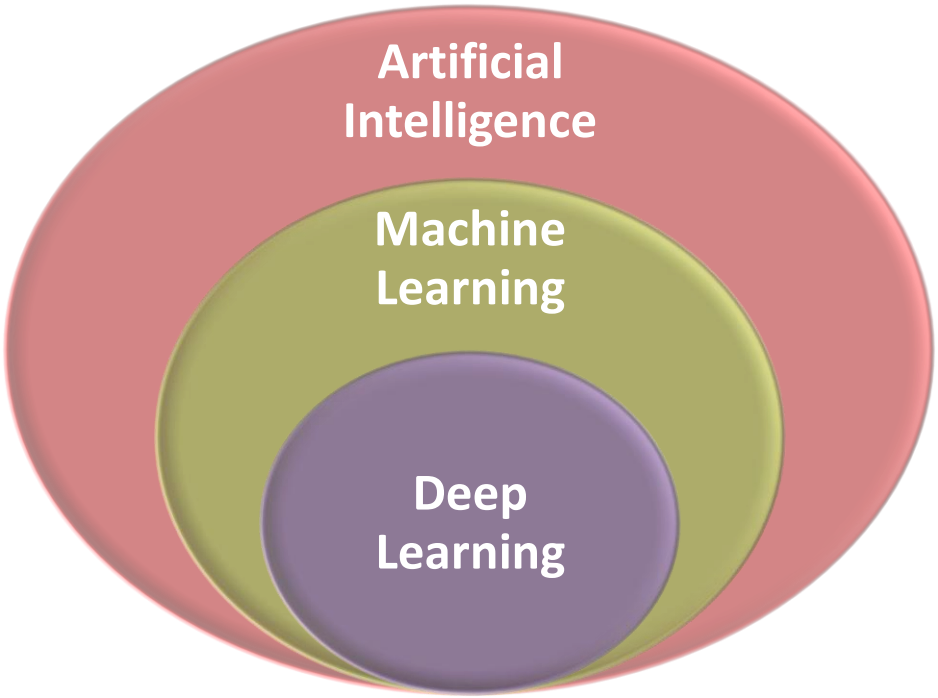
Deep Learning

Deep Learning

- Deep learning is an immensely rich subfield of machine learning, with powerful applications ranging from machine perception to natural language processing, all the way up to creative AI.



Artificial Intelligence, Machine Learning and Deep Learning

Artificial Intelligence

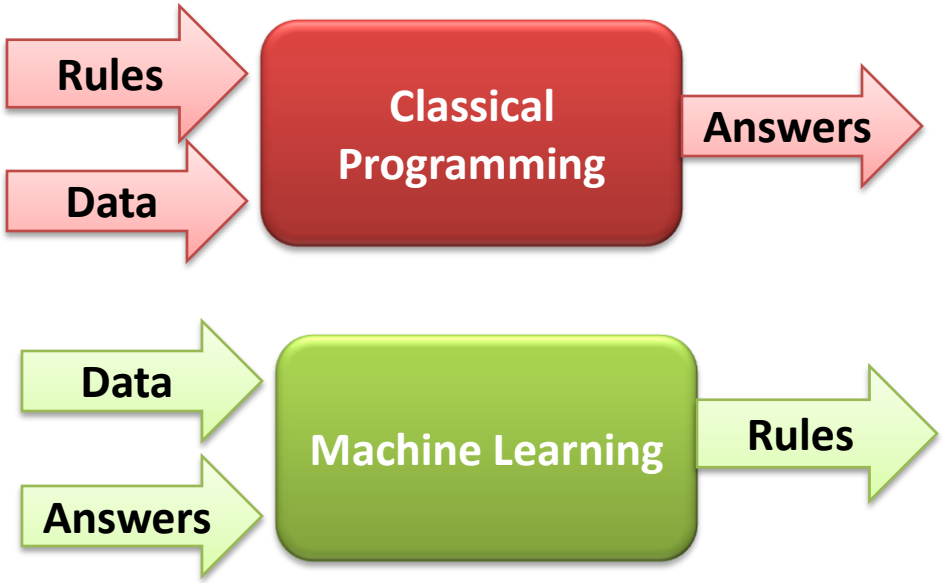
- A concise definition of **AI** would be:

“ the effort to automate intellectual tasks normally performed by humans. “

- AI is a very general field which encompasses machine learning and deep learning

Machine Learning

- With Machine Learning, humans would input data as well as the answers expected from the data, and out would come the rules.
- These rules could then be applied to new data to produce original answers.



To do machine learning, we need three things:

- ***Input data points.***
- ***Examples of the expected output.***
- ***A way to measure if the algorithm is doing a good job, to measure the distance between its current output and its expected output.***

  This is used as a feedback signal to adjust the way the algorithm works.

  This adjustment step is what we call "learning".

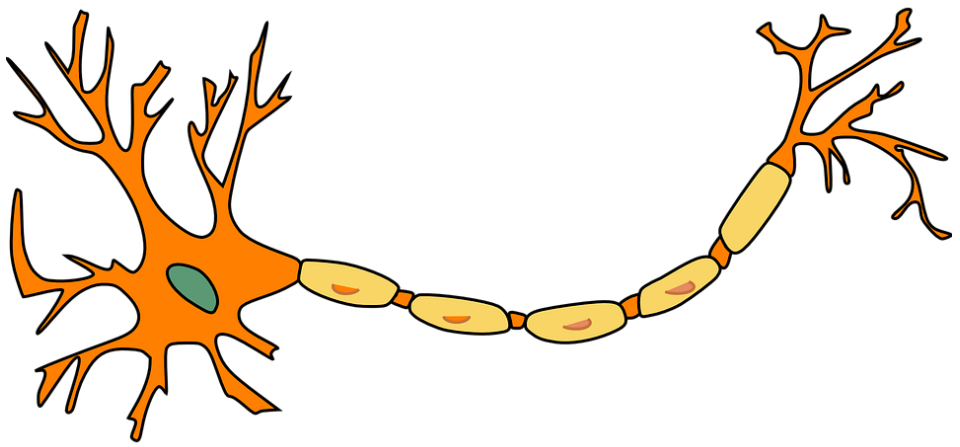
Deep Learning

- ***Deep learning is a specific subfield of machine learning.***

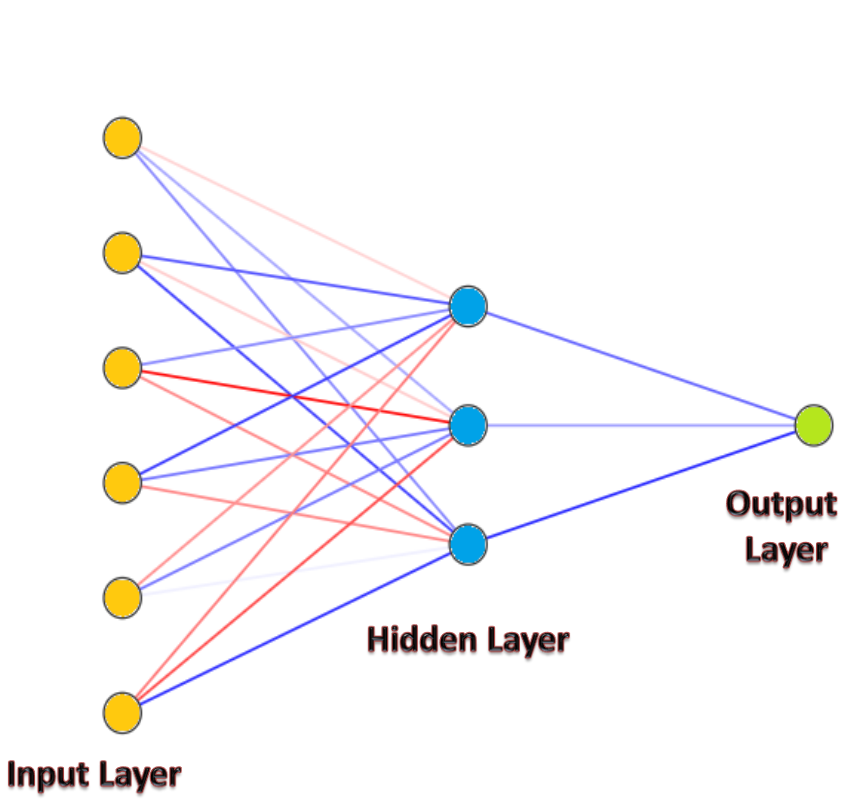
- ***The "deep" in "deep learning" simply stands for this idea of successive layers of***

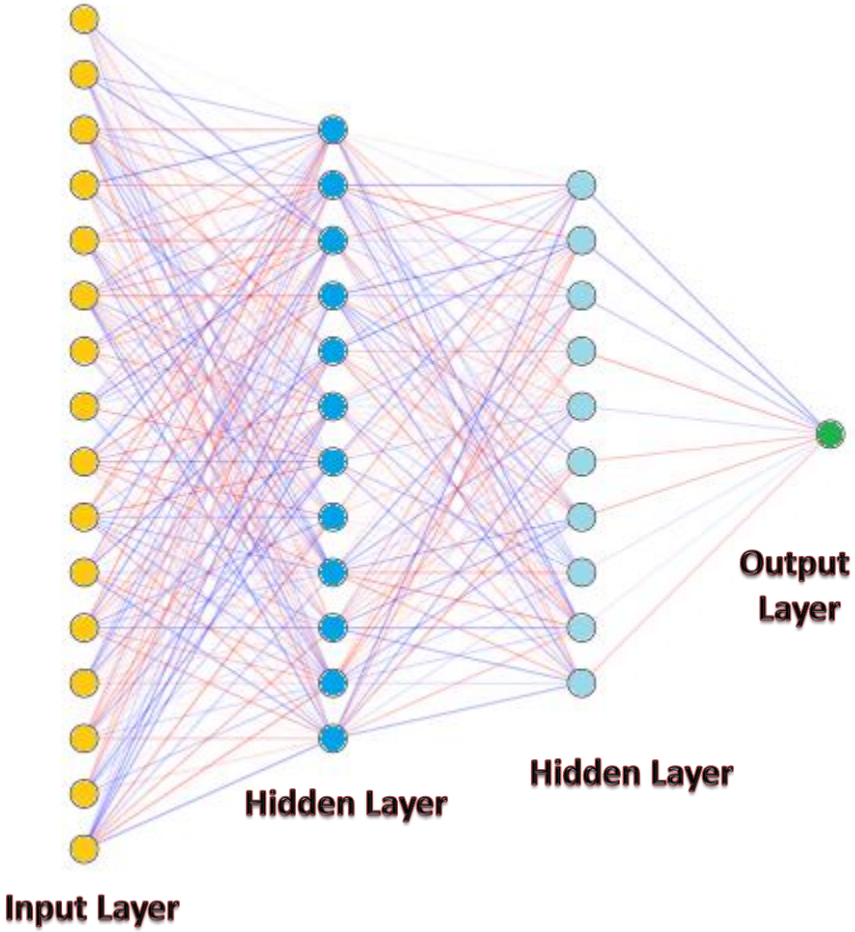
Deep Learning

- In deep learning, these layered representations are learned via models called "neural networks", structured in literal layers stacked one after the other.
- The term "neural network" is a reference to neurobiology.



Deep Neural Networks

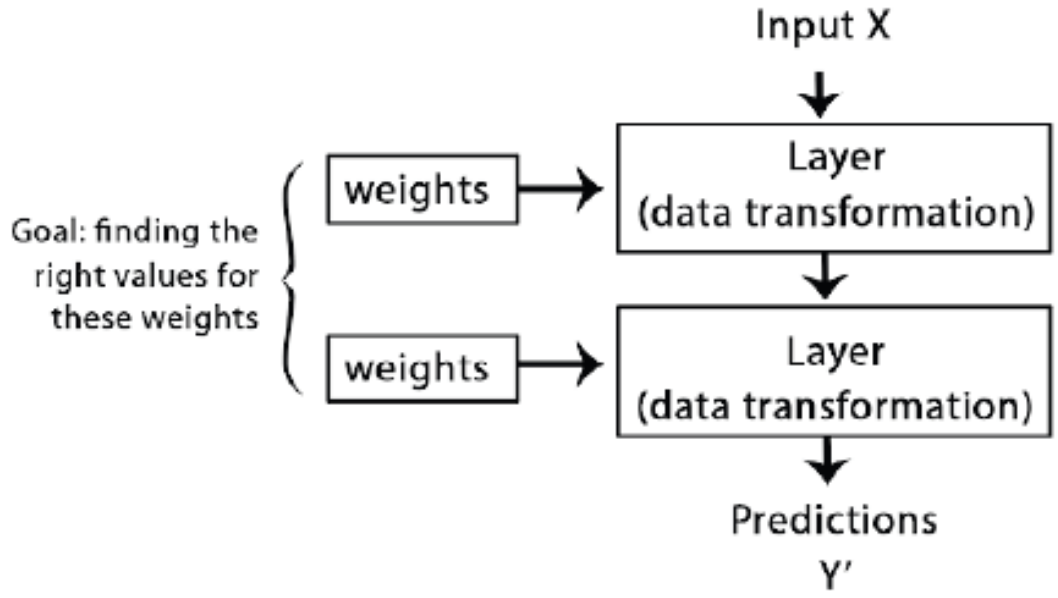




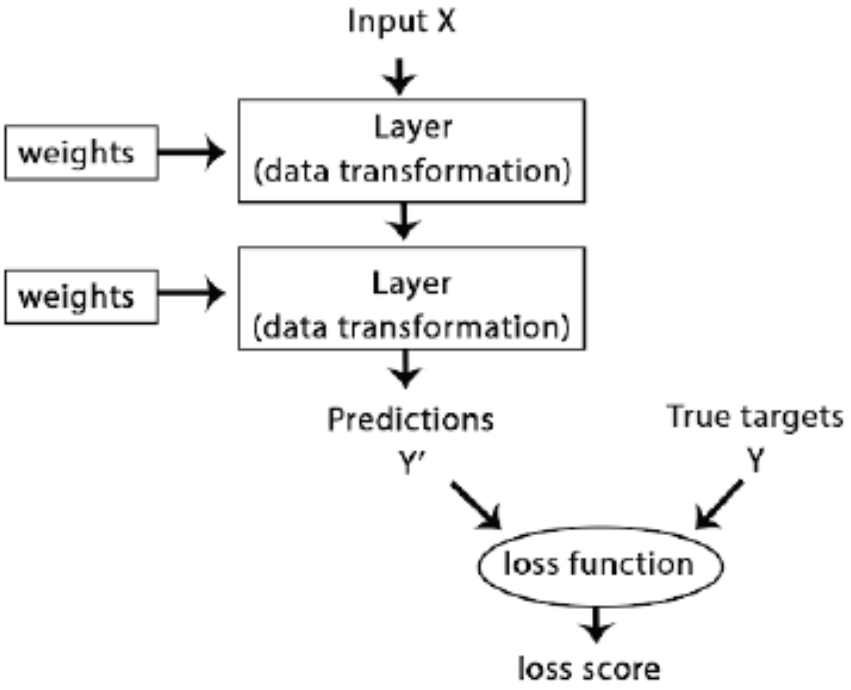
Understanding how deep learning works

- *Deep neural networks do input-to-target mapping via a deep sequence of simple data transformations called "layers", and that these data transformations are learned by exposure to examples.*

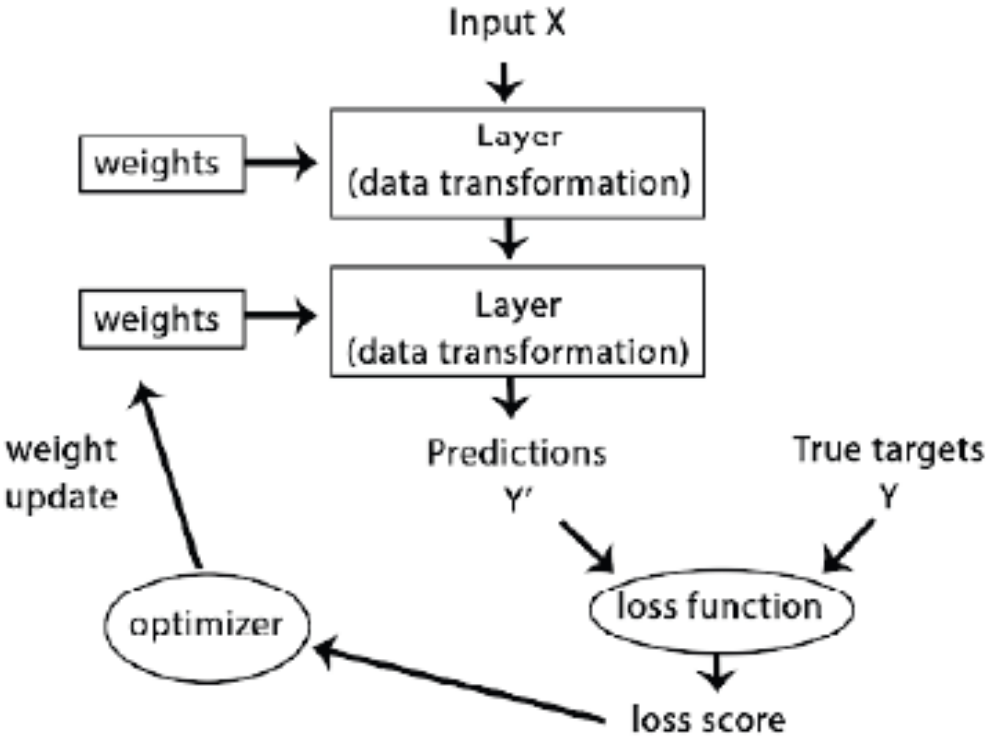
- The specification of what a layer does to its input data is stored in the layer's **"weights"**, which in essence are a bunch of numbers.
- Weights are also sometimes called the **"parameters"** of a layer.
- **"learning"** will mean finding a set of values for the weights of all layers in a network, such that the network will correctly map your example inputs to their associated targets.



- To control the output of a neural network, you need to be able to measure how far this output is from what you expected.
- This is the job of the **"loss function"** of the network, also called **"objective function".**
- The loss function takes the predictions of the network and the true target , and computes a distance score, capturing how well the network has done on this specific example.



- The fundamental trick in deep learning is to use the loss score as a feedback signal to adjust the value of the weights by a little bit, in a direction that would lower the loss score for the current example.
- This adjustment is the job of the "optimizer", which implements what is called the **"backpropagation"** algorithm, the central algorithm in deep learning.



What deep learning has achieved so far

- Deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

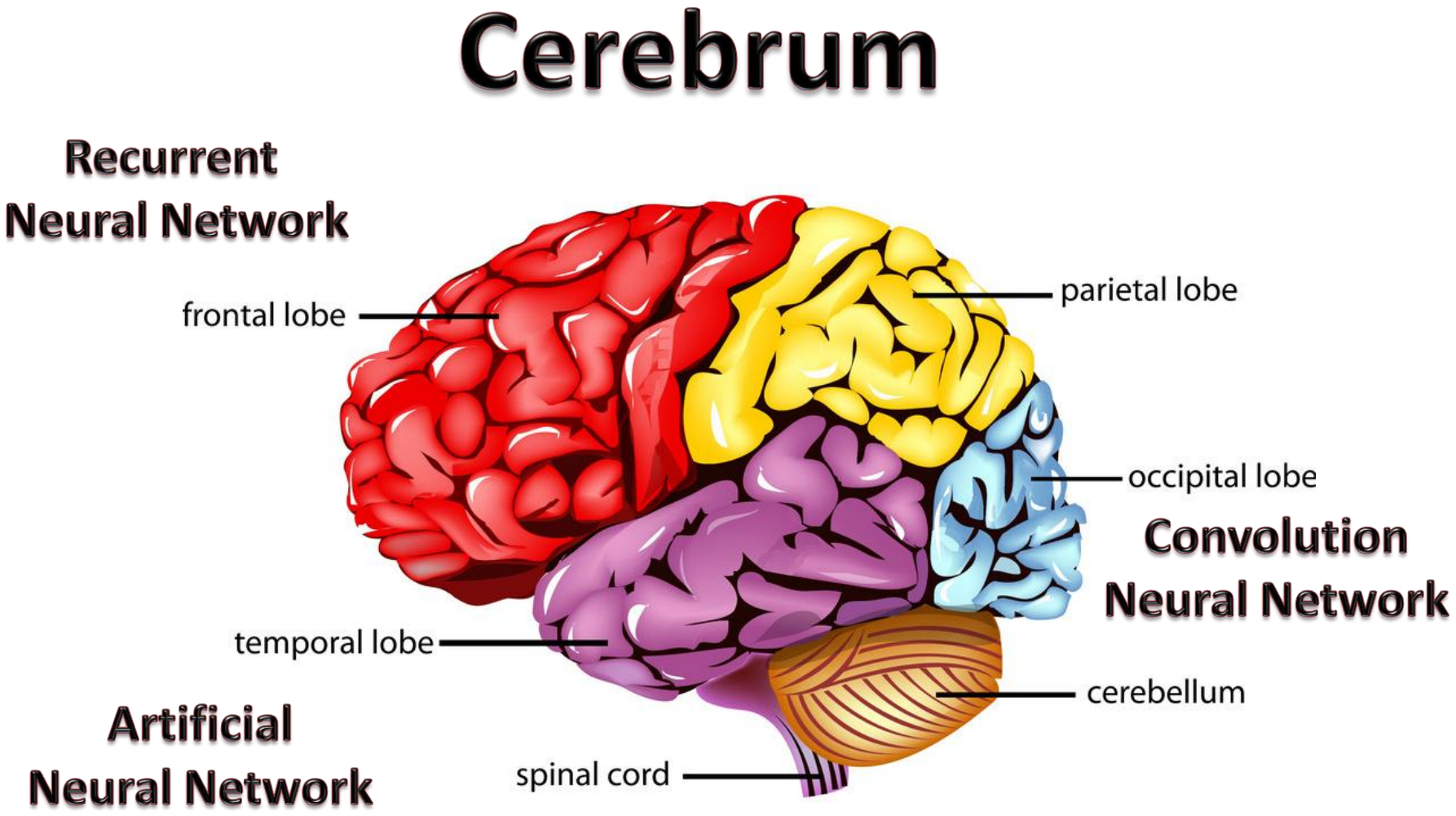
✓ *Near-human level image classification.*
✓ *Near-human level speech recognition.*
✓ *Near-human level handwriting transcription.*
✓ *Improved text-to-speech conversion.*
✓ *Digital assistants such as Amazon Alexa.*
✓ *Improved ad targeting, as used by Google, Bing.*
✓ *Improved search results on the web.*

What makes deep learning different

- Deep learning took off so quickly is primarily that it offered better performance on many problems.
- Deep learning is also making problem-solving much easier, because it completely automates what used to be the most crucial step in a machine learning workflow:

"feature engineering".

Human Brain


Cerebrum
Recurrent
Neural Network
frontal lobe
parietal lobe
occipital lobe
Convolution
Neural Network
temporal lobe
cerebellum
Artificial
Neural Network
spinal cord

Neural Network Arch.

Input Layer

- This layer accepts input features.
- It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer.

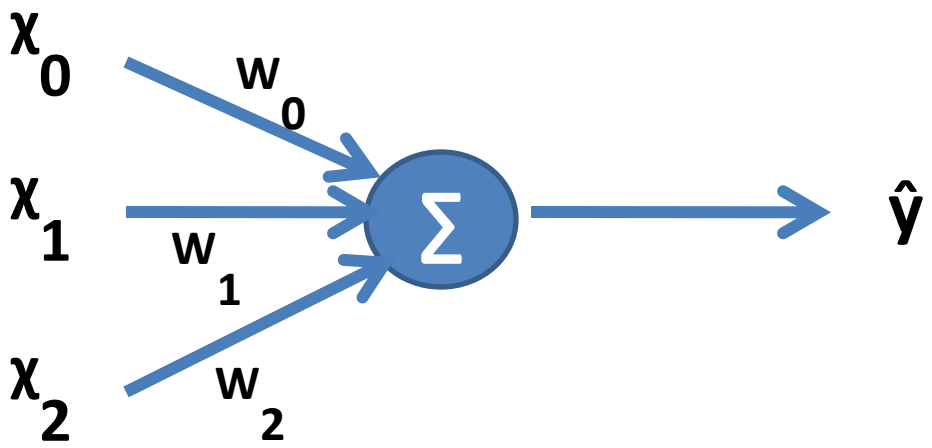
Hidden Layer

- Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network.
- Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.

Output Layer

- This layer returns the final output computed by the network to the application.

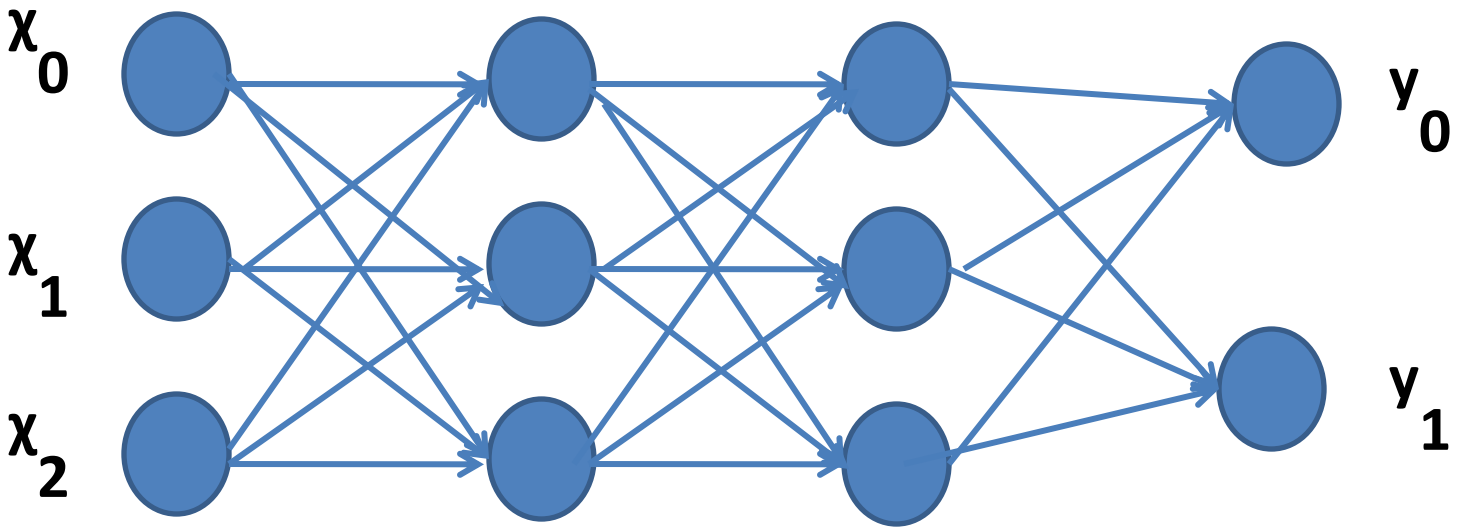
Neural Network with Single Neuron



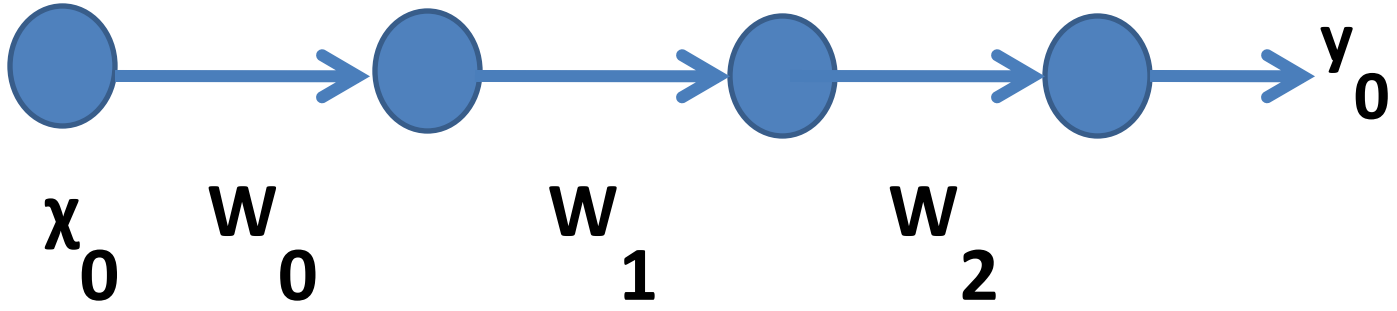
$$\hat{y} = x_0 w_0 + x_1 w_1 + x_2 w_2 = \sum_i x_i w_i$$

$$\begin{pmatrix} w_0 & w_1 & w_2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

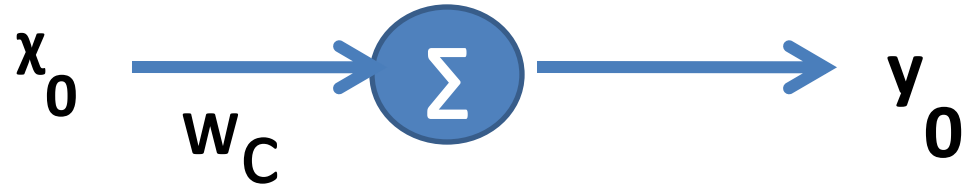
Muti-layer Perceptron



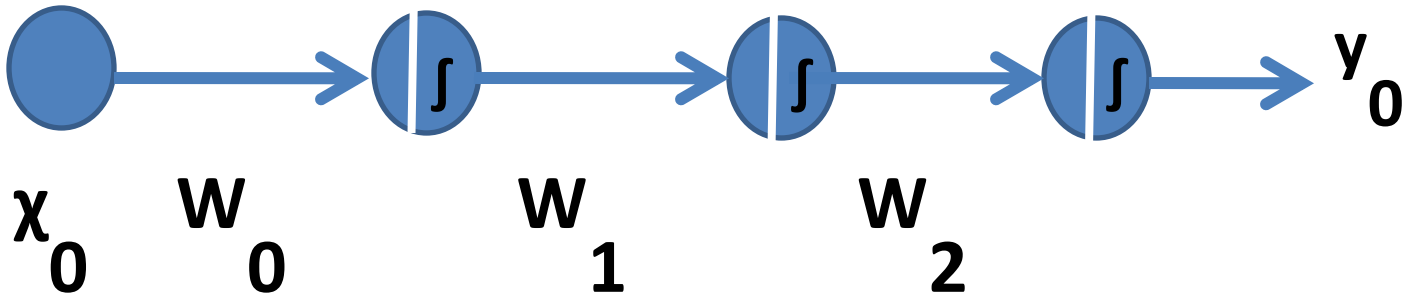
Linear Function



$$y_0 = x_0 w_0 w_1 w_2 = x_0 w_C$$

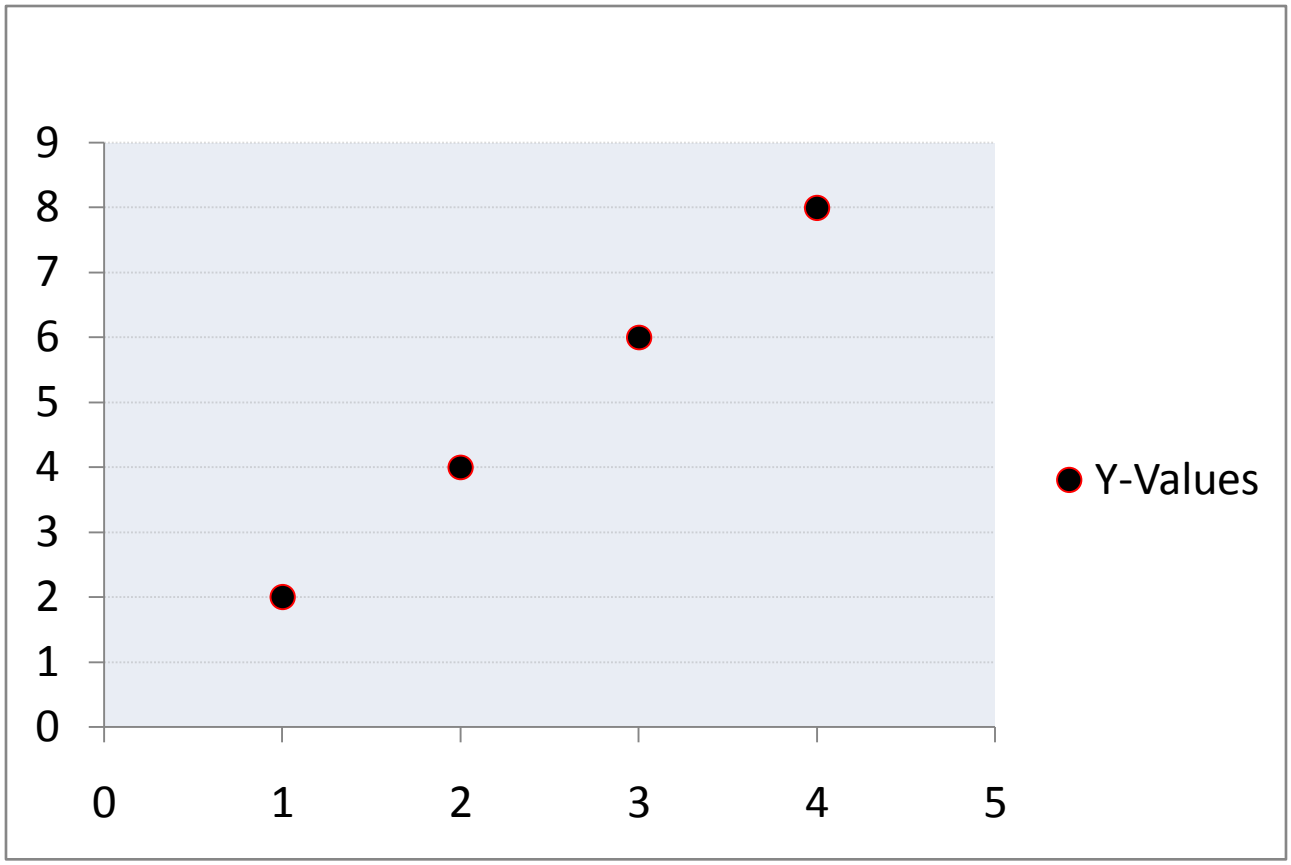


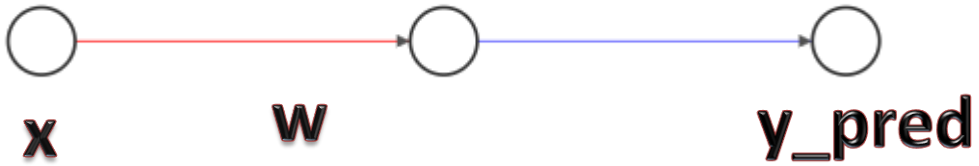
Non-Linear Function



$$y_0 = \sigma(\sigma(\sigma(x_0 W_0) W_1) W_2)$$

Simple Example





y_pred = xw

E = $(\hat{y}-y)^2 = (xw-y)^2$ *sum of squared Error*

$$\frac{\partial \mathbf{E}}{\partial \mathbf{w}} = 2x(xw-y)$$ Derivative

$$\mathbf{w} - \alpha \frac{\partial \mathbf{E}}{\partial \mathbf{w}} = \mathbf{w} - \alpha\, \mathbf{2x(xw-y)}$$ Update Rule

- Random weight initialization w=0.5
- learning rating $\alpha = 0.1$

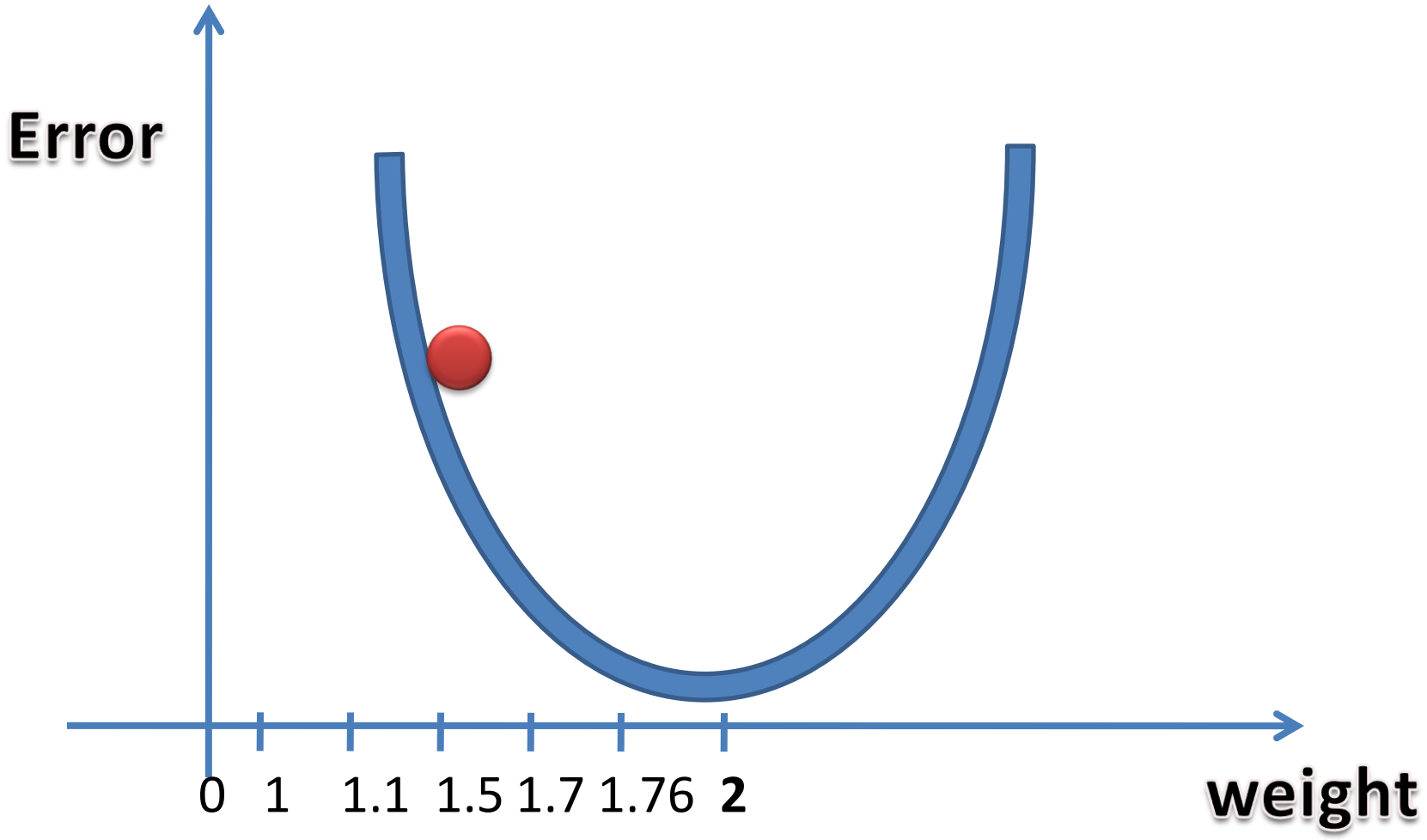
- $w_{new} = w - 0.1 * 2x(xw-y)$
- (x,y)
- (2,4) w=0.5 ← 0.5 - 0.1*2*2(2*0.5-4) = 1.7
- (1,2) w =1.7 ← 1.7-0.1*2*1(1*1.7-2) = 1.76
- (3,6) w= 1.76 ← 1.76-0.1*2*3(3*1.76-6)=2.192
- w ~ 2

Gradient Descent

- Gradient Descent is used while training a machine learning model.
- A gradient measures how much the output of a function changes if you change the inputs a little bit.
- It simply measures the change in all weights with regard to the change in error.

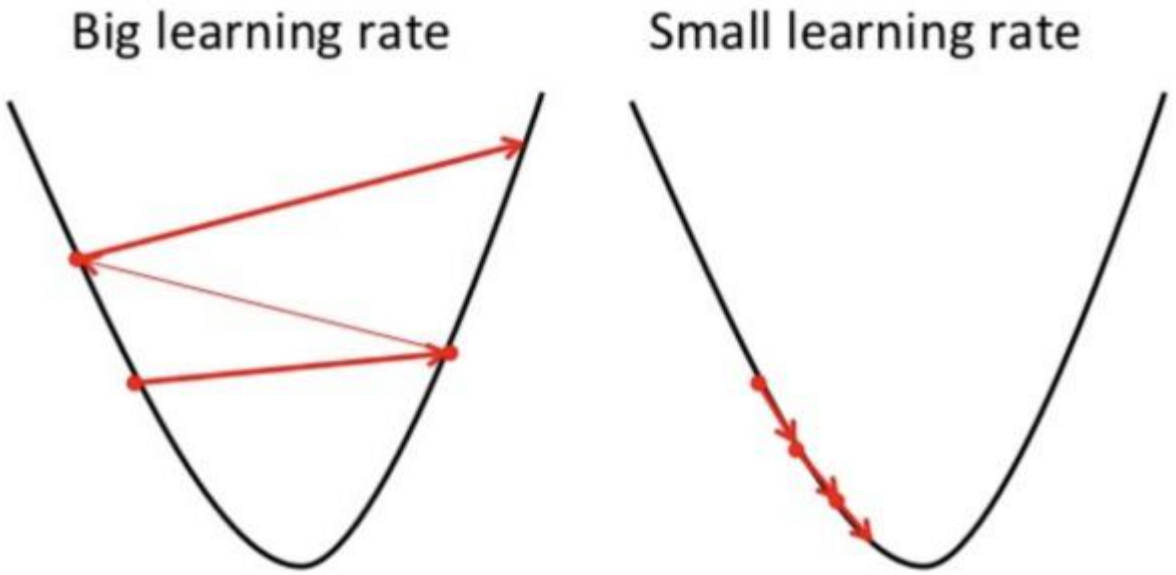
Oversimplified Gradient Descent:

- Calculate slope at current position
- If slope is negative, move right
- If slope is positive, move left
- (Repeat until slope == 0)

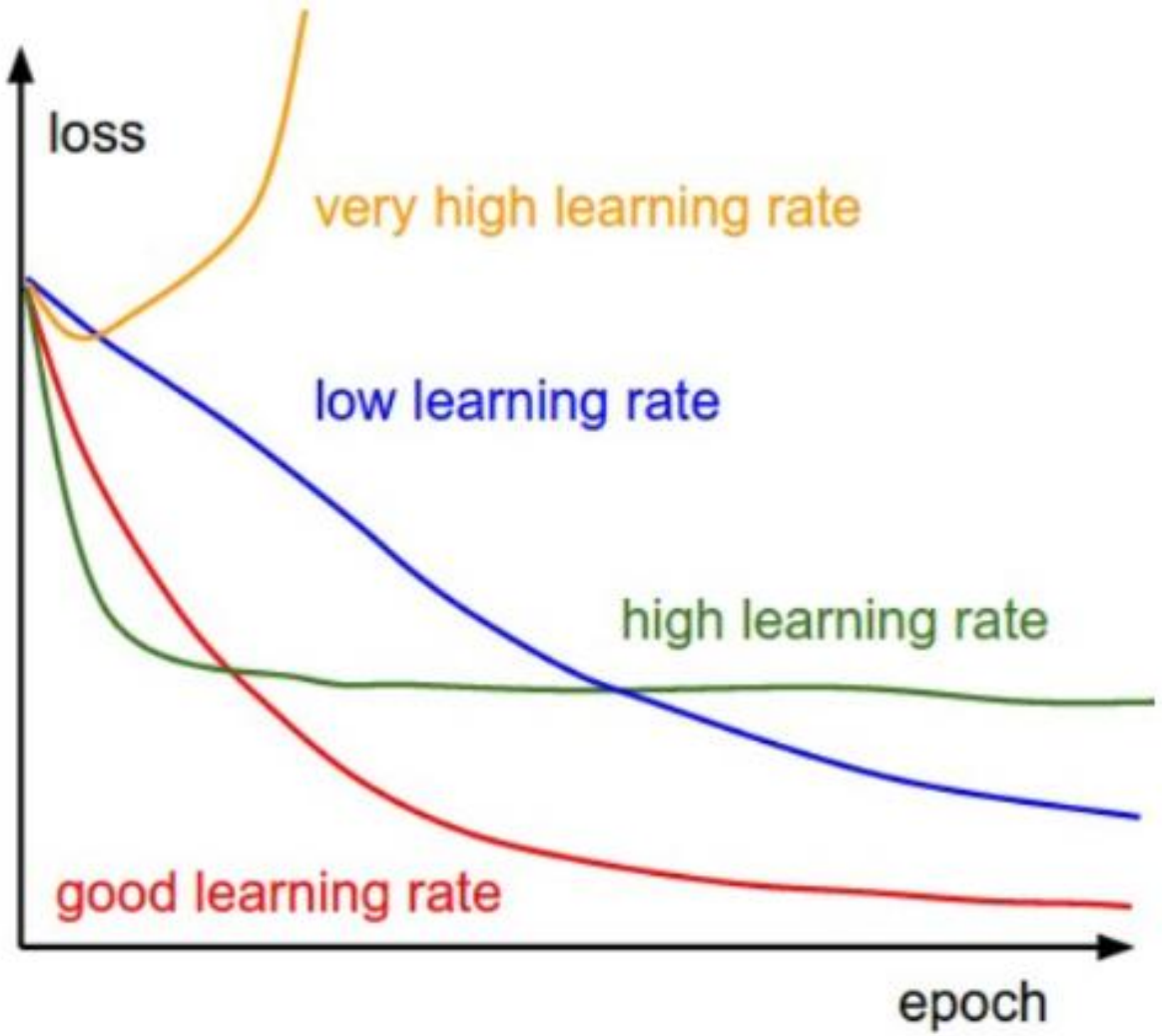


Learning rate

1. **if it is too small, then the model will take some time to learn.**
2. **if it is too large, model will converge as our pointer will shoot and we'll not be able to get to minima.**



- ***Picking an appropriate learning rate can be troublesome. A learning rate that is too low will lead to slow training and a higher learning rate will lead to overshooting of slope.***



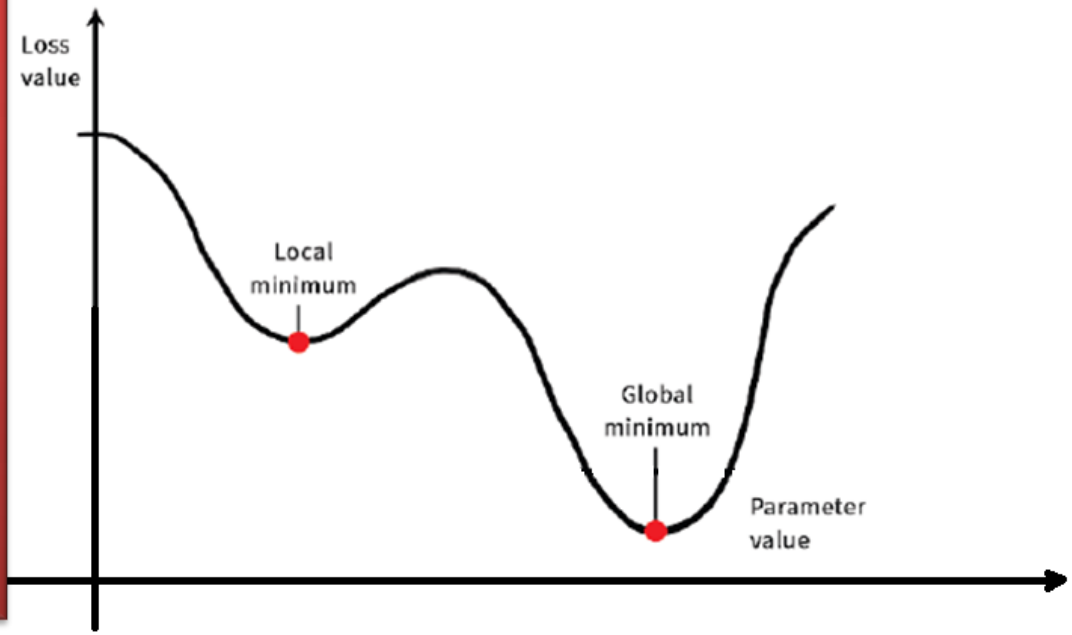
SGD Variants

- There exists multiple variants of SGD that differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients.
- "SGD with momentum", "Adagrad", "RMSprop“ these variants are known as **"optimization methods" or "optimizers"**.

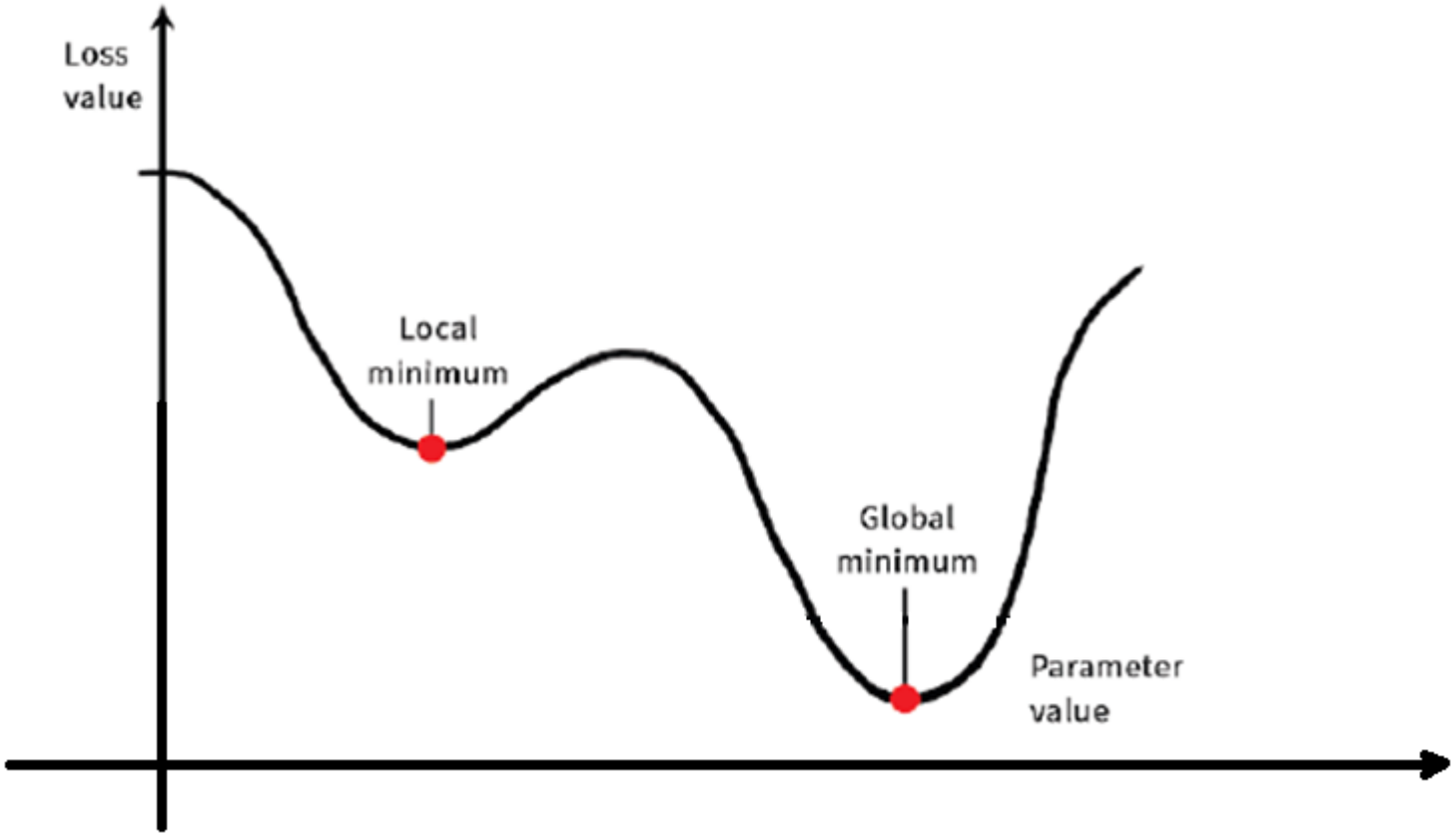
Momentum

- Momentum addresses two issues with SGD: convergence speed, and local minima.

If the parameter considered was being optimized via SGD with a small learning rate, then the optimization process would get stuck at the local minimum, instead of making its way to the global minimum.



Global Minima & Local Minima



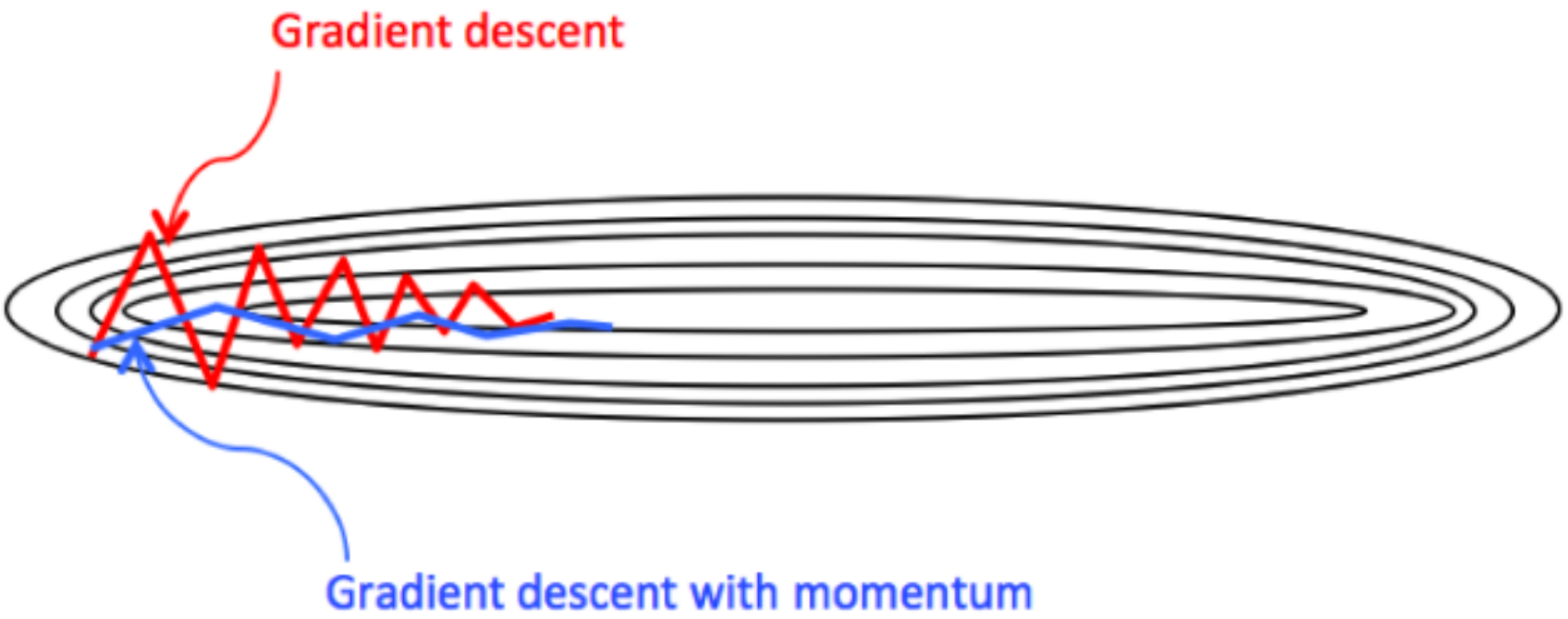
Momentum

- A way to avoid such issues is to use "momentum", which draws inspiration from physics.
- This means updating the parameter **w** not only on the current gradient value but also based on the previous parameter update.

Momentum

past_velocity = 0.
momentum = 0.1 # A constant momentum factor

while loss > 0.01: # Optimization loop
    w, loss, gradient = get_current_parameters()
    velocity = **past_velocity** * **momentum** + learning_rate * gradient
    w = w + **momentum** * **velocity** - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)


Gradient descent
Gradient descent with momentum

Optimizer

- **RMSprop**: Root Mean Square Propagation
- RMSprop is another adaptive method which retains the learning rate for each parameter but uses a moving average over the gradients to make optimization more suited for more non-convex optimization

- **Adam optimizer**: Another algorithm that uses adaptive method. Adam stands for Adaptive momentum estimation, it tends to combine the best part of RMSprop & momentum optimizer

Adaptive Momentum

- Adam method is used to accelerate the gradient descent algorithm by taking into consideration the exponentially weighted average of the gradients.

- **new weight ←(old weight)-(learning rate)(gradient)**
- **new weight ←(old weight)-(learning rate)(gradient) + past gradient**
- **(accumulator)←(old accumulator)(momentum)+gradient**
- **momentum → weighted average of past gradients**
- **new weight ←(old weight)-(learning rate)(accumulator)**

Gradient descents Strategies

- **Stochastic Gradient Descent:** Parameters are updated after computing the gradient of error with respect to a single training example

- **Batch Gradient Descent:** Parameters are updated after computing the gradient of error with respect to the entire training set

- **Mini-Batch Gradient Descent:** Parameters are updated after computing the gradient of error with respect to a subset of the training set

Stochastic Gradient Descent

- ***Instead of going through all examples, Stochastic Gradient Descent (SGD) performs the parameters update on each example . Therefore, learning happens on every example.***

- **Advantages : —**

  a. Easy to fit in memory
  b. Computationally fast
  c. Efficient for large dataset

- **Disadvantages :-**

  a. Due to frequent updates steps taken towards minima are very noisy.
  b. Noise can make it large to wait.
  c. Frequent updates are computationally expensive.

Batch Gradient Descent

- ***Batch Gradient Descent is a greedy approach, when we add all examples on each iteration when performing the updates to the parameters. Therefore, for each update, we have to sum over all examples.***

Advantages :-

a. Less noisy steps
b. produces stable GD convergence.
c. Computationally efficient as all resources aren't used for single sample but rather for all training samples

Disadvantages :-

a. Additional memory might be needed.
b. It can take long to process large database.
c. Approximate gradients

Mini Batch Gradient Descent

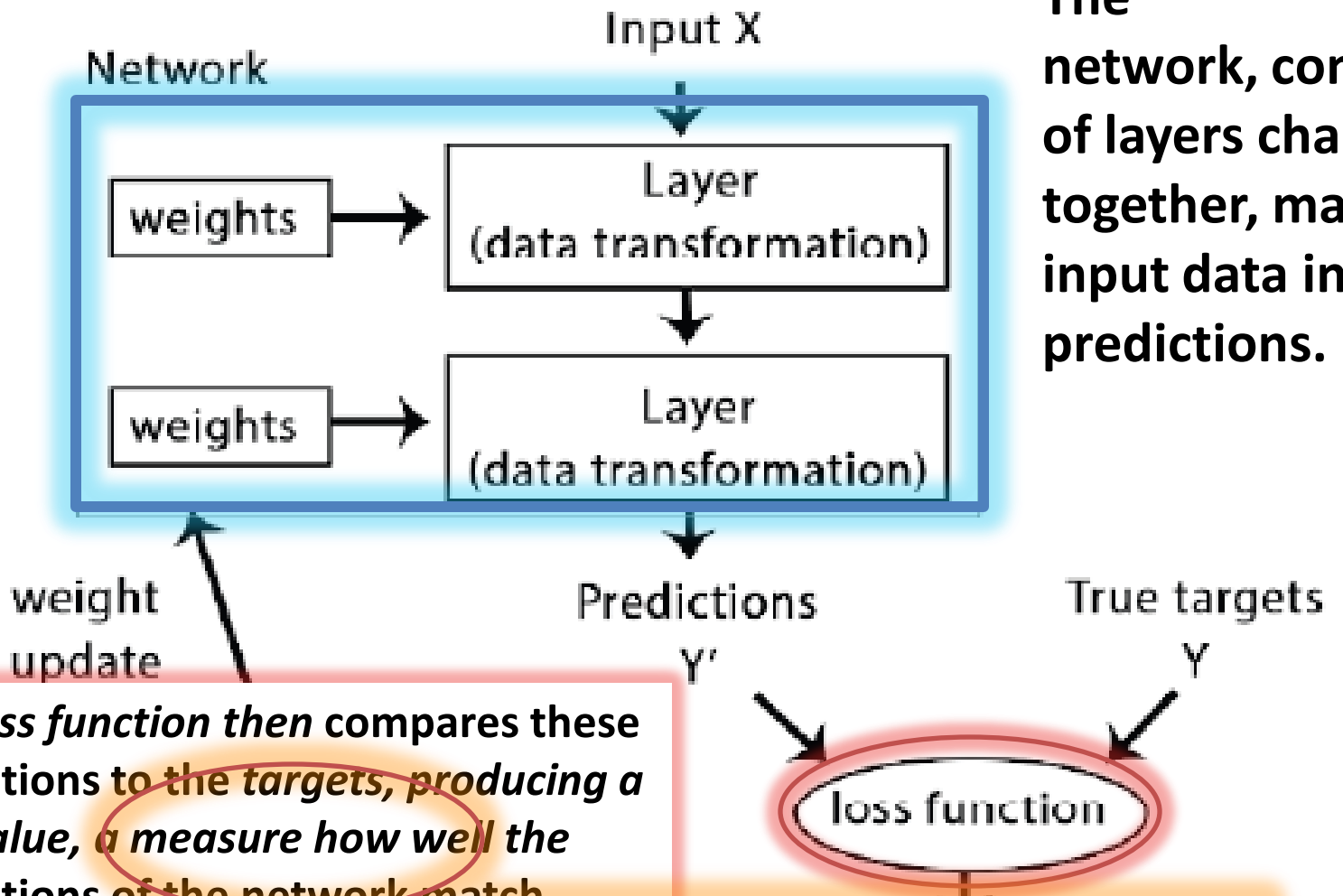
- ***Instead of going over all examples, Mini-batch Gradient Descent sums up over lower number of examples based on the batch size. Therefore, learning happens on each mini-batch of b examples.***

Advantages :-

a. Easy fit in memory.
b. Computationally efficient.
c. Stable error go and convergence.

Anatomy of a neural network

- Training a neural network revolves around the following objects:
- *Layers, which are combined into a network (or model).*
- The *input data and corresponding targets.*
- The *loss function, which defines the feedback signal which is used for learning.*
- The *optimizer, which determines how the learning proceeds.*



The network, composed of layers chained together, maps the input data into predictions.

The *loss function then* compares these predictions to the *targets, producing a loss value, a measure how well the* predictions of the network match what was expected.

The *optimizer uses this loss value* to update the weights of the network.

Activation Function

- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it.

- The purpose of the activation function is to *introduce non-linearity* into the output of a neuron.

Activation Function

- Neural network has neurons that work in correspondence of *weight, bias* and their respective activation function.
- In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output.
- This process is known as *back-propagation*.
- Activation functions make the back-propagation possible

Activation Function

- **Sigmoid Function :**
- Usually used in output layer of a binary classification, where result is either 0 or 1
- As value for sigmoid function lies between 0 and 1 only
- Result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

Activation Function

- **tanh() Function :**
- Hyperbolic tangent function (also known as tanh)
- The hyperbolic tangent function outputs in the range (-1, 1), thus mapping strongly negative inputs to negative values.

Activation Function

- **RELU (***Rectified linear unit***)**

- It is the most widely used activation function.

- ReLu is less computationally expensive because it involves simpler mathematical operations.

Regression Loss Functions

- **Mean Squared Error Loss**
  - ❖ Mean squared error is calculated as the average of the squared differences between the predicted and actual values.
  - ❖ ‘mean_squared_error‘
- **Mean Absolute Error Loss**
  - ❖ ‘mean_absolute_error‘

Binary Loss Functions

- **Binary Cross-Entropy Loss**
  - ❖ It is intended for use with binary classification where the target values are in the set {0, 1}
  - ❖ '*binary_crossentropy*'
- **Categorical crossentropy**
  - ❖ A loss function that is used in multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one.
  - ❖ Softmax is the only activation function recommended to use with the categorical crossentropy loss function.

Keras

- Keras, the Python deep learning library
- one of the most widely used deep learning frameworks
- A big part of that success is that Keras has always put ease of use and accessibility front and center.

Keras

- Keras has the following key features:

✓ It allows the same code to run on CPU or on GPU, seamlessly.

✓ It has a user-friendly API which makes it easy to quickly prototype deep learning models.

✓ It has build-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.

Keras

- There are two ways to define a model
- Using the “**Sequential class** “

  (only for linear stacks of layers, which is the most common network architecture by far)

- Using the **"functional API“**

  (for directed acyclic graphs of layers, allowing to build completely arbitrary architectures)

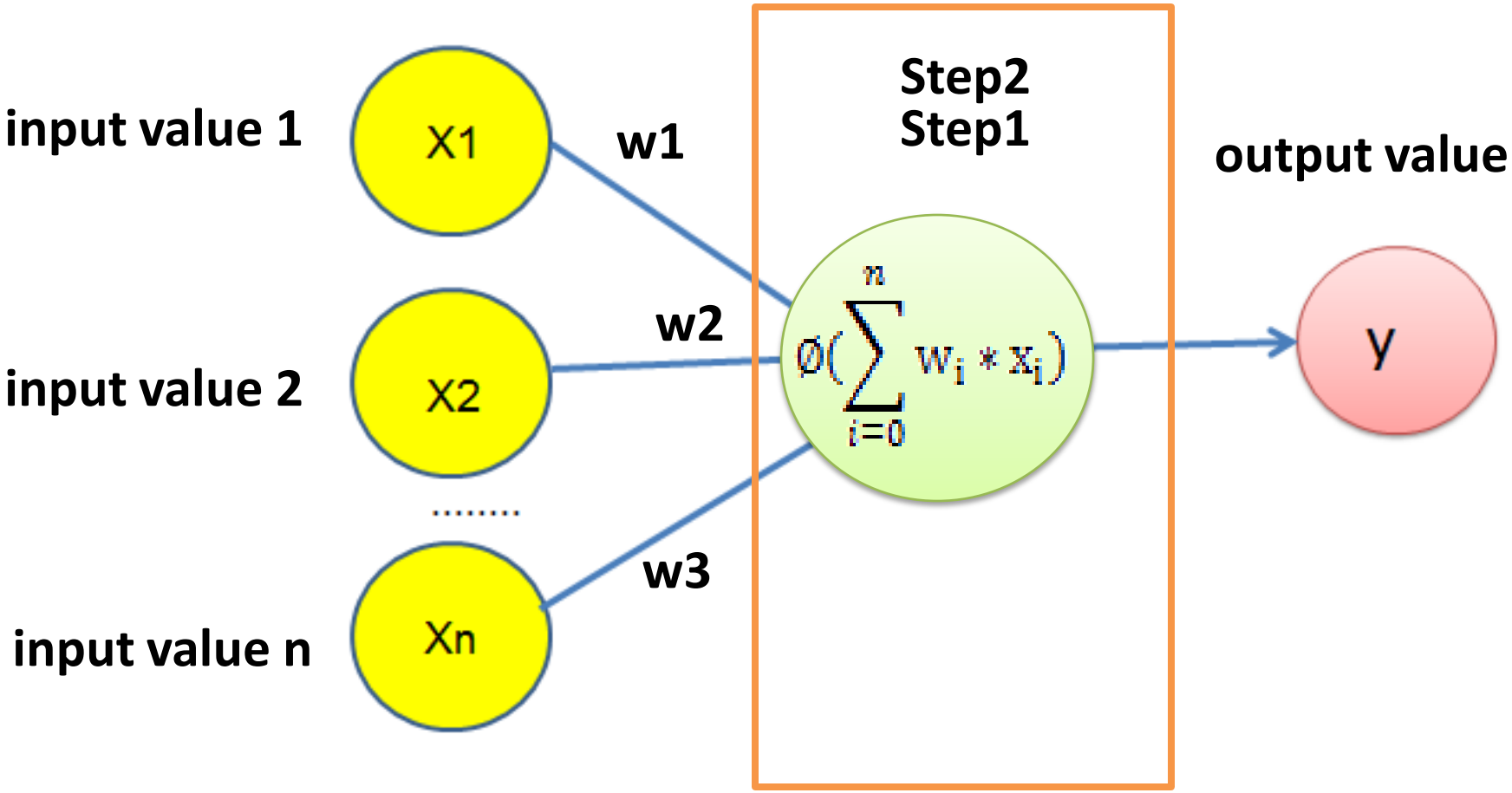
A network definition using the Sequential model

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

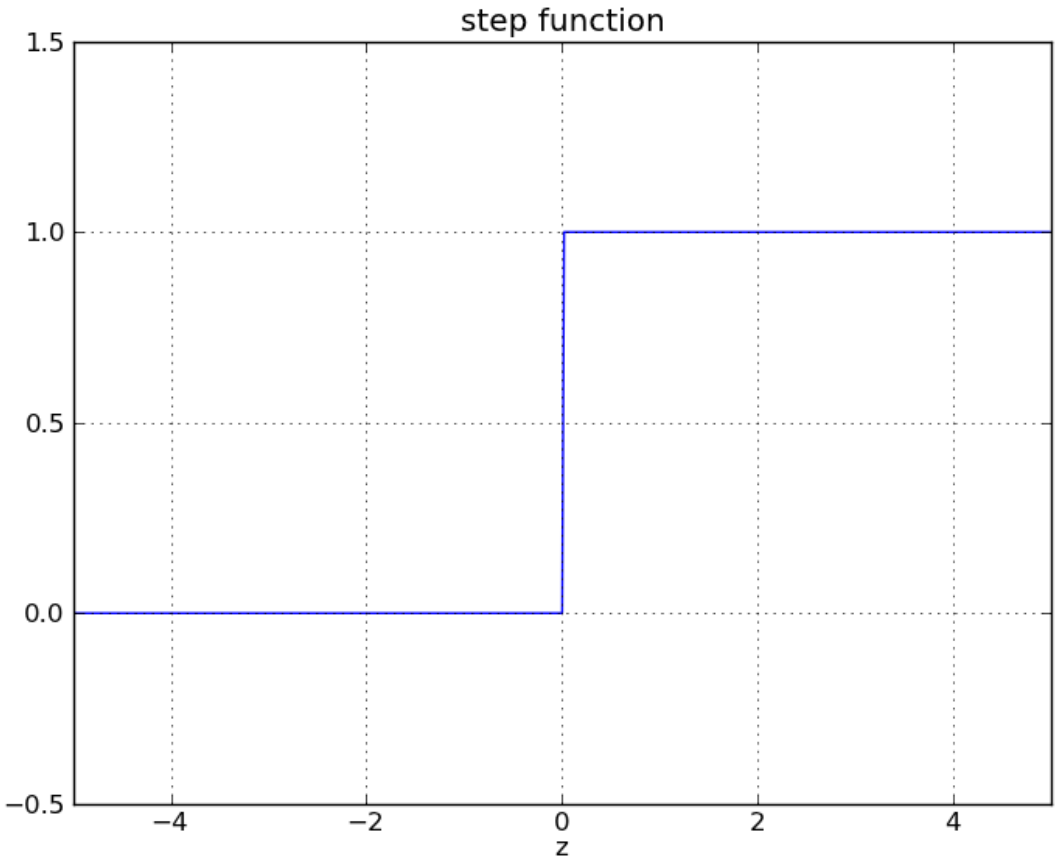
A network definition using the functional API

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(input=input_tensor, output=output_tensor)
```

ANN


input value 1
input value 2
input value n
X1
X2
Xn
w1
w2
w3
Step2
Step1
$\emptyset(\sum_{i=0}^{n} w_i * x_i)$
output value
y

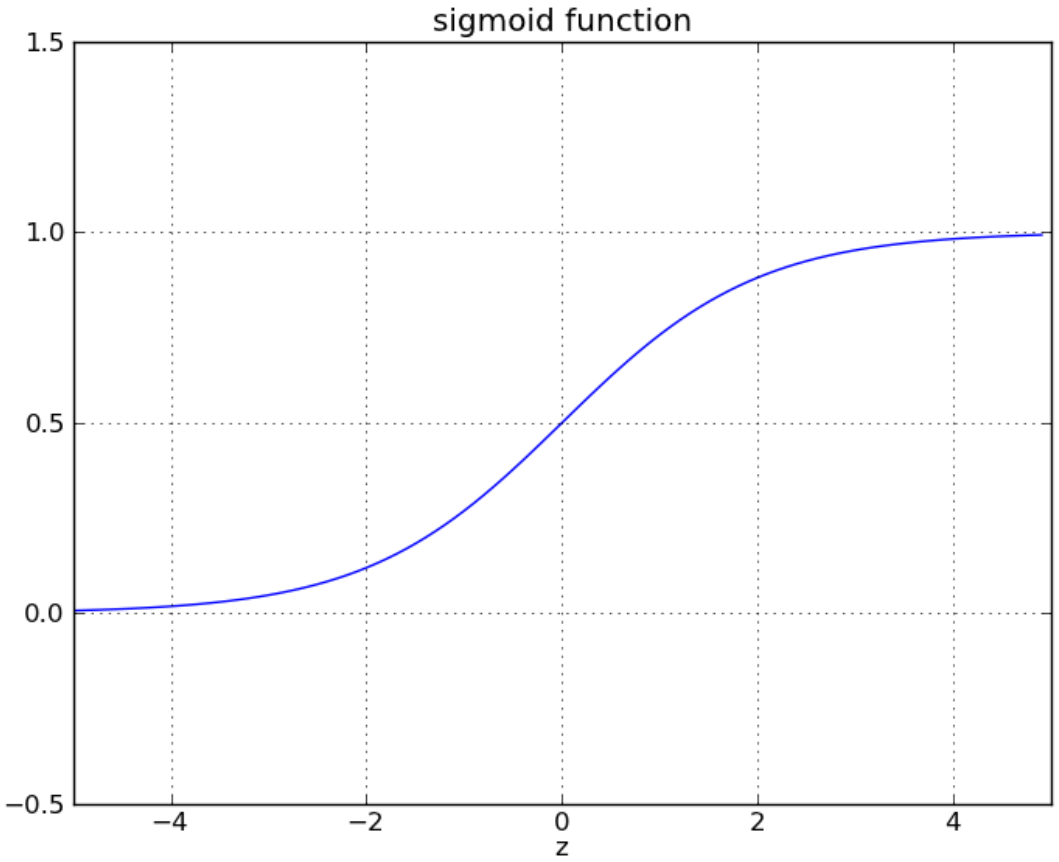
Threshold Function



$$\phi(x) = \begin{cases} 1 \text{ if } x >= 0 \\ 0 \text{ if } x < 0 \end{cases}$$

$$\sum_{i=0}^{n} w_i * x_i$$

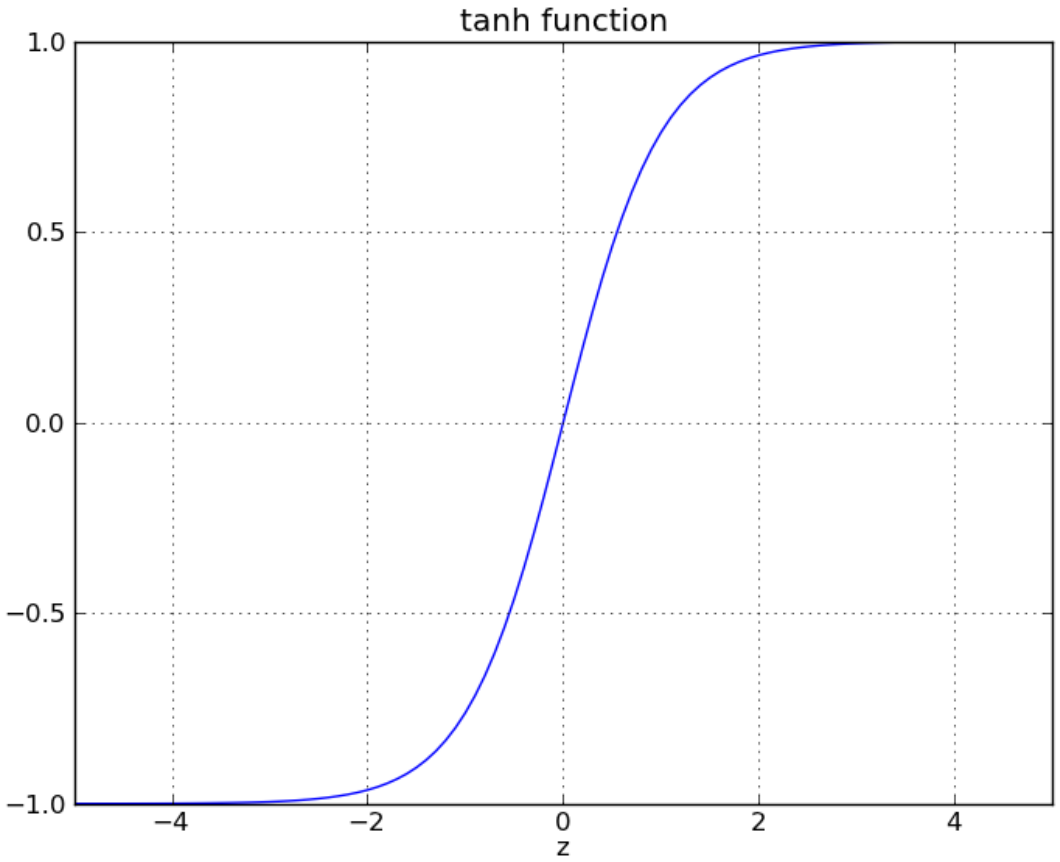
Sigmoid



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

$$\sum_{i=0}^{n} w_i * x_i$$

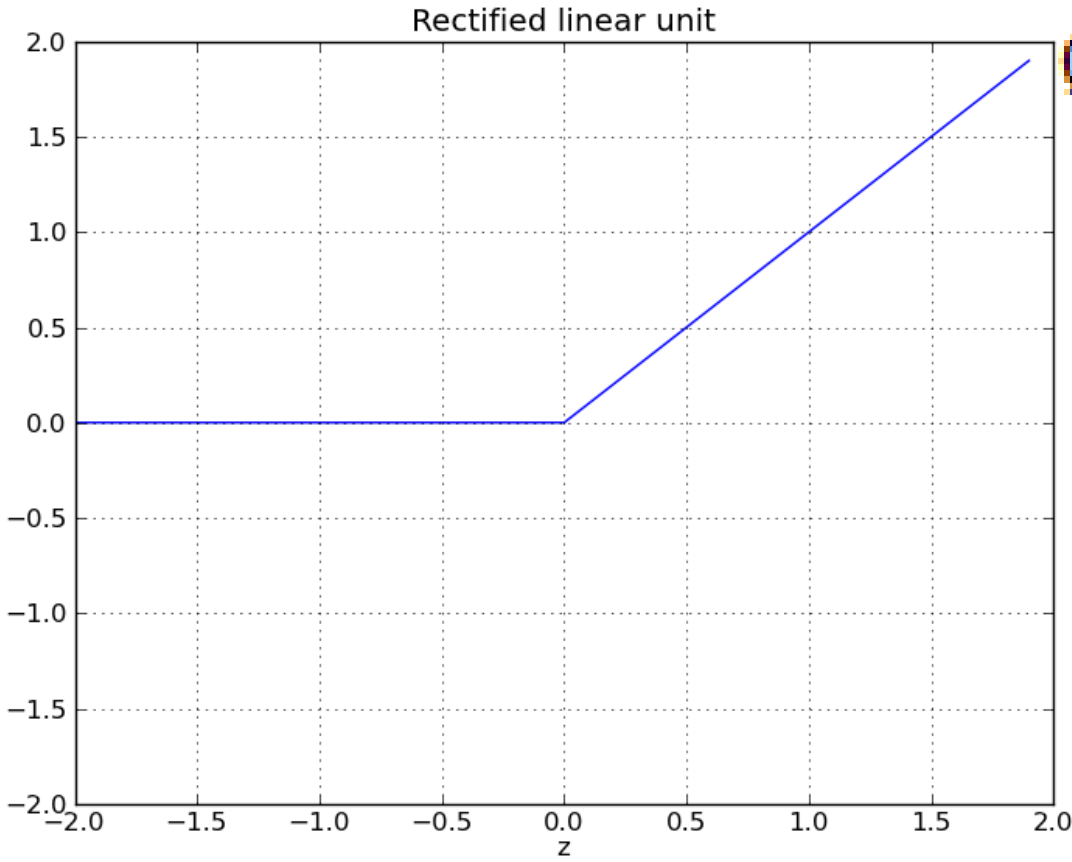
tanh



$$\emptyset(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$\sum_{i=0}^{n} w_i * x_i$$

ReLu



$\emptyset(x)$ = *max(x,0)*

$$\sum_{i=0}^{n} w_i * x_i$$

Overfitting

- When the accuracy of our model on the validation data would peak after training for a number of epochs, and would then start decreasing. In other words, our model would *overfit* to the training data
- It's often possible to achieve high accuracy on the *training set*, what we really want is to develop models that generalize well to a *testing set* (or data they haven't seen before).

Underfitting

- The opposite of overfitting is *underfitting*.
- If the model is not powerful enough, or has simply not been trained long enough.
- This means the network has not learned the relevant patterns in the training data.

Overfitting

- If we train for too long though, the model will start to overfit and learn patterns from the training data that don't generalize to the test data
- We need to strike a balance

Prevent overfitting

- To prevent overfitting, the best solution is to use more training data
- A model trained on more data will naturally generalize better.
- When that is no longer possible, the next best solution is to use techniques like regularization.

Regularization

- These place constraints on the quantity and type of information your model can store.
- If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

Prevent overfitting

- The simplest way to prevent overfitting is to reduce the size of the model, i.e. the number of learnable parameters in the model (which is determined by the number of layers and the number of units per layer).
- Deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

Prevent overfitting

- There is no magical formula to determine the right size or architecture of your model (in terms of the number of layers, or the right size for each layer).
- We will have to experiment using a series of different architectures.
- To find an appropriate model size, it's best to start with relatively few layers and parameters, then begin increasing the size of the layers or adding new layers until you see diminishing returns on the validation loss.

Regularization Techniques

- Two common regularization techniques—**weight regularization** and **dropout**

Weight regularization

- A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular".
  - ***This is called "weight regularization“***

Weight regularization

- It is done by adding to the loss function of the network a cost associated with having large weights.
- This cost comes in two flavors:

➢ L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients

➢ L2 regularization, where the cost added is proportional to the square of the value of the weights coefficients

Weight regularization

- keras.layers.Dense(16,

kernel_regularizer=keras.regularizers.l2(0.001),

activation="relu", input_shape=(10,))

l2(0.001) means that every coefficient in the weight matrix of the layer will add

0.001 * weight_coefficient_value2**

to the total loss of the network.

Dropout

- Dropout is one of the most effective and most commonly used regularization techniques for neural networks
- Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training.

Dropout

- The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5.

```
keras.layers.Dense(16, activation=tf.nn.relu,
input_shape=(NUM_WORDS,))
    keras.layers.Dropout(0.5)
```

Summary

- The most common ways to prevent overfitting in neural networks:

- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.