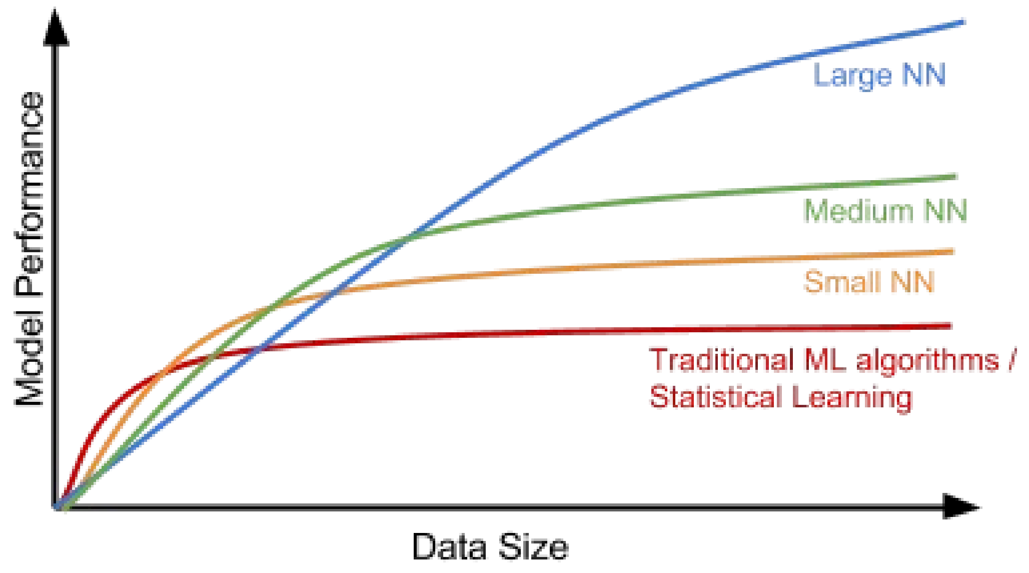
Slides for session 1

3546 Deep Learning

Module 1 introduction

- Learning Outcomes
- Describe the current applications of Deep Learning and propose new applications
- Recognize and discuss the moral issues of AI
- Recall key concepts from the Machine Learning course that you will need for this course
- The basic structure of a neural network model
- The anatomy of a neuron, including various activation functions
- Neural network training techniques like weight initialization and backpropagation
- Use Information Theory to reason about the quality of a model



introduction

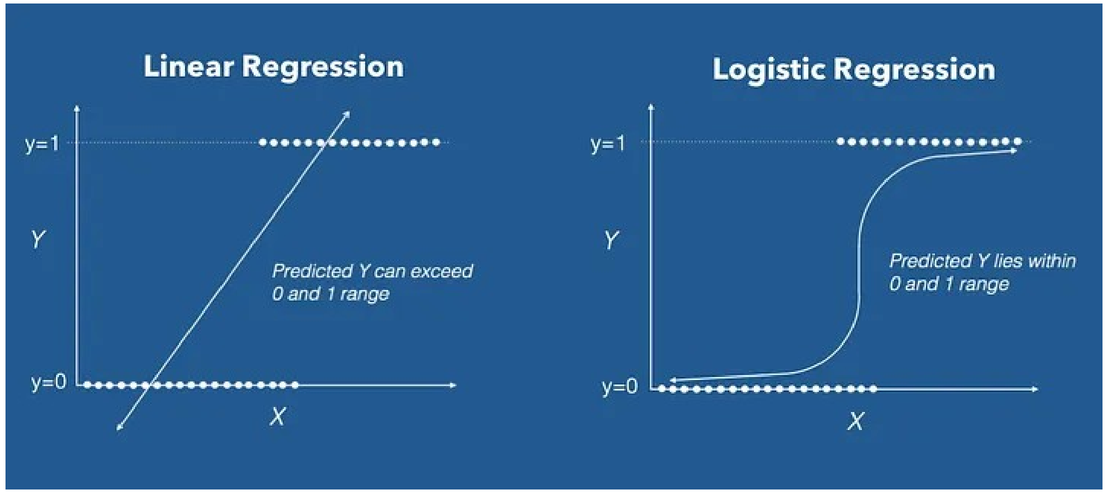
- Main reasons why this particular group of algorithms is taking off:
    - Explosion of data
    - Advanced computations
    - New algorithms

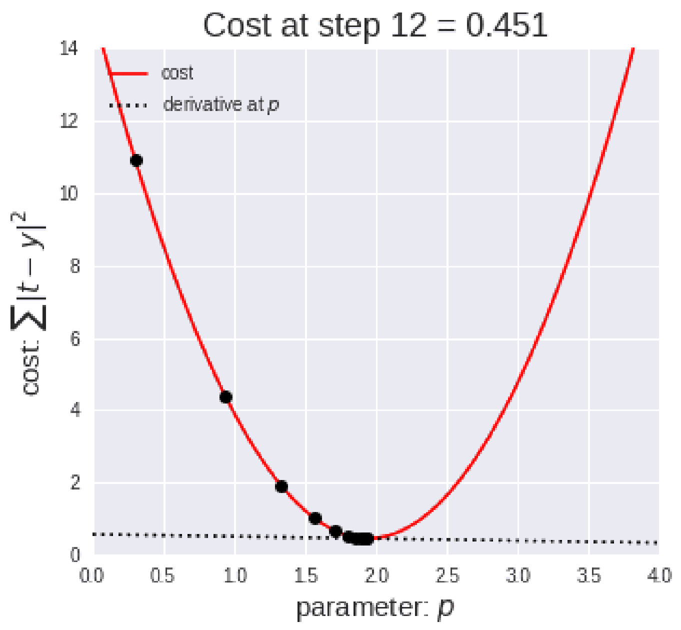
introduction

- **Big Data:**
- outperform traditional ML techniques when the size of the data increases
-
- **Advanced Computation:**
- the development of computers to handle and perform computations on big data.
- Things like GPUs, TPUs, distributed computing, and cloud technologies that have democratized the field a
-
- **Algorithm Development:**
- Flexible nature of how neural networks are constructed
- New approaches and applications of neural networks are constantly being developed and tested.
- Much of the cutting-edge techniques and research is made available to the public
- Like smart phones and the Internet of Things.

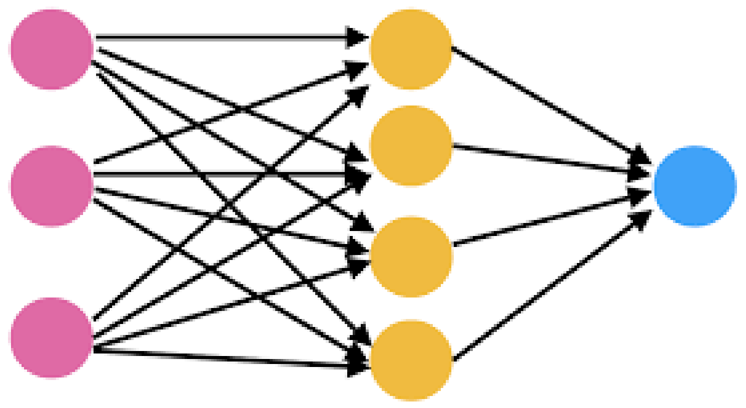
Recap of relevant ML concepts

- Cost function:
  - Can also be called the error function
- How do we reduce the error value?
  - Gradient Descent
- Logistic regression:
  - classifier to give us a set of outputs or classes based on probability
  - when we pass the inputs through a prediction function (sigmoid) and returns a probability score between 0 and 1


Linear Regression
y=1
Y
y=0
X
Predicted Y can exceed 0 and 1 range
Logistic Regression
y=1
Y
y=0
X
Predicted Y lies within 0 and 1 range



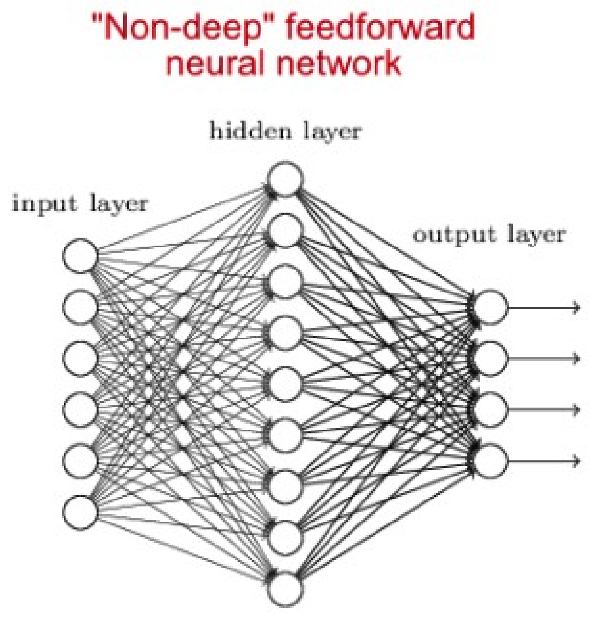
Basics of Neural Networks

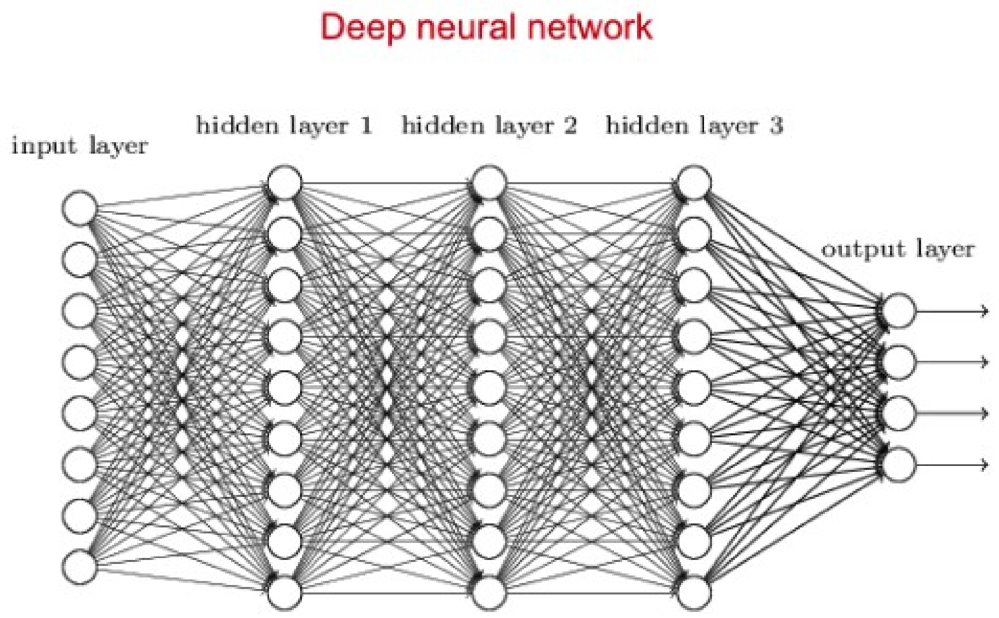


- An Artificial Neural Network (ANN) - or 'neural network'
- A computational model
- Inspired by the way biological neural networks in the human brain process information

Basics of Neural Networks

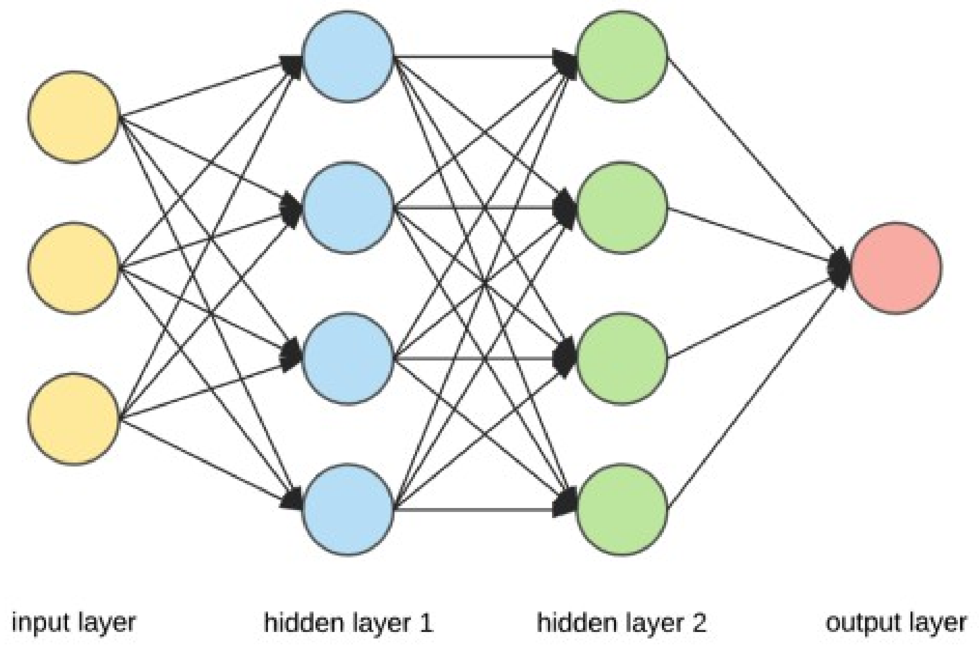
Neural Network versus Deep learning





- A neural network -> an input layer, a hidden layer, and an output layer
- Deep learning -> several hidden layers

Neural Network: Components



Neurons: core processing units of the network

Input layer: receives info

Output layer: predicts the final output

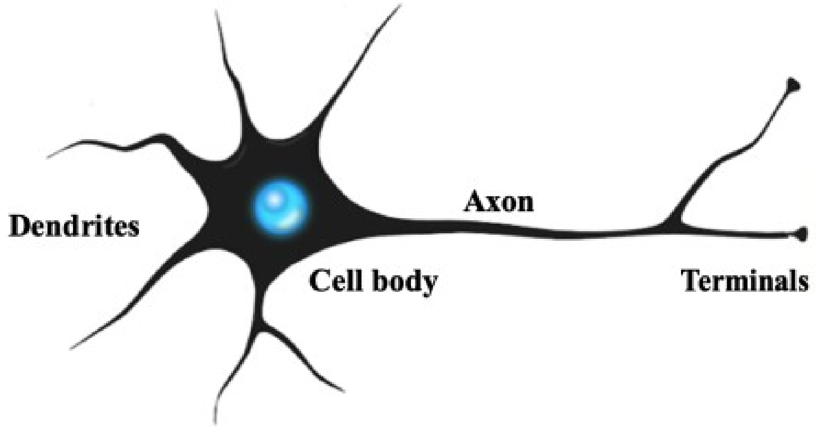
Hidden layers: perform most of the computations

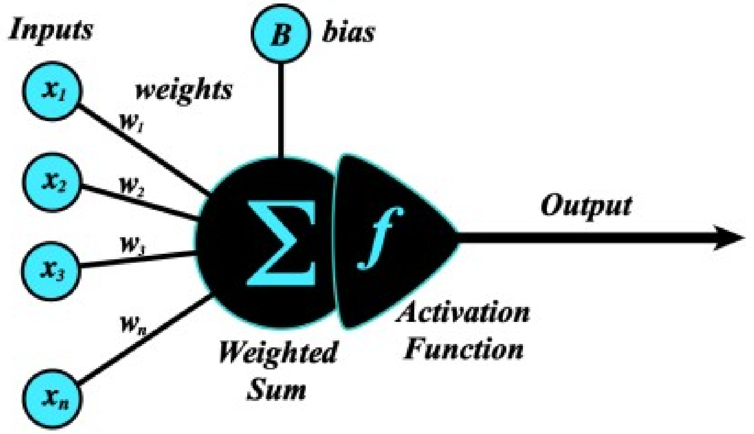
Layer: the fundamental building block for deep learning, consisting of a set of nodes working together

Fundamental component

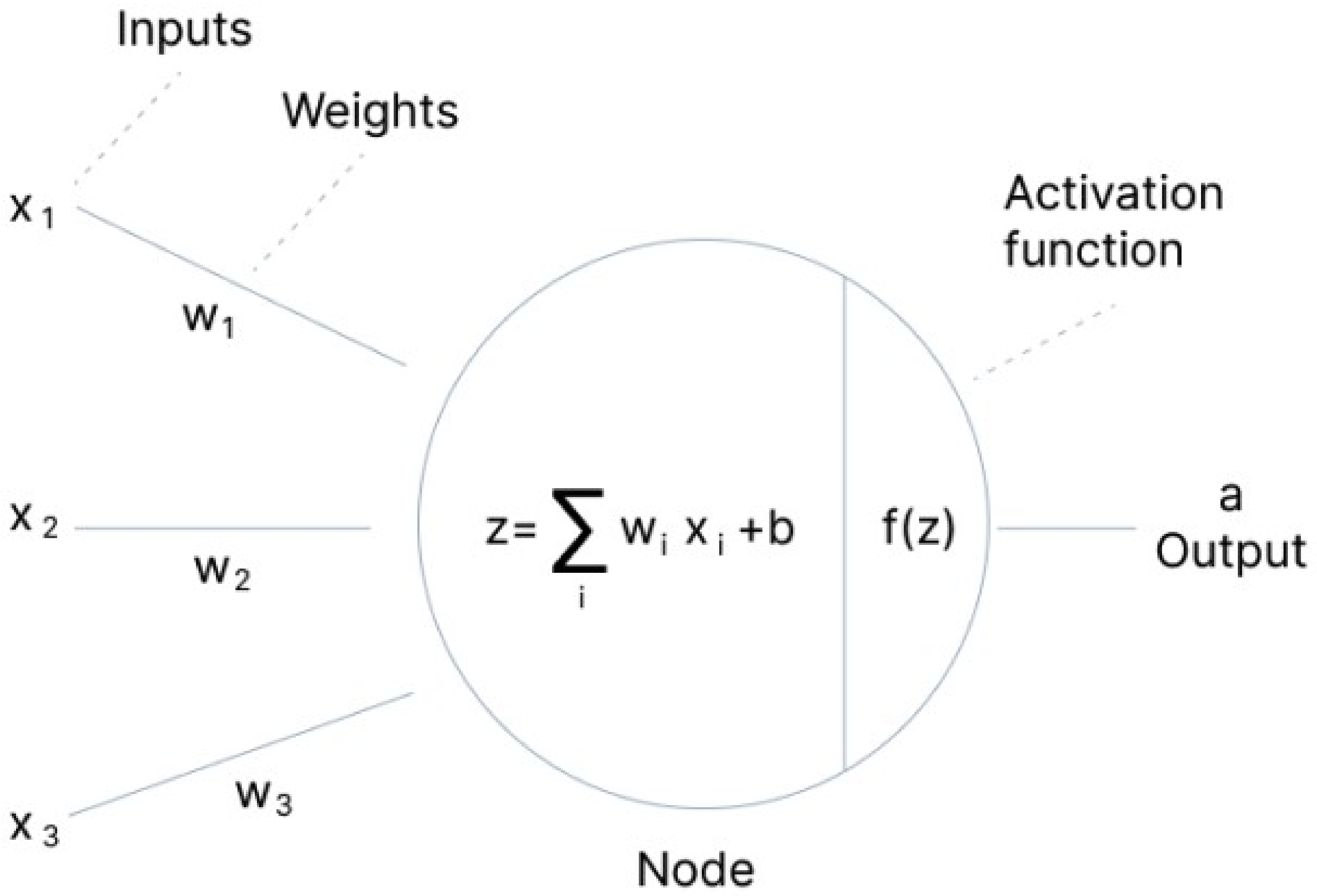
Neurons:

A biological nerve cell
receives input stimuli from neighboring nerves through its ***dendrites***.
If the sum of these stimuli is sufficient to create membrane depolarization in the neuron's ***cell body***
an electrical output signal will be transmitted down the ***axon*** to its ***terminals***





Activation function



Activation function

- Activation functions are used at the end of a hidden unit to introduce non-linear complexities to the model

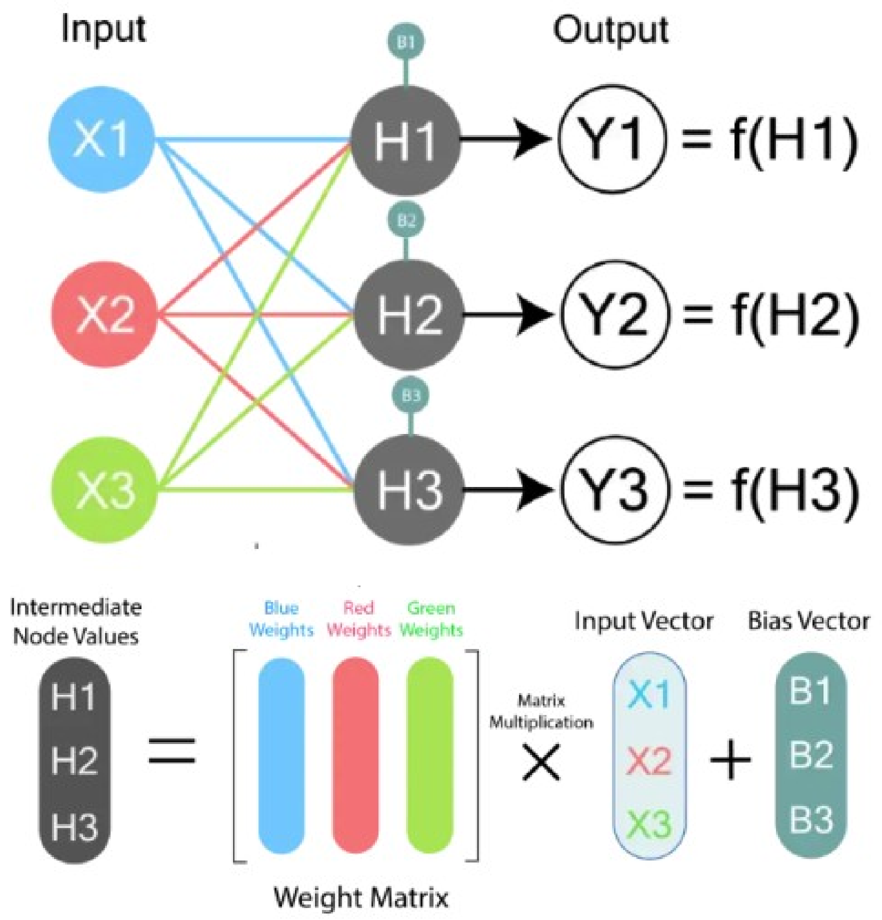
  Linear calculation + activation function

  Linear calculation (prior to activation)

- Involves multiplying the weight vector by the input vector and adding a bias to produce the output or Y value
- Each neuron performs what is essentially a linear regression calculation

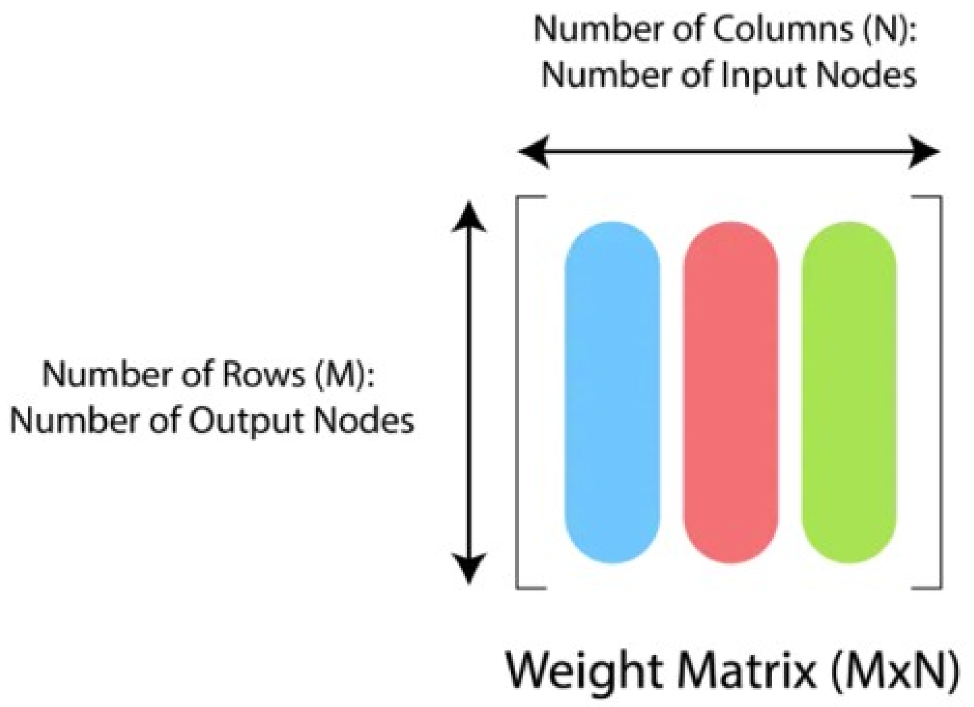
Basics of Neural Networks

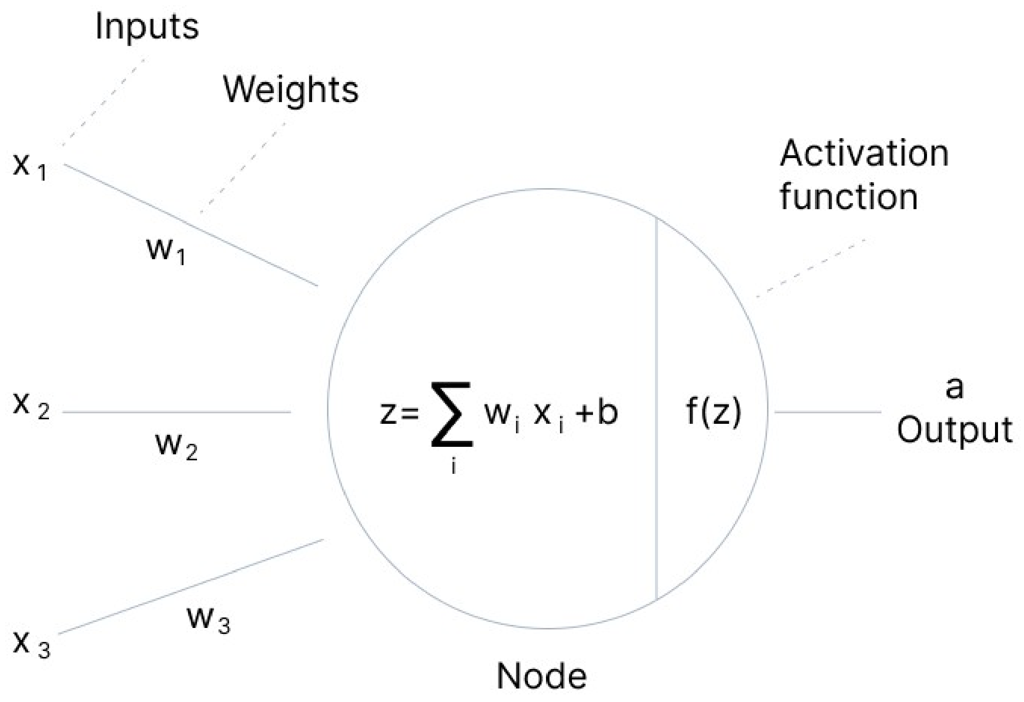
- Weights (corresponding to each input node) are vectors and arranged horizontally



Basics of Neural Networks

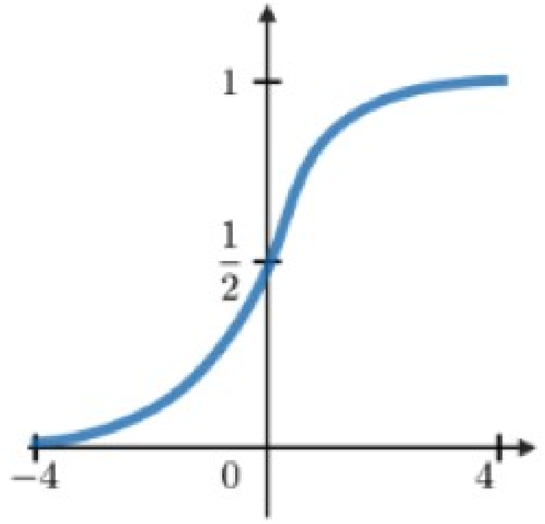
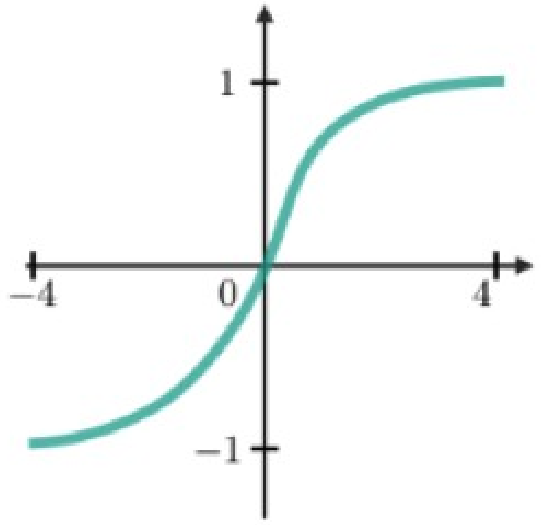
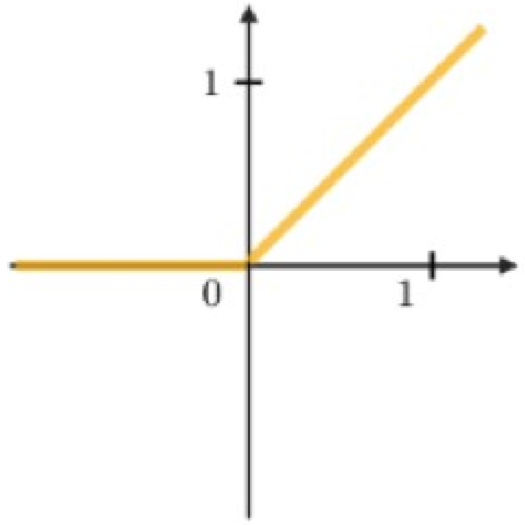
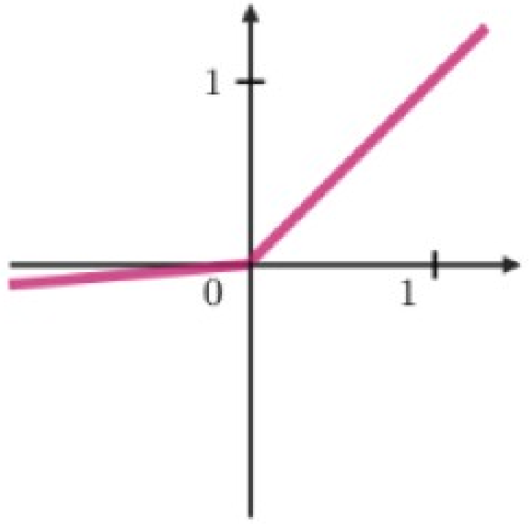
Weight Matrix:



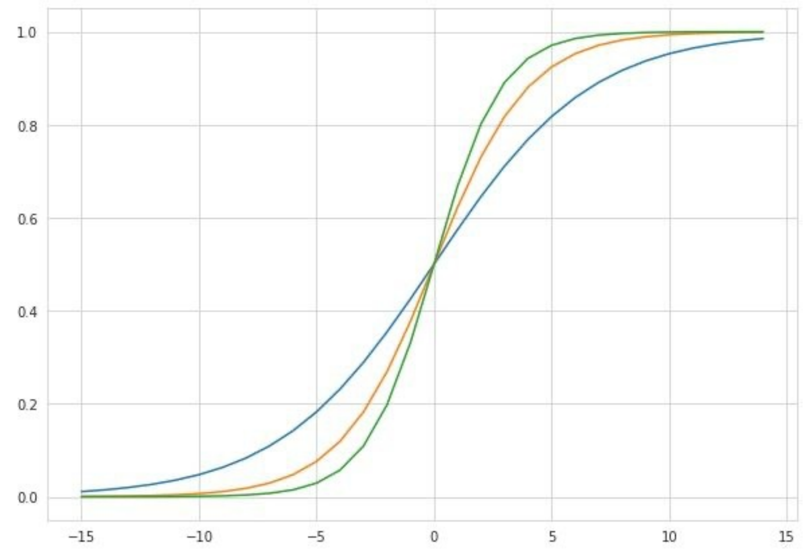


- This linear output (**linear activation**) is then fed through the activation function
- The purpose of the activation function is to introduce nonlinearities between units
- Without this, the output of all neuron groups would simply be linear combinations of the others

Common activation functions

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

Basics of Neural Networks



Bias

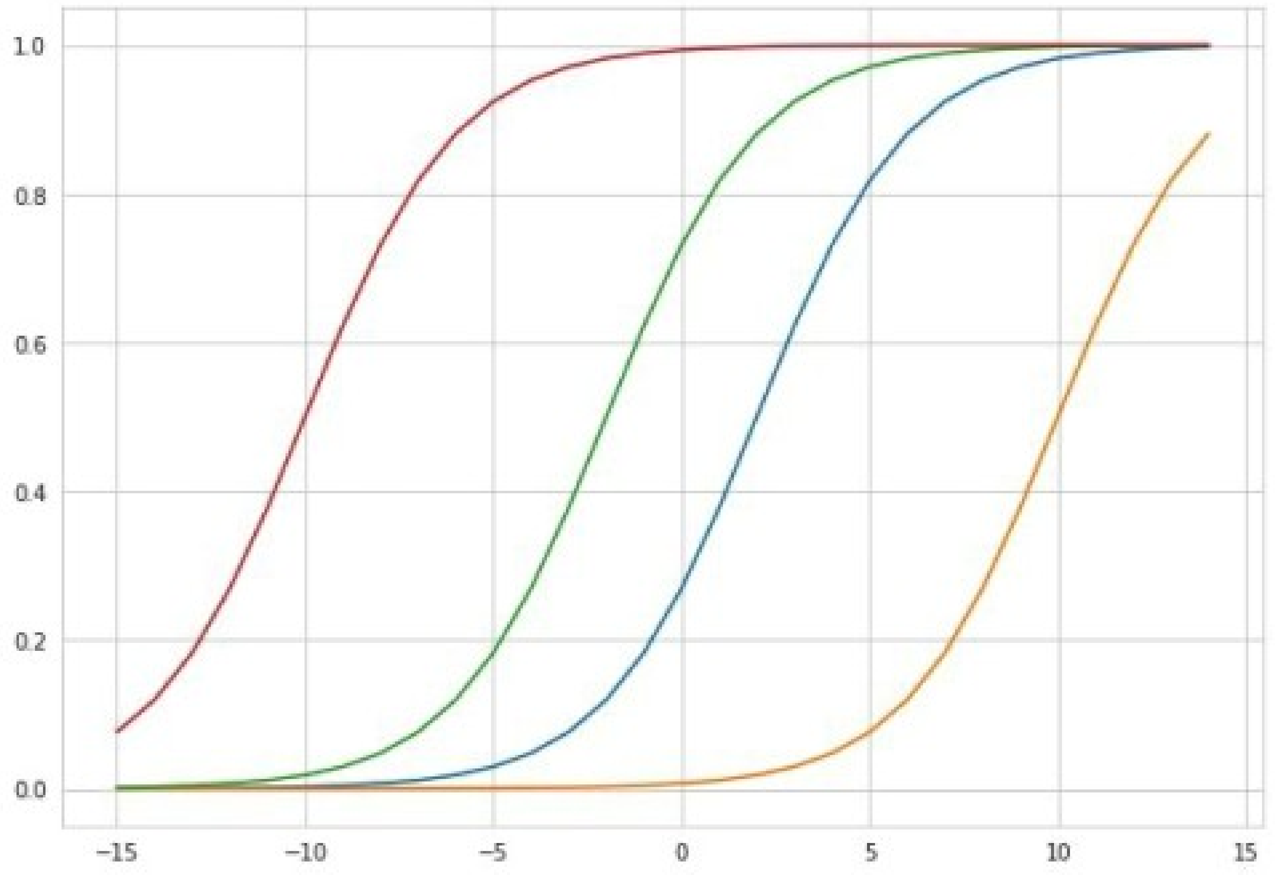
$$sigmoid\ function = \frac{1}{1 + e^{-(w*x+b)}}$$

- the constant which is added to the product of features and weights
- It is used to offset the result
- It helps the models to shift the activation function towards the positive or negative side

On changing the values of 'w', only the steepness of the curve will change

Bias

There is only one way to shift the origin and that is to include bias ‘b’

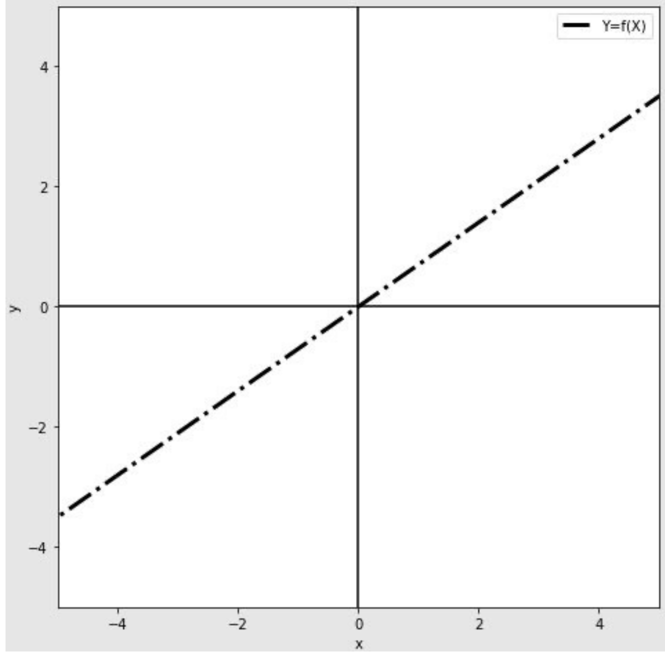


Bias

A given neural network computes the function Y=f(X)

X = feature vector

Y = output vector

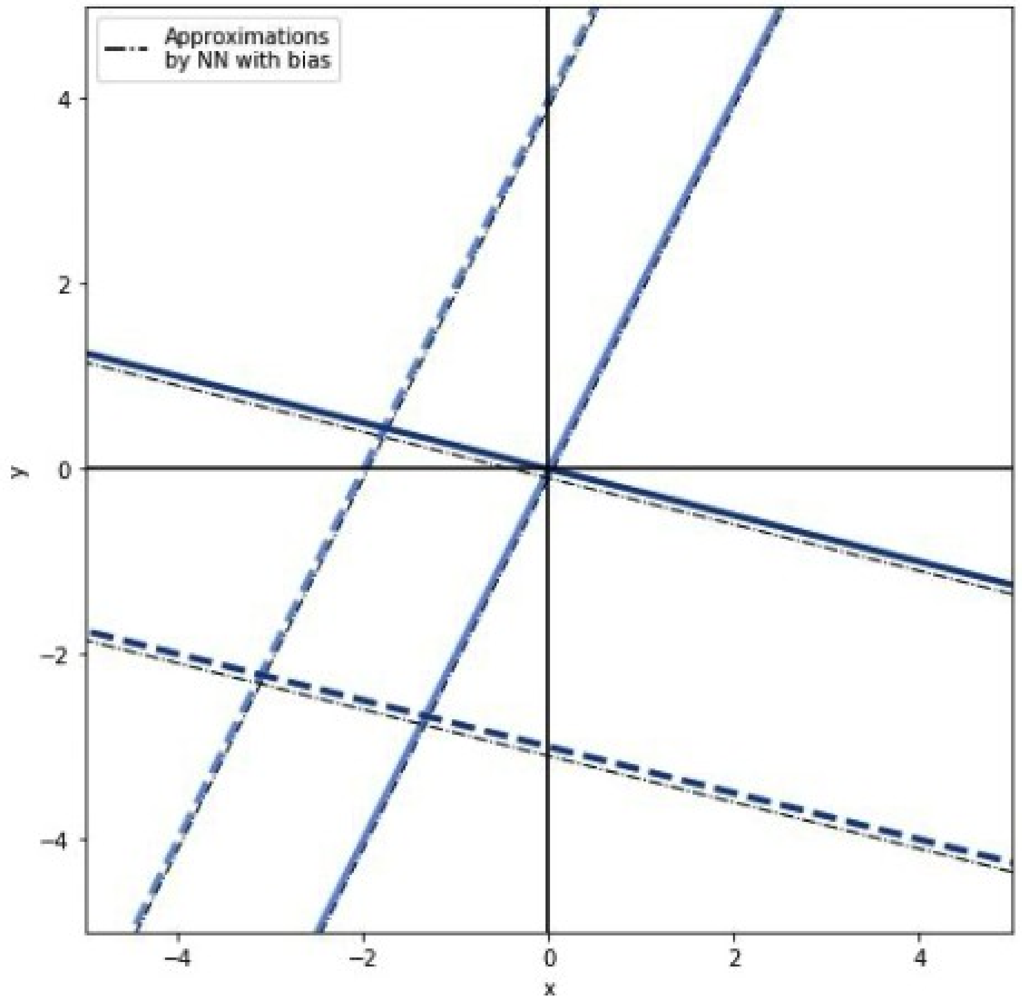


- If the given neural network has weight 'W' then it can also be represented as Y=f(X,W)
- If the dimensionality of both X and Y is equal to 1, the function can be plotted in a two-dimensional
- Such a neural network can approximate any linear function of the form y=mx + c
- **If c=0**
  - y=f(x)=mx
  - the neural network can approximate only the functions passing through origin

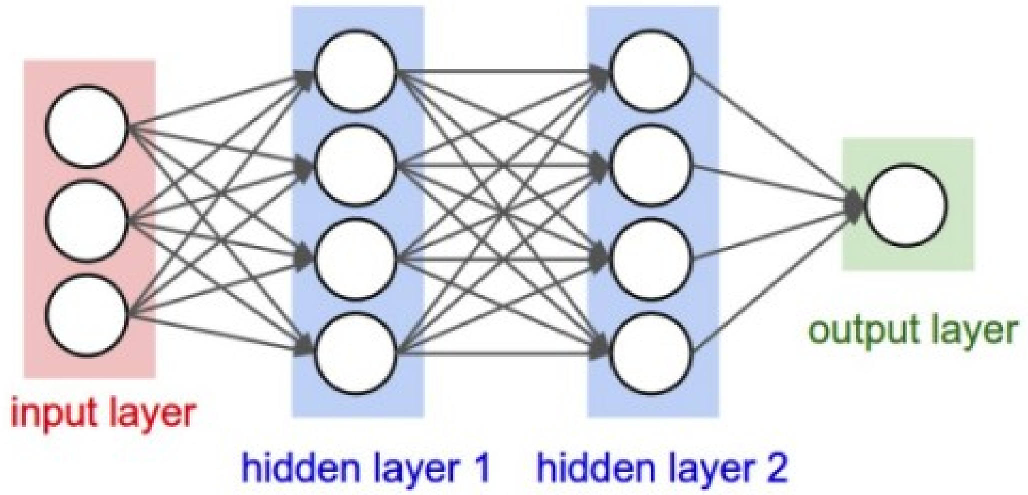
Bias

If a function includes the constant term c

- it can approximate any of the linear functions in the plane

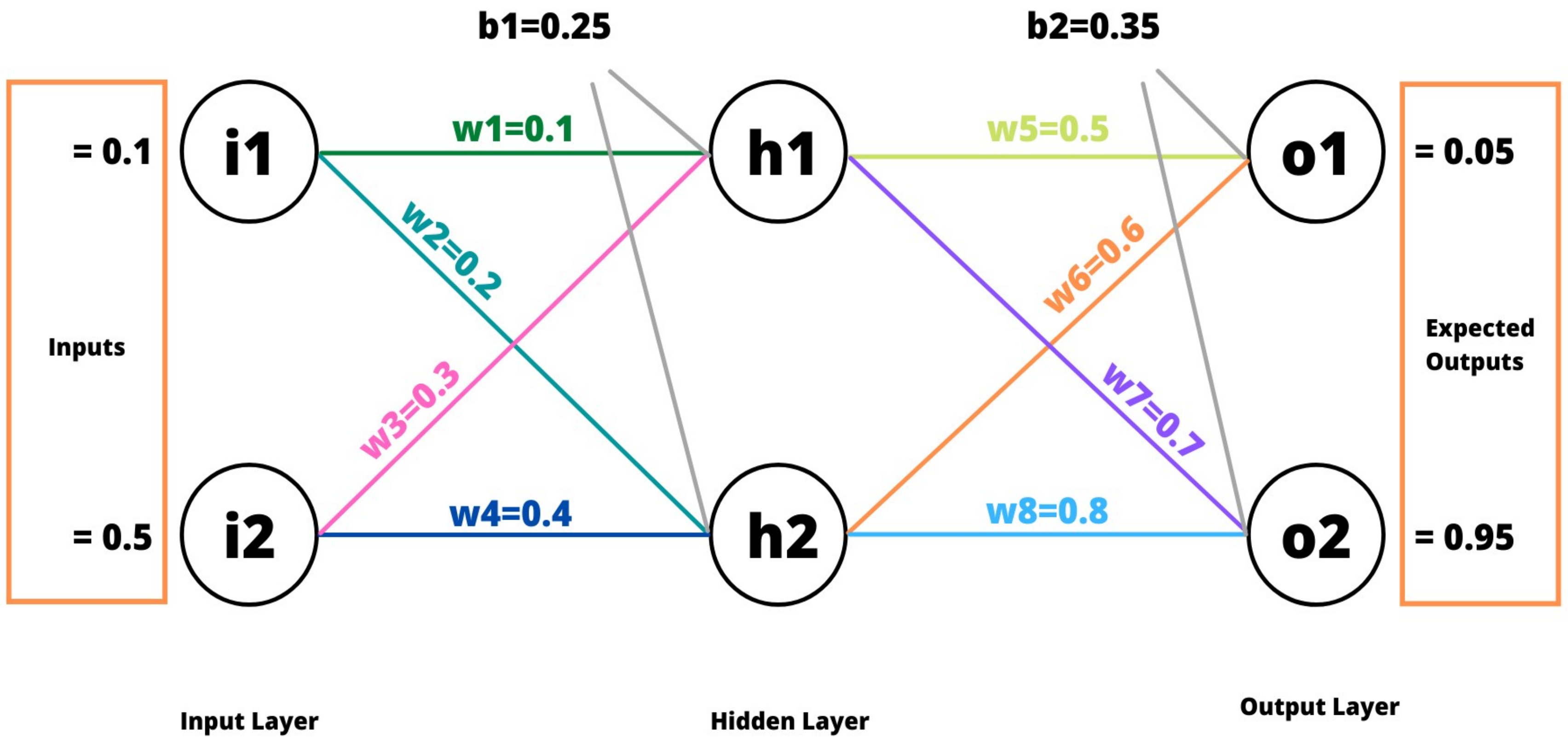


Forward propagation



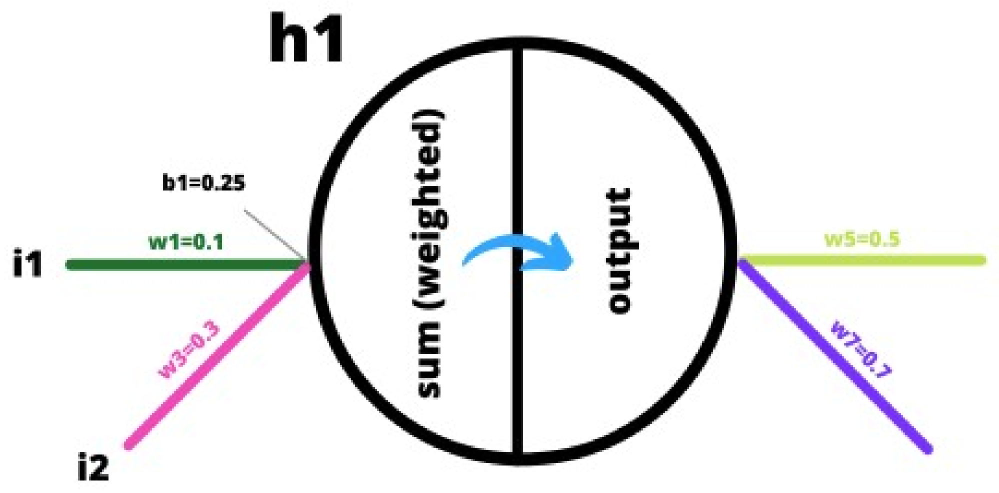
- Stacking a bunch of neurons on top of one another, layer by layer, and connecting these layers via weight vectors
- The outputs of layer 1 get fed as inputs to layer 2, the outputs of layer 2 get fed into layer 3

Example



- The Input nodes provide information from the outside world to the network
- Each data point is fed to each neuron of the input layer
  - each pixel has a gray scale value ranging from 0 for back to 1 for white
- They go through the channels
  - Each channel assign a weight (input*weight) and their some is sent to the next layer
- Then in the hidden layer 1, a bias value is added

Example



$$sum_{h1} = i_1 * w_1 + i_2 * w_3 + b_1$$

$$sum_{h1} = 0.1 * 0.1 + 0.5 * 0.3 + 0.25 = 0.41$$

- Then the value goes through the activation function which determines whether the neuron gets activated or not
- The result from the activation function becomes an input to the next layer (until the next layer is an Output Layer)

Example

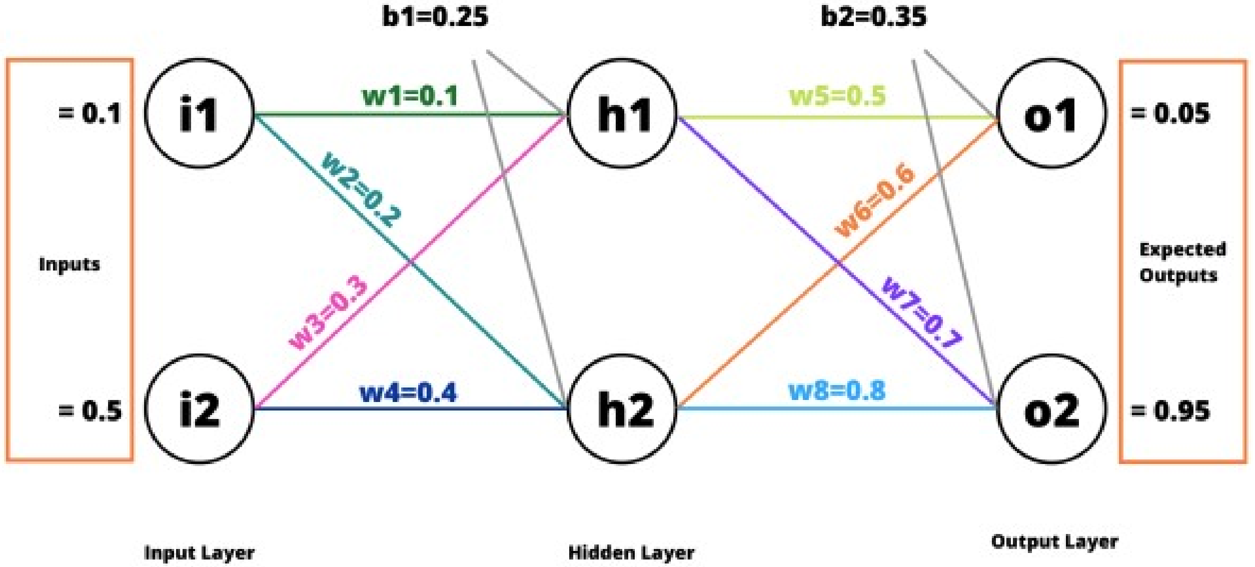
$$output_{h1} = \frac{1}{1 + e^{-sum_{h1}}}$$

$$output_{h1} = \frac{1}{1 + e^{-0.41}} = 0.60108$$

The same process for H2:

$$sum_{h2} = i_1 * w_2 + i_2 * w_4 + b_1 = 0.47$$

$$output_{h2} = \frac{1}{1 + e^{-sum_{h2}}} = 0.61538$$



Network Error

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

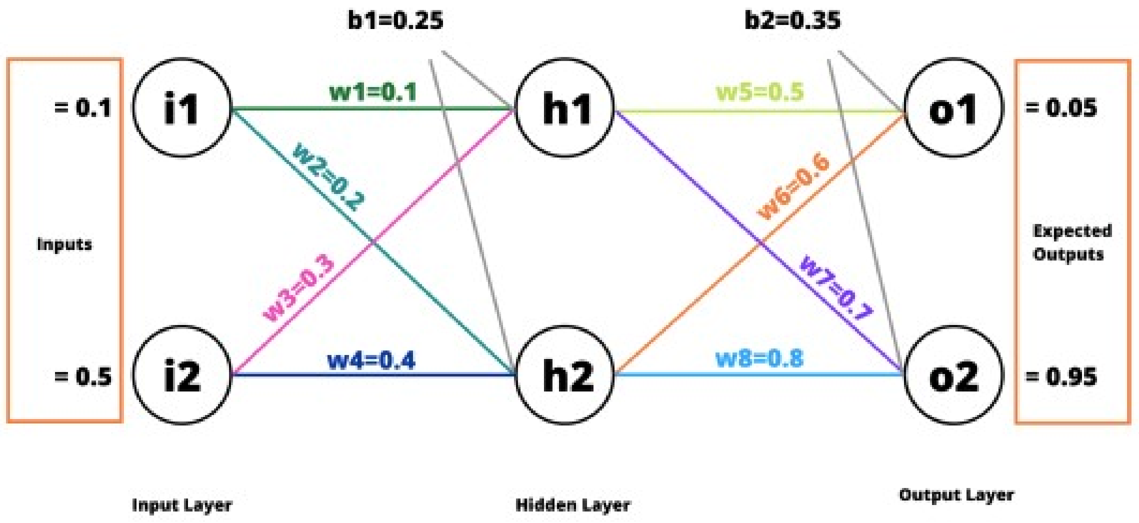
$$E_1 = \frac{1}{2}(target_1 - output_{o1})^2$$

$$E_1 = \frac{1}{2}(0.05 - 0.73492)^2 = 0.23456$$

$$E_2 = \frac{1}{2}(target_2 - output_{o2})^2$$

$$E_2 = \frac{1}{2}(0.95 - 0.77955)^2 = 0.01452$$

$$E_{total} = E_1 + E_2 = 0.24908$$



Backpropagation

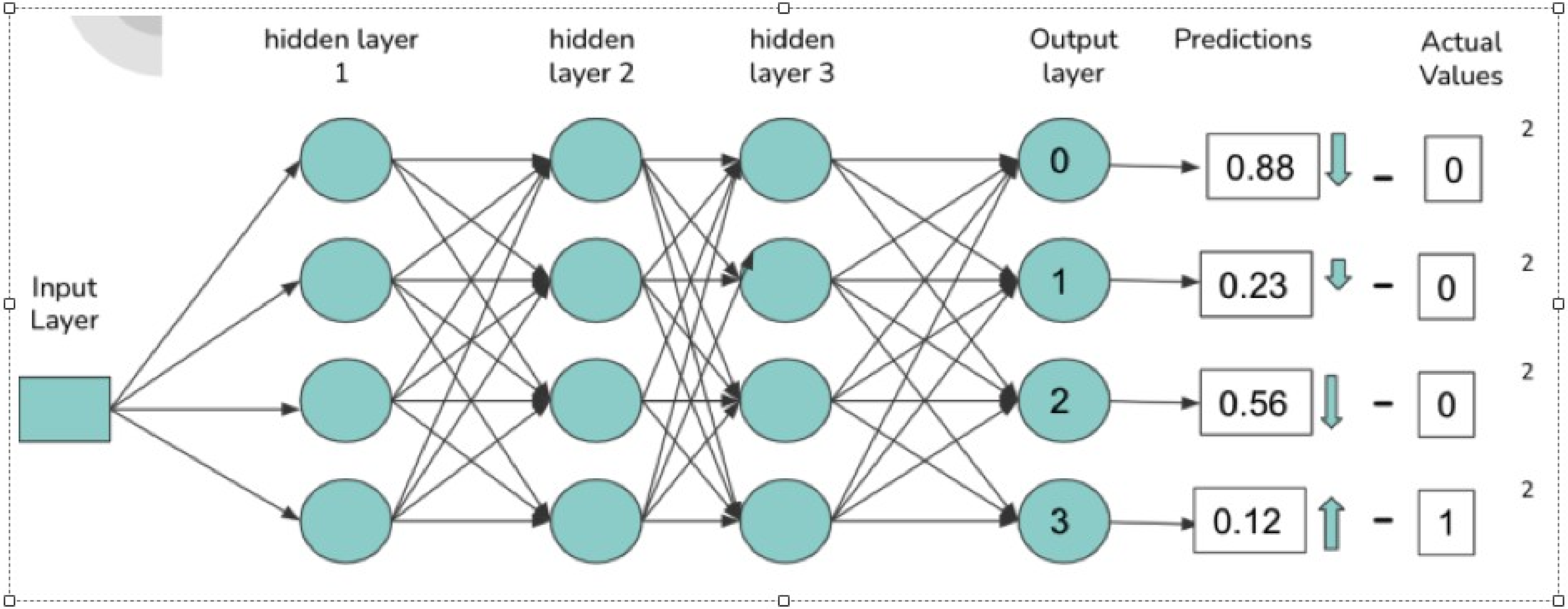
backward pass / Backprop

The goal of backpropagation is to adjust the weights and biases throughout the neural network based on the calculated cost

1. First: **forward pass**
   - The algorithm goes through the graph in the forward direction
     - from the inputs to the output
     - to compute the value of each node
   - The same as when making predictions

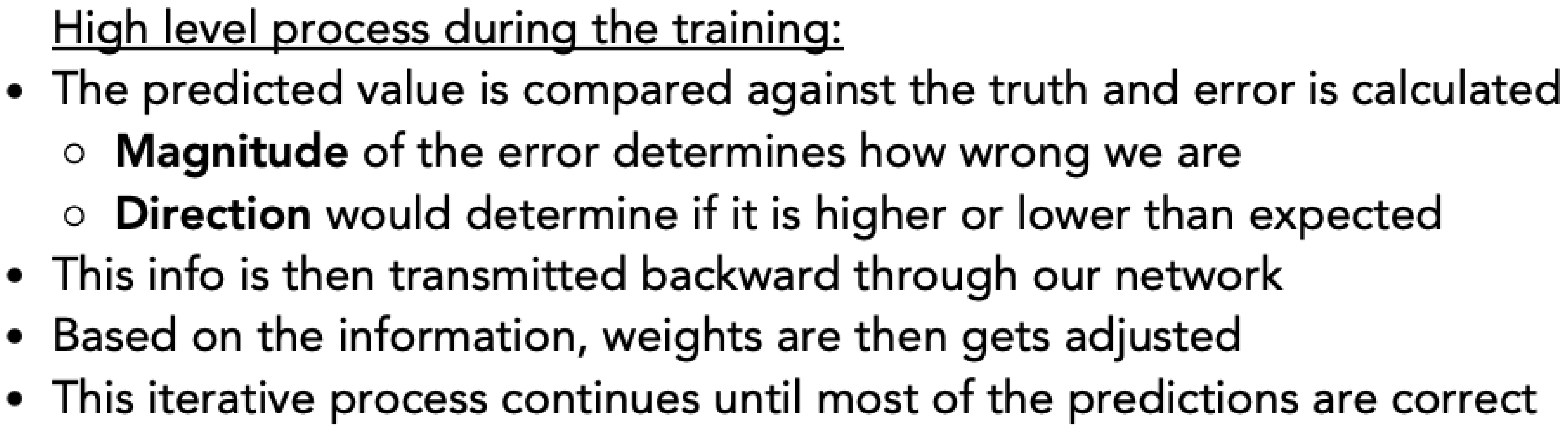
2. Move in the **opposite** direction <u>**to train the model**</u>

2. Move in the **opposite** direction **to train the model**



- Backpropagation is a method **to update the weights** in the neural network
  - by taking into account: the actual output vs. the desired output
  - To minimize the cost function (loss)

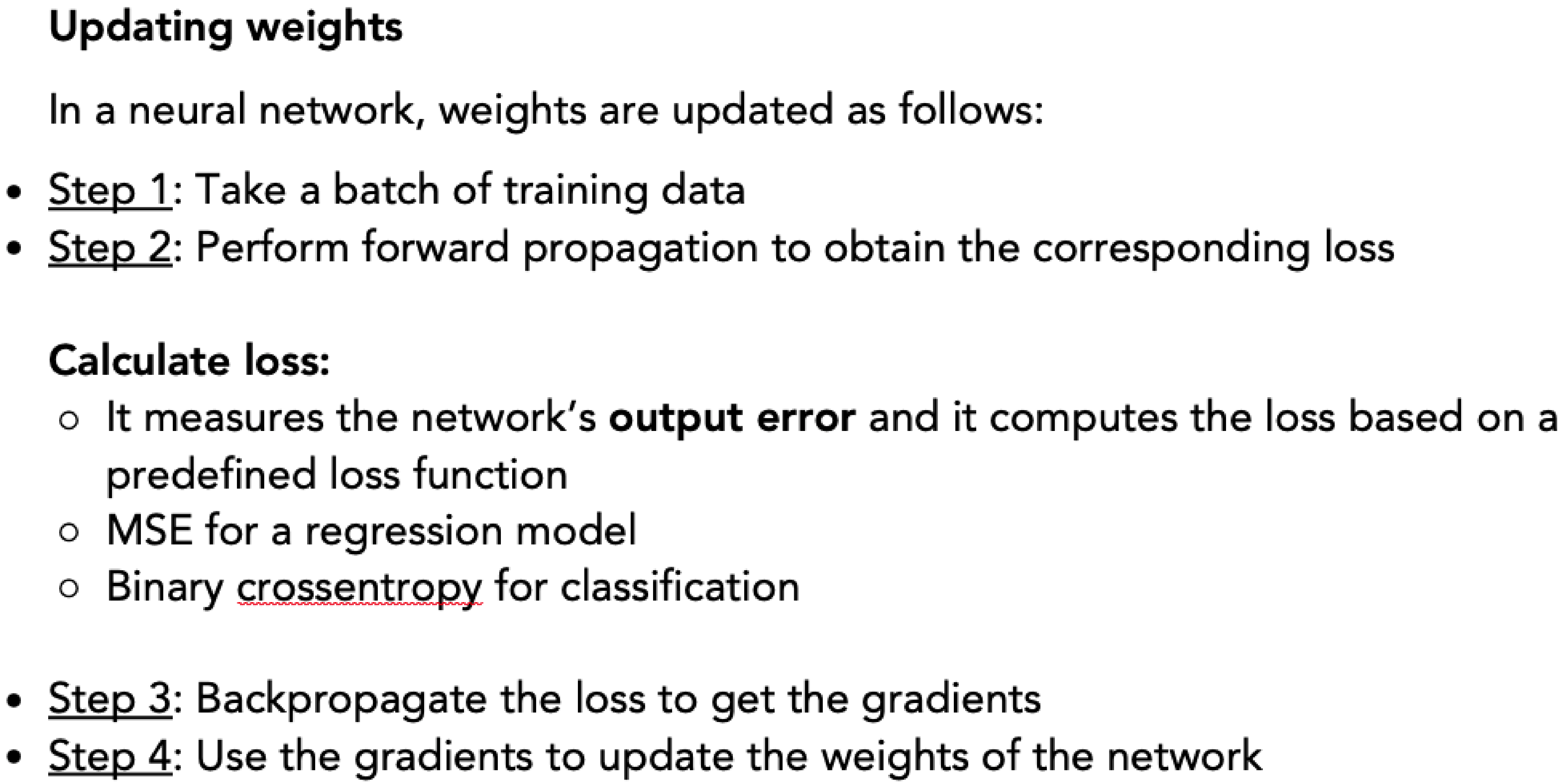
Backpropagation



<u>High level process during the training:</u>

- The predicted value is compared against the truth and error is calculated
  - **Magnitude** of the error determines how wrong we are
  - **Direction** would determine if it is higher or lower than expected
- This info is then transmitted backward through our network
- Based on the information, weights are then gets adjusted
- This iterative process continues until most of the predictions are correct

Backpropagation



Updating weights

In a neural network, weights are updated as follows:

- Step 1: Take a batch of training data
- Step 2: Perform forward propagation to obtain the corresponding loss

Calculate loss:

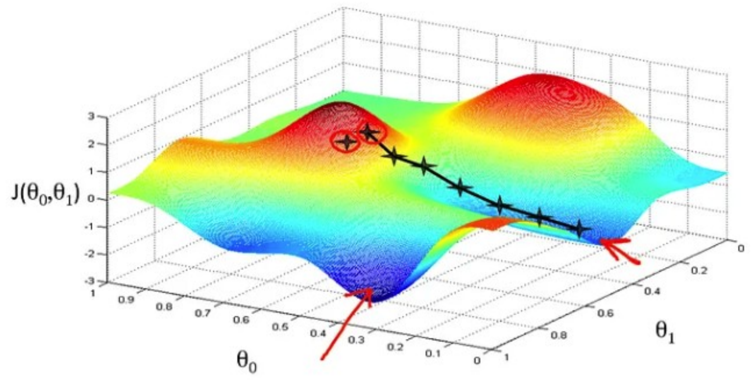
- It measures the network's **output error** and it computes the loss based on a predefined loss function
- MSE for a regression model
- Binary crossentropy for classification

- Step 3: Backpropagate the loss to get the gradients
- Step 4: Use the gradients to update the weights of the network

Backpropagation

Backpropagation is an algorithm that works by calculating the gradient of the loss function:

- points us in the direction of the value that minimizes the loss function
- It relies on the chain rule of calculus to calculate the gradient backward through the layers of a neural network
- Using gradient descent, we can iteratively move closer to the minimum value for cost by taking small steps in the direction given by the gradient

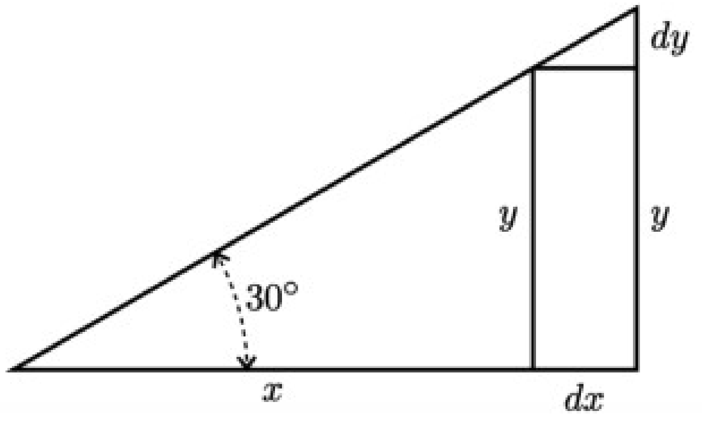


The adjustment works by finding the gradient of the cost function through the chain rule of calculus

derivative of y with respect to x
Objective: how a change in a variable x by the fraction dx affects a related variable y

Use calculus to do that

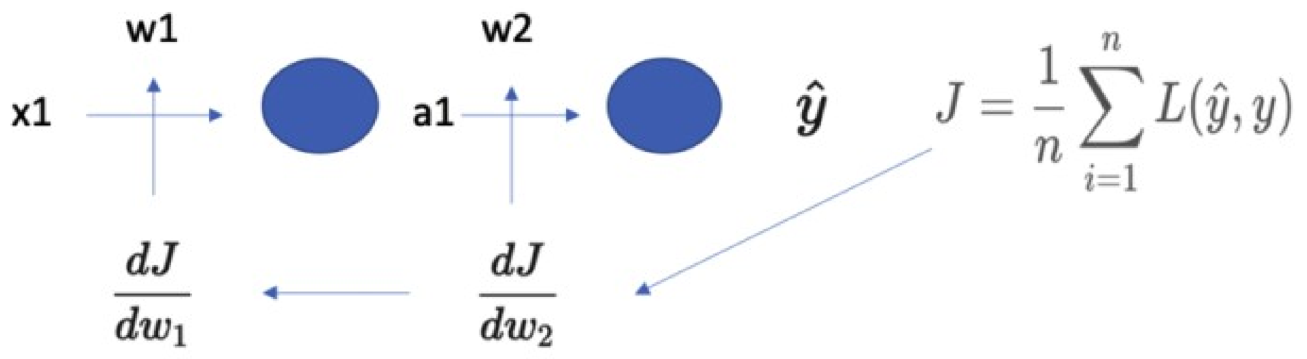
$$\frac{dy}{dx}$$



Backpropagation

- Interested in: how the change in weights affects the error expressed by the cost
- Ultimate goal is to find the set of weights that minimizes the cost and thus the error
- Intermediate goal is to find the negative gradient of the cost function with respect to the weights, which points us in the direction of the desired minimum

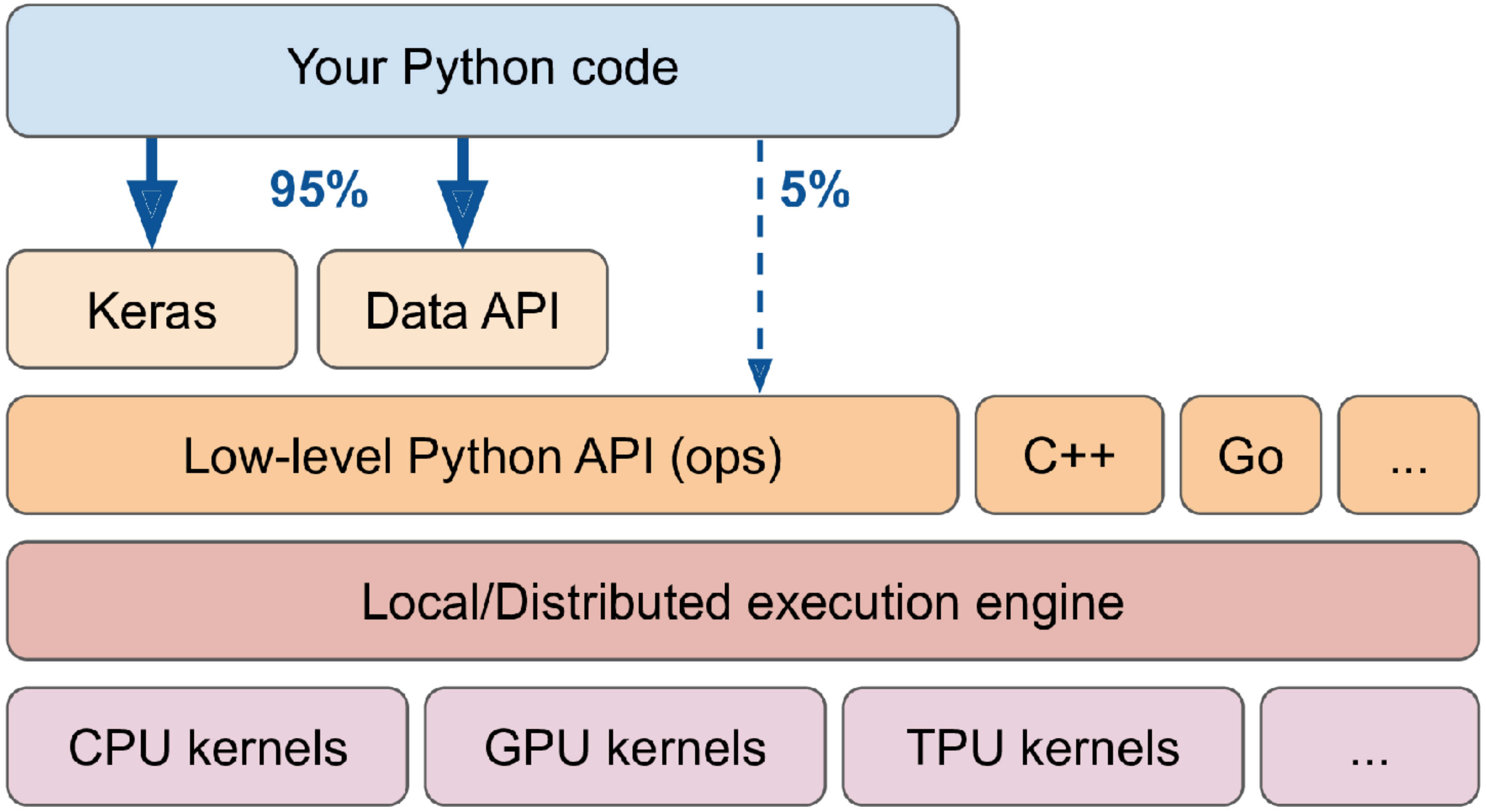
$$z_1 = w_1 x_1 \;\rightarrow\; a_1 = \sigma(z_1) \;\rightarrow\; z_2 = w_2 a_1 \;\rightarrow\; \hat{y} = \sigma(z_2) \;\rightarrow\; J = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y)$$



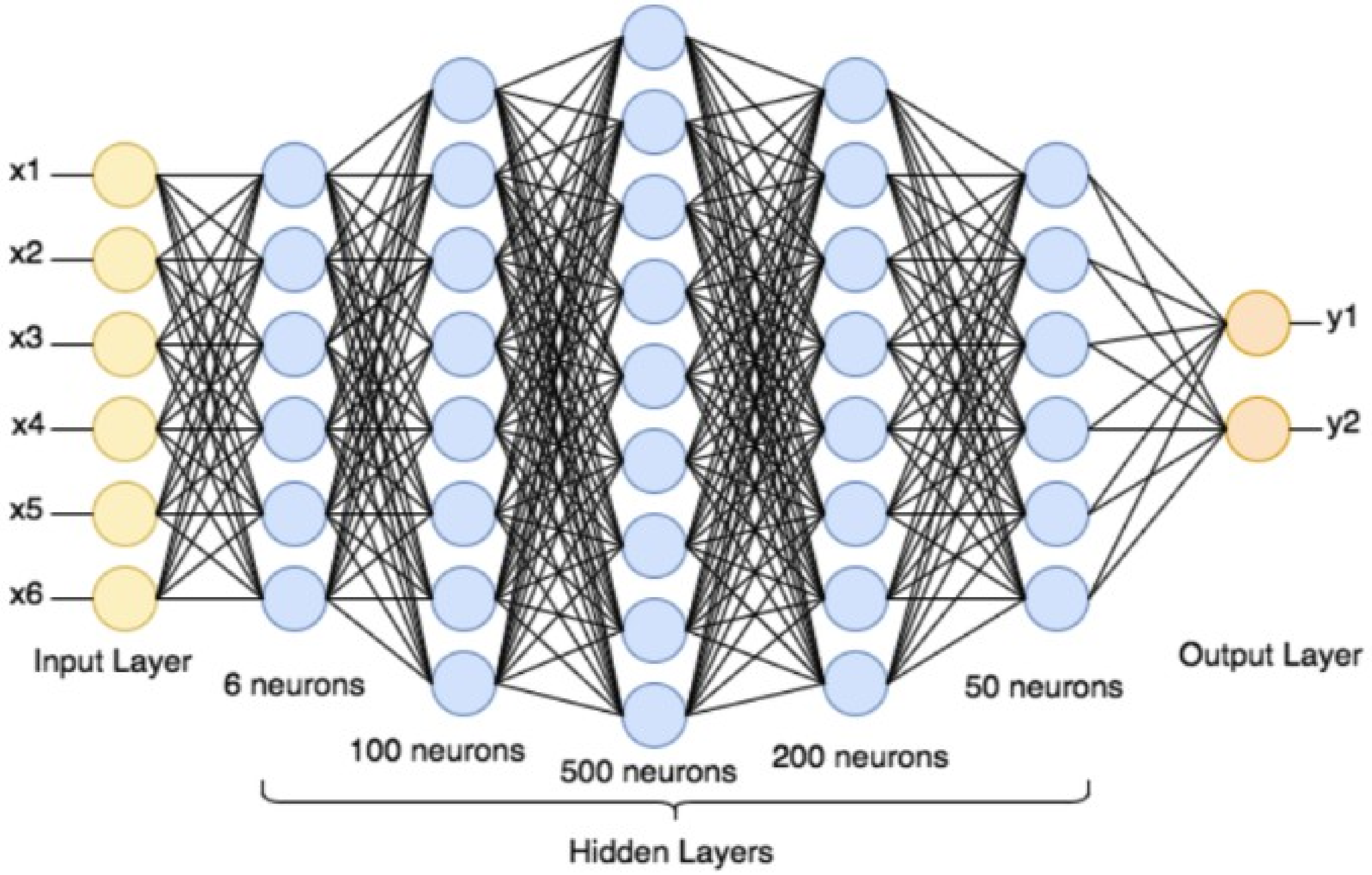
$$z_2 = w_2 a_1 \;\rightarrow\; \hat{y} = \sigma(z_2) \;\rightarrow\; J = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y)$$

backward differentiation through several intermediate functions

Neural Nets with Tensorflow and Keras



Fine-Tuning of Neural Networks



Fine-Tuning of Neural Networks

Number of Hidden Layers:

- Real-world data is often structured in a hierarchical fashion, and deep neural networks can take advantage of this by modeling increasing levels of granularity at each layer

- Example of image classification:
    - lower hidden layers are able to model low-level structures (e.g. line segments of various shapes and orientations),
    - intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g. squares, circles),
    - and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g. faces).

Fine-Tuning of Neural Networks

Choosing the Number of Neurons:

- The number of neurons in the input and output layers is determined by the type of input and output your task requires.
- For example, the MNIST task, for which your model tries to classify images of size 28x28 into 10 classes, requires:
  - 28*28=784 input neurons
  - and 10 output neurons

Fine-Tuning of Neural Networks

Best practices

- need to experiment to find the best approach
- A good place to start:
  - somewhere between 32 and 128 neurons per layer
  - outside that range there is risk of underfitting or overfitting the training set
- One strategy for choosing the number of neurons per hidden layer is to make all layers have same number of neurons, and then gradually increase them until the model starts overfitting the training data

Hyperparameters

Variables which determines:

- The network structure (e.g: Number of Hidden Units)
- How the network is trained (e.g: Learning Rate)

Hyperparameters are set before training (before optimizing the weights and bias)

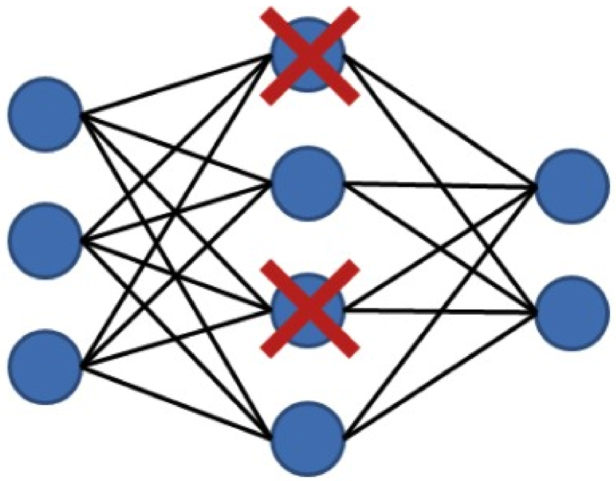
Hyperparameters

Number of Hidden Layers and units

- How to choose a number?
  - Generally, keep adding layers until the test error does not improve anymore
- Many hidden units within a layer with regularization techniques can increase accuracy
- Smaller number of units may cause **underfitting**

Hyperparameters

Dropout



- A regularization technique to avoid overfitting
  - To **increase the validation accuracy** -> **increasing the generalizing power**

General practices:

- use a small dropout value of 20%-50% of neurons with 20% providing a good starting point
- A probability too low has minimal effect
- A probability too high results in under-learning
- When using a larger network:
  - likely to get better performance when dropout is used on a larger network
  - giving the model more of an opportunity to learn independent representations

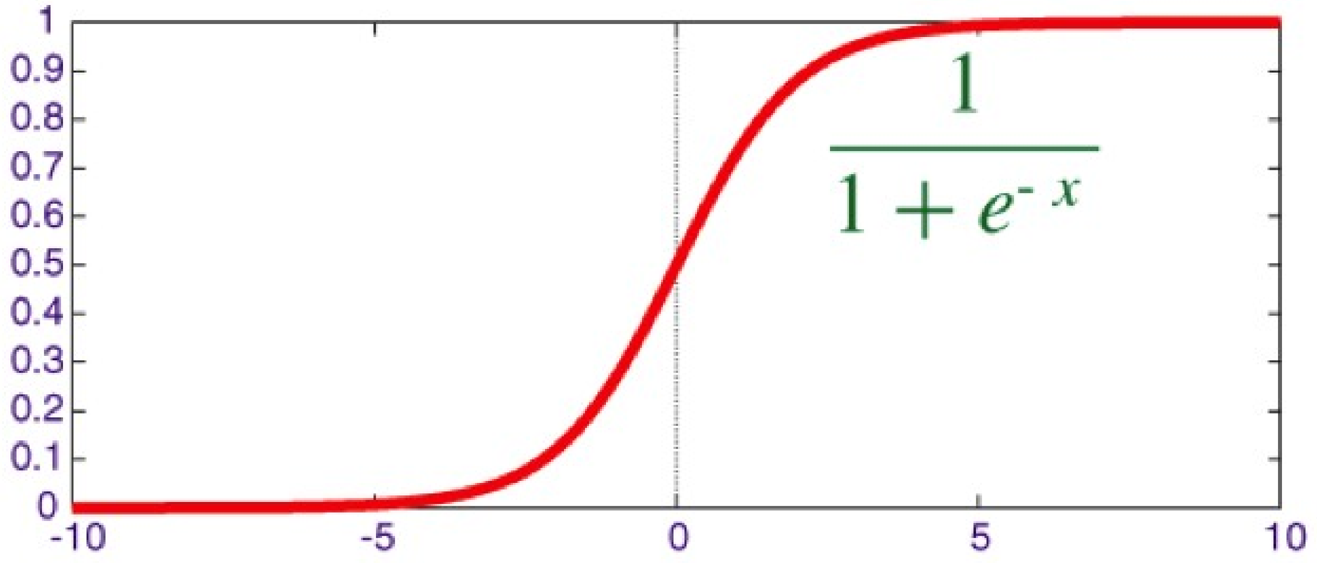
Hyperparameters

Network Weight Initialization

- Ideally, it may be better to use different weight initialization schemes according to the activation function used on each layer
- Mostly **uniform distribution** is used

Hyperparameters

Activation function



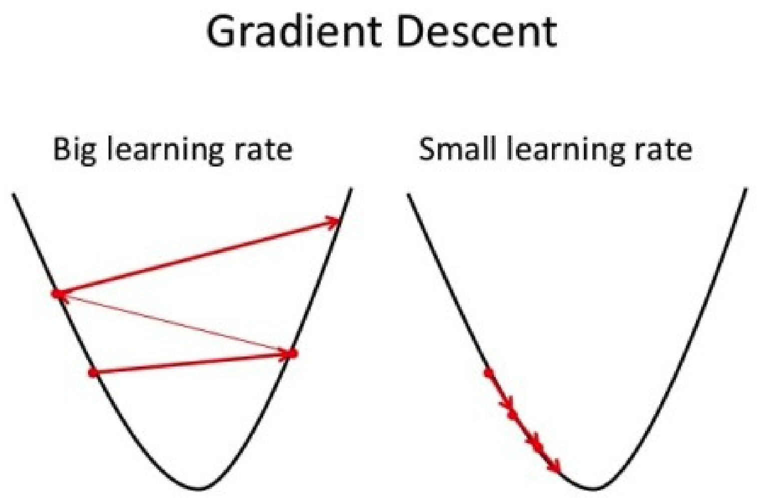
- Used to **introduce nonlinearity** to models, which allows deep learning models to learn nonlinear prediction boundaries

<u>General practices:</u>

- the **rectifier activation function** is the most popular
- **Sigmoid** is used in the output layer while making **binary predictions**
- **Softmax** is used in the output layer while making **multi-class predictions**

Hyperparameters

Learning Rate



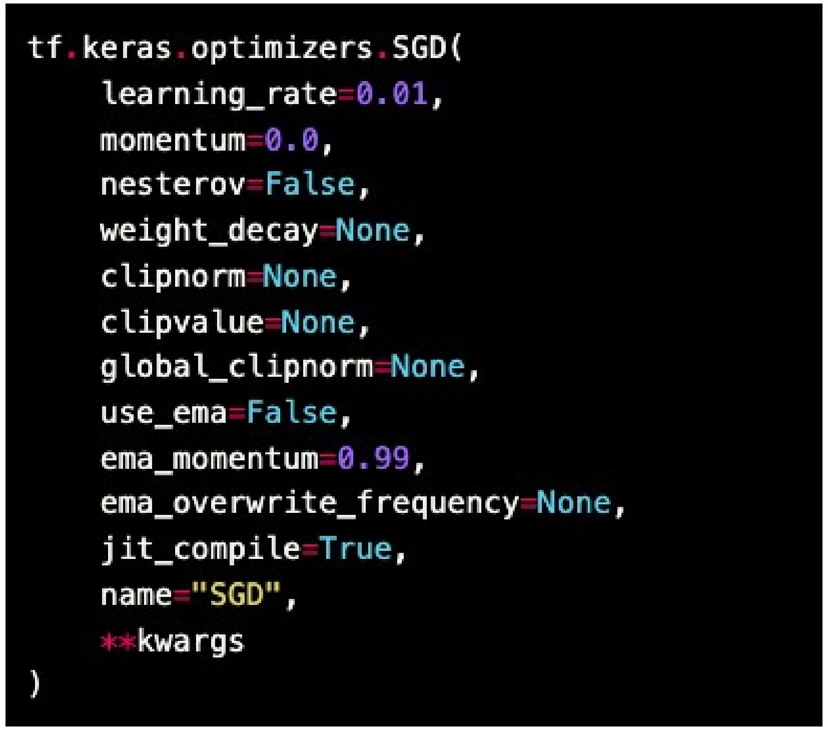
- The learning rate defines how quickly a network updates its parameters
  - **Low learning rate** slows down the learning process but converges smoothly
  - **Larger learning rate** speeds up the learning but may not converge

- Usually a **decaying Learning rate** is preferred
  - It starts training the network with a large learning rate and then slowly reducing/decaying it until local minima is obtained
  - It is empirically observed to help both optimization and generalization

Hyperparameters

Momentum

- Momentum helps to know the direction of the next step with the knowledge of the previous steps
- It helps to prevent oscillations
- A typical choice of momentum is between 0.5 to 0.9

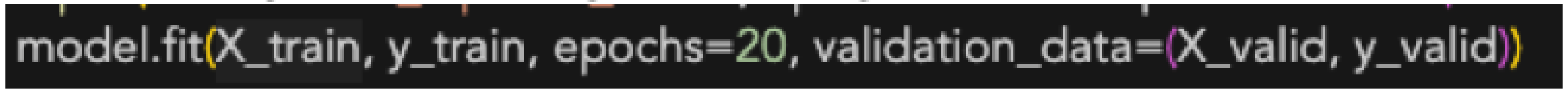


```
tf.keras.optimizers.SGD(
    learning_rate=0.01,
    momentum=0.0,
    nesterov=False,
    weight_decay=None,
    clipnorm=None,
    clipvalue=None,
    global_clipnorm=None,
    use_ema=False,
    ema_momentum=0.99,
    ema_overwrite_frequency=None,
    jit_compile=True,
    name="SGD",
    **kwargs
)
```

Hyperparameters

Number of epochs

- Number of epochs is the number of times the whole training data is shown to the network while training
- Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing(overfitting)



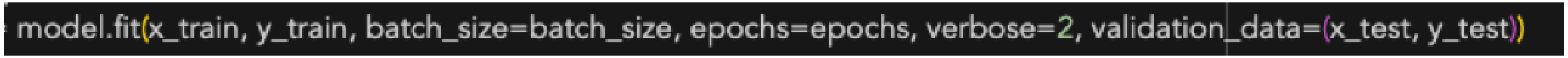
```
model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
```

one **epoch** = one forward pass and one backward pass of *all* the training examples

Hyperparameters

Batch size

the number of samples that will be propagated through the network



```
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=2, validation_data=(x_test, y_test))
```

batch size = the number of training examples in one forward/backward pass.
The higher the batch size, the more memory space you'll need

Example:
you have 1050 training samples and you want to set up a batch_size equal to 100.

- The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network
- Next, it takes the second 100 samples (from 101st to 200th) and trains the network again