

# Artificial Neural Network and Deep Learning

**MET CS Generative AI**

Reza Rawassizadeh

# Outline

- The Rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- Activation Functions
- Cost Function and Neural Network Optimizers
- Backpropagation
- Regularization in Neural Network

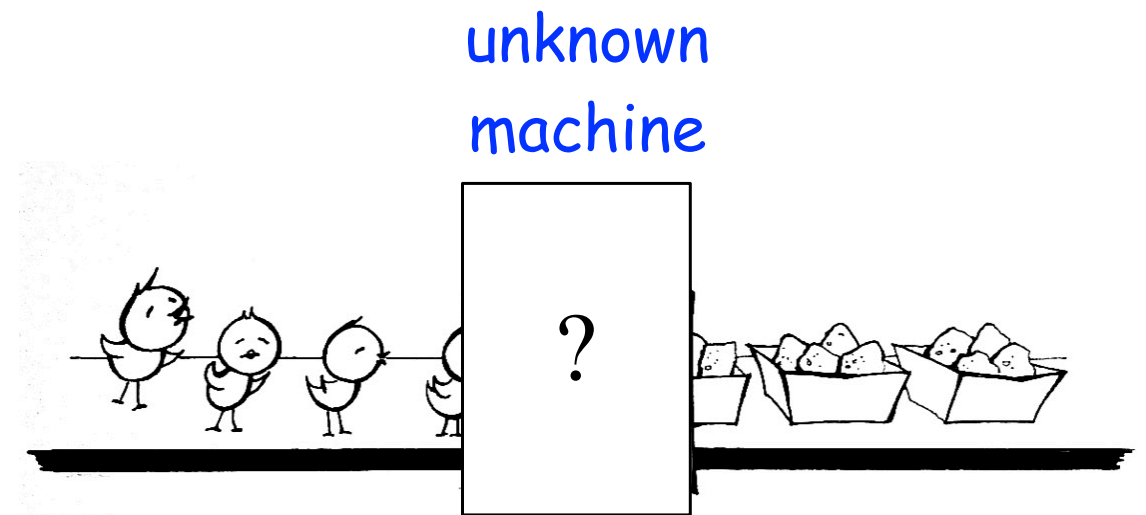
# Outline

- **The Rationale behind Deep Learning**
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- Activation Functions
- Cost Function and Neural Network Optimizers
- Backpropagation
- Regularization in Neural Network

# Why Deep learning is both popular and powerful?

- Deep Learning has revolutionized many scientific disciplines. People whose research works sound funny and too theoretical before 2012 are now celebrities in academia, not just in computer science in other disciplines.
- There are three reasons that deep learning is very popular:
- The first reason is its superior accuracy.
- The second reason that makes deep learning saliently superior to other machine learning algorithms, there is no need for feature engineering in these algorithms.
- The third reason is their capability to work with unstructured data. Other algorithms that we have described all require to have structured data. From tabular formats like CSV to a time series and signals. On the other hand, deep learning algorithms are working very well for unstructured data such as image, audio, and raw text data.

- Supervised machine learning can be interpreted as a *function approximation problem*. We have some data, and there is an underlying function that is not known, and we try to understand that function.
- We have a machine that receives chicken as input and produces nuggets as output. The process of supervised machine learning is to identify what is this unknown machine (function).
- This technique is also called **function approximation**. The name of this unknown function is often called the **target function**. We do not know this function, and therefore, we use machine learning to approximate this function.



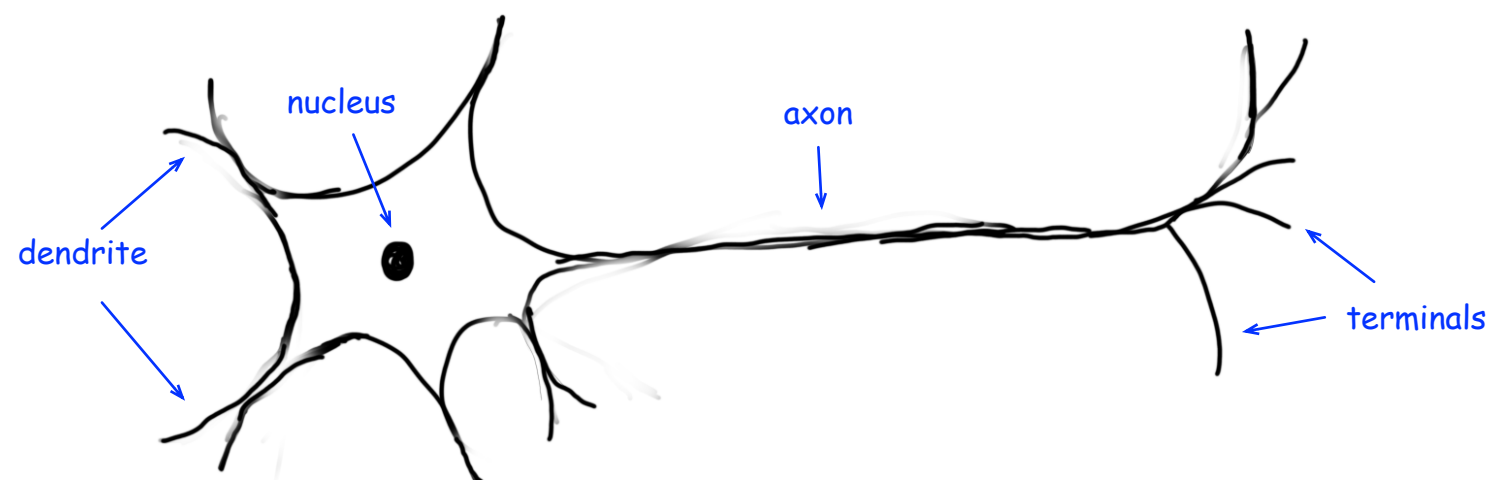
**Deep learning methods are non-linear function approximation methods.**

# Universal Approximation Theory

- *Universal Approximation Theory* is a mathematical theory that indicates that any continuous function could be approximated by a neural network.
- In simple words, given enough training data, there is a neural network that can model anything, and thus, neural networks have a kind of universality.
- Nevertheless, the network might get so large that no computer can handle it. *Theoretically, everything in nature could be modeled mathematically as a combination of linear and non-linear functions.* Therefore, it is not wrong to say that a neural network can model everything but requires a significant amount of data.

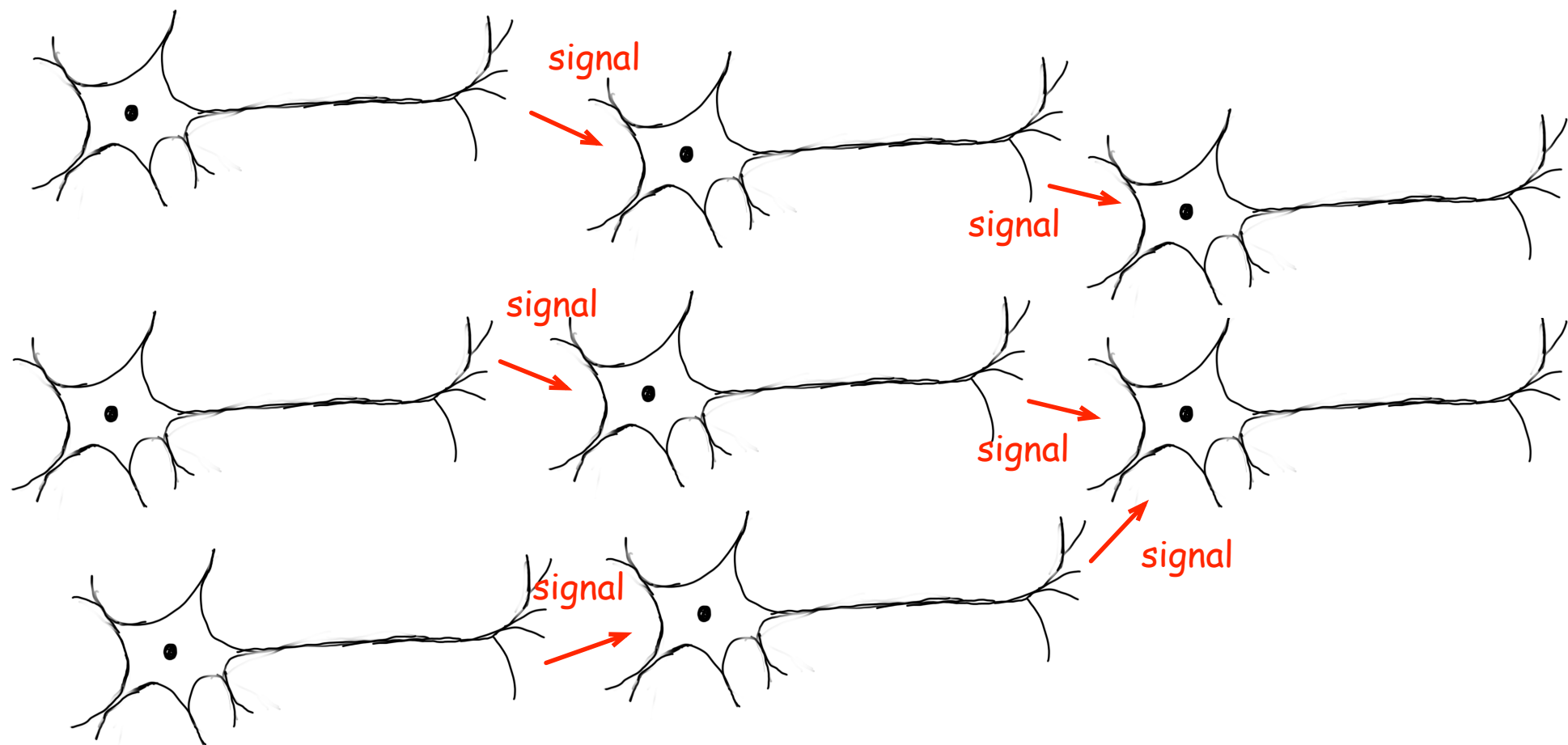
# Biological Neuron

- The brain is composed of neurons that are connected to each other, the process of thinking and decision making is done by sending electrical pulses between neurons.
- Here is a simplified biological neuron, which is composed of three components, dendrite, axon, and terminals.
- Neurons transmit an electrical signal from *dendrites* to terminals through *axons*. Then the signal passed between neurons with the same structure.



# Biological Neural Network

- A simplified structure of the brain could be shown as a set of neurons that passes the electrical signal along with each other.





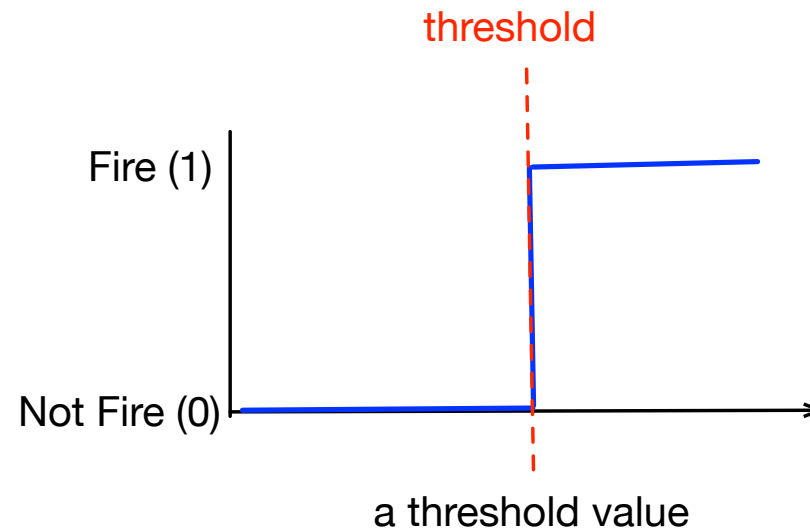
# Biological Neuron Connection and Seeking for a Connection

- <https://www.youtube.com/shorts/DETL1I5Byeo>
- [https://www.youtube.com/watch?v=9ebifjoFtJs&ab\\_channel=DrLilaLandowski](https://www.youtube.com/watch?v=9ebifjoFtJs&ab_channel=DrLilaLandowski)

# When does a neuron fire a signal?

- The process of transferring electrical signals through neurons is called firing. When a neuron fires a signal?
- Scientific evidence suggests that a nucleus of a neuron should reach a *threshold* of receiving an electrical signal and it fires the signal *only if that threshold is reached*.
- If the nucleus does not reach that threshold it does not fire the signal.
- This makes sense because noises, which are electrical signals less than the threshold are not transferred and only information which is useful will be transferred.

# When does a neuron fire a signal?



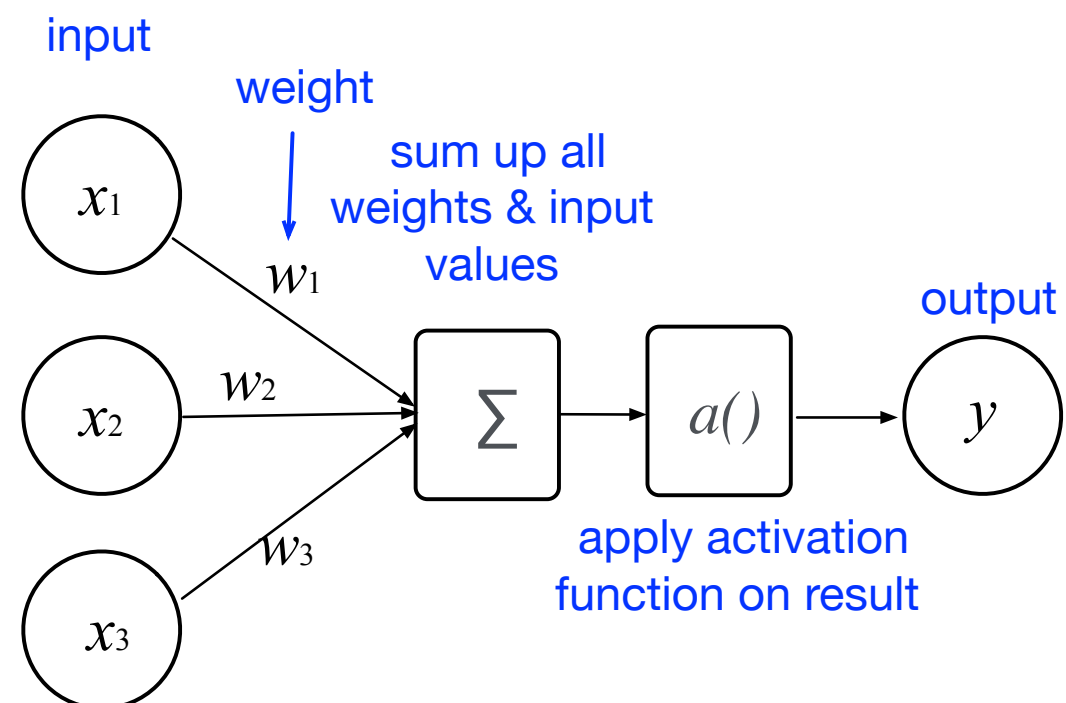
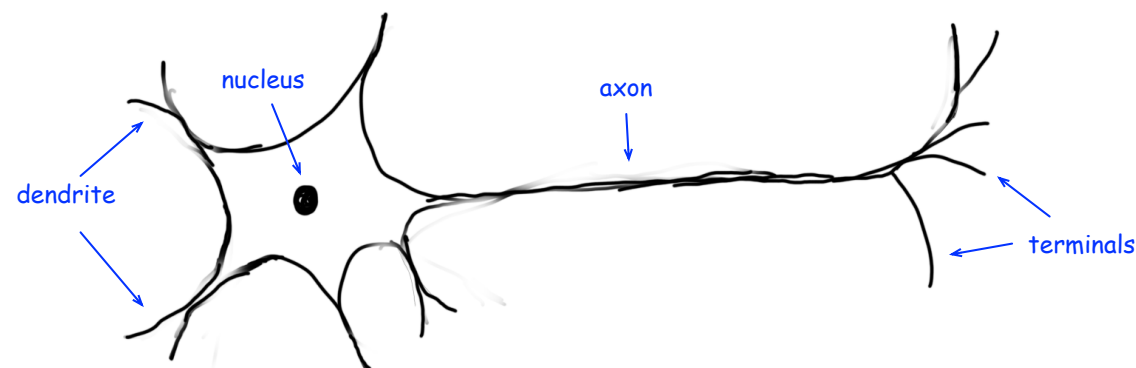
- The process of transferring electrical signals through neurons is called firing. When a neuron fires a signal?
- Scientific evidence suggests that a nucleus of a neuron should reach a *threshold* of receiving an electrical signal and it fires the signal *only if that threshold is reached*.
- If the nucleus does not reach that threshold it does not fire the signal.
- This makes sense because noises, which are electrical signals less than the threshold are not transferred and only information which is useful will be transferred.

# Outline

- The rationale behind Deep Learning
- **Artificial Neural Network and its Concepts**
- Perceptron and Multilayer Perceptron
- Activation Functions,
- Cost Function and Neural Network Optimizers
- Backpropagation
- Regularization in Neural Network

# Artificial Neuron

- The first digital neuron, inspired by the biological neuron, was proposed by McCulloch and Pitts, [McCulloch '43], and it is known as the MCP neuron. Nevertheless, it was too limited, and thus it is not widely in use anymore. Perceptron is created in 1958 [Rosenblatt '58], and it is the simplest artificial neural network (ANN) that is still in use.
- An artificial neuron is a mathematical model with a set of inputs (similar to biological dendrites), and each input is associated with a weight. Then, the activation function (similar to a biological nucleus) calculates the value and sends the result to the output (similar to biological dendrites).



# Artificial Neuron Components

- **Weight:** We have explained that each signal inside a biological neuronal network could have a different amplitude (e.g. micro-voltage), and these will be simulated in perceptron by weights.
- To better understand the need for weight, consider the prediction error, which is written as  $error = predicted\ value - actual\ value$ . If the *error* is non-zero, the perceptron algorithm changes the  $w$  of input variables and redoes the process to reduce the prediction error. *Weights are very similar to coefficients that have been used in regressions.*
- There are different weight initialization methods (at random, with all zeroes, with small values around zero, and so forth), and the user should decide on the weight. Increasing the weight increases the complexity of the network, and thus, it is better to keep weights small.

# Artificial Neuron Components

- **Bias:** Similar to linear regression, which has a constant value, each neuron also has a bias, presented as  $b$ . *Weights are used to adjust the strength of the connections between neurons. Biases, on the other hand, are used to adjust the output of a neuron before it is passed through the activation function.*
- In other words, bias can be used to measure how easy it is to fire the perceptron.
- **Activation function:** It is a function that receives the sum of weighted inputs and calculates a threshold value (take a look at the Step function).
- Based on the result of the threshold value, the activation function decides the output value. It could be a linear or non-linear function. Linear here means that there exists a hyperplane that can be modeled with an algebraic equation, and this hyperplane can classify the dataset (separate data points). However, non-linear activation functions such as the Sigmoid function are more popular.

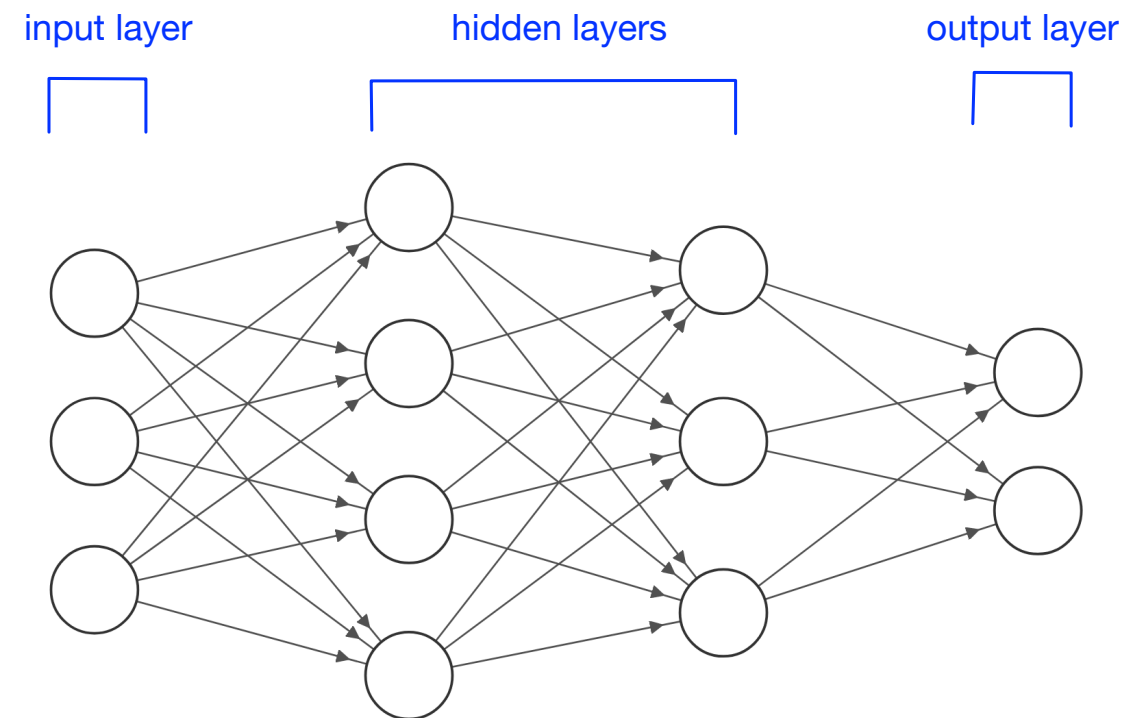
# Artificial Neuron Components

- **Bias:** Similar to linear regression which has a constant value, each neuron also has a bias, presented as  $b$ . *Weights are used to adjust the strength of the connections between neurons. Biases, on the other hand, are used to adjust the output of a neuron before it is passed through the activation function.* In other words, bias can be used to measure how easy is to fire the perceptron.
- **Activation Function:** Now, that we have understood these concepts, we could say that an artificial neuron is nothing more than linear regression and that its result is fed into an activation function,  $y = z(wx + b)$ .
- Based on the result of the threshold value the activation function decides the output value. It could be a linear or non-linear function. Linear here means that there exists a hyperplane that can be modeled with an algebraic equation, and this hyperplane can classify the dataset (separate data points). However, non-linear activation functions such as the Sigmoid function are more popular.



# Neural Network Components

- **Neural Network** is a set of connected neurons in which the output of some neurons are inputs of other neurons,. All neural networks have three layers: input, hidden, and output.
- **Input layer** is the input variable(s) that we feed into the neural network.
- **Hidden layers** are layers that are located between the input layer and output layer. The simplest neural network has one neuron as a hidden layer. Deep learning algorithms have two or more hidden layers, and because of that, they are called **deep neural networks**.



# Neural Network Components

- Since hidden layers transform the data and are different from the input or output that are known, we cannot interpret hidden layers. Therefore, they are black boxes called hidden layers.
- The **output layer** outputs the result of the neural network algorithm. Similar to the input, the output of a neural network is also a number, which we can convert back into its original value by decoding it.

# Neural Network Components

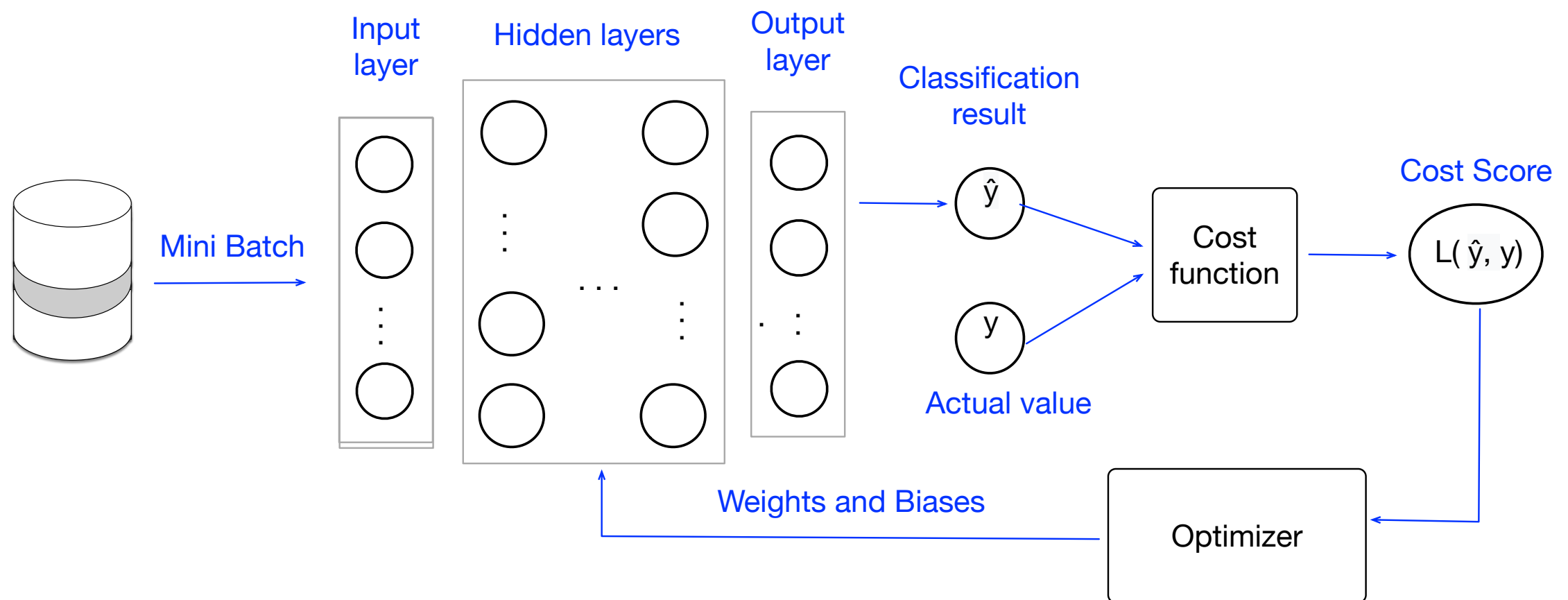
- There are three common tasks that could be done with an artificial neural network: *binary classification*, *multi-class classification*, and *regression*. Therefore, the type and number of neurons in the output layer depend on the problem we intend to solve.
- The objective of a neural network algorithm is to assign a correct label to the unlabelled data.
- This objective will be achieved by ***assigning proper values to weights and testing the network, then reconfiguring weights and biases and testing the network. This happens several times until the neural network provides satisfactory accuracy.***

# Neural Network Components

- To measure neural network accuracy, we measure how different the machine assigned labels (predicted values) are from the correct labels (actual values). This will be measured by the **cost (objective) function**.
- recall: *the cost function is used to measure loss.*
- The neural network algorithm identifies a loss score for each neuron. Then, after each iteration, it reconfigures the network's weights and biases to reduce the loss score of each neuron.
- The **optimizer** will reconfigure the network's weights and biases, such as SGD, using this process.
- The optimizer uses the loss value to update weights in the network and thus reduces the loss value for the next round (epoch).

# Neural Network Architecture

- A neural network has three core components, **layers**, **cost function**, and **optimizer**.



# Neural Network Architecture

- A neural network has three core components, **layers**, **cost function**, and **optimizer**.

Learning in the context of a neural network refers to the process of *finding weights and biases that minimize the cost function result (cost score) for the given dataset.*

*Updating the weights toward improving the accuracy of the neural network is the task of the optimizer.*

Weights and Biases

Optimizer

**Can I say an artificial neuron is a linear regression plus activation function?**

# Outline

- The rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- **Perceptron and Multilayer Perceptron**
- Activation Functions,
- Cost Function and Neural Network Optimizers
- Backpropagation
- Regularization in Neural Network



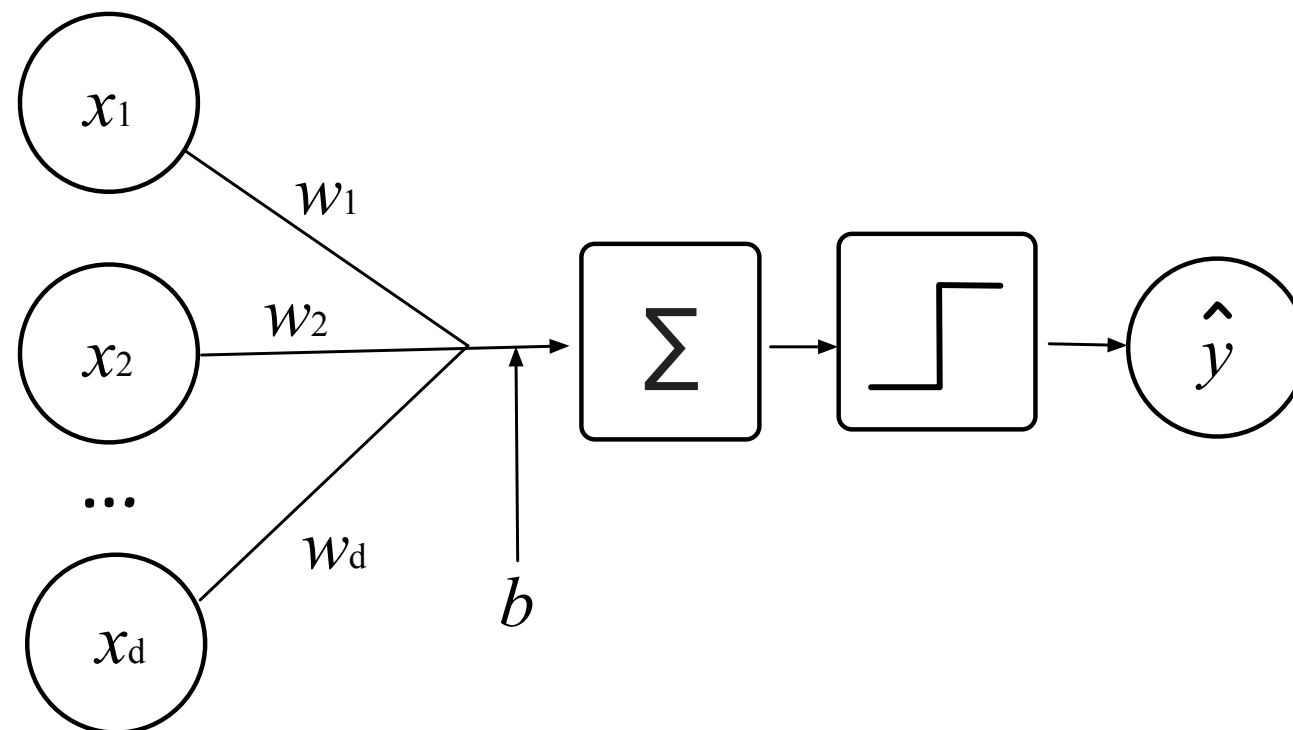
# Perceptron Algorithm

- Perceptron [Rosenblatt '58] receives a vector of input values and calculates a linear combination of input variables' values.
- If the results are greater than a threshold, the output will be equal to 1, otherwise, the output will be equal to  $-1$ .
- Assuming our input vector has  $d$  number of features  $(x_1, x_2, \dots, x_d)$ , the following equations can be used to formalize the binary classification of the perceptron algorithm.

$$\hat{y} = a\left(\sum_{i=1}^d x_i w_i + b\right)$$

# Perceptron Algorithm

- Here we see a single perceptron that receives a ' $d$ ' dimensional input data. ' $b$ ' presents the bias which is added to the result of summation. The first square is summing all weights and inputs together. The second square presents the activation function.



**What if we use a Sigmoid as activation function of a perceptron, do you recall anything?**

# Perceptron Algorithm

- Perceptron (similar to regression algorithms) operates based on the assumption that there is a hyperplane, which separates two sides of the dataset from each other. It starts with initial weights that are zeros. Then it classifies the dataset.
- Next, it goes back and checks the data points that it has been misclassified by analyzing the cost of the predicted value ( $\hat{y}$ ) which is acquired by comparing it to the actual value ( $y$ ).
- Then, the neural network changes the weight of misclassified data points to reduce the cost function, again performs the classification, and then checks the new result.
- This process continues until it reaches a specific number of iterations or weights cannot be further changed. The process of going back to the network and changing the value of weights to improve the accuracy of output is called **backpropagation** or **(Back-Prop)**, which will be explained later in detail.

# Multilayer Perceptron (Logical Operation)

- Take a look at the following logical variables and their operators. To recall logic from high school.

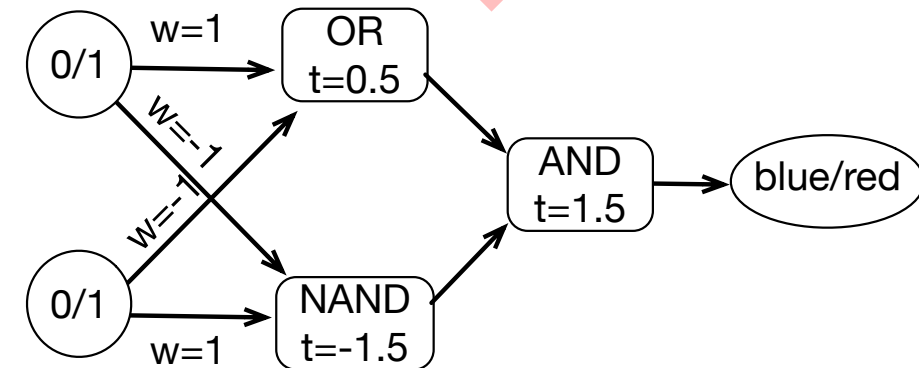
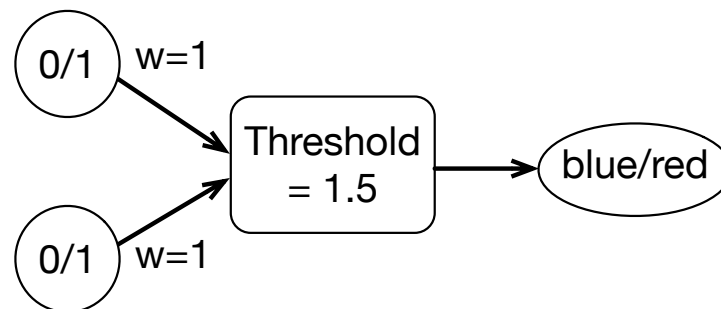
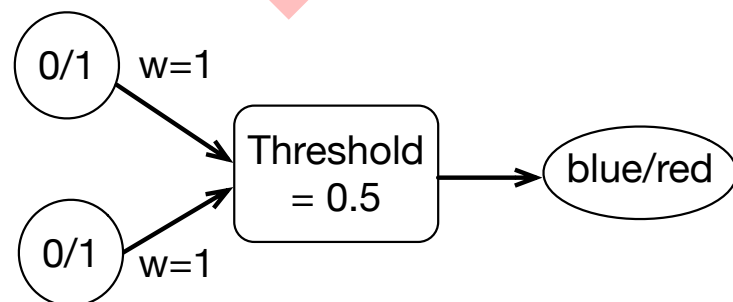
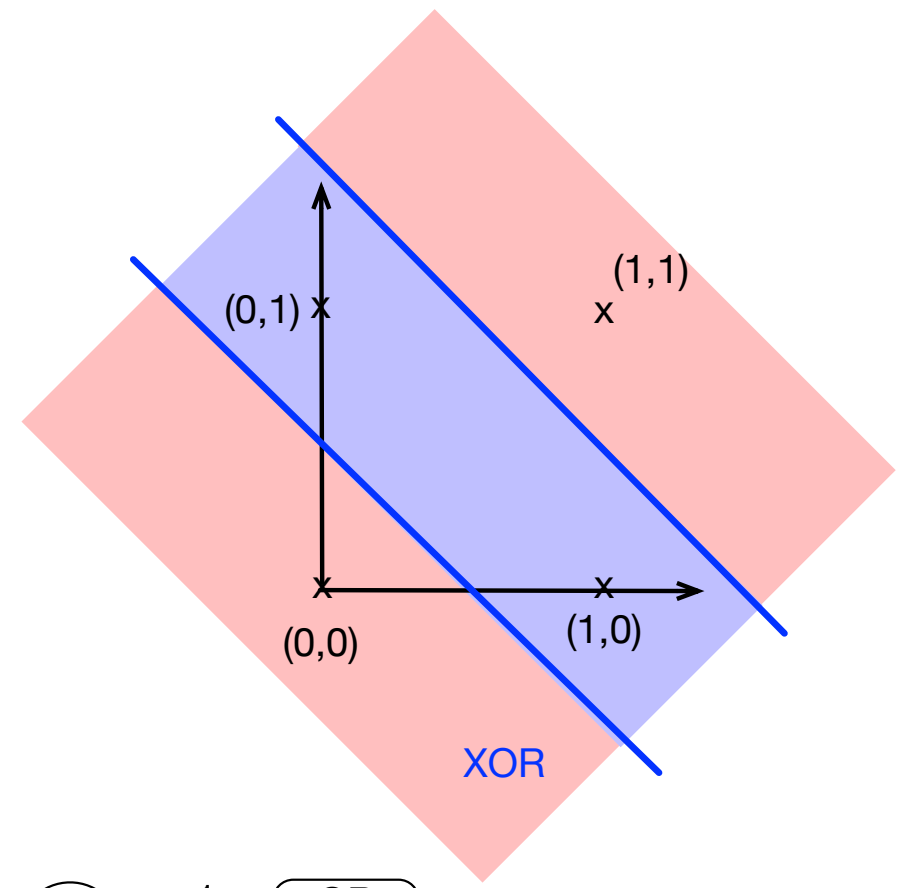
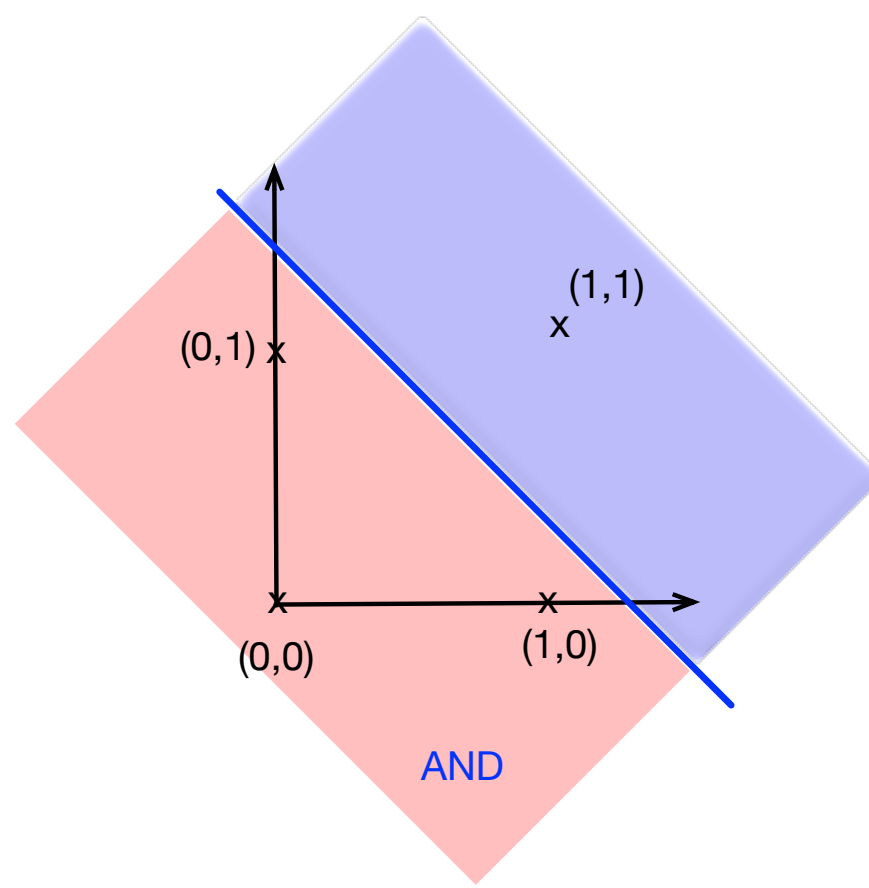
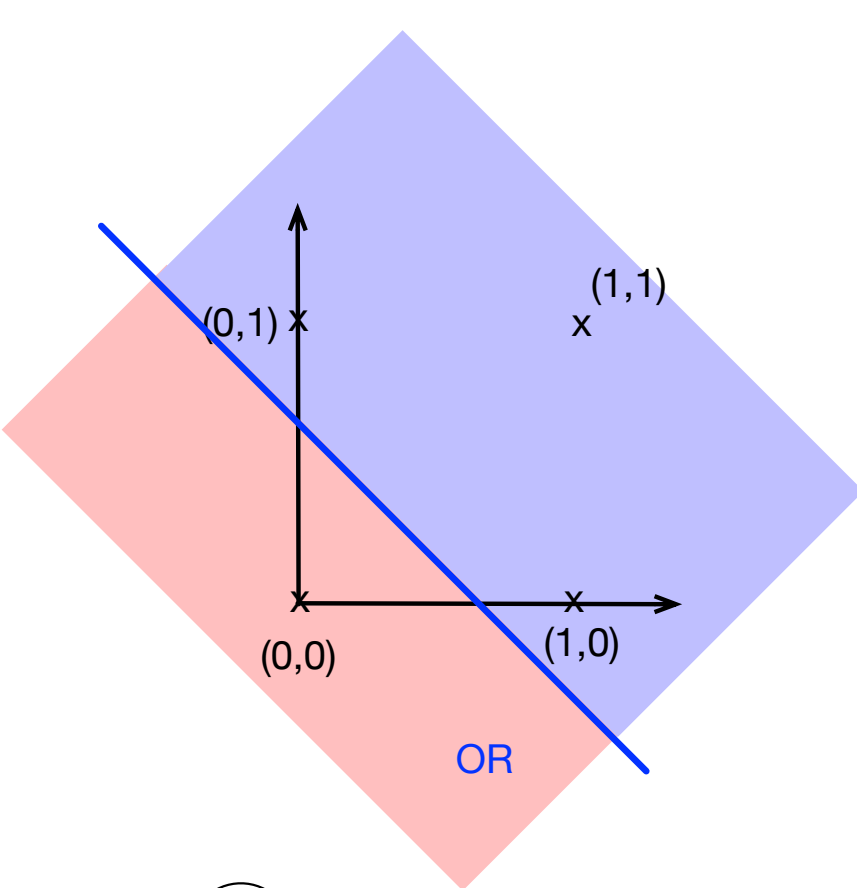
A	B	Not A	Not B	A AND B	A NAND B	A OR B	A XOR B
0	0	1	1	0	1	0	0
0	1	1	0	0	1	1	1
1	0	0	1	0	1	1	1
1	1	0	0	1	0	1	0

# Multilayer Perceptron

## (Linearity of Perceptron)

- We have explained layer-single perceptron, it is good for classifying linearly separable datasets. In the context of the logical operator, the perceptron algorithm can be used to represent boolean AND, OR, and NAND functions.
- However, a single layer perceptron cannot represent XOR boolean function. In other words, it is not possible to use it for non-linearly separable dataset [Minsky '69] with a single perceptron.
- However, it is possible to model XOR with multiple perceptrons (a network of perceptrons)

# Multilayer Perceptron (Linearity of Perceptron)

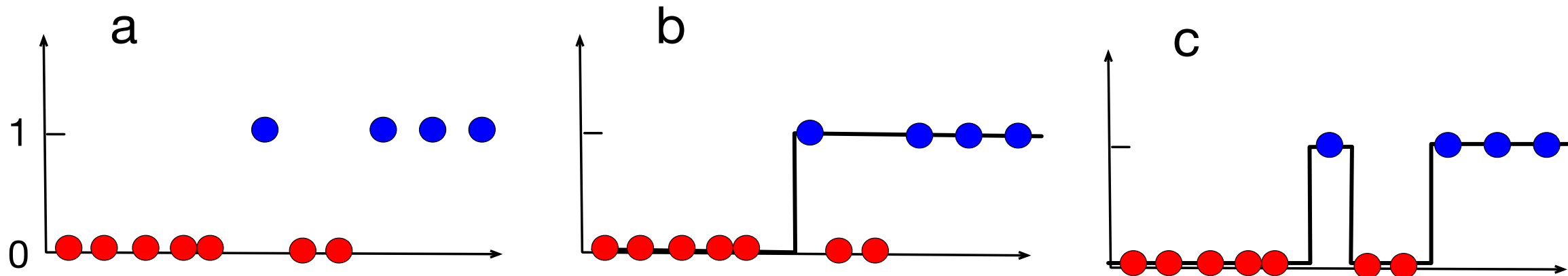


Blue = 1  
Red = 0

All these perceptrons use a *step function* as their activation function.

# MultiLayer Perceptron Example

- One step function can not classify blue and red dots, but with two step functions (result of using MLP) we can classify them correctly.





# Perceptron Example

```
#source: https://python-course.eu/machine-learning/perceptron-class-in-sklearn.php
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
import numpy as np
from sklearn.datasets import load_iris
import random
from sklearn.metrics import classification_report

iris = load_iris()
iris.target_names
targets = (iris.target==0).astype(np.int8)
print(targets)

datasets = train_test_split(iris.data,
                             targets,
                             test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets

p = Perceptron(random_state=42,
                max_iter=10,
                tol=0.001)
p.fit(train_data, train_labels)

sample = random.sample(range(len(train_data)), 10)
for i in sample:
    print(i, p.predict([train_data[i]]))

print(classification_report(p.predict(test_data), test_labels))
```

# Outline

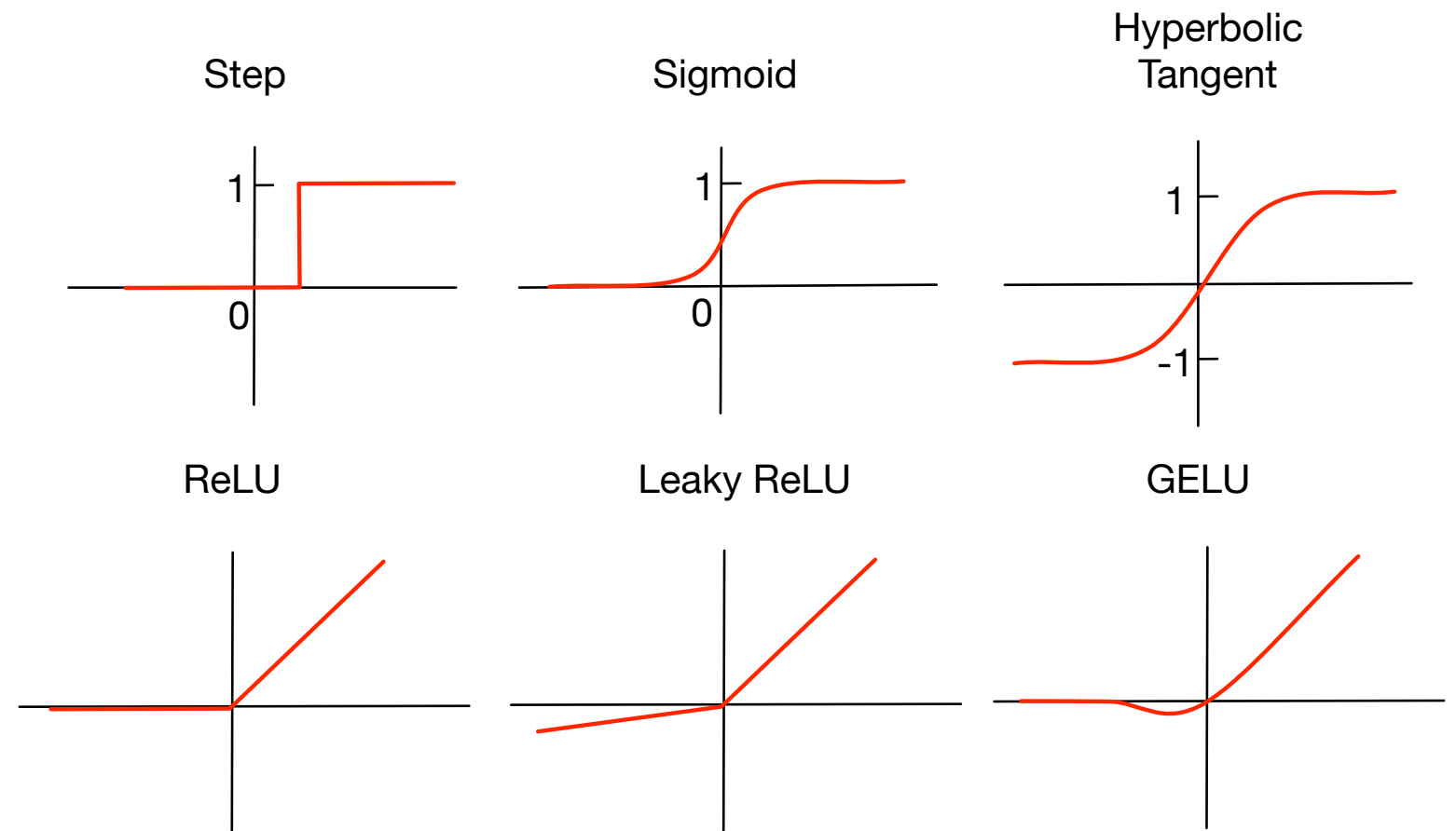
- The rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- **Activation Functions**
- Cost Function and Neural Network Optimizers
- Backpropagation
- Regularization in Neural Network

# Activation Function

Activation functions are mathematical equations that are used to set *boundaries on each neuron and decide the output value of a neuron.*

We can write the **output of each neuron** as

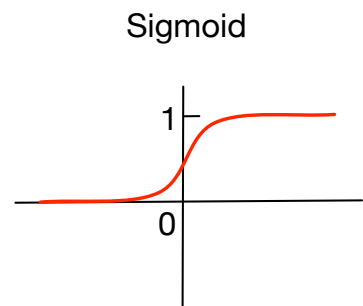
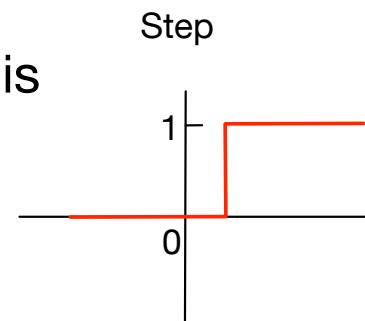
$z = wx + b$  and  $a(z)$  is the activation function which changes  $z$  to the final output for a neuron, i.e.,  $a(z)$ .



# Activation Functions

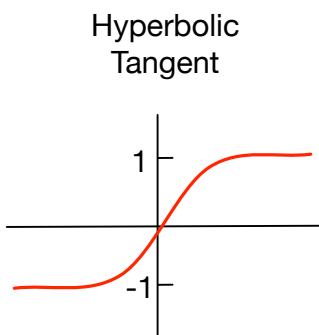
- **Step function:** The simplest form of binary class activation function is the step function which is used for binary classification. Its threshold is  $z$  and if it is larger than zero the signal will be fired ( $\hat{y} = 1$ ) otherwise, it will be 0 ( $\hat{y} = 0$ ).
- **Sigmoid function:** It is more smooth than the step function and it uses the Sigmoid equation regression. The Sigmoid function is written as follows:

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



Sigmoid is more popular than step function, because it is slightly more sensitive to small changes in the output value, and it is better suited to report *probabilities* because it gives a value between 0 and 1, step function only gives either 0 or 1, and no other numbers in between.

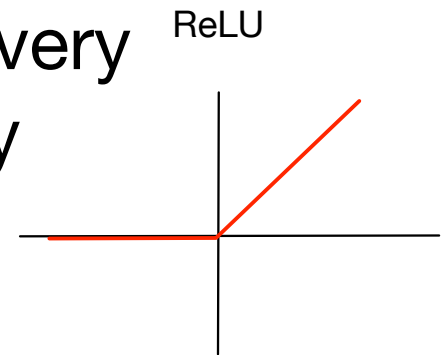
- **Hyperbolic Tangent function:** it is another activation function and its equation is written as follows:  $a(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ .



It calculates a ratio between hyperbolic sine and hyperbolic cosine, i.e, the ratio of the half-difference and half-sum of two exponential functions in the points  $z$  and  $-z$ . This activation function is very similar to the Sigmoid function, but its range is from  $-1$  to  $1$ . When we do not like to have zero as the lower boundary of the activation function, this activation function can be used.

# Activation Functions

- **Rectified Linear Unit (ReLU) function:** ReLU [Nair '10] is a very common activation function (probably the most popular binary activation function).



We can easily write it as  $\max(0, z)$ . It means if the output value is less than zero, the ReLU activation function considers it as 0; otherwise, the value of the output is the actual output value.

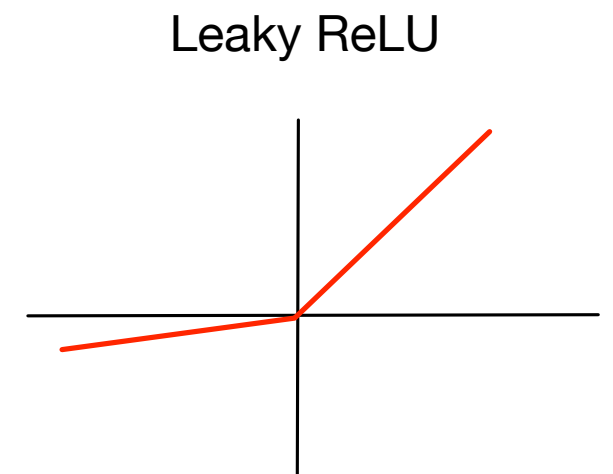
ReLU is a useful activation function while dealing with the vanishing gradient problem, which will be explained later.

The following equation formalizes the ReLU. 
$$a(z) = \begin{cases} 0 & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$$

# Activation Functions

**Leaky RELU (LReLU):** LReLU [Maas '13] is another common activation function that allows a very small negative value to have a non-zero variable. Not having zero is useful because it mitigates the challenge of *vanishing gradient* better than ReLU, which is called *dying ReLU*. LReLU equation is written as follows.

$$a(z) = \begin{cases} 0.01z & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$$



Having zero gradients can also result in slow learning, by using ReLU and LReLU resolves this.

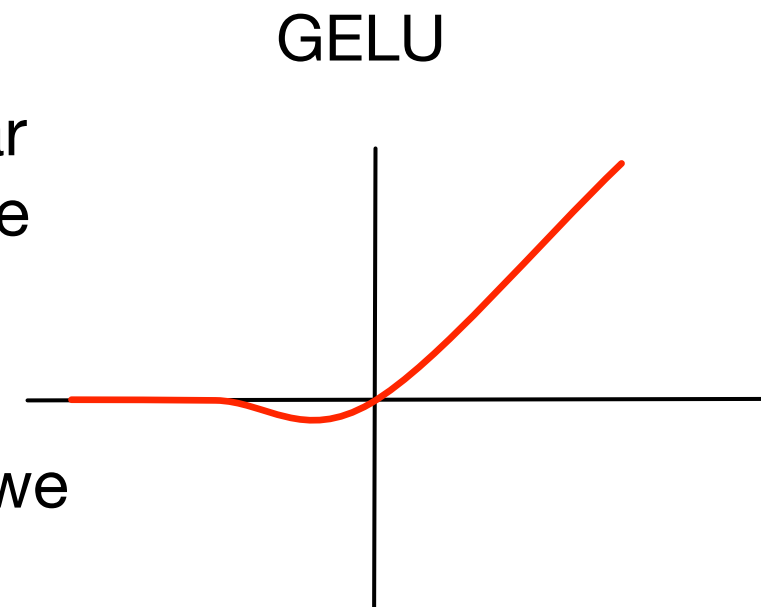
In other words, The derivative of the ReLU is 1 in the positive part, and 0 in the negative part. The derivative of the LReLU is 1 in the positive part, and it is a small fraction in the negative part.

# Activation Functions

**Gaussian Error Linear Units (GELU):** The GELU is a nonlinear activation function based on the standard Gaussian cumulative distribution function [Hendrycks '16].

Later, we will learn to regularize a neural network sometimes, we set some weights randomly to zero, i.e., drop out. GELU activation has this characteristic as well and some weights will be multiplied by zero.

In particular, assuming the  $\Phi(x)$  is the standard Gaussian cumulative distribution function (CDF), each data point  $x$  will be multiplied by this value and  $x$ .  $\Phi(x)$  is the output of GELU activation.



# Softmax Activation Functions

- Softmax is an activation function that can be used for multiclass output values. We cannot visualize it, because it has multi-dimensions (equal to the number of output classes). Assuming we have  $k$  number of classes, the equation to calculate the softmax is written as follows: 
$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}} \quad \text{for } i = 1, \dots, k$$
- Softmax activation function calculates probability distributions of the particular output over  $k$  different classes of labels.
- For example, we have a neural network to recognize if the given image includes a human, cat, or dog, then our  $k=3$ . Softmax calculates the probability of each target class over all possible target classes.
- In summary, based on the number of possible outputs, Softmax assigns each of the outputs a probability value, and the one with the highest probability is the correct output.



# Why we need Activation Function?

- Neural networks are linear transformations and chaining many linear transformations ends up having a linear transformation again and by just having a linear transformation the neural network cannot solve complex problems.
- For example, if we have  $f(x) = x - 1$  and  $g(x) = 2x + 1$ , chaining them will be  $f(g(x)) = (2x + 1) - 1 = 2x$ , which is still a linear function.
- Therefore, activation functions are used to enable the neural network to perform a non-linear transformation.

# Outline

- The rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- Activation Functions
- **Cost Functions and Neural Network Optimizers**
- Backpropagation
- Regularization in Neural Network

# Neural Network Cost Function

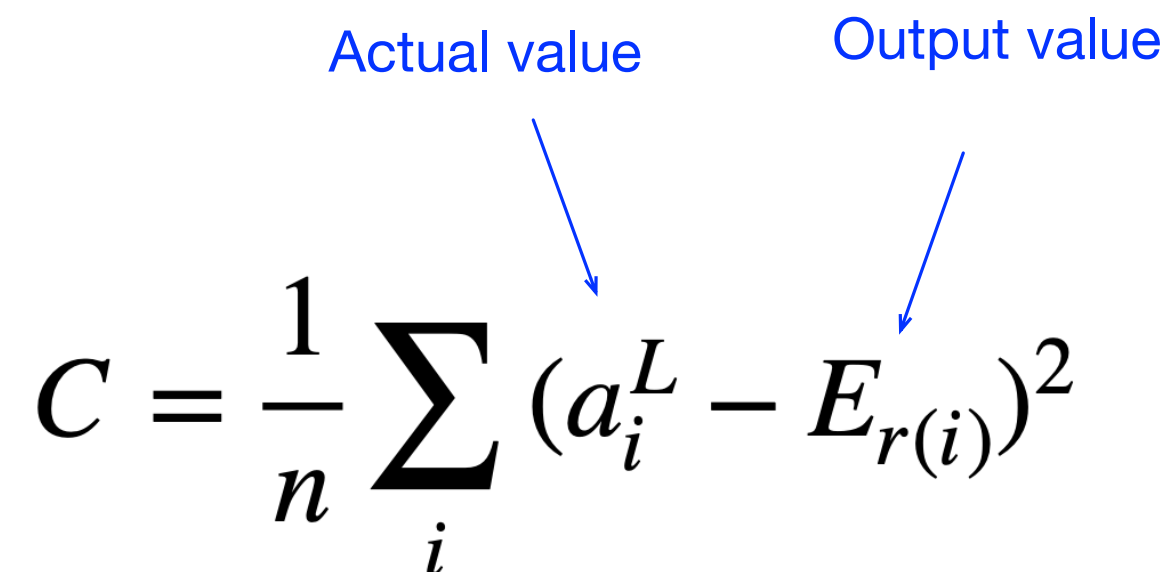
- We have explained that cost function is one the of three main components of a neural network.
- In the context of a neural network, a cost function uses the given input to measure how accurate is the output.
- After the neural networks construct the output the algorithm uses to cost function to report the loss score  $L(\hat{y}, y)$ . This loss score is calculated by the cost function

# Neural Network Cost Function

- A cost function for a single neuron is written as  $C = (W, B, S_r, E_r)$ , in which  $C$  presents the cost value,  $W$  presents weights,  $B$  presents biases,  $S_r$  presents the input of a single training sample (e.g., one record of data), and  $E_r$  presents the desired output of the given input training sample ( $S_r$ ).
- For the entire network, which has  $n$  layers and we have  $m$  number of weights in the last layer, we will have a cost function that depends on all of the weights  $C(w_1^{(1)}, w_2^{(1)}, \dots, w_m^{(n)})$ .
- Note that we use superscript when we report something specific to a layer. For example,  $a^{(2)}(x)$  means the value of the activation function at the second layer.

# Quadratic Cost (Mean Squared Error)

- Assuming  $y(x)$  presents the real value of output,  $a^L(x)$  or  $y$  presents the actual value in the last layer ( $L$  presents the last layer), and  $E_{r(i)}$  (or  $\hat{y}$ ) presents the output value.
- The quadratic cost (MSE) equation is written as follows.



The diagram shows the quadratic cost equation with two blue arrows pointing to specific terms. One arrow points from the text 'Actual value' to the term  $a_i^L$  in the equation. The other arrow points from the text 'Output value' to the term  $E_{r(i)}$  in the equation.

$$C = \frac{1}{n} \sum_i (a_i^L - E_{r(i)})^2$$

# Cross Entropy

- Cross entropy is a mathematical function that measures the differences between two statistical distributions. Assuming  $P(x)$  is one distribution and  $Q(x)$  is the second distribution, their cross entropy  $H(P,Q)$  is written as follows:

$$H(P, Q) = - \sum_x P(x) \cdot \log Q(x)$$

Assuming the dataset has  $n$  data points and  $i$  is the index of the class label, its equation is written as follows.

$$C = - \sum_i [E_{r(i)} \cdot \ln(a_i^L) + (1 - E_{r(i)}) \ln(1 - (a_i^L))]$$

The description of variables in this equation is exactly the same as the one we have explained for the quadratic cost function.

# Kullback-Leibler Divergence

KL-Divergence function is used to quantify the differences (or measuring similarity) between two probability distributions, which is written as follows:

$$D_{KL}(P, Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}$$

In the context of neural networks, we need a cost function to compare the distribution of output values with the actual values. Therefore, it is written as follows:

$$C = \sum_i E_{r(i)} \log \frac{E_{r(i)}}{a_i^L}$$

# Hellinger (Bhattacharyya) Distance

Hellinger distance [Hellinger '09] is also used to quantify the similarity between two probability distributions. Therefore, outside the context of cost function we can formalize Hellinger distance as follows:

$$H(P, Q) = \frac{1}{\sqrt{2}} ||\sqrt{P} - \sqrt{Q}||_2 \text{ or } H(P, Q) = \frac{1}{\sqrt{2}} \sum_i (\sqrt{P} - \sqrt{Q})^2$$

If we intend to ensure that the cost function result will stay between 0 and 1, we can use Hellinger distance. It is a  $L_2$  norm (Euclidean norm) that is used for probability distributions and its equation is written as follows:

$$C = \frac{1}{\sqrt{2}} \sum_i (\sqrt{a_i^L} - \sqrt{E_{r(i)}})^2$$



# Summary of Cost Functions

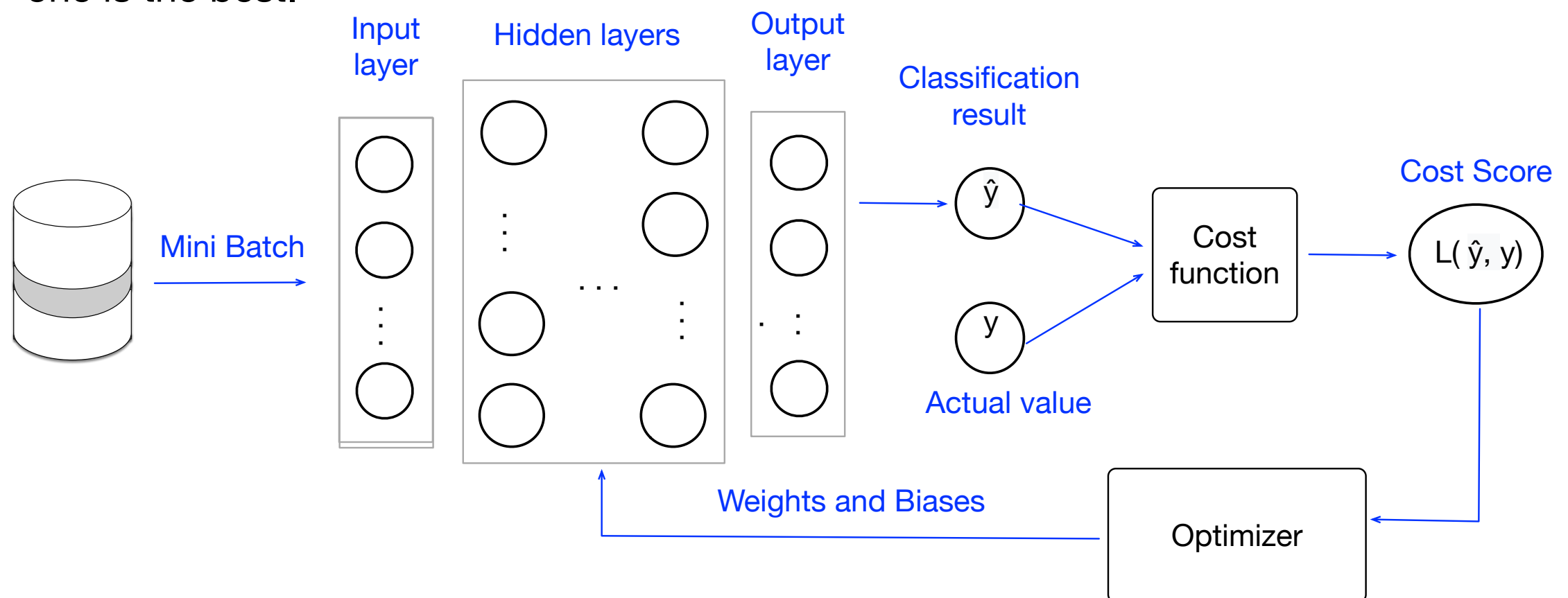
- if we deal with a regression problem we use the quadratic cost function.
- If we deal with classification we can use Cross Entropy, KL-Divergence, or Hellinger distance.
- If we deal with a binary classification, we can use Binary Cross Entropy.

# Neural Network Optimizers

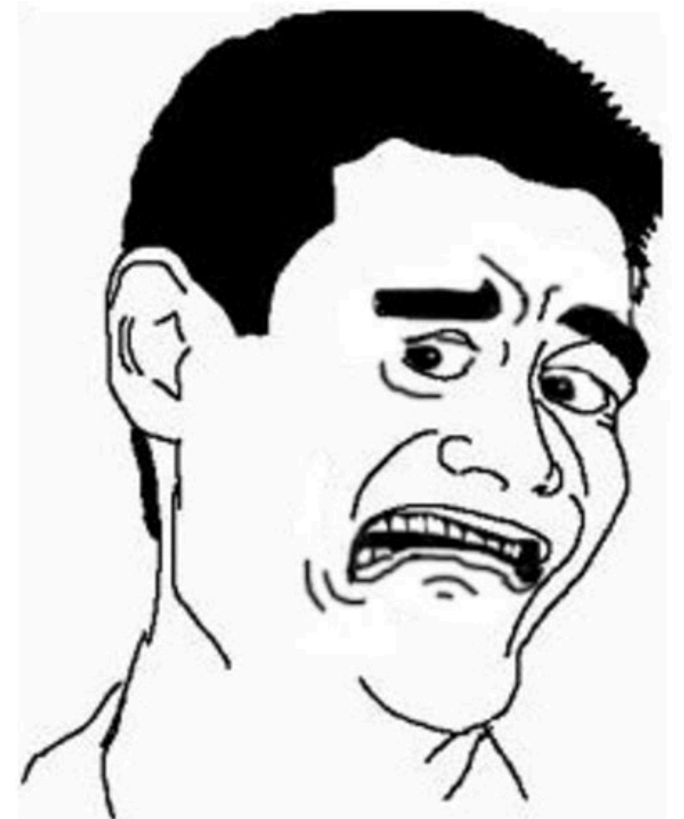
- The optimizer's job is to reduce the cost.
- In other words, the goal of the optimizer is to increase the accuracy of the algorithm by changing the model parameters.
- In the context of the neural network algorithm, this will be achieved by *minimizing the cost function by changing weight and biases*.

# Neural Network Optimizers

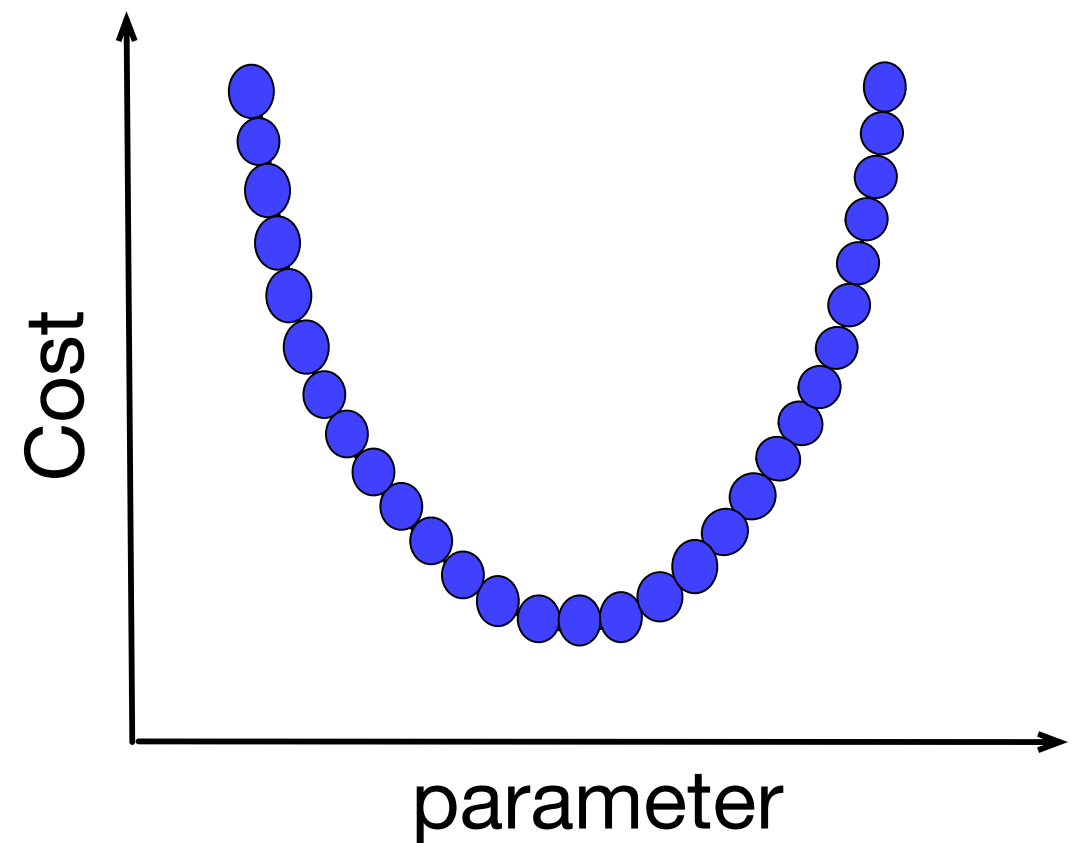
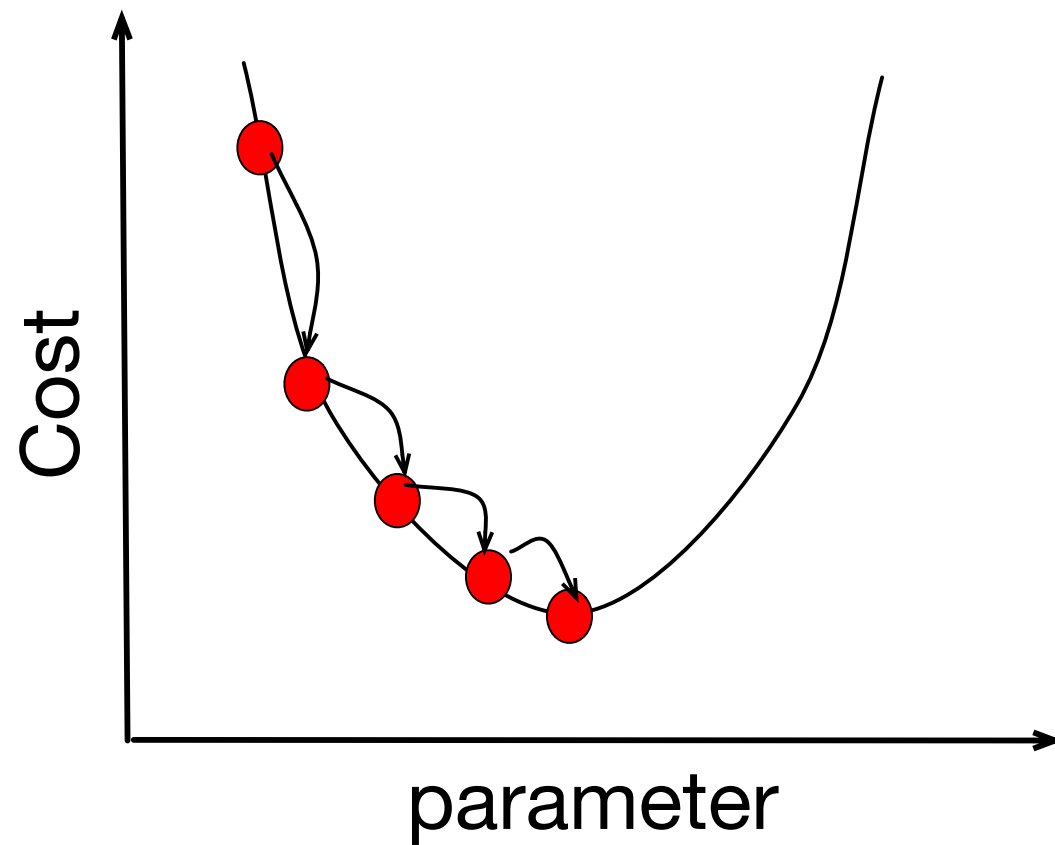
- At the end of each epoch (each iteration on the dataset), the optimizer changes weights and biases in such a way that the loss score will be reduced in the next epoch. In other words, by changing weights and biases the algorithm can reduce the loss score.
- Now a question arises, what is the best combination of weights that increases the result accuracy?
- To answer it we can try every possible combination of weight and biases and see which one is the best.



- Let's assume we have a fully connected neural network that has one input layer with four neurons, two hidden layers each with five neurons and two output neurons.
- In this case, we will have  $(4 \times 5) + (5 \times 5) + (5 \times 2) = 55$  weights and biases to configure.
- The neural network algorithm starts by assigning a random weight. Let's say we limit the weight changes to a number that can vary between 0 to 100. Therefore, to find the best weight configurations, we need to test  $100 \times 100 \dots \times 100 = 100^{55}$  possible combinations for experimenting and identify which combinations lead to the best minima. This is definitely not computationally possible, even with the strongest supercomputer in the world

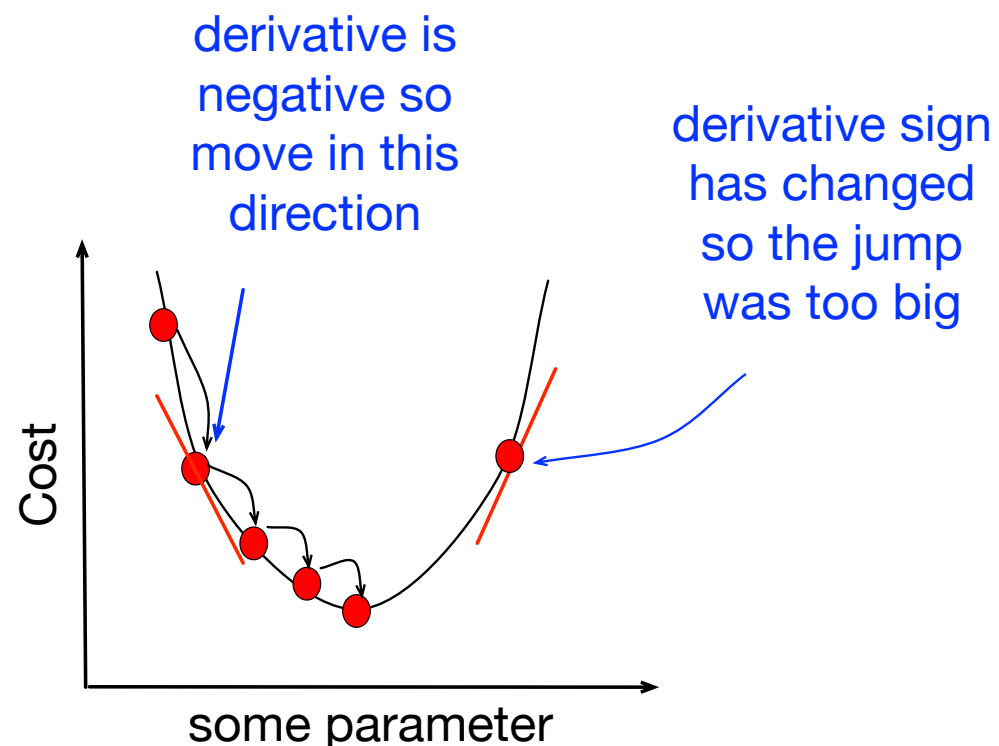


# Minima in a Simple Convex Function



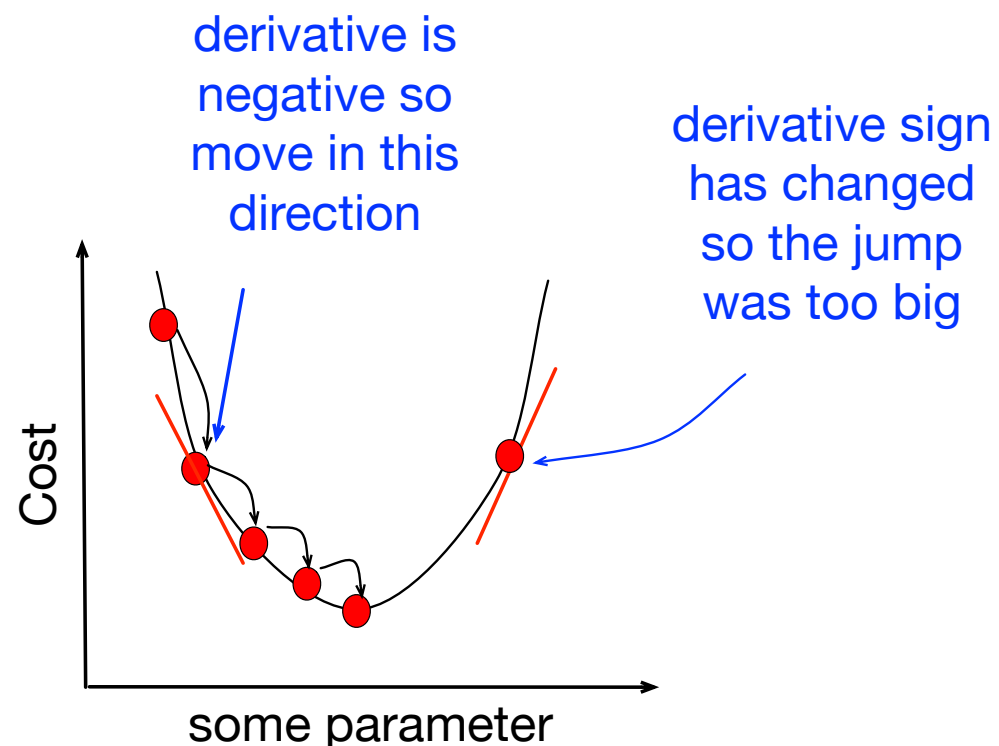
# Neural Network Optimizers

- We use a method such as the gradient descent to determine the next point and skip some points which are not useful.
- In other words, from the current data point, we need to find **(i) direction** and **(ii) step size** toward reaching the global minima and avoid experimenting with every single data point.



# Neural Network Optimizers

- At each point, the Gradient Descent algorithm calculates the deviation, if the gradient sign did not change, it means that it should take the next move in this direction to move toward minima, if the sign changes (from negative to positive), it means the gradient jump is too big and it should take the next move on the opposite direction to reach the minima.



# Decaying Learning Rate in Gradient Descent

- However, having constant size jumps is not efficient, because if step sizes are small it takes a long time to reach the minima and if they are large it might pass the minima. Therefore, a better approach is customizing the step size and this will be done through a decaying learning rate. Also if the jump is too big, then the gradient sign will change.

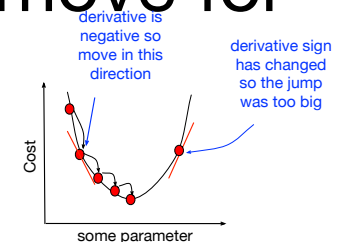


# Limitation of Gradient Descent

- We have explained that there are three types of Gradient Descent including SGD, BGD and Mini BGD.
- Gradient Descent until now is the most popular optimization approach that is being used for neural network optimization, but it has some limitations.
- It can identify the direction of the movement, but it does not specify the step size.

# Stochastic Gradient Descent

- While using SGD, the next value for the parameter will be calculated as:
- *next point = current point — (learning rate . slope (derivative of that particular parameter))*
- Assuming  $w_{t+1}$  is the weight (or any other parameter ) at the next epoch,  $w_t$  is the weight of the current epoch,  $\alpha$  is the learning rate and  $\nabla G( )$  is the gradient function.
- We estimate a value for  $w_{t+1}$  via  $w_{t+1} = w_t - \alpha \cdot \nabla G(w_t)$ . In simple words,  $\alpha \cdot \nabla G(w_t)$  specifies the amount of the move for the next step.



# Stochastic Gradient Descent Problems

1. We can see from the described equation that the direction is changing, but the step size is fixed. A fixed step size might trap the optimizer into a local minimum.
2. The noise of SGD is high, due to its stochastic (random) behavior.

# Stochastic Gradient Descent Problems <— Solution

- This problem can be reduced by using an exponentially weighted average ( $\beta$ ).
- In other words, the exponentially weighted average introduces a decay coefficient to a series of numbers. Therefore, applying this exponentially weighted average reduces the impact of older data points, and thus the noise.
- This is known as SGD with momentum.

# What is Exponentially Weighted Average

- Assume we have a sequence of numerical data, i.e.,  $\{s_1, s_2, \dots, s_n\}$  by using a weighted average, we can have a sequence of transformed numerical data, i.e.,  $\{s'_1, s'_2, \dots, s'_n\}$  and each  $s'$  at position  $t$  is calculated as  $s'_t = \beta s'_{t-1} + (1 - \beta)s_t$
- For example, three sequential data in the transformed sequence with weighted averages will be written as follows.

$$s'_t = \beta s'_{t-1} + (1 - \beta)s_t$$

$$s'_{t-1} = \beta s'_{t-2} + (1 - \beta)s_{t-1}$$

$$s'_{t-2} = \beta s'_{t-3} + (1 - \beta)s_{t-2}$$

Therefore, by combining these series and applying some mathematical simplification to them, we can have the following:

$$s'_t = \dots + \beta\beta(1 - \beta)s_{t-2} + \beta(1 - \beta)s_{t-1} + (1 - \beta)s_t$$

We can see that as the data gets older in the series, its impact on the equation gets smaller because more numbers of  $\beta$ s will be multiplied by it.

# What is Exponentially Weighted Average

- Assume we have a sequence of numerical data, i.e.,  $\{s_1, s_2, \dots, s_n\}$  by using a weighted average, we can have a sequence of transformed numerical data, i.e.,  $\{s'_1, s'_2, \dots, s'_n\}$  and each  $s'$  at position  $t$  is calculated as  $s'_t = \beta s'_{t-1} + (1 - \beta)s_t$
- For example, three sequential data in the transformed sequence with weighted averages will be written as follows.

$$s'_t = \beta s'_{t-1} + (1 - \beta)s_t$$

$$s'_{t-1} = \beta s'_{t-2} + (1 - \beta)s_{t-1}$$

$$s'_{t-2} = \beta s'_{t-3} + (1 - \beta)s_{t-2}$$

Therefore, by combining these series and doing some mathematical simplification to them, we can have the following:

Let's use a plain  
Mathematical example on  
the board.

$$s'_t = \dots + \beta\beta(1 - \beta)s_{t-2} + \beta(1 - \beta)s_{t-1} + (1 - \beta)s_t$$

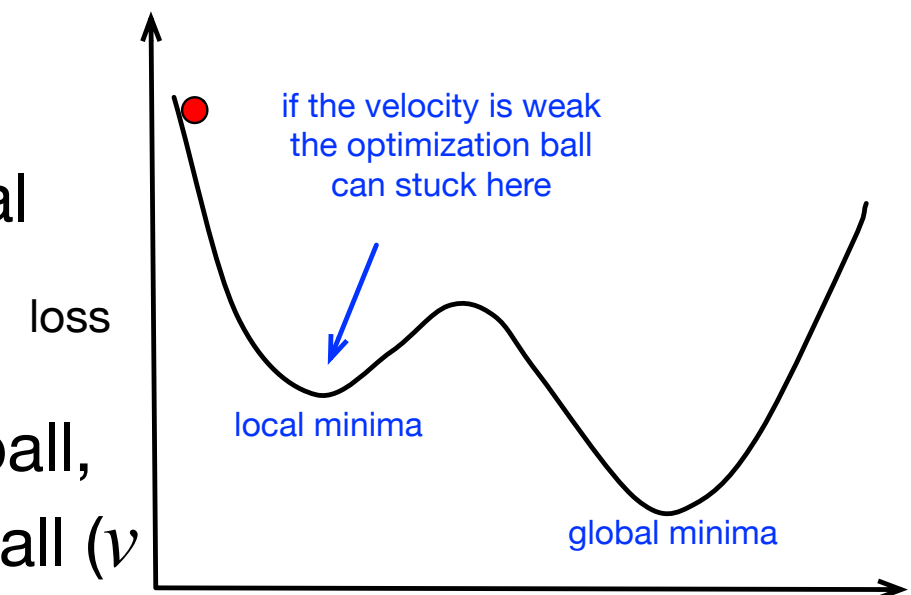
We can see that as the data gets older in the series, its impact on the equation gets smaller because more numbers of  $\beta$ s will be multiplied by it.

# SGD with Momentum

- SGD with momentum uses an exponentially weighted average to create a velocity. To understand velocity, assume we are rolling a ball inside the bowl shape function and our intention is to get the ball into global minima.
- Derivatives present the acceleration of the moving ball, and momentums add to the velocity of the moving ball ( $v$  or *retained gradient*)

# SGD with Momentum

- SGD with momentum uses an exponentially weighted average to create a velocity. To understand velocity, assume we are rolling a ball inside the bowl shape function and our intention is to get the ball into global minima.
- Derivatives present the acceleration of the moving ball, and momentums add to the velocity of the moving ball ( $v$  or *retained gradient*)
- **Momentum ( $\eta$  or *momentum coefficient*) is used to increase the velocity of the optimization ball** and thus it will be fast enough to jump out of the local minima and move forward toward global minima. In other words, *momentum is reducing the oscillation of the optimizer algorithm and thus it can reach the minima faster.*





# SGD with Momentum Algorithm

- The SGD with momentum algorithm can be described as follows:

$$v_t = 0 \quad \# \text{ velocity}$$

$$\eta = 0.9 \quad \# \text{ momentum coefficient}$$

$$\alpha = 0.01$$

*while* ( $w_t$  *not converged*) { *#e.g. converged in this context: loss*  $< 0.1$

$$v_{t+1} = v_t \cdot \eta + \alpha \cdot \nabla G(w_t)$$

$$w_{t+1} = w_t - \alpha \cdot v_{t+1}$$

$$v_t = v_{t+1}$$

$$t = t + 1$$

}

$$\text{SGD: } w_{t+1} = w_t - \alpha \cdot \nabla G(w_t)$$

# Momentum as a Hyperparameter

- In Keras, you can only need to write the following for the model and the SGD is the SGD with momentum.

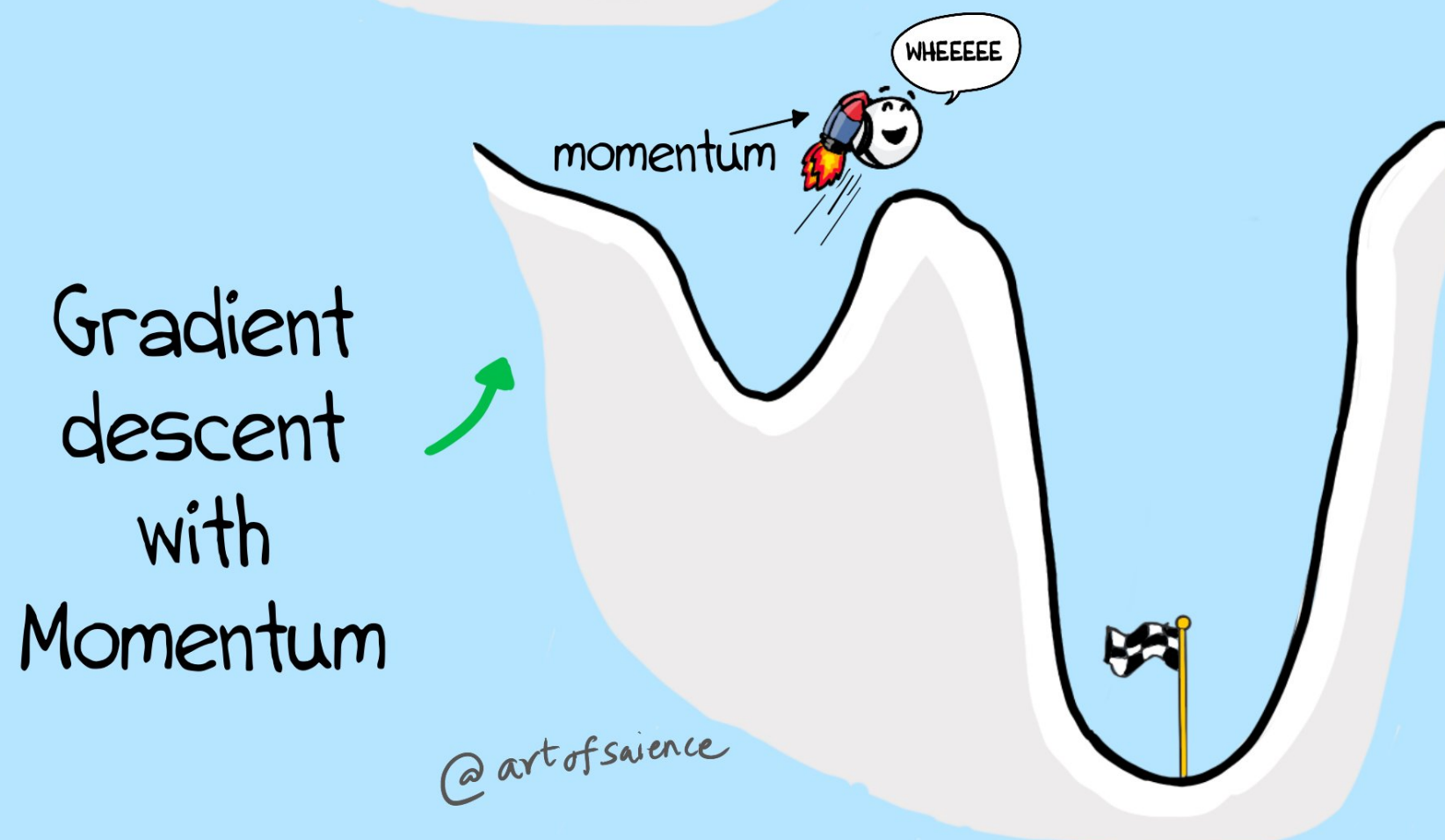
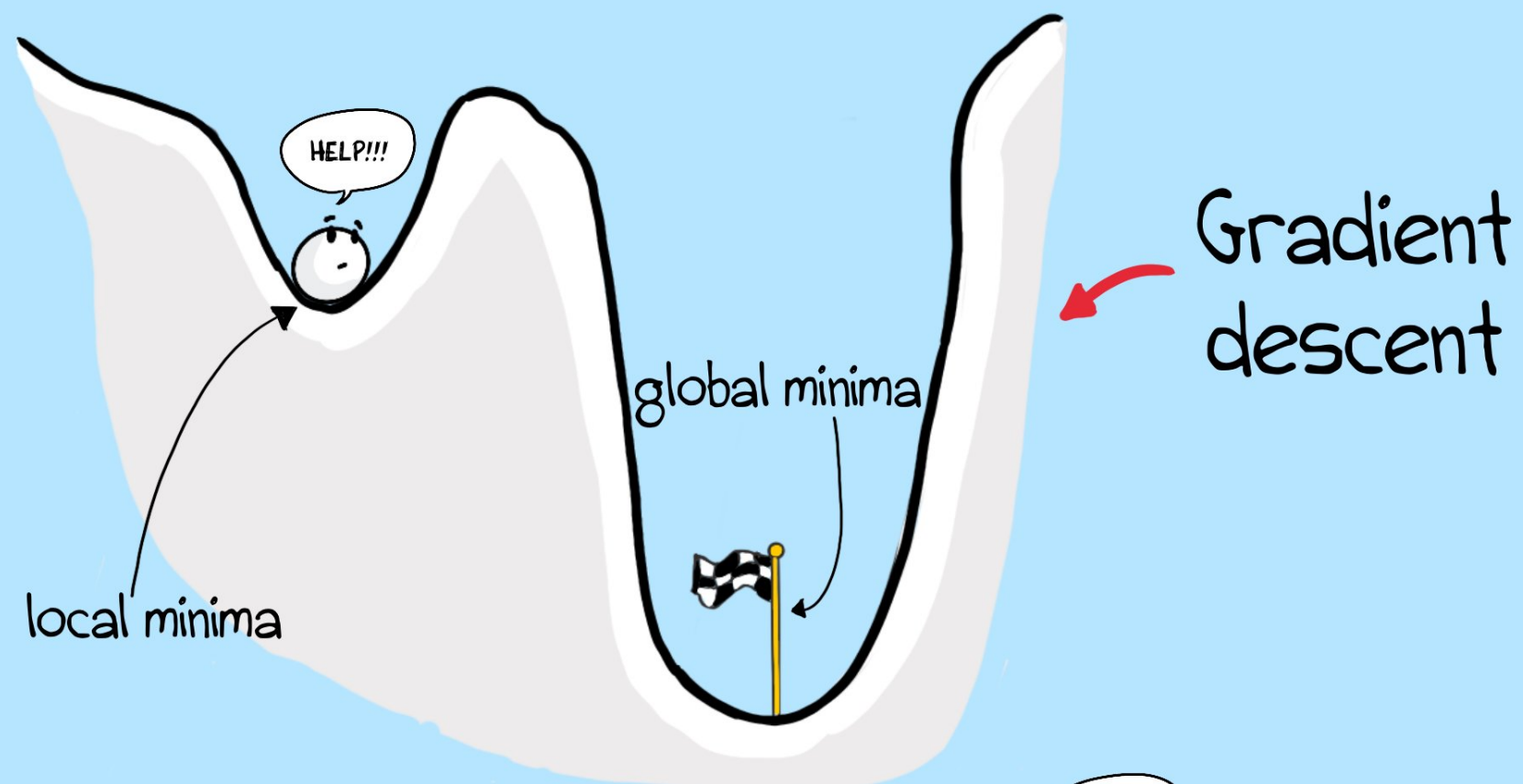
```
model.compile(loss='categorical_crossentropy', optimizer='SGD')
```

if you need to give some parameter value, you can use following code in Tensorflow:

```
tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.0,  
nesterov=False, name='SGD', **kwargs)
```

In Pytorch, it is written as follows:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```



@artofscience

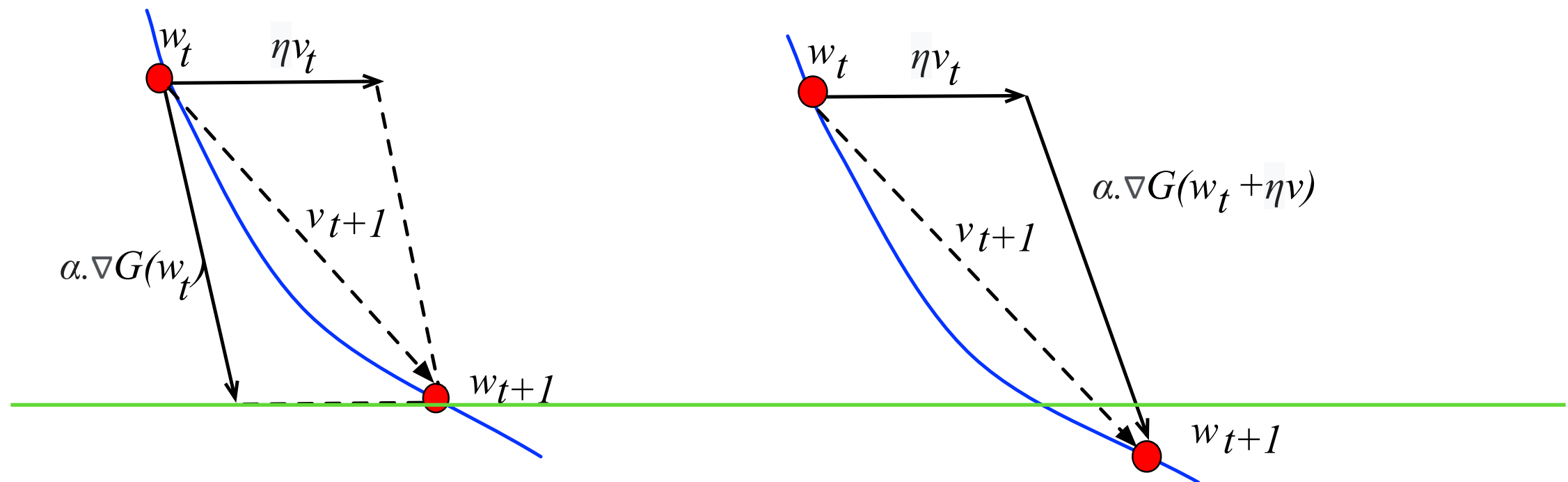
# Nesterov Momentum

- By using classical momentum (the one that we have explained), the gradient is always moving toward the correct direction, but momentum may not necessarily move in the correct direction.
- Nesterov Accelerated Gradient (NAG) or Nesterov momentum [Nesterov '83] improves the next step and if the momentum goes in the wrong direction, then the gradient can go toward the correct direction.

# Nesterov Momentum

(left) classical momentum for calculating the next parameter,

(right) Nesterov momentum is used to calculate the next parameter on the cost function. The blue line presents a small part of the cost function, and each new  $w$  is moving toward minima.



- In both examples, the momentum moves in the wrong direction, but in the Nesterov one, the gradient is started after the momentum, and thus it moves more toward the correct direction.
- While using pure SGD, the next point on the cost function will be determined by the following equation:  $w_{t+1} = w_t + -\alpha \cdot \nabla G(w_t)$
- By using SGD with the momentum, the next point on the cost function will be determined as:

$$w_{t+1} = w_t + v_{t+1} \cdot \eta - \alpha \cdot \nabla G(w_t) \leftarrow \text{momentum}$$

- Nesterov calculates velocity and weight differently from classical momentum, and it uses the following equations to calculate them.

$$v_{t+1} = v_t \cdot \eta + \alpha \cdot \nabla G(w_t + \eta v_t) \leftarrow \text{Nesterov Momentum}$$

$$w_{t+1} = w_t + v_{t+1} \cdot \eta$$

# Adagrad

- Until now we assume that the learning rate is a constant.
- Experiments show that a learning rate in a multi-dimensional (or features) dataset in some dimensions (or features) is changing very fast and in some dimensions (or features) it is changing very slowly.
- For example, usually, weights change more frequently than biases, which change less frequently.
- This means that a neural network could benefit from a *dynamically changing learning rate that adapts its change pace to each feature*.

# Adagrad

- The Adagrad algorithm [Duchi '11] adaptively scales the learning rate for each weight.
- It adapts the learning rate of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical/previous squared values.



- Assuming  $\alpha'$  is a learning rate for Adagrad, we can re-write the SGD equation as  $w_{t+1} = w_t - \alpha'_t \cdot \nabla G(w_t)$ .
- Here  $\alpha'_t$  is specified based on the previous epoch and previous weights, therefore, we add a subscript as  $t$ , which presents the learning rate in  $t$  iteration (epoch). The value for  $\alpha'_t$  will be calculated as follows:  $\alpha'_t = \frac{\eta}{\sqrt{G_t + \epsilon}}$
- $\epsilon$  is used to be sure the denominator will never be zero, and it is recommended to set it  $10^{-8}$ , which is an extremely small number.
- $\eta$  is a constant value similar to the learning rate, and it is recommended to have it 0.001.
- $G_t$  is the variable that is performing the magic. It is the *sum of squares of all previous gradients up to the current  $t$* , as it is calculated by using the following equation.

$$G_t = \sum_{i=1}^t \nabla G(w_t)^2$$

- The use  $G_t$  enables Adagrad to reduce the impact of parameters that have large gradients (features that are frequent in the dataset) and increases the impact of parameters that have low gradients (features that are not frequent in the dataset).

# AdaGrad

$$\alpha'_t = \frac{\eta}{\sqrt{G_t + \epsilon}}$$
$$w_{t+1} = w_t - \alpha'_t \cdot \nabla G(w_t)$$
$$G_t = \sum_{i=1}^t \nabla G(w_t)^2$$

a constant value  $\eta$

used to avoid having zero in denominator  $\epsilon$

- let's substitute the equation with some number and see the result. Assume we have a frequent feature ( $f_1$ ) and its  $G_t = 3.46$  and another less frequent feature ( $f_2$ ) that its  $G_t = 0.16$ .

$$\alpha'_t(f_1) = \frac{0.01}{\sqrt{3.46 + 10^{-8}}} = \frac{0.01}{1.86} = 0.005$$

and

$$\alpha'_t(f_2) = \frac{0.01}{\sqrt{0.16 + 10^{-8}}} = \frac{0.01}{0.4} = 0.025$$

Now by comparing 0.005 and 0.025 we can realize that the magic that Adagrad does is based on the sum of previous gradients of a parameter.

# Summary of AdaGrad

a constant value

used to avoid having zero in denominator

$$w_{t+1} = w_t - \alpha'_t \cdot \nabla G(w_t)$$
$$\alpha'_t = \frac{\eta}{\sqrt{G_t + \epsilon}}$$
$$G_t = \sum_{i=1}^t \nabla G(w_t)^2$$

- *Adagrad reduces the focus on the parameter that is always happening by decreasing their learning rate and allows parameters that have lots of zeros (sparse features) to have larger learning rate. <— socialist*

# RMSprop

- Similar to Adagrad, RMSprop [Hinton '12] on mitigating focuses the challenge of having gradients of different sizes (too large or too small). RMSprop builds on top of the **Rprop** [Riedmiller '93]
- Rprop uses (i) the sign of gradient, (ii) adapting the step size to each gradient.
- Rprop first checks the sign of two consecutive gradients. If the sign has been changed, this means the jump was too large, and thus, the minima have been passed and not reached.
- In the next step, it decreases the jump size by multiplying it to a number less than 1, e.g.  $\eta^- = 0.5$ . If the sign has not been changed in two consecutive gradients, this means the jump was correct, and we are moving in the correct direction. Therefore, the step size could be increased by multiplying it to a number larger than 1, e.g.,  $\eta^+ = 1.2$ . For example, if  $\nabla G(w_t) = -1$  and  $\nabla G(w_{t+1}) = -2$ , this means the algorithm is moving in the correct direction, and step size can increase, and it calculates the next weight as follows:  $w_{t+1} = w_t + \eta^+ \nabla G(w_t) = 1.2$ .

# RProp Limitation

- Rprop is very good when we have a small dataset. As soon as the dataset gets large we should go for mini batch Gradient Descent (mini BGD), and Rprop cannot handle mini BGD properly.
- For example, we have five mini-batches, whose gradients are as follows: -0.4, -0.3, -0.25, -0.2, -0.14, -0.1, 0.7, 0.9. Here, Rprop increases the weight six times and decreases it only once from (-0.1 to 0.7). This means RProp coefficients are canceling each other, and the weights grow larger, despite insignificant changes in gradient.
- RMSProp performs a small improvement on the Rprop to resolve it.

# RMSprop

- RMSprop is similar to Adagrad with slight differences while using Adagrad, the next weight will be determined by the following equation:

- $$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla G(w_t),$$

- While using RMSprop the next weight will be determined by the following equation:

- $$w_{t+1} = w_t - \frac{\eta}{\sqrt{E(G_t) + \epsilon}} \nabla G(w_t)$$

- In this equation  $E(G_t)$  is the exponentially weighted average gradient, and it is calculated as follows:  $E(G_t) = \beta E(G_{t-1}) + (1 - \beta) \nabla G(w_t)$ .
- $\beta$  is the exponentially weighted average parameter that is assigned by the user and it is usually 0.9.  $\nabla G(w_t)$  (the gradient of the cost function with respect to  $w_t$ ),

# RMSprop Summary

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E(G_t) + \epsilon}} \nabla G(w_t)$$

$$E(G_t) = \beta E(G_{t-1}) + (1 - \beta) \nabla G(w_t)$$

- *RMSprop is similar to Adagrad, but it divides the learning rate by an exponentially decaying average of squared gradient and benefits from the sign-based decision of Rprop*

# Adam

- Adam (adaptive Gradient Descent) [Kingma '14] is another popular optimization algorithm that combines the advantages of both SGD with Momentum and RMSprop together.
- It operates by taking large jumps and as the slope is getting closer to zero it starts to take smaller jumps.



# Adam Algorithm

$$\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$$

*while* ( $w_t$  not converged) {

$$g_t = \nabla G(w_t)$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1)g_t \quad \text{\#first moment estimate (Momentum)}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2)g_t^2 \quad \text{\#second moment estimate (RMSprop)}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \text{\# corrected first-moment estimate}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad \text{\# corrected second-moment estimate}$$

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

}

We can see that a squared gradient is used to scale the learning rate like RMSprop, and a moving average of the gradient is used instead of the gradient itself as it is done in SGD with momentum.

# Few more Optimizers !



Schmidt, R. M., Schneider, F., & Hennig, P. (2021, July). Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning* (pp. 9367-9376). PMLR.

AccleGrad	(Levy et al., 2018)	HyperAdam	(Wang et al., 2019b)
ACClip	(Zhang et al., 2020)	K-BFGS/K-BFGS(L)	(Goldfarb et al., 2020)
AdaAlter	(Xie et al., 2019)	KF-QN-CNN	(Ren & Goldfarb, 2021)
AdaBatch	(Devanikonda et al., 2017)	KFAC	(Martens & Grosse, 2015)
AdaBayes/AdaBayes-SS	(Aitchison, 2020)	KFLR/KFRA	(Botev et al., 2017)
AdaBelief	(Zhuang et al., 2020)	L4Adam/L4Momentum	(Rolínek & Martius, 2018)
AdaBlock	(Yun et al., 2019)	LAMB	(You et al., 2020)
AdaBound	(Luo et al., 2019)	LaProp	(Ziyin et al., 2020)
AdaComp	(Chen et al., 2018)	LARS	(You et al., 2017)
AdaDelta	(Zeiler, 2012)	LHOPT	(Almeida et al., 2021)
Adafactor	(Shazeer & Stern, 2018)	LookAhead	(Zhang et al., 2019)
AdaFix	(Bae et al., 2019)	M-SVAG	(Balles & Hennig, 2018)
AdaFom	(Chen et al., 2019a)	MADGRAD	(Defazio & Jelassi, 2021)
AdaFTRL	(Orabona & Pál, 2015)	MAS	(Landro et al., 2020)
Adagrad	(Duchi et al., 2011)	MEKA	(Chen et al., 2020b)
ADAHESIAN	(Yao et al., 2020)	MTAdam	(Malkiel & Wolf, 2020)
Adai	(Xie et al., 2020)	MVRC-1/MVRC-2	(Chen & Zhou, 2020)
AdaLoss	(Teixeira et al., 2019)	Nadam	(Dozat, 2016)
Adam	(Kingma & Ba, 2015)	NAMSB/NAMSG	(Chen et al., 2019b)
Adam <sup>+</sup>	(Liu et al., 2020b)	ND-Adam	(Zhang et al., 2017a)
AdamAL	(Tao et al., 2019)	Nero	(Liu et al., 2021b)
AdaMax	(Kingma & Ba, 2015)	Nesterov	(Nesterov, 1983)
AdamBS	(Liu et al., 2020c)	Noisy Adam/Noisy K-FAC	(Zhang et al., 2018)
AdamNC	(Reddi et al., 2018)	NosAdam	(Huang et al., 2019)
AdaMod	(Ding et al., 2019)	Novograd	(Ginsburg et al., 2019)
AdamP/SGDP	(Heo et al., 2021)	NT-SGD	(Zhou et al., 2021b)
AdamT	(Zhou et al., 2020)	Padam	(Chen et al., 2020a)
AdamW	(Loshchilov & Hutter, 2019)	PAGE	(Li et al., 2020b)
AdamX	(Tran & Phong, 2019)	PAL	(Mutschler & Zell, 2020)
ADAS	(Eliyahu, 2020)	PolyAdam	(Orvieto et al., 2019)
AdaS	(Hosseini & Platanotis, 2020)	Polyak	(Polyak, 1964)
AdaScale	(Johnson et al., 2020)	PowerSGD/PowerSGDM	(Vogels et al., 2019)
AdaSGD	(Wang & Wiens, 2020)	Probabilistic Polyak	(de Roos et al., 2021)
AdaShift	(Zhou et al., 2019)	ProbLS	(Mahsereci & Hennig, 2017)
AdaSqrt	(Hu et al., 2019)	PStorm	(Xu, 2020)
Adathm	(Sun et al., 2019)	QHAdam/QHM	(Ma & Yaras, 2019)
AdaX/AdaX-W	(Li et al., 2020a)	RAdam	(Liu et al., 2020a)
AEGD	(Liu & Tian, 2020)	Ranger	(Wright, 2020b)
ALI-G	(Berrada et al., 2020)	RangerLars	(Grankin, 2020)
AMSBound	(Luo et al., 2019)	RMSProp	(Tieleman & Hinton, 2012)
AMSGrad	(Reddi et al., 2018)	RMSTerov	(Choi et al., 2019)
AngularGrad	(Roy et al., 2021)	S-SGD	(Sung et al., 2020)
ArmijoLS	(Viswani et al., 2019)	SAdam	(Wang et al., 2020b)
ARSG	(Chen et al., 2019b)	Sadam/SAMSGrad	(Tong et al., 2019)
ASAM	(Kwon et al., 2021)	SALR	(Yue et al., 2020)
AutoLRS	(Jin et al., 2021)	SAM	(Foret et al., 2021)
AvaGrad	(Savarese et al., 2019)	SC-Adagrad/SC-RMSProp	(Mukkamala & Hein, 2017)
BAdam	(Salas et al., 2018)	SDProp	(Ida et al., 2017)
BGAdam	(Bai & Zhang, 2019)	SGD	(Robbins & Monro, 1951)
BPGGrad	(Zhang et al., 2017b)	SGD-BB	(Tan et al., 2016)
BRMSProp	(Aitchison, 2020)	SGD-G2	(Ayadi & Turinici, 2020)
BSGD	(Hu et al., 2020)	SGDEM	(Ramezani-Kebera et al., 2021)
C-ADAM	(Tutunov et al., 2020)	SGDHess	(Tran & Cufkosky, 2021)
CADA	(Chen et al., 2021)	SGDM	(Liu & Luo, 2020)
Cool Momentum	(Borysenko & Byshkin, 2020)	SGDR	(Loshchilov & Hutter, 2017)
CProp	(Preechakul & Kijstrikul, 2019)	SHAdagrad	(Huang et al., 2020)
Curveball	(Henriques et al., 2019)	Shampoo	(Anil et al., 2020; Gupta et al., 2018)
Dadam	(Nazari et al., 2019)	SignAdam++	(Wang et al., 2019a)
DeepMemory	(Wright, 2020a)	SignSGD	(Bernstein et al., 2018)
DGNOpt	(Liu et al., 2021a)	SKQN/S4QN	(Yang et al., 2020)
DiffGrad	(Dubey et al., 2020)	SM3	(Anil et al., 2019)
EAdam	(Yuan & Gao, 2020)	SMG	(Tran et al., 2020)
EKFAC	(George et al., 2018)	SNGM	(Zhao et al., 2020)
Eve	(Hayashi et al., 2018)	SoftAdam	(Pettermann et al., 2019)
Expectigrad	(Daley & Amato, 2020)	SRSRGD	(Wang et al., 2020a)
FastAdaBelief	(Zhou et al., 2021a)	Step-Tuned SGD	(Castera et al., 2021)
FRSGD	(Wang & Ye, 2020)	SWATS	(Keskar & Socher, 2017)
G-AdaGrad	(Chakrabarti & Chopra, 2021)	SWNTS	(Chen et al., 2019c)
GADAM	(Zhang & Gouza, 2018)	TAdam	(Ilboudo et al., 2020)
Gadam	(Granzio et al., 2020)	TEKFAC	(Gao et al., 2020)
GOALS	(Chae et al., 2021)	VAdam	(Khan et al., 2018)
GOLS-I	(Kafka & Wilke, 2019)	VR-SGD	(Shang et al., 2020)
Grad-Avg	(Purkayastha & Purkayastha, 2020)	vSGD-b/vSGD-g/vSGD-l	(Schaul et al., 2013)
GRAPES	(Dellaferera et al., 2021)	vSGD-fd	(Schaul & LeCun, 2013)
Gravilon	(Kelterborn et al., 2020)	WNGrad	(Wu et al., 2018)
Gravity	(Bahrami & Zadeh, 2021)	YellowFin	(Zhang & Mitliangkas, 2019)
HAdam	(Jiang et al., 2019)	Yogi	(Zaheer et al., 2018)

# Optimizer Experiments

<https://ruder.io/optimizing-gradient-descent>

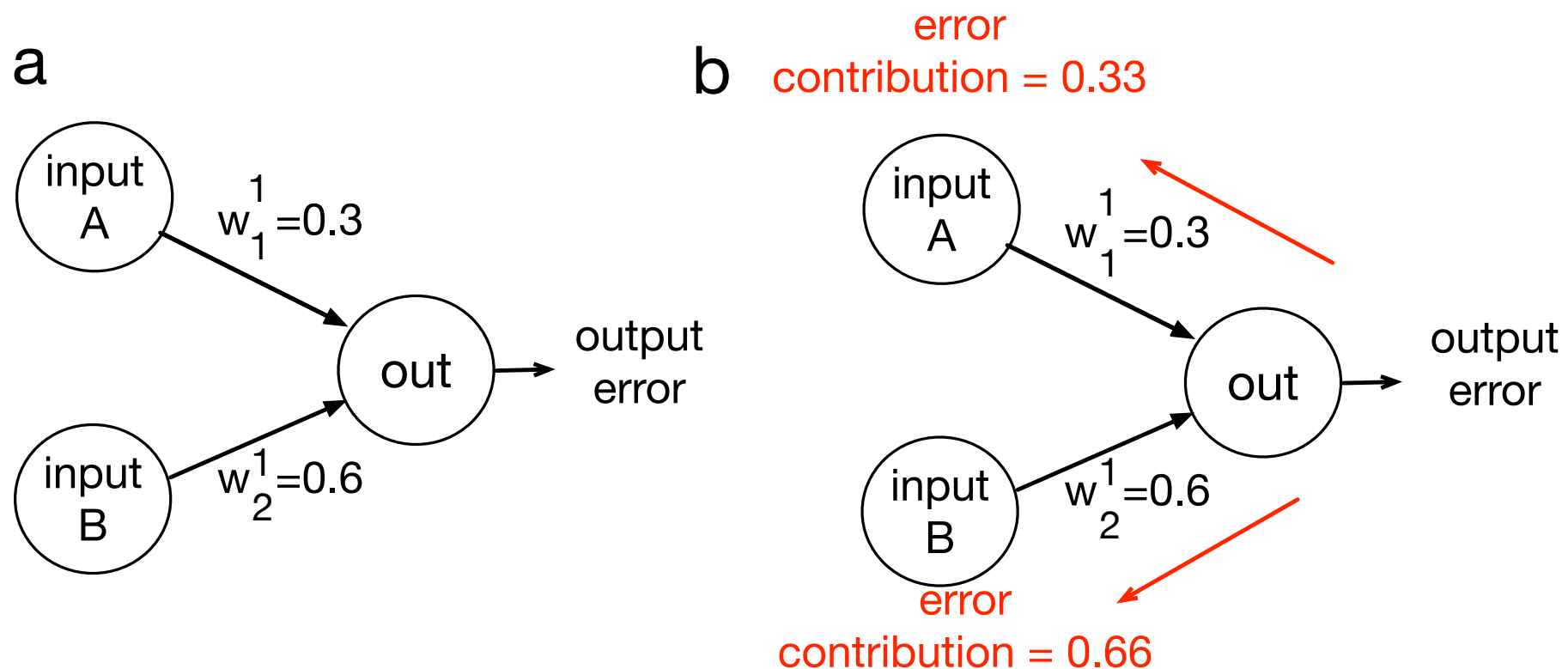
# Outline

- The rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- Activation Functions
- Cost Functions and Neural Network Optimizers
- **Backpropagation**
- Regularization in Neural Network

# Backpropagation

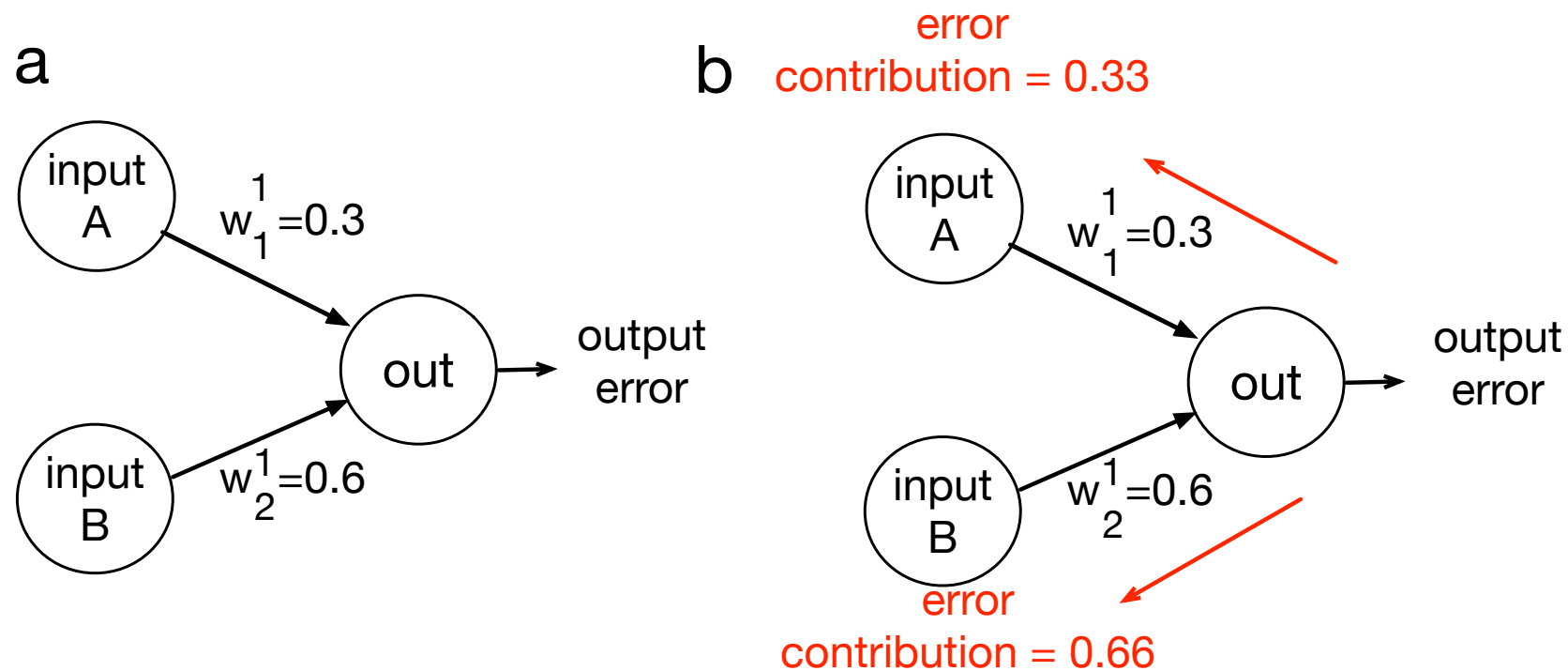
- The Backpropagation (Backprop) algorithm was introduced back in 1960 [Kelley '60, Bryson '62], and later, in 1986, it was generalized and popularized by [Rumelhart '86].
- We have explained that the objective of the cost function is *to change weight and biases toward reducing the output's accuracy (reducing the loss score)* and thus making a better prediction.
- How does the neural network change weight and biases to improve its accuracy? Starting from the output, it “goes back” to the network after each epoch, and then reconfigures weight and biases to reduce the loss score.

# Backpropagation



the contribution of input neuron A to the error can be calculated as  $\frac{0.3}{0.3 + 0.6} = 0.33$

and the contribution of input neuron B will be calculated as  $\frac{0.6}{0.3 + 0.6} = 0.66$

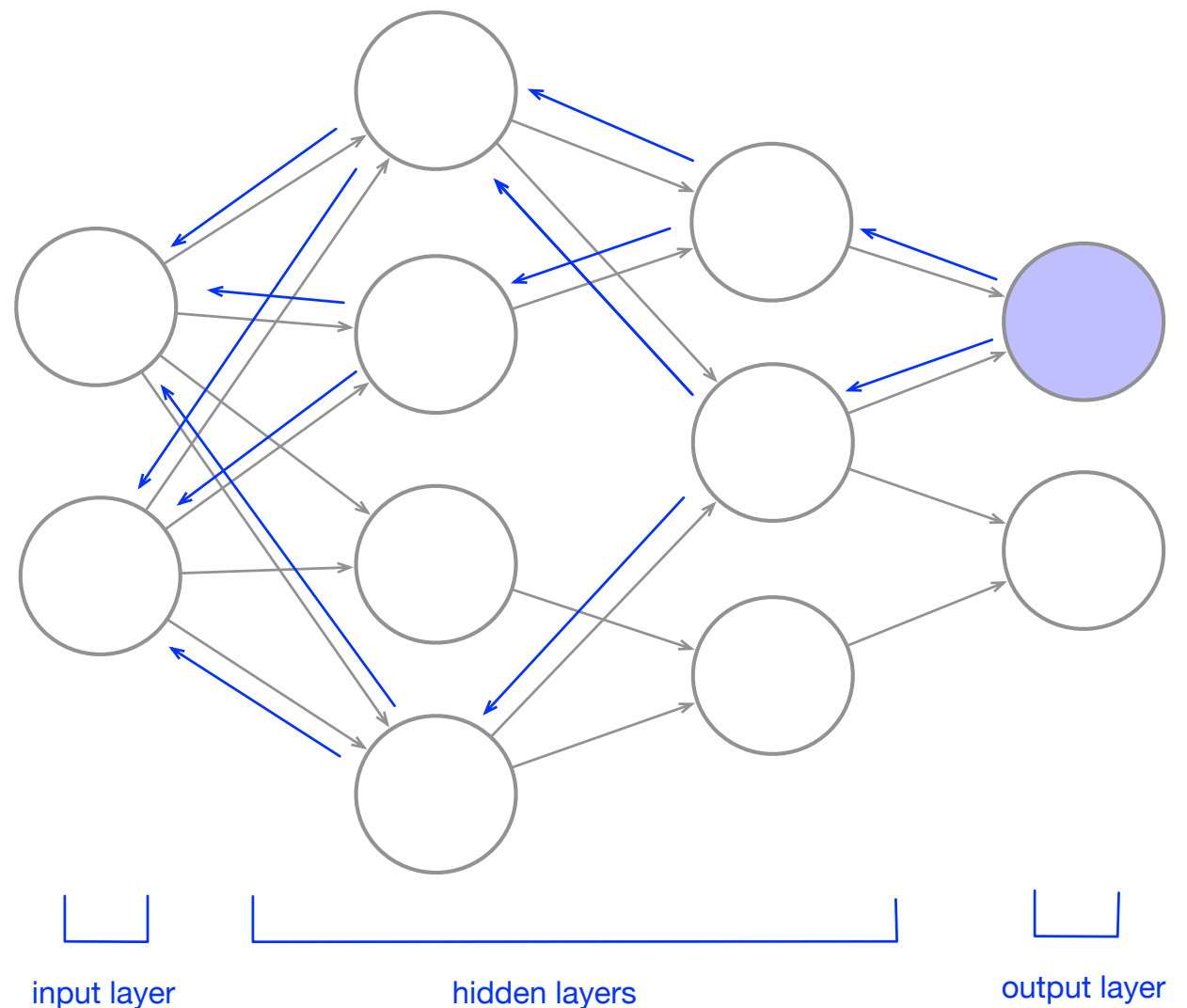


We can see in Figure a, we have used weights to forward the signal to the output layer, this process is called the “forward step”.

Next, after the output error has been identified, a neural network uses the weight to “propagate” back the signal from the output layer to the input layer. This process is called “Backpropagation” or “Backprop”.

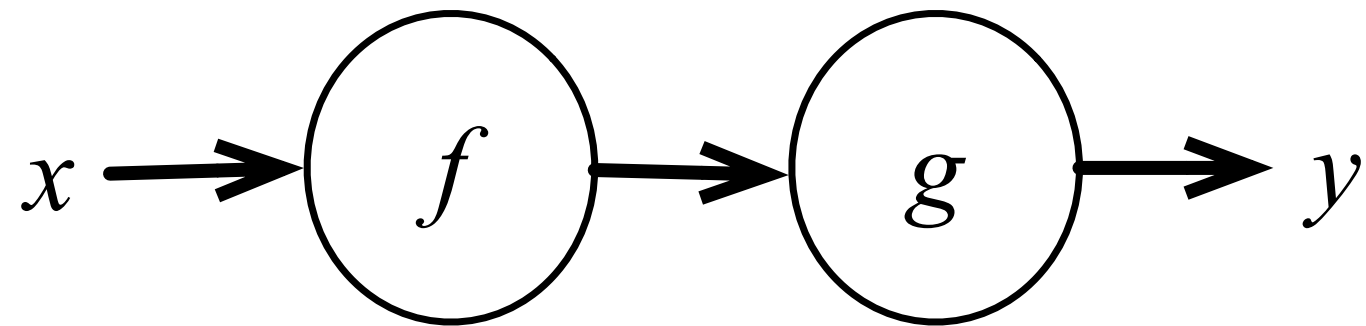
To summarize, the Backprop algorithm splits the error of output neurons across the previous neurons proportional to the incoming weights to this neuron.

- The Backpropagation algorithm assumes the *error in a neuron (hidden or output) is the sum of the splits errors in all other previous nodes linked to this neuron.*
- For example, the blue lines in this Figure visualize the propagation of the error from the blue output neuron to the first layer neurons.
- If there is more than one hidden layer, *the errors of hidden layers split again proportional across all previous links between input and hidden layers connected to that particular neuron.*





# Partial Derivative



$y = g(f(x))$  and by using the chain rule, we can present the partial derivative of  $y$  over  $x$ , as follows:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} \times \frac{\partial g}{\partial f} \times \frac{\partial f}{\partial x}$$

# Implementing Backprop with Matrix Multiplication

- To implement Backpropagation, we can use matrix multiplication. Therefore, the errors of layer  $n$  could be written as a matrix ( $e_n$ ), which is the multiplication of transpose of weight matrix ( $w^T$ ) time matrix of errors for the next layer ( $e_{n+1}$ ), as follows:  $e_n = w_{n+1}^T \cdot e_{n+1}$

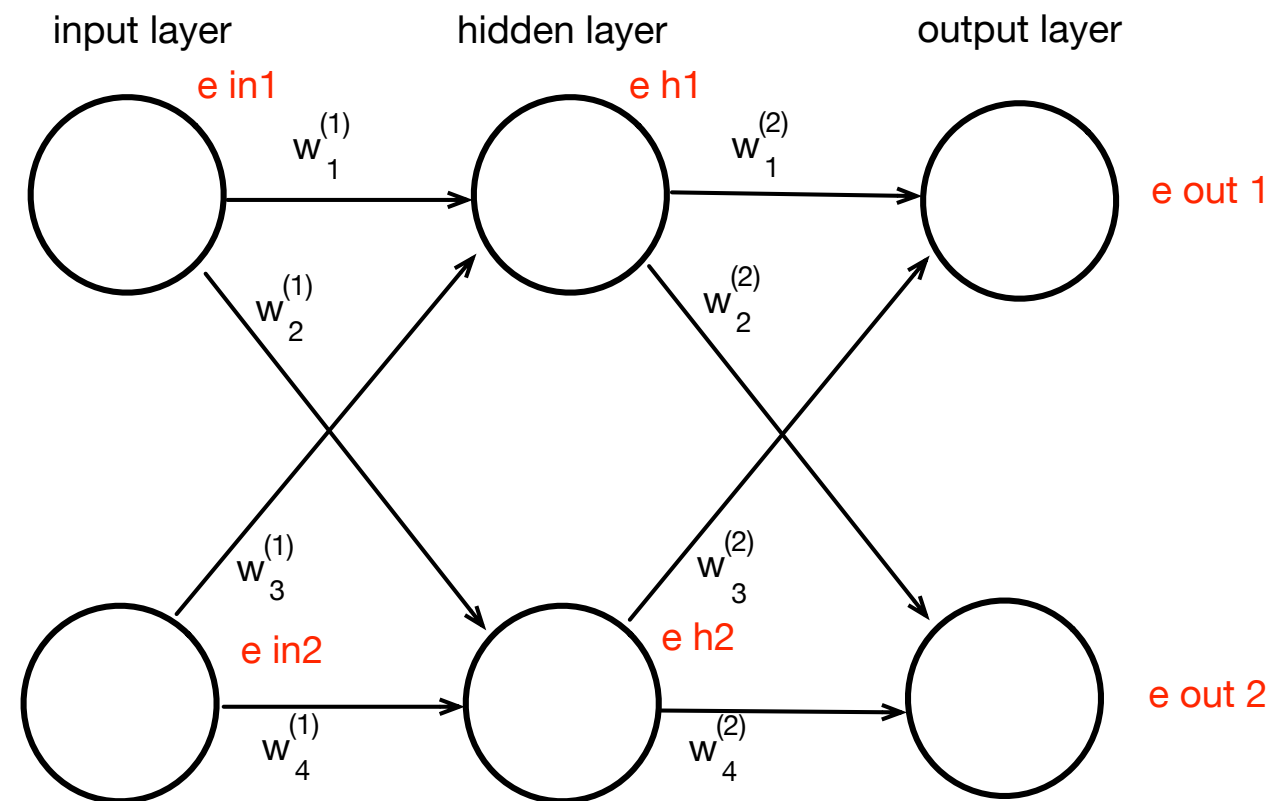
# Implementing Backprop with Matrix Multiplication

Output layer errors ( $e_{out_1}$ ,  $e_{out_2}$ ) are known at the end of each epoch, and the errors of the hidden layer ( $e_h$ ) could be calculated as follows:

$$e_h = \begin{bmatrix} e_{h1} \\ e_{h2} \end{bmatrix} = \begin{bmatrix} w_1^{(2)} & w_2^{(2)} \\ w_3^{(2)} & w_4^{(2)} \end{bmatrix} \times \begin{bmatrix} e_{out_1} \\ e_{out_2} \end{bmatrix}$$

Respectively, errors of the input ( $e_{in}$ ) layer will be calculated as:

$$e_{in} = \begin{bmatrix} e_{in1} \\ e_{in2} \end{bmatrix} = \begin{bmatrix} w_1^{(1)} & w_2^{(1)} \\ w_3^{(1)} & w_4^{(1)} \end{bmatrix} \times \begin{bmatrix} e_{h1} \\ e_{h2} \end{bmatrix}$$



# Formalizing Backprop

- The output of the neuron in the first layer, can be written as  $z = wx + b$ ,  $x$  is the input, but in the other neurons.
- After the input layer, we do not have  $x$ , instead of input neuron data, we only have the output of the activation function. Therefore, we refer to the neuron output as  $z$ . In other words,  $z^{(l)}$  is defined by weight and biases at level  $l$  for the given input that comes from the previous layer.
- We can generalize it (removing layer and node information) and write it as  $a = \sigma(z)$ , here  $z$  presents the output of the previous layer,  $\sigma$  presents the activation function, and thus we can say  $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$ . For example, if we refer to the very last layer of the network as  $L$ , and for the very last layer we have  $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$ , we know that  $a^{(L)} = \sigma(z^{(L)})$ .

# Formalizing Backprop

- We have explained that  $error = predicted - actual$ . Assuming the actual value is presented as  $y_j$ , and the error is presented as  $E$  we can write the following equation for an error of neuron  $j$  and last layer (output layer)  $L$ :  $E_j^{(L)} = a_j^{(L)} - y_j$ . We consider them as a matrix (generalizing it), and thus remove the  $j$ th parameter, and end up with  $E^{(L)} = a^{(L)} - y$  to calculate the error matrix in the last layer  $L$ .

# Formalizing Backprop

- We should understand how much loss score (or error) changes as weight and biases change? Or how sensitive is the loss score to changes in  $w$  and  $b$ ?
- We can write the error equation as a partial derivative (check Chapter 8 to recall partial derivative) of the error to the weight at layer  $l$ , i.e.,  $\frac{\partial E}{\partial w^{(l)}}$ .
- In other words, we are calculating the error with respect to the weights at layer  $l$ .
- Based on the mathematical notation we have described above, and by using the “chain rule” of derivative (check Chapter 8), we can rewrite  $\frac{\partial E}{\partial w^{(l)}}$  with the partial derivative as follows:

$$\frac{\partial E}{\partial w^{(l)}} = \frac{\partial E}{\partial a^{(l)}} \times \frac{\partial a^{(l)}}{\partial z^{(l)}} \times \frac{\partial z^{(l)}}{\partial w^{(l)}}$$

# Formalizing Backprop

$$\frac{\partial E}{\partial w^{(l)}} = \frac{\partial E}{\partial a^{(l)}} \times \frac{\partial a^{(l)}}{\partial z^{(l)}} \times \frac{\partial z^{(l)}}{\partial w^{(l)}}$$

- It is just applying chain rules and simply using  $z$  and  $a$  to find how sensitive is the loss score ( $E$ ) to changes in weight ( $w$ ).
- For biases, we can use the same equation as follows:

$$\frac{\partial E}{\partial b^{(l)}} = \frac{\partial E}{\partial a^{(l)}} \times \frac{\partial a^{(l)}}{\partial z^{(l)}} \times \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

#-----Step 1 (initialing parameters)-----

*initialize  $w, b, \alpha$  & stop\_criteria #*

*for  $i = 1$  to  $m$  {*

#-----Step 2 (forward propagation)-----

*for  $j = 1$  to  $L$  {*

*if ( $j=1$ ) then  $a^{(1)} = x^{(i)}$  else  $a^{(j)} = \sigma(z^j)$*

*}*

# -----Step 3 (Calculate error vectors)-----

$E^{(L)} = a^{(L)} - y^{(i)}$

*for  $k = (L-1)$  to  $2$  {*

$E^{(k)} = (w^{(k+1)})^T \times E^{k+1} \odot \sigma'(z^{(k)})$  #  $\odot$  presents a Hadamard product of two matrices.

*}*

#----- Step 4 (compute gradients and update weight and biases) -----

*for all ( $w$  and  $b$ ) {*

$$w_{new}^{(l)} = w_{old}^{(l)} - \alpha \frac{\partial E}{\partial w_{old}^{(l)}}$$

$$b_{new}^{(l)} = b_{old}^{(l)} - \alpha \frac{\partial E}{\partial b_{old}^{(l)}}$$

*}*


*}*




# Operation used for Tensors

$$A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}, \quad B = \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix}$$


$$A * B = \begin{bmatrix} ag & bh & ci \\ aj & bk & cl \\ dg & eh & fi \\ dj & ek & fl \end{bmatrix}$$

 **Khatri-rao**


$$A \otimes B = \begin{bmatrix} a \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} & b \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} & c \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} \\ d \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} & e \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} & f \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} \end{bmatrix}$$

 **Kronecker**

$$A \odot B = \begin{bmatrix} ag & bh & ci \\ dj & ek & fl \end{bmatrix}$$

 **Hadamard**

$$A \cdot B^T = \dots$$

 **dot product**

#-----Step 1 (initialing parameters)-----

# This step initialize learning rate, weights and biases, and also the threshold for stopping criteria will be specifies.

initialize  $w, b, \alpha$  &  $stop\_criteria$  #  $\alpha$  is learning rate

for  $i = 1$  to  $m$  { #  $m$  is the number of data points in the training set

#-----Step 2 (forward propagation)-----

This step computes the activation for all layers.

$a$  is the predicted value extracted from the activation function and  $x$  is the input variable.

for  $j = 1$  to  $L$  { #  $L$  is the number of layers

if ( $j=1$ ) then  $a^{(1)} = x^{(i)}$  # Since the activation function does not exist at the first layer, which is the input layer, at this layer  $z = wx + b$  and  $a^{(1)}$  is the input.

else  $a^{(j)} = \sigma(z^j)$  # Now there is no  $x$  available (because we are talking about layer 2 and other layers) for each layer, the algorithm computes  $z$  and  $a$ , recall that  $z^l = w^l a^{l-1} + b^l$

}

# -----Step 3 (Calculate error vectors)-----

$E^{(L)} = a^{(L)} - y^{(i)}$  # in the last layer, we know  $y$ , which is the actual output.

for  $k = (L-1)$  to  $2$  {# this loop computes other error vectors ( $E^{(L-1)}, E^{(L-2)}, \dots, E^{(2)}$ ) for other layers (except the last one) until it reaches layer 2, there is no error for layer 1, because the input layer does not have error.

$E^{(k)} = (w^{(k+1)})^T \times E^{k+1} \odot \sigma'(z^{(k)})$  #  $\odot$  presents a Hadamard product of two matrices.  $(w^{(k+1)})^T$  is the transpose of weights for the next layer (to prepare the for matrix operation, they are transposed).  $\sigma'(z^{(k)})$  is a vector of derivatives of activation function results at layer  $k$ .  $\odot \sigma'(z^{(k)})$  moves the error backward through the activation function in layer  $k$ .

}

#----- Step 4 (compute gradients and update weight and biases) -----

# This step computes partial derivate of the gradient with respect to weight and biases.

for all ( $w$  and  $b$ ) {

$w_{new}^{(l)} = w_{old}^{(l)} - \alpha \frac{\partial E}{\partial w_{old}^{(l)}} \# \frac{\partial E}{\partial w_{old}^{(l)}}$  is the partial gradient of the cost function result (loss score) with respect to  $w$ . At the first run, we assume it is 0. This gradient will be acquired using the chained rule of partial derivative.

$b_{new}^{(l)} = b_{old}^{(l)} - \alpha \frac{\partial E}{\partial b_{old}^{(l)}} \# \frac{\partial E}{\partial b_{old}^{(l)}}$  is a partial gradient of the cost function result (loss score) result with respect to  $b$ .

At the first run we assume it is 0.

}

} # closes 'for  $i = 1$  to  $m$ ' loop

# Forward and Backward pass example with mathematics

Consider a forward pass starting from  $x_0$  as input and three layers with activation function  $\sigma_R$  we can formalize the forward pass  $f$  follows,  $f(x_0; w) = \sigma_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$ .

To perform the backward pass and compute the gradients, we need to use the chain rule of differentiation. We will start by computing the gradient of the loss with respect to the output of the network:

$\nabla f = \nabla \sigma_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$ . where  $\nabla$  denotes the gradient and  $\sigma_R$  is the activation function.

Next, we apply the chain rule to compute the gradient of the loss with respect to the parameters of the last layer:

$$\nabla w_3 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2)$$

$$\nabla b_3 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$$

Here  $\sigma'_R$  is the derivative of the activation function. Next, we use the chain rule again to compute the gradient of the loss with respect to the parameters of the second layer:

$$\nabla w_2 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \cdot w_3 \cdot \sigma'_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) \cdot \sigma_R(w_1 x_0 + b_1)$$

$$\nabla b_2 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \cdot w_3 \cdot \sigma'_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2)$$

Finally, we compute the gradient of the loss with respect to the parameters of the first layer:

$$\nabla w_1 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \cdot w_3 \cdot \sigma'_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) \cdot w_2 \cdot \sigma'_R(w_1 x_0 + b_1) \cdot x_0$$

$$\nabla b_1 = (\nabla f) \cdot \sigma'_R(w_3 \cdot \sigma_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \cdot w_3 \cdot \sigma'_R(w_2 \cdot \sigma_R(w_1 x_0 + b_1) + b_2) \cdot w_2 \cdot \sigma'_R(w_1 x_0 + b_1)$$

# Take some break

- <https://playground.tensorflow.org>

# Outline

- The rationale behind Deep Learning
- Artificial Neural Network and its Concepts
- Perceptron and Multilayer Perceptron
- Activation Functions
- Cost Functions and Neural Network Optimizers
- Backpropagation
- **Regularization in Neural Network**

# Regularization Methods

- Vanishing/Exploding Gradient Problem
- Weight initialization
- Gradient Clipping
- Batch Normalization
- Drop out
- Early Stopping

# Regularization in Neural Network

- A successful machine learning algorithm should avoid overfitting.
- We can think of overfitting as just memorizing the data (not learning) and matching everything to the training dataset. Therefore, if new data arrives into the system (test data) that does not match the existing data, the algorithm can not determine a label for it.
- Regularizations reduce the overfitting error.

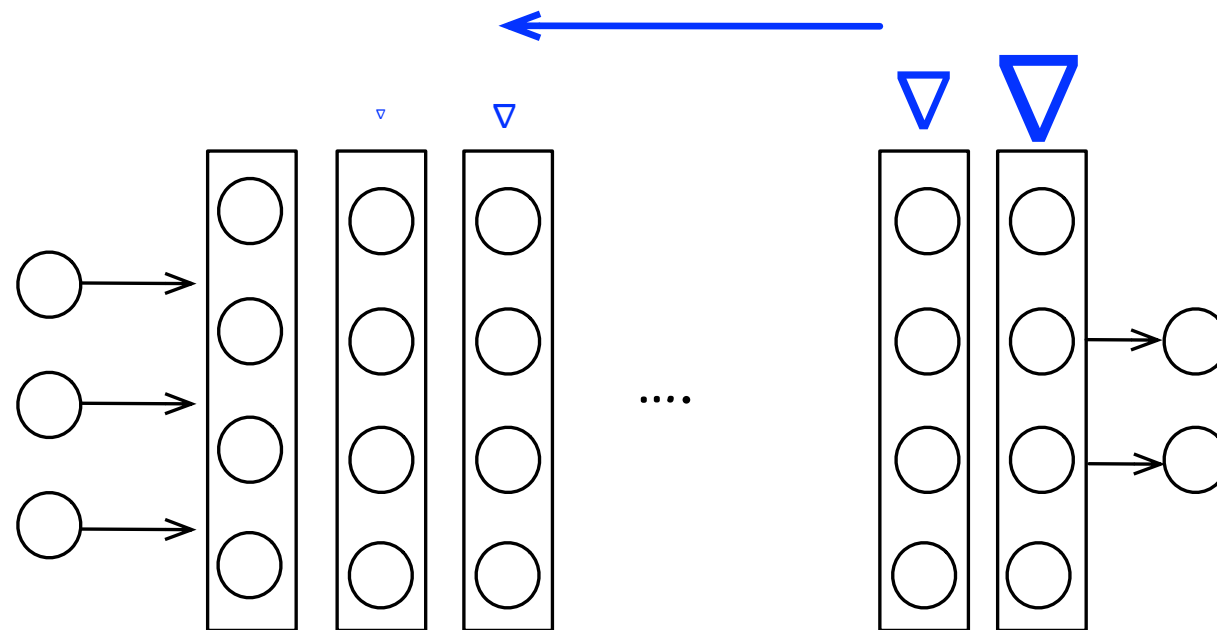
# Vanishing and Exploding Gradients

- We have learned that a neural network operates in three steps.
- First, it begins from the input and it goes to the output and performs a prediction, i.e., a forward pass.
- Second, the loss function compares the prediction result with the ground truth dataset and measures the error. In simple words, the output of a loss function is an error value.
- In the third step, the neural network uses the error value and backpropagation algorithm to calculate the gradient for each neuron in the network. Gradients (partial gradients with respect to weights and biases) are values that are used by the network to adjust its weights and biases to reduce error.



# Vanishing Gradients

- As the network moves from output toward the input layer, the gradient gets smaller by the chain rule, and thus the weight adjustment is getting smaller and smaller.
- In a more technical sense, the gradient is exponentially shrinking as the backpropagation moves toward the input neuron.
- A big gradient reveals a big adjustment, a small gradient reveals a small adjustment on weight.

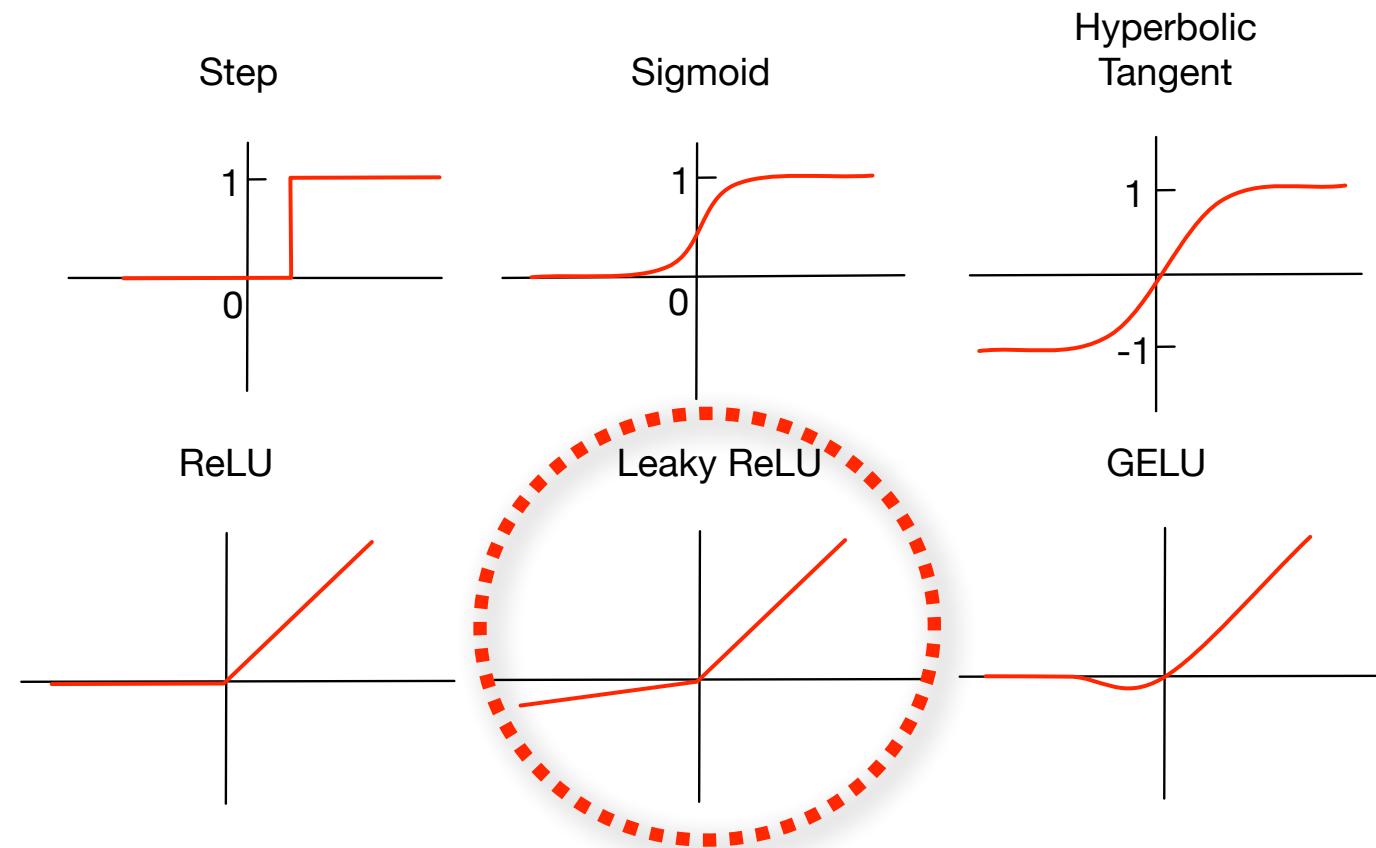


# Vanishing Gradient

- When a network is deep (has many hidden layers), the gradient gets smaller and smaller until it vanishes, and thus weights on the layers close to the input layer never get updated.
- For example, if the gradient of the last layer close to the output layer is going to be 0.5 or less than one, as we go deep in the network, it gets smaller and smaller because it gets multiplied to some smaller number, for example, it will be  $0.5 \times 0.4 \times 0.1 = 0.002$ . Then the weight in the input layer will be  $w_{new}^{(l)} = w_{old}^{(l)} - \alpha \times 0.002$  and we know a learning rate is also a small number such as 0.01, so we will have  $w_{new}^{(l)} = w_{old}^{(l)} - 0.00002$
- Neural Network regularization methods reduce the vanishing gradient and exploding gradients.

# Exploding Gradient

- The same thing could happen with gradients larger than one, and thus it makes a very big gradient that does not let SGD to get close to minima, which is **exploding gradient**.



- There are approaches that do not solve this problem but try to mitigate them.
- One approach is the use of ReLU or Leaky-ReLU as activation functions and avoiding using hyperbolic tangent as activation function, which is prone to the vanishing gradient.
- They are not enough.

# Weight Initialization

- There are two approaches that use a more subtle weight initialization and do not perform the weight initialization at the beginning completely random. These approaches are known as Xavier [Glorot '10] and He (a.k.a Kaiming) [He '15].
- Xavier initialization uses random weights that have a normal distribution, and not completely random weights.
- He initialization (a.k.a Kaiming initialization) proposes a weight initialization for non-linear activation functions such as ReLU and Leaky ReLU while taking into account the non-linearity of non-linear activation functions. Weights are initialized to have a normal distribution, with zero mean and standard deviation of  $\sqrt{2/n^2}$  ( $n$  is the number of weights to configure), and biases are initialized to zero.

# Gradient Clipping

- Gradient Clipping cuts off gradients before they reach a predefined limit. This limitation can be specified by using a transformation and normalizing the range of the gradient.
- For example, we cut off all gradients that are larger than a specific threshold, e.g. 1, or smaller than a specific threshold, e.g. -1. Then do not use gradients outside this range for the backpropagation algorithm to adjust weights.

# Batch Normalization

## (Batch norm)

- If we intend to have a neural network that handles different numerical ranges we should normalize them.
- For example, we would like to have a neural network to predict the happiness of our users based on their age and annual income. The age is a number between 0 to 120, but income is widely varied for example in the U.S. could be from 10,000\$ per year to >10,000,000,000\$. Therefore, we need to bring these variables into the same range.
- In the context of the neural network, this process is called input **Batch Normalization** or **Batch Norm** [Loffe '15]. It brings the scale of all data between 0 and 1.
- The batch norm is applied on a layer basis.

# Batch Norm Algorithm

- (i) Assuming that  $m$  is the batch size of the input, first, mini-batch mean ( $\mu_B$ ) and mini-batch variance ( $\sigma_B^2$ ) for each batch of the input will be calculated.

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

- (ii) Next, the output of an activation function will be substituted with the z-normalized value of it, before passing it to the next layer. z-score and it is written as:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B}$$

- (iii) After the value (output of activation function) of each neuron is normalized, then it multiplies the output by an arbitrary parameter (scale)  $\gamma$  and adds another arbitrary parameter (shift)  $\beta$ , to the result, as follows:

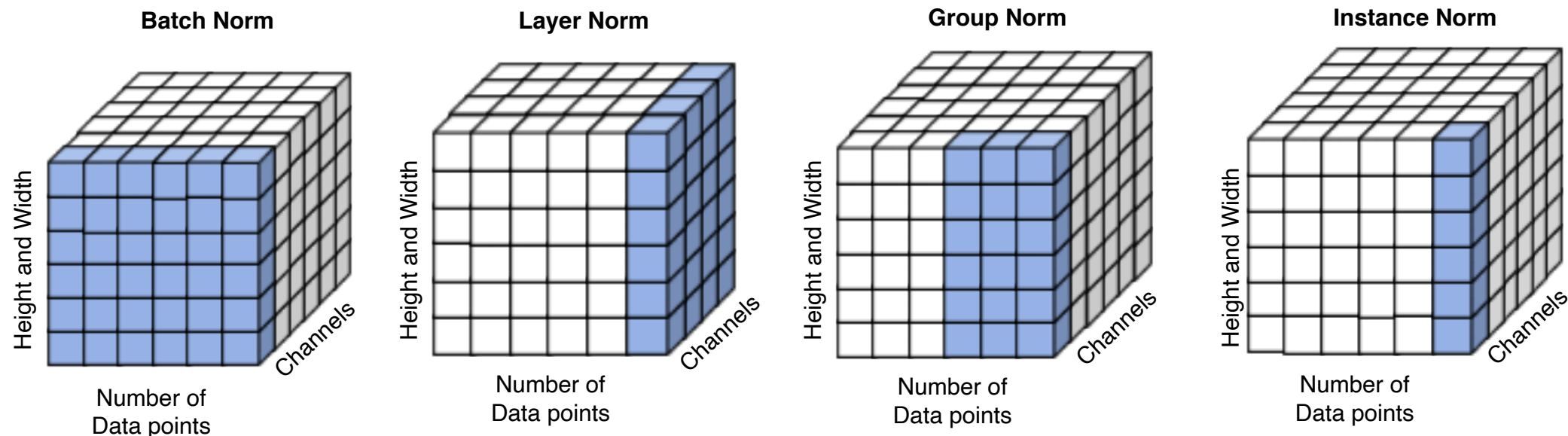
$$y_i = (\hat{x}_i \times \gamma) + \beta$$

Similar to weight and bias parameters, both of these parameters (scale and shift) are getting optimized during the training process, and the optimizer configures them. Therefore, we can write the equation of batch normalization as follows:

$$y_i = \gamma \frac{\hat{x}_i - \mu_B}{\sigma_B} + \beta$$



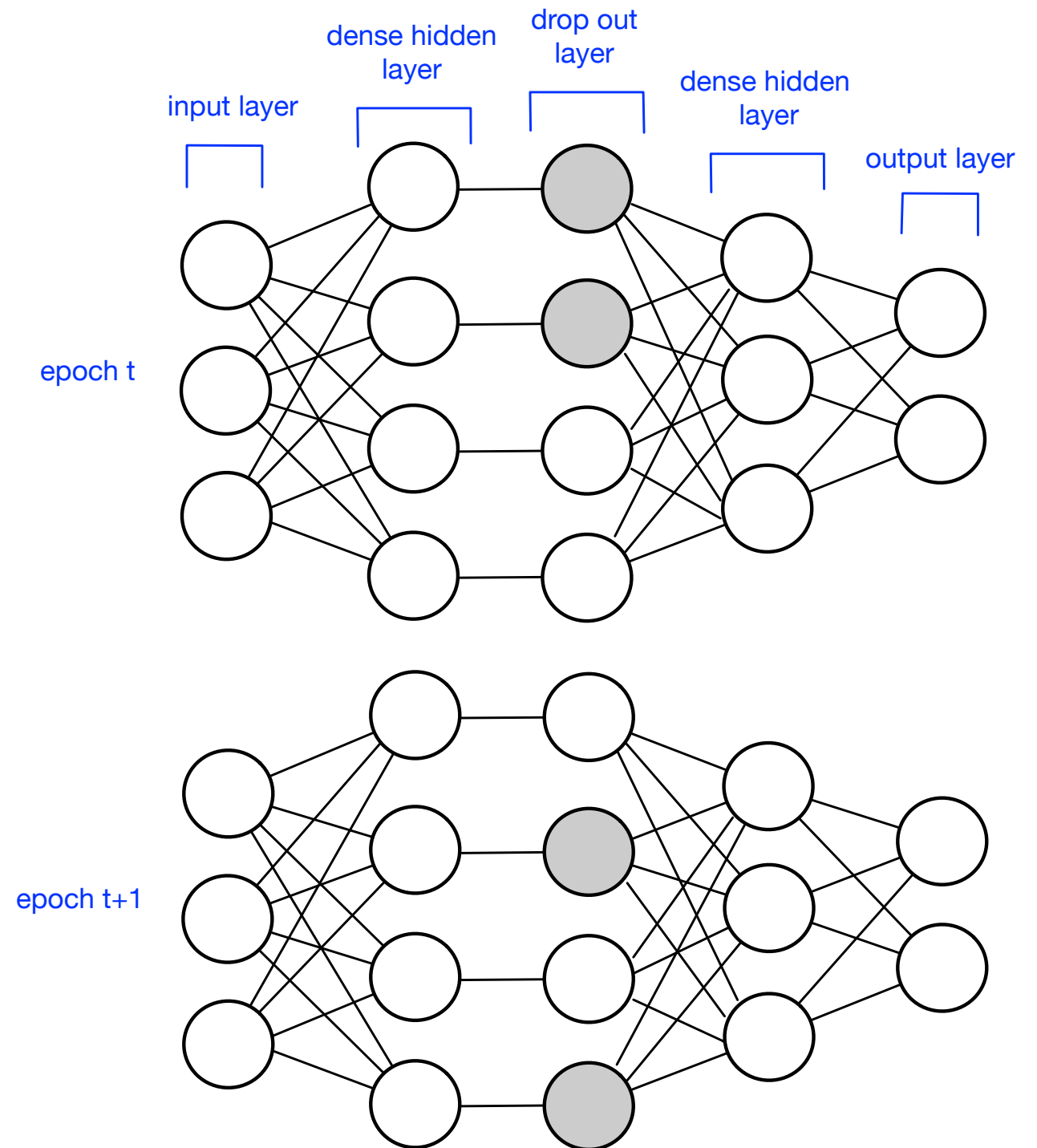
# Other forms of Normalization



- **Instance Normalization** is a type of batch normalization that is applied for a specific instance. For example, if the neural network applies a Batch Norm on a set of images, instance normalization applies the normalization on every single image. In other words, it treats *each input sample separately for normalization*. Therefore, mean and variance are calculated for each channel of an individual sample across both x and y (spatial) dimensions and not for batches of input samples.
- **Layer Normalization** tries to address the Batch Norm's dependency on the batch sizes, which is not possible for RNN networks (we will explain RNN later in this Chapter). To handle this limitation, Layer Normalization [Ba '16] introduces a normalization across channel dimensions *instead of batches*.
- **Group Normalization** tries to address Batch Norm's need for large memory because of a need for a large set of batches [Wu '18]. Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization, which makes it independent of batch sizes.

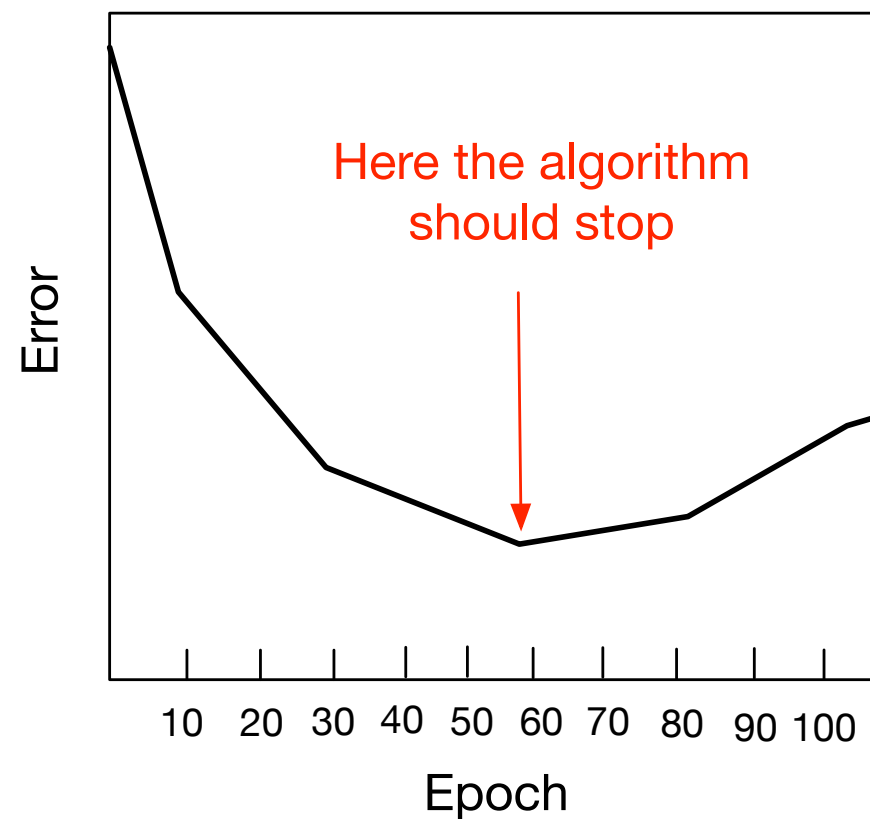
# Dropout

- During the training phase in each epoch, a random number of neurons output will be converted to zero and does not provide any information to the next layer.
- Usually, this process is performed by adding a layer that is called the **dropout** layer.
- Dropout is a very effective approach to prevent overfitting.



# Early Stopping

- An image is worth than a thousand words



# Pytorch Tutorial

[https://pytorch.org/tutorials/beginner/  
deep\\_learning\\_60min\\_blitz.html](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)