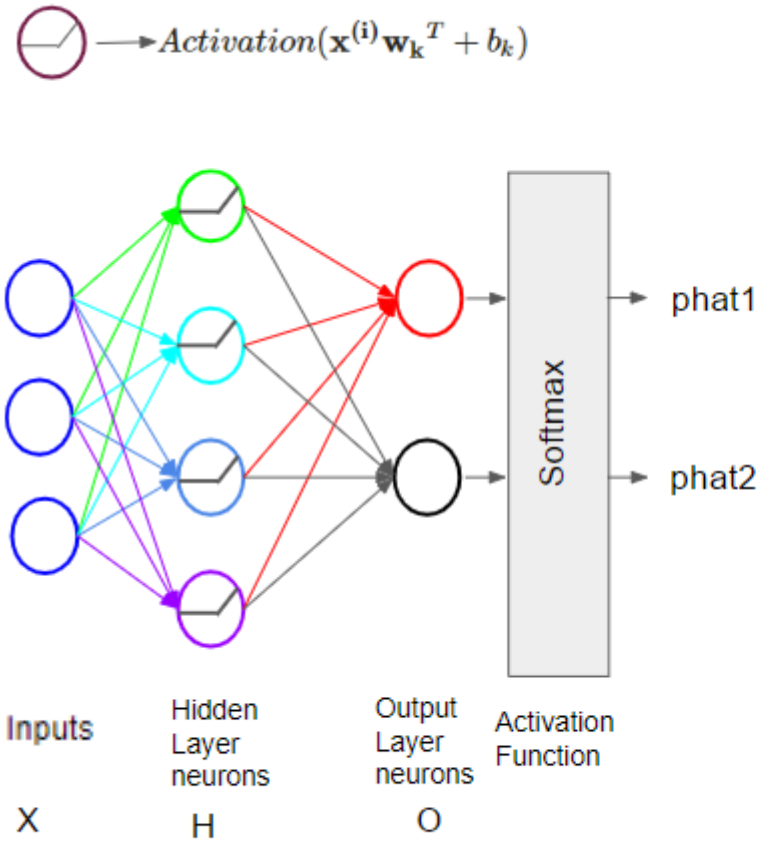


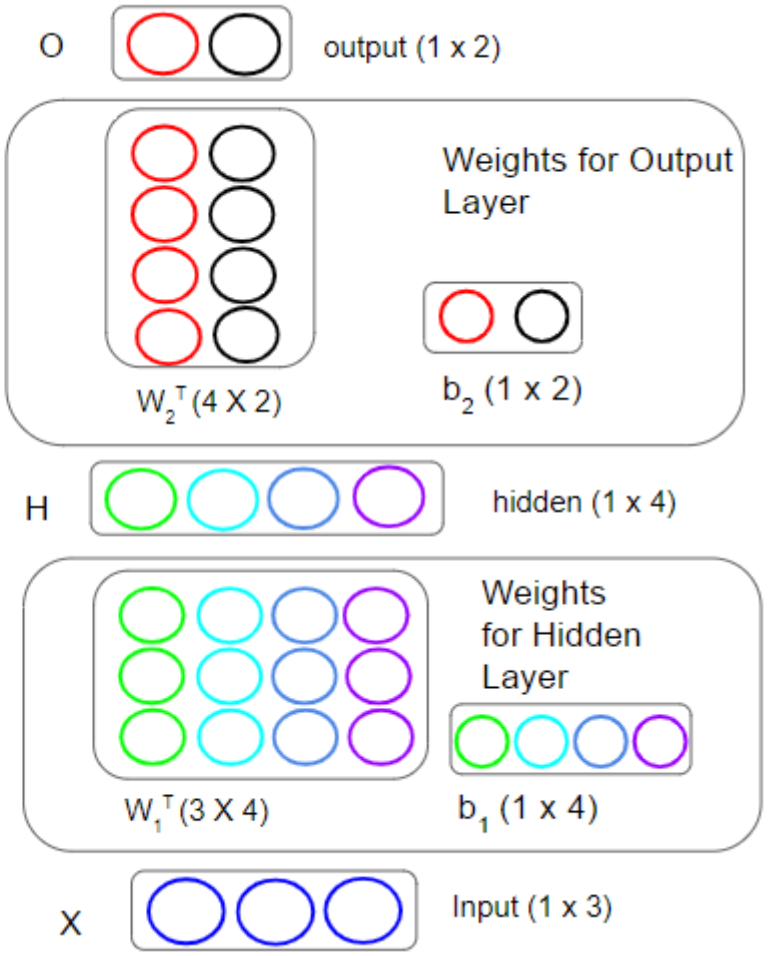
Multiclass Classification

$$H = \sigma(XW_1^T + b_1)$$

$$O = HW_2^T + b_2$$

$$\hat{\mathrm{P}} = Softmax(O)$$





Summary – Layers/Activations/Loss Functions

	Regression	Binary Classification	Multi-class Classification	Multi Label Classification
# input neurons	Depends on the problem (your data) (Number of input variables – (e.g. for 28x28 black & White images = 28*28 =784)			
# hidden layers	Depends on problem (Hyperparameter)			
# neurons per hidden layer	Depends on problem (Hyperparameter)			
# output neurons	One	One (Two)	Number of Classes	Number of Classes
Hidden activation	Hyperparameter (sigmoid, tanh, ReLU, SELU, GELU, MISH etc.)			
Output activation	None	Sigmoid (Softmax)	Softmax	Sigmoid
Loss Function	MSE	Binary Cross Entropy (Cross-Entropy)	Cross-Entropy	Binary Cross Entropy

Training Neural Network

Gradient Descent

SGD – Linear Regression

$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x_1$

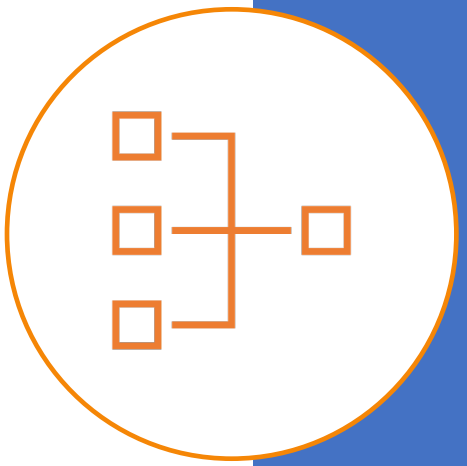
Let us assume $\theta_0 = 0$

X	Y
1	1
2	2
3	3

$\hat{y} = h_\theta(x) = \theta_1 x_1$

Cost Function: $J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

Goal : Find value of θ_1 that minimizes $J(\theta_1)$



Least Squares Normal Equation

Cost Function:

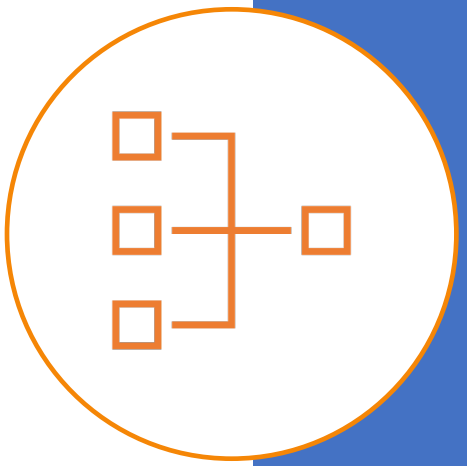
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Matrix Notation:

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

Normal Equation (Solution to minimizing the cost function with respect to θ):

$$\theta = (X^T X)^{-1} X^T y$$



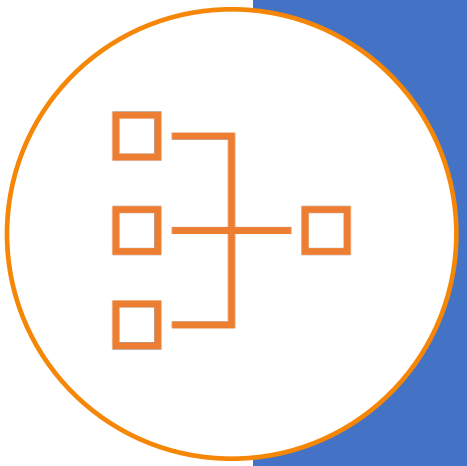
Note : Feature scaling do not change prediction or accuracy. No need to do feature scaling if we are using Normal Equation without regularization.

Proofs are available at these links for those who are interested:
https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)
http://eli.thegreenplace.net/2014/derivation-of-the-normal-equation-for-linear-regression

Normal Equation Pros and Cons

Cons:

- The computational complexity of linear regression is very high
- There is no controlling parameters



Pros:

- Prediction is very fast

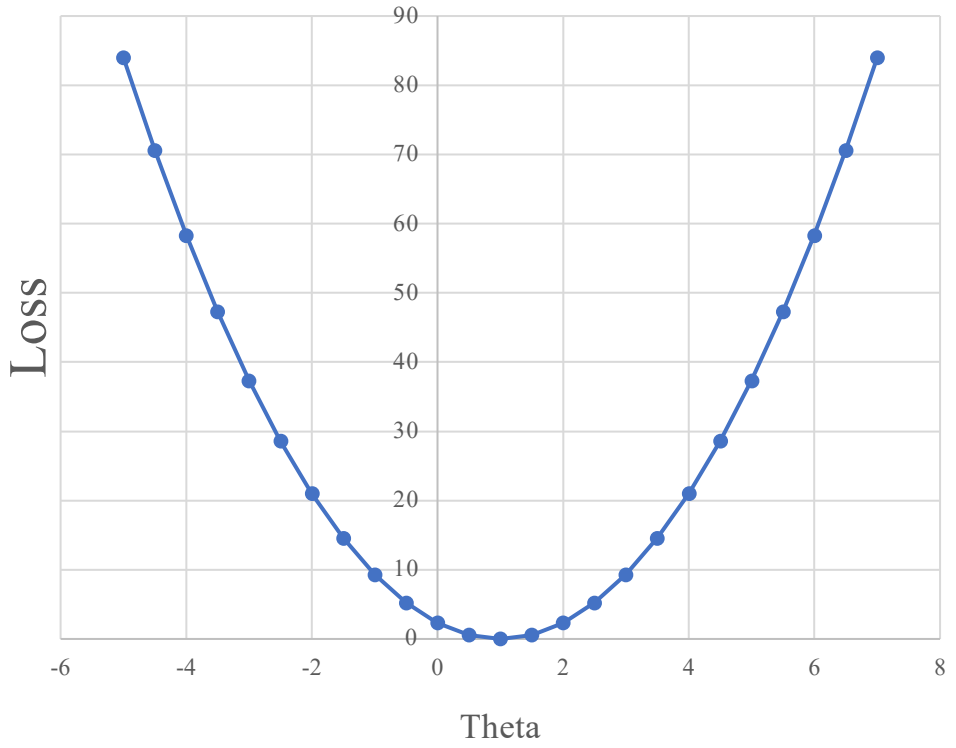
Loss Function

$h_\theta(X)$: For fixed theta this is a function of x

X	Y	$h(X\|\theta = 1)$	$h(X\|\theta = 2)$
1	1	1	2
2	2	2	4
3	3	3	6

$J(\theta_1)$ (function of parameter θ)

θ	Cost
-1	9.33
0	2.33
1	0
2	2.33
3	9.33



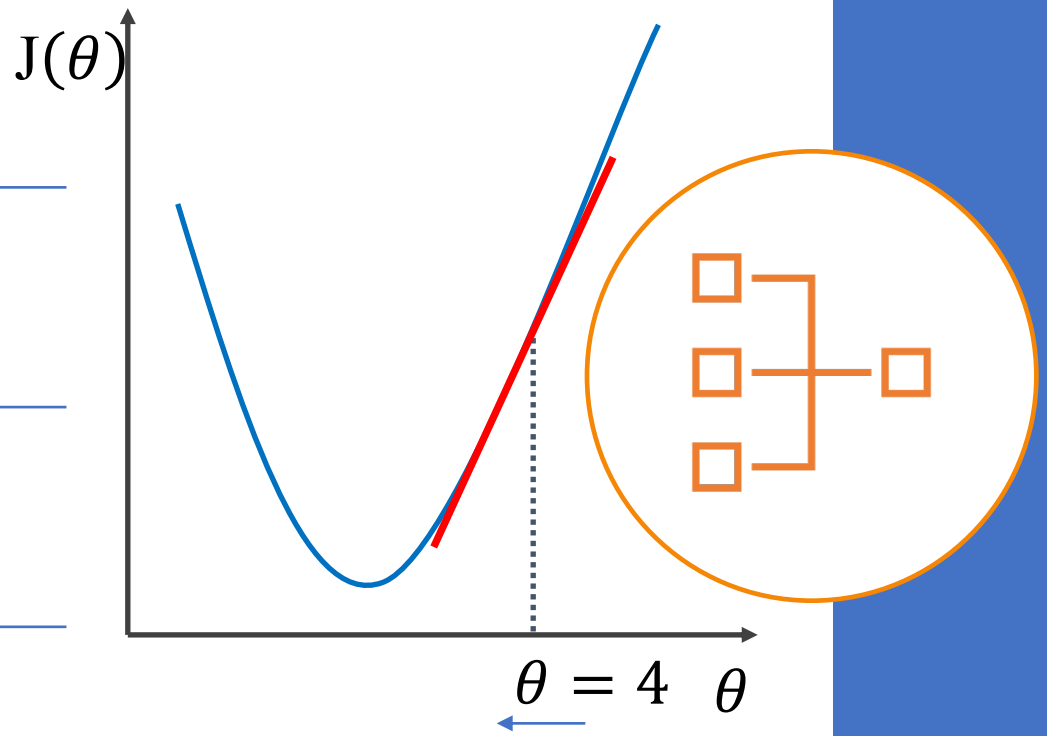
Convex Function

Gradient Descent (GD) (idea)

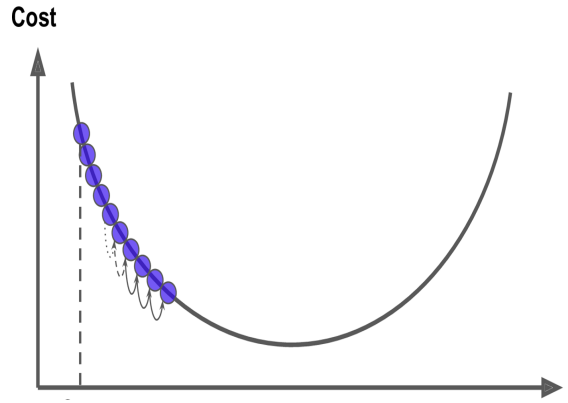
Start with a random value of θ (e.g. $\theta = 4$)

Compute the gradient (derivative) of $J(\theta)$ at point $\theta = 4$.

Recompute θ as: $\theta = \theta - \lambda * gradient$



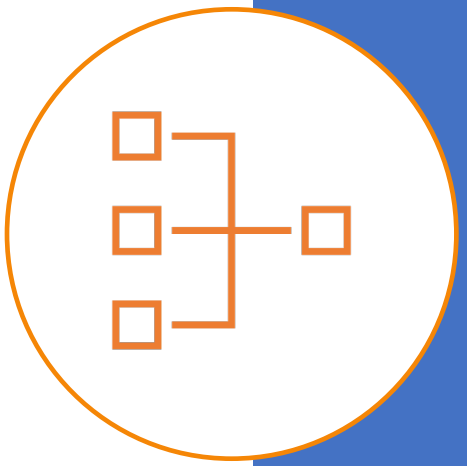
Gradient Descent



Repeat until convergence :$\{\theta_1 := \theta_1 - \alpha \frac{\partial J(\theta)}{\partial \theta_1}\}$

Intuition for Gradient Descent:

$\frac{\partial J(\theta)}{\partial \theta_1}$ gives change of loss with θ_1. For our example in previous slides $\frac{\partial J(\theta)}{\partial \theta_1}$



- is zero for $\theta_1 = 1$, cost is not changing with θ
- is positive for $\theta_1 > 1$, this means cost is increasing with increase in θ_1 , hence decrease θ_1
- is negative for $\theta_1 < 1$, this means cost is decreasing with increase in θ_1 , hence increase θ_1

Gradient Descent- Two Parameters

$\lambda = 0.01$

Initialize θ_0 and θ_1 randomly

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

for e = 0, num_epochs **do**

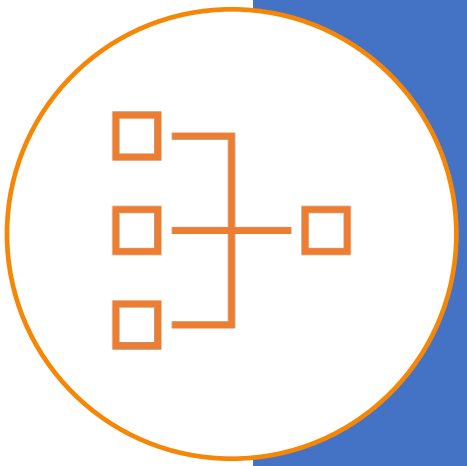
Compute: $dJ(\theta_0, \theta_1)/d\theta_0$ and $dJ(\theta_0, \theta_1)/d\theta_1$

Update θ_0: $\theta_o = \theta_0 - \lambda dJ(\theta_0, \theta_1)/d\theta_0$

Update θ_1: $\theta_1 = \theta_1 - \lambda dJ(\theta_0, \theta_1)/d\theta_1$

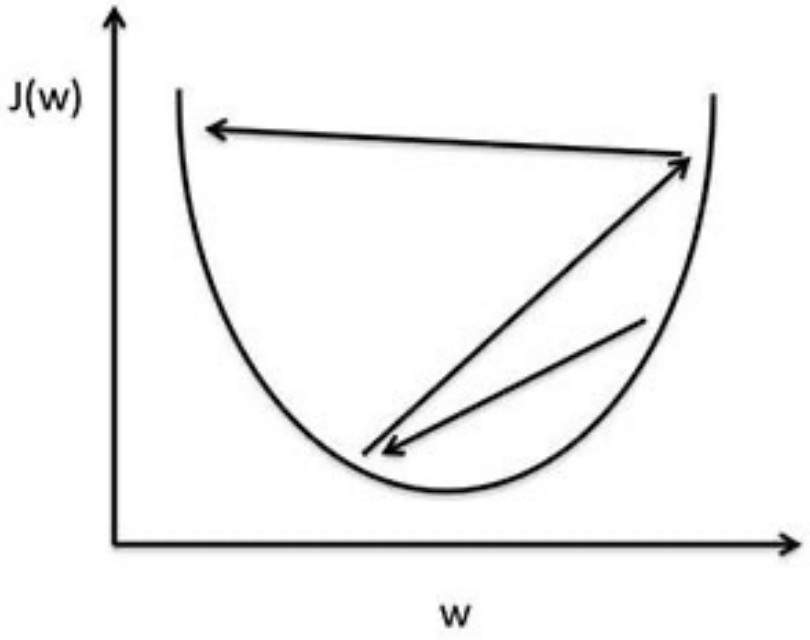
Print: $J(\theta_0, \theta_1)$ // Useful to see if this is becoming smaller or not.

end

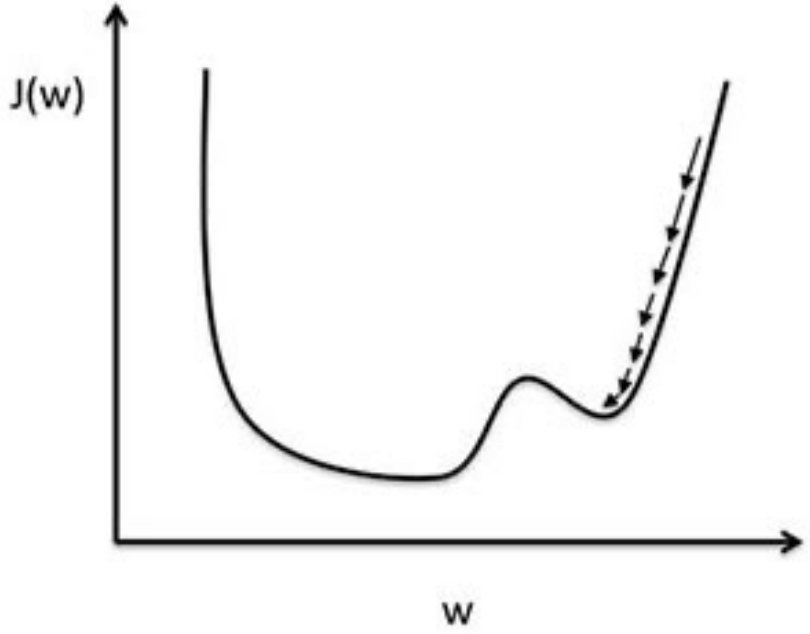


- An epoch is one complete pass through the entire training dataset, during which the model's parameters are updated to minimize the loss of function.
- Each Update is based on complete dataset

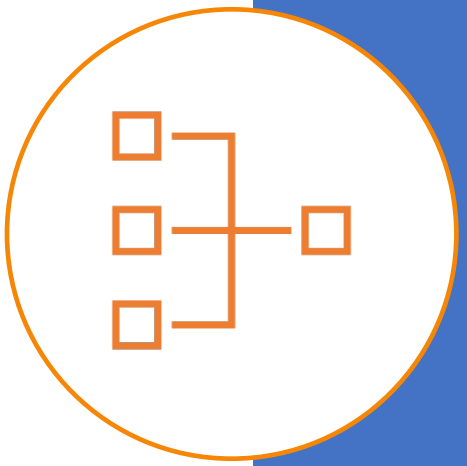
Effect of learning rate



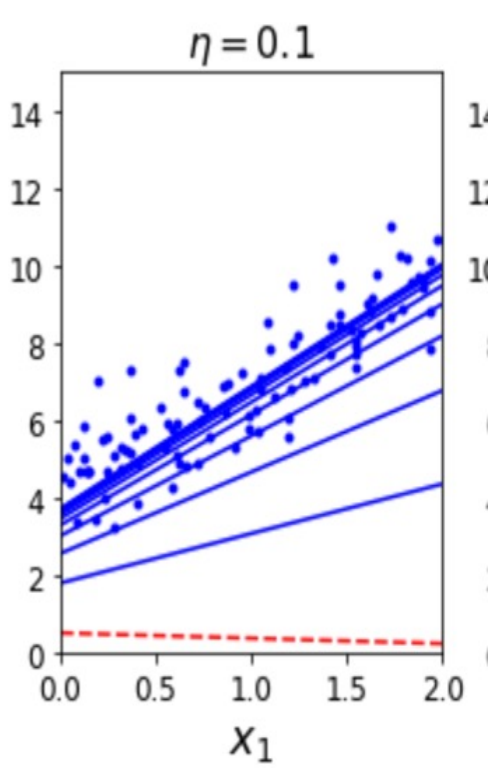
Large Learning Rate

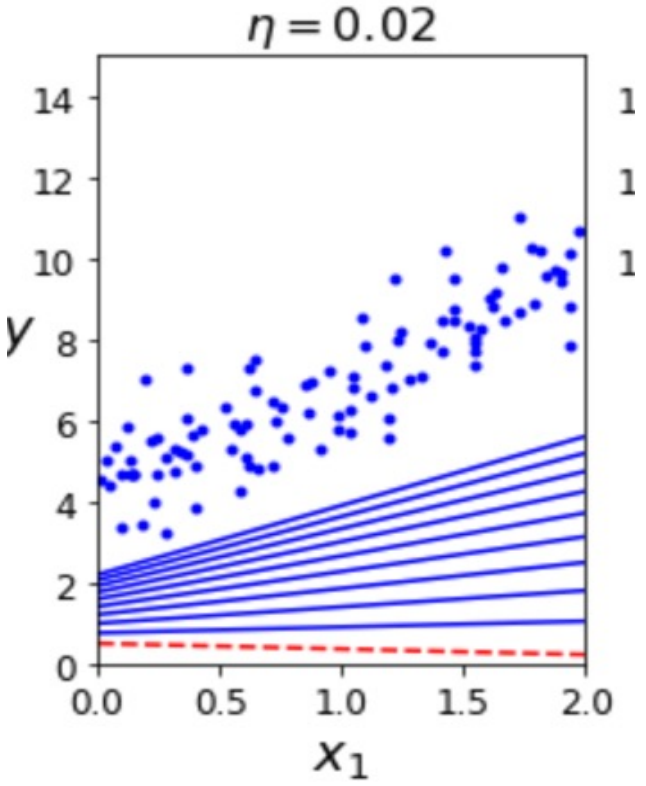


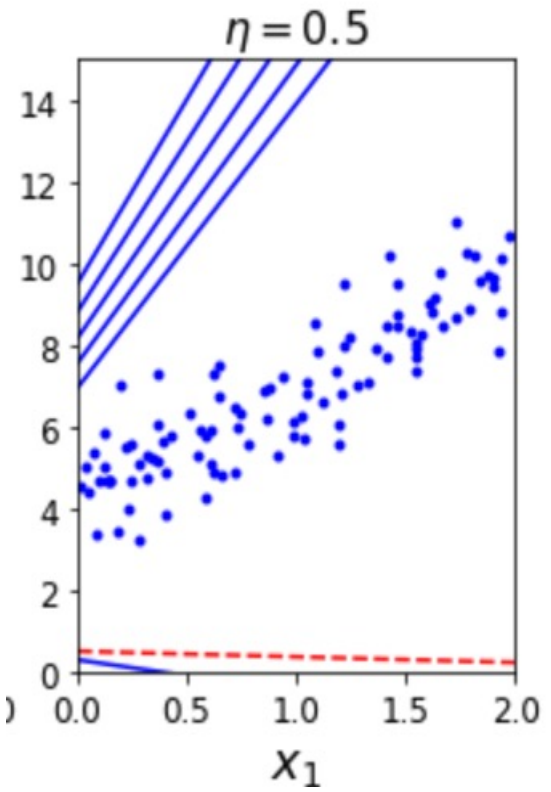
Small Learning Rate

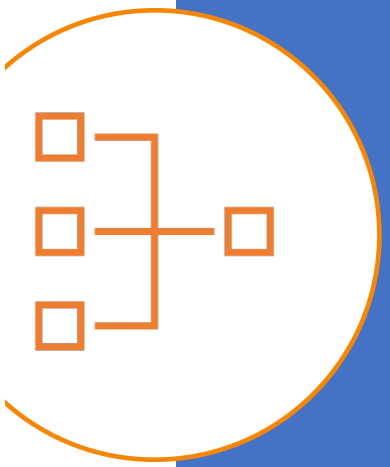


Effect of learning rate

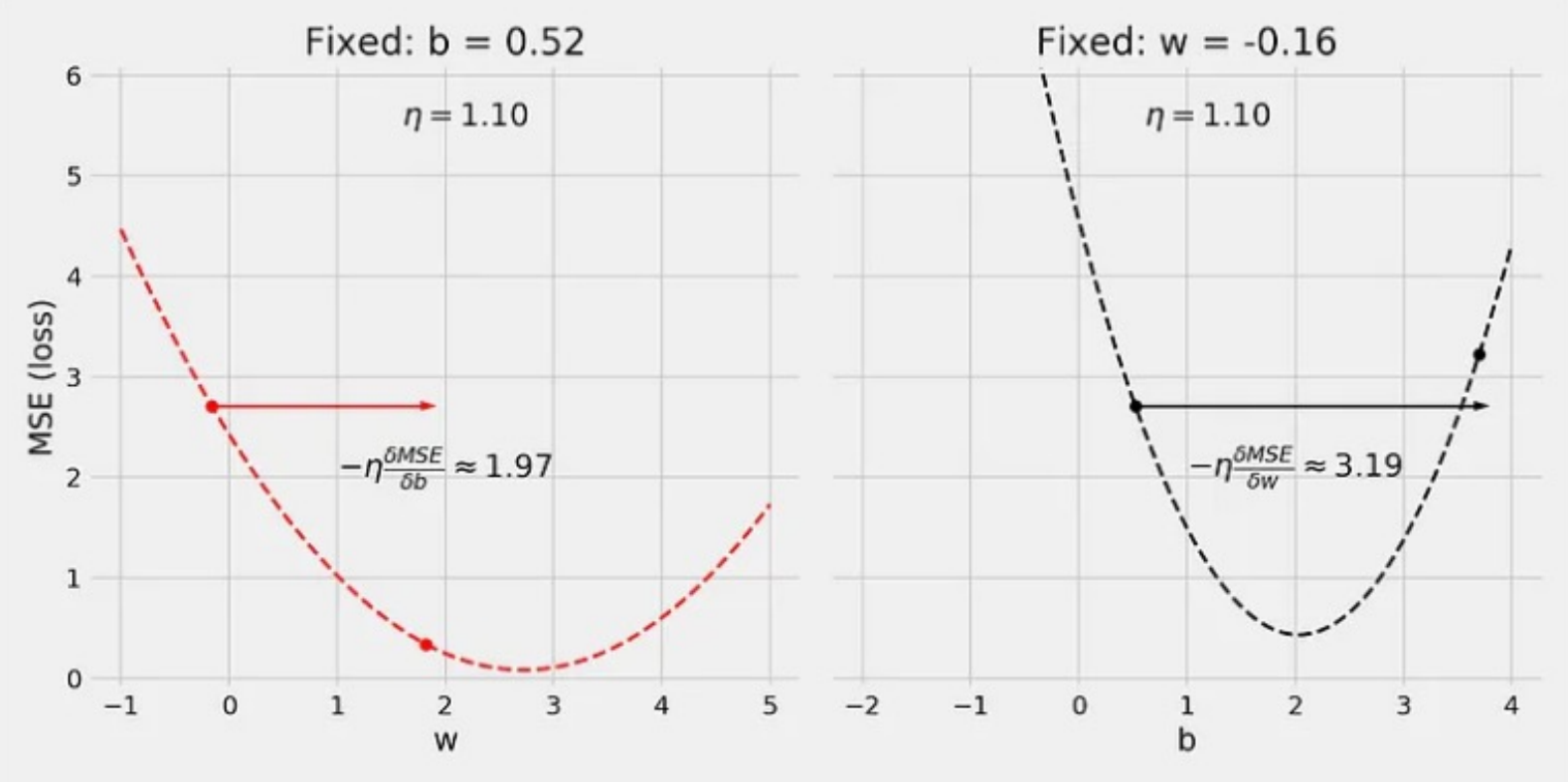


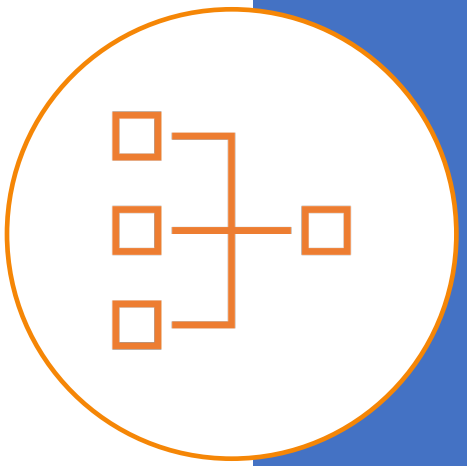






Effect of Scaling



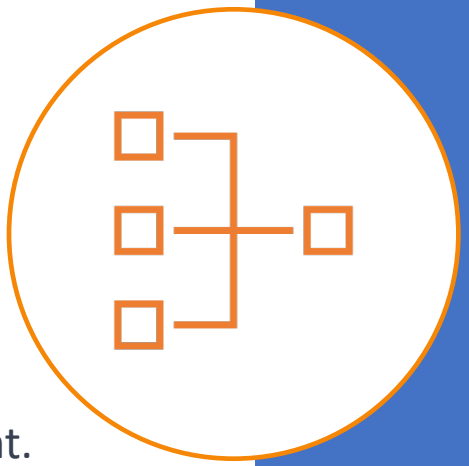


Source: https://towardsdatascience.com/gradient-descent-the-learning-rate-and-the-importance-of-feature-scaling-6c0b416596e1

Why We Need Scaling for Gradient Descent

- **Steps Proportional to Gradient:**
  - Gradient descent updates parameters by steps proportional to the negative gradient to minimize the loss.

- **Influence of Feature Scale:**
  - The scale of features influences gradient magnitudes and convergence rate.
  - Large-scale features can lead to large gradients, causing oscillations or unstable convergence.
  - Small-scale features may result in tiny gradients, significantly slowing convergence.



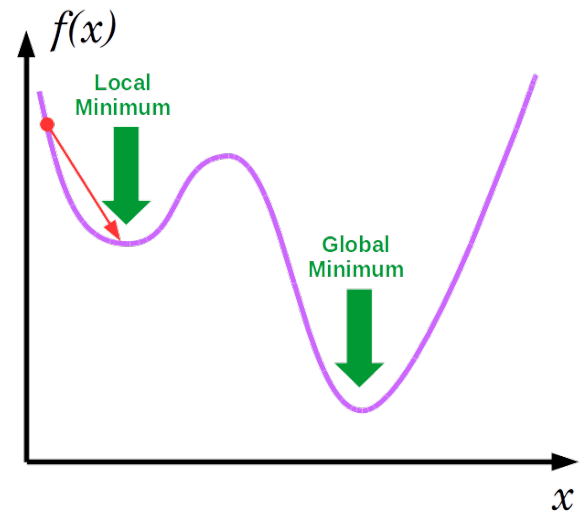
- **Problems of Disparate Ranges:**
  - Vastly differing feature scales can skew the optimization process, making it inefficient.

- **Solution via Scaling:**
  - Scaling features to a standardized range, such as [0, 1] or [-1, 1], for uniformity.
  - This equalizes the scale of the gradients, making them comparable across features.

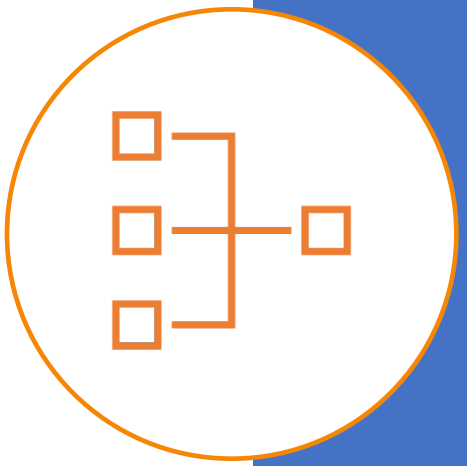
- **Benefits of Scaling:**
  - Ensures that gradient descent takes appropriately sized and balanced steps.
  - Facilitates more efficient and stable minimization of the loss function

Pros and Cons

- Advantages:
  - Simple and often quite effective on ML tasks
  - Often very scalable



- Drawbacks
  - Might find a local minimum
  - Only applies to smooth function (differentiable)



Fortunately, the MSE cost function for a Linear Regression model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve.

Gradient Descent- Two Parameters

expensive

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$\lambda = 0.01$

Initialize θ_0 and θ_1 randomly

for e = 0, num_epochs **do**

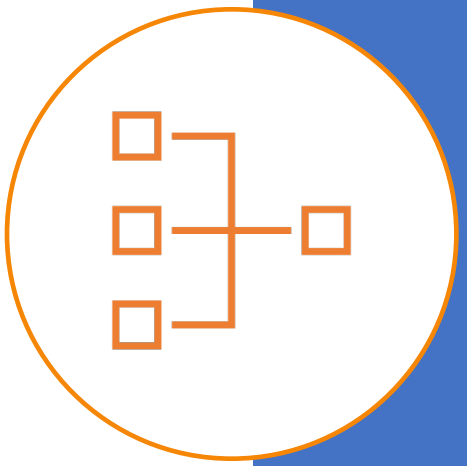
Compute: $dJ(\theta_0, \theta_1)/d\theta_0$ and $dJ(\theta_0, \theta_1)/d\theta_1$

Update θ_0: $\theta_o = \theta_0 - \lambda dJ(\theta_0, \theta_1)/d\theta_0$

Update θ_1: $\theta_1 = \theta_1 - \lambda dJ(\theta_0, \theta_1)/d\theta_1$

Print: $J(\theta_0, \theta_1)$

end



- An epoch is one complete pass through the entire training dataset, during which the model's parameters are updated to minimize the loss of function.
- Each Update is based on complete dataset

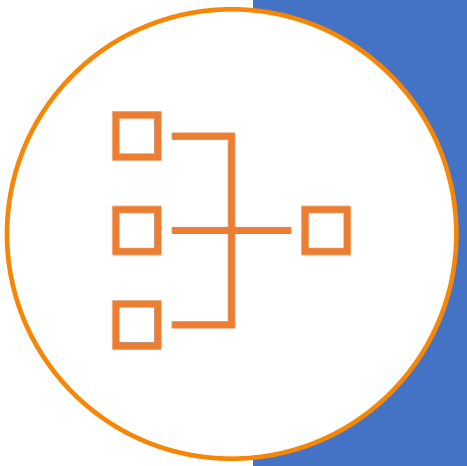
Gradient Descent- Mini Batch

$\lambda = 0.01$

Initialize θ_0 and θ_1 randomly

for e = 0, num_epochs **do**

$$j(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i \epsilon B}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$



for b = 0, num_batches **do**

Compute: $dj(\theta_0, \theta_1)/d\theta_0$ and $dj(\theta_0, \theta_1)/d\theta_1$

Update θ_0: $\theta_o = \theta_0 - \lambda dj(\theta_0, \theta_1)/d\theta_0$

Update θ_1: $\theta_1 = \theta_1 - \lambda dj(\theta_0, \theta_1)/d\theta_1$

Print: $j(\theta_0, \theta_1)$ // Useful to see if this is becoming smaller or not.

end

- An epoch is one complete pass through the entire training dataset, during which the model's parameters are updated to minimize the loss of function.
- Each Update is based on a batch of observations

Gradient Descent Variations

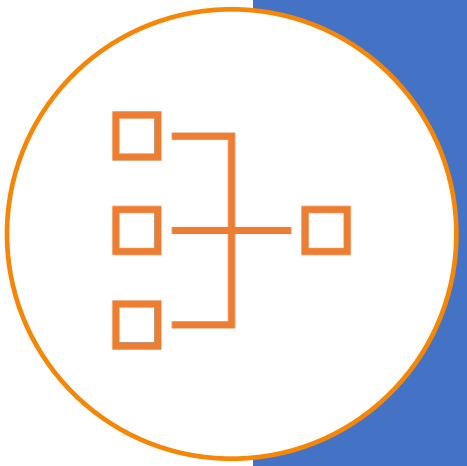
X1	X2	Y	Yhat	Error (e)	E * x1	E * x2

- Batch Gradient Descent

- 1 update per epoch
- All 12 observations will be used in each update of parameters

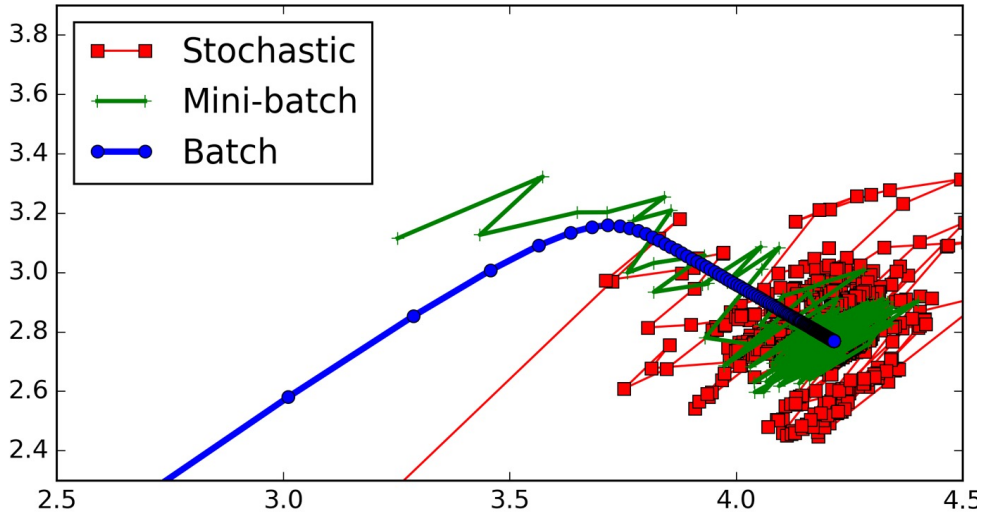
- Mini Batch Gradient Descent

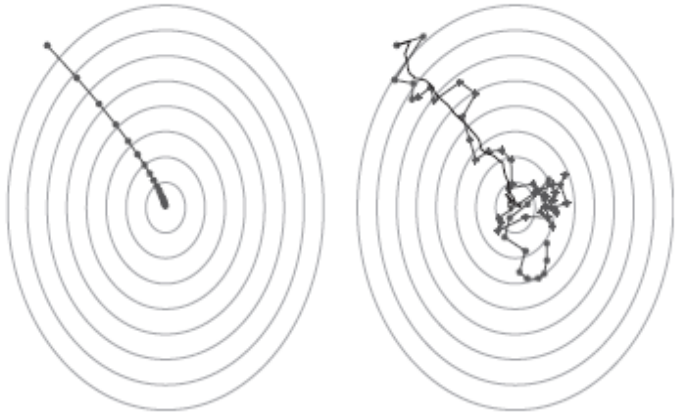
- Let us say Batch size is 3
- Gradient is calculated based on 3 observations
- In an epoch, we will make 4 updates



Gradient Descent Variations

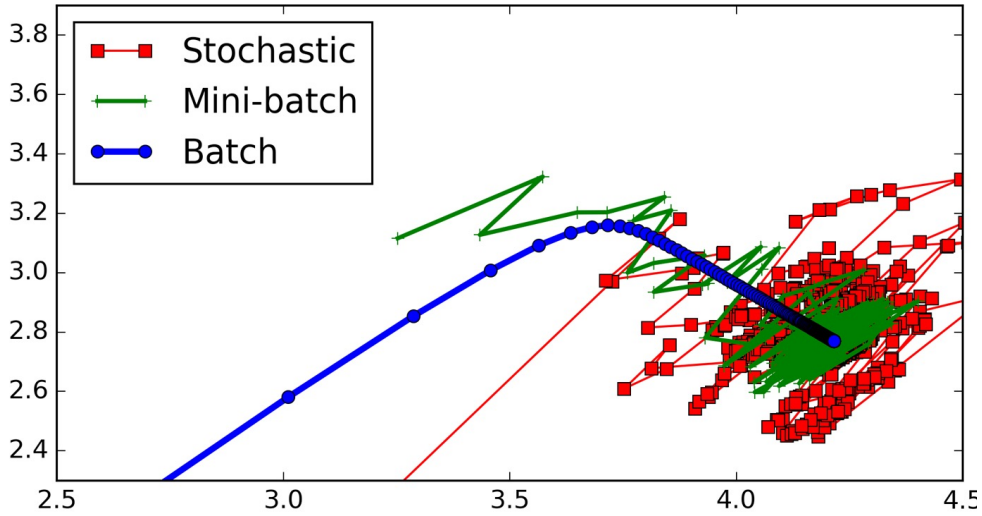
- Batch Gradient Descent : Uses whole training data at every iteration
- Stochastic Gradient Descent: Each step uses a random instance in the training set
- Mini Batch Gradient Descent : Each step uses a subset of instance in the training set

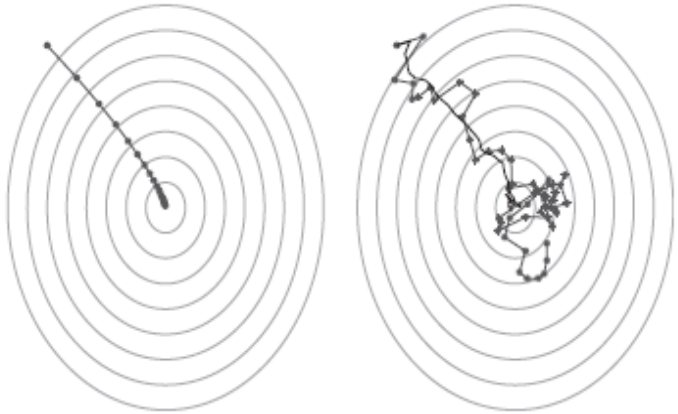




Scaling (Normalization for Gradient Descent)

- Batch Gradient Descent : Uses whole training data at every iteration
- Stochastic Gradient Descent: Each step uses a random instance in the training set
- Mini Batch Gradient Descent : Each step uses a subset of instance in the training set





Gradient Descent

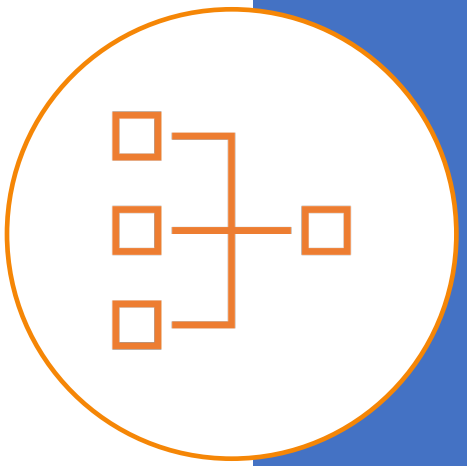
Goal: Adjust weights (model parameters) to improve model

Direction : Opposite to sign of gradient

Magnitude : Proportional to gradient

Learning rate : Use the learning rate to control the magnitude

Learning rate is hyperparameter, $\theta's$ are model parameters



A Recipe for Machine Learning

1. Given training data:

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

2. Choose each of these:

- Decision function

$$\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

- Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

4. Train with SGD:

(take small steps opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

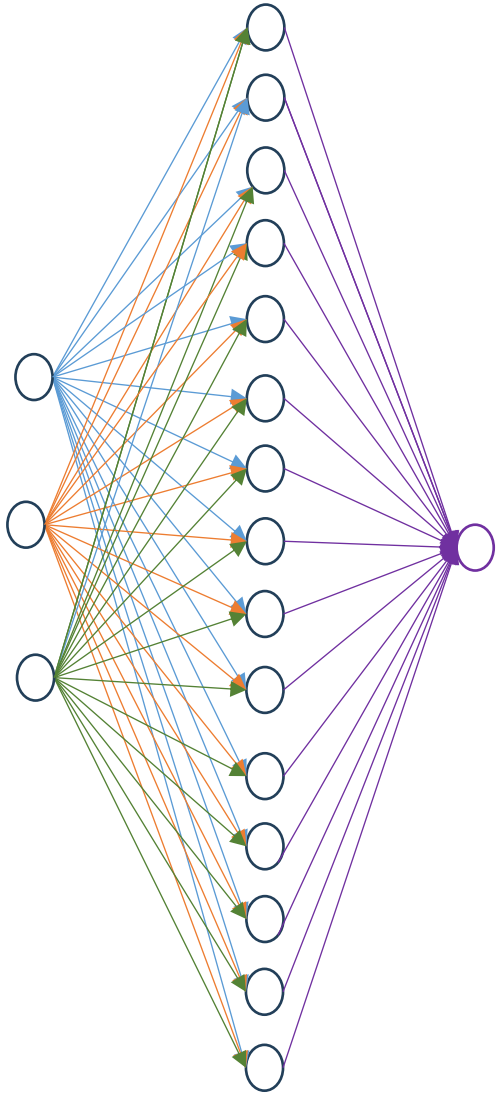
Training Neural Network

Training Loops

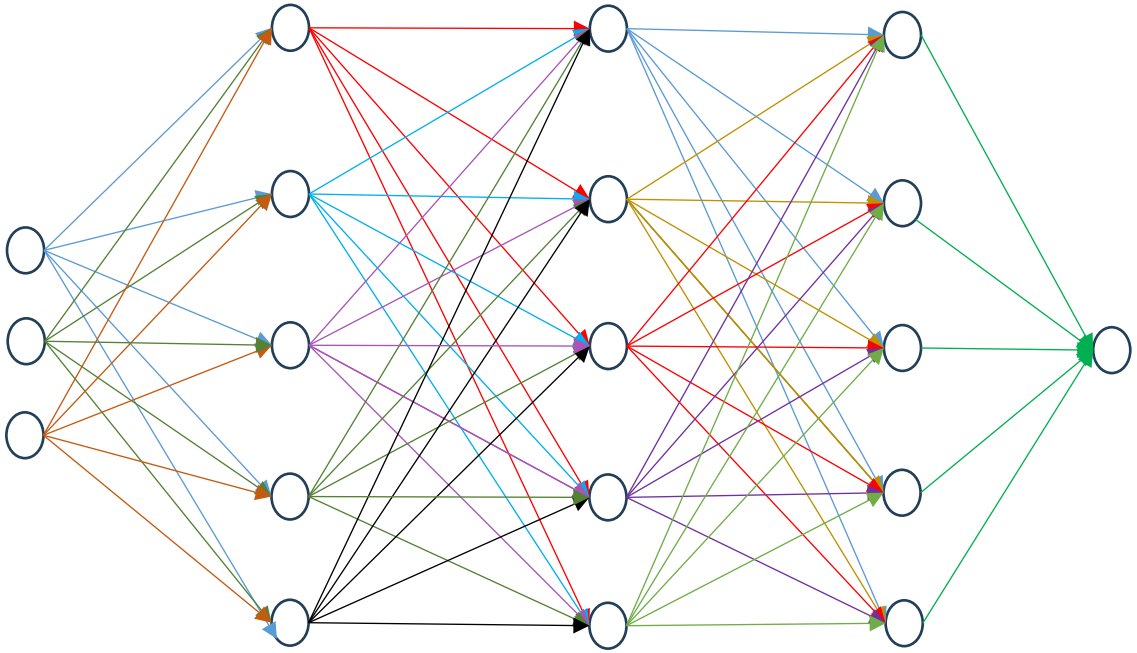
```
Initialize model parameters with small random values
for each training epoch: # Repeat the process for each epoch
    for each batch in the training dataset: # Iterate over each batch of data
        # Step 1: Calculate Predictions
        - Feed the batch of input data through the model to obtain predictions.

        # Step 2: Calculate Loss
        - Compare the predictions to the actual target values to calculate the loss.

        # Step 3: Perform Backpropagation
        - Calculate the gradients of the loss function with respect to the model's parameters

        # Step 4: Update Model Parameters
        - Adjust the model parameters in the opposite direction of the gradients .

    # After training on all batches, evaluate the model performance on the validation dataset:
    for each batch in the validation dataset:
        # Step 5: Calculate Validation Predictions
        - Use the model to predict the validation data without updating the model parameters.

        # Step 6: Calculate Validation Loss
        - Calculate the loss on the validation dataset to monitor model performance.

    # Step 7: Print Progress
    - Output the progress, showing the epoch number and the validation loss.
```

Deep Vs. Wide

Wide Vs Deep Networks



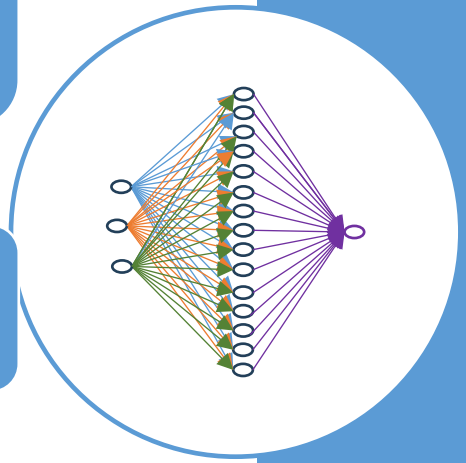
Wide and Shallow



Narrow and Deep

Wide and Shallow

Given enough data, a shallow network with one hidden layer can approximate any arbitrary function.



Disadvantages:

- Needs a lot of neurons in the hidden layer
- Is prone to overfitting (memorization vs. learning

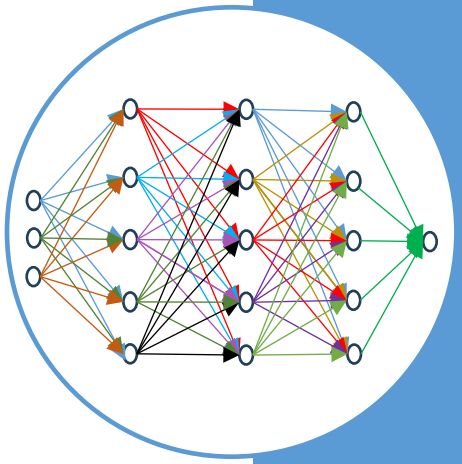
Narrow and Deep

Advantages:

- Needs fewer Units
- Generalizes better

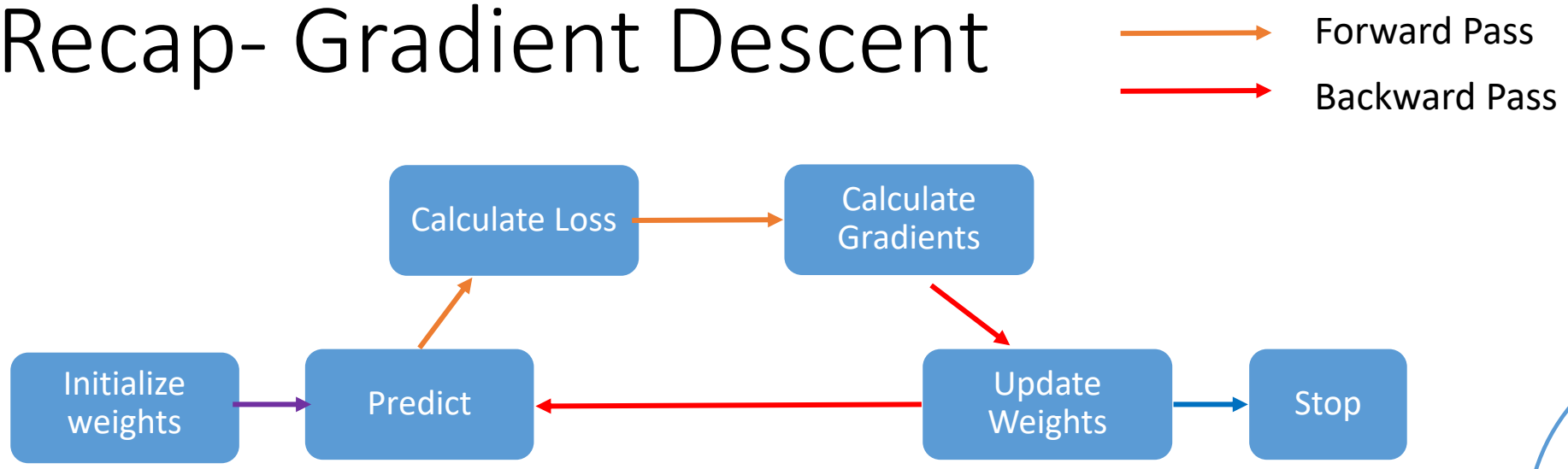
Disadvantages

- Harder to train



Numerical Stability

Recap- Gradient Descent

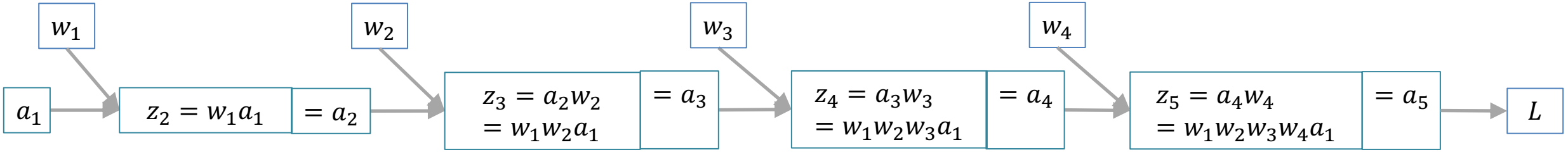


Goal: Optimize model performance by finding suitable weights.

- Define a loss function - a lower value indicates better model performance.
- Calculate gradients - these represent the change in loss with respect to weights. Positive gradients imply a decrease in weight to minimize loss, while negative gradients indicate an increase in weight.
- Utilize gradients to guide weight adjustments, leading to continuous improvement in model performance.

Forward Pass

Five layers, one neuron in each layer, no bias, and linear activation function



$$a_5 = w_1 w_2 w_3 w_4 a_1$$

$$a_l = \mathrm{a}_1 \times \left(\prod_{l=1}^{L-1} w_l \right)$$

Exploding Activations

$$a_L = \mathrm{a}_1 \times \left(\prod_{l=1}^{L-1} w_l \right)$$

$$\mathrm{a}_1 = 1, L = 101$$

$$w_1 = w_2 \cdots = w_{L-1} = w = 0.5$$

$$a_2 = (w)^1 = 1.5$$

$$a_6 = (w)^5 = 7.6$$

$$a_{101} = (w)^{100} = (1.5)^{100} = 4.06 \times (10)^{17}$$

Vanishing Activations

$$a_L = \mathrm{a}_1 \times \left(\prod_{l=1}^{L-1} w_l \right)$$

$\mathrm{a}_1 = 1, L = 101$

$w_1 = w_2 \cdots = w_{L-1} = w = 0.5$

$a_2 = (w)^1 = 0.5$

$a_6 = (w)^5 = 0.03125$

$a_{101} = (w)^{100} = (0.5)^{100} = 8 \times (10)^{-31} \approx 0$

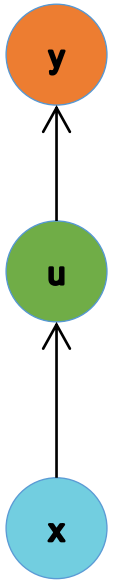
Backward Pass

Computation graph – Chain Rule

Given:
$y = g(u)\ and\ u = h(x)$

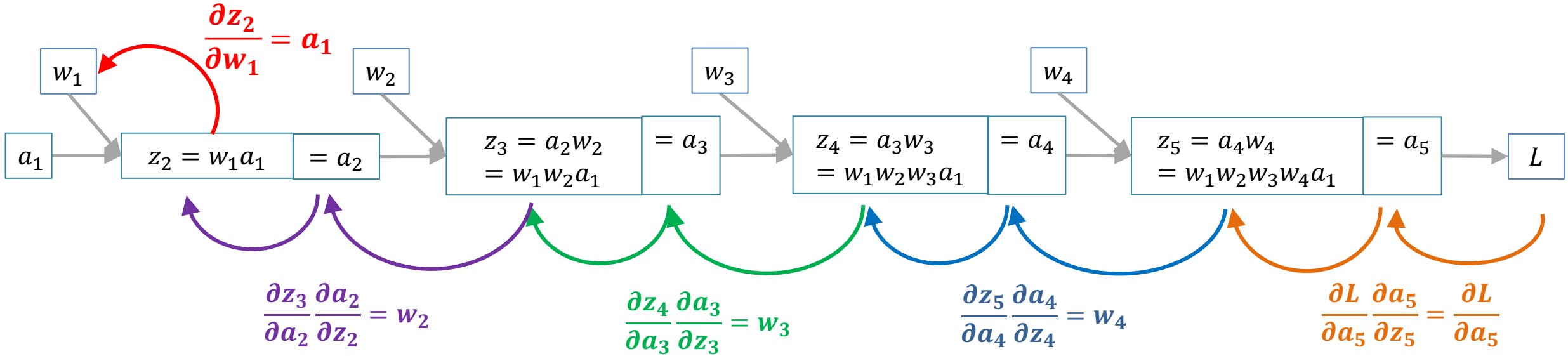
Chain Rule :

$$\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx}$$



Backpropagation is just repeated application of the **chain rule**

Five layers, one neuron in each layer, no bias, and linear activation function



$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_5} w_4 w_3 w_2 a_1$$

Exploding Gradients

$$\delta_l = \delta_L \times \left(\prod_{l=L-1}^{l} w_l \right)$$

$\delta_{101} = 1, L = 101$

$w_1 = w_2 \cdots = w_{L-1} = w = 1.5$

$\delta_{100} = \delta_{101}(w)^1 = 1.5$

$a_{96} = \delta_{101}(w)^5 = 7.6$

$\delta_1 = (w)^{100} = (1.5)^{100} = 4.06 \times (10)^{17}$

Vanishing Gradients

$$\delta_l = \delta_L \times \left(\prod_{l=L-1}^{l} w_l \right)$$

$$\delta_{101} = 1, L = 101$$

$$w_1 = w_2 \cdots = w_{L-1} = w = 0.5$$

$$\delta_{100} = \delta_{101}(w)^1 = 0.5$$

$$a_{96} = \delta_{101}(w)^5 = 0.03125$$

$$\delta_1 = (w)^{100} = (1.5)^{100} = 8 \times (10)^{-31} \approx 0$$

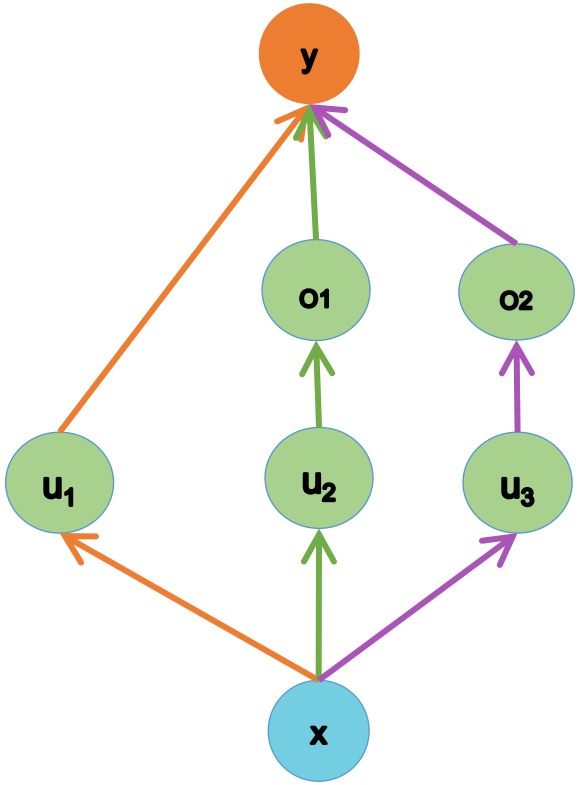
Computation graph – Backward pass

$$Path1 = \frac{\partial y}{\partial u_1} * \frac{\partial u_1}{\partial x}$$

$$Path2 = \frac{\partial y}{\partial o_1} * \frac{\partial o_1}{\partial u_2} * \frac{\partial u_2}{\partial x}$$

$$Path3 = \frac{\partial y}{\partial o_2} * \frac{\partial o_2}{\partial u_3} * \frac{\partial u_3}{\partial x}$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u_1} * \frac{\partial u_1}{\partial x} + \frac{\partial y}{\partial o_1} * \frac{\partial o_1}{\partial u_2} * \frac{\partial u_2}{\partial x} + \frac{\partial y}{\partial o_2} * \frac{\partial o_2}{\partial u_3} * \frac{\partial u_3}{\partial x}$$



Backpropagation is just the repeated application of the **chain rule**

Issues with Gradient Exploding

VALUE OUT OF RANGE
INFINITY VALUE

SENSITIVE TO
LEARNING RATE (LR)

Issues with Gradient Vanishing

No progress in training	Irrespective of learning rate
Severe with bottom layers	Only top layers are well trained

Stabilize Training

Weight Initialization

Weight initialization

Constant Initialization: Set all weights to C

- Every neuron in the network computes the same output → computes the same gradient → same parameter updates

Normal Initialization: Set all weights to random small numbers

- Every neuron in the network computes different output → computes different gradient → different parameter updates
- "Symmetry breaking“
- Problem: variance that grows with the number of inputs.
- Initializing according to $\mathcal{N}(0,0.01)$ works well for small networks, but not guaranteed to work for deep neural networks

Initialization for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Normalization

Batch Normalization

1. $$\boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

2. $$\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

3. $$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$$

4. $$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

$$\widehat{\mathbf{v}} \leftarrow \widehat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

Batch Normalization- Benefits

Faster model convergence

- Use higher learning Rate
- Use large learning rate decay

- Mitigates the problems with non-zero centered activation functions

- Improves stability and Quality

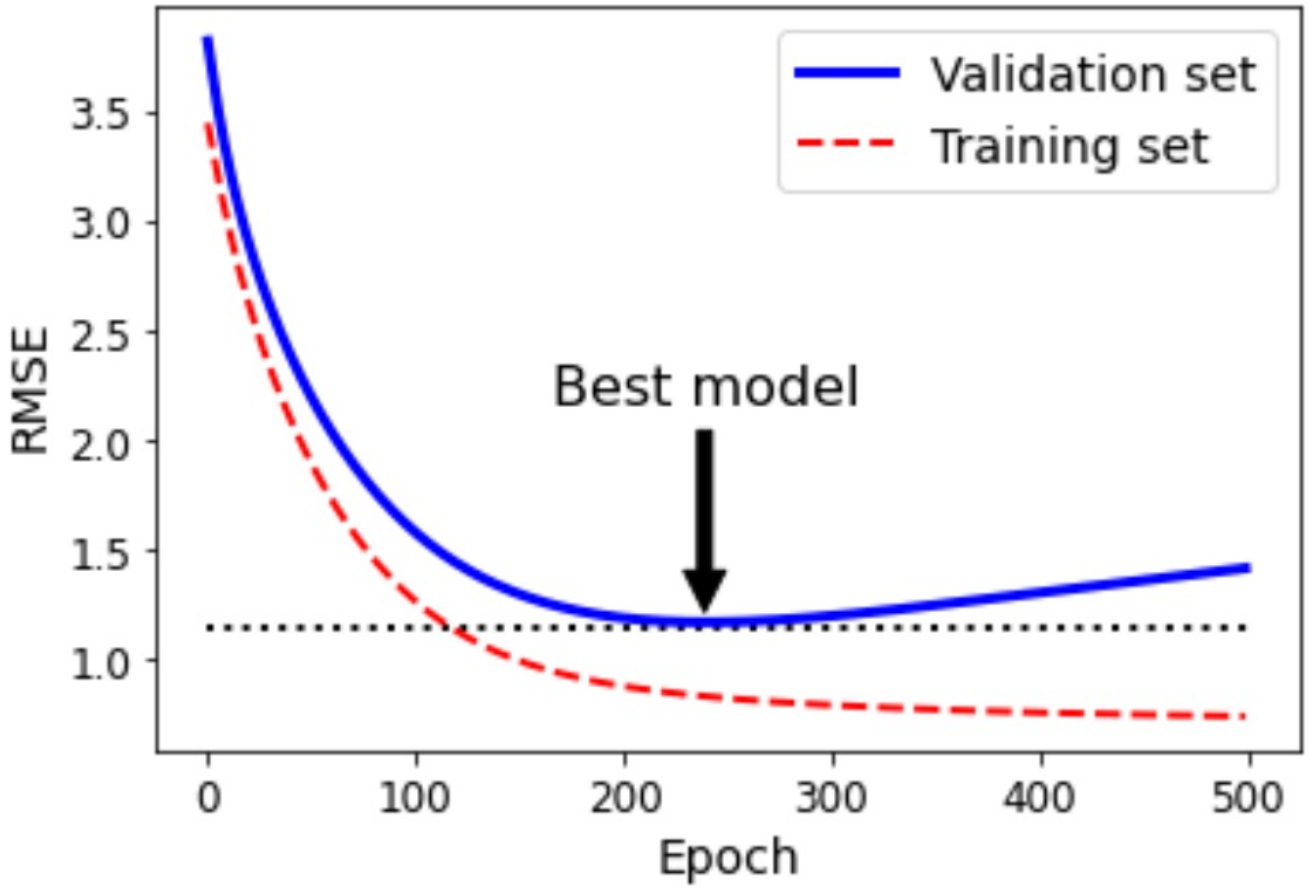
- Better regularization (sometimes you do not need to use dropout)

Gradient Clipping

- Gradient Clipping -Clip the gradients during backpropagation so that they never exceed some threshold.
- This technique is most often used in recurrent neural networks, as Batch Normalization is tricky to use in RNNs,
- For other types of networks, BN is usually sufficient.

Regularization

Early Stopping



L2-Norm

$$\min \ \ell(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

- Hyper-parameter λ controls regularization importance
- $\lambda = 0$: no effect
- $\lambda \to \infty, \mathbf{w}^* \to \mathbf{0}$

L2-Norm

Update Rule

- Compute the gradient

$$\frac{\partial}{\partial \mathbf{w}}\left(\ell(\mathbf{w}, b)+\frac{\lambda}{2}\|\mathbf{w}\|^2\right)=\frac{\partial \ell(\mathbf{w}, b)}{\partial \mathbf{w}}+\lambda \mathbf{w}$$

- Update weight at time t

$$\mathbf{w}_{t+1}=(1-\eta\lambda)\mathbf{w}_t-\eta\frac{\partial \ell(\mathbf{w}_t, b_t)}{\partial \mathbf{w}_t}$$

  - Often $\eta\lambda < 1$, so also called weight decay in deep learning

Dropout - Motivation

- A good model should be robust under modest changes in the input
- Training with input noise equals to Tikhonov Regularization
- Dropout: inject noises into internal layers

Dropout - Motivation

Add Noise without Bias

- Add noise into ***x*** to get ***x'***, we hope

$$\mathbf{E}[\mathbf{x}'] = \mathbf{x}$$

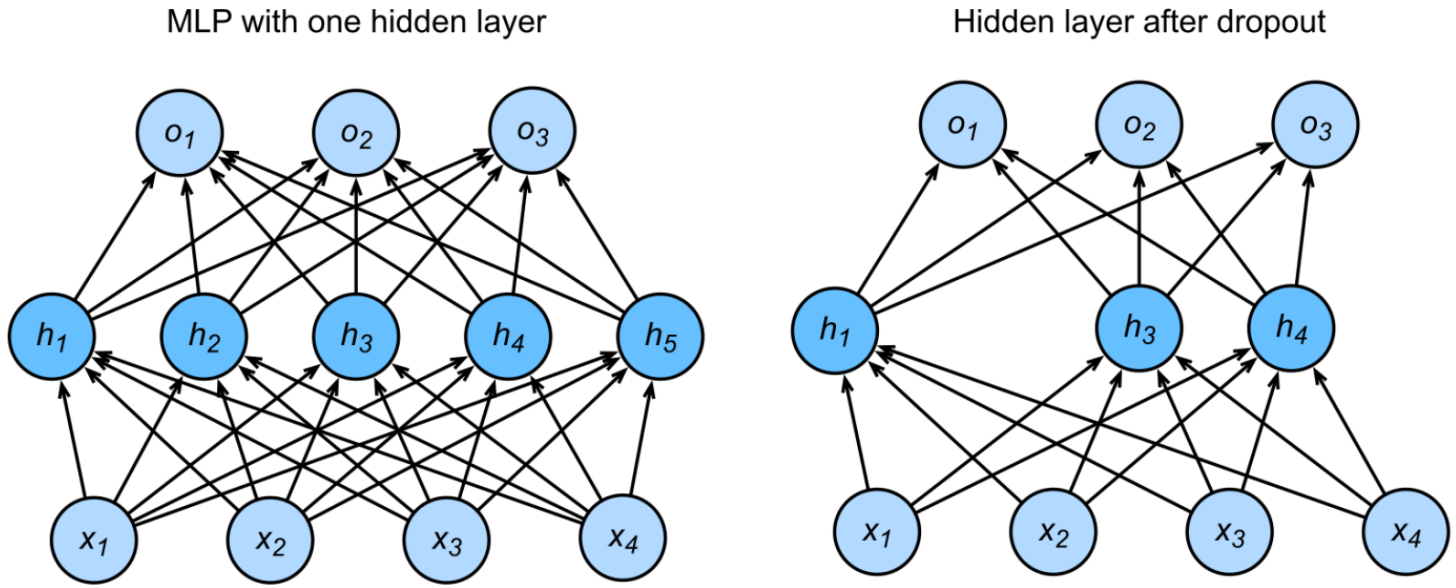
- Dropout perturbs each element by

$$x_i' = \begin{cases} 0 & \text{with probablity } p \\ \frac{x_i}{1-p} & \text{otherise} \end{cases}$$

Dropout

- Often apply dropout on the output of hidden fully-connected layers

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h}' + \mathbf{b}_2$$
$$\mathbf{y} = \text{softmax}(o)$$



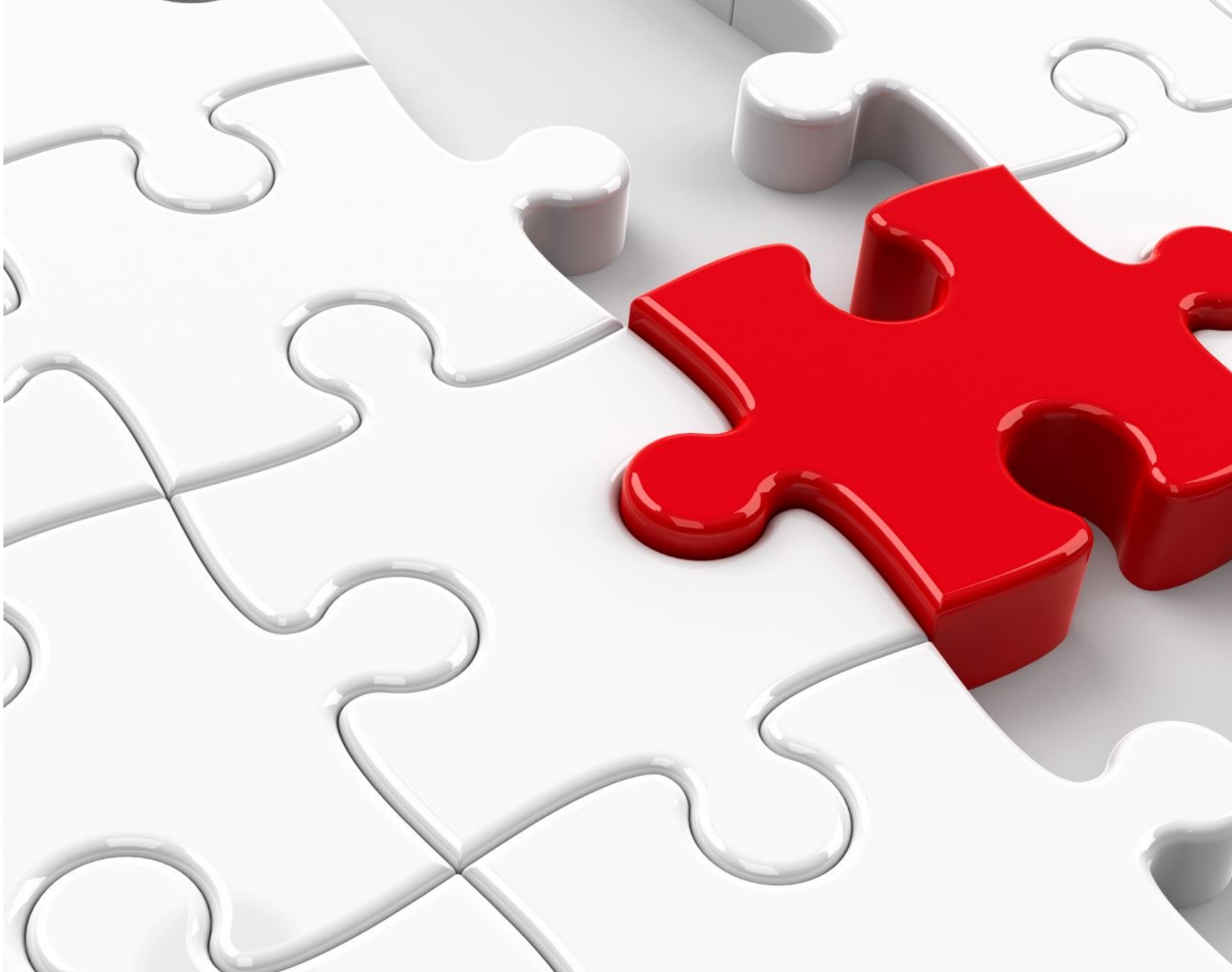
Dropout

- Regularization is only used in training
- The dropout layer for inference is

$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$

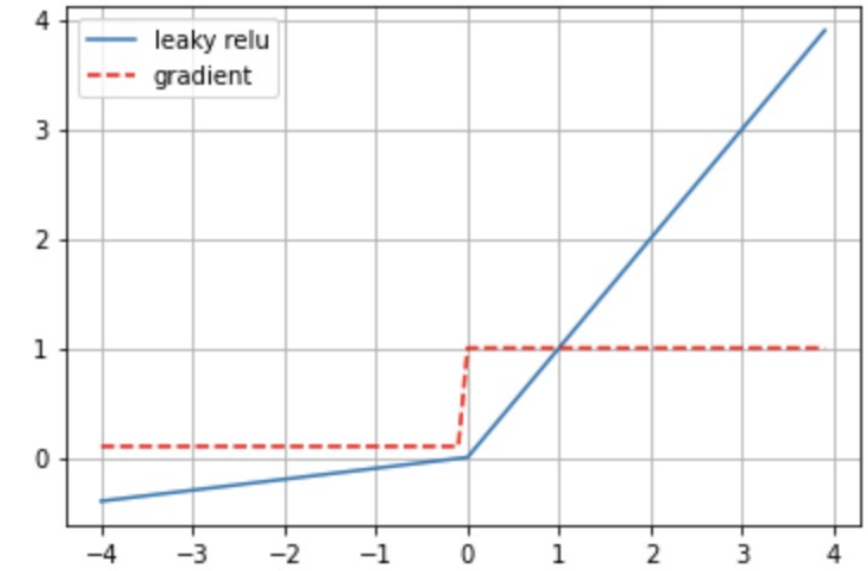
  - Guarantee deterministic results

Activation Functions



Leaky ReLU

Leaky ReLUα(z) = $\max(\alpha z, z)$, α typically set to 0.01

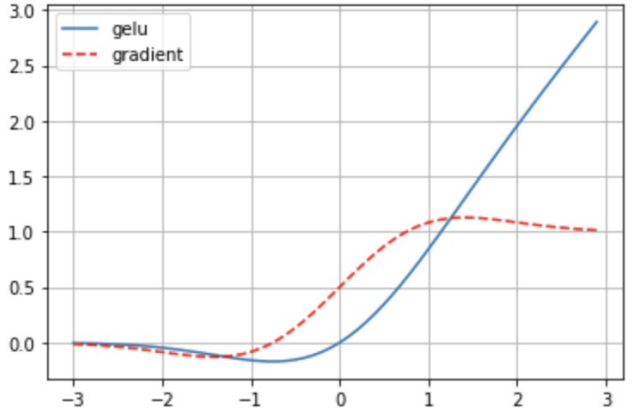


Randomized leaky ReLU (RReLU)- α is picked randomly in a given range during training and is fixed to an average value during testing

Parametric leaky ReLU (PReLU)
α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter).

Gaussian Error Linear Unit (GELU)

$$GELU(x) = 0.5z\left(1 + \tanh\left(\sqrt{2/\pi}\,(x + 0.044715\,x^3)\right)\right)$$



- Combines stochastic regularization like dropout, zoneout with deterministic non-linearity like ReLU
- Multiply the input by zero or one, but the values of this zero-one mask are stochastically determined
- Specifically multiply neuron input x by $m \sim Bernauli(\Phi(x))$, where $\Phi(x) = \mathrm{P}(\mathrm{X} \leq \mathrm{x}), \mathrm{X} \sim \mathcal{N}(0,1)$
- Take the expected value of transformation to get deterministic non-linearity:

$$GELU(x) = E(xm) = xE(m) = x\Phi(x) \approx 0.5z\left(1 + \tanh\left(\sqrt{2/\pi}\,(x + 0.044715\,x^3)\right)\right)$$

- **GELUs are used in GPT-3, BERT, and most other Transformers**

Faster Optimizers

Variations of Gradient Descent. There are 3 main ways how they differ:

- **Adapt the "gradient component"** $\left(\frac{\partial l}{\partial w}\right)$
  Use *aggregate of multiple gradients*. For example, use the exponential moving average of gradients.

- **Adapt the "learning rate component"**
  Adapt the learning rate according to the *magnitude* of the gradient(s).

- **Both (1) and (2)**
  Adapt both the gradient component and the learning rate component.

Faster Optimizers

Class	Year	Learning Rate	Gradient
Momentum	1964		
AdaGrad	2011		
RMSprop	2012		
Adadelta	2012		
Nestrov	2013		
Adam	2014		
Nadam	2015		
AdaMax	2015		
AMSGrad	2018		

Comparison

Class	Convergence Speed	Convergence Quality
SGD	*	***
SGD(momentum=…)	**	***
SGD(momentum=…, nesterov=True)	**	***
Adagrad	***	*(stops too early)
RMSprop	***	**or***
Adam	***	**or***
Nadam	***	**or***
AdaMax	***	**or***

(* is bad, ** is average, and *** is good).

Learning Rate Schedule and Optimal Learning Rate

- Use SGD with Learning Rate schedule (One cycle LR is state of the art at the moment).

- Optimal Learning Rate

References

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition by Aurélien Géron

Dive into Deep Learning :https://d2l.ai/
by Aston Zhang and Zachary C. Lipton and Mu Li and Alexander J. Smola