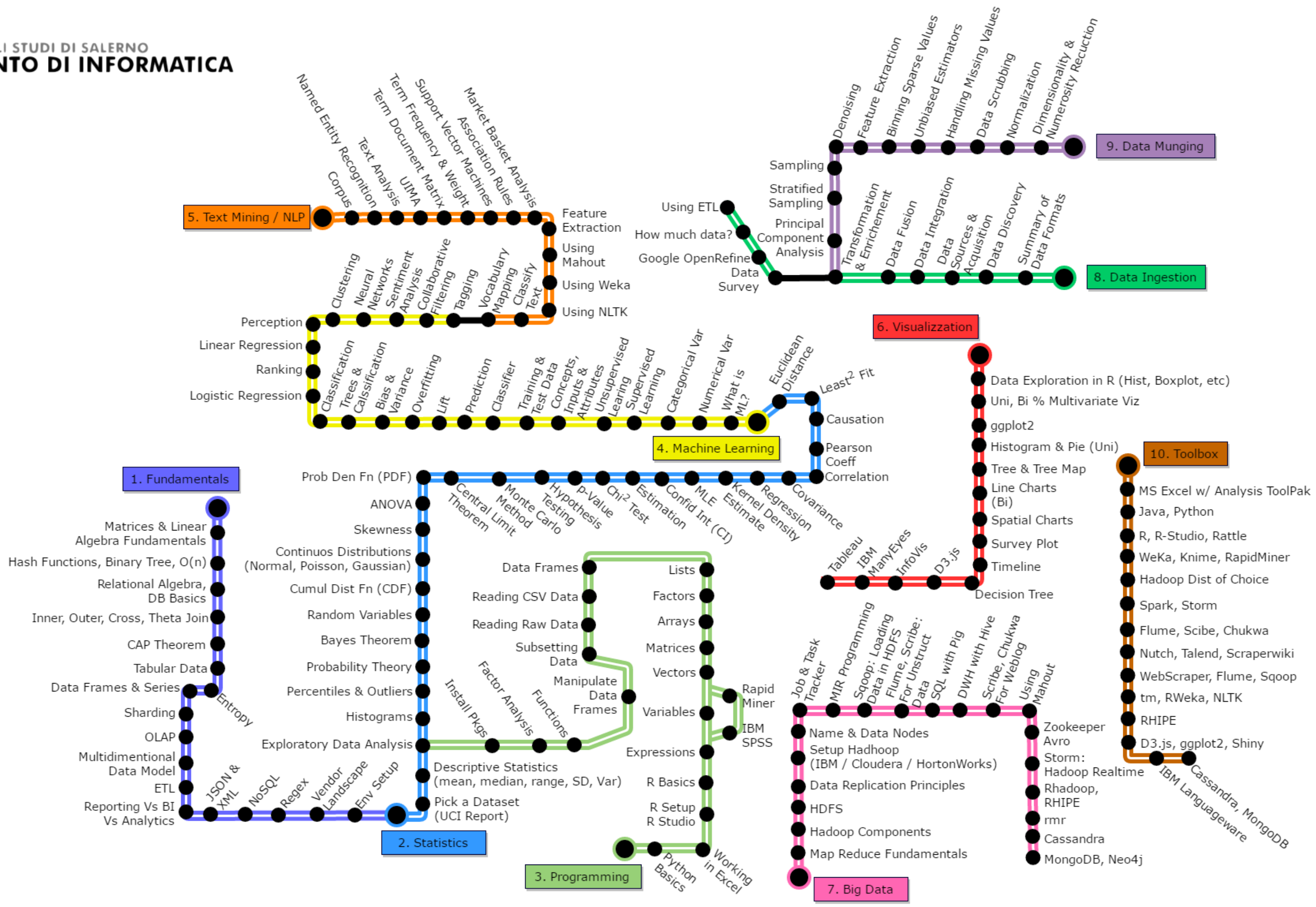
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

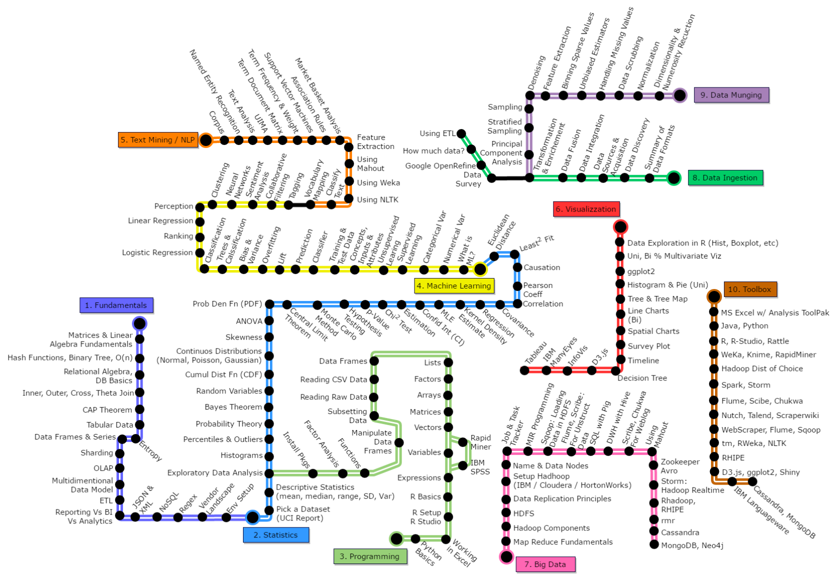
Introduction to Artificial Neural Networks

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

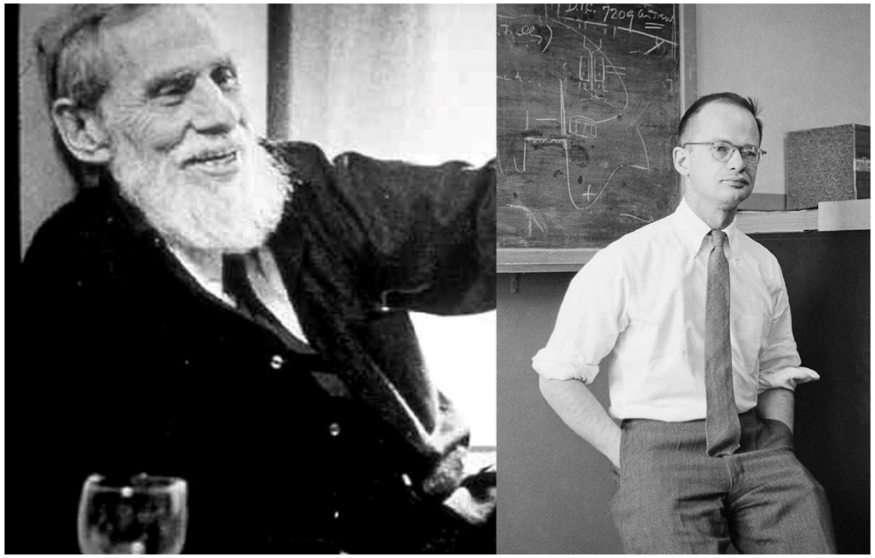
Outline

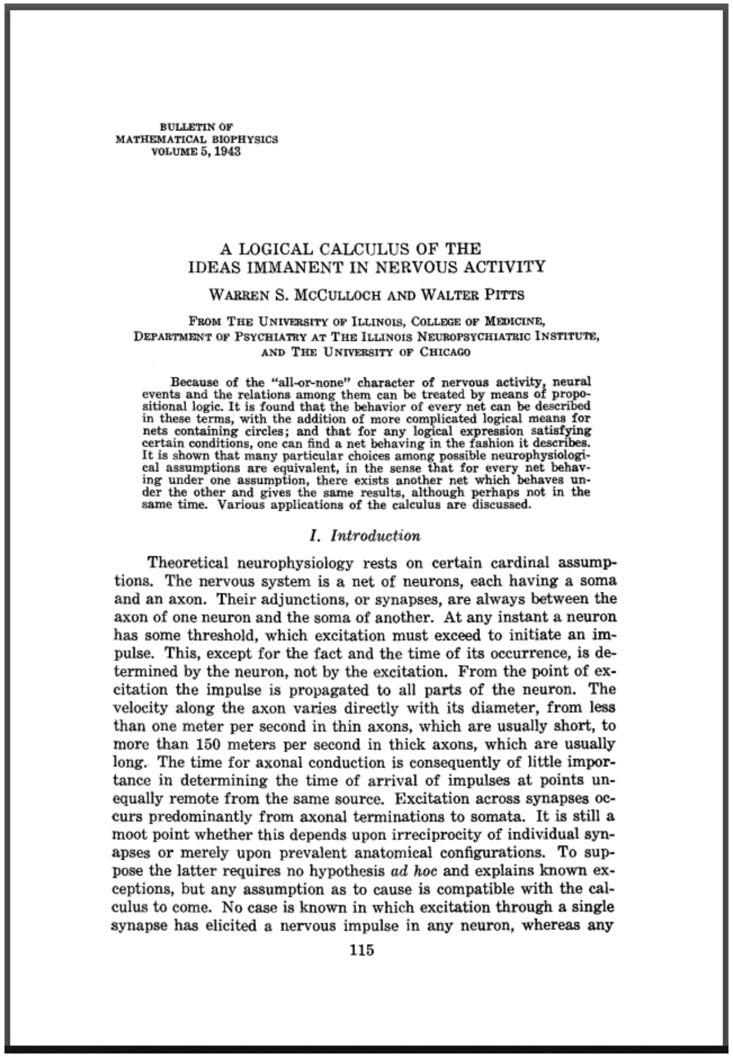
- From Biological to Artificial Neurons
- Artificial Neural Network
  - Artificial Neuron
  - Perceptron
  - Model
  - Step Function
- Training perceptron
- Multi-Layer perceptron and backpropagation
- Training a Deep Neural Network
- A complete Example



From Biological to Artificial Neurons (1)

- Artificial Neural Networks (ANN) were first introduced back in **1943** by the neurophysiologist *Warren McCulloch* and the mathematician *Walter Pitts*





BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

I. Introduction

Theoretical neurophysiology rests on certain cardinal assumptions. The nervous system is a net of neurons, each having a soma and an axon. Their adjunctions, or synapses, are always between the axon of one neuron and the soma of another. At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse. This, except for the fact and the time of its occurrence, is determined by the neuron, not by the excitation. From the point of excitation the impulse is propagated to all parts of the neuron. The velocity along the axon varies directly with its diameter, from less than one meter per second in thin axons, which are usually short, to more than 150 meters per second in thick axons, which are usually long. The time for axonal conduction is consequently of little importance in determining the time of arrival of impulses at points unequally remote from the same source. Excitation across synapses occurs predominantly from axonal terminations to somata. It is still a moot point whether this depends upon irreciprocity of individual synapses or merely upon prevalent anatomical configurations. To suppose the latter requires no hypothesis *ad hoc* and explains known exceptions, but any assumption as to cause is compatible with the calculus to come. No case is known in which excitation through a single synapse has elicited a nervous impulse in any neuron, whereas any

115

From Biological to Artificial Neurons (2)

- The early successes of **ANNs** until the **1960s** led to the widespread belief that we would soon be conversing with truly intelligent machines
- In the early **1980s** there was a revival of interest in ANNs as new network architectures were invented and better training techniques were developed
- But by the **1990s**, powerful alternative Machine Learning techniques such as Support Vector Machines were favored by most researchers
- In the last few years, however, we are now witnessing yet another wave of interest in **ANNs**

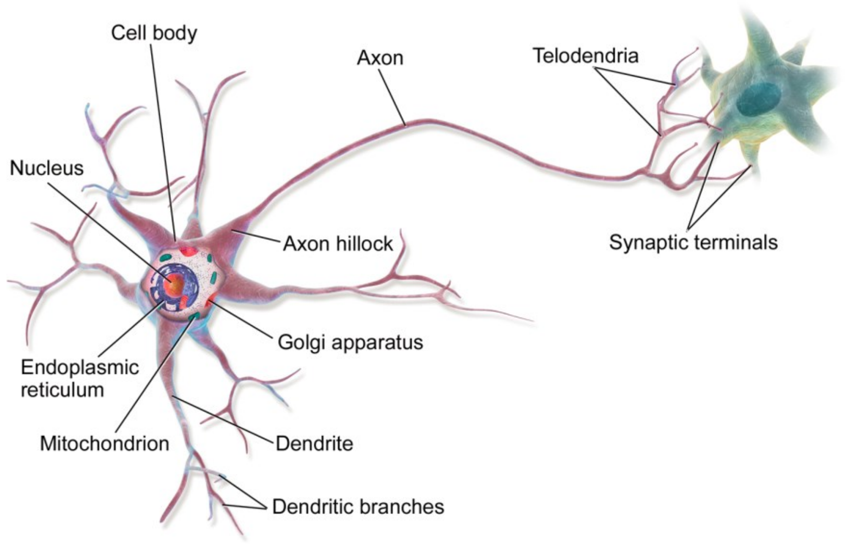
From Biological to Artificial Neurons (3)

- There are a few good reasons to believe that this one is different and will have a much more profound impact on our lives:
  - There is now a huge quantity of data available to train neural networks
  - The **tremendous** increase in computing power since the 1990s
  - The training algorithms have been improved
  - Convergence to local optima is rare in practice
  - When occur, Local optima often are close to optimal solution
  - ANNs have entered a virtuous circle of funding and progress

Biological Neurons

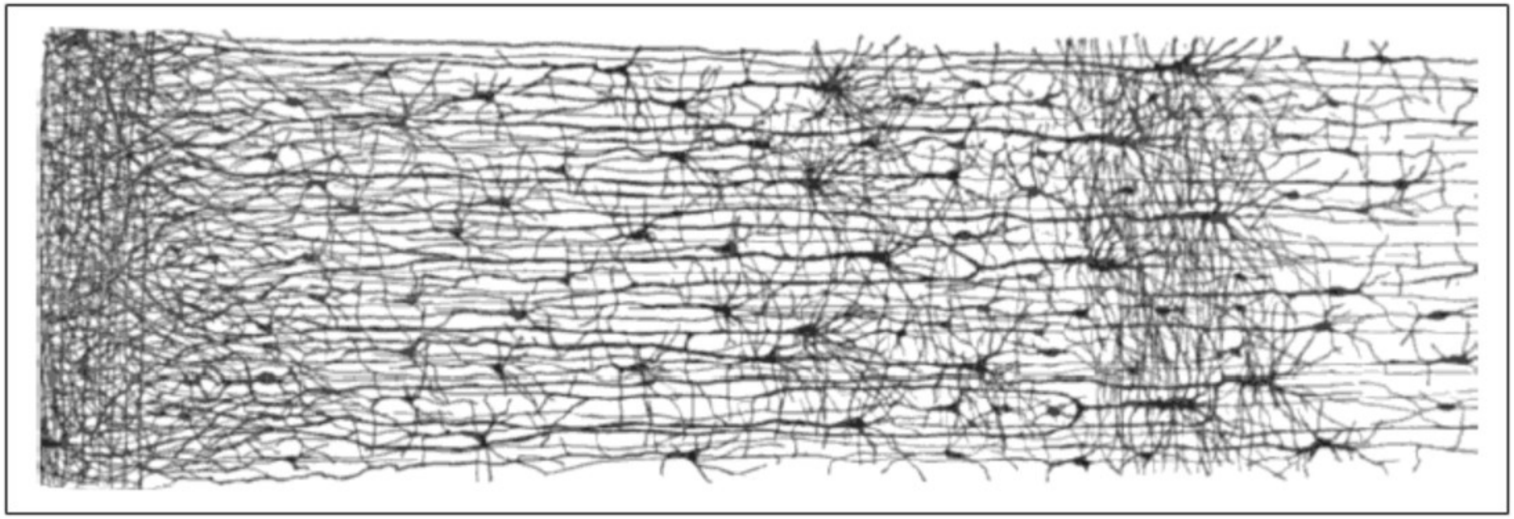
- The cell of animal cerebral cortexes is composed of:
  - A body containing the *nucleus* and most cell's complex components
  - Many branching extensions called *dendrites*
  - A long extension called axon, which splits into many branches called telodendria, at the tip of which there are synaptic terminals, connected to dendrites (or directly to the cell body) of other neurons

- Neurons receive short electrical impulses called signals from other neurons via synapses.
- Upon receiving a sufficient number of signals from other neurons within few milliseconds, a neuron fires its own signals



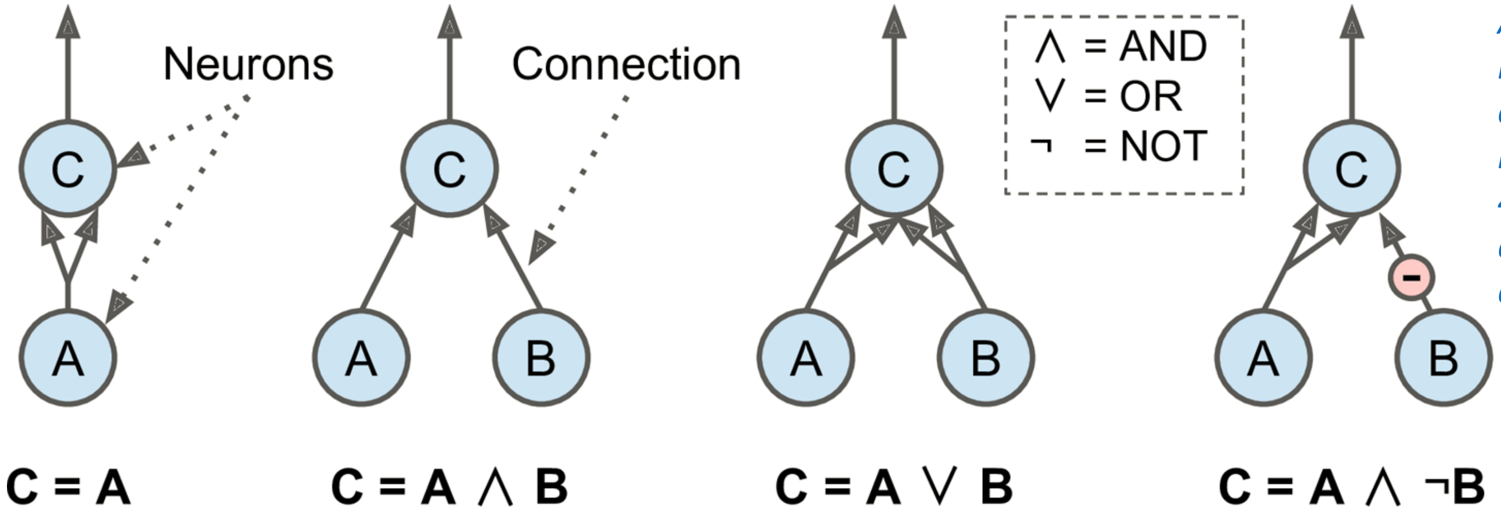
Architecture of Biological Neural Nets

- Biological neurons are organized in networks of billions of neurons, each connected to thousands other neurons
- Such networks enable highly complex computations
- Active research is still focusing on the architecture of Biological NN (BNN), but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers



Artificial Neuron

- McCulloch and Pitts proposed a simple model Artificial Neuron
- It has one or more binary (on/off) inputs and one binary output
- It activates its output when more than a certain number of its inputs are active (2 in the example)

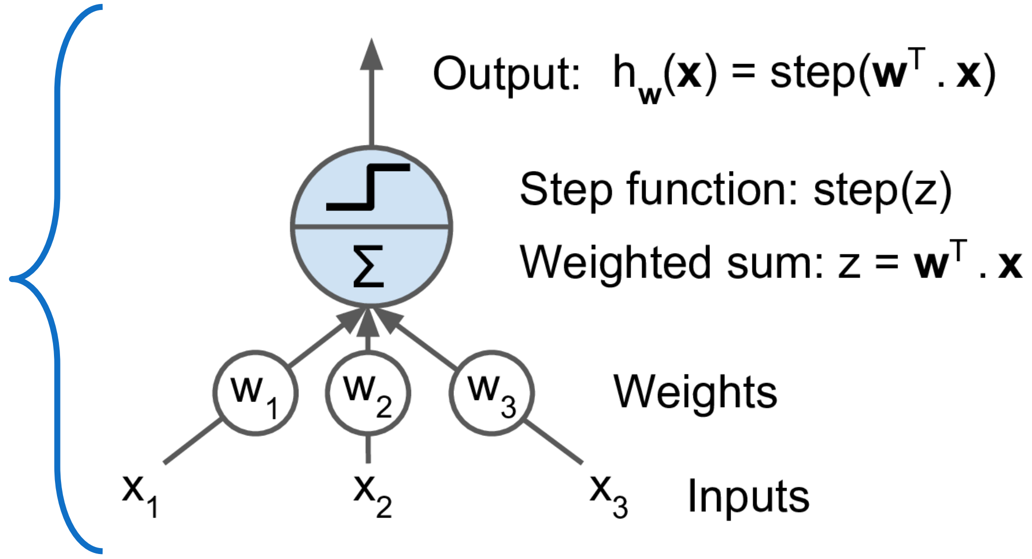


As in biological neurons an input connection can inhibit neuron's activity, in the 4th NN neuron C is activated only if A is active and B is off.

Perceptron (1)

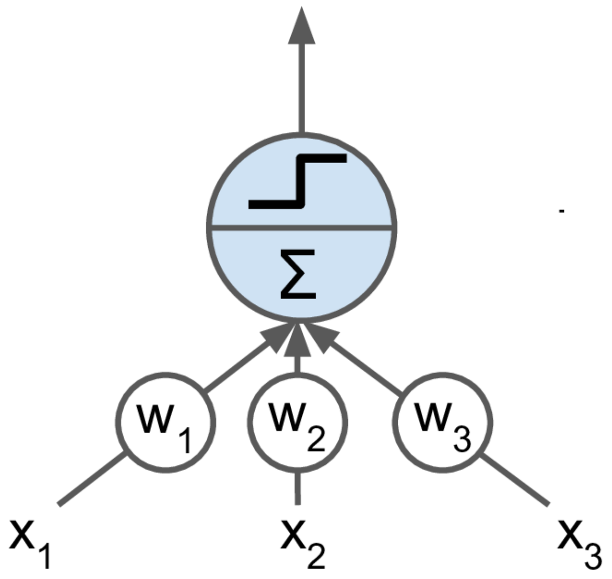
- The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt
- Inputs are numbers instead of binary on/off values

Artificial Neuron is called Linear Threshold Unit (LTU)



Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{w}^T . \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^T . \mathbf{x}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

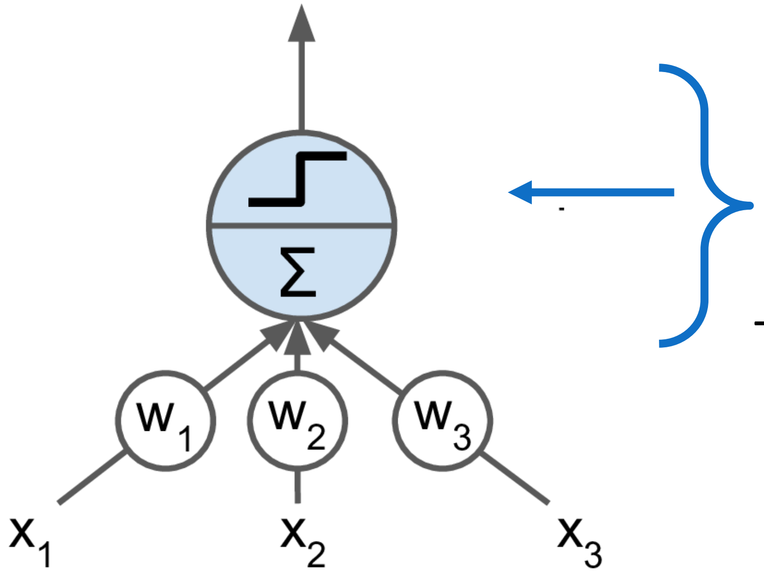
Perceptron (2)



$\left.\right\}$ Weighted sum of its inputs

$(z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x})$

Perceptron (3)



Applies a step function to that sum and outputs the result:

$h_w(x) = \mathbf{step}(z) = \mathbf{step}(w^T \cdot x)$

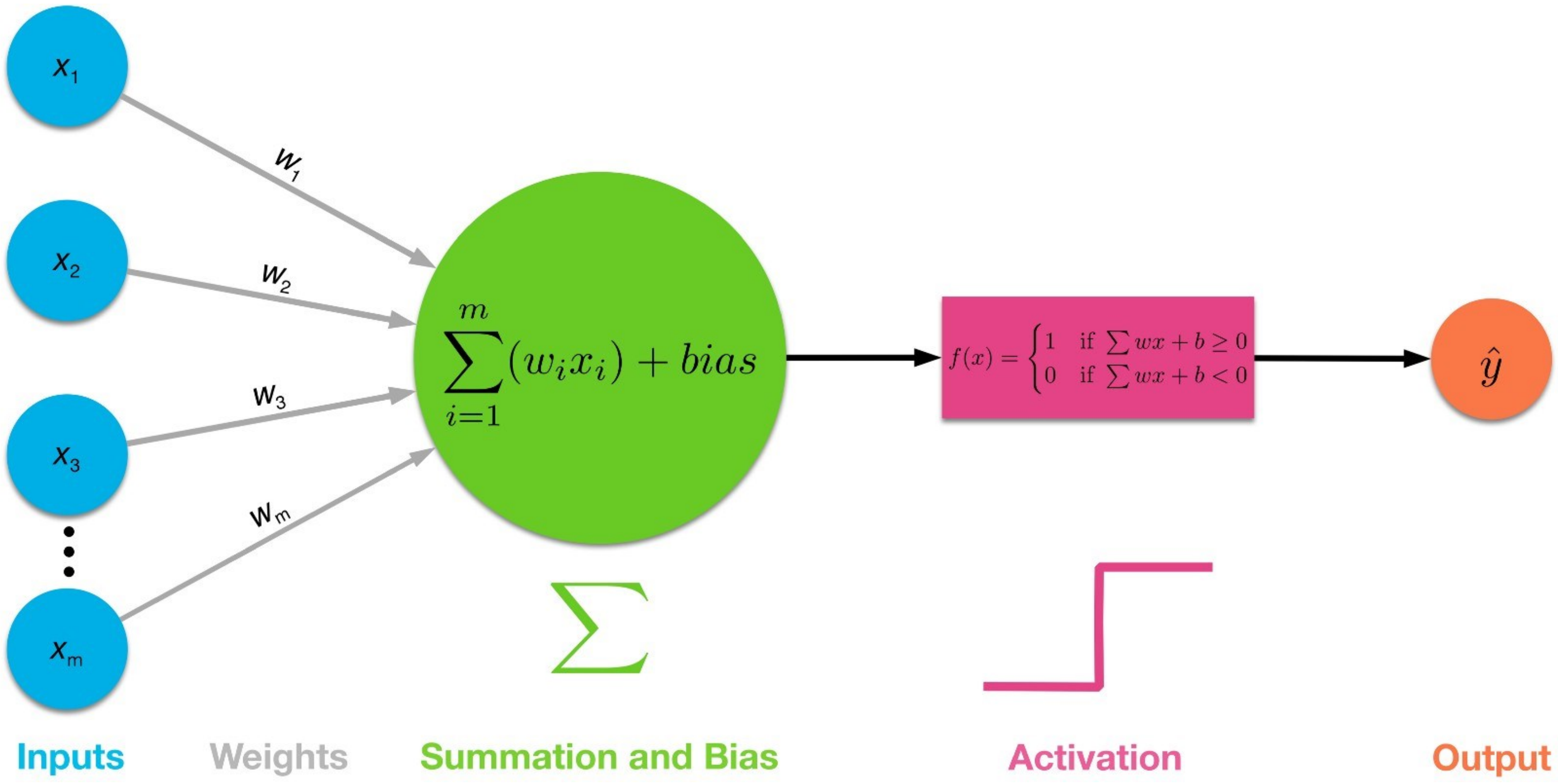
Typical step functions are *Heavyside* or *Sign*

Weighted sum of its inputs

$(z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = w_T \cdot x)$

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

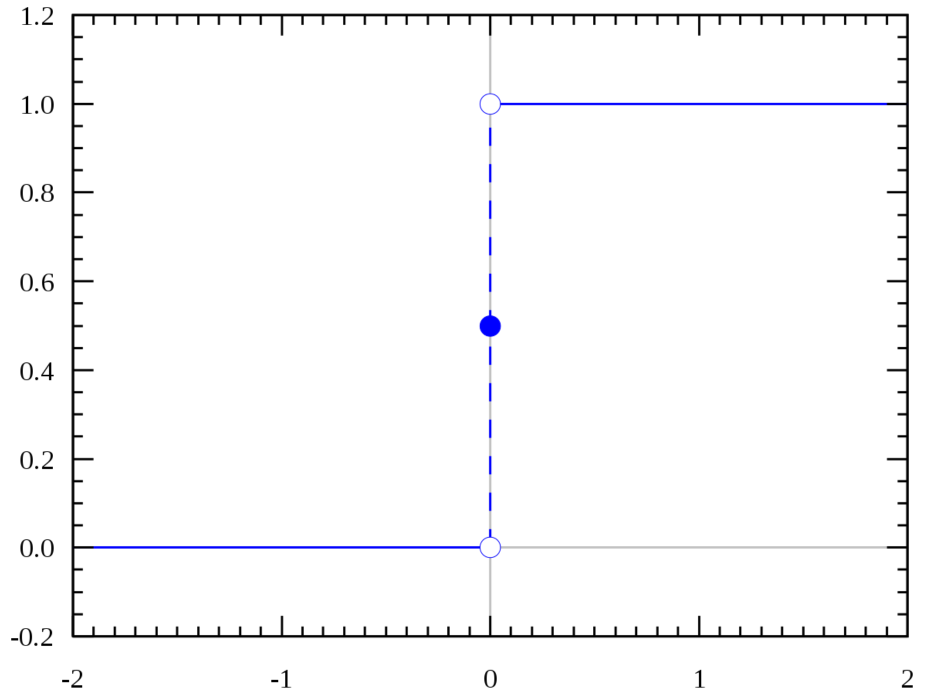
Model



x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

Heaviside Step Function

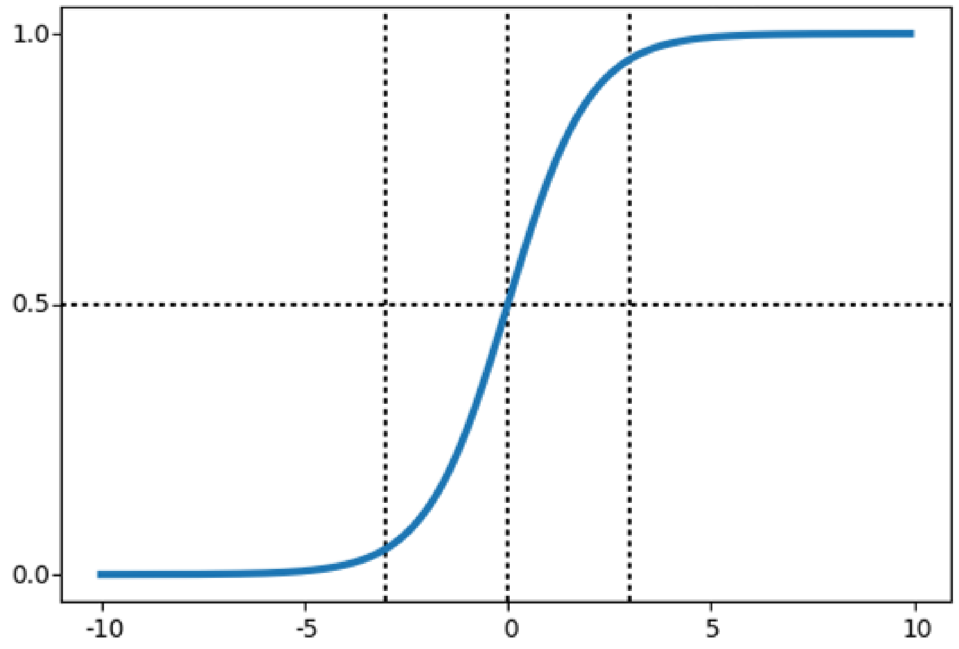
- A step function will output 1 if the input is higher than a certain threshold, 0 otherwise



$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Sigmoid Function (1)

- The main reason why we use sigmoid function is because it exists between (0 to 1)

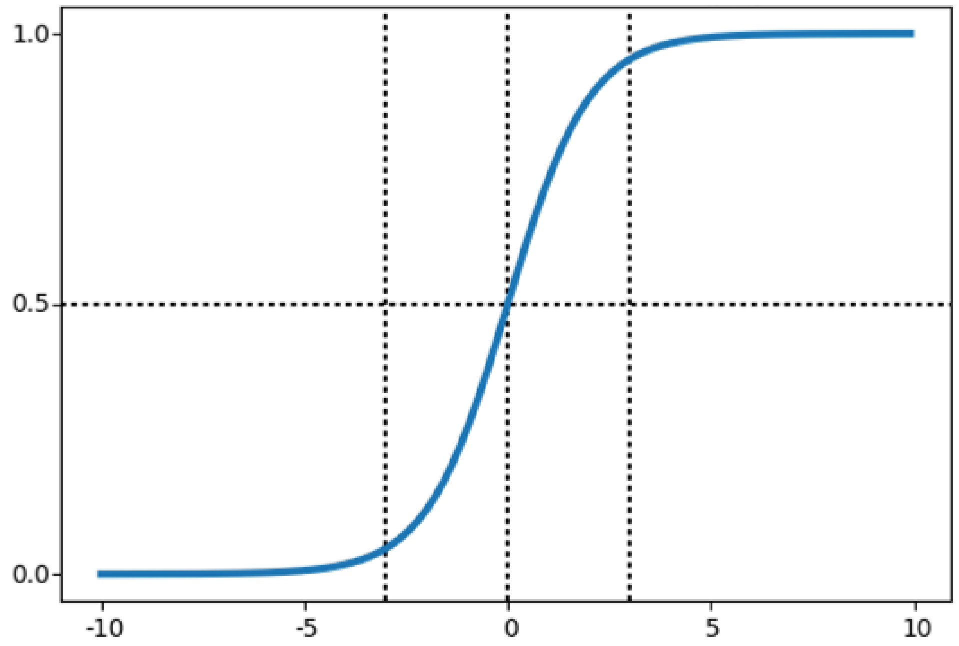


$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$z = \sum_{i=1}^{m} w_i x_i + bias$$

Sigmoid Function (2)

- The main reason why we use sigmoid function is because it exists between (0 to 1)



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

$$z = \sum_{i=1}^{m} w_i x_i + bias$$

The problem is that it does not have any negative value

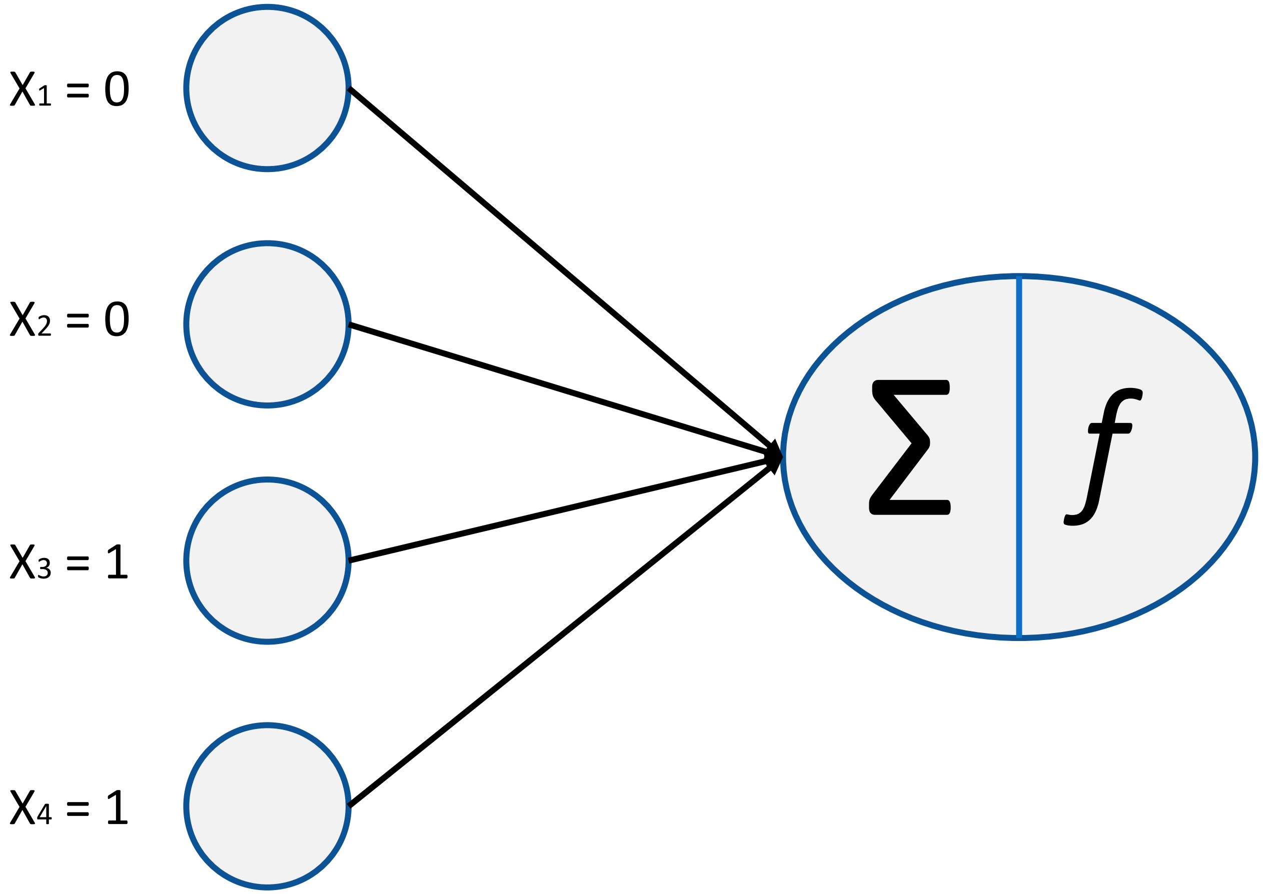
Sigmoid Function (3)

- Sigmoid function is used for models where we have to predict the probability as an output

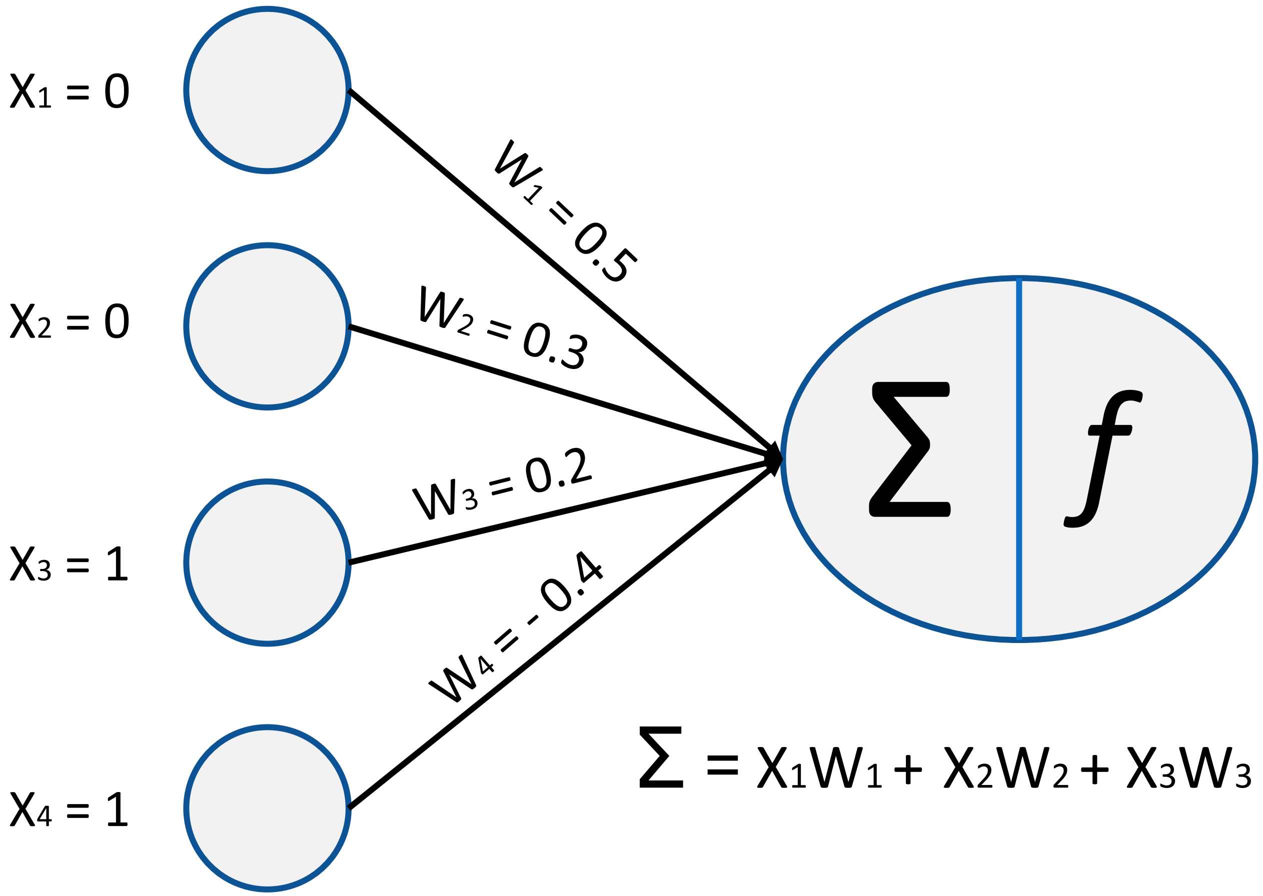
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

- The function is **differentiable:** That means, we can find the slope of the sigmoid curve at any two points.
- The function is **monotonic** but its derivative is not
- <u>The logistic sigmoid function can cause a neural network to get stuck at the training time</u>

Example 1

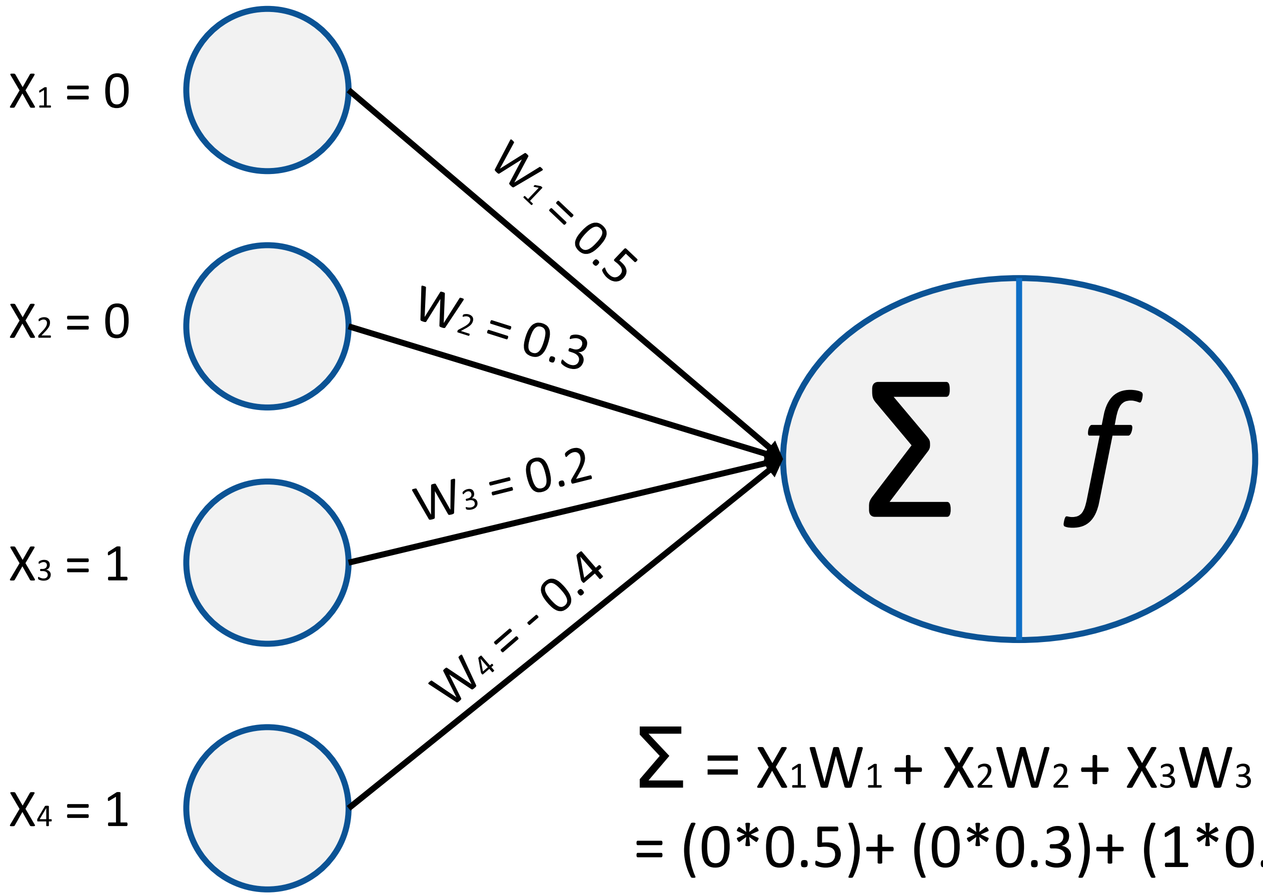


Example 1



$$\Sigma = X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4$$

Example 1

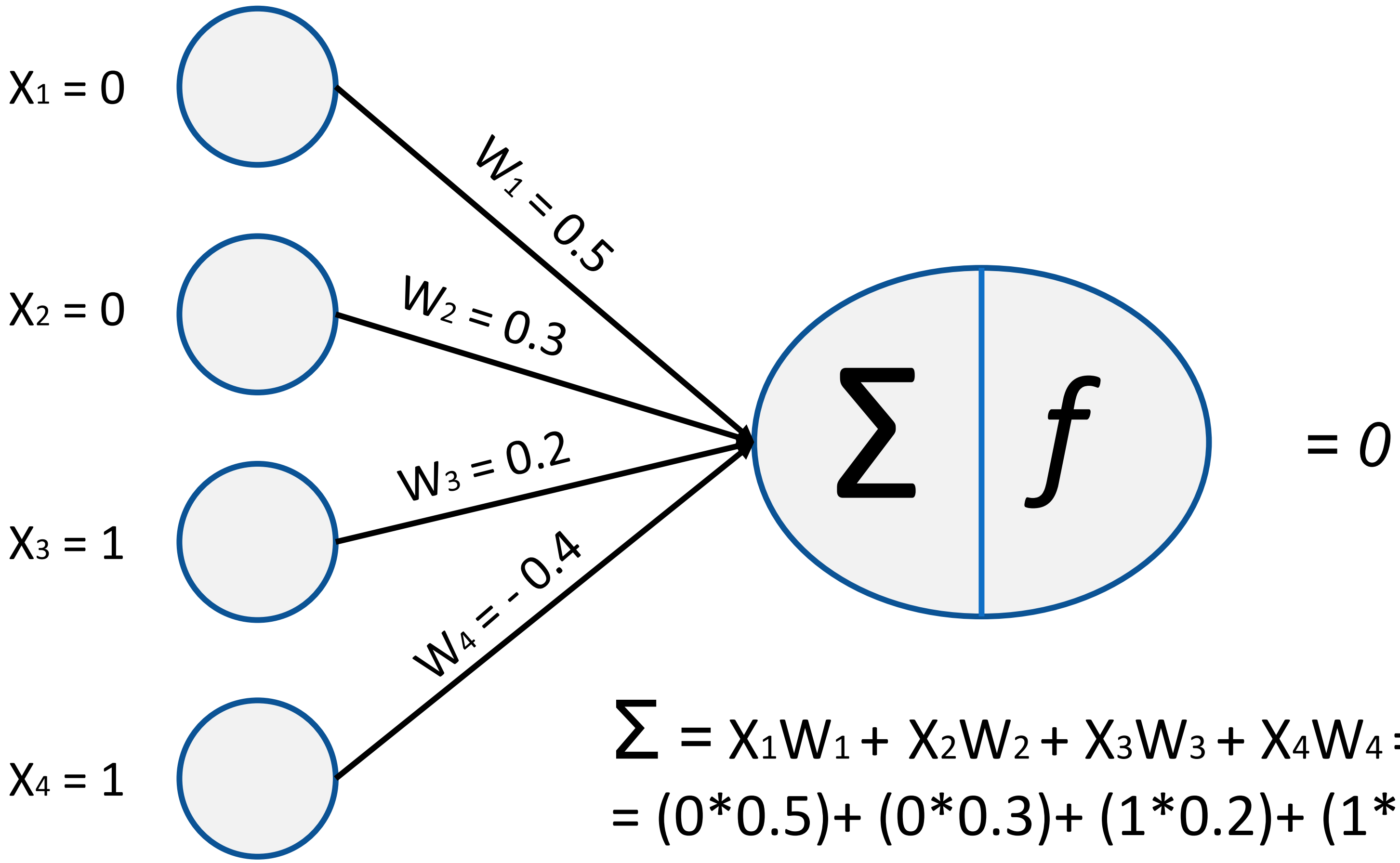


$\Sigma = X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 =$

$= (0*0.5)+ (0*0.3)+ (1*0.2)+ (1*-0.4)$

$= -0.2$

Example 1



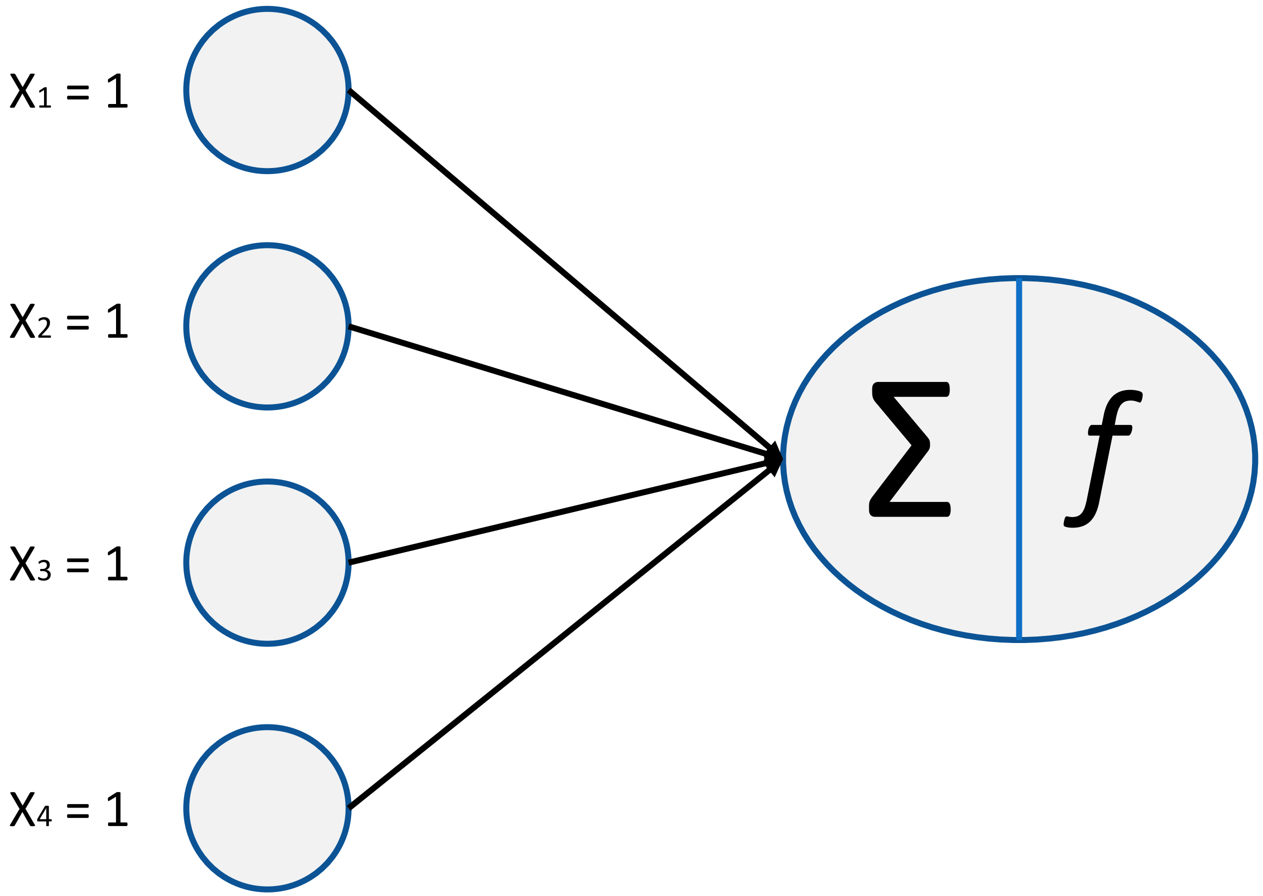
$\Sigma = X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 =$

$= (0*0.5)+ (0*0.3)+ (1*0.2)+ (1*-0.4)$

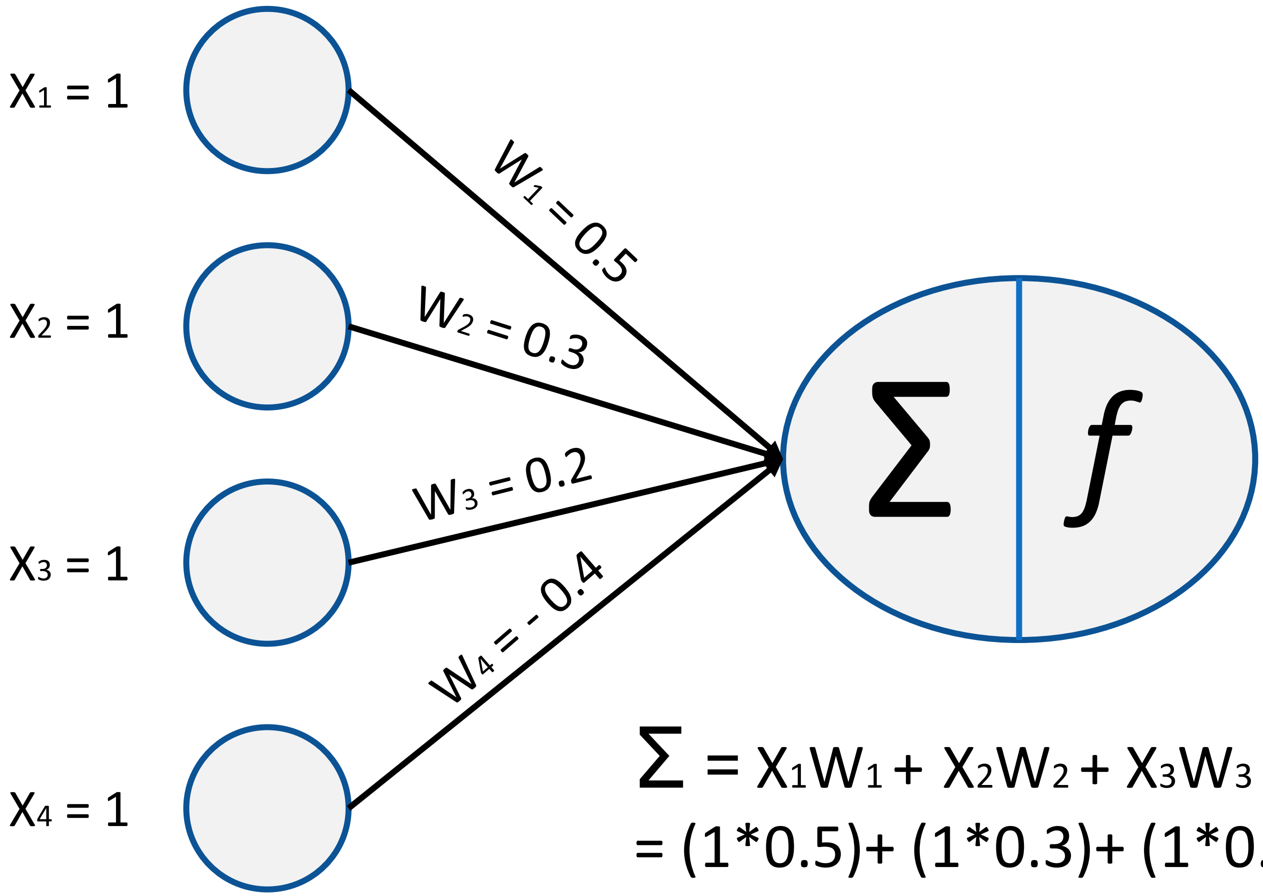
$= -0.2$

f = heavyside stepfunction(-0.2) = 0

Example 2



Example 2

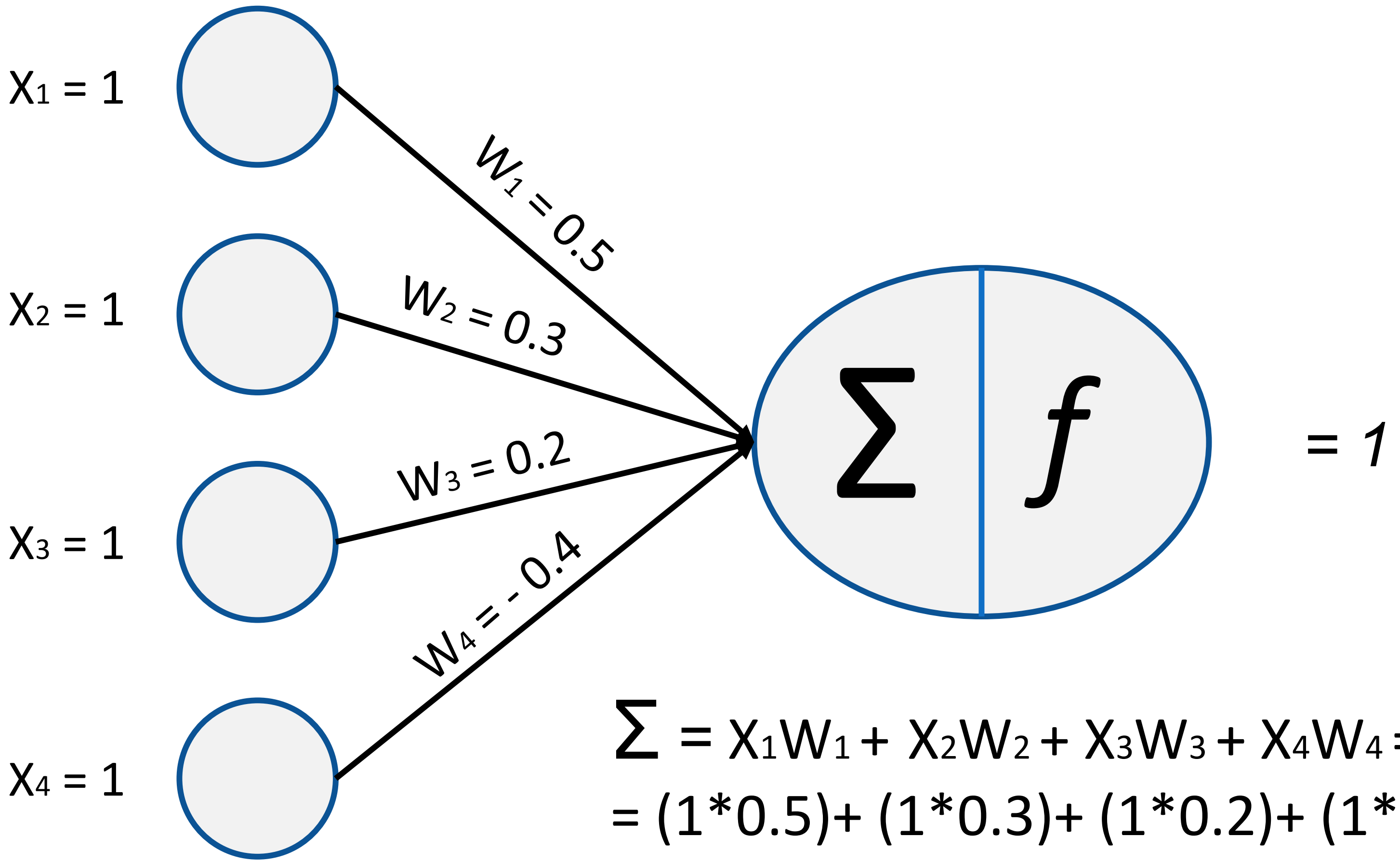


$\Sigma = X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 =$

$= (1*0.5)+ (1*0.3)+ (1*0.2)+ (1*-0.4)$

$= 0.6$

Example 2

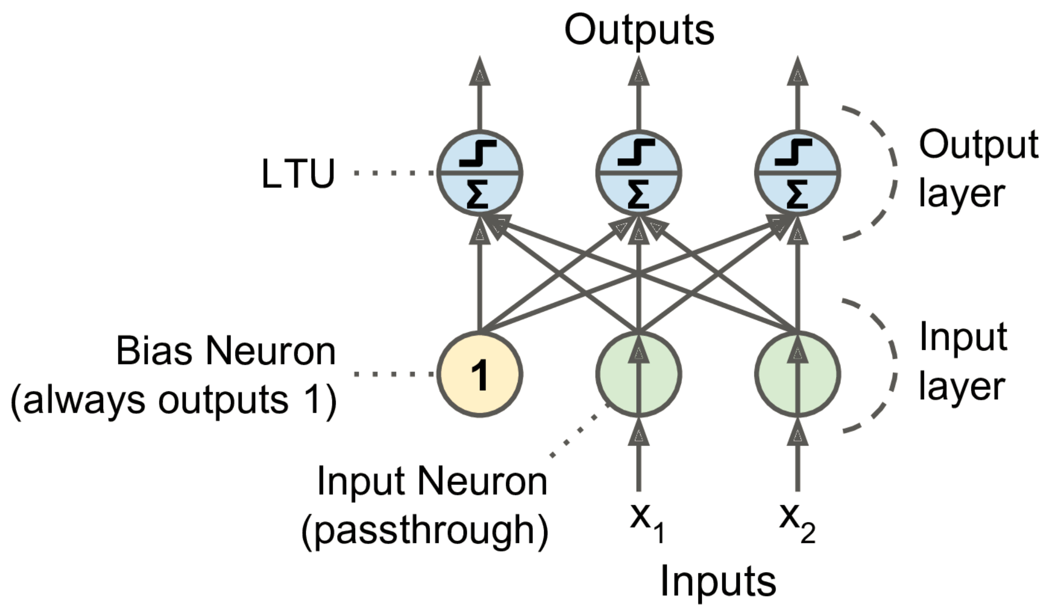


$$\Sigma = X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 =$$
$$= (1*0.5)+ (1*0.3)+ (1*0.2)+ (1*-0.5)$$
$$= 0.6$$

f = heavyside stepfunction(0.6) = 1

Multioutput Classifier

- Classifies instances simultaneously into three different binary classes



- The bias feature is represented by a special neuron called a *bias neuron*, which outputs 1 all the time

How is a Perceptron trained? (1)

- Frank Rosenblatt proposed a training algorithm inspired to Hebb's rule, who suggests that when a biological neuron often triggers another one, their connection grows stronger
  - **Hebb's rule:** the connection weight between two neurons is increased whenever they have the same output

Connection weight between the i^{th} input neuron and the j^{th} output neuron is increased $\longrightarrow$ $w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$

How is a Perceptron trained? (2)

- Frank Rosenblatt proposed a training algorithm inspired to Hebb's rule, who suggests that when a biological neuron often triggers another one, their connection grows stronger
  - **Hebb's rule:** the connection weight between two neurons is increased whenever they have the same output

Learning rate

Connection weight between the i^{th} input neuron and the j^{th} output neuron is increased

$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

Perceptron's learning rule

How is a Perceptron trained? (3)

- Frank Rosenblatt proposed a training algorithm inspired to Hebb’s rule, who suggests that when a biological neuron often triggers another one, their connection grows stronger
  - **Hebb’s rule:** the connection weight between two neurons is increased whenever they have the same output

Learning rate

The output of the j^{th} output neuron for current training instance

Connection weight between the i^{th} input neuron and the j^{th} output neuron is increased

$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

Target output of the j^{th} output neuron for current training instance

How is a Perceptron trained? (4)

- Frank Rosenblatt proposed a training algorithm inspired to Hebb’s rule, who suggests that when a biological neuron often triggers another one, their connection grows stronger
  - **Hebb’s rule:** the connection weight between two neurons is increased whenever they have the same output

Learning rate

The output of the j^{th} output neuron

Connection weight between the i^{th} input neuron and the j^{th} output neuron is increased

$$\longrightarrow w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \longleftarrow$$

The i^{th} input value

Target output of the j^{th} output neuron

How is a Perceptron trained? (5)

- Perceptrons are trained using a variant of this rule that takes into account the <u>**error**</u> made by the network
- The Perceptron is fed one training instance at a time, and for each instance it makes its predictions
- For every output neuron that produced a wrong prediction, it <u>**reinforces**</u> the connection weights from the inputs that would have contributed to the correct prediction

Perceptron Convergence Theorem

- Since the decision boundary of an output neuron is linear, Perceptrons are incapable of learning complex patterns (like Logistic Regression classifiers).
- If training instances are linearly separable, Rosenblatt demonstrated that this algorithm converges to a solution.
- This is called the Perceptron convergence theorem

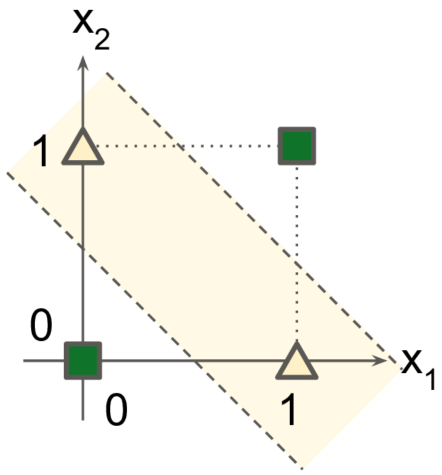
Perceptron Example

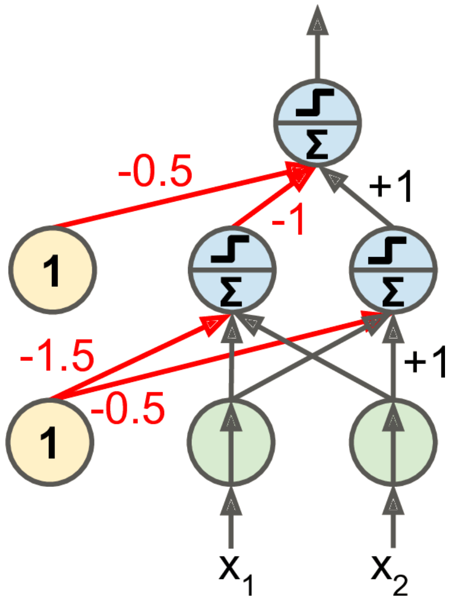
- Scikit-Learn provides a Perceptron class that implements a single LTU network
- Example with iris dataset

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]]) # petal length and width
```

Multi-Layer Perceptron (1)

- Perceptrons are incapable of solving some trivial problems (e.g. the *Exclusive OR* (XOR) classification problem)
- The limitations can often be eliminated by stacking multiple Perceptrons in a *Multi-Layer Perceptron (MLP)*
- MLP can solve the XOR problem:

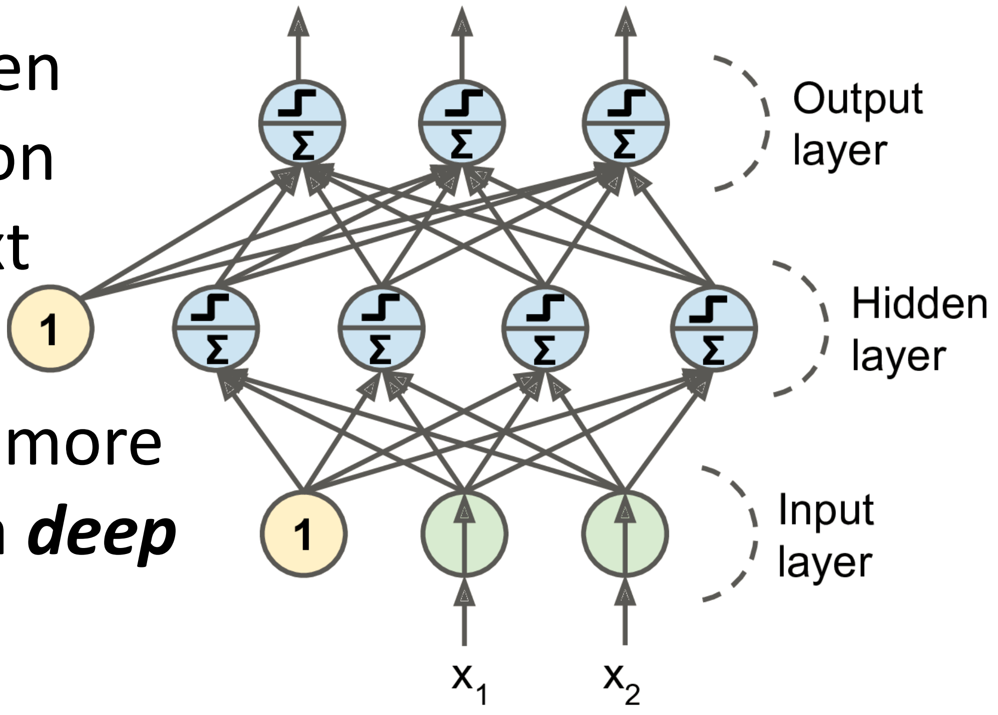




		Network output
(0,0)	→	0
(0,1)	→	1
(1,0)	→	1
(1,1)	→	0

Multi-Layer Perceptron (2)

- A MLP is composed of:
  - one (passthrough) **input layer**;
  - one or more layers of LTUs called **hidden layers;**
  - one final layer of LTUs called the **output layer.**
- Input layer and each hidden layers include a bias neuron fully connected to the next layer
- When an ANN has two or more hidden layers, it is called a ***deep neural network*** (DNN)



Backpropagation (1)

- **Problem**: How to train MLPs?
- In 1986 D. E. Rumelhart et al. published a groundbreaking article introducing the *backpropagation* training algorithm
- Today we would describe it as Gradient Descent using reverse-mode autodiff

Backpropagation (2)

- Training phases:
  - For each training instance the algorithm:
    - Feeds the instance in the network
    - Computes the output of every neuron in each consecutive layer (forward pass)
    - Measures the network's output error: the difference between the desired output and the actual output
    - Computes how much each neuron in the last hidden layer contributed to each output neuron's error, until it reaches the input layer
    - Measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network
  - The last step of the backpropagation algorithm is a Gradient Descent step on all the **connection weights** in the network, using the error gradients measured earlier

Backpropagation (3)

- In other words, for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step)
- To make it work properly, authors replaced the step function with the logistic function:

$$\sigma(z) = 1 / (1 + \exp(-z))$$

- Because the step function contains only flat segments, so there is no gradient to work with, while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step

Alternative MLP’s Step Functions

- Two other popular activation functions are:
  - The hyperbolic tangent function:

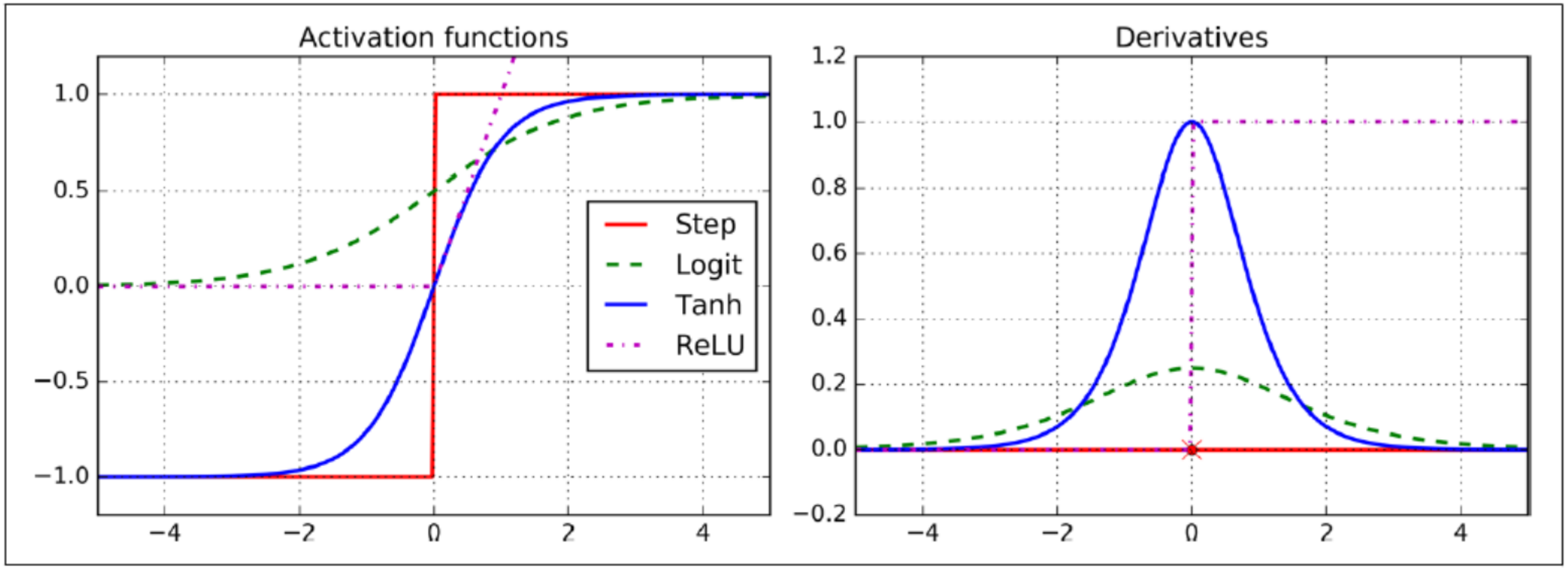
    $$\tanh(z) = 2\sigma(2z) - 1$$

    - like the logistic function it is S-shaped, continuous, and differentiable
    - Its output ranges from –1 to 1 (instead of 0 to 1), which tends to make each layer’s output more or less normalized (i.e., centered around 0) at the beginning of training, which often helps speed up convergence
  - The ReLU function:

    $$\mathrm{ReLU}(z) = \max(0, z)$$

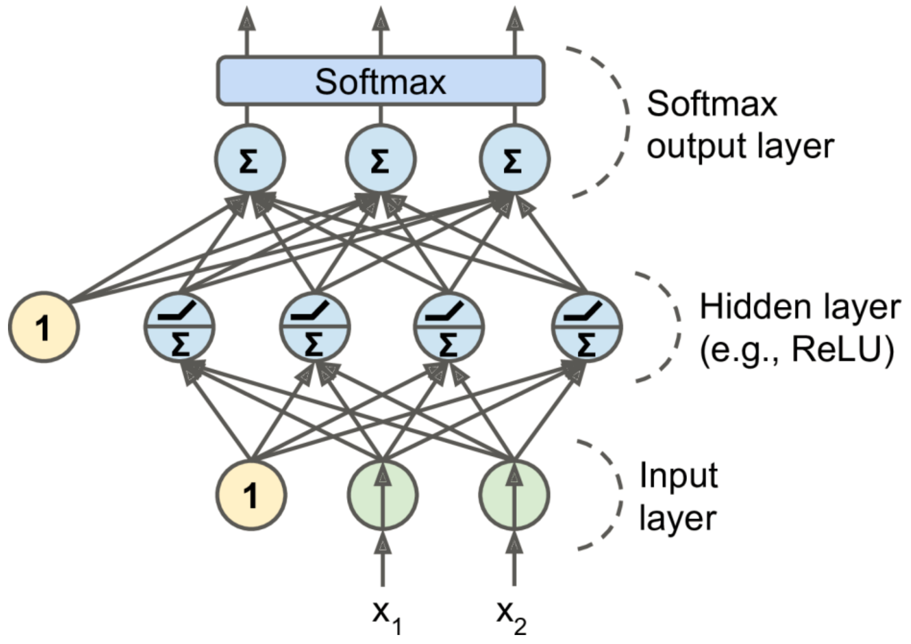
    - It is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around)
    - In practice works very well and is fast to compute

Tanh and ReLU Step Functions



Classification

- Binary Classification
  - Each output corresponds to a different binary class:
    - Spam/Ham
    - Urgent/Not-urgent
    - Ecc.
- General Classification
  - When the classes are exclusive the output layer is typically modified by replacing the individual activation functions by a shared ***softmax*** function



Biological neurons seem to follow a roughly sigmoid (Sshaped) activation function, so researchers for long stuck to sigmoid functions. But ReLU generally works better in ANNs. This is one case where the biological analogy was misleading

Softmax function

- Softmax function is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for \; j = 1, \ldots, k$$

- Transforms the values in the range **[0,1]** that add up to **1**
- Generally, the softmax activation function is used in the last layer (e.g. classification of handwritten digits)
- We choose the class with the highest probability

TensorFlow’s High-Level API

- The simplest way to train an MLP with TensorFlow is to use the high-level API TF.Learn
- The **DNNClassifier** class permits to train a deep NN with any number of hidden layers and a softmax output layer to output class probabilities
- Example: the following code trains a DNN with:
  - Two hidden layers: one with 300 neurons, and the other with 100
  - a softmax output layer with 10 neurons

```
import tensorflow as tf
feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                         feature_columns=feature_cols)
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # wrap DNNClassifier in a Scikit-Learn compatibility helper
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000) # 40,000 training iterations using 50 instances
```

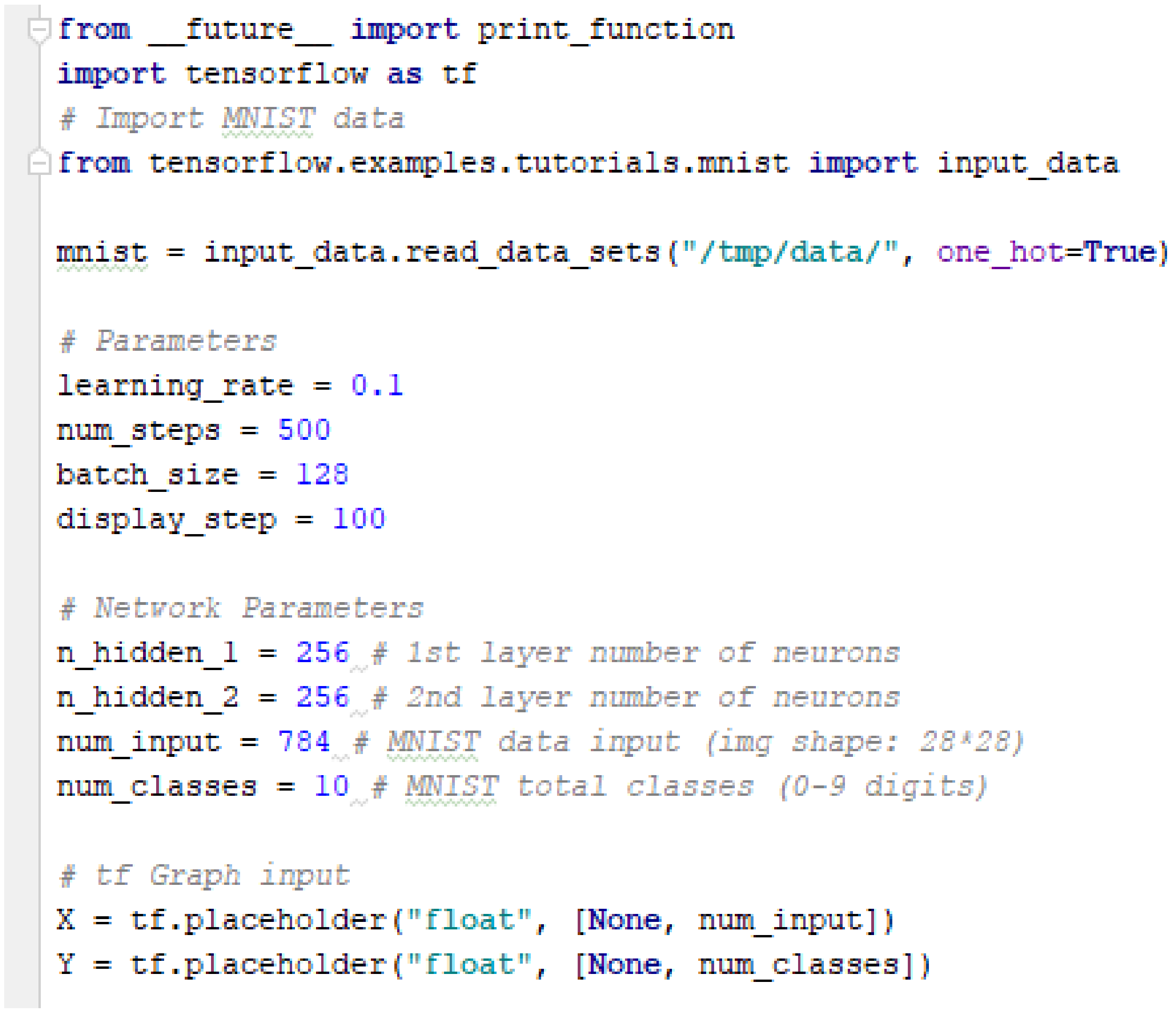
Training a DNN Using Plain TensorFlow

- If we want more control on the architecture of the NN, we might use TensorFlow’s lower-level Python API (introduced in Chapter 9).
- We will use this API to implement the Minibatch Gradient Descent and train it on the MNIST (handwritten digits) dataset.
- First step we build the TensorFlow graph
- Second step we run the graph to train the model

Training a DNN Using Plain TensorFlow

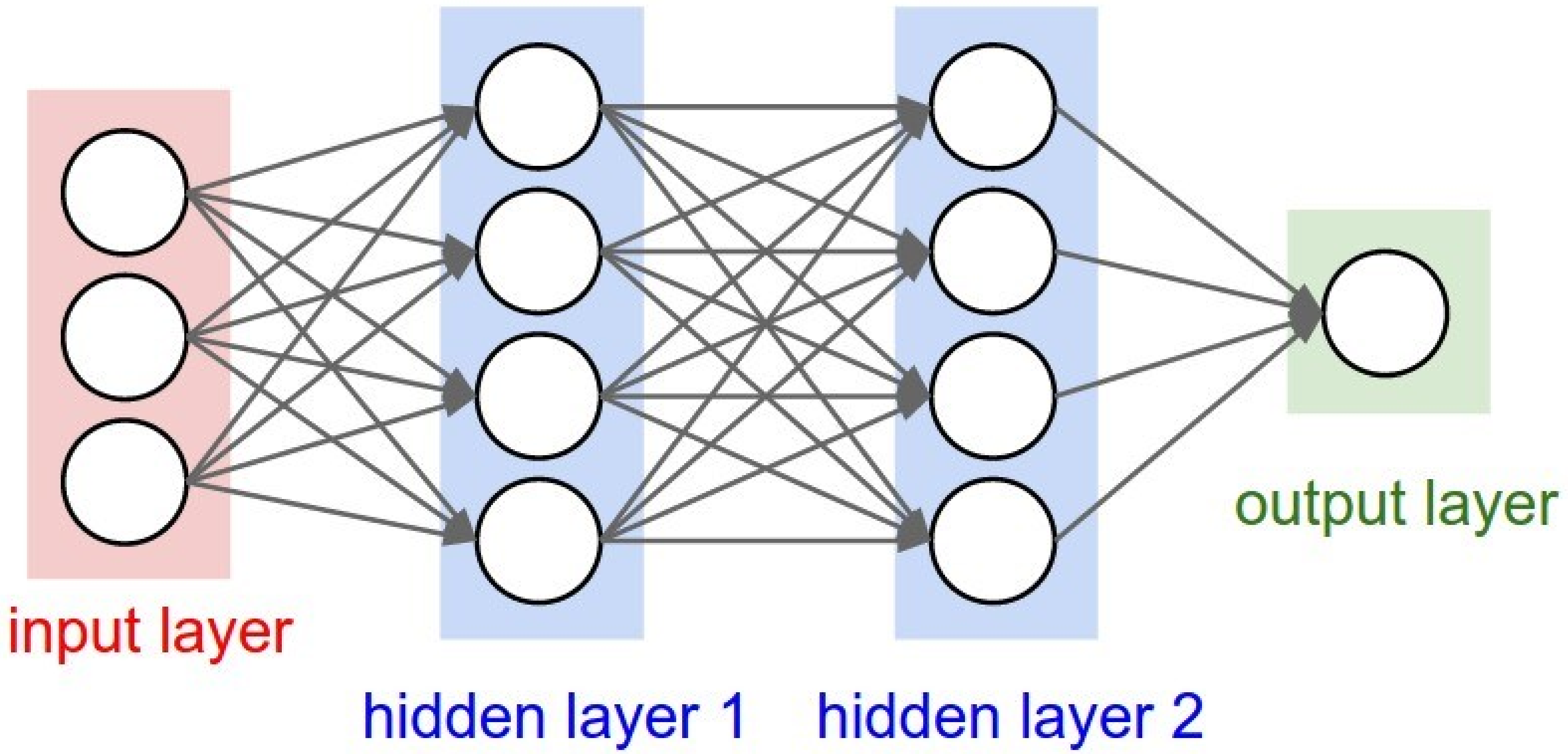
- This example is using MNIST handwritten digits
- The dataset contains:
  - 60,000 examples for training
  - 10,000 examples for testing
- The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1
- Each image has been flattened and converted to a 1-D NumPy array of 784 features (28*28)

Example: Construction Phase (1)

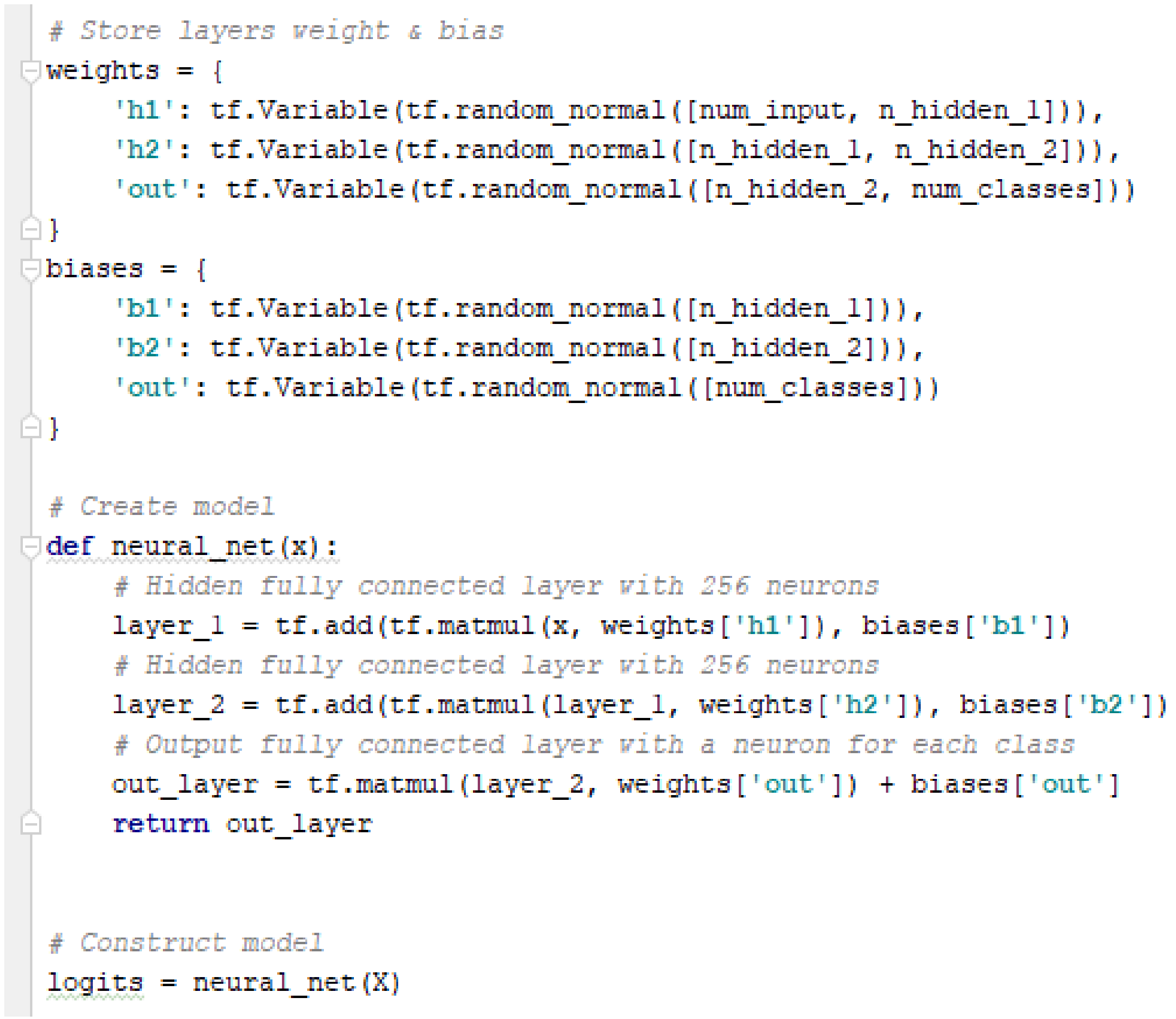


```python
from __future__ import print_function
import tensorflow as tf
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Parameters
learning_rate = 0.1
num_steps = 500
batch_size = 128
display_step = 100

# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

Neural Network Example (2)



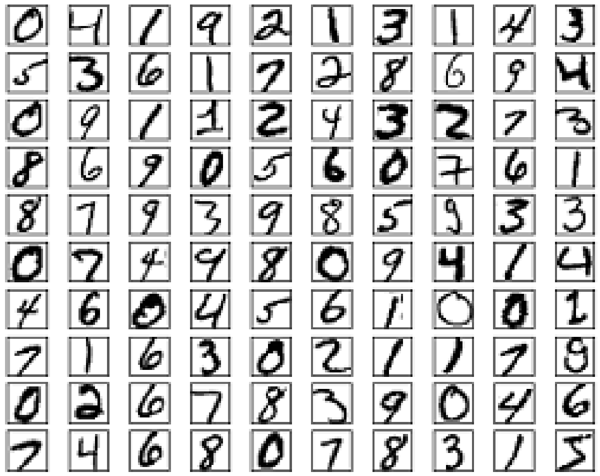
Example: Layers Initialization (3)



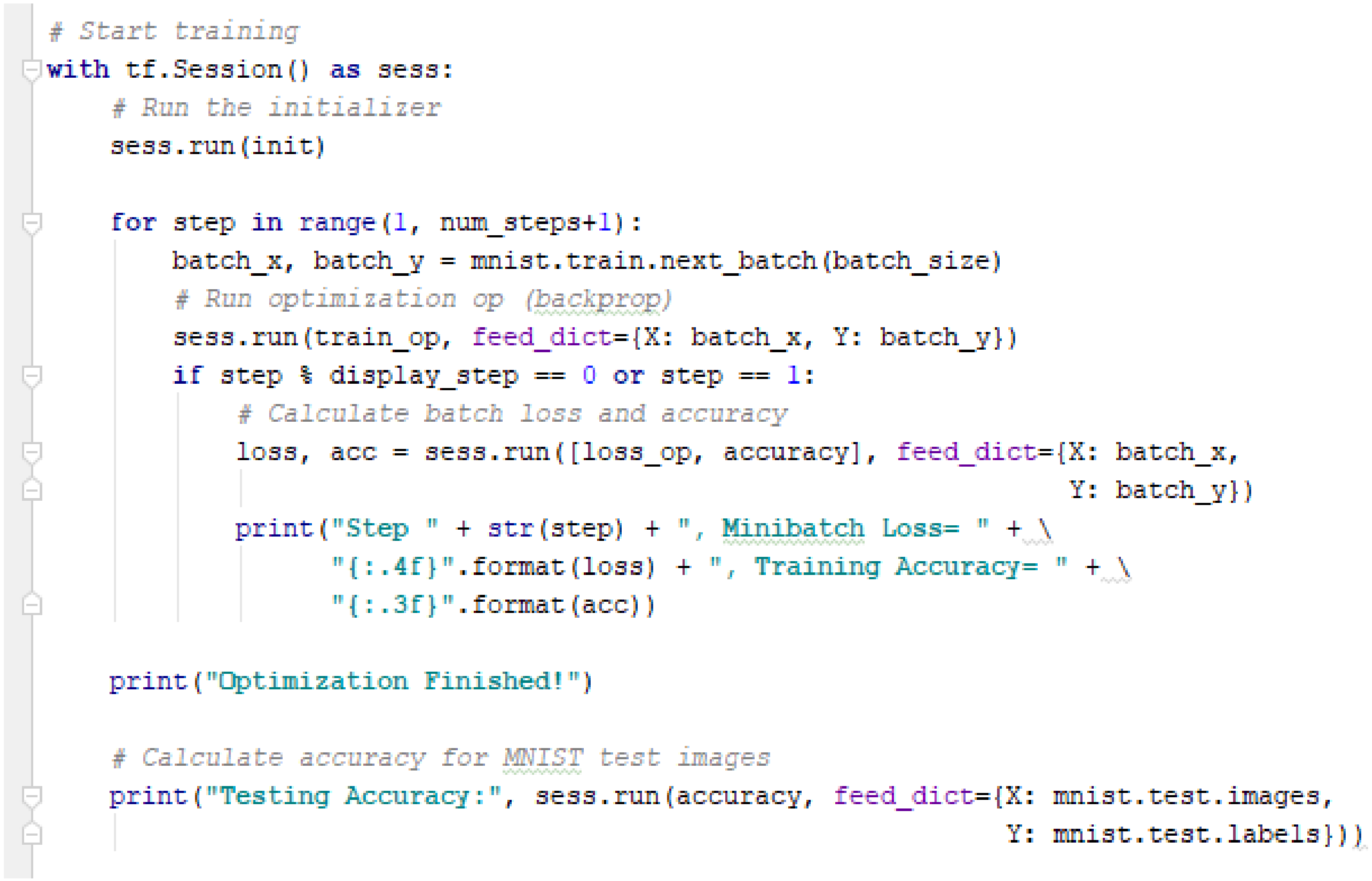
```python
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}

# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer


# Construct model
logits = neural_net(X)
```

Example: Optimizer (4)

```
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```



Example: Execution Phase (5)



```python
# Start training
with tf.Session() as sess:
    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: mnist.test.images,
                                                             Y: mnist.test.labels}))
```