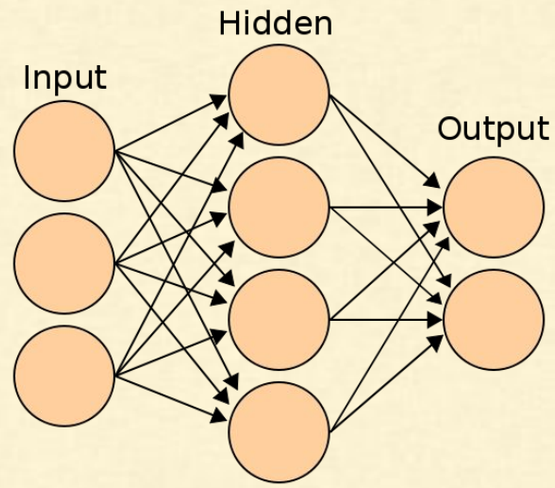
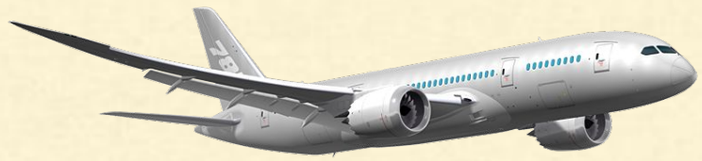
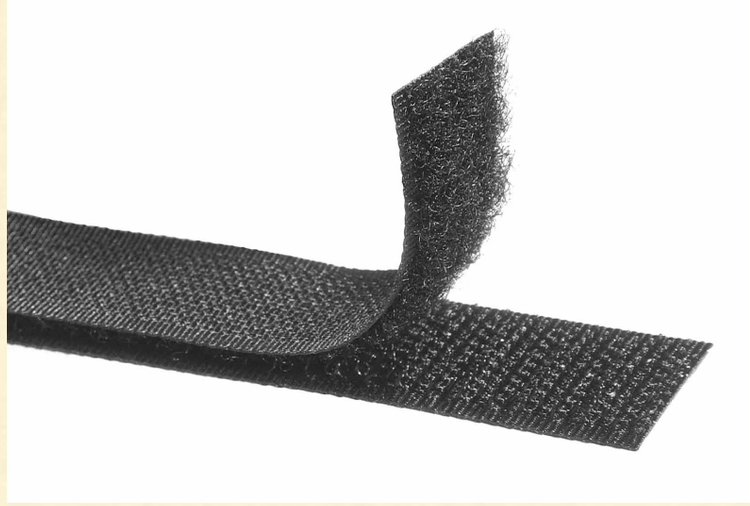

Hidden
Input
Output

Inspirations From Nature

Birds inspired us to fly



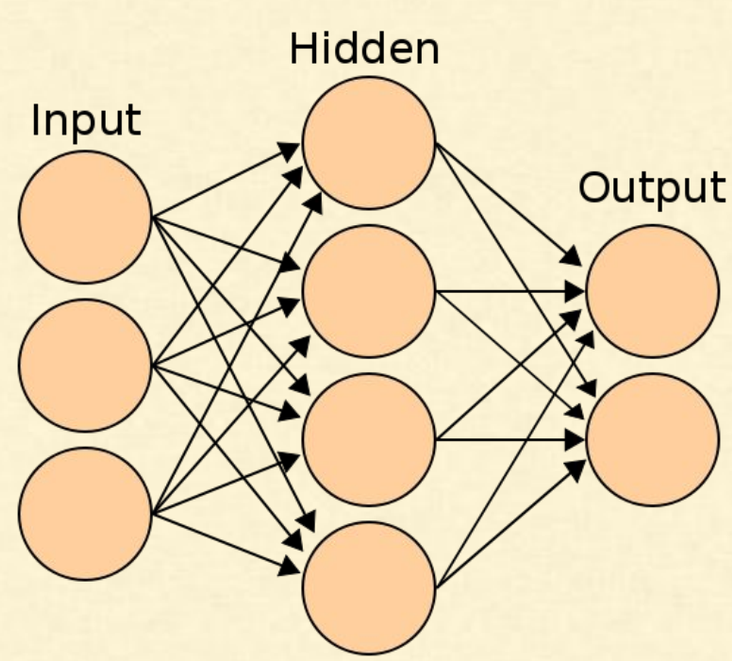
Burdock plants inspired Velcro



Inspiration from the Brain

- It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine
- This is the key idea that inspired **Artificial Neural Networks (ANNs)**

Your Brain



Artificial Neural Networks

Power of Artificial Neural Networks

- ANNs are at the very core of Deep Learning
- They are
  - Versatile
  - Powerful
  - Scalable
  - And ideal to tackle
    - Large and highly complex
    - Machine Learning tasks

Applications of Artificial Neural Networks

- Classifying billions of images
  - Google Images

Applications of Artificial Neural Networks

- Powering speech recognition services
  - Apple's Siri

Applications of Artificial Neural Networks

- Recommending the best videos to watch
  - To hundreds of millions of users every day
  - E.g., YouTube

Applications of Artificial Neural Networks

- Learning to beat the world champion
  - At the game of Go
  - By examining millions of past games and
  - Then playing against itself
  - E.g., DeepMind's AlphaGo

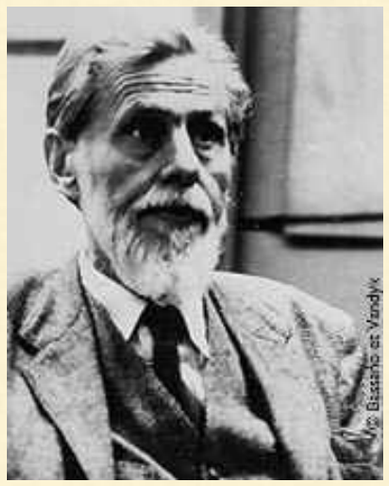
What we will learn?

In this topic we will cover -

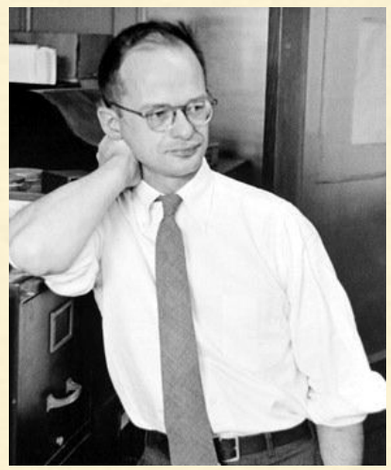
- Introduction to **Artificial Neural Networks**
- Tour of the very first **ANN architectures**
- Multi-Layer Perceptrons (MLPs)
- Implement one using **TensorFlow** to tackle the **MNIST** digit classification problem

History of Artificial Neural Networks

- Artificial Neural Networks were
  - First introduced in **1943** by
    - The neurophysiologist **Warren McCulloch** and
    - The mathematician **Walter Pitts**
  - In the paper "**A Logical Calculus of Ideas Immanent in Nervous Activity**"



Warren McCulloch



Walter Pitts

History of Artificial Neural Networks

- They presented a model of
  - How biological neurons might work together
  - In animal brains to perform
  - Complex computations
- This was the first artificial neural network architecture

- Since then many other architectures have been invented

History of Artificial Neural Networks

- The early successes of ANNs until **1960s** led to the
  - Belief that we would soon be conversing with intelligent machines
  - But due to various reasons (which we will cover shortly)
    - ANNs entered a long dark era

History of Artificial Neural Networks

- In the early **1980s** there was a revival of interest in ANNs
  - As new network architectures were invented
  - And better training techniques were developed

History of Artificial Neural Networks

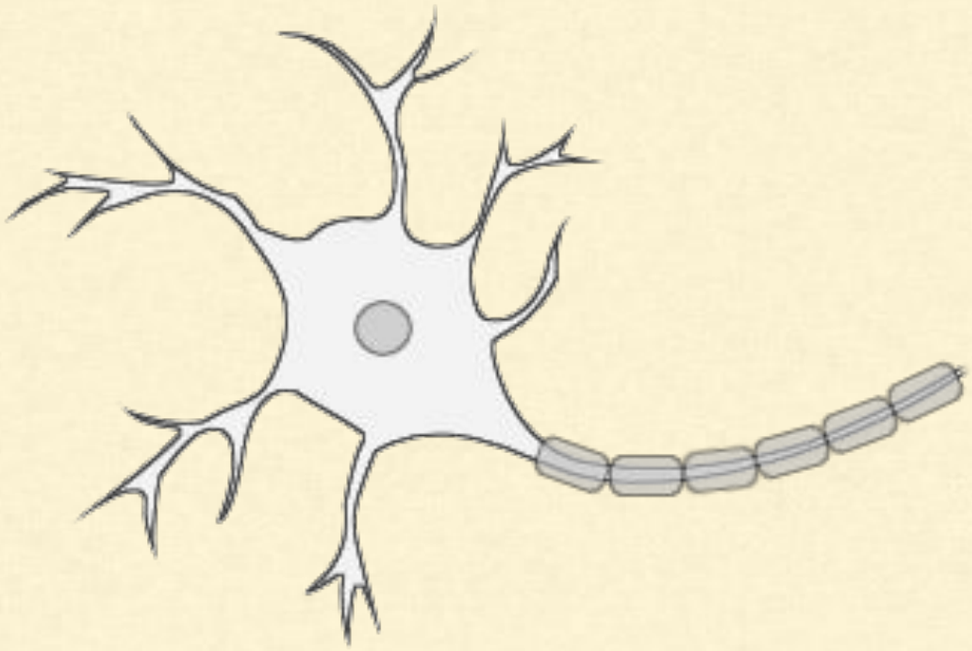
- But by the **1990s**, powerful alternative Machine Learning techniques
  - Such as Support Vector Machines were favored by most researchers
  - As they seemed to offer better results and
  - Stronger theoretical foundations

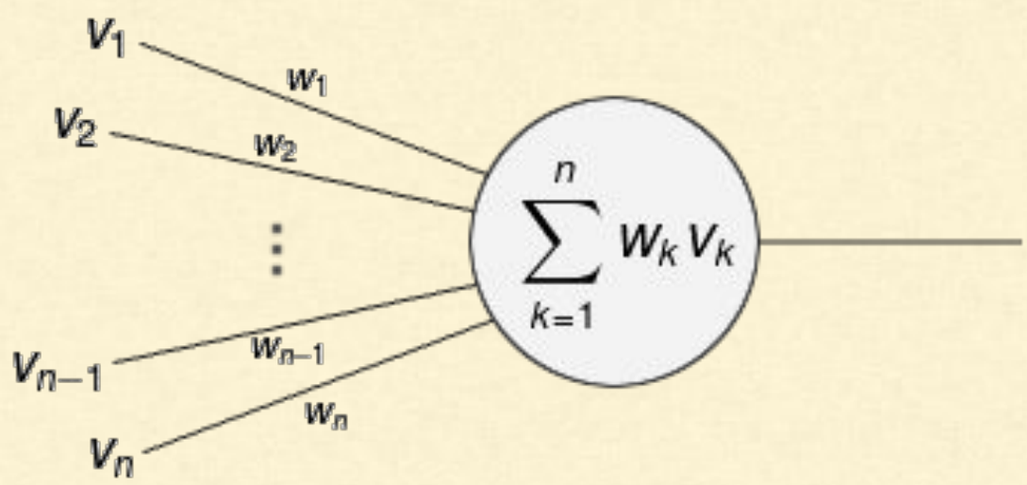
Why ANN's are relevant today?

- Huge quantity of data available
  - To train neural networks
- ANNs frequently outperform
  - Other ML techniques on
  - Very large and complex problems
- Tremendous increase in computing power
  - Which makes it possible to train large neural networks
  - In a reasonable amount of time
  - Also, gaming industry
    - Produced powerful GPU cards

Logical Computations with Neurons

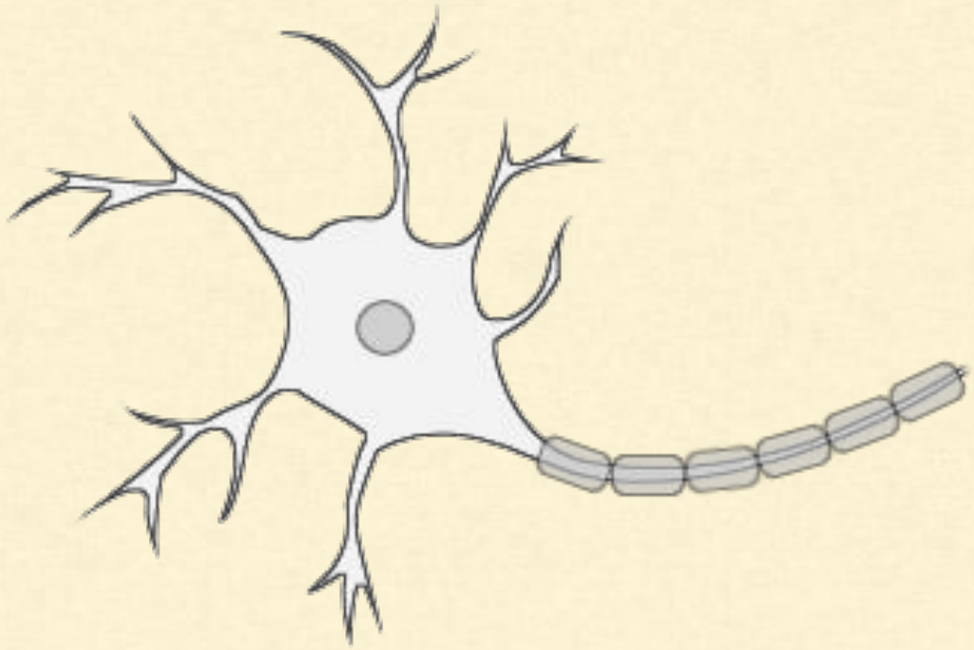
- **Warren McCulloch** and **Walter Pitts** proposed
  - A very simple model of the biological neuron
  - Known as an **artificial neuron**
- It has one or more binary (on/off) inputs and one binary output.

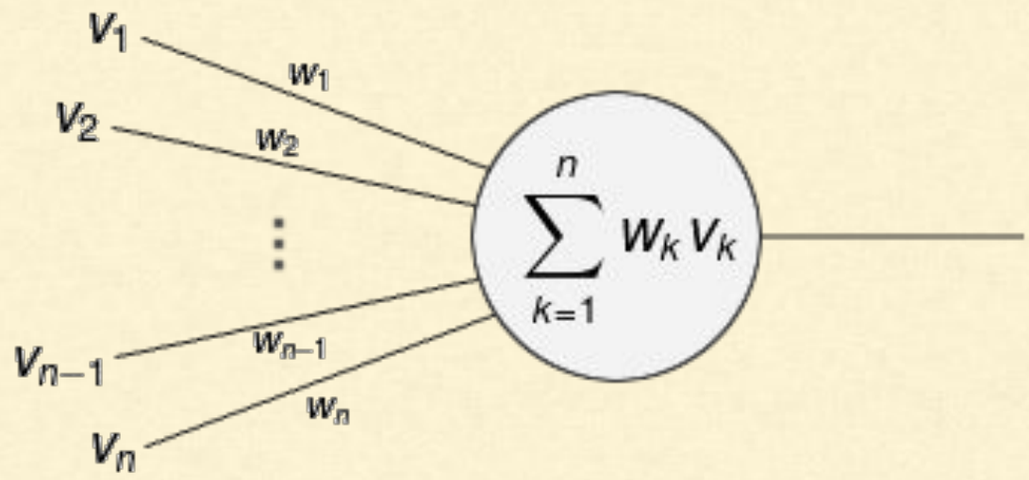




Logical Computations with Neurons

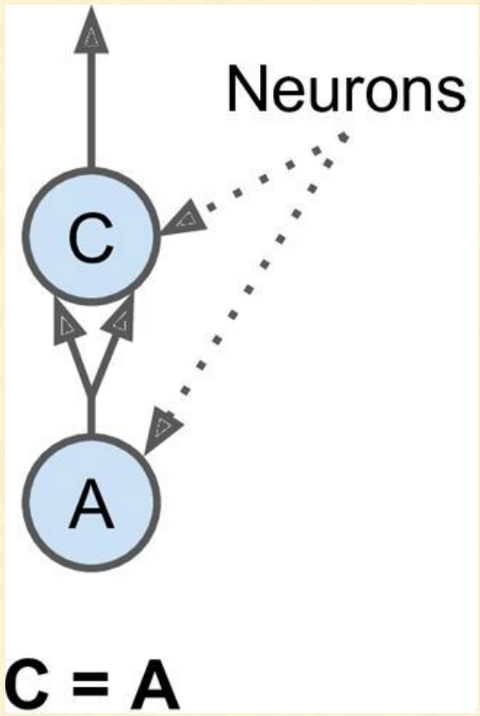
- The artificial neuron simply activates
  - Its output when more than
  - A certain number of its inputs are active





Logical Computations with Neurons

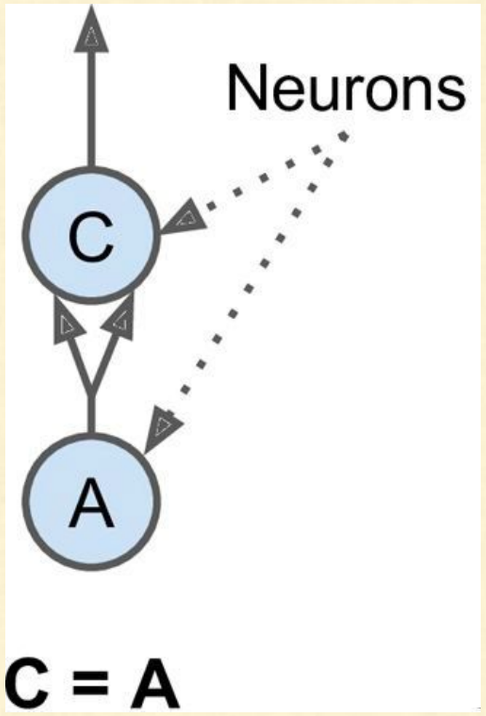
- Let's build a few ANNs that perform various logical computations
- Here we will assume that a neuron is activated when **at least two** of its inputs are active



Logical Computations with Neurons

ANN - 1

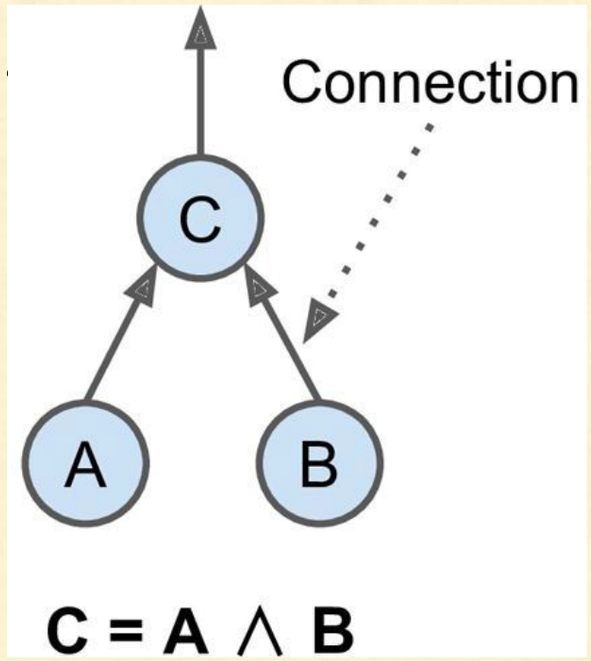
- If neuron A is activated then
  - Neuron C gets activated as well
    - It receives two input signals from neuron A
  - But if neuron A is off, then neuron C is off as well



Logical Computations with Neurons

ANN - 2

- It performs a logical AND
- Neuron C is activated only
    - When both neurons A and B are activated
- A single input signal is not enough
    - To activate neuron C

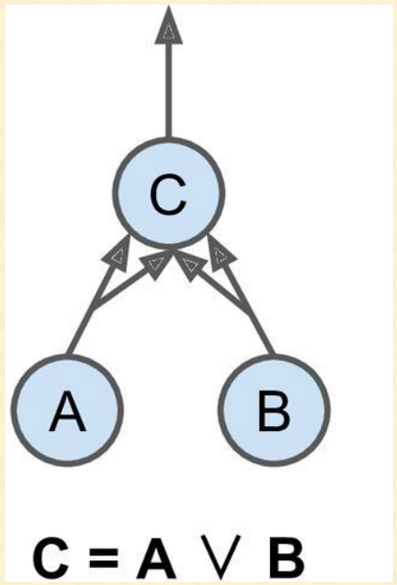


∧ = AND

Logical Computations with Neurons

ANN - 3

- It performs a logical OR
- Neuron C gets activated if either
  - Neuron A or
  - Neuron B is activated
  - Or both

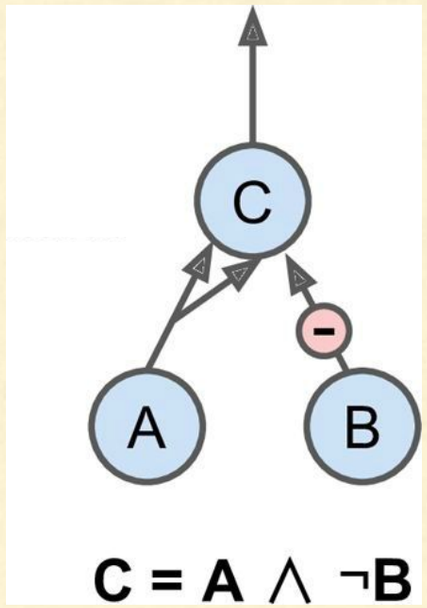


V = OR

Logical Computations with Neurons

ANN - 4

- Neuron C is activated only
  - If neuron A is active and neuron B is off
- If neuron A is active all the time
  - Then we get a logical NOT
- Neuron C is active when neuron B is off and vice versa



¬ = NOT

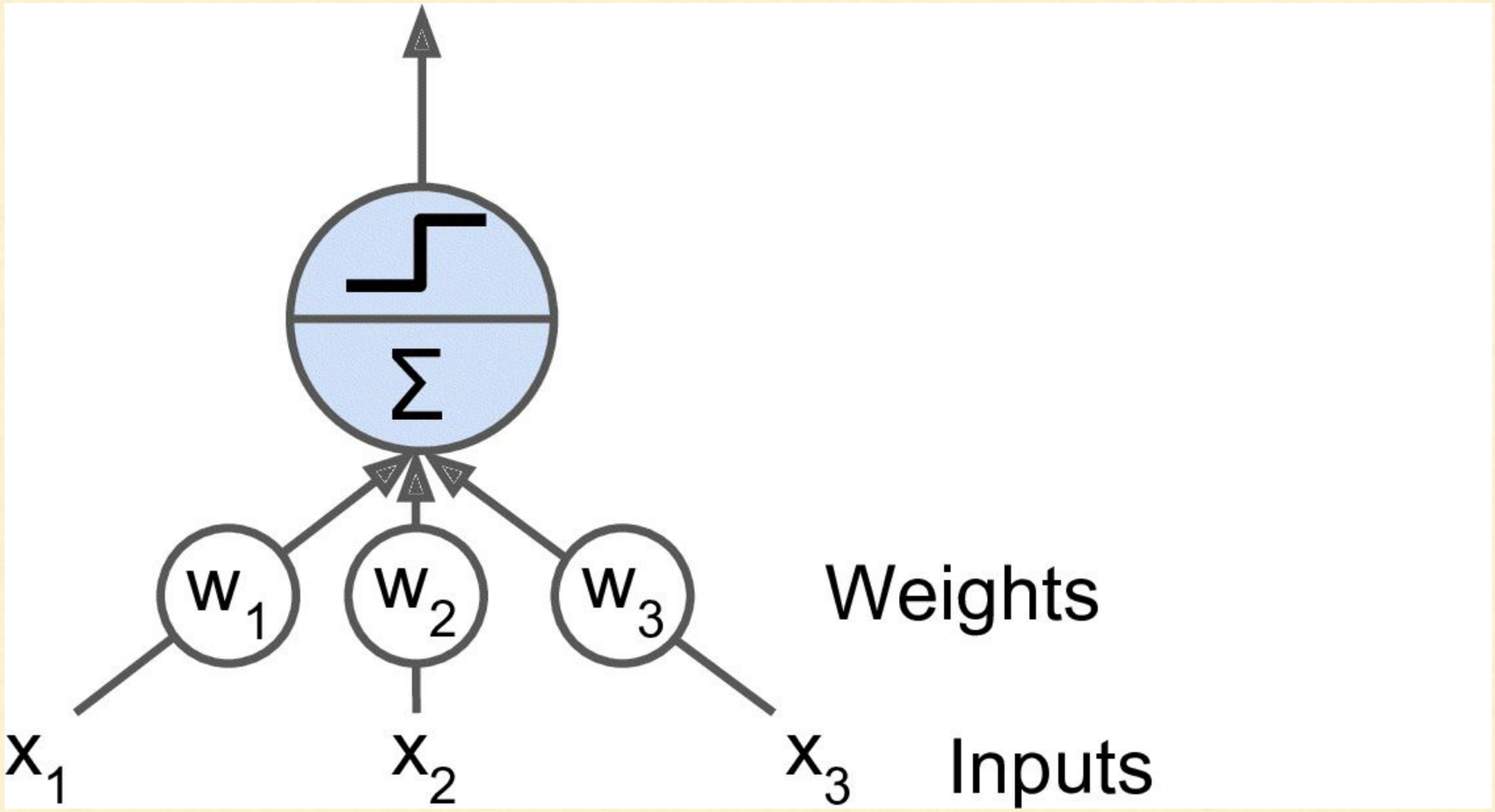
The Perceptron

- It is one of the simplest ANN architectures
- Invented in 1957 by Frank Rosenblatt
- Based on a slightly different artificial neuron called a **linear threshold unit (LTU)**

The Perceptron

Linear threshold unit

- The inputs and output are now numbers, instead of binary on/off values
- Each input connection is associated with a weight



The Perceptron

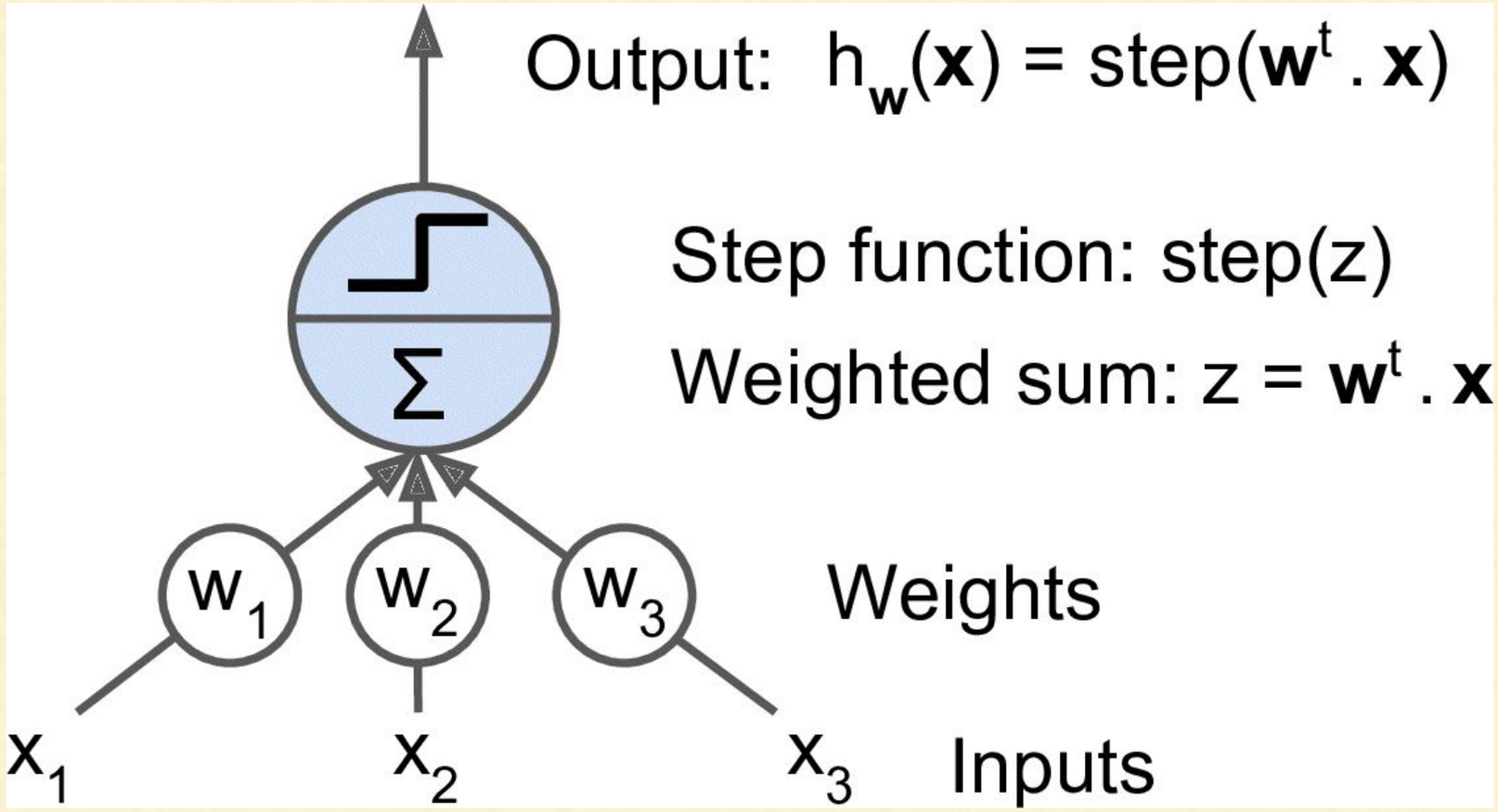
Linear threshold unit

- It computes a weighted sum of its inputs

$$z = w1\ x1 + w2\ x2 + \cdots + wn\ xn = wT \cdot x$$

- Then applies a step function to that sum and outputs the result

$$hw(x) = step\ (z) = step\ (wT \cdot x)$$



The Perceptron

Linear threshold unit

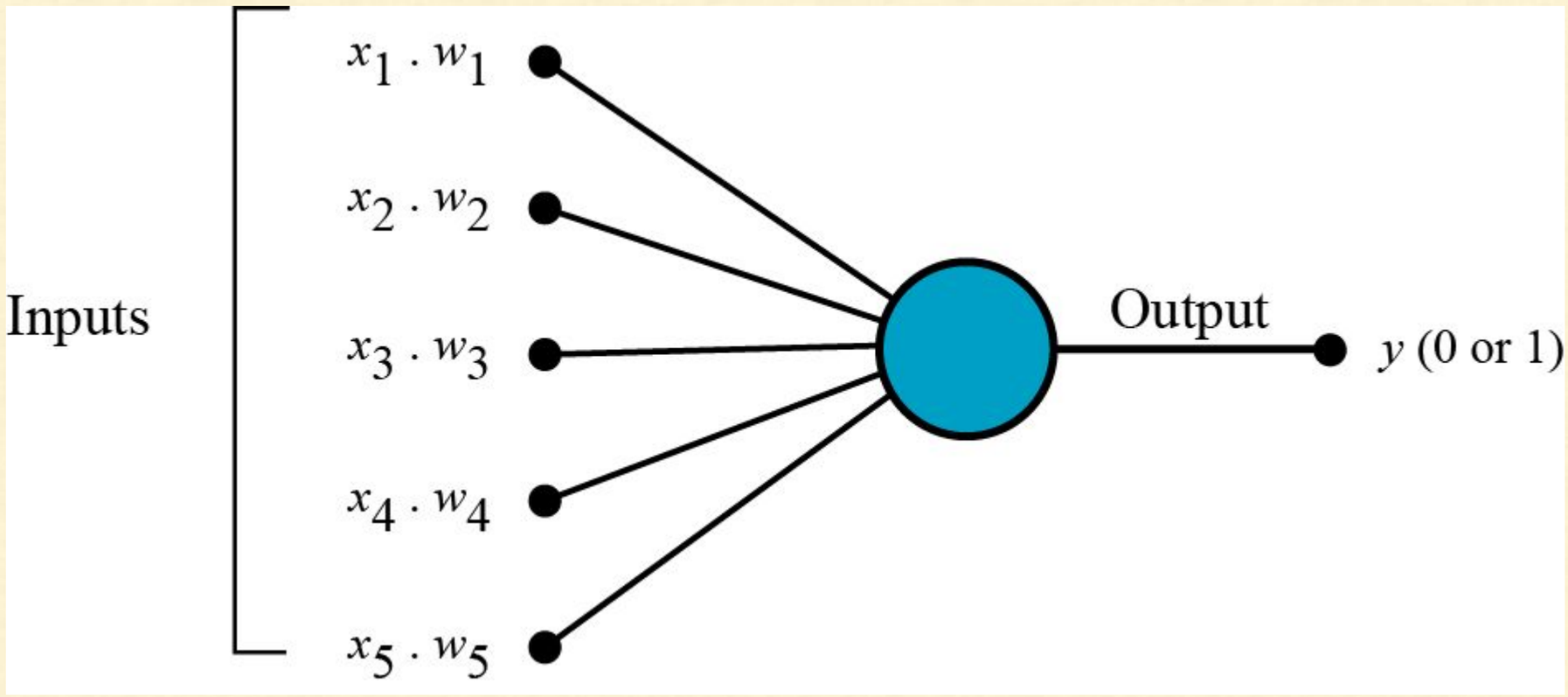
- The most common step function used in Perceptrons is the **Heaviside** step function
- Sometimes the **sign** function is used instead

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

The Perceptron

Linear threshold unit

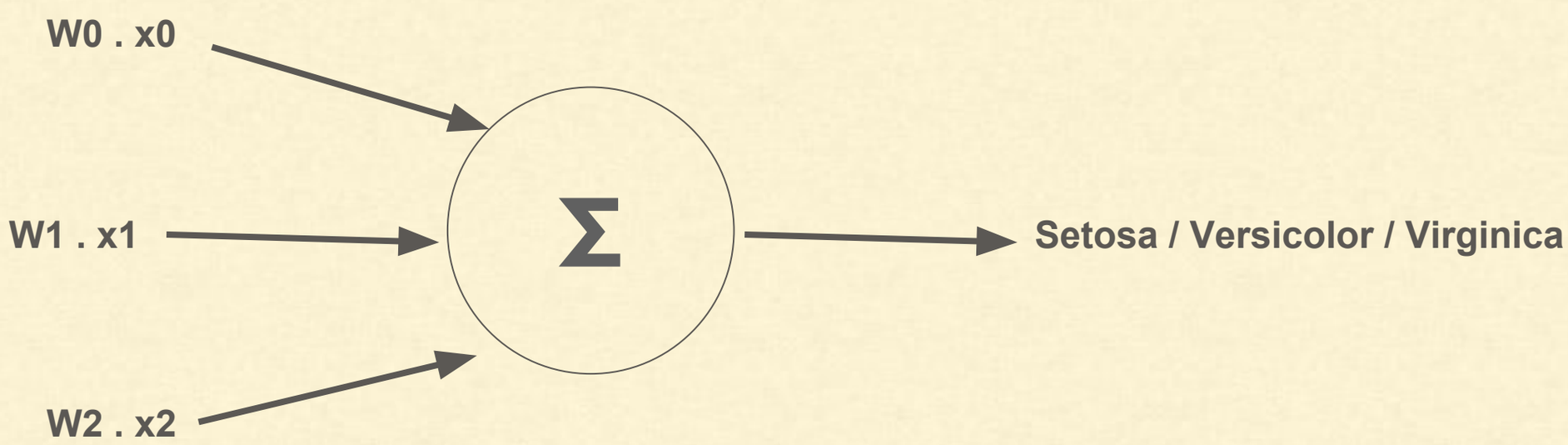
- A single LTU can be used for simple linear binary classification.
- It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class
- Just like a Logistic Regression classifier or a linear SVM



The Perceptron

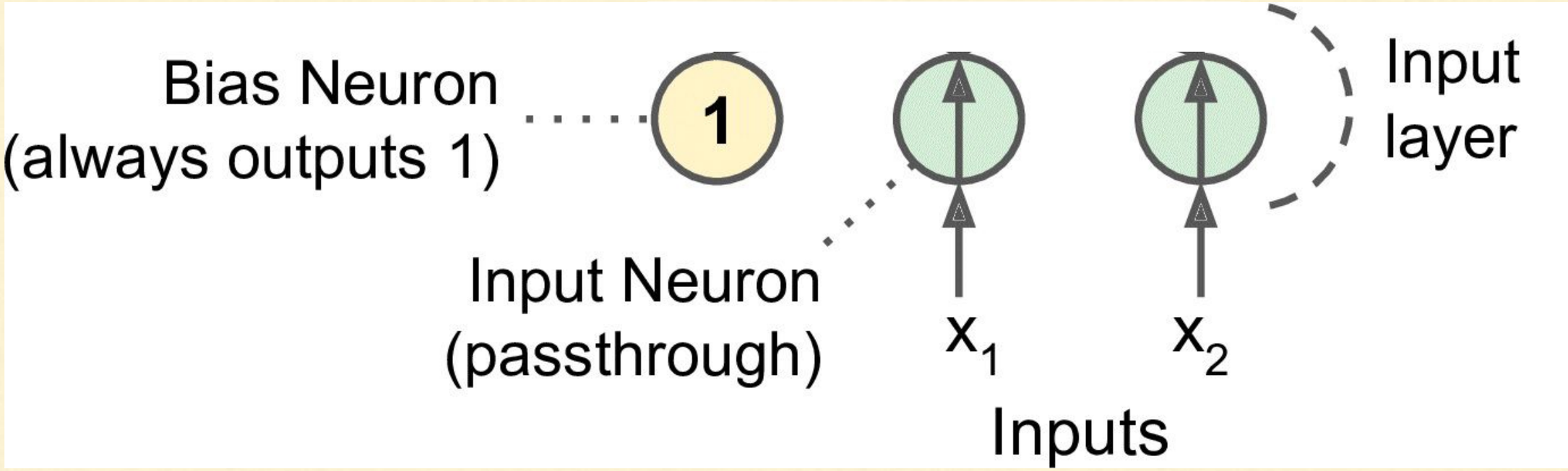
Linear threshold unit

- For example, you could use a single LTU to classify iris flowers based on the petal length and width
- And adding an extra bias feature x0 = 1
- Training an LTU means finding the right values for w0, w1, and w2



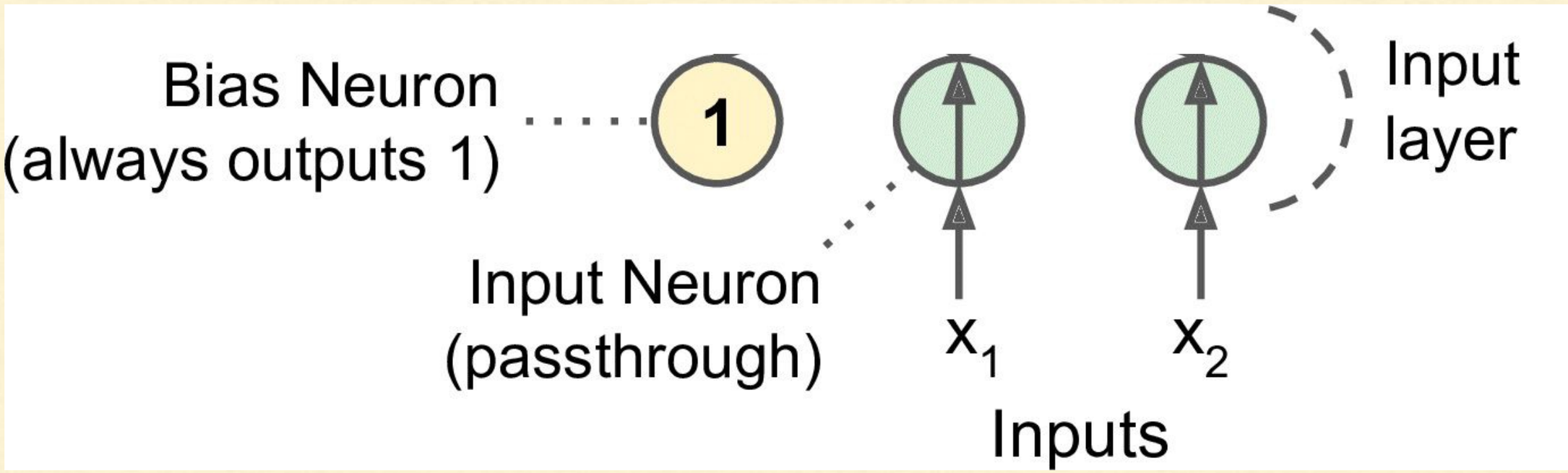
The Perceptron

- A Perceptron is simply composed of a single layer of LTUs, with each neuron connected to all the inputs.
- These connections are often represented using special passthrough neurons called **input neurons**

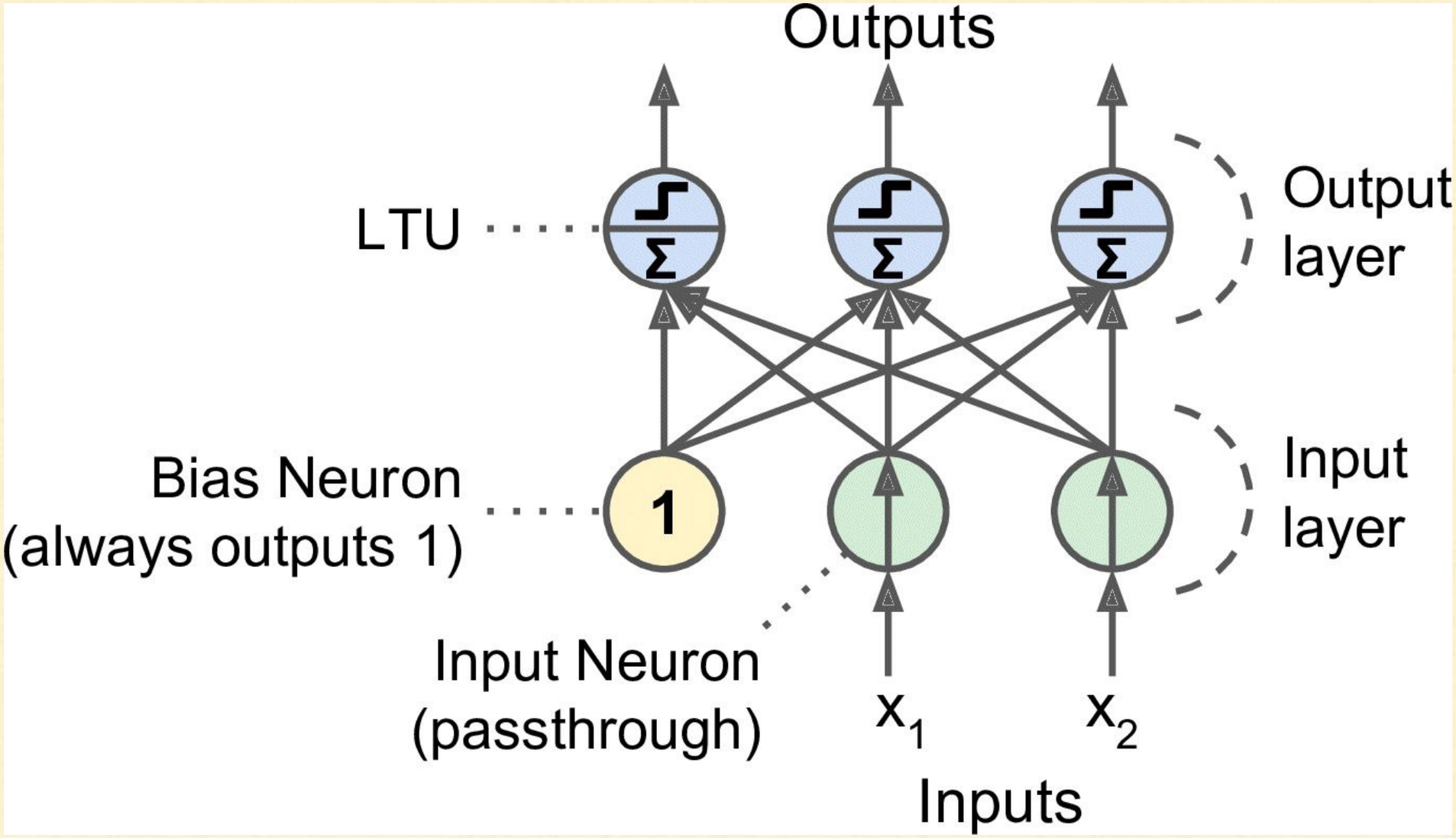


The Perceptron

- Input neurons just output whatever input they are fed
- An extra bias feature is generally added (x0 = 1)
- This bias feature is typically represented using a special type of neuron called a **bias neuron**, which just outputs **1** all the time



The Perceptron



- Above figure shows a Perceptron with two inputs and three output
- This Perceptron can classify instances simultaneously into three different binary classes, which makes it a **multioutput classifier**.

Training a Perceptron

Hebb's Rule

- In order to understand how we train a perceptron we need to understand Hebb's rule.
- In his book **The Organization of Behavior**, published in 1949, **Donald Hebb** suggested that when a biological neuron often triggers another neuron, the connection between these two neurons grows stronger.

The idea was later summarized by Siegrid Löwel in this catchy phrase:

"Cells that fire together, wire together."

Training a Perceptron

Perceptrons are trained using a variant of this rule that takes into account the error made by the network; it does not reinforce connections that lead to the wrong output.

The main steps are:

- The Perceptron is fed one training instance at a time
- For each instance it makes its predictions
- For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction

Training a Perceptron

Perceptron learning rule (weight update)

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

- $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate.

Training a Perceptron

Perceptron convergence theorem

- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns.
- However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.
- This is called the **Perceptron convergence theorem**.

Training a Perceptron

Let's build our own perceptron using Scikit-Learn
We will test it on the iris dataset

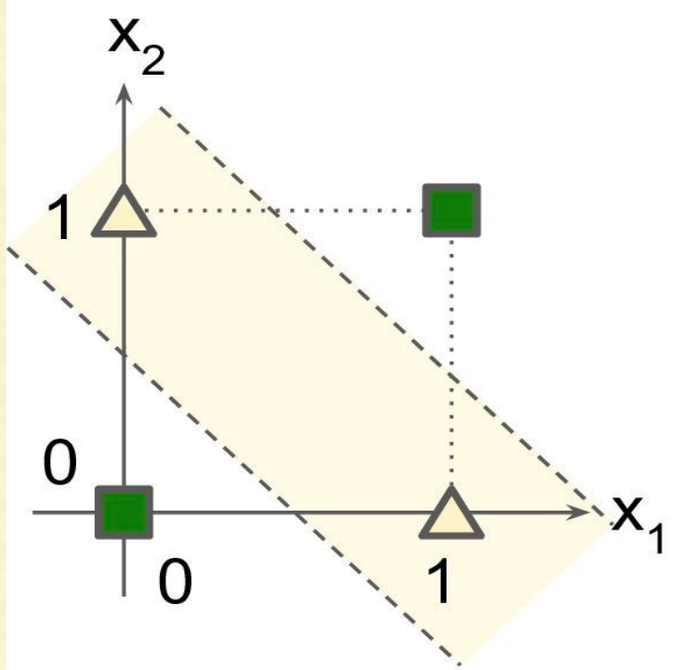
```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Perceptron
>>> iris = load_iris()
>>> X = iris.data[:, (2, 3)] # petal length, petal width
>>> y = (iris.target == 0).astype(np.int) # Iris Setosa?
>>> per_clf = Perceptron(random_state=42)
>>>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

Run it on Notebook

Drawbacks of Perceptron model

Contrary to Logistic Regression classifiers, Perceptrons do not output a class probability, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

They are incapable of solving some trivial problems like the Exclusive OR (XOR) classification problem.

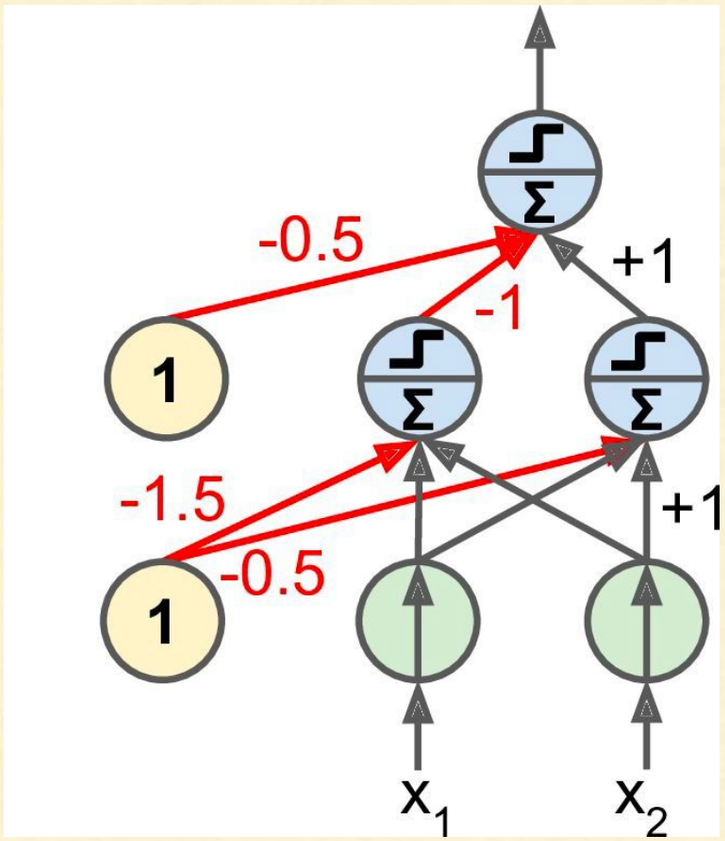


Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

Multi-Layer Perceptron (MLP)

Solving the XOR problem

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**
- In particular, an MLP can solve the XOR problem

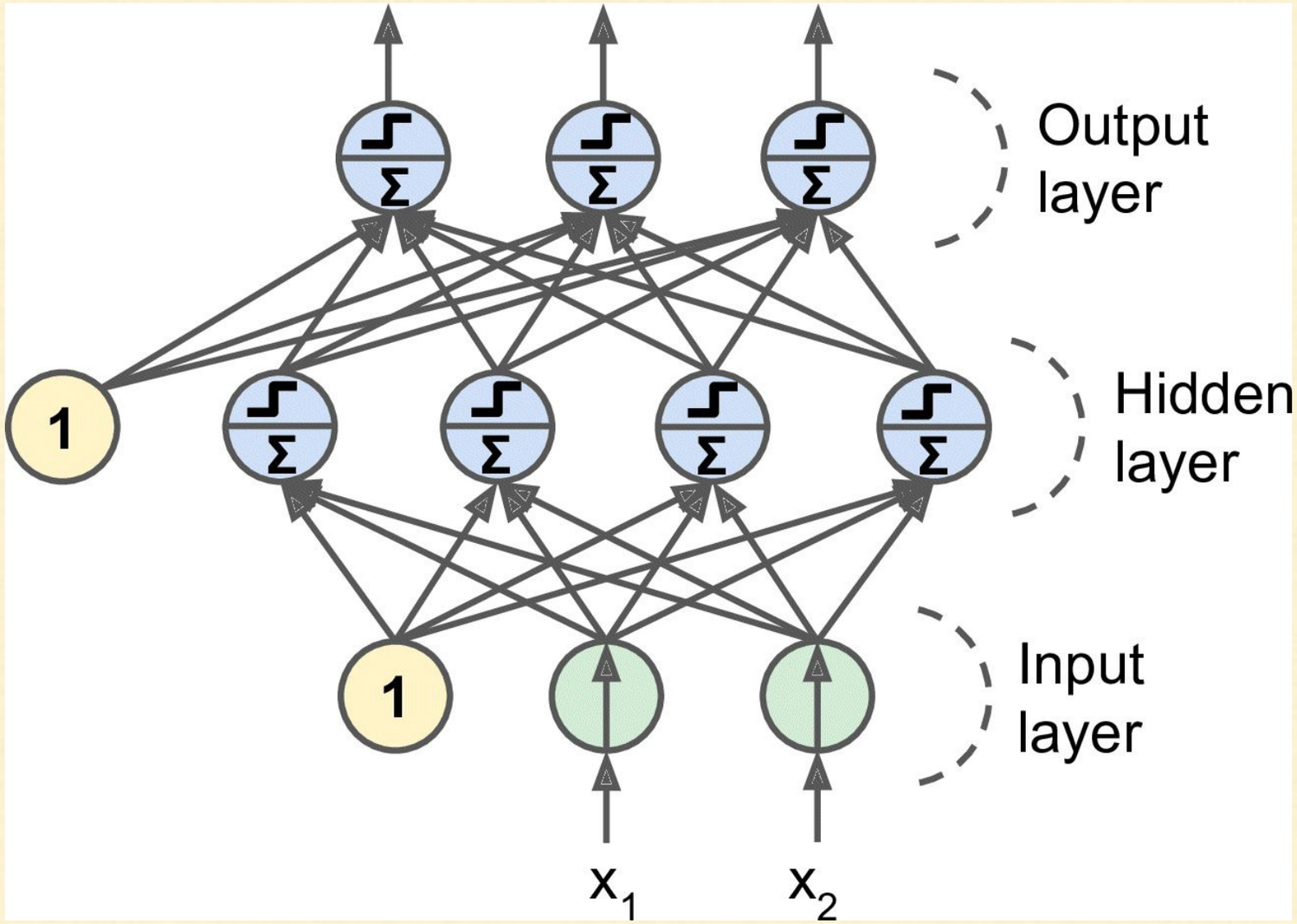


Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

Multi-Layer Perceptron (MLP)

An MLP is composed of

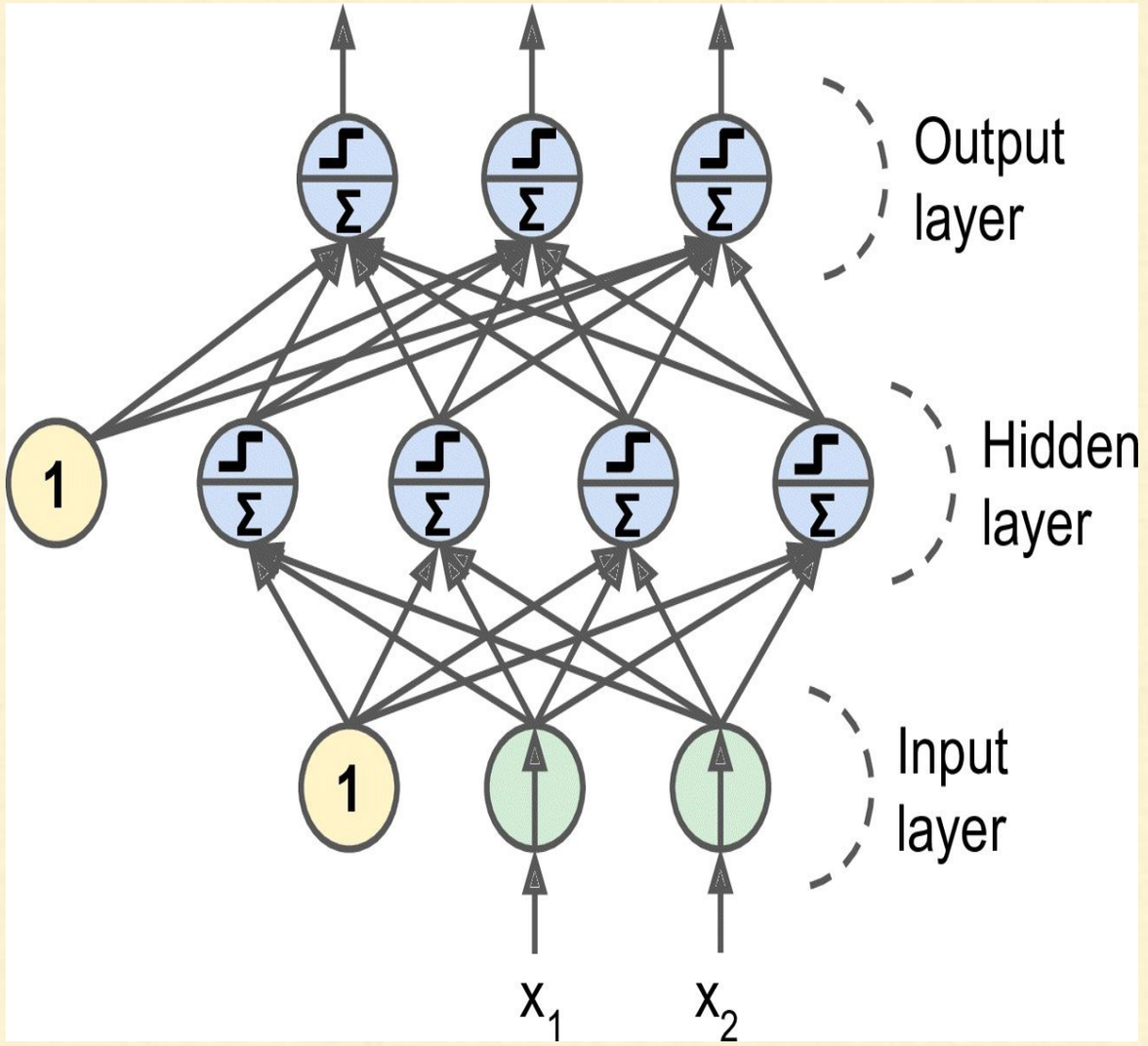
- One **input layer**
- One or more layers of LTUs, called **hidden layers**
- And one final layer of LTUs called the **output layer**



Multi-Layer Perceptron (MLP)

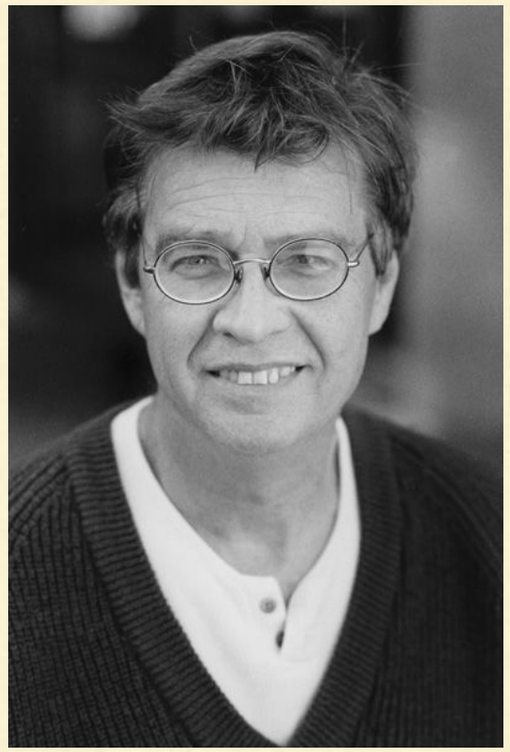
Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

Deep Neural Network
When an ANN has two or more hidden layers, it is called a deep neural network (DNN)



Multi-Layer Perceptron and Backpropagation

- For many years researchers struggled to find a way to train MLPs, without success
- 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

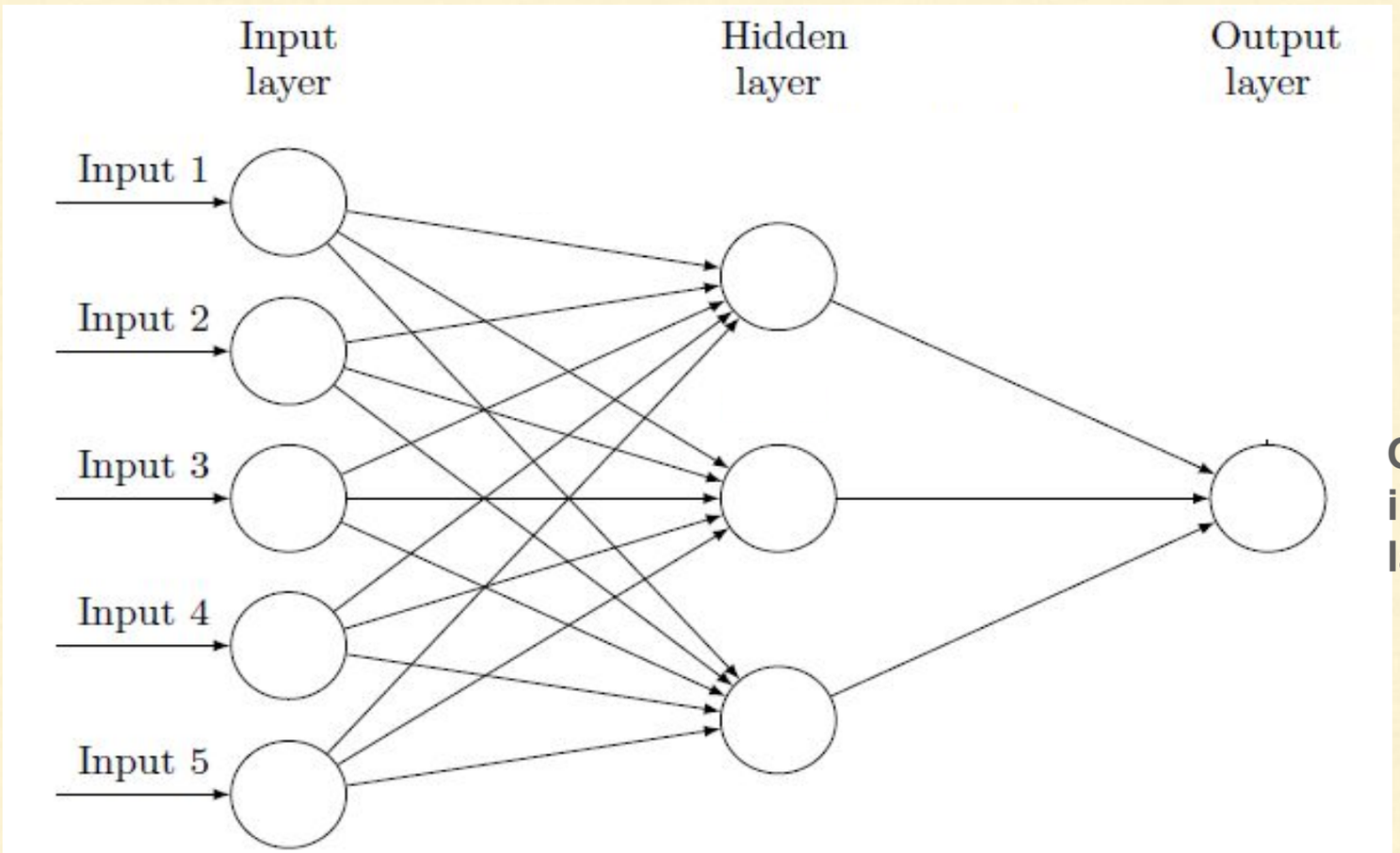


Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

For each training instance

- The algorithm feeds it to the network
- And computes the output of every neuron in each consecutive layer
- This is the forward pass, just like when making predictions

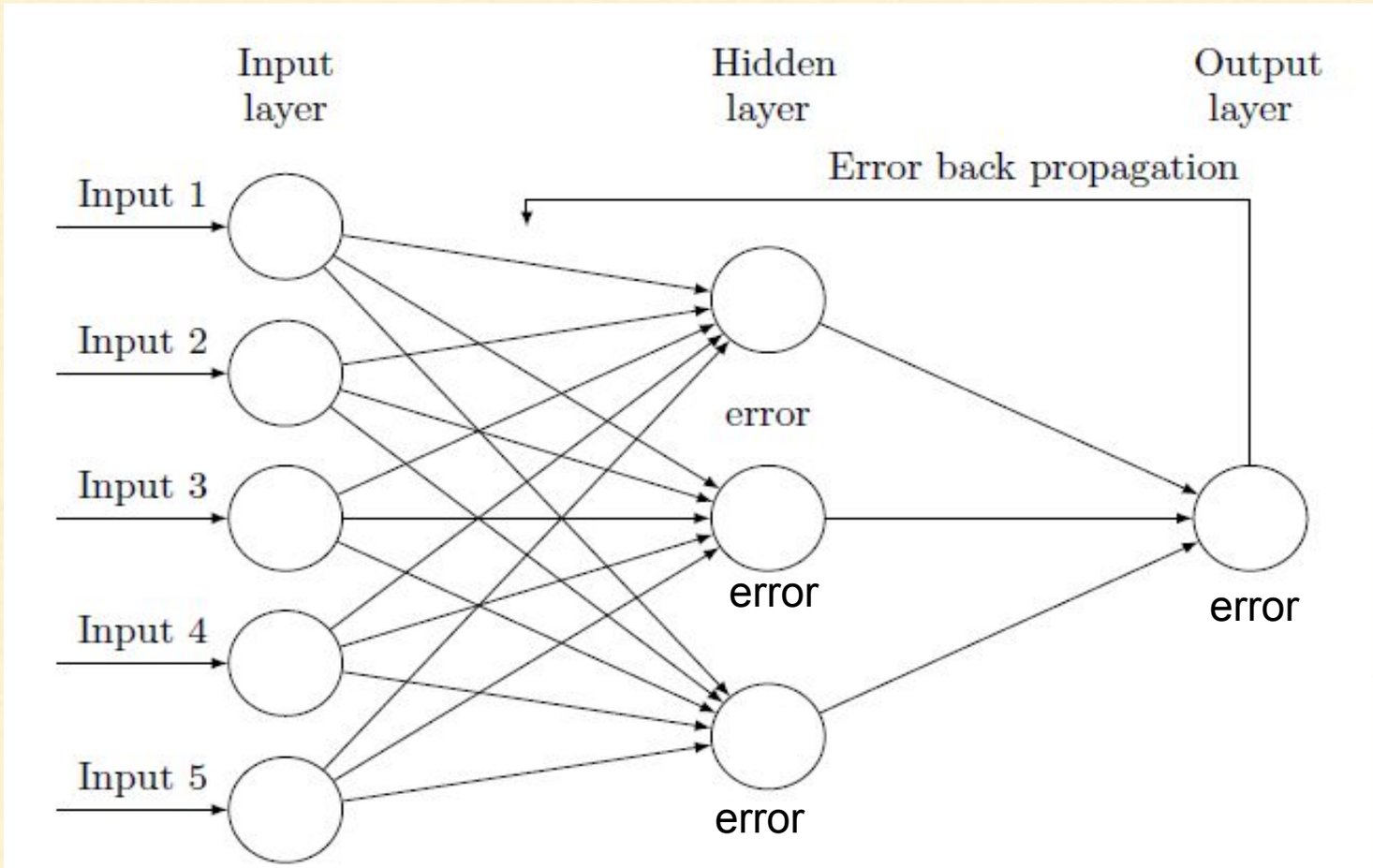


Output of every neuron in each consecutive layer is found

Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

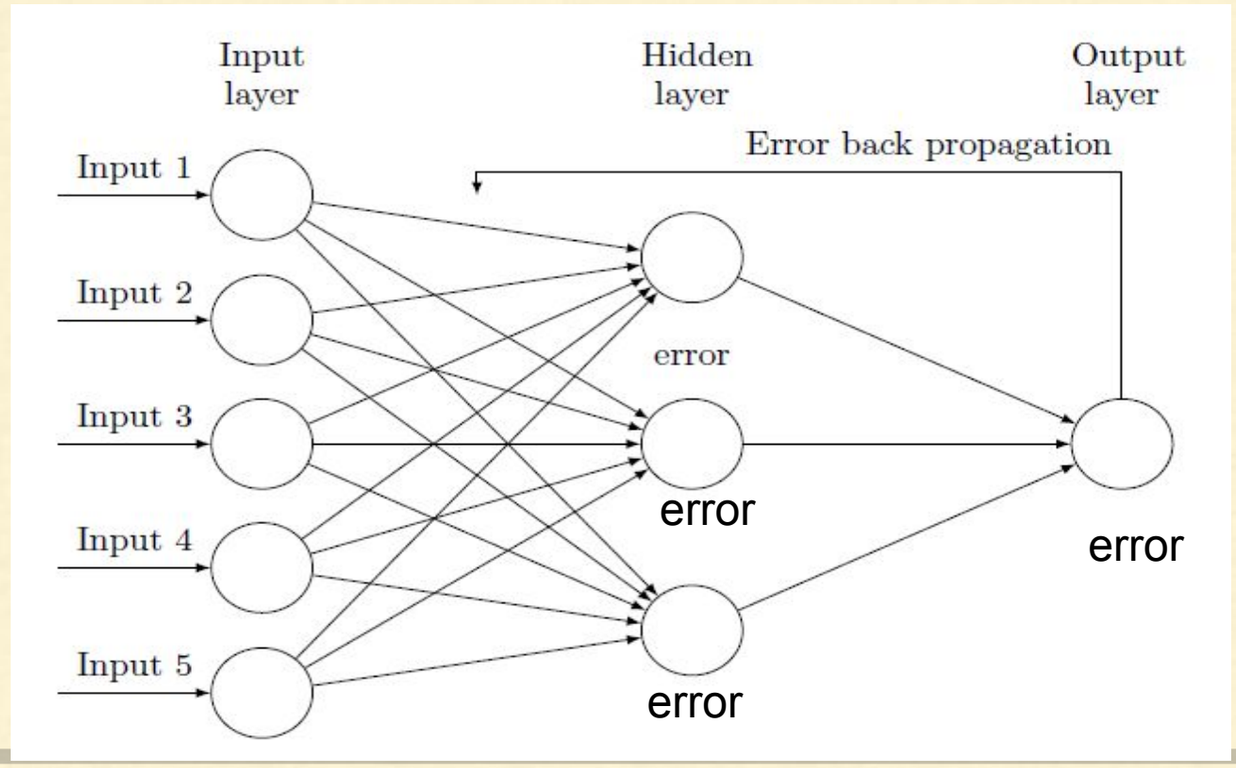
- Then it measures the network's output error i.e., the difference between the desired output and the actual output of the network
- It then computes how much each neuron in the last hidden layer contributed to each output neuron's error.



Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

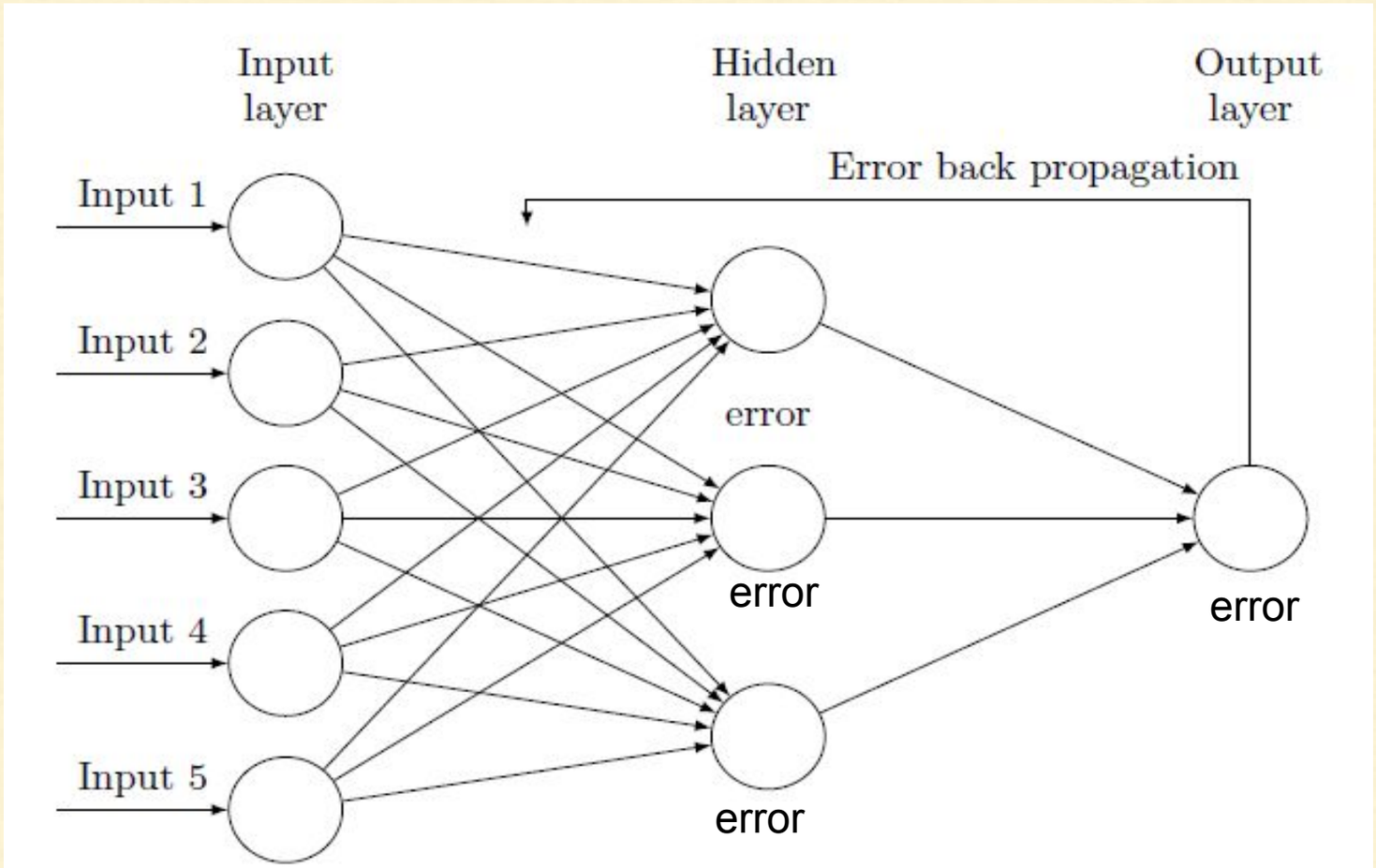
- It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer — and so on until the algorithm reaches the input layer.
- The last step of the backpropagation algorithm is a **Gradient Descent step** on all the connection weights in the network, using the error gradients measured earlier.



Multi-Layer Perceptron and Backpropagation

What is Backpropagation?

- This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network. Hence the name of the algorithm is **Backpropagation**



Multi-Layer Perceptron and Backpropagation

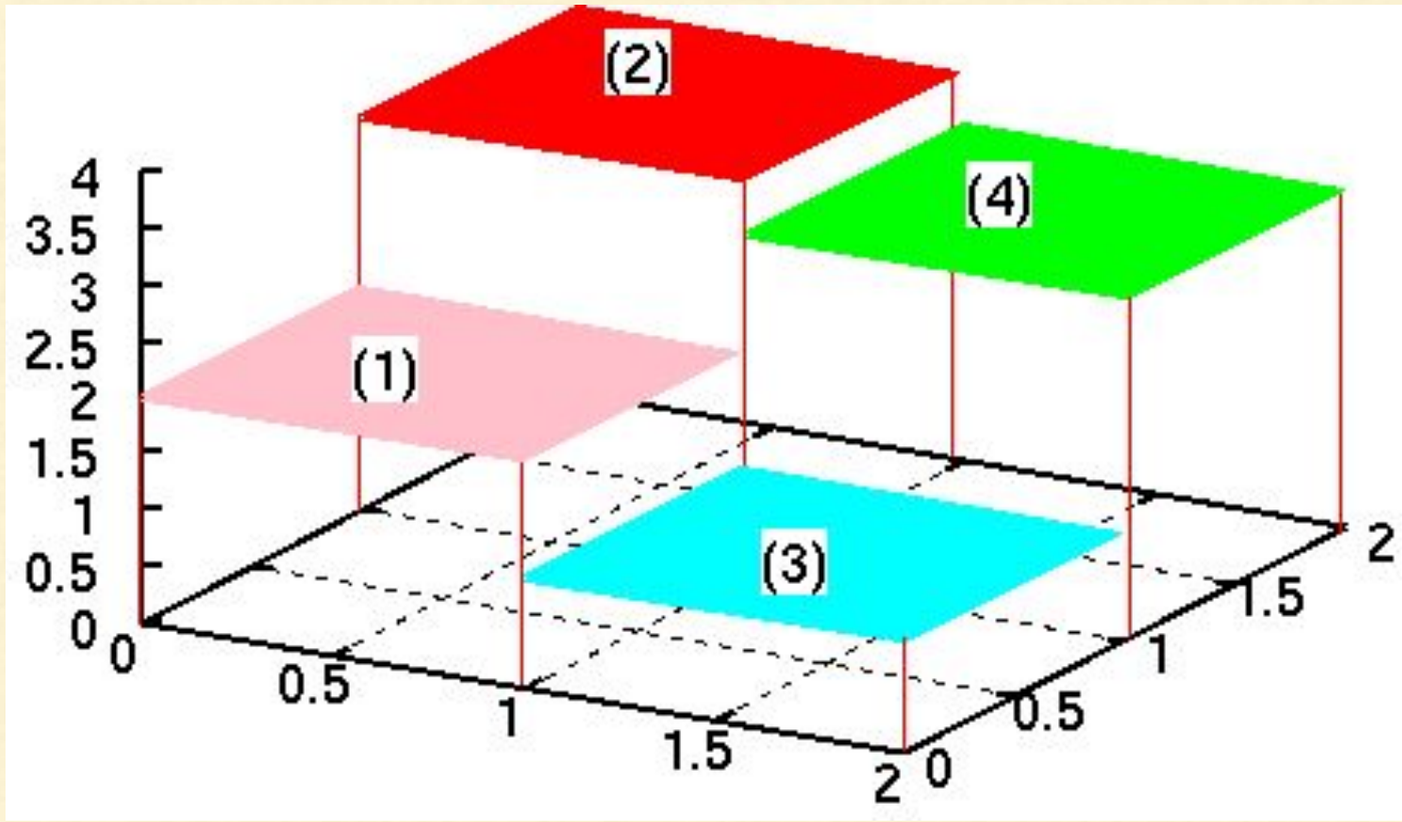
Let's summarize what we learnt about Backpropagation

1. **Forward pass -** for each training instance the backpropagation algorithm first makes a prediction
2. Measures the error
3. **Reverse pass -** then goes through each layer in reverse to measure the error contribution from each connection
4. **Gradient Descent step -** finally slightly tweaks the connection weights to reduce the error

Multi-Layer Perceptron and Backpropagation

How can we apply Gradient Descent to a output from step function ??

- Step function contains only **flat segments**, so there is no gradient to work with
- Gradient Descent cannot move on a flat surface



Multi-Layer Perceptron and Backpropagation

How can we apply Gradient Descent to a output from step function ??

The solution

- In order for Gradient Descent to work properly, the authors made a key change to the MLP's architecture
- They replaced the step function with the logistic function,

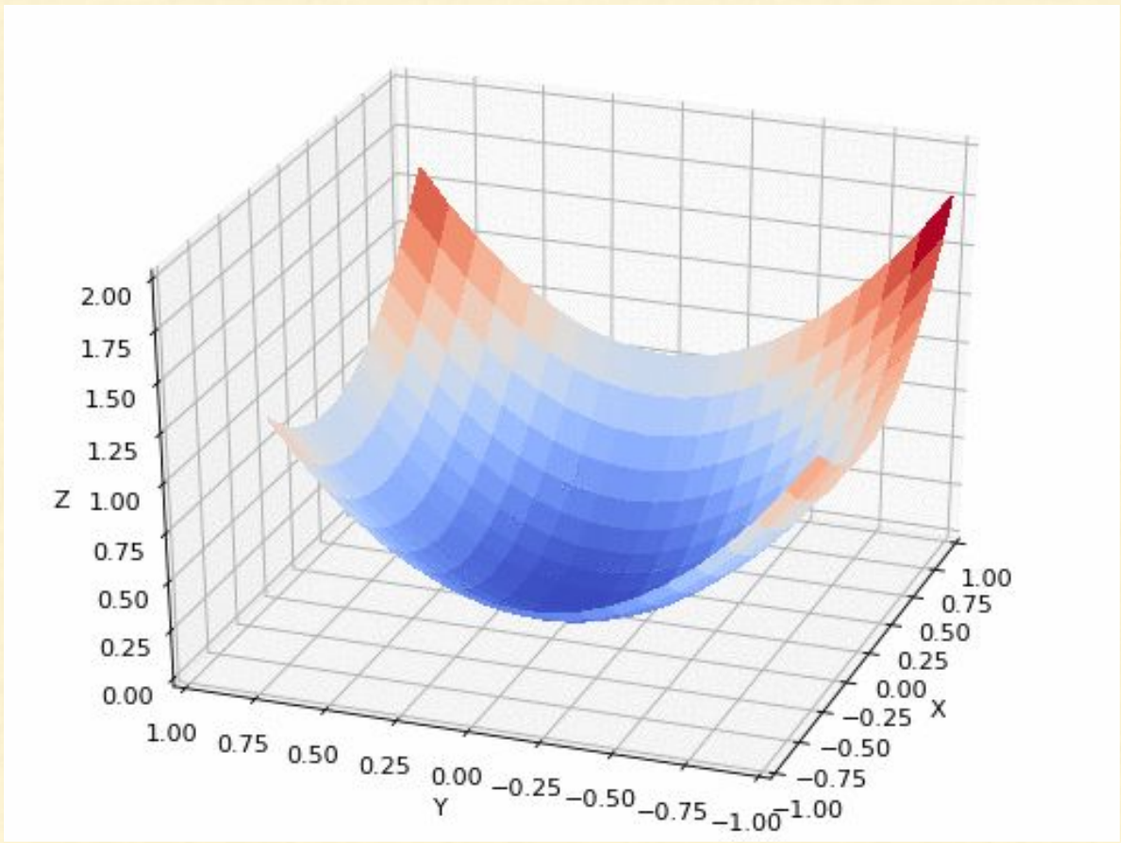
$$\sigma(z) = 1 / (1 + \exp(-z))$$

Multi-Layer Perceptron and Backpropagation

How can we apply Gradient Descent to a output from step function ??

The solution

- The logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step.



Multi-Layer Perceptron and Backpropagation

Other Activation Functions

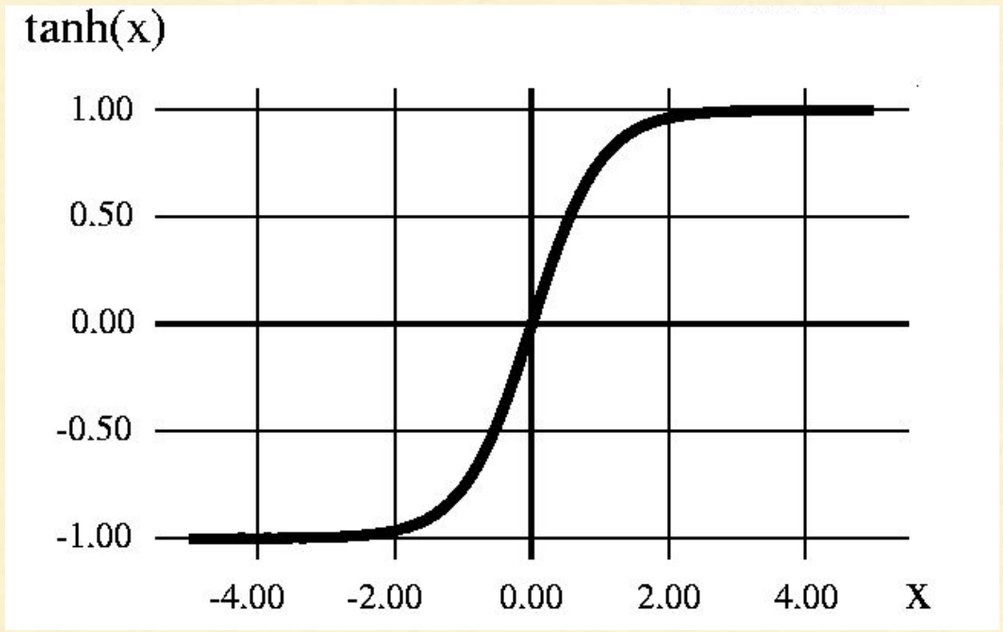
The backpropagation algorithm may be used with other activation functions, instead of the logistic function.

Two other popular activation functions are

- *The hyperbolic tangent function $\tanh(z) = 2\sigma(2z) - 1$*
- *The ReLU function*

Multi-Layer Perceptron and Backpropagation

Other Activation Functions

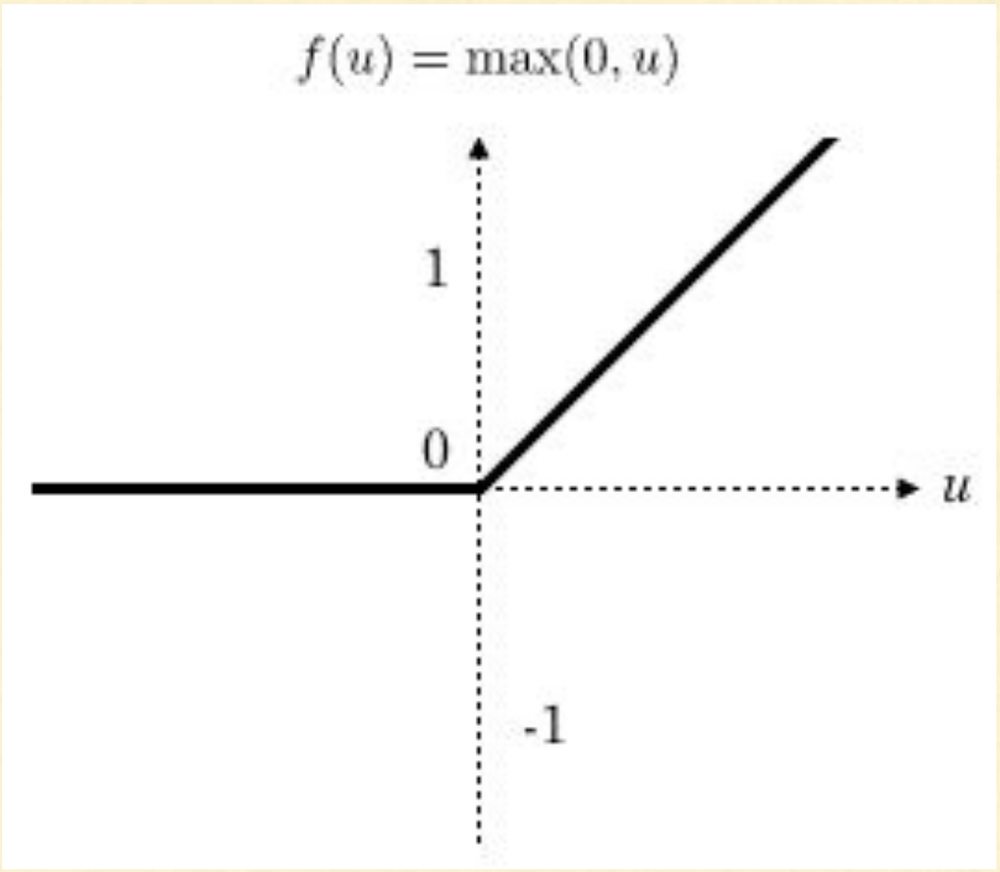


The hyperbolic tangent function tanh
(z) = 2σ(2z) – 1

- Just like the logistic function it is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function),
- It tends to make each layer's output more or less normalized (i.e., centered around 0) at the beginning of training.
- This often helps speed up convergence.

Multi-Layer Perceptron and Backpropagation

Other Activation Functions

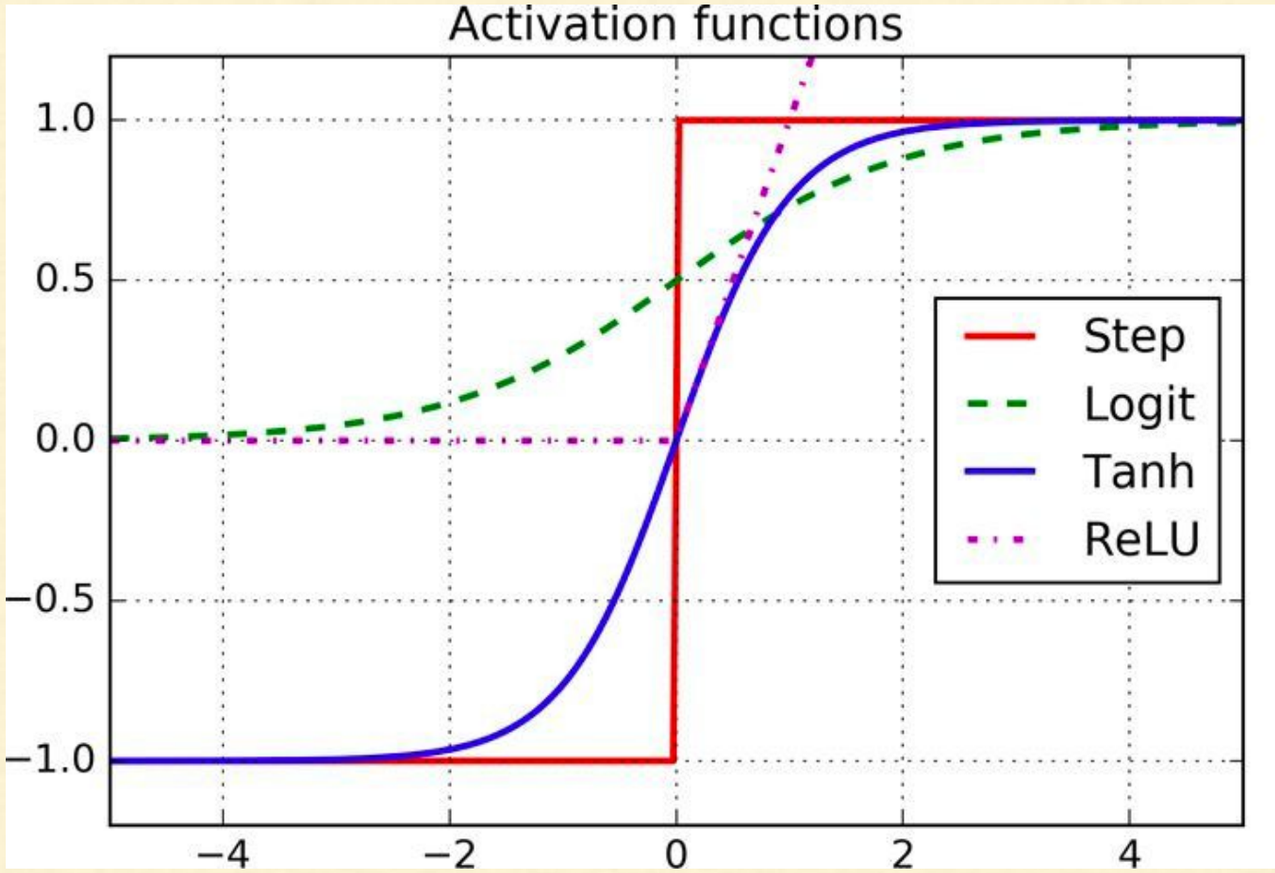


The ReLU function
ReLU (z) = max (0, z)

- It is continuous but unfortunately not differentiable at z = 0
- The slope changes abruptly, which can make Gradient Descent bounce around.
- In practice it works very well and has the advantage of being fast to compute.

Multi-Layer Perceptron and Backpropagation

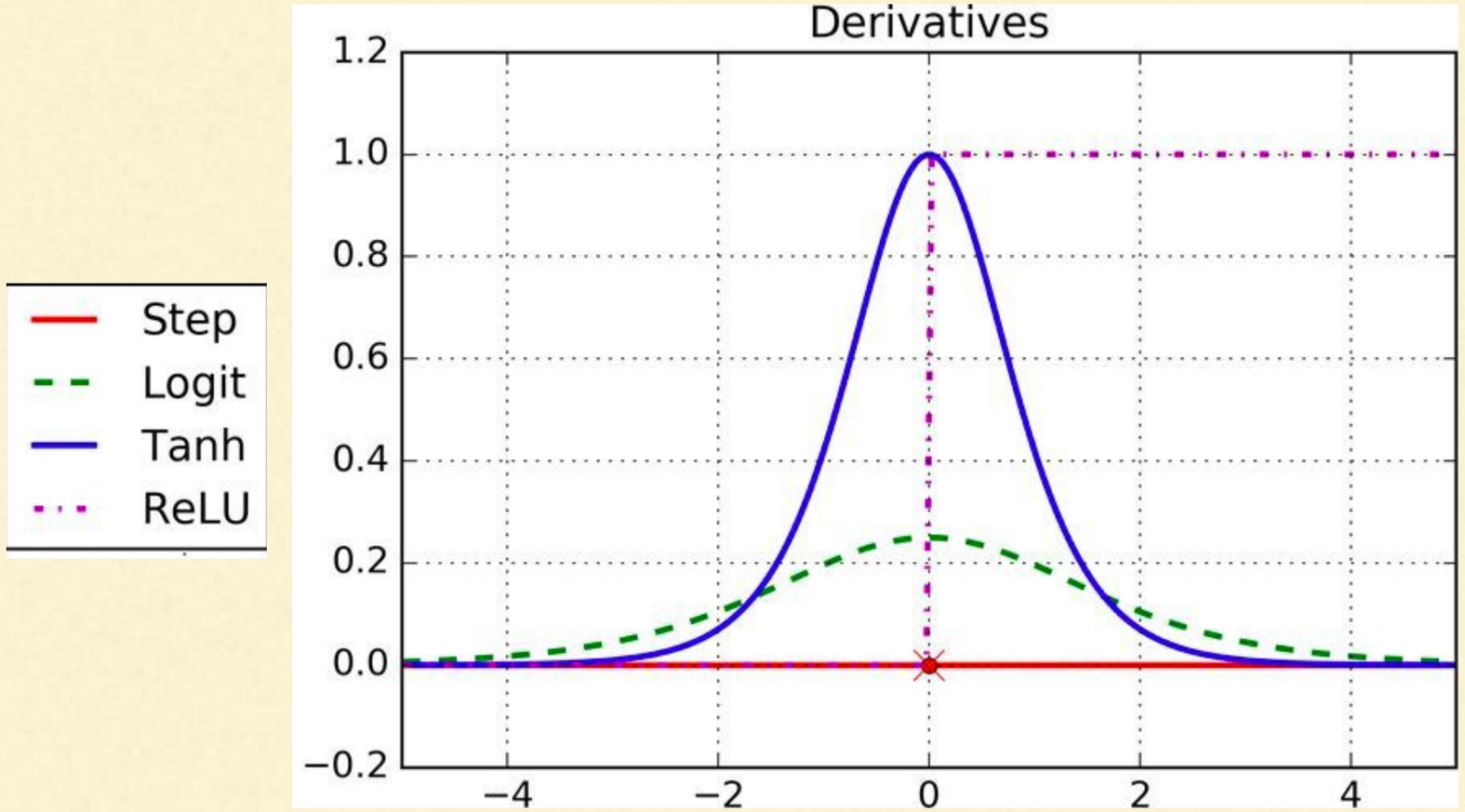
Comparison of Different Activation Functions



The above plot shows the variation of different activation functions.

Multi-Layer Perceptron and Backpropagation

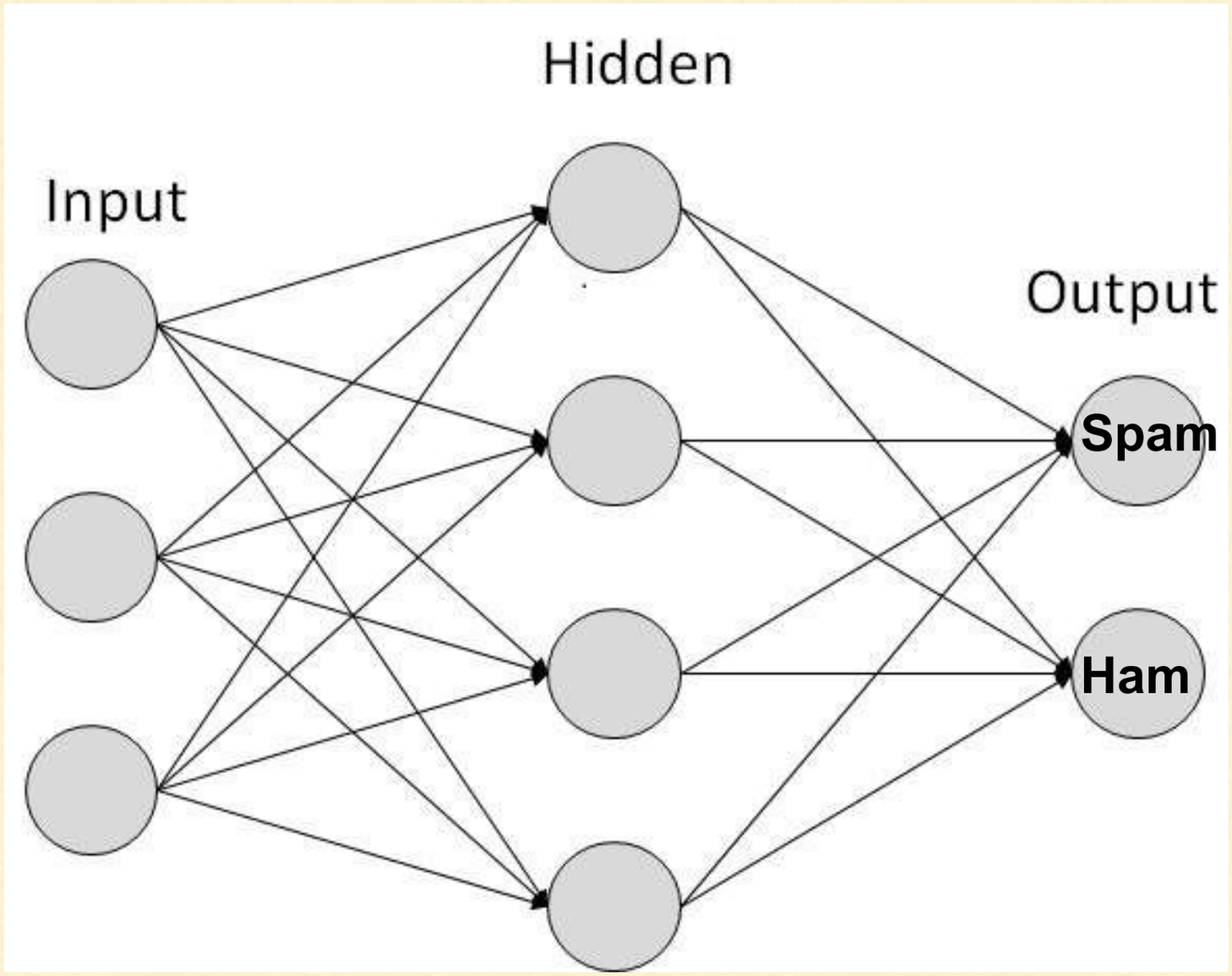
Comparison of Different Activation Functions



The above plot shows the derivatives of different activation functions.

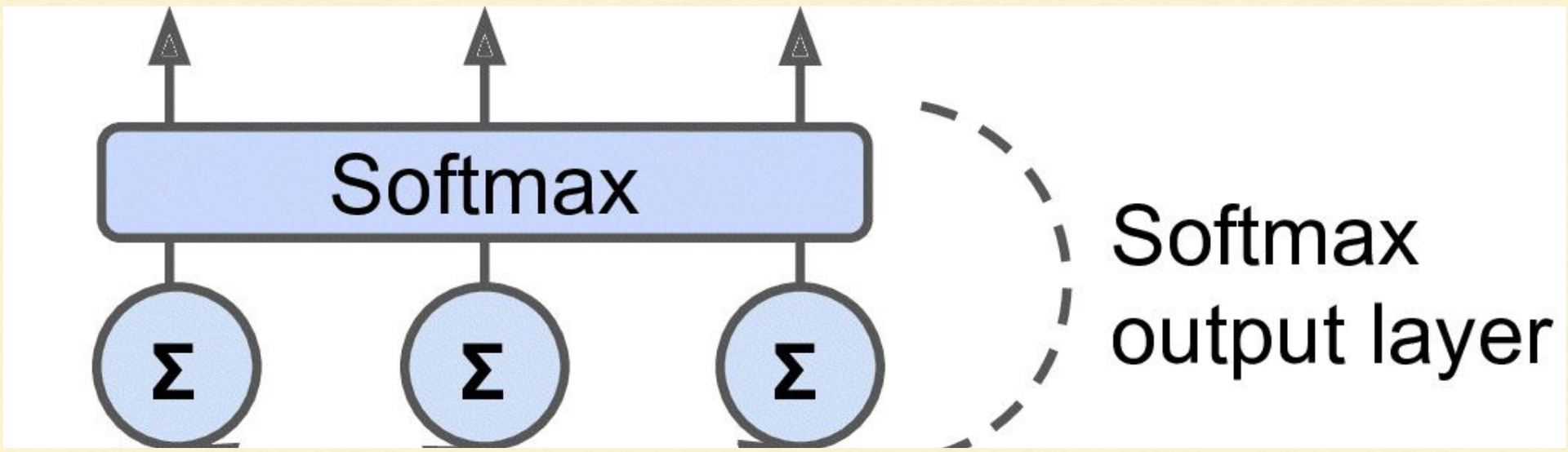
Classification with Multi-Layer Perceptron

- An MLP is often used for classification, with each output corresponding to a different binary class
- Eg. spam/ham, urgent/not-urgent



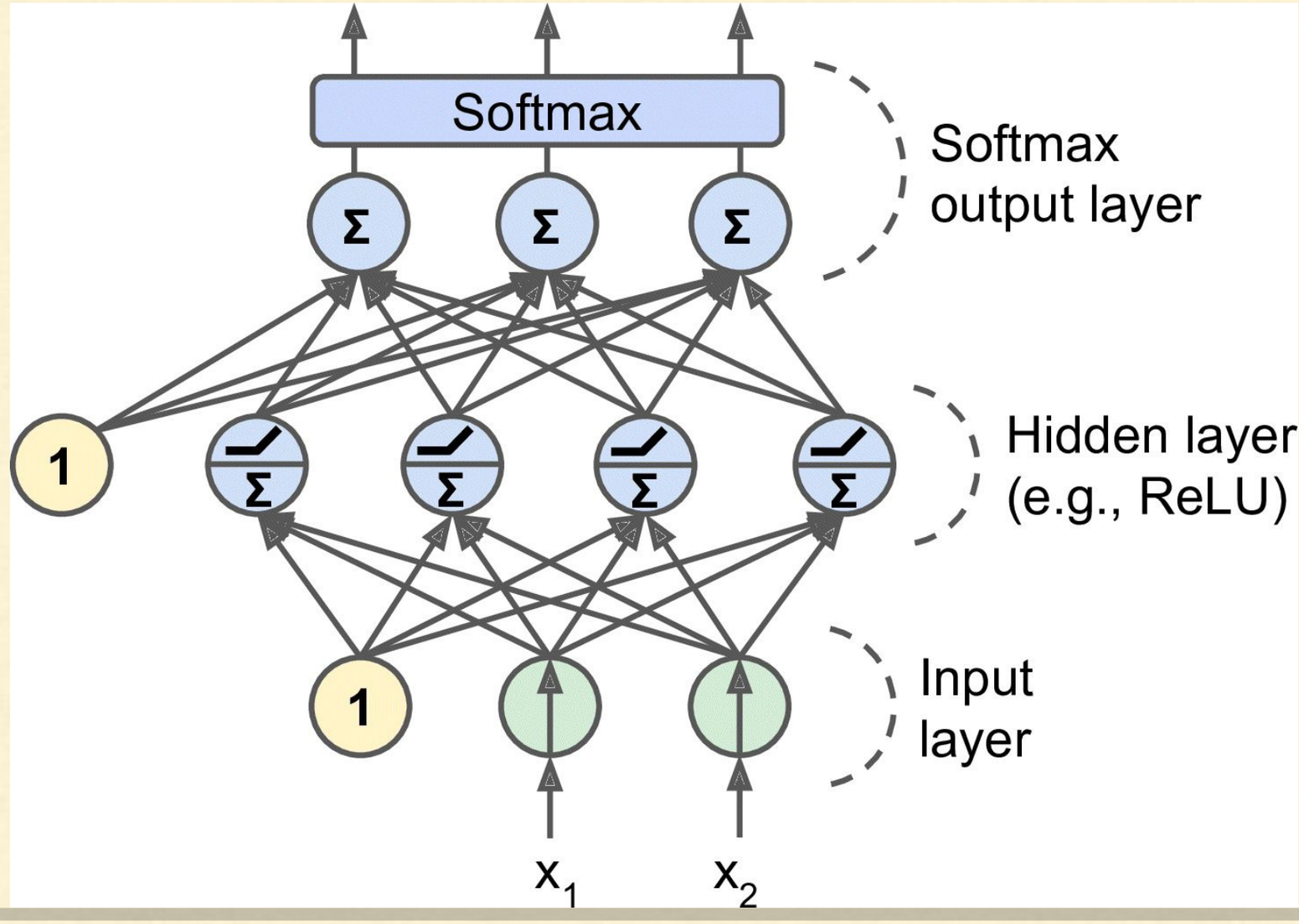
Classification with Multi-Layer Perceptron

- When the classes are exclusive, the output layer is typically modified by replacing the individual activation functions by a shared softmax function
- E.g., classes 0 through 9 for digit image classification



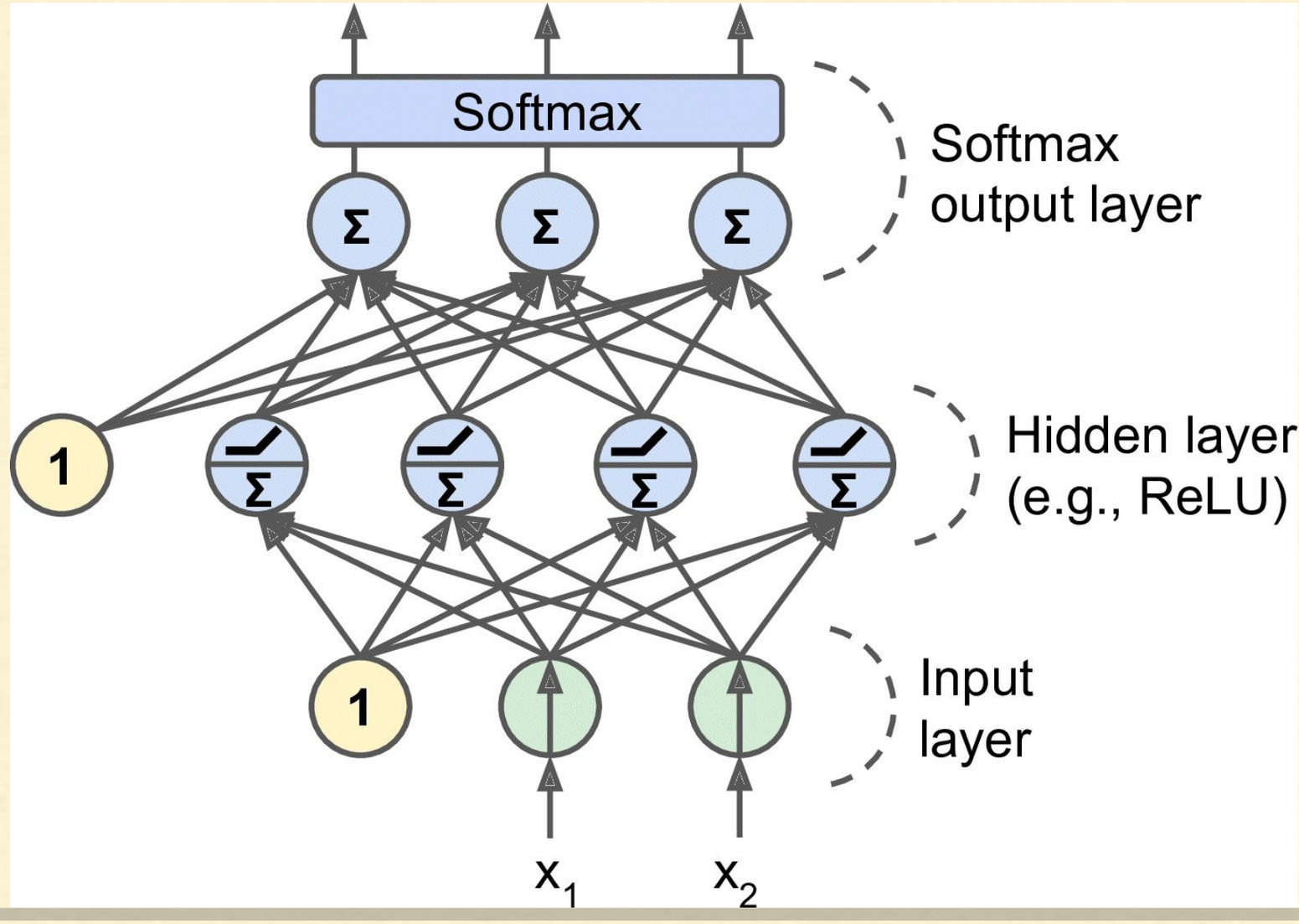
Classification with Multi-Layer Perceptron

- The output of each neuron corresponds to the estimated probability of the corresponding class
- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a feedforward neural network (FNN)



Classification with Multi-Layer Perceptron

- The output of each neuron corresponds to the estimated probability of the corresponding class
- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a feedforward neural network (FNN)



Hands-On

Training an MLP with TensorFlow's High-Level API

Training an MLP with TensorFlow's API

- The simplest way to train an MLP with TensorFlow is to use the high-level API **TF.Learn**, which is quite similar to Scikit-Learn's API
- The DNNClassifier class is used to train a deep neural network with any number of hidden layers, and a softmax output layer to output estimated class probabilities.

Training an MLP with TensorFlow's API

The following code trains a DNN for classification with two hidden layers (one with 300 neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

```
>>> import tensorflow as tf
>>> feature_columns =
tf.contrib.learn.infer_real_valued_columns_from_input(X_
rain)
>>> dnn_clf =
tf.contrib.learn.DNNClassifier(hidden_units=[300, 100],
n_classes=10, feature_columns=feature_columns)
>>> dnn_clf.fit(x=X_train, y=y_train, batch_size=50,
steps=40000)
```

Run it on Notebook

Training an MLP with TensorFlow's API

Let us run this code on the MNIST dataset after scaling it
Let's import the MNIST dataset

```
>>> from tensorflow.examples.tutorials.mnist import
input_data
>>> mnist = input_data.read_data_sets("/tmp/data/")
>>> X_train = mnist.train.images
>>> X_test = mnist.test.images
>>> y_train = mnist.train.labels.astype("int")
>>> y_test = mnist.test.labels.astype("int")
```

Run it on Notebook

Training an MLP with TensorFlow's API

Let us run this code on the MNIST dataset after scaling it
Let's fit our Model to the MNIST dataset

```
>>> import tensorflow as tf
>>> config = tf.contrib.learn.RunConfig(tf_random_seed=42)
>>> 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, config=config)
>>> dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # if TensorFlow >= 1.1
>>> dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

Run it on Notebook

Training an MLP with TensorFlow's API

Let us run this code on the MNIST dataset after scaling it
Let's test the accuracy of our model

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = dnn_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred['classes'])
0.9829
```

We get a model that achieves over 98.29% accuracy on the test set!!
That's better than the best model we trained so far !!

Run it on Notebook

Training an MLP with TensorFlow's API

- Under the hood, the DNNClassifier class creates all the neuron layers, based on the ReLU activation function
- We can change this by setting the activation_fn hyperparameter.
- The output layer relies on the softmax function, and the cost function is cross entropy

Please take care of the following point when using TF.Learn

- The TF.Learn API **is still quite new**, so some of the names and functions used in these examples may evolve over time

Training a DNN Using Plain TensorFlow

Training a DNN Using Plain TensorFlow

TensorFlow's lower level Python API provides more control over the architecture of the network

In this section we will

- Build the same model as before using TensorFlow's lower level Python API
- We will implement Mini-batch Gradient Descent to train it on the MNIST dataset.

Training a DNN Using Plain TensorFlow

The following steps will be involved in the building of our model.

- The first step is the construction phase, building the TensorFlow graph.
- The second step is the execution phase, where you actually run the graph to train the model

Training a DNN Using Plain TensorFlow

Construction Phase

- First we need to import the tensorflow library.
- Then we must specify the number of inputs and outputs,
- And then set the number of hidden neurons in each layer

```
>>> import tensorflow as tf
>>> n_inputs = 28*28 # MNIST
>>> n_hidden1 = 300
>>> n_hidden2 = 100
>>> n_outputs = 10
```

Run it on Notebook

Training a DNN Using Plain TensorFlow

Construction Phase

- Next, we can use placeholder nodes to represent the training data and targets.
- The shape of X is only partially defined. It will be a 2D tensor (i.e., a matrix), with instances along the first dimension and features along the second dimension

```
>>> X = tf.placeholder(tf.float32, shape=(None,
n_inputs), name="X")
>>> y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Run it on Notebook

Training a DNN Using Plain TensorFlow

Construction Phase

>>> X = tf.placeholder(tf.float32, shape=**(None, n_inputs)**, name="X")

- The number of features is going to be 28 x 28 (one feature per pixel),
- But we don't know yet how many instances each training batch will contain.
- So the shape of X is **(None, n_inputs)**

Training a DNN Using Plain TensorFlow

Construction Phase

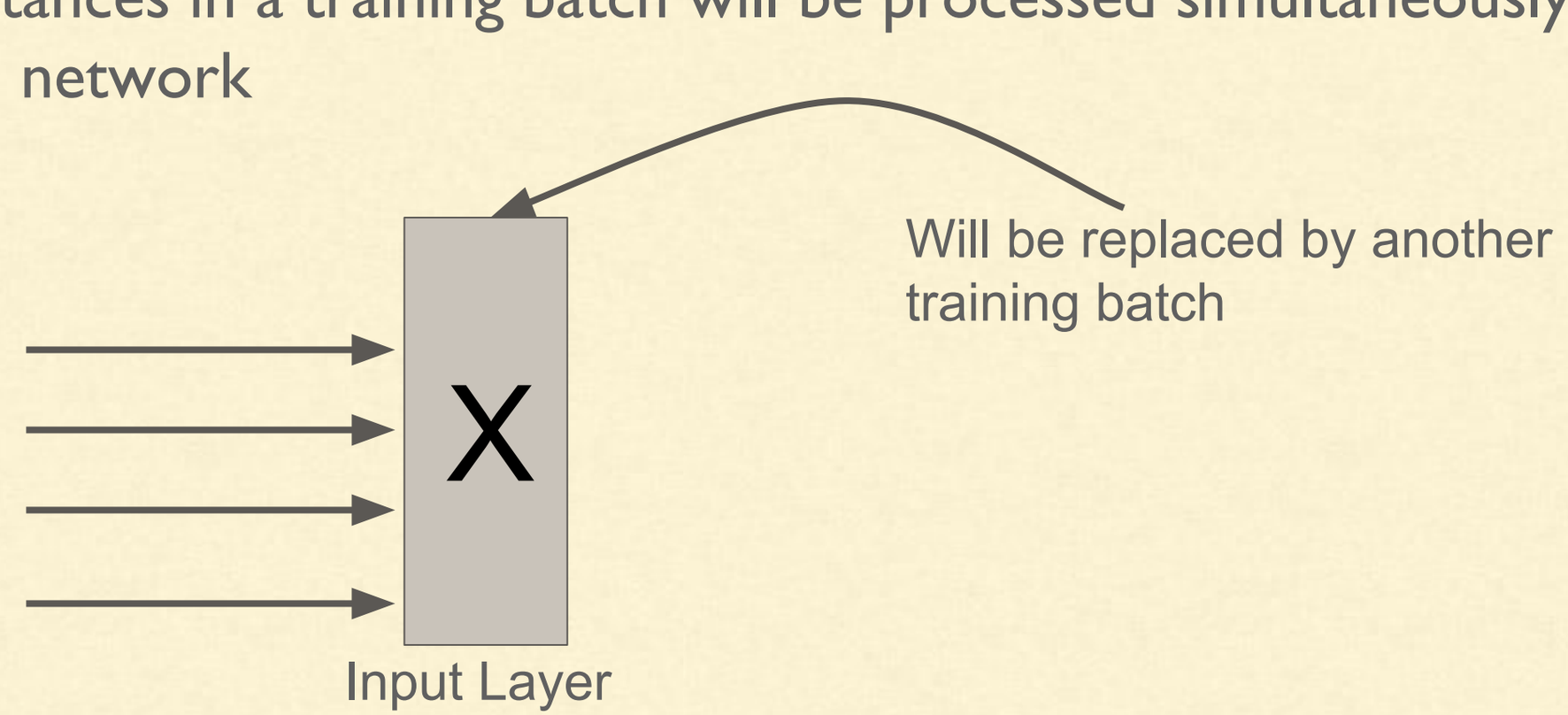
```
>>> y = tf.placeholder(tf.int64, shape=(None), name="y")
```

- Similarly, we know that y will be a 1D tensor with one entry per instance
- But again we don't know the size of the training batch at this point,
- So the shape is **(None)**.

Training a DNN Using Plain TensorFlow

Construction Phase
Now let's create the actual Neural Network

- The placeholder X will act as the input layer; during the execution phase, it will be replaced with one training batch at a time
- All the instances in a training batch will be processed simultaneously by the neural network



Training a DNN Using Plain TensorFlow

Construction Phase
Now let's create the actual Neural Network

- Now you need to create the two hidden layers and the output layer.
- The two hidden layers are **almost identical**: they differ only by the inputs they are connected to and by the number of neurons they contain.
- The output layer **is also very similar**, but it uses a softmax activation function instead of a ReLU activation function.

The best way will be to define a function to create the layers.

Training a DNN Using Plain TensorFlow

Construction Phase
Now let's create the actual Neural Network

- So let's create a **neuron_layer()** function that we will use to create one layer at a time.
- It will need parameters to specify the inputs, the number of neurons, the activation function, and the name of the layer

Training a DNN Using Plain TensorFlow

Construction Phase

Now let's create the actual Neural Network

```
>>> def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons),stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation=="relu":
           return tf.nn.relu(z)
        else:
           return z
```

Run it on Notebook

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com