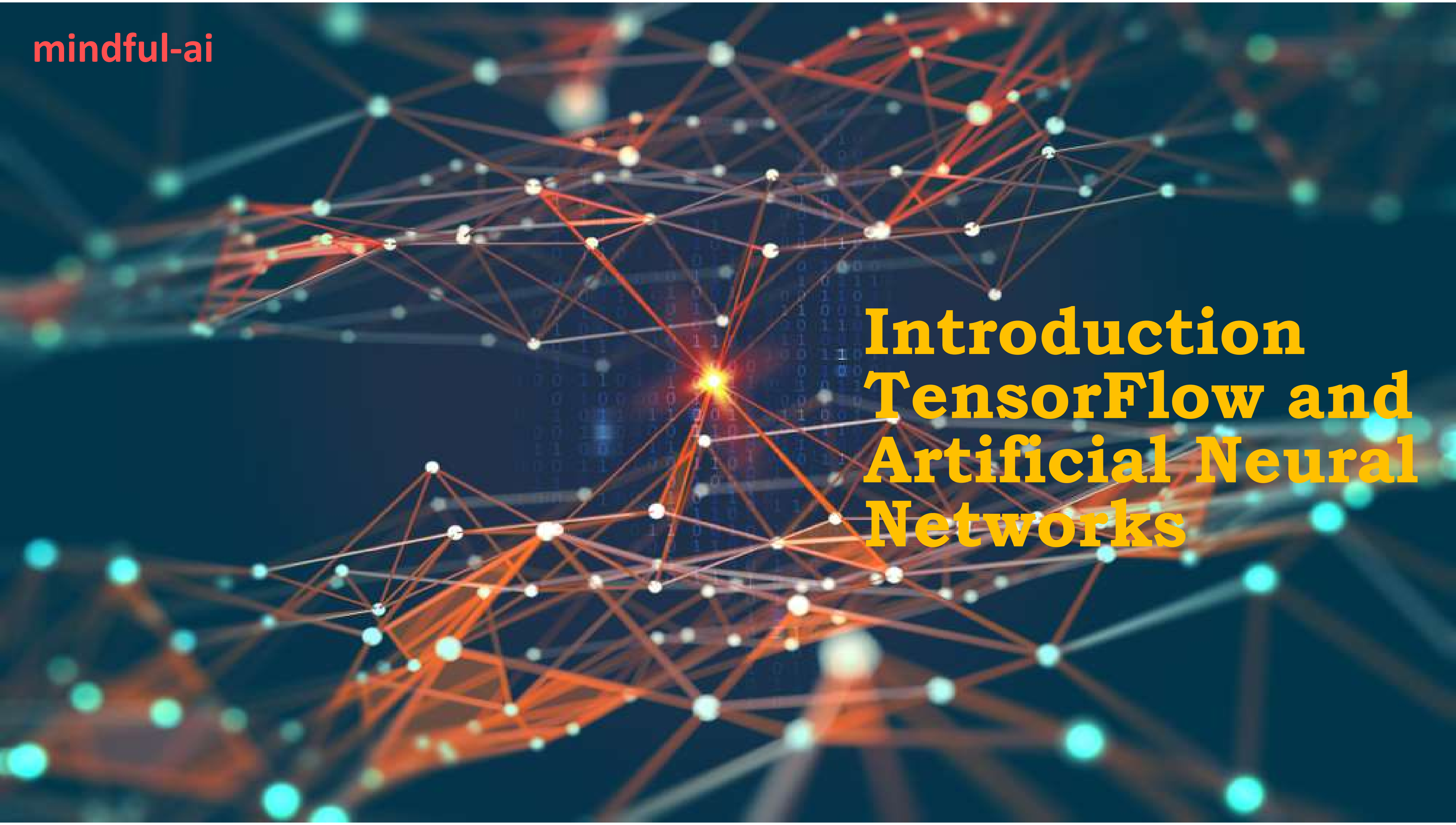
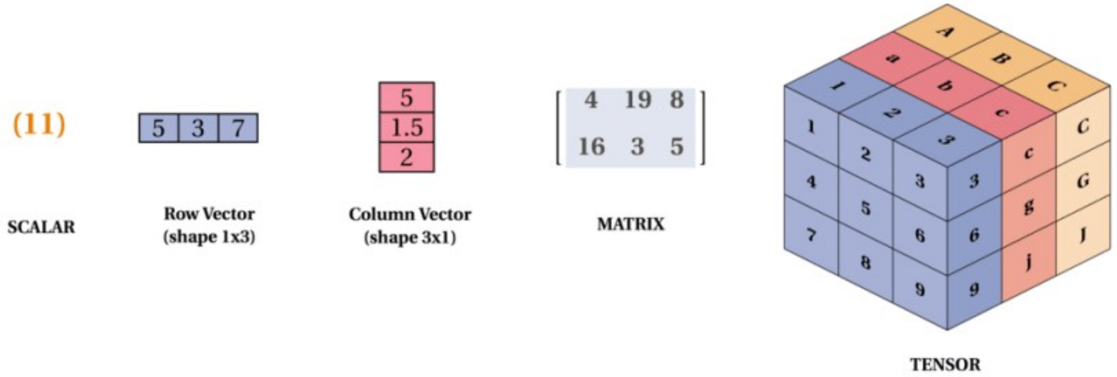
mindful-ai



Introduction TensorFlow and Artificial Neural Networks

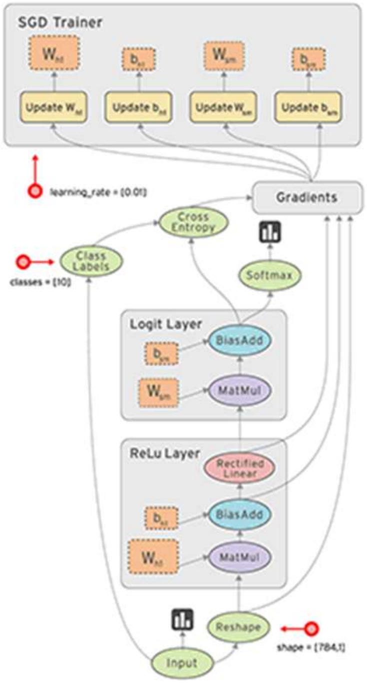
Tensor

- In mathematics, a tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space
- Tensors are the standard way in which data is represented in TensorFlow

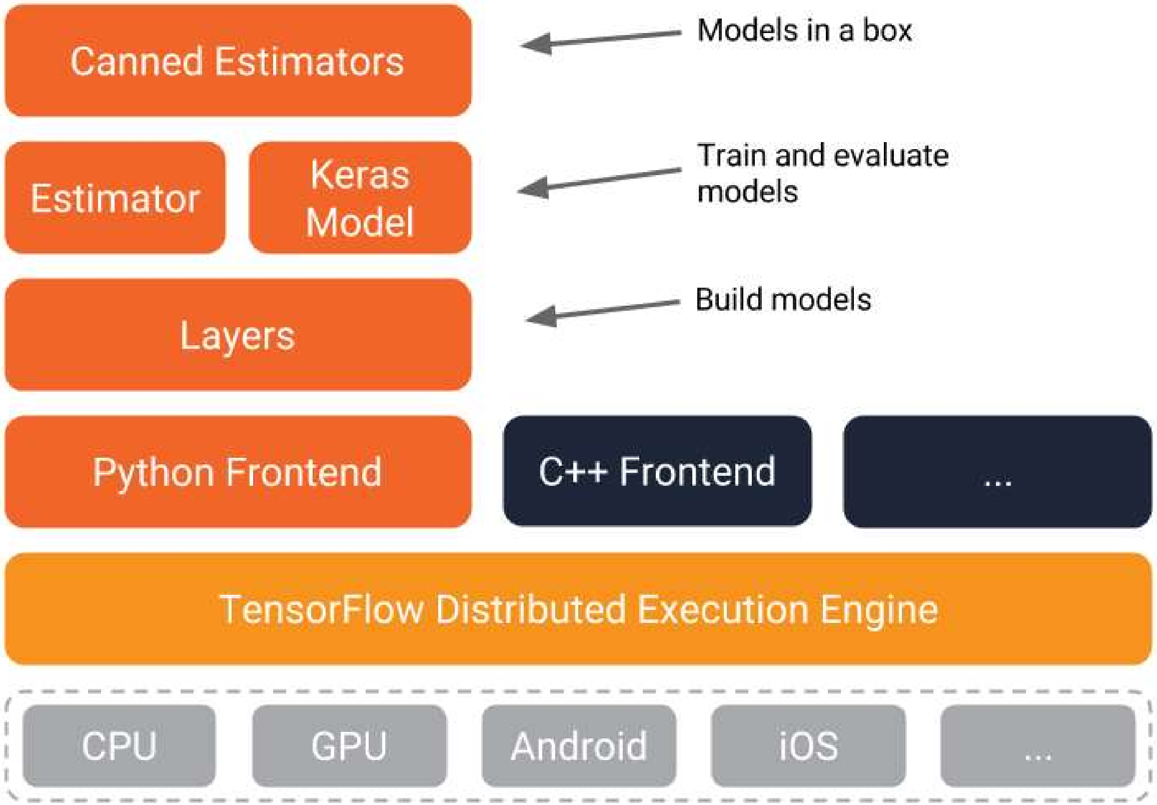


TensorFlow

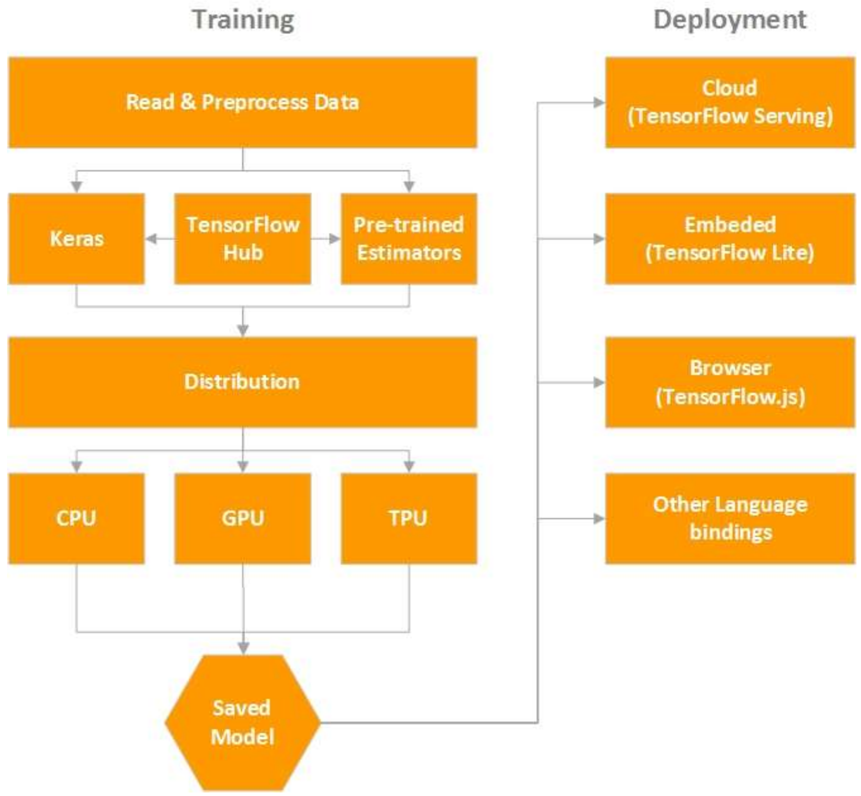
- TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.
- TensorFlow is simply referring to the flow of the Tensors in the computational graph.



TensorFlow Architecture



TensorFlow Architecture



Installation and Basic Practice

- Consult: https://www.tensorflow.org/install
  - **pip install tensorflow** will do the job just fine
- You can configure TensorFlow for GPU and CPU
  - In late 2010, Stanford researchers found that GPU was also very good at matrix operations and algebra so that it makes them very fast for doing these kinds of calculations. Deep learning relies on a lot of matrix multiplication.
- Let’s do some basic practice with TensorFlow
  - Initialization of Tensors
  - Mathematical Operations
  - Indexing
  - Reshaping
  - Placeholders
  - Graphs

Graphs

- TensorFlow makes use of a graph framework. The graph gathers and describes all the series computations done during the training. The graph has lots of advantages:
  - It was done to run on multiple CPUs or GPUs and even mobile operating system
  - The portability of the graph allows to preserve the computations for immediate or later use. The graph can be saved to be executed in the future.
  - All the computations in the graph are done by connecting tensors together
- A tensor has a node and an edge. The node carries the mathematical operation and produces an endpoints outputs. The edges the edges explain the input/output relationships between nodes

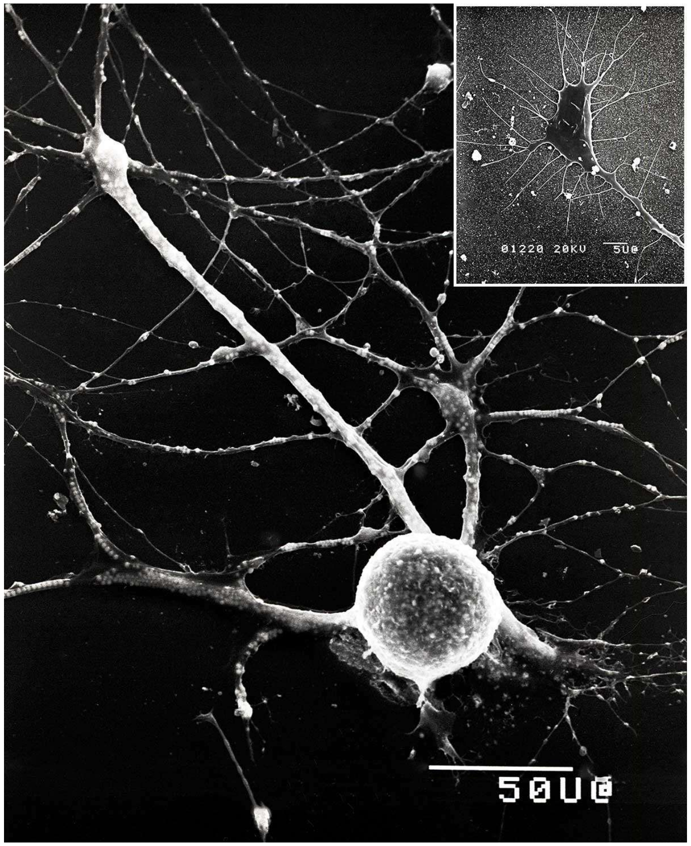
TensorBoard

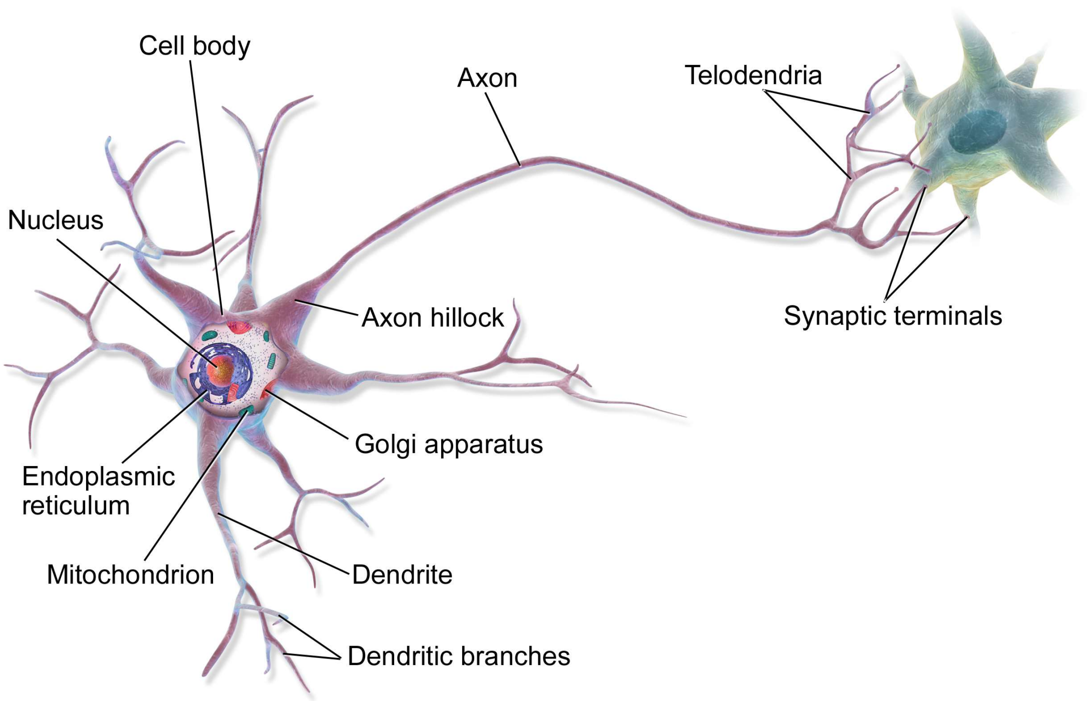
- TensorBoard is a visualization software that comes with any standard TensorFlow installation
- In Google’s words: “The computations you'll use TensorFlow for (like training a massive deep neural network) can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard.”
- It is generally used for two main purposes:
  - Visualizing the Graph
  - Writing Summaries to Visualize Learning

Sessions

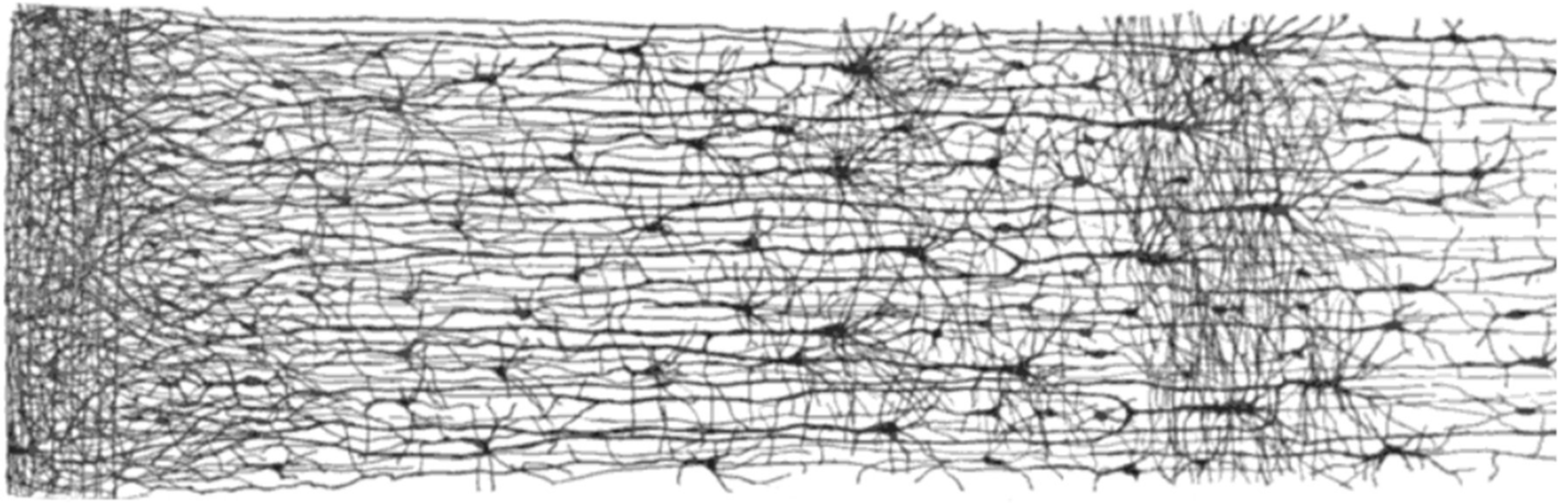
- To compute anything, a graph must be launched in a session. Technically, session places the graph ops on hardware such as CPUs or GPUs and provides methods to execute them.

Biological Neuron





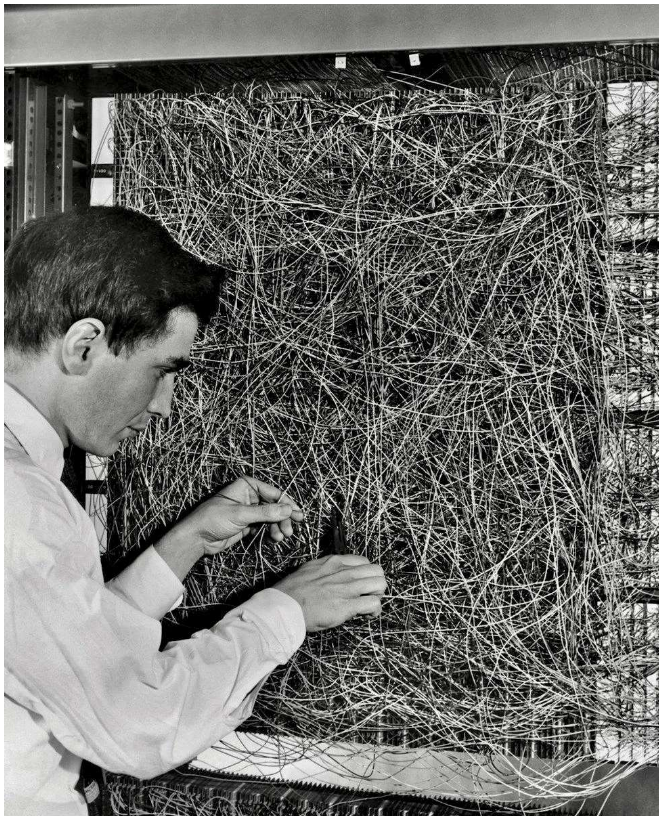
Multiple Layers of Neural Network



Multiple Layers in a biological neural network
in the human cortex

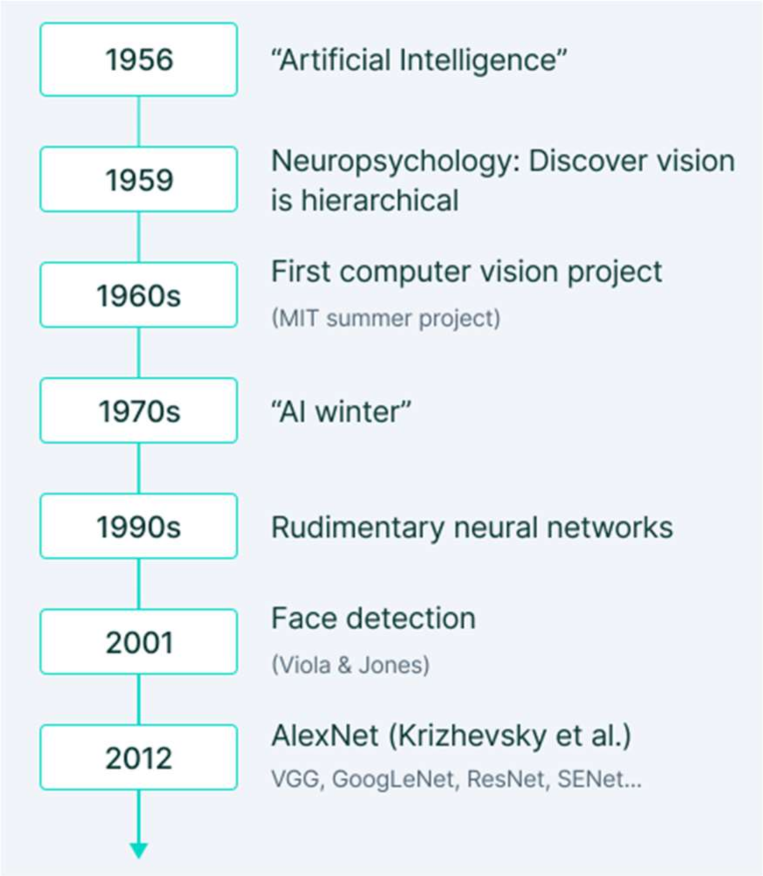
Perceptron

- A perceptron was a form of neuron/neural network introduced in 1958 by Frank Rosenblatt
- Rosenblatt's book **Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms**, summarized his work on perceptrons at the time
- Amazingly, even back then, he had forseen a huge potential:
  - “… perceptron may eventually be able to learn, make decisions and translate languages.”



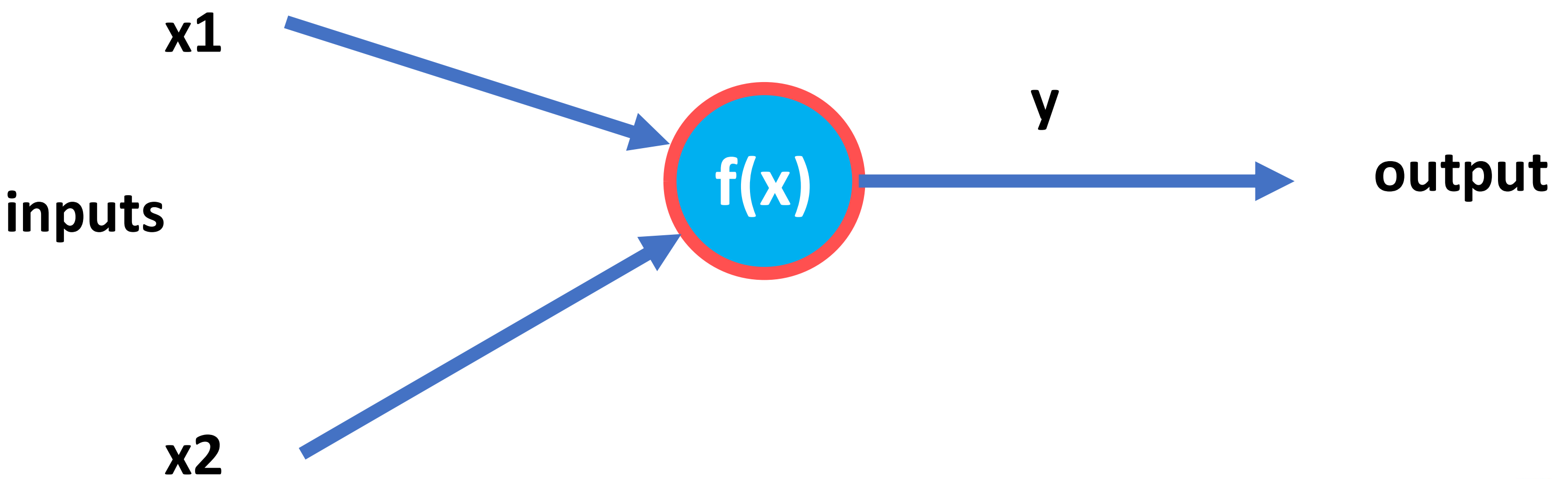
Perceptron

- In 1969, Marvin Minsky and Seymour Papert published their book Perceptrons
- It suggested that there were severe limitations to what perceptrons could do
- This marked the beginning of what is known as AI Winter, with little funding into AI and Neural Networks in the 1970s
- Fortunately for us, we now know the amazing power of neural networks, which all stem from the simple perceptron model.
- Let’s now see how a simple biological neuron can be modelled into perceptron



1956	“Artificial Intelligence”
1959	Neuropsychology: Discover vision is hierarchical
1960s	First computer vision project (MIT summer project)
1970s	“AI winter”
1990s	Rudimentary neural networks
2001	Face detection (Viola & Jones)
2012	AlexNet (Krizhevsky et al.) VGG, GoogLeNet, ResNet, SENet...

Perceptron: Building a Model

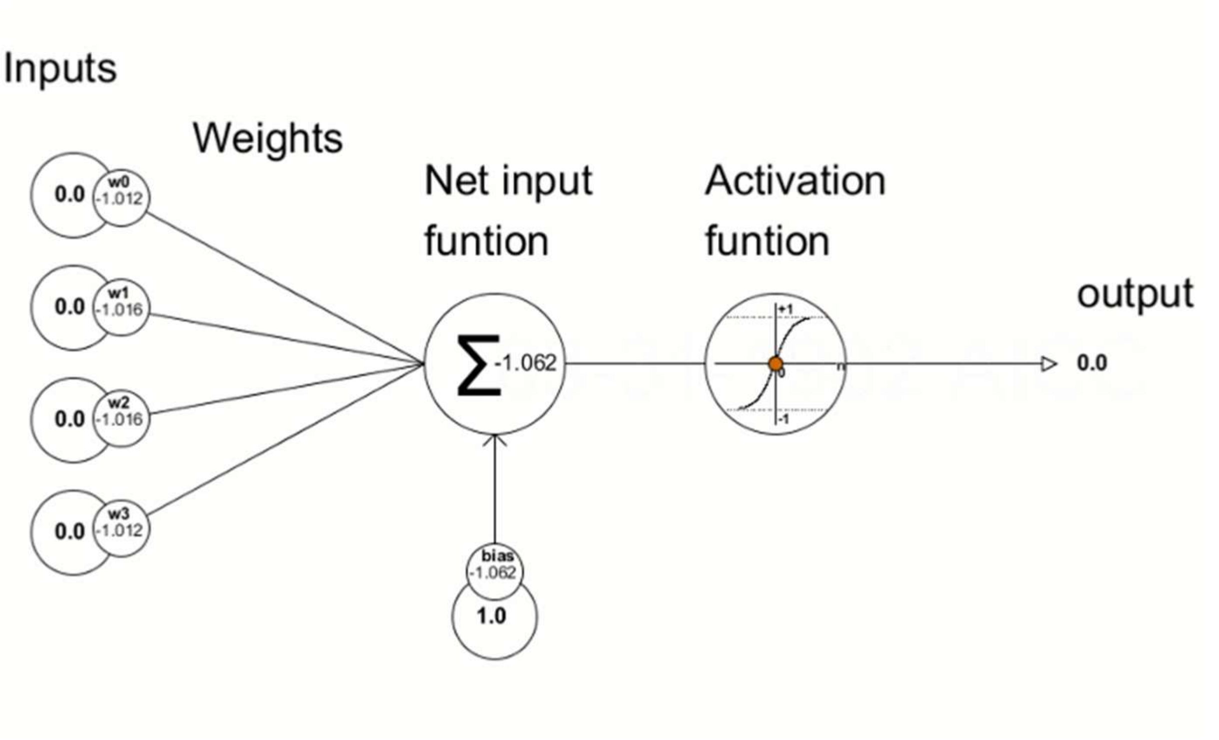


$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

Bias

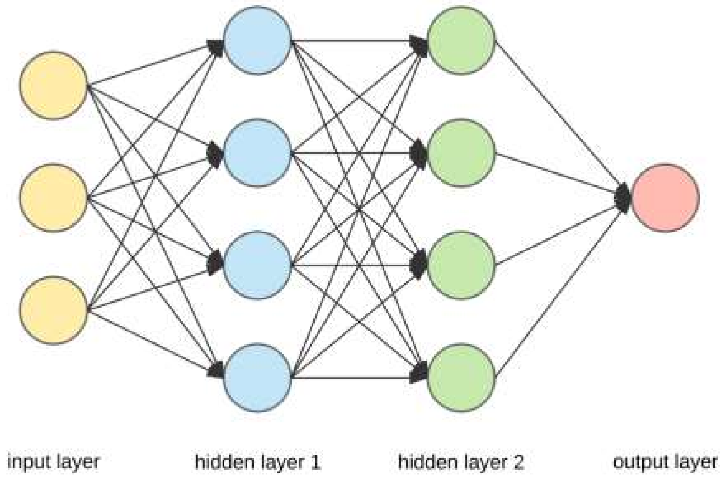
- The “bias” is a threshold that the neuron (perceptron) has placed on the inputs
- The input x*w should cross the bias to be effectively used by the neuron
- Suppose b = -10, x*w should cross 10 to be useful

Working


Inputs
Weights
Net input
funtion
Activation
funtion
output
0.0
w0
-1.012
0.0
w1
-1.016
0.0
w2
-1.016
0.0
w3
-1.012
Σ -1.062
+1
0
-1
0.0
bias
-1.062
1.0

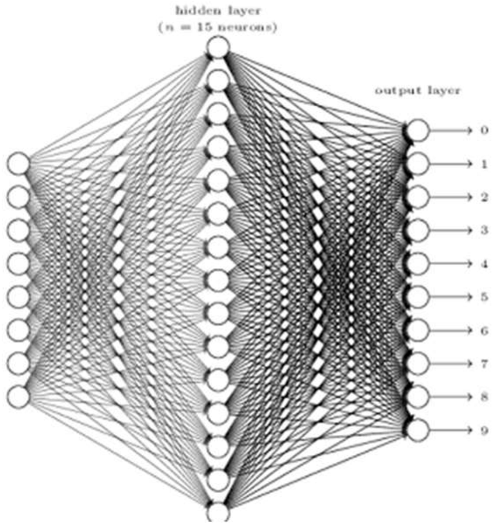
Neural Networks

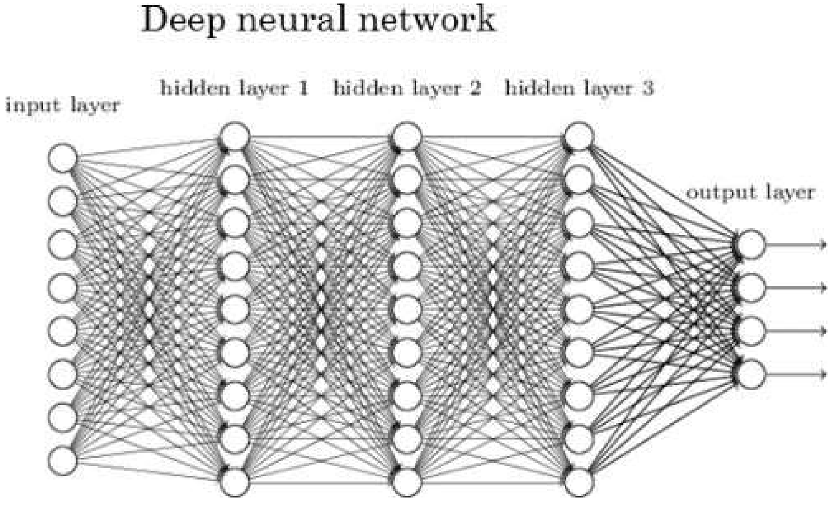
- A single perceptron will not be useful for learning anything significant. Therefore we build a network of perceptrons interconnected in layers.
- It is called as the **multi-layer perceptron model**



The hidden layers become more and more like black boxes

Deep Neural Networks





Neural Networks

- What is incredible about the neural network framework is that it can be used to approximate any continuous function
- Zhou Lu and later on Baris Hanin proved mathematically that Neural Networks can approximate any convex continuous function
- Reference: https://en.wikipedia.org/wiki/Universal_approximation_theorem

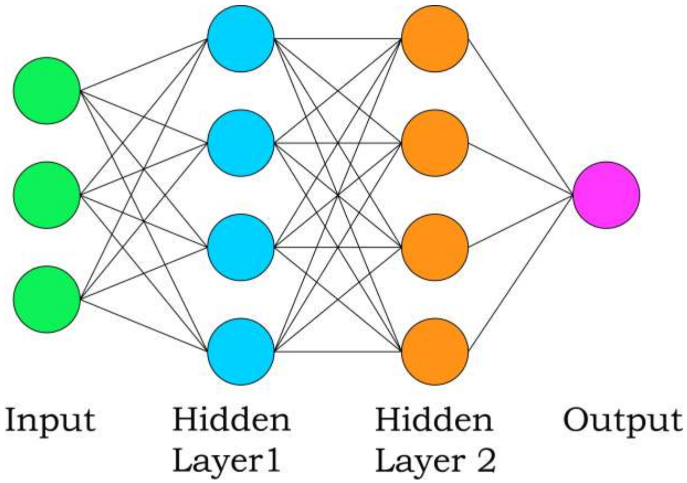
Neural Networks

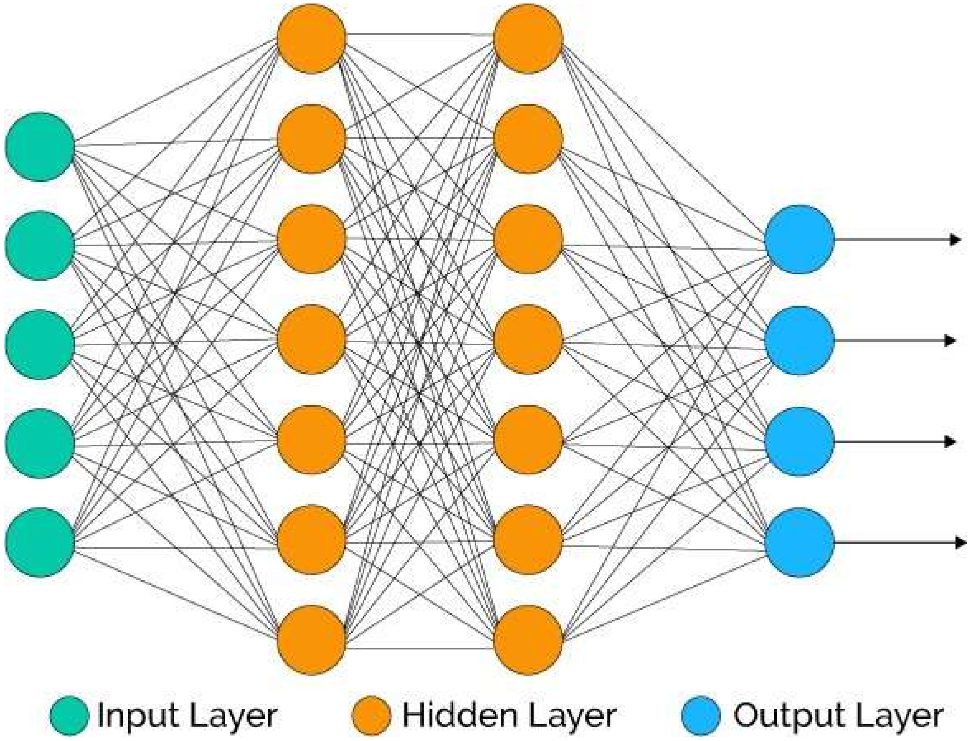
- Previously in out simple model we saw that the perceptron itself contained a very simple summation function f(x)
- For most cases however that won’t be useful, we might want to set constraints to the outputs, especially classification tasks
- In classification tasks, it would be useful to have all the outputs fall between 0 and 1
- These values can then present probability assignments for each class
- We will have to use **activation functions** to set boundaries to output values from the neuron

Multi-class Classification

- Notice all activation functions make sense for a single output, either a continuous label or trying to predict a binary classification.
- But, what should be do if we have a multi-class situation?
- There are two main types of multi-class situation:
    - Non-exclusive classes: a data point can belong to multiple classes
        - Example: photo can have multiple tags such as family, beach, vacation, etc
    - Mutually exclusive classes: a data point belong to only one class
        - Example: photo can belong to grey scale or color, but not both at the same time
- One easy way to organize multiple classes is to simply have one output node per class

Multi-class Classification

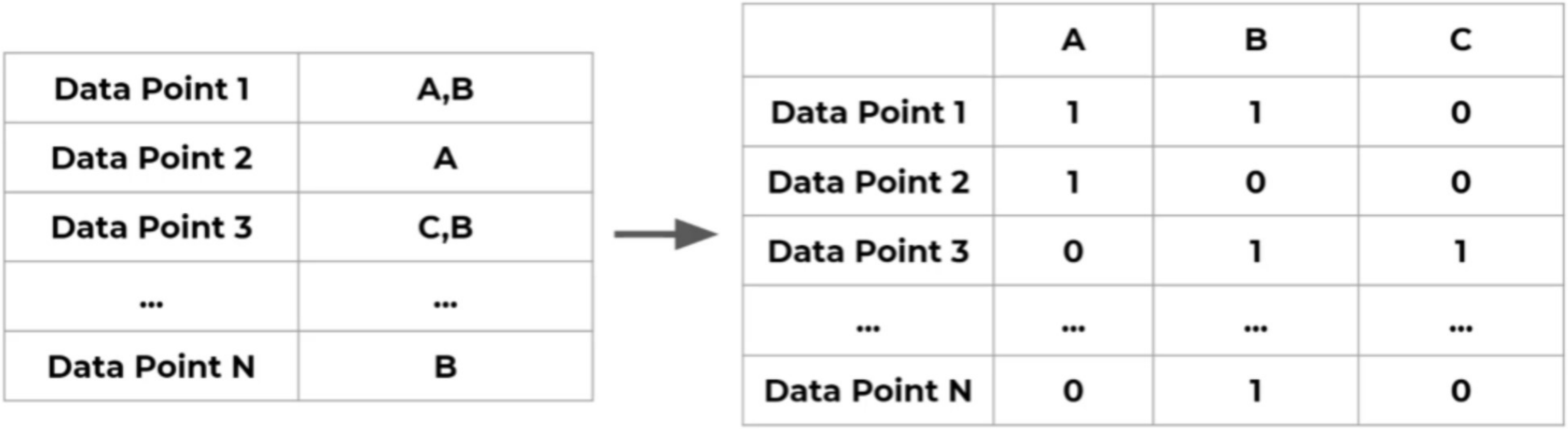




Multi-class Classification

Multi-class Classification

- Non-exclusive classes: data can belon to multiple classes

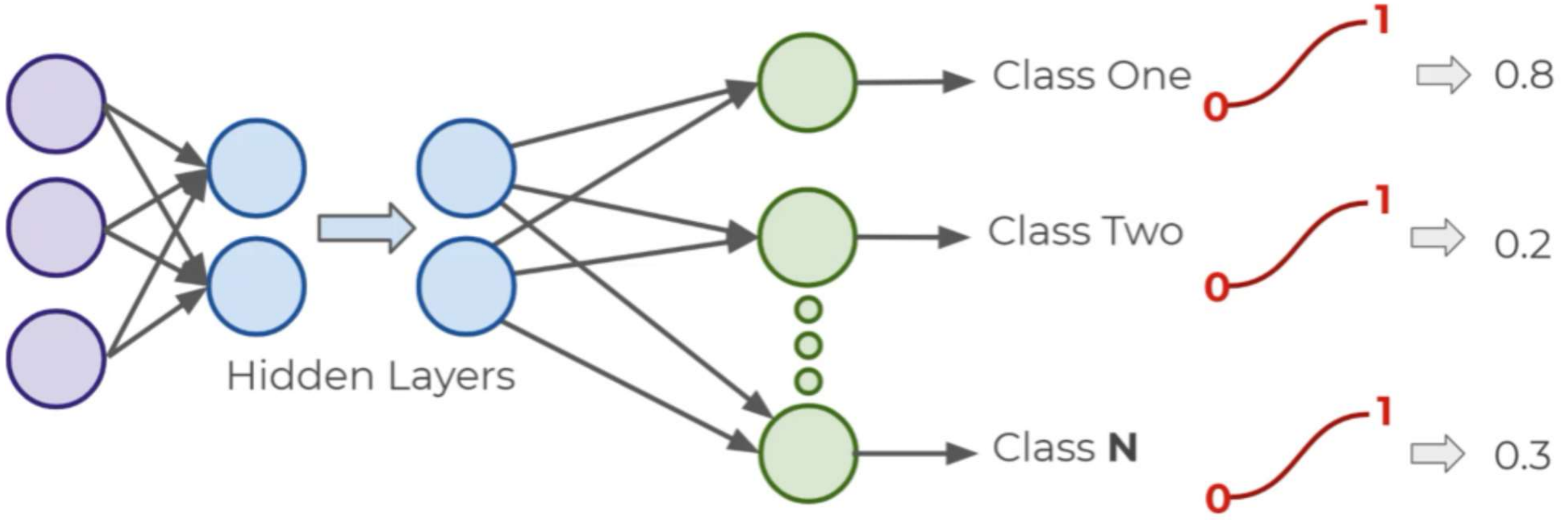


Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

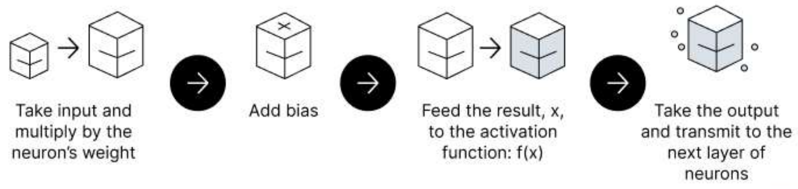
Multi-class Classification

- Sigmoid Function can be used at the output for non-exclusive classes the output of which is a probability measure

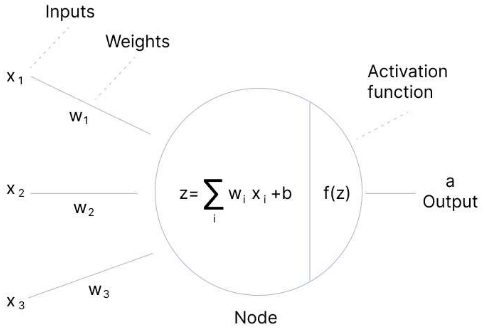


Activation Function

- The activation function of a node defines the output of that node given an input or set of inputs.
- The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.


Take input and multiply by the neuron's weight
Add bias
Feed the result, x, to the activation function: f(x)
Take the output and transmit to the next layer of neurons

V7 Labs

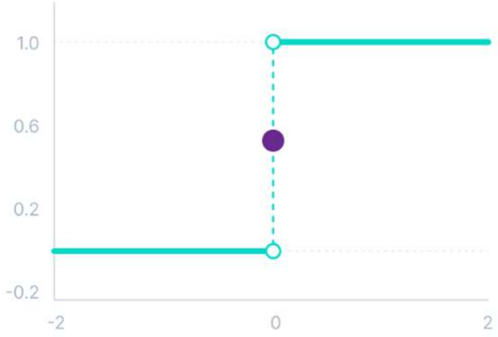


V7 Labs

Binary Step Function

- Binary step function depends on a threshold value that decides whether a neuron should be activated or not.
- Here are some of the limitations of binary step function:
  - It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
  - The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

$$f(x) = \begin{cases} 0 & for\, x < 0 \\ 1 & for\, x \geqslant 0 \end{cases}$$

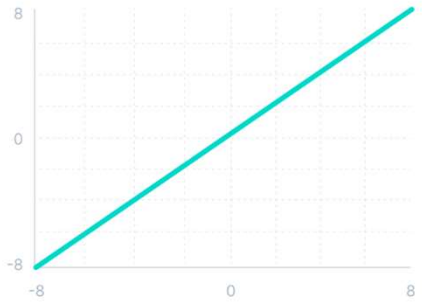


Linear Activation Function

- The linear activation function is also known as Identity Function where the activation is proportional to the input.
- However, a linear activation function has two major problems :
    - It’s not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
    - All layers of the neural network will collapse into one if a linear activation function is used.
    - No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

Linear Activation Function

$$f(x) = x$$



Non-Linear Activation Functions

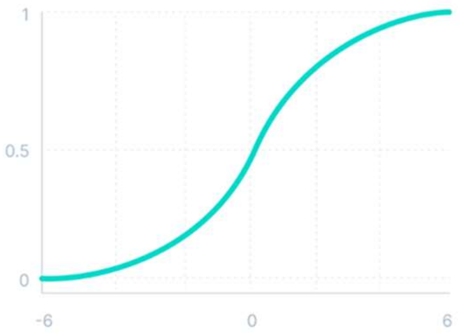
- Non-linear activation functions solve the following limitations of linear activation functions:
  - They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.
  - They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation in a neural network.

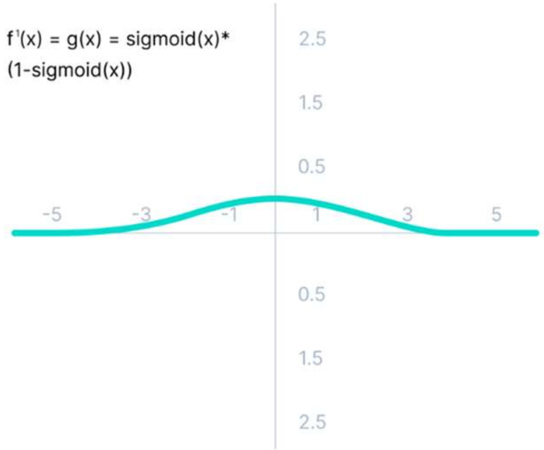
Sigmoid Activation Function

- This function takes any real value as input and outputs values in the range of 0 to 1.
- Here’s why sigmoid/logistic activation function is one of the most widely used functions:
  - It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
  - The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

Sigmoid Activation Function

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





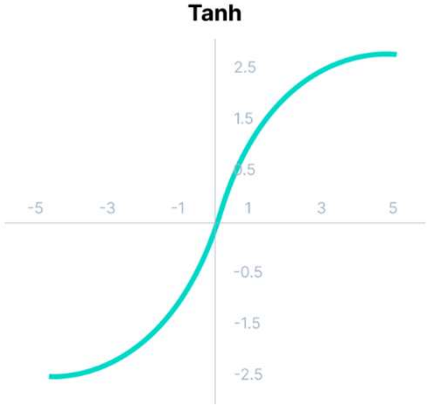
It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the Vanishing gradient problem.

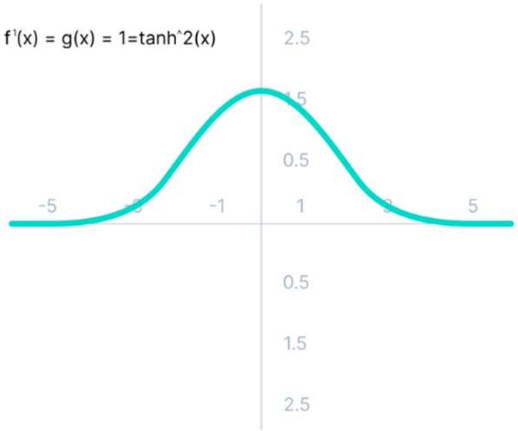
Hyperbolic Tangent

- Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1.
- Advantages of using this activation function are:
    - The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
    - Usually used in hidden layers of a neural network as its values lie between -1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

Hyperbolic Tangent

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$





As you can see— it also faces the problem of vanishing gradients similar to the sigmoid activation function. Plus the gradient of the tanh function is much steeper as compared to the sigmoid function.

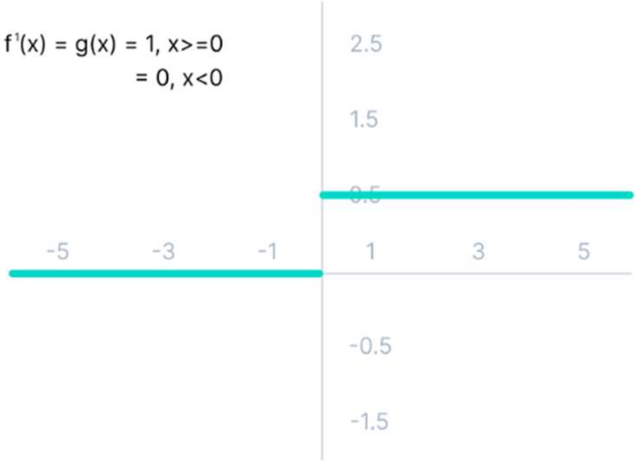
💡 Note: Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.

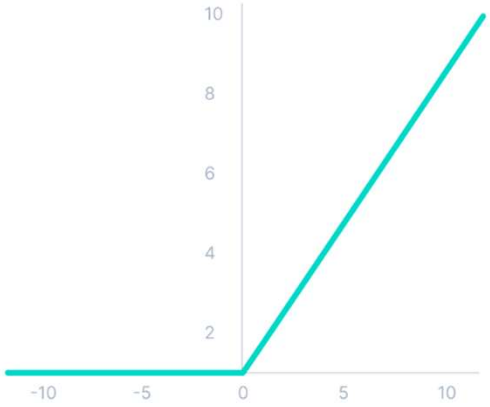
ReLU

- ReLU stands for Rectified Linear Unit.
- The main catch here is that the ReLU function does not activate all the neurons at the same time. The neurons will only be deactivated if the output of the linear transformation is less than 0.
- The advantages of using ReLU as an activation function are as follows:
  - Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
  - ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

ReLU

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


$f'(x) = g(x) = 1,\ x>=0$
$= 0,\ x<0$



ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

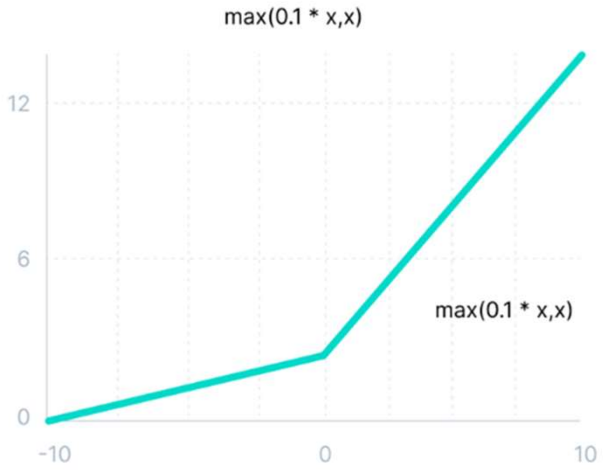
Also, The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

Leaky ReLU

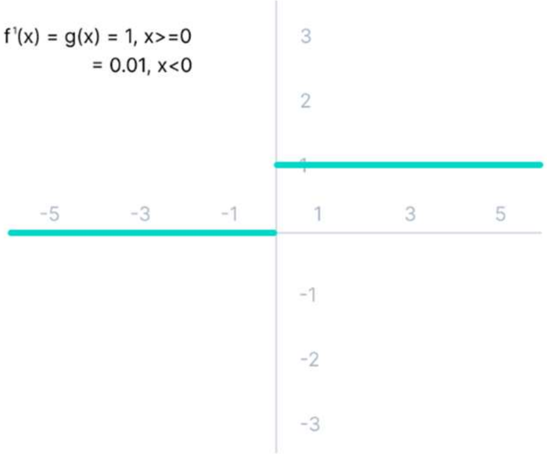
- Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.
- The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

- By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

Leaky ReLU



$$f(x) = max\ (0.1x, x)$$



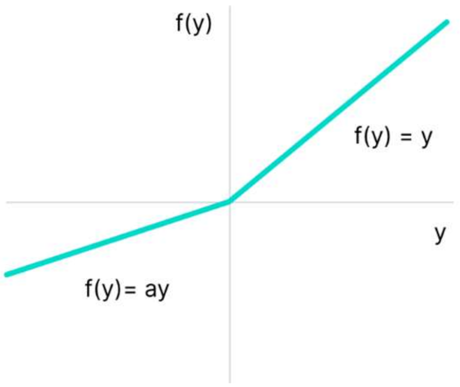
The predictions may not be consistent for negative input values. The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

Parametric ReLU

- Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient’s becoming zero for the left half of the axis.
- This function provides the slope of the negative part of the function as an argument a. By performing backpropagation, the most appropriate value of a is learnt.
- The parameterized ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.
- This function’s limitation is that it may perform differently for different problems depending upon the value of slope parameter ***a.***

Parametric ReLU

$$f(x) = max\ (ax, x)$$

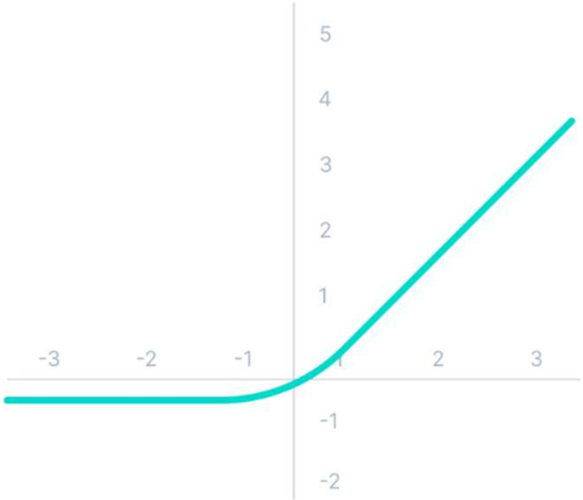


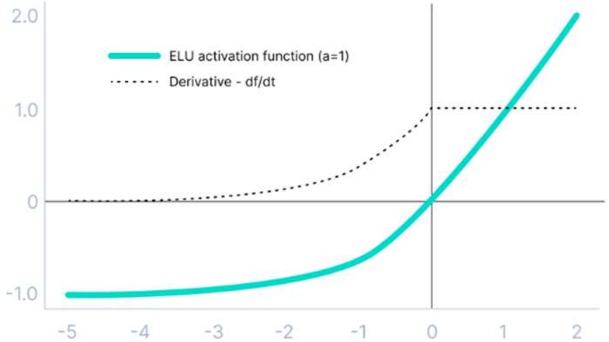
Exponential Linear Units (ELU)

- Exponential Linear Unit, or ELU for short, is also a variant of ReLU that modifies the slope of the negative part of the function.
- ELU uses a log curve to define the negativ values unlike the leaky ReLU and Parametric ReLU functions with a straight line.
- ELU is a strong alternative for f ReLU because of the following advantages:
    - ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.
    - Avoids dead ReLU problem by introducing log curve for negative values of input. It helps the network nudge weights and biases in the right direction.

Exponential Linear Units (ELU)

$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha\left(e^{x} - 1\right) & for\ x < 0 \end{cases}$$





$$f'(x) = \begin{cases} 1 & for\ x\ \geqslant 0 \\ f(x) + \alpha & for\ x\ < 0 \end{cases}$$

Softmax Function

- The Softmax function is described as a combination of multiple sigmoids.
- In the sigmoid function: Let’s suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it? The answer is: We can’t.
- The above values don’t make sense as the sum of all the classes/output probabilities should be equal to 1.
- It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.
- It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Softmax Function

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

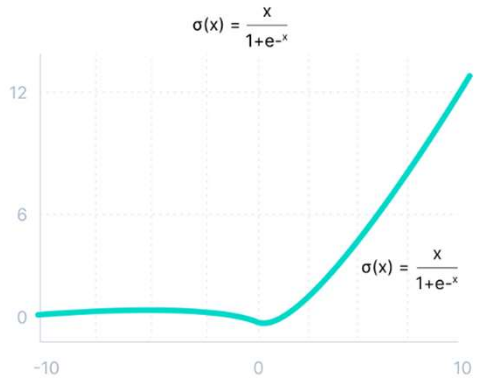
You can see now how softmax activation function make things easy for multi-class classification problems.

Swish

- It is a self-gated activation function developed by researchers at Google.
- Swish consistently matches or outperforms ReLU activation function on deep networks applied to various challenging domains such as image classification, machine translation etc.
- This function is bounded below but unbounded above i.e. Y approaches to a constant value as X approaches negative infinity but Y approaches to infinity as X approaches infinity.

Swish

$$f(x) = x*sigmoid(x)$$



Here are a few advantages of the Swish activation function over ReLU:

Swish is a smooth function that means that it does not abruptly change direction like ReLU does near x = 0. Rather, it smoothly bends from 0 towards values < 0 and then upwards again.
Small negative values were zeroed out in ReLU activation function. However, those negative values may still be relevant for capturing patterns underlying the data. Large negative values are zeroed out for reasons of sparsity making it a win-win situation.
The swish function being non-monotonous enhances the expression of input data and weight to be learnt.

Choosing the Right Activation Function

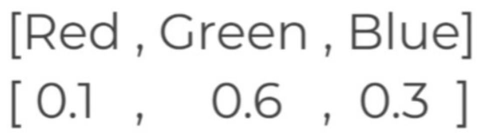
- You need to match your activation function for your output layer based on the type of prediction problem that you are solving—specifically, the type of predicted variable.
- As a rule of thumb, you can begin with using the ReLU activation function and then move over to other activation functions if ReLU doesn't provide optimum results.
- And here are a few other guidelines to help you out.
  - ReLU activation function should only be used in the hidden layers.
  - Sigmoid/Logistic and Tanh functions should not be used in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients).
  - Swish function is used in neural networks having a depth greater than 40 layers.

Choosing the Right Activation Function

- Regression - Linear Activation Function
- Binary Classification - Sigmoid/Logistic Activation Function
- Multiclass Classification - Softmax
- Multilabel Classification – Sigmoid
- Convolutional Neural Network (CNN): ReLU activation function.
- Recurrent Neural Network: Tanh and/or Sigmoid activation function

Multi-class Classification and Softmax

- Mutually exclusive classes: what to do when each data point can only have a single class assigned to it?
- We can use a very clever **softmax** function
  - It calculates probabilities distribution of the event over k different events
  - It also calculates probabilities of each target class over all possible target classes
  - The range will be 0 – 1. The sum of all the probabilities will be equal to one,
  - The target class chosen will have the highest probability
  - You get this sort of output:



```
[Red , Green , Blue]
[ 0.1  ,    0.6  , 0.3  ]
```

Evaluating a Model: Cost Function

- After the network creates its predictions we have two questions:
  - How do we evaluate it?
  - How to update the network’s weights and biases
- We need to take estimated outputs of the network and then compare them to the real values of the label
- The cost function also known as the loss function must be a kind of average so it can output a single value
- We can keep track of out loss/cost during training to monitor network performance

Cost Function

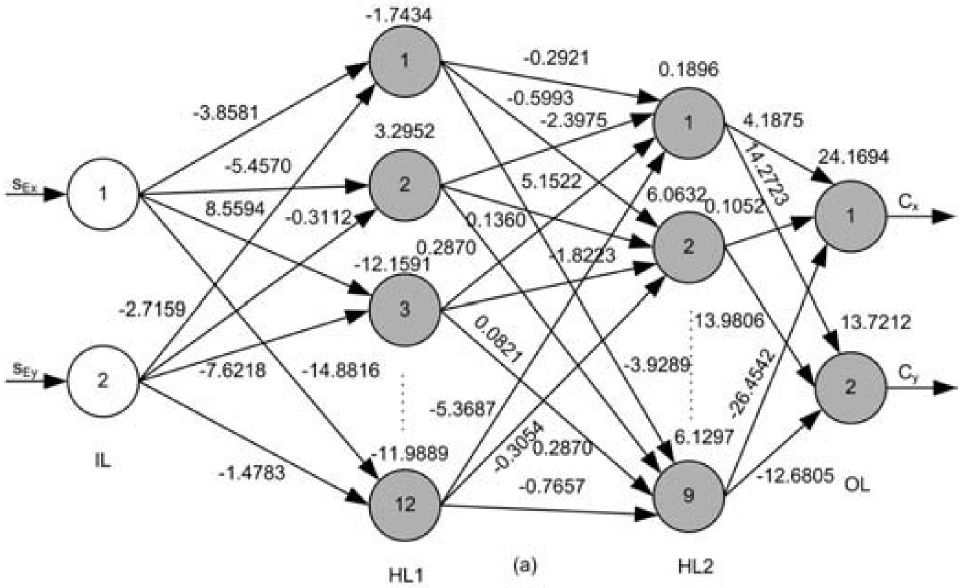
- In general a cost function is a function of weights, biases, input of a training sample and desired output of the training sample:
- One very common cost function is the quadratic cost function
- We simply calculate the difference between the real values y(x) against our predicted values a(x):

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

L -> layer, a activation function which contains information about weights and biases

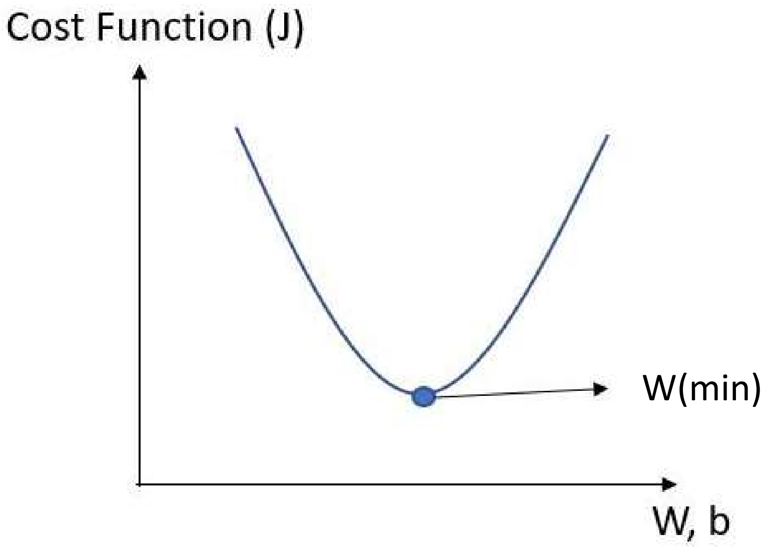
Cost Function

- Here’s a small network with all it’s parameters labelled
- How do we calculate the cost function and minimize it?
  - Which particular weights minimize the cost function

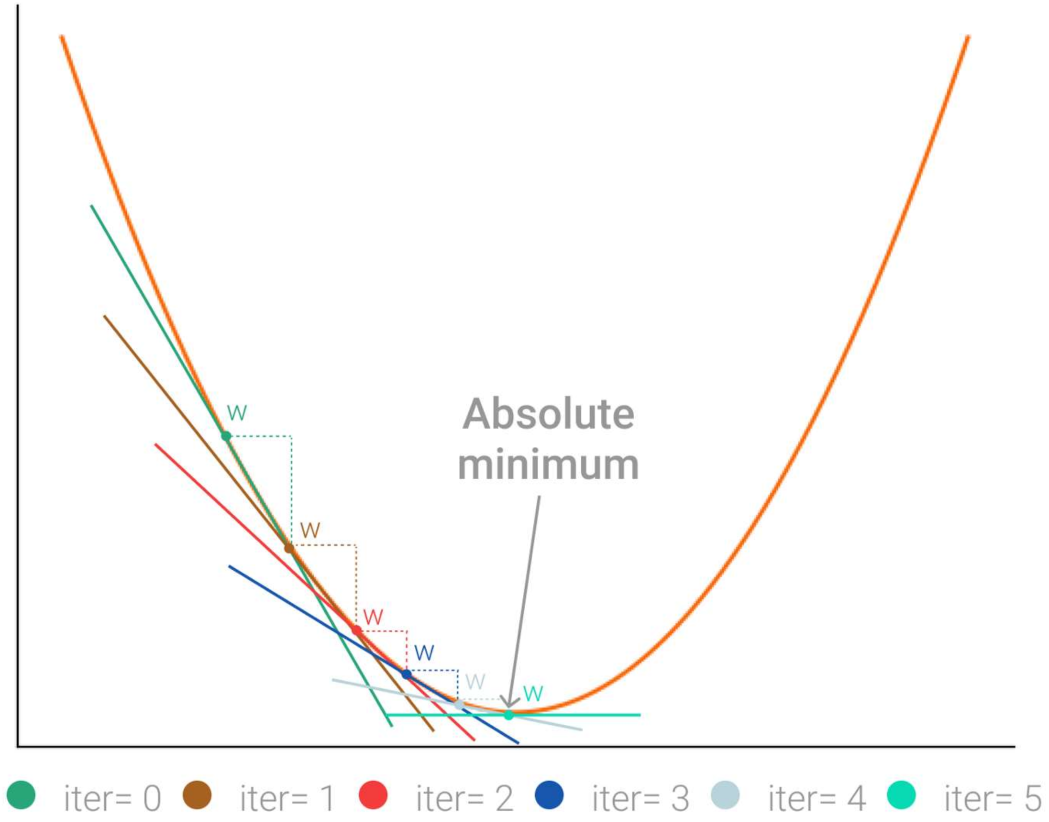


Cost Function

- For simplicity, let's imagine we only has one weight in our cost functions w, we want to minimize our loss
- That would mean, we need to figure out what value of w results in the minimum c(w)
- In this case, we have a plot like this:
- Real cost function is real complex
  - Use a stochastic process
  - Use Gradient descent

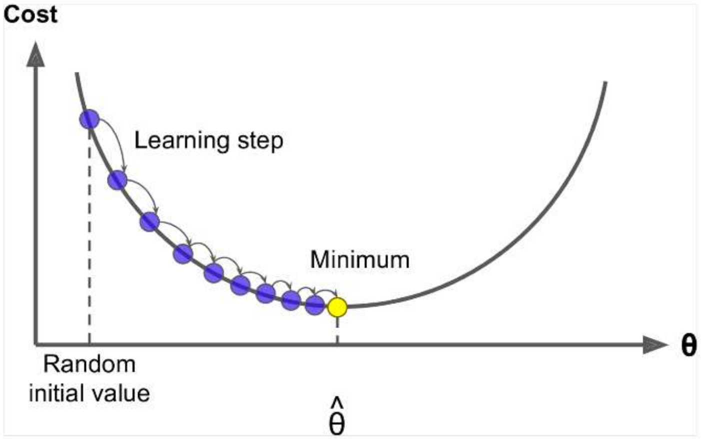


Gradient Descent



Gradient Descent: Learning Rate

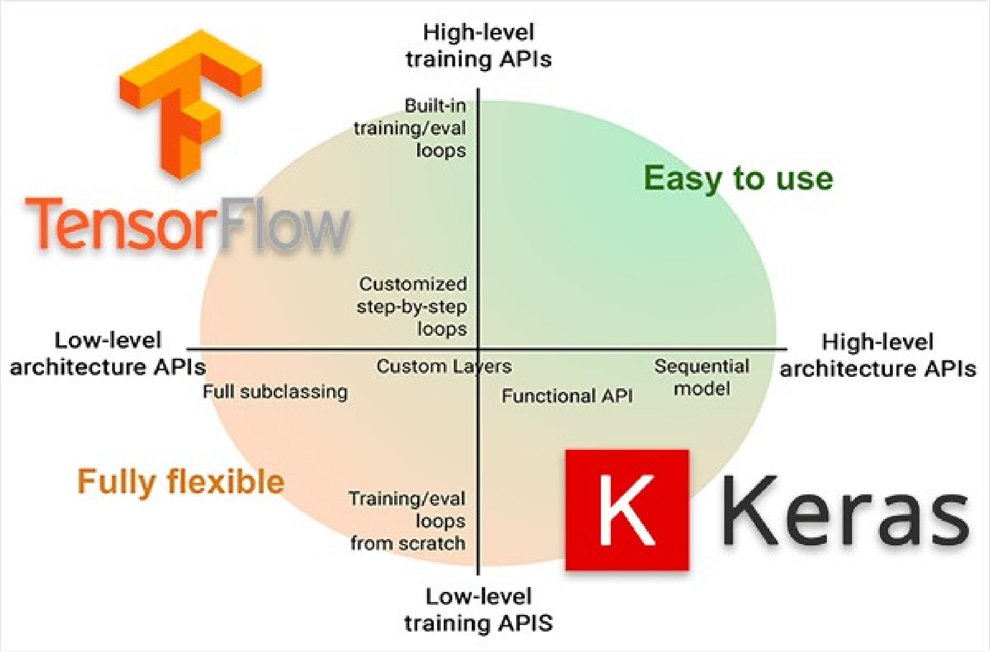
- Larger step size: you might miss absolute minimum point
- Small step size: you will take more time to reach absolute minimum
- The step size is called as the **Learning Rate**
    - Step sizes were equal previously
    - But, we could start with larger steps, then go smaller as we realize the slope gets closer to zero, this is called **adaptive gradient descent**



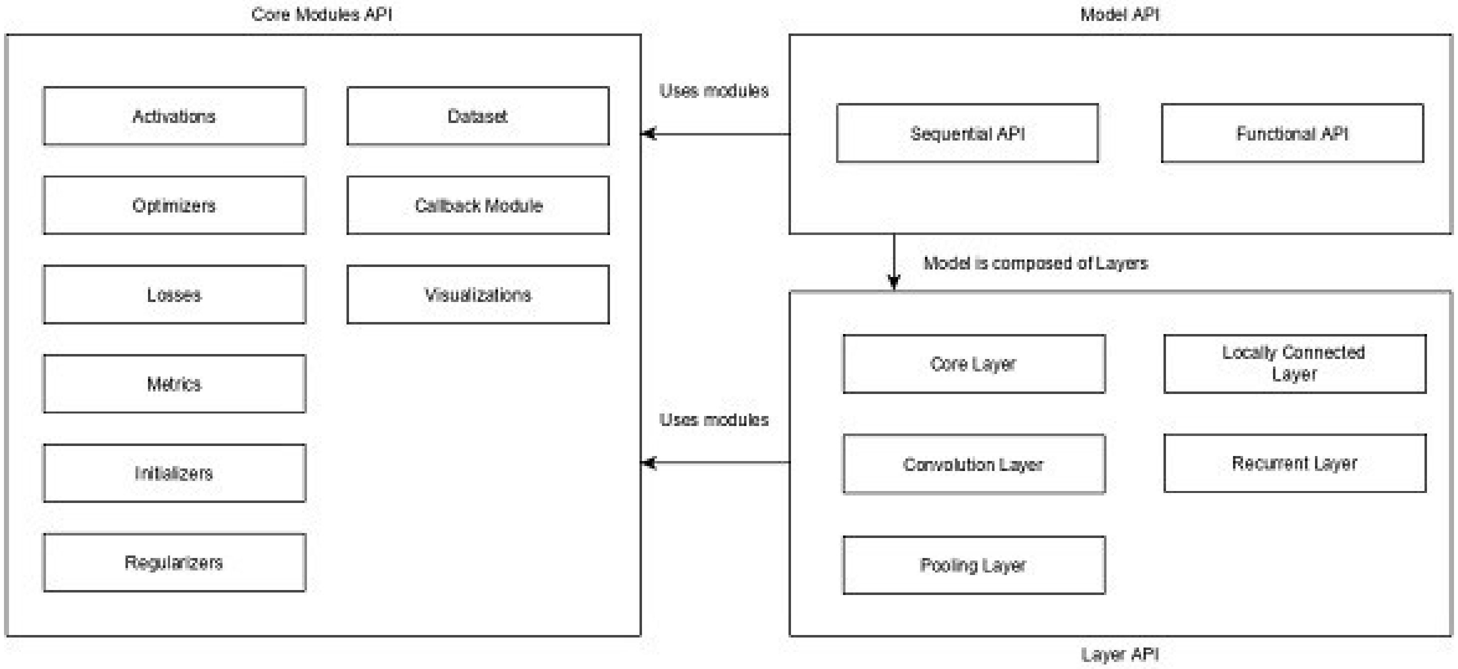
Keras

- Keras provides a complete framework to create any type of neural networks.
- Keras API can be divided into three main categories −
  - Model
  - Layer
  - Core Modules

TensorFlow and Keras



Keras Architecture



Sequential Model

- Sequential model is basically a linear composition of Keras Layers. Sequential model is easy, minimal as well as has the ability to represent nearly all available neural networks.

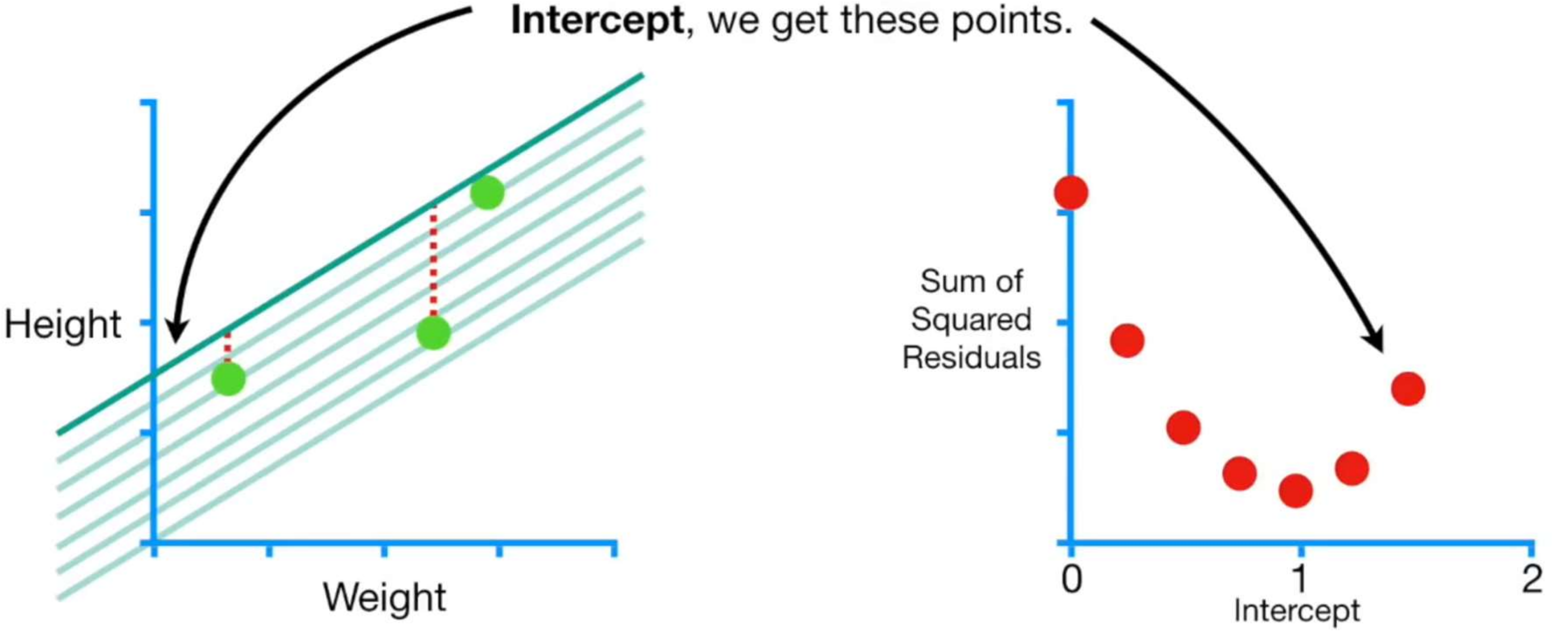
Layers

- Each Keras layer in the Keras model represent the corresponding layer (input layer, hidden layer and output layer) in the actual proposed neural network model. Keras provides a lot of pre-build layers so that any complex neural network can be easily created. Some of the important Keras layers are specified below,
  - Core Layers
  - Convolution Layers
  - Pooling Layers
  - Recurrent Layers

Core Modules

- Keras also provides a lot of built-in neural network related functions to properly create the Keras model and Keras layers. Some of the function are as follows −
    - **Activations module** − Activation function is an important concept in ANN and activation modules provides many activation function like softmax, relu, etc.,
    - **Loss module** − Loss module provides loss functions like mean_squared_error, mean_absolute_error, poisson, etc.,
    - **Optimizer module** − Optimizer module provides optimizer function like adam, sgd, etc.,
    - **Regularizers** − Regularizer module provides functions like L1 regularizer, L2 regularizer, etc.,

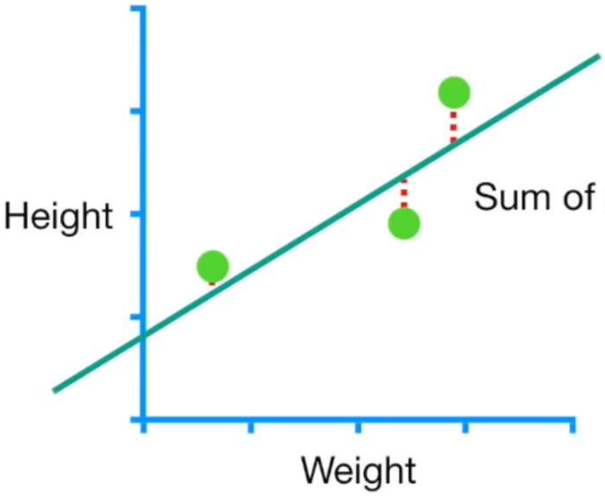
Gradient Descent



And for increasing values for the **Intercept**, we get these points.

Gradient Descent

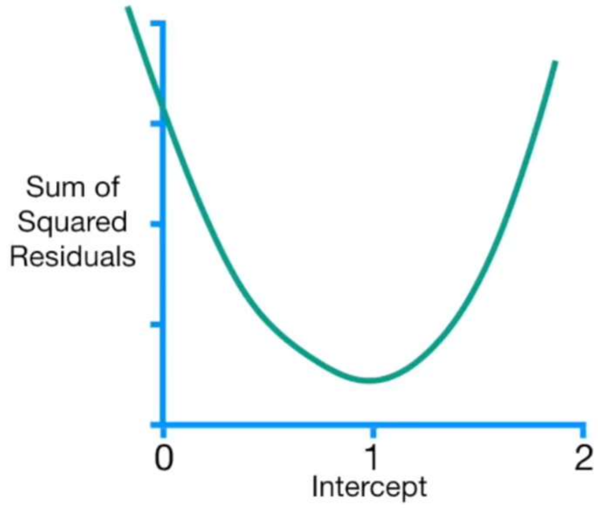
Using SS (Residuals as Loss Function)



$$\begin{aligned}\text{Sum of squared residuals} &= (\mathbf{1.4} - (\text{intercept} + 0.64 \times \mathbf{0.5}))^2 \\ &+ (\mathbf{1.9} - (\text{intercept} + 0.64 \times \mathbf{2.3}))^2 \\ &+ (\mathbf{3.2} - (\text{intercept} + 0.64 \times \mathbf{2.9}))^2\end{aligned}$$

Gradient Descent

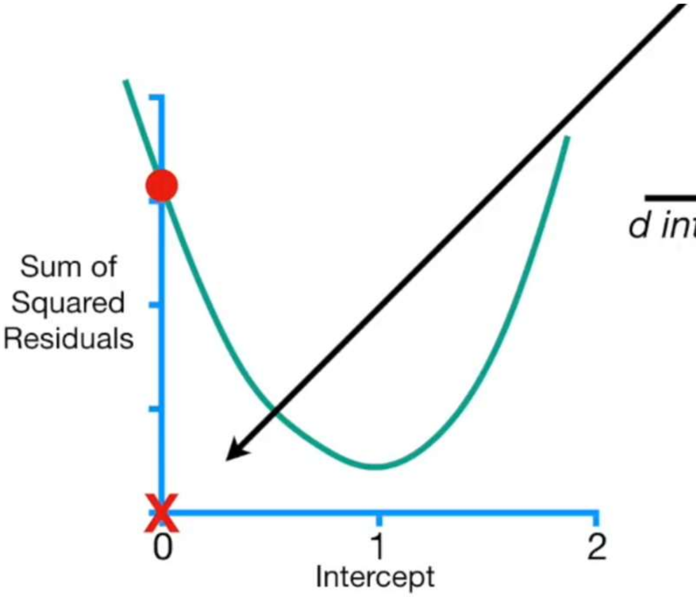
Take the Derivative of the Loss Function



$$\begin{aligned}\frac{d}{d\ \textit{intercept}}\ \text{Sum of squared residuals} = \\ -2(\mathbf{1.4} - (\text{intercept} + 0.64 \times \mathbf{0.5}) \\ + -2(\mathbf{1.9} - (\text{intercept} + 0.64 \times \mathbf{2.3})) \\ + -2(\mathbf{3.2} - (\text{intercept} + 0.64 \times \mathbf{2.9}))\end{aligned}$$

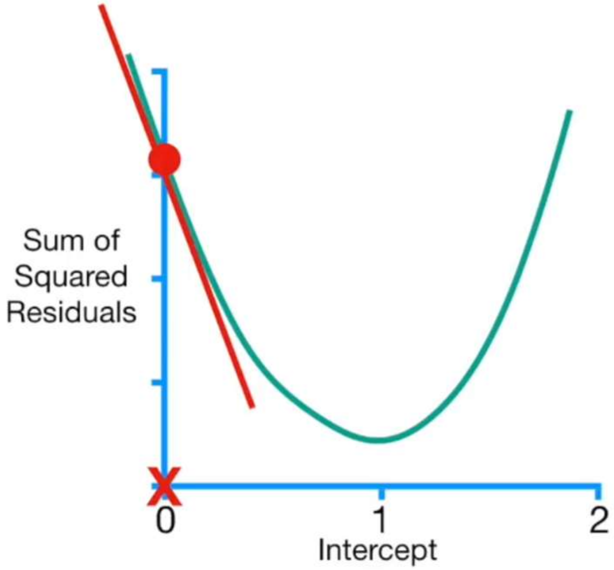
Gradient Descent

Pick random value for the intercept, in this case 0



$$\frac{d}{d\ intercept}\ \text{Sum of squared residuals} = -2(\mathbf{1.4} - (\text{intercept} + 0.64 \times \mathbf{0.5})$$
$$+ -2(\mathbf{1.9} - (\text{intercept} + 0.64 \times \mathbf{2.3}))$$
$$+ -2(\mathbf{3.2} - (\text{intercept} + 0.64 \times \mathbf{2.9}))$$

Gradient Descent



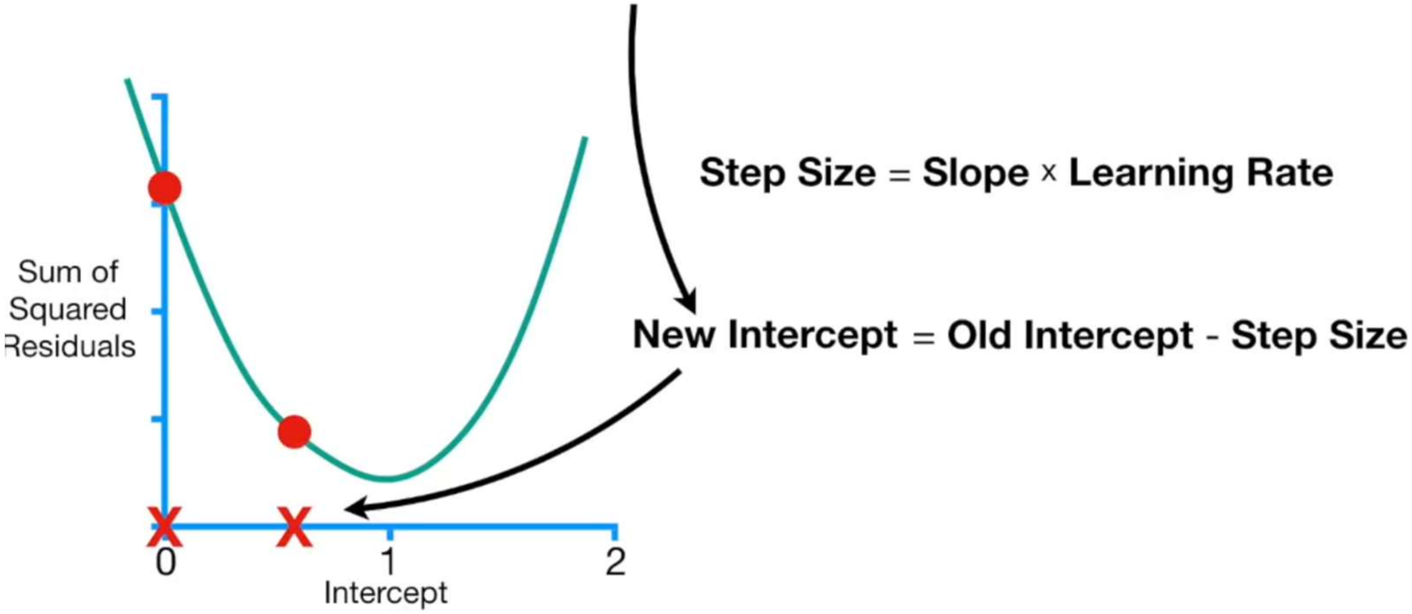
Calculate the step size

Step Size = **Slope** × **Learning Rate**

In practice, minimum learning rate is 0.001

Gradient Descent

Calculate the new intercept

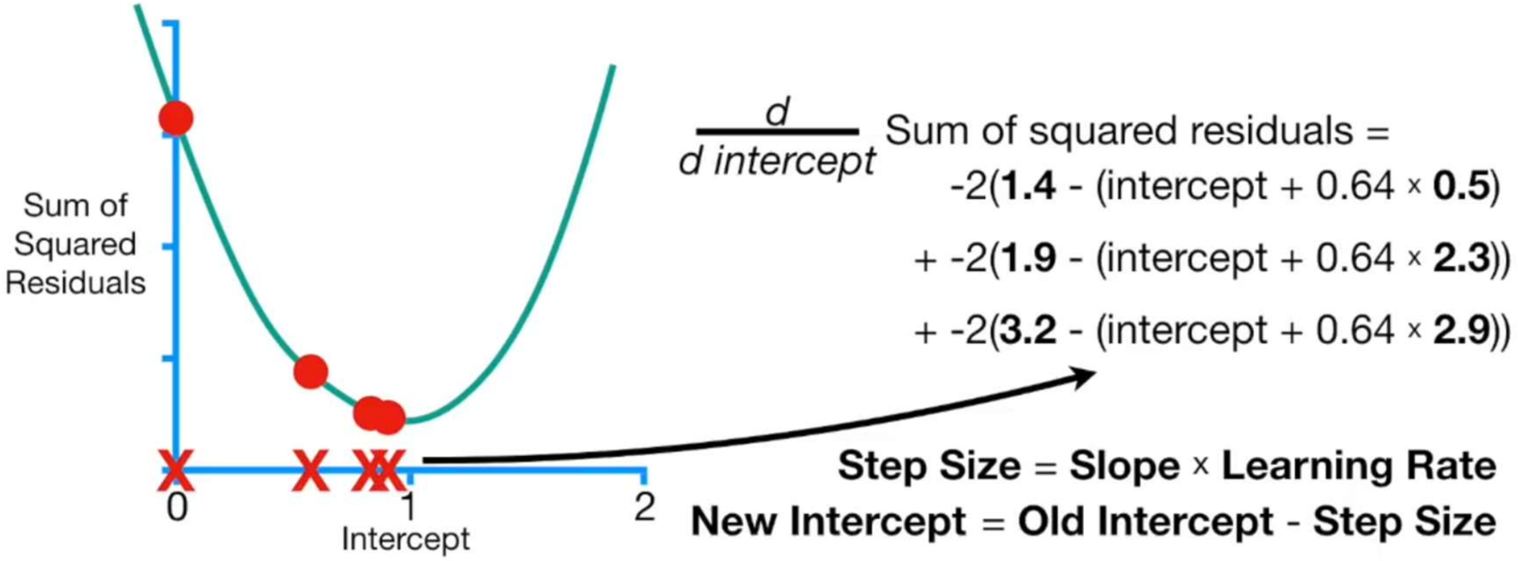


Step Size = **Slope** × **Learning Rate**

New Intercept = **Old Intercept** - **Step Size**

Gradient Descent

Plug the new intercept to the derivative and repeat everythinguntil Step Size is close to 0



$$\frac{d}{d\,intercept}\text{ Sum of squared residuals} =$$

$$-2(\mathbf{1.4} - (\text{intercept} + 0.64 \times \mathbf{0.5})$$

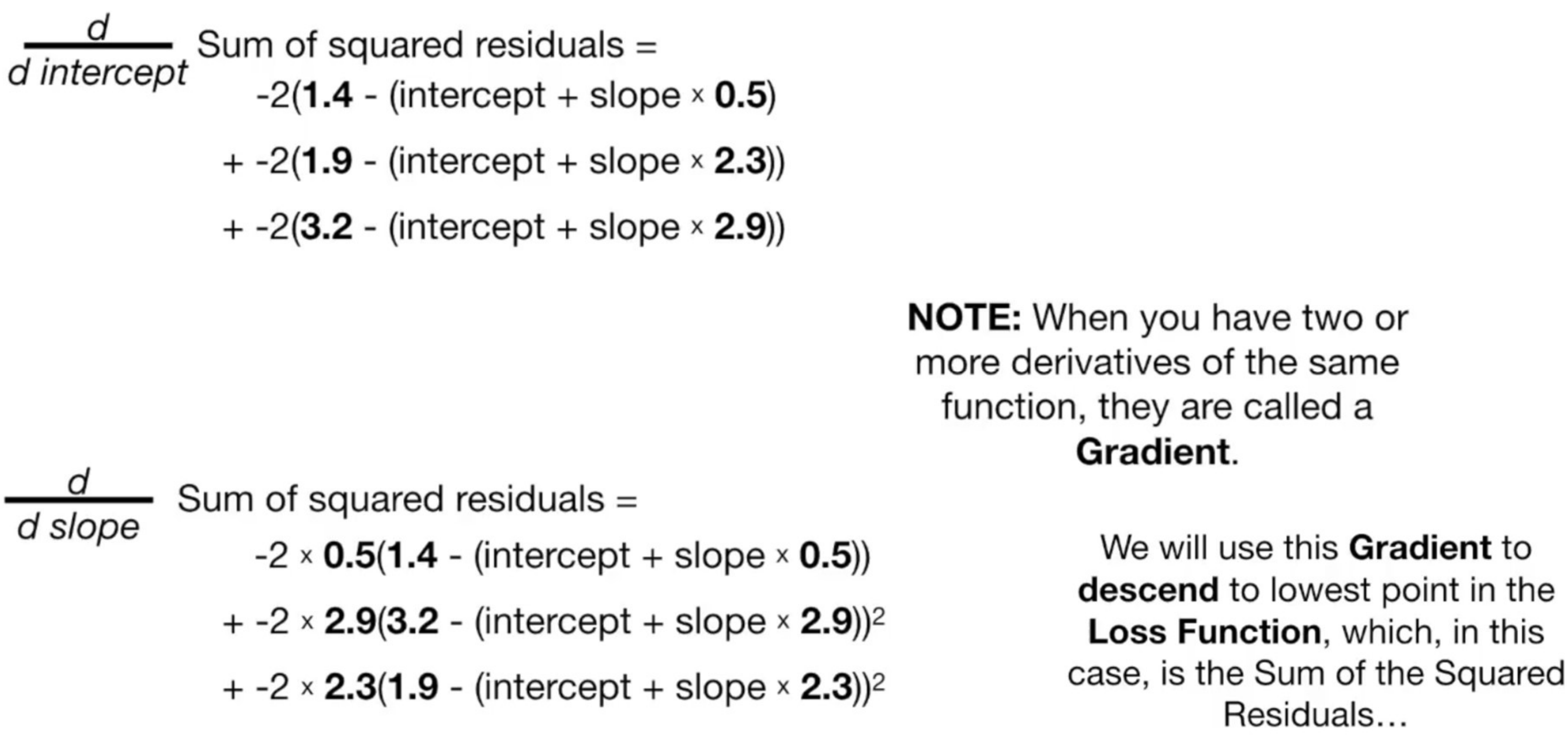
$$+ -2(\mathbf{1.9} - (\text{intercept} + 0.64 \times \mathbf{2.3}))$$

$$+ -2(\mathbf{3.2} - (\text{intercept} + 0.64 \times \mathbf{2.9}))$$

Step Size = **Slope** × **Learning Rate**

New Intercept = **Old Intercept** - **Step Size**

Gradient Descent



$$\frac{d}{d\ intercept}\ \text{Sum of squared residuals} =$$

$$-2(\mathbf{1.4} - (\text{intercept} + \text{slope} \times \mathbf{0.5})$$

$$+ -2(\mathbf{1.9} - (\text{intercept} + \text{slope} \times \mathbf{2.3}))$$

$$+ -2(\mathbf{3.2} - (\text{intercept} + \text{slope} \times \mathbf{2.9}))$$

NOTE: When you have two or more derivatives of the same function, they are called a **Gradient**.

$$\frac{d}{d\ slope}\ \text{Sum of squared residuals} =$$

$$-2 \times \mathbf{0.5}(\mathbf{1.4} - (\text{intercept} + \text{slope} \times \mathbf{0.5}))$$

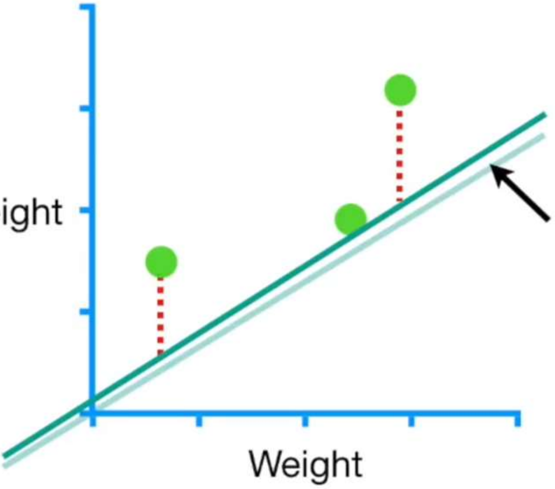
$$+ -2 \times \mathbf{2.9}(\mathbf{3.2} - (\text{intercept} + \text{slope} \times \mathbf{2.9}))^2$$

$$+ -2 \times \mathbf{2.3}(\mathbf{1.9} - (\text{intercept} + \text{slope} \times \mathbf{2.3}))^2$$

We will use this **Gradient** to **descend** to lowest point in the **Loss Function**, which, in this case, is the Sum of the Squared Residuals…

Gradient Descent!!!

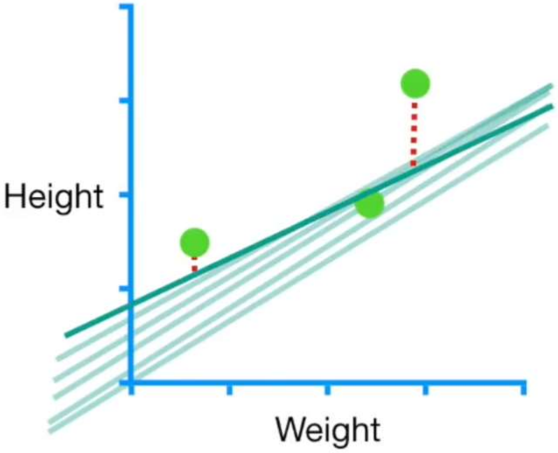
Gradient Descent



New Intercept = 0 - (-0.016) = 0.016

…and this is the new line (with **Slope = 1.008** and **Intercept = 0.016**) after the first step.

New Slope = 1 - (-0.008) = 1.008

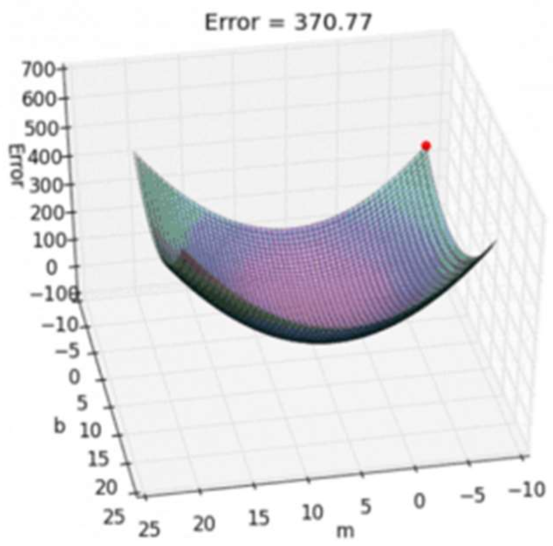


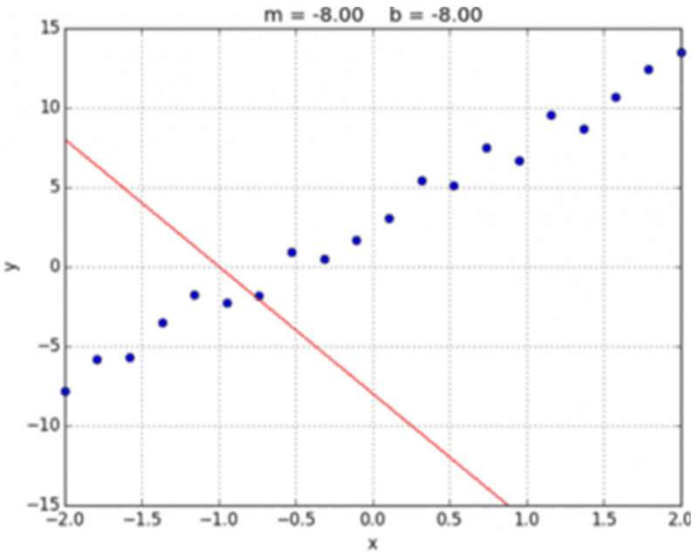
After repeated steps! Actually speaking it’s same as SS

Gradient Descent: Summary

- No matter what loss function you choose, gradient descent works the same way!!
- Step 1: Take derivative of the loss function
- Step 2: Pick random values for the parameters
- Step 3: Plug parameter values in to the derivatives
- Step 4: Calculate step sizes
- Step 5: Calculate new parameters
- Repeat 3 to 5 until step size is small or you reach the maximum number of steps

Gradient Descent





RMSProp Optimizer

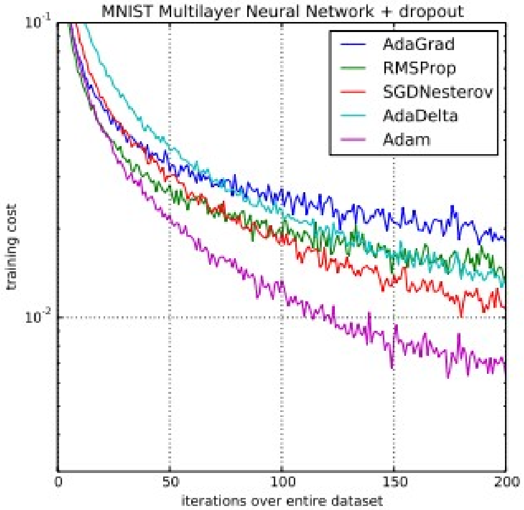
- RMSprop is a gradient-based optimization technique used in training neural networks. It was proposed by the father of back-propagation, Geoffrey Hinton
- Rmsprop was developed as a stochastic technique for mini-batch learning.
- Gradients of very complex functions like neural networks have a tendency to either vanish or explode as the data propagates through the function.
- RMSprop deals with the above issue by using a moving average of squared gradients to normalize the gradient. This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.

RMSProp Optimizer

- Simply put, RMSprop uses an adaptive learning rate instead of treating the learning rate as a hyperparameter. This means that the learning rate changes over time.

ADAM

- In 2015, Kingma and Ba published their paper: “ADAM: A Method for Stochastic Optimization”
- ADAM is a much more efficient way of searching for these minimums



Batch

- The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.
- Think of a batch as a for-loop iterating over one or more samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.
- A training dataset can be divided into one or more batches.

Batch Gradient

- In Batch Gradient Descent, all the training data is taken into consideration to take a single step.
- We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters.
- So that’s just one step of gradient descent in one epoch.

Stochastic Gradient

- Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way.
- In SGD we do the following steps:
  - Take an example
  - Feed it to Neural Network
  - Calculate it’s gradient
  - Use the gradient we calculated in step 3 to update the weights
  - Repeat steps 1–4 for all the examples in training dataset
- SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

Mini Batch Gradient

- Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets.
- But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.
- Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch.

Mini Batch Gradient

- The following steps are done in a single epoch:
  - Pick a mini-batch
  - Feed it to Neural Network
  - Calculate the mean gradient of the mini-batch
  - Use the mean gradient we calculated in step 3 to update the weights
  - Repeat steps 1–4 for the mini-batches we created

Batch: Summary

- **Batch Gradient Descent**. Batch Size = Size of Training Set
- **Stochastic Gradient Descent**. Batch Size = 1
- **Mini-Batch Gradient Descent**. 1 < Batch Size < Size of Training Set

Epoch

- The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.
- One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches.
- For example, an epoch that has one batch is called the batch gradient descent learning algorithm.

Batch, Epoch: What’s the Difference?

- The batch size is a number of samples processed before the model is updated.
- The number of epochs is the number of complete passes through the training dataset.
- The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset.
- The number of epochs can be set to an integer value between one and infinity. You can run the algorithm for as long as you like and even stop it using other criteria besides a fixed number of epochs, such as a change (or lack of change) in model error over time.

Back Propagation

- Fundamentally, we want to know how the cost function results changes with respect to the weights in the network, so we can update the weights to minimize the cost function
- Back-propagation is just a way of propagating the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updating the weights in such a way that minimizes the loss by giving the nodes with higher error rates lower weights and vice versa.