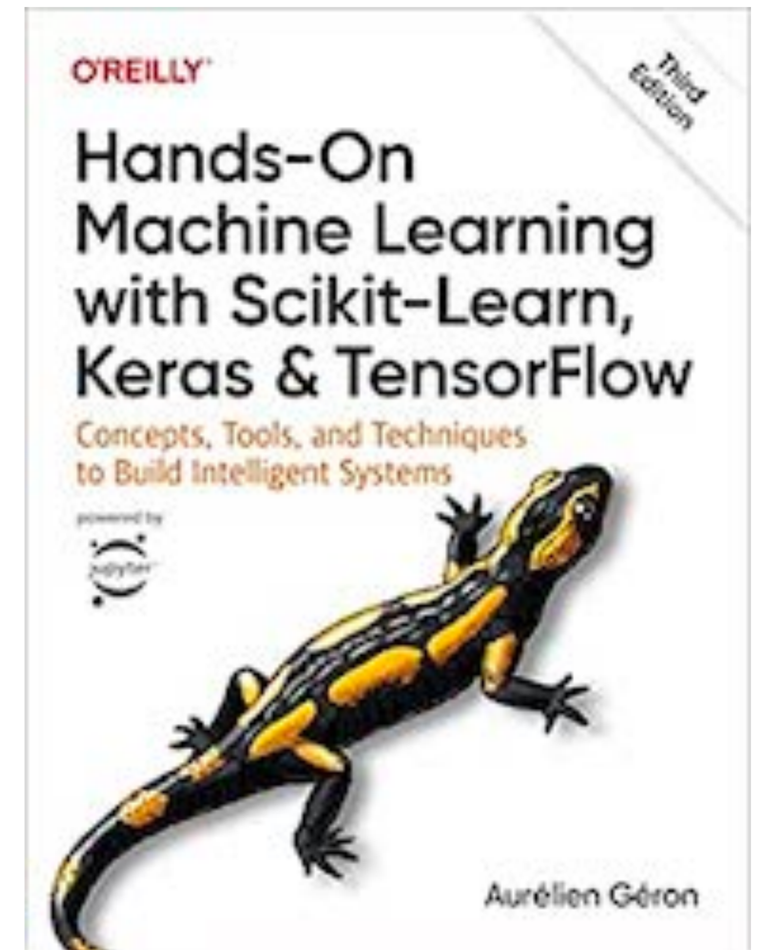
Machine Learning Security

10 Introduction to Artificial Neural Networks with Keras



Made Oct 21, 2023

Artificial Neural Networks (ANNs)

- Inspired by our brains
- ANNs are at the very core of deep learning
- Versatile, powerful, and scalable; used by
  - Google Images
  - Apple's Siri
  - YouTube

Topics

- From Biological to Artificial Neurons
- Biological Neurons
- Logical Computations with Neurons
- The Perceptron
- The Multilayer Perceptron and Backpropagation
- Regression MLPs
- Classification MLPs
- Implementing MLPs with Keras
- Fine-Tuning Neural Network Hyperparameters

From Biological to Artificial Neurons

History of ANNs

- Introduced in 1943 for *propositional logic*
- Long winter for ANNs
- In 1990s, other ML systems were developed, such as support vector machines
- Now there's a new wave of interest in ANNs

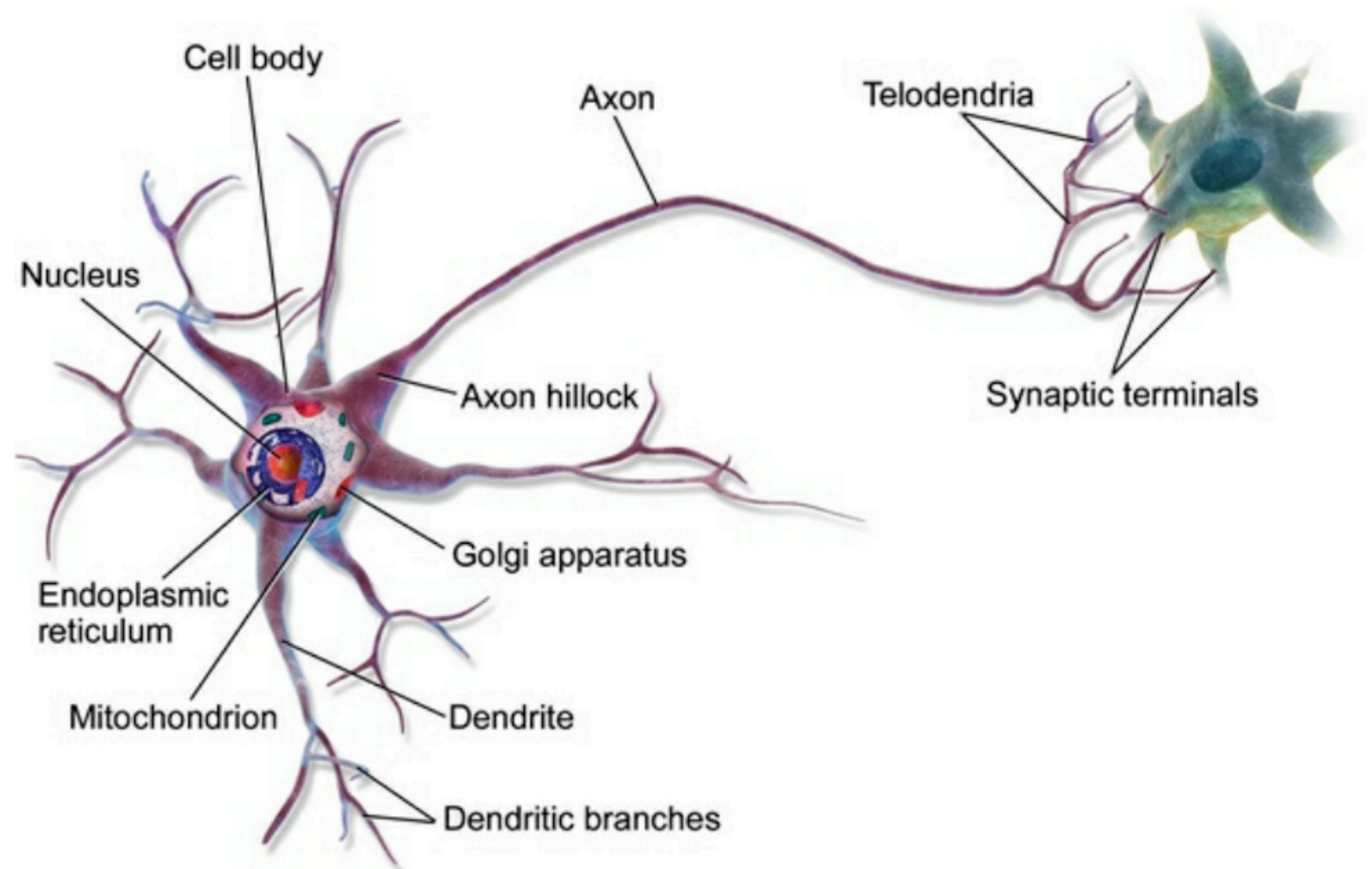
The Case for ANNs

- Huge quantity of data now available
  - ANNs often outperform on very large and complex problems
- Training algorithms have improved
- Theoretical problems like local minima turned out to be benign in practice
  - Local optima often almost as good as the global optimum
- ANNs have entered a virtuous cycle of funding and progress

Biological Neurons

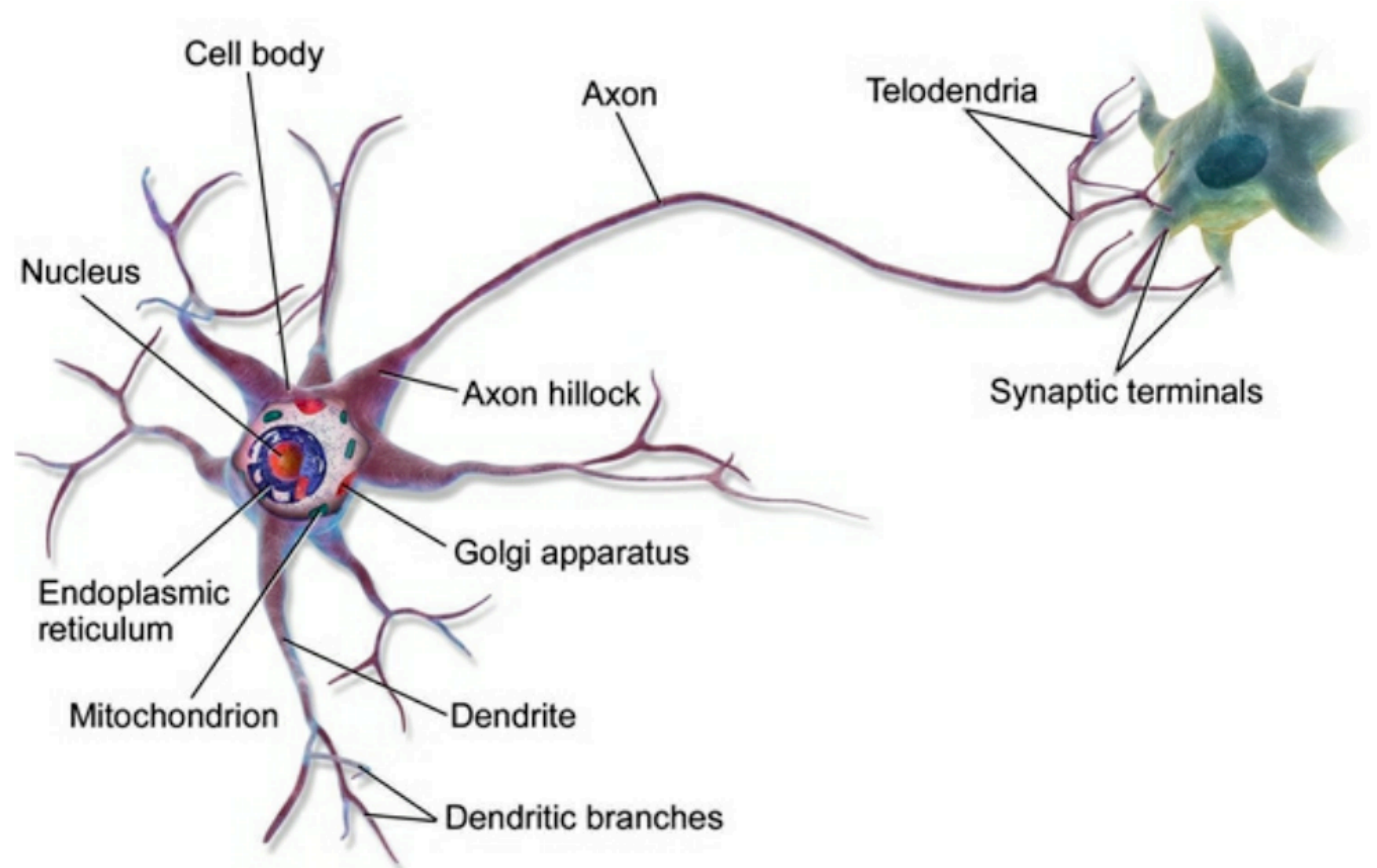
Biological Neurons

- ***dendrites*** are the branching extensions
- ***axon*** carries the output signal far away



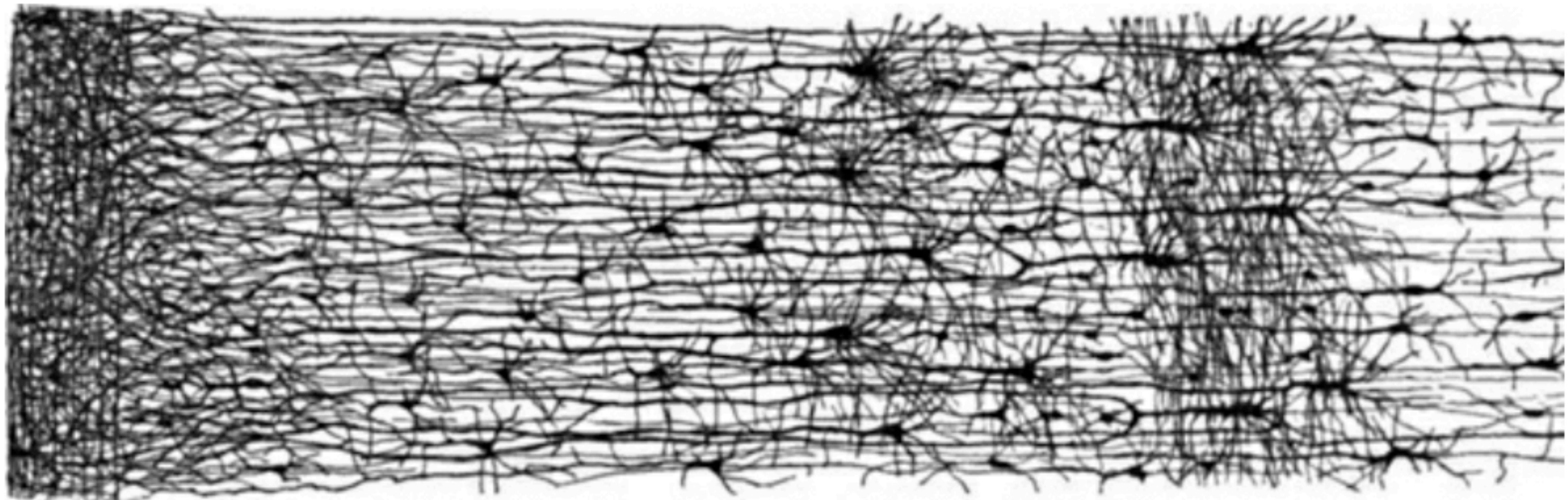
Biological Neurons

- Short electric pulses called ***action potentials*** travel along the axons and release ***neurotransmitters*** at the ***synapses***
- When a neuron receives enough neurotransmitters, it fires



Biological Neural Networks

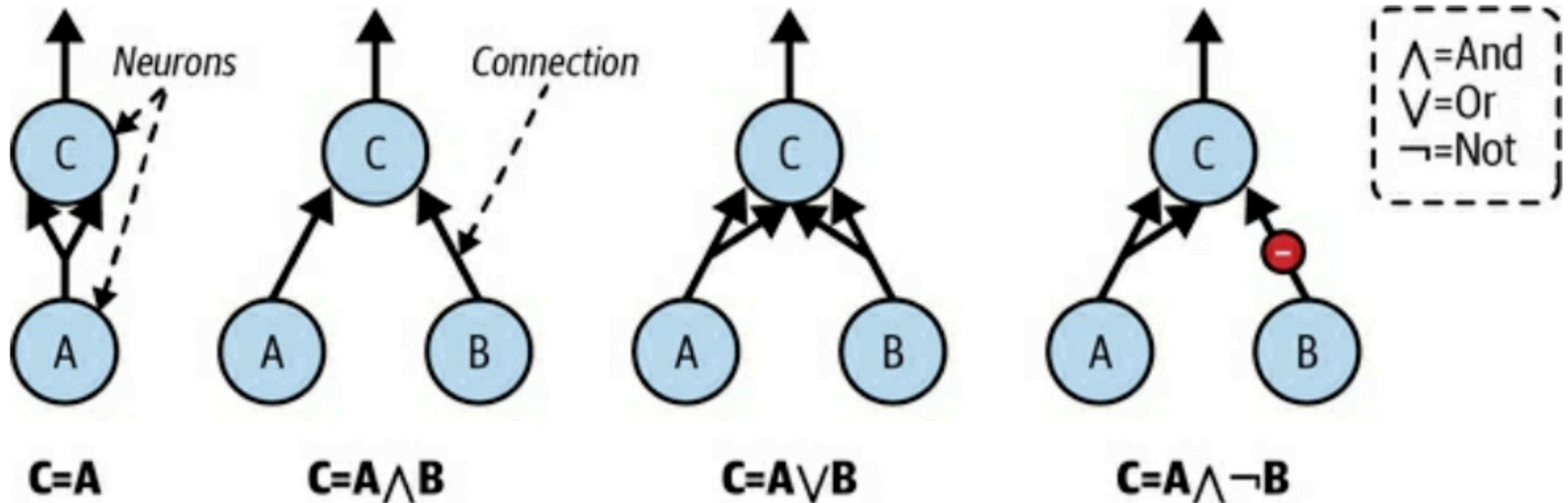
- Each neuron is simple
- Computations are done by the network of many neurons working together
- This is a sample of human cortex



Logical Computations with Neurons

Logical Computations with Neurons

- Simple On/Off neurons act like logic gates



The Perceptron

The Perceptron

- Developed in 1957
- Neurons are
  - ***Threshold Logic Units*** (TLUs) or
  - ***Linear Threshold Units*** (LTUs)
- Computes a linear function of inputs

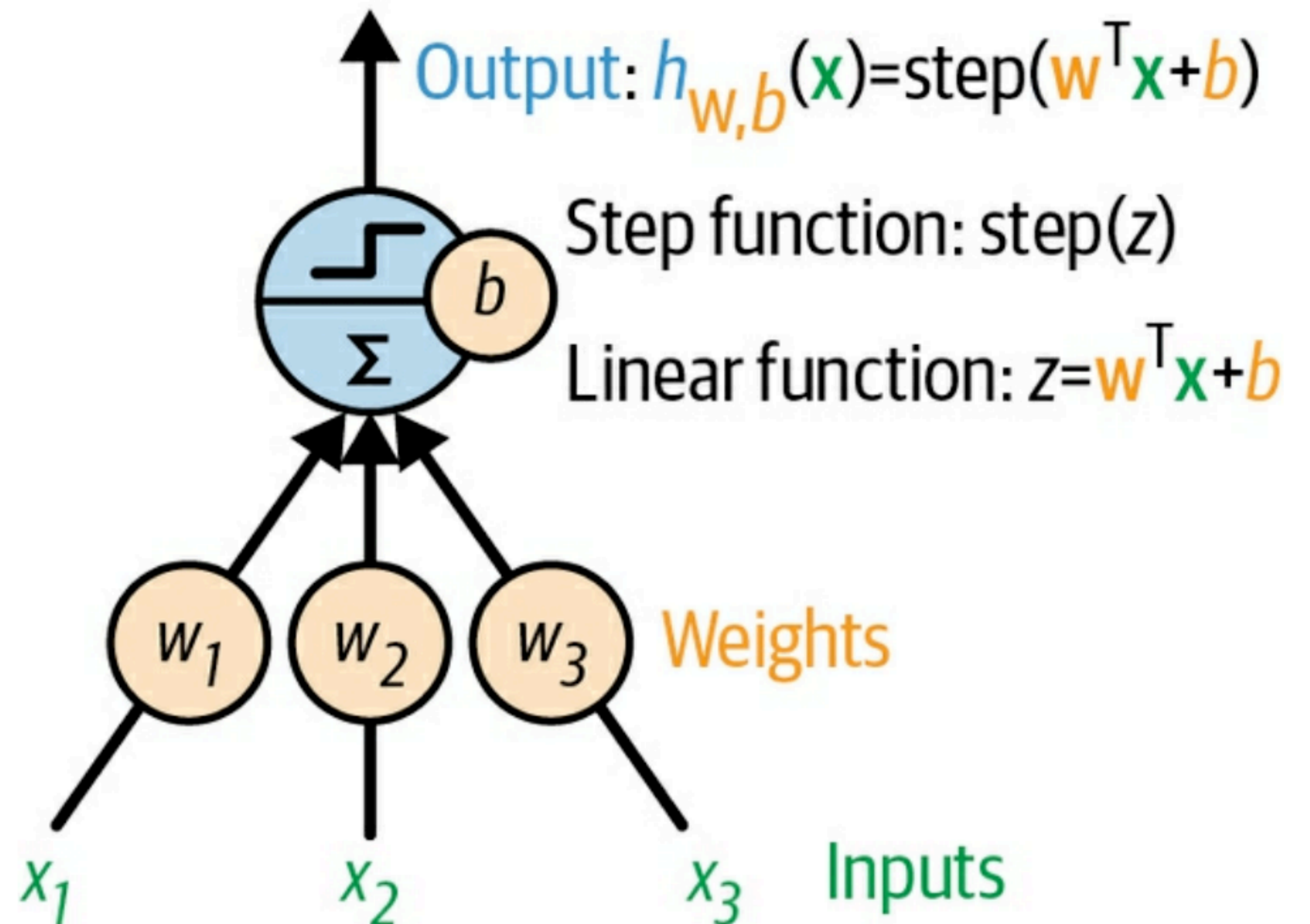
$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$$

- Applies a step function to the result

$$h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$$

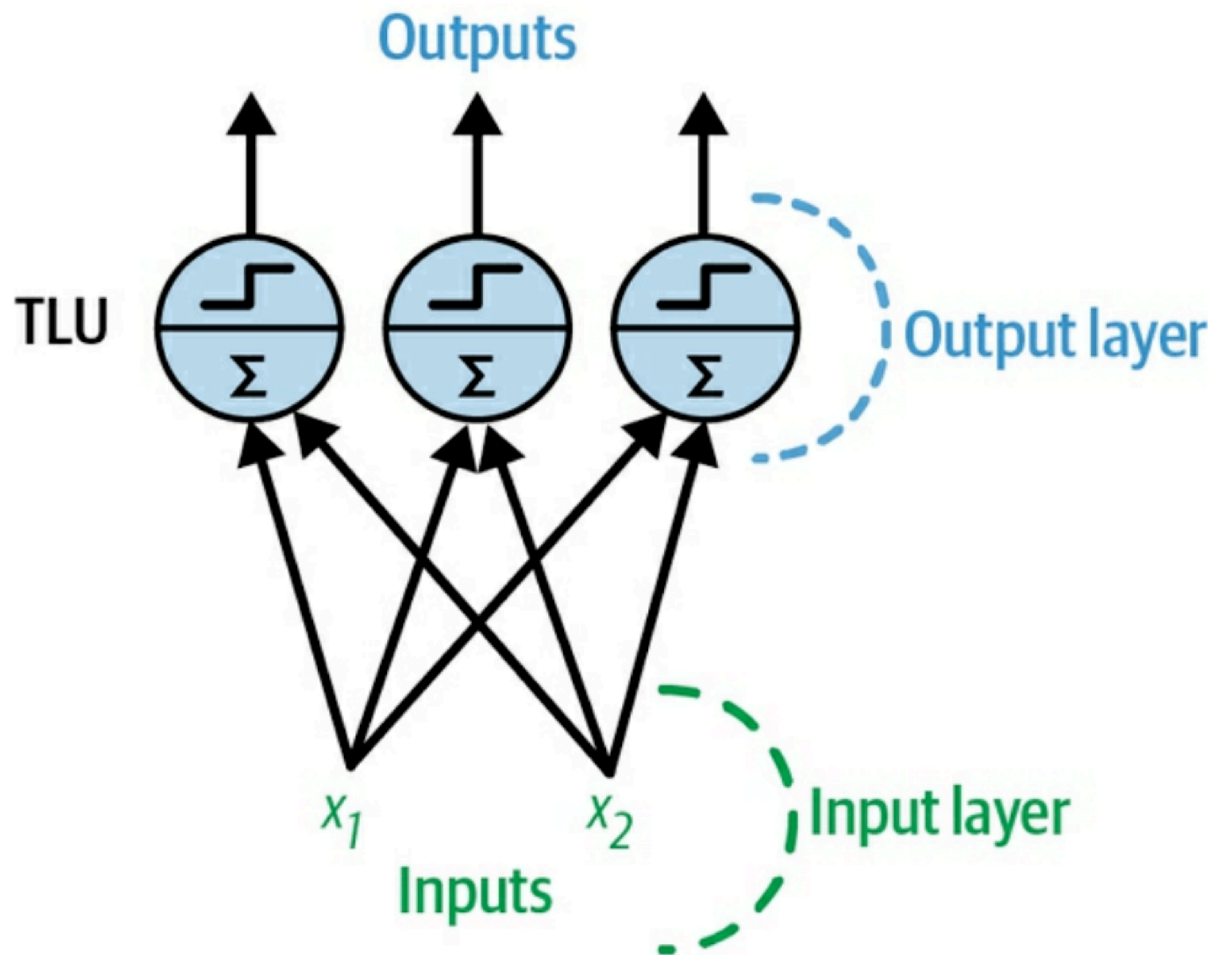
TLU (Threshold Logic Unit)

- One TLU can perform binary classification
- Input data like petal width and height
- Output sorts inputs into two categories
- Training will determine the correct weights ***w*** and bias ***b***



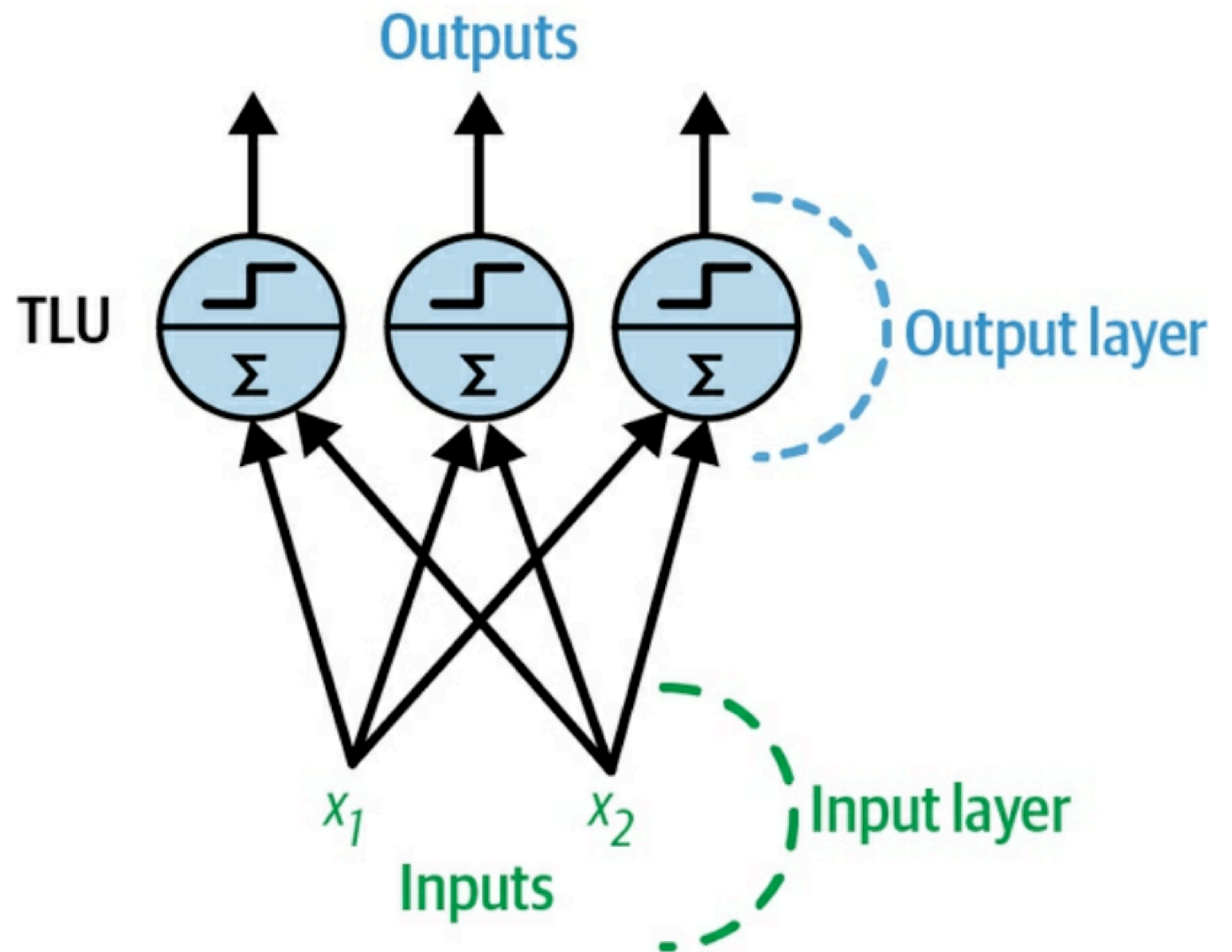
Perceptron

- A layer of TLUs
- Every TLU is connected to every input
  - ***fully connected layer*** or
  - ***dense layer***
- Inputs are called the ***input layer***
- TLUs are the ***output layer***



Perceptron

- This perceptron can classify instances into three classes



Training a Perceptron

- ***Hebb's rule*** or ***Hebbian Learning***
  - When a neuron triggers another neuron often
  - The connection between them gets stronger
  - "Cells that fire together, wire together"
- Perceptrons use the error of a prediction also
  - Reinforces connections that reduce error

Training a Perceptron

- Resembles stochastic gradient descent

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

In this equation:

- **$w_{i,j}$ is the connection weight between the ith input and the jth 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 (see Chapter 4).**

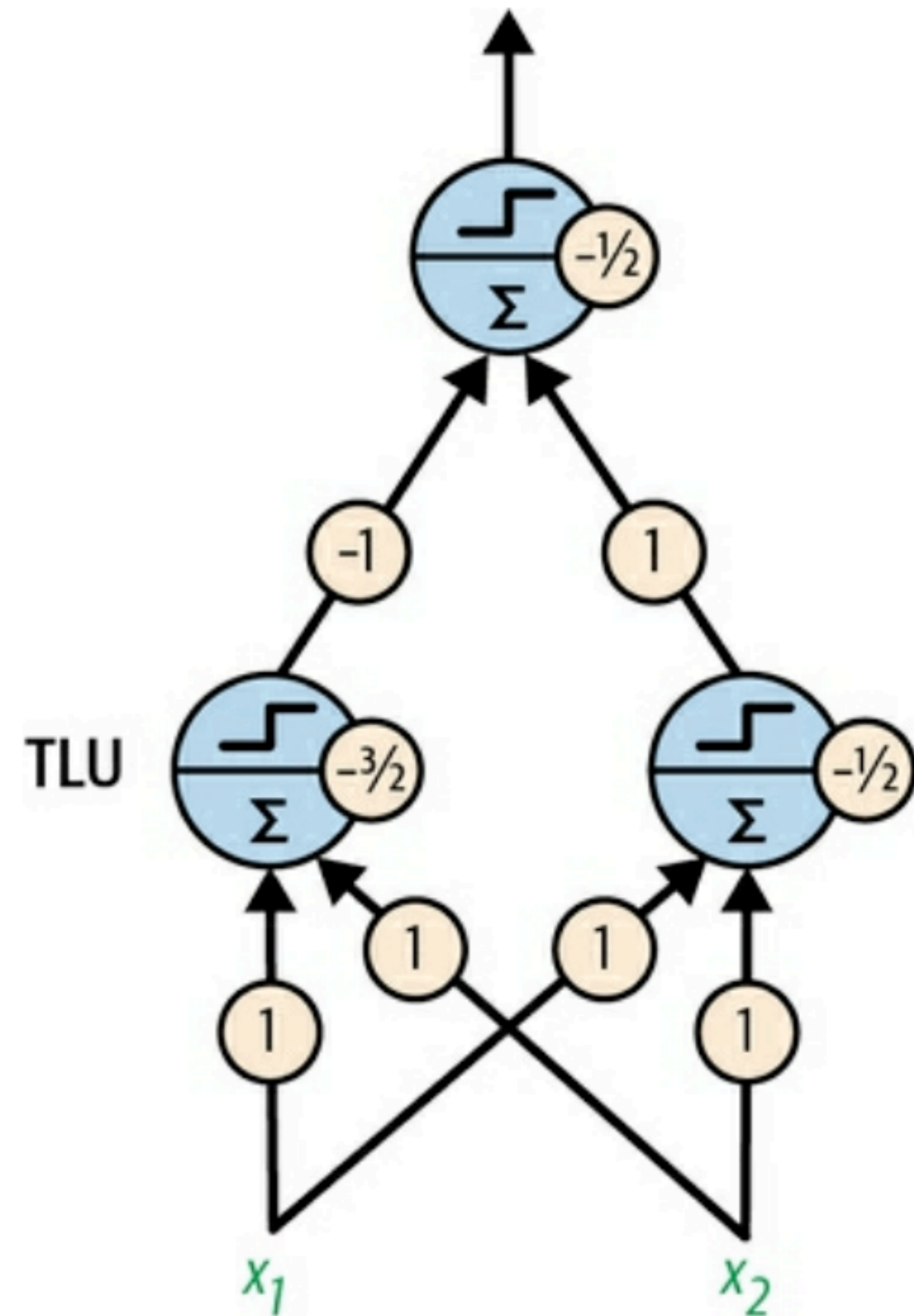
Limitations of a Perceptron

- Calculation is linear in all inputs
- Cannot compute XOR

	x1 = 0	x1 = 1
x2 = 0	0	1
x2 = 1	1	0

Two Perceptron Layers Can Do XOR

- Inputs at bottom
  - All weights 1; Bias: -3/2 and -1/2
  - Each neuron outputs with a step function
    - If > 0, output 1
    - If <= 0, output 0



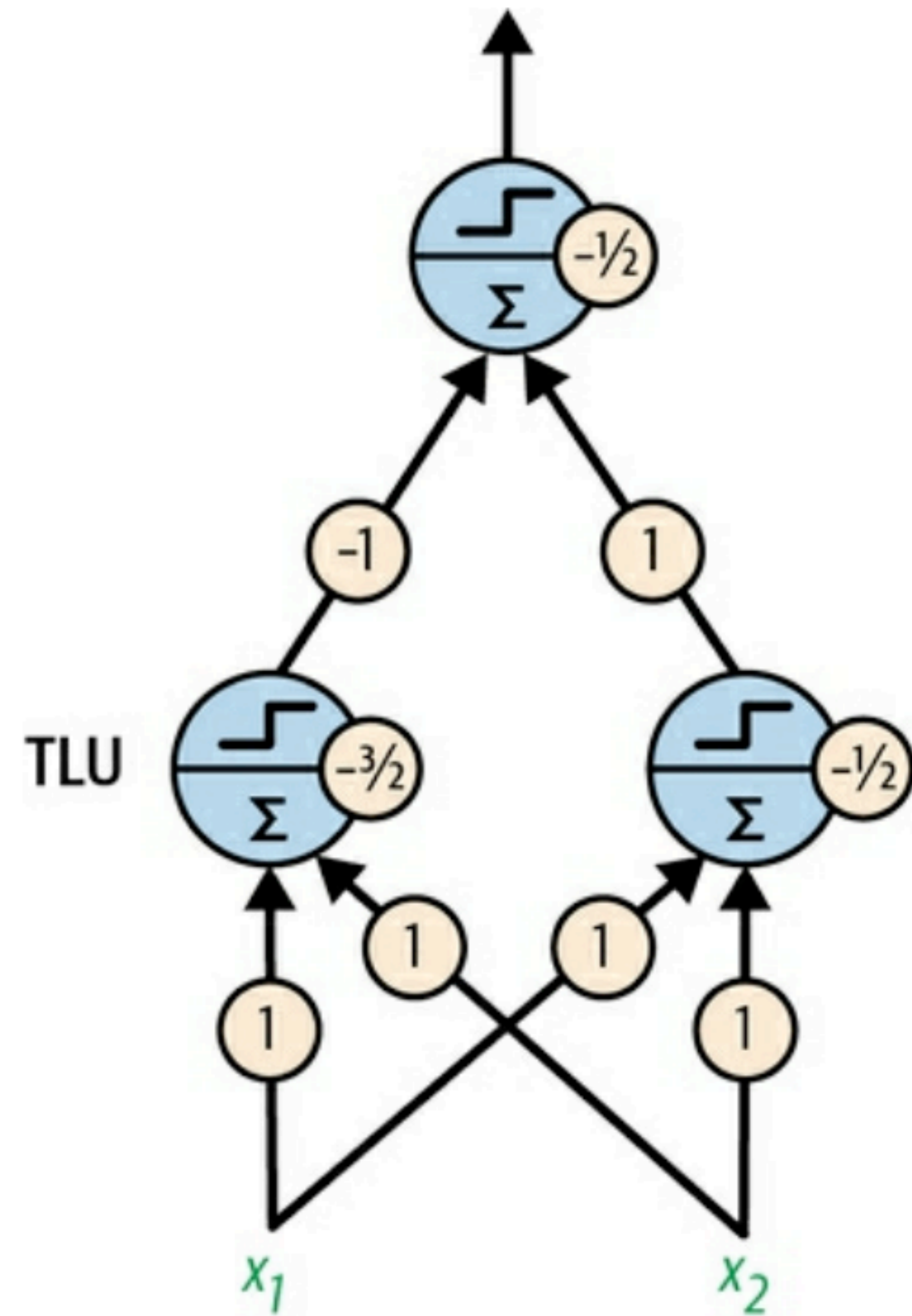
Two Perceptron Layers Can Do XOR

0

0 + 0 - 1/2 = -1/2

0	0
0 + 0 - 3/2 = -1.5	0 + 0 - 1/2 = -0.5

0	0



Two Perceptron Layers Can Do XOR

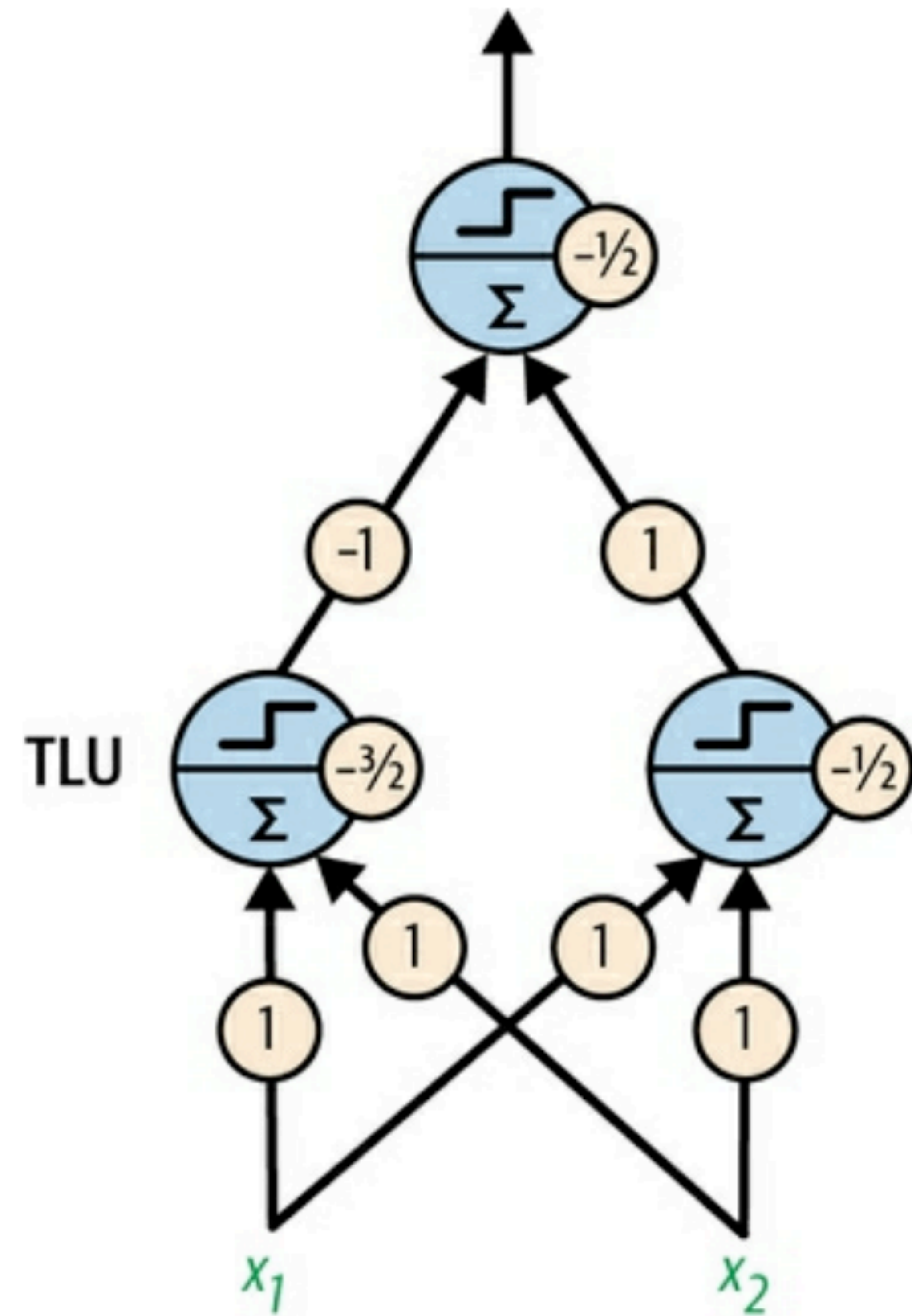
0

-1 + 1 - 1/2 = -1/2

1 1

1 + 1 - 3/2 = 0.5 1 + 1 - 1/2 = 1.5

1 1



Two Perceptron Layers Can Do XOR

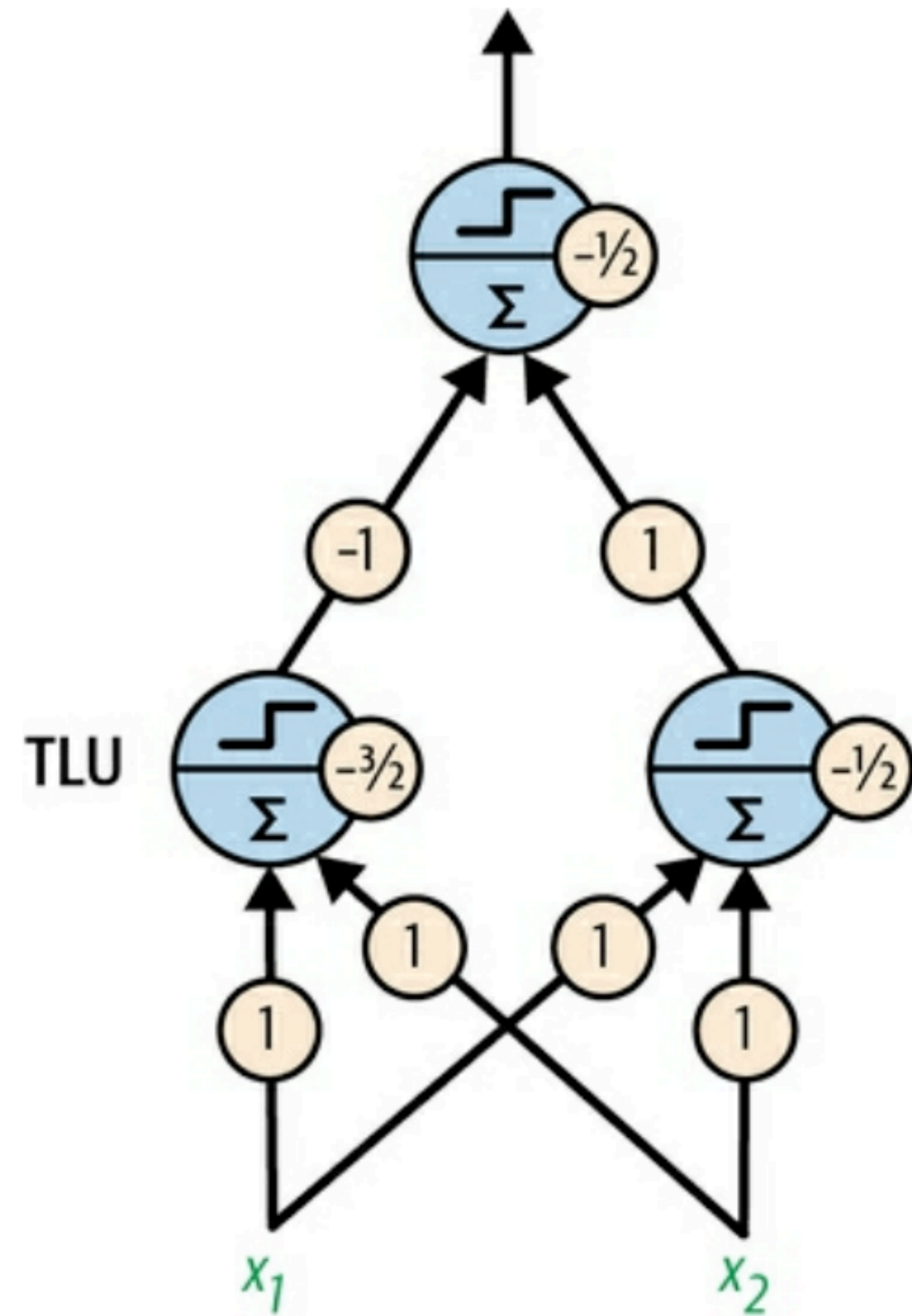
1

0 + 1 - 1/2 = 1/2

0   1

0 + 1 - 3/2 = -0.5   0 + 1 - 1/2 = 0.5

0   1



Two Perceptron Layers Can Do XOR

1

0 + 1 - 1/2 = 1/2

0

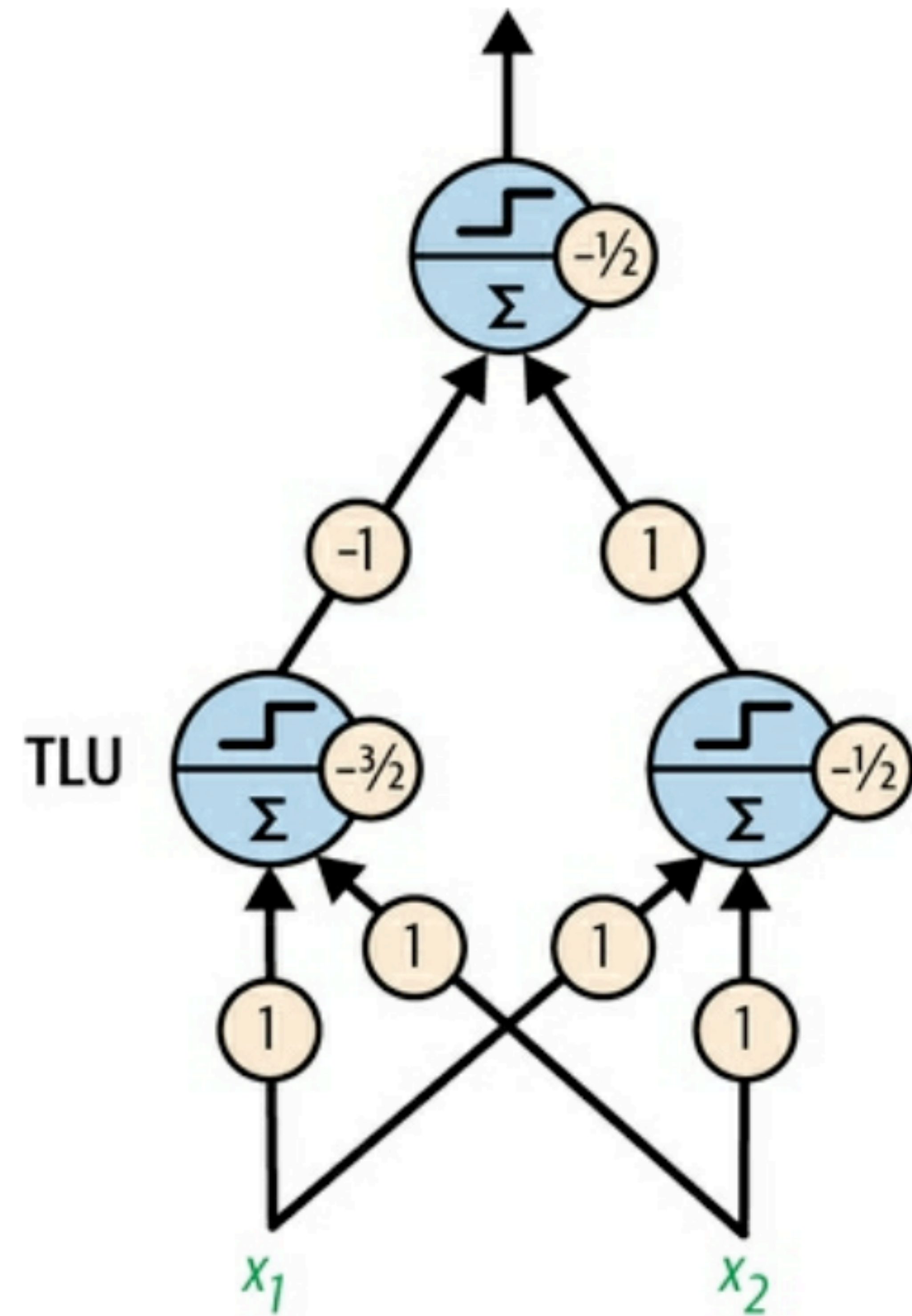
1 + 0 - 3/2 = -0.5

1

1 + 0 - 1/2 = 0.5

1

0



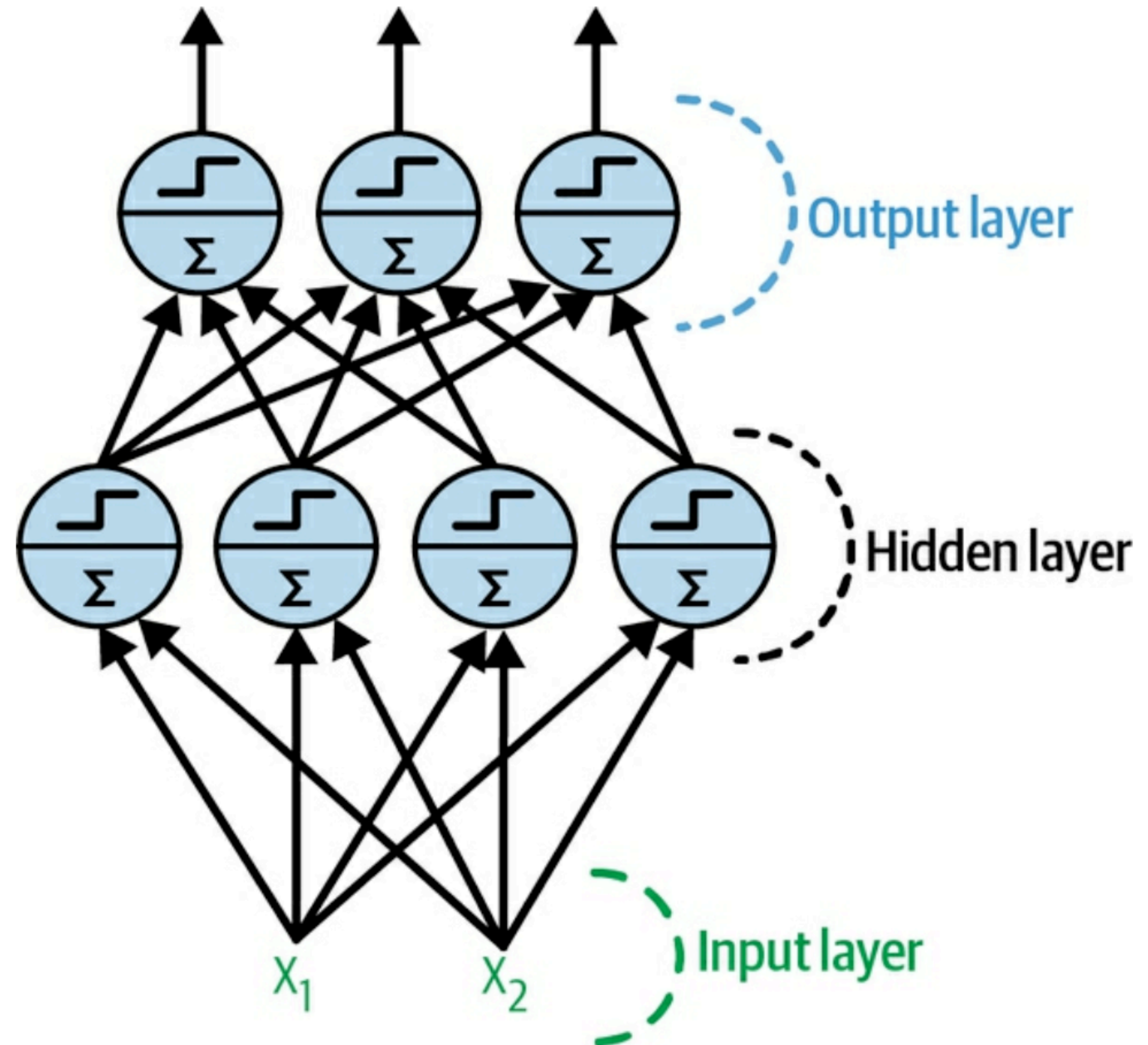
Kahoot!

Ch 10a

The Multilayer Perceptron and Backpropagation

Multilayer Perceptron

- Signal flows one direction
- This is a ***feedforward neural network (FNN)***
- If there's a deep stack of hidden layers, it's a ***deep neural network***



Backpropagation

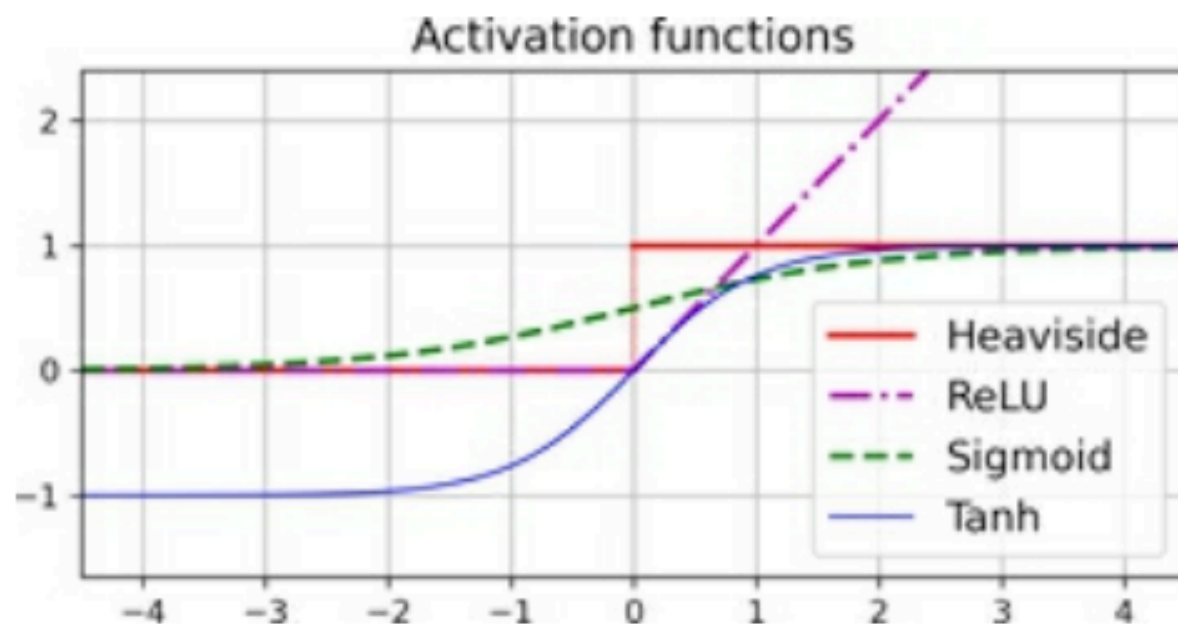
- In 1970, ***reverse-mode automatic differentiation*** or ***reverse-mode autodiff*** was developed
- Can calculate the gradient of the error for all parameters
    - In two passes through the network
    - One forward, one backward
- Makes gradient descent much easier to calculate
- The combination of reverse-mode autodiff and gradient descent is called ***backpropagation*** or ***backprop***

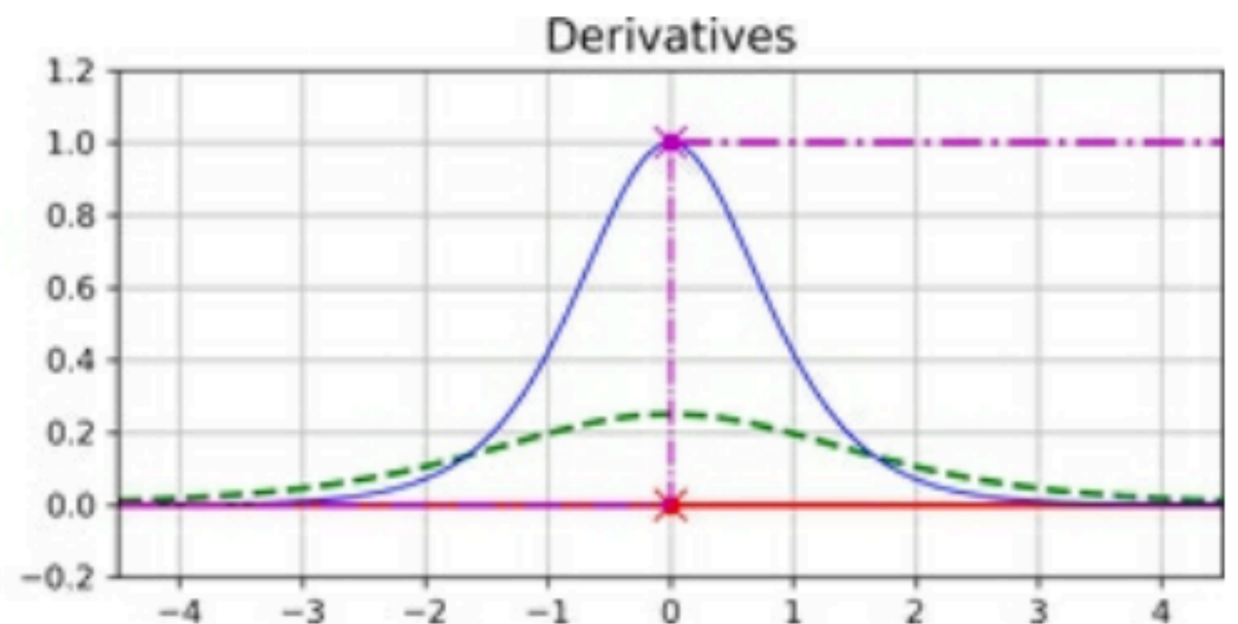
Backpropagation Steps

- Initialize all weights randomly
- Use a mini-batch, such as 32 instances
- Run the batch through the network to the end, keeping all calculated values
- Measure the network error
- Compute how much each weight and bias contributed to the error analytically with the chain rule
- Perform a gradient descent step in the direction to best lower the error

Activation Functions

- Must use a function with a gradient, not a step function, like:
  - Sigmoid $\sigma(z) = 1 / (1 + \exp(-z))$
  - Hyperbolic tangent $\tanh(z) = 2\sigma(2z) - 1$
  - Rectified linear unit function $ReLU(z) = \max(0, z)$
    - Fast to compute
    - The default

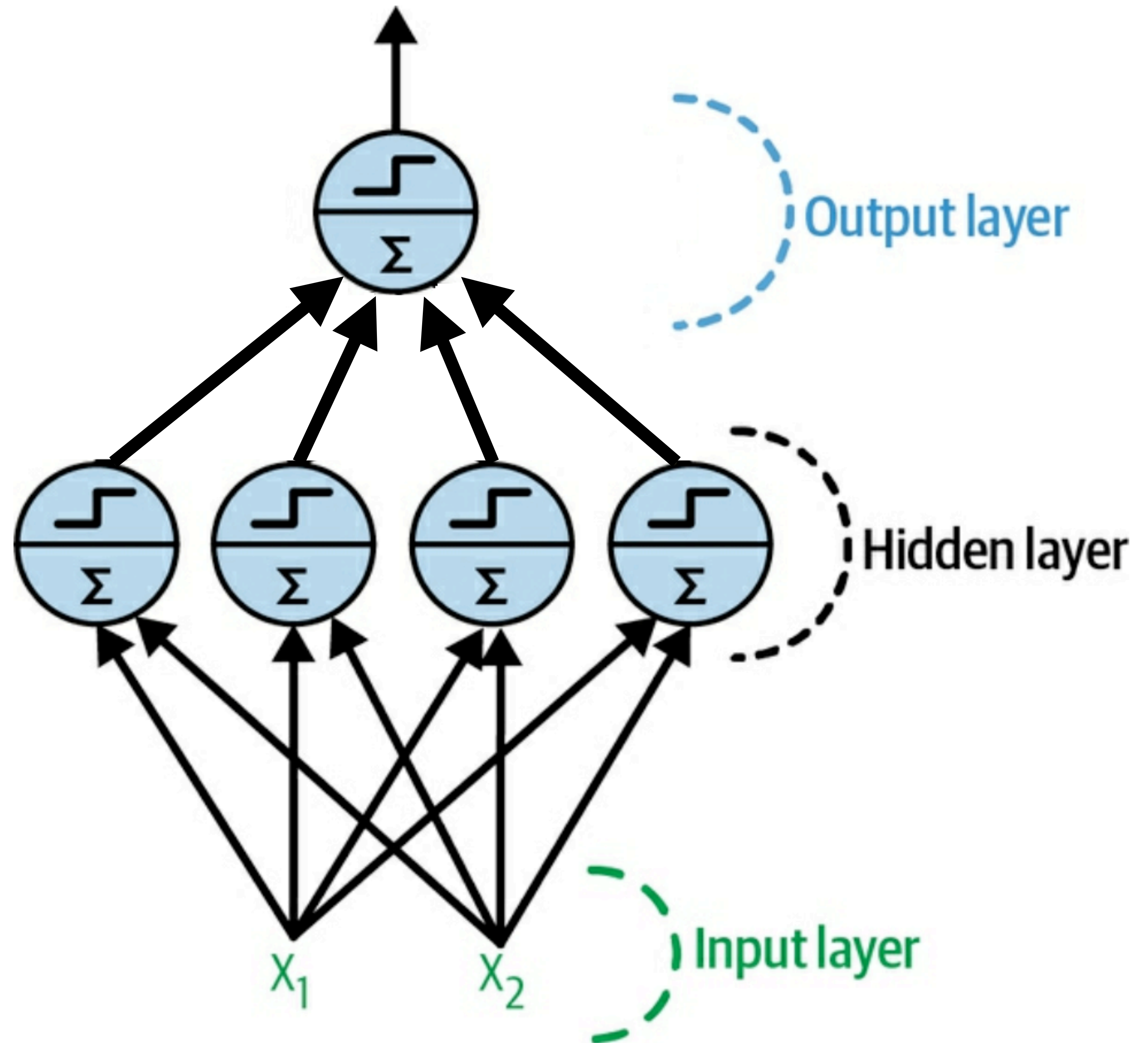




Regression MLPs

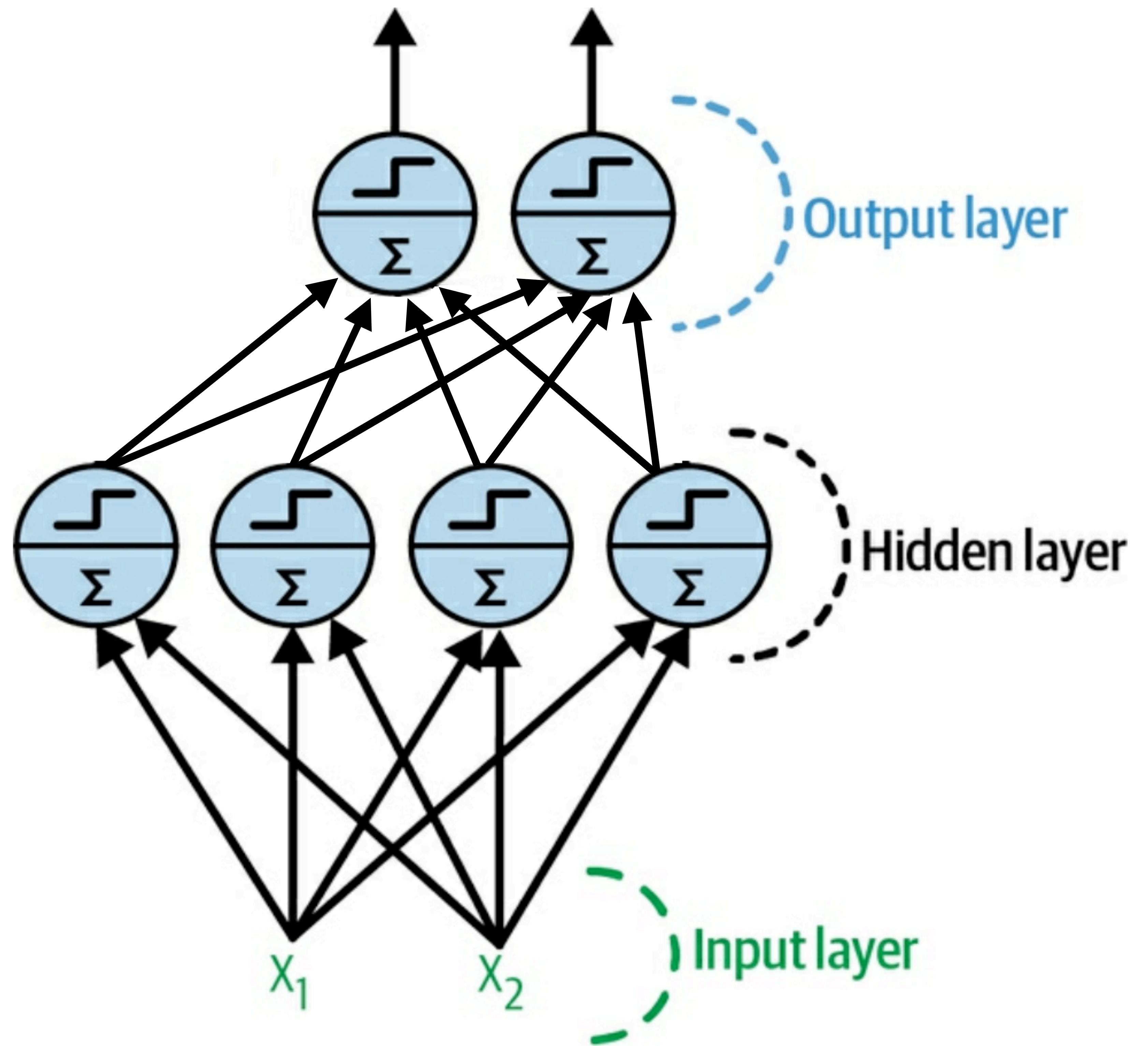
Predicting a Single Value

- One output neuron



Predicting a 2-D Value

- Two output neurons
- Ex: specify X and Y coordinates of the center of an object in a image



Activation Functions

- Output neuron's activation function
  - For varying value, leave its output unchanged
  - To guarantee positive value, use
    - ReLU, or
    - ***softplus*** $\text{softplus}(z) = \log(1 + \exp(z))$
      - Smooth variant of ReLU
    - To restrict range, use sigmoid or tanh

Error Measurement

- Usually, mean squared error
- If you have a lot of outliers, use
  - Absolute error, or
  - ***Huber loss***, which combines both

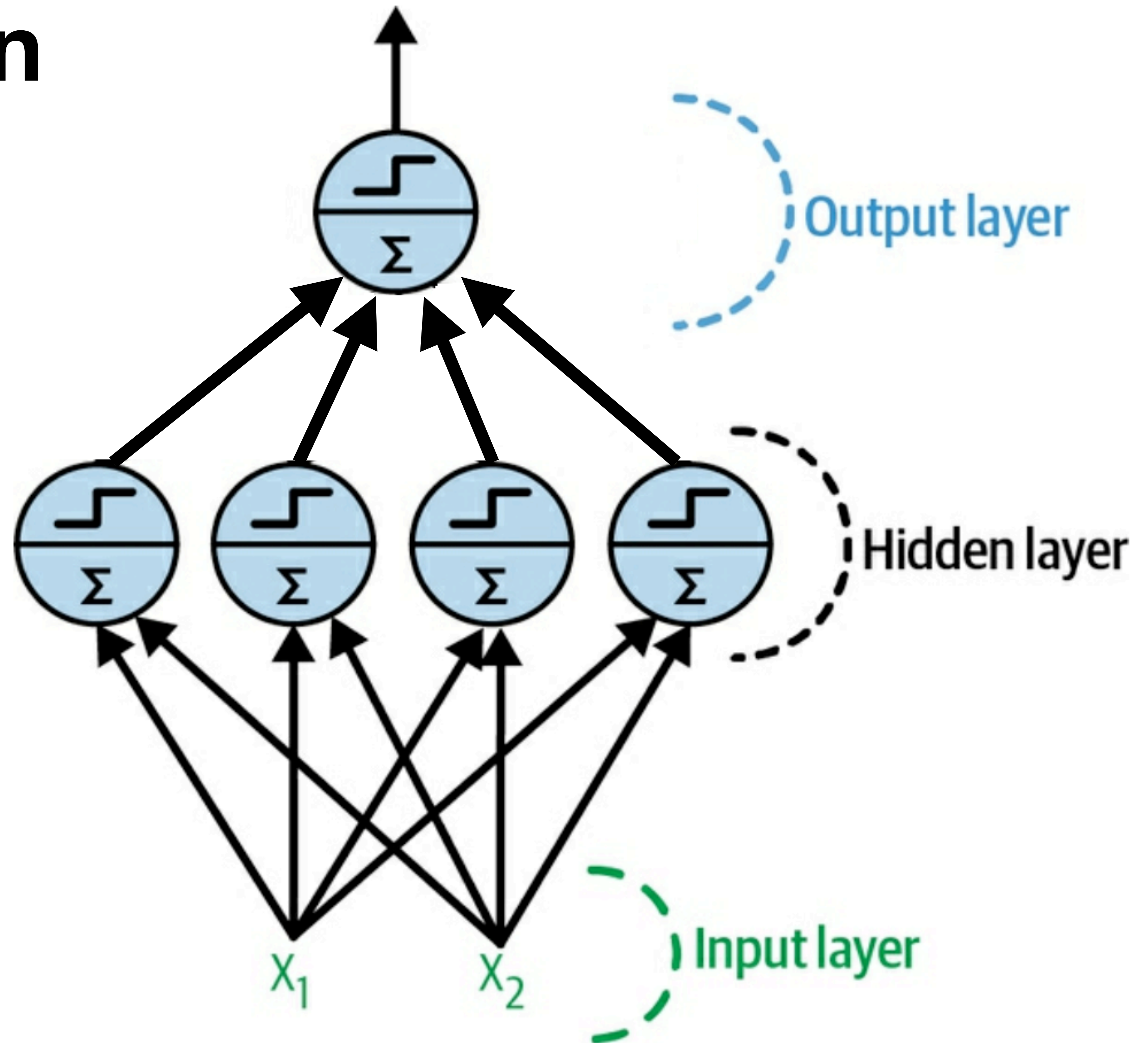
Typical Regression MLP Architecture

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

Classification MLPs

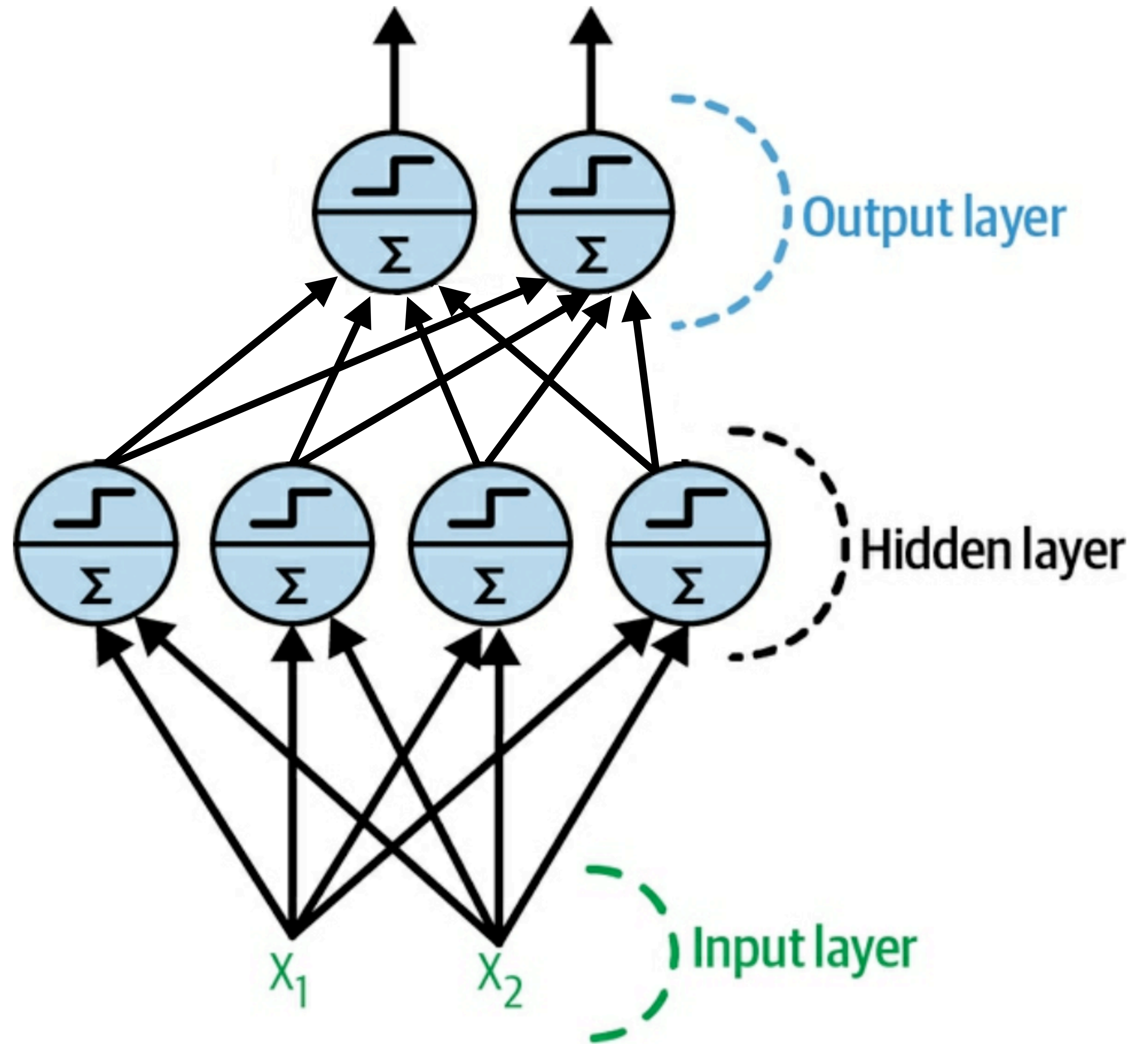
Binary Classification

- One output neuron
- Sigmoid activation function
- Output between 0 and 1
- Probability of positive class



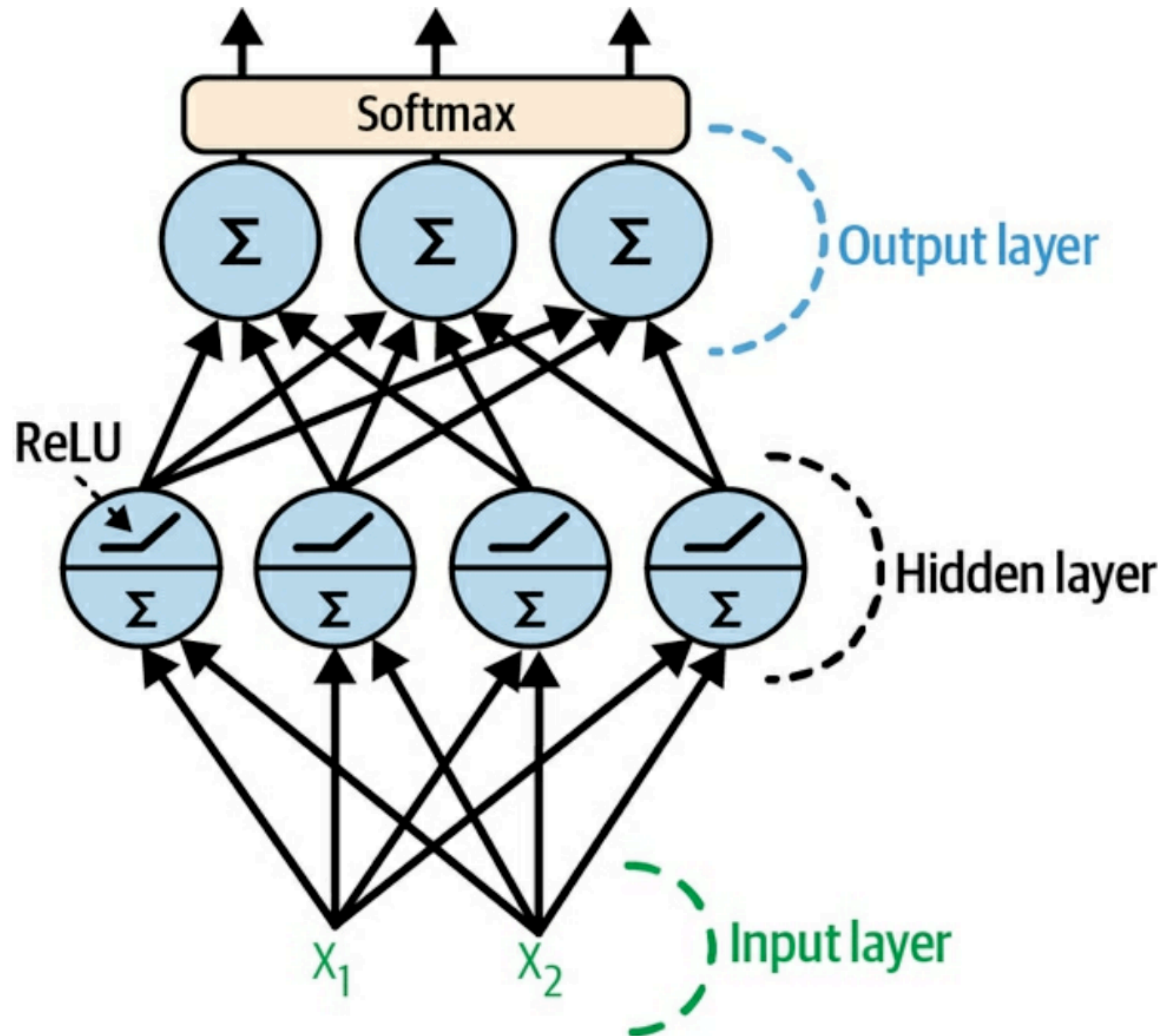
Multilabel Classification

- Ex: label email **spam** and **urgent**
- Two output neurons
- Using sigmoid activation function
- Outputs are probabilities of each positive class



Multilabel Classification

- If only one class per instance is allowed, use softmax activation function for the whole output layer
- Makes total probability one



Softmax

- Assigns a probability to each class
- The total of them is always 1

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k\left(\mathbf{x}\right)\right)}{\sum_{j=1}^{K} \exp\left(s_j\left(\mathbf{x}\right)\right)}$$

- ***K* is the number of classes.**
- **s(x) is a vector containing the scores of each class for the instance x.**
- **$\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance x belongs to class *k*, given the scores of each class for that instance.**

Typical Classification MLP Architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

Cross Entropy

- Cost function for multilabel classification
- If predictions are correct, that is near 1 for correct predictions, the cross entropy is near 0
- Erroneous predictions make the cross entropy larger

Equation 4-22. Cross entropy cost function

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

In this equation, $y_k^{(i)}$ is the target probability that the *i*th instance belongs to class *k*. In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Kahoot!

Ch 10b

Implementing MLPs with Keras

ML 101: Computer Vision

- Fashion MNIST
- 28x28 grayscale images of clothing

Loading the Dataset

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

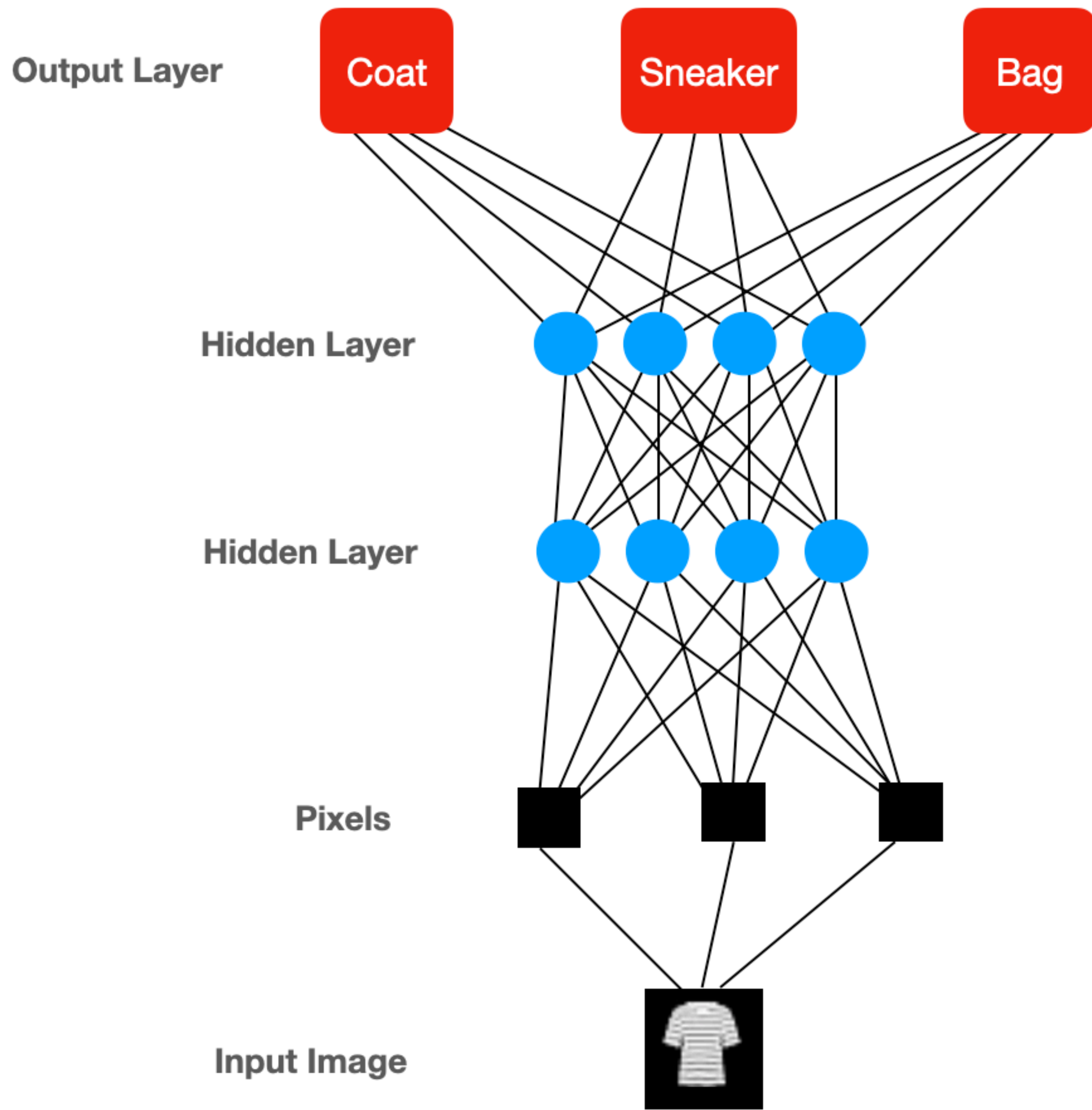
```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

- Load images
- Set aside a validation set
- Divide by 255 to normalize the brightness values

Creating the Model

- **Sequential** makes the usual model, with a single stack of layers
- **Flatten** converts the 28x28 array to a one-dimensional list of 784 values
- **Dense** makes the hidden layers, with the ReLU activation function
- The final **Dense** makes the output neurons, with softmax to make all the probabilities total to 1

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```



Output Layer
Coat
Sneaker
Bag
Hidden Layer
Hidden Layer
Pixels
Input Image

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 300)               235500

 dense_1 (Dense)             (None, 100)               30100

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model

- **Crossentropy** is the appropriate error measure for a task where the model must assign only one of many labels
- **sgd** is Stochastic Gradient Descent (with backpropagation)
  - You usually want to specify learning_rate; here we accept the default of 0.01
- **metrics=["accuracy"]** measures accuracy during training and evaluation

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Training with fit()

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Epoch 1/30
1719/1719 [==============================] - 2s 989us/step
  - loss: 0.7220 - sparse_categorical_accuracy: 0.7649
  - val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332
```

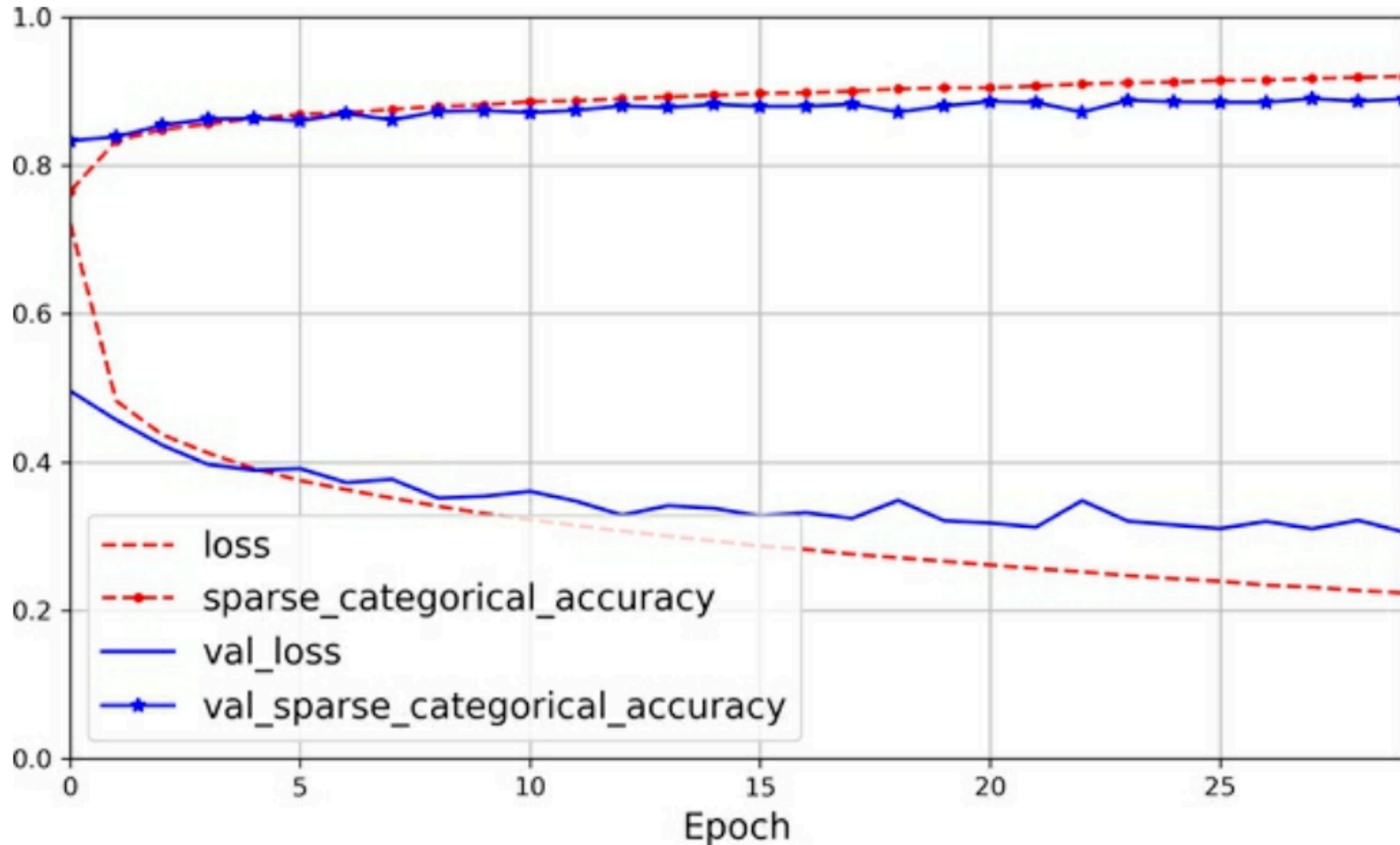
- **1719** is the number of mini-batches
- accuracy on training set and validation set are shown

Training with fit()

```
Epoch 30/30
1719/1719 [==============================] - 2s 963us/step
  - loss: 0.2235 - sparse_categorical_accuracy: 0.9200
  - val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

- More accurate on training set than validation set
- A small amount of overfitting

Learning Curves

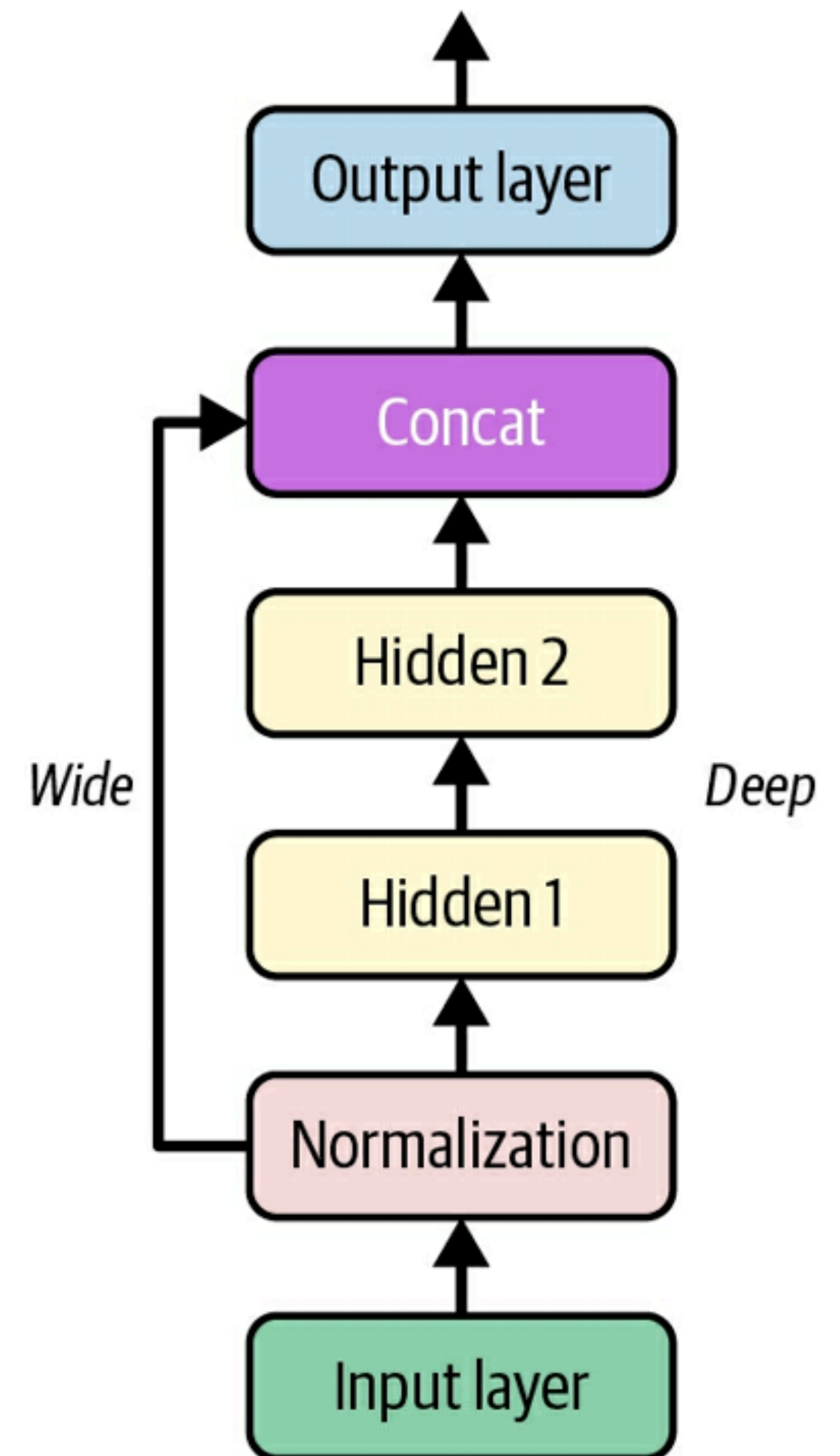


Improving Performance

- Tune the hyperparameters
  - First, adjust **learning_rate**
  - Then try changing number of layers, neurons per layer, and activation functions

Wide & Deep Neural Network

- Introduced in 2016
- Inputs connect directly to outputs
- It can learn both
    - **deep patterns** (through all the layers), and
    - **simple rules** (through the short path)



Creating Layers

```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)
```

- **normalization** layer standardizes inputs
- **hidden** layers operate as usual
- **concatenate** layer combines all the inputs into one tensor
- **output** layer operates as usual

Putting the Layers in Order

- **concat** combines the normalized input and the deep learning output
- **model** creates the model

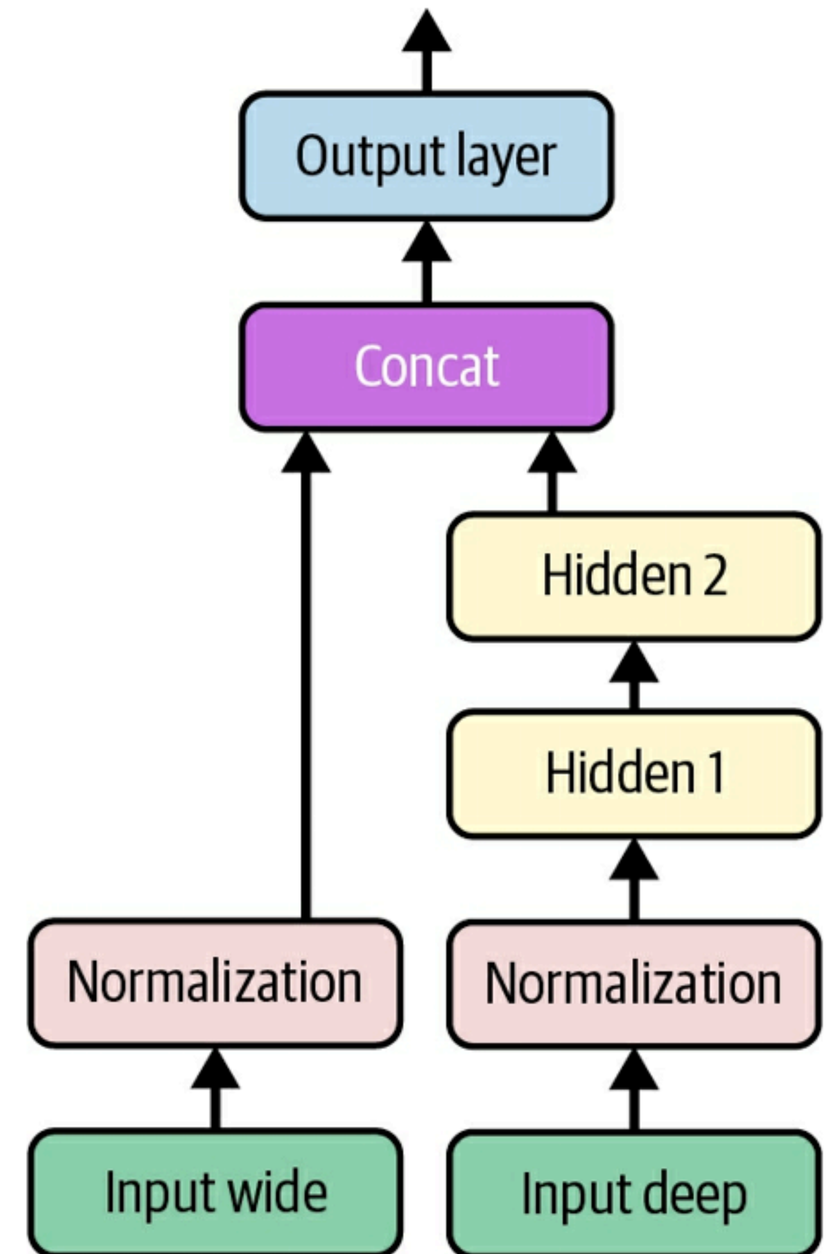
```
input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```

Handling Multiple Inputs

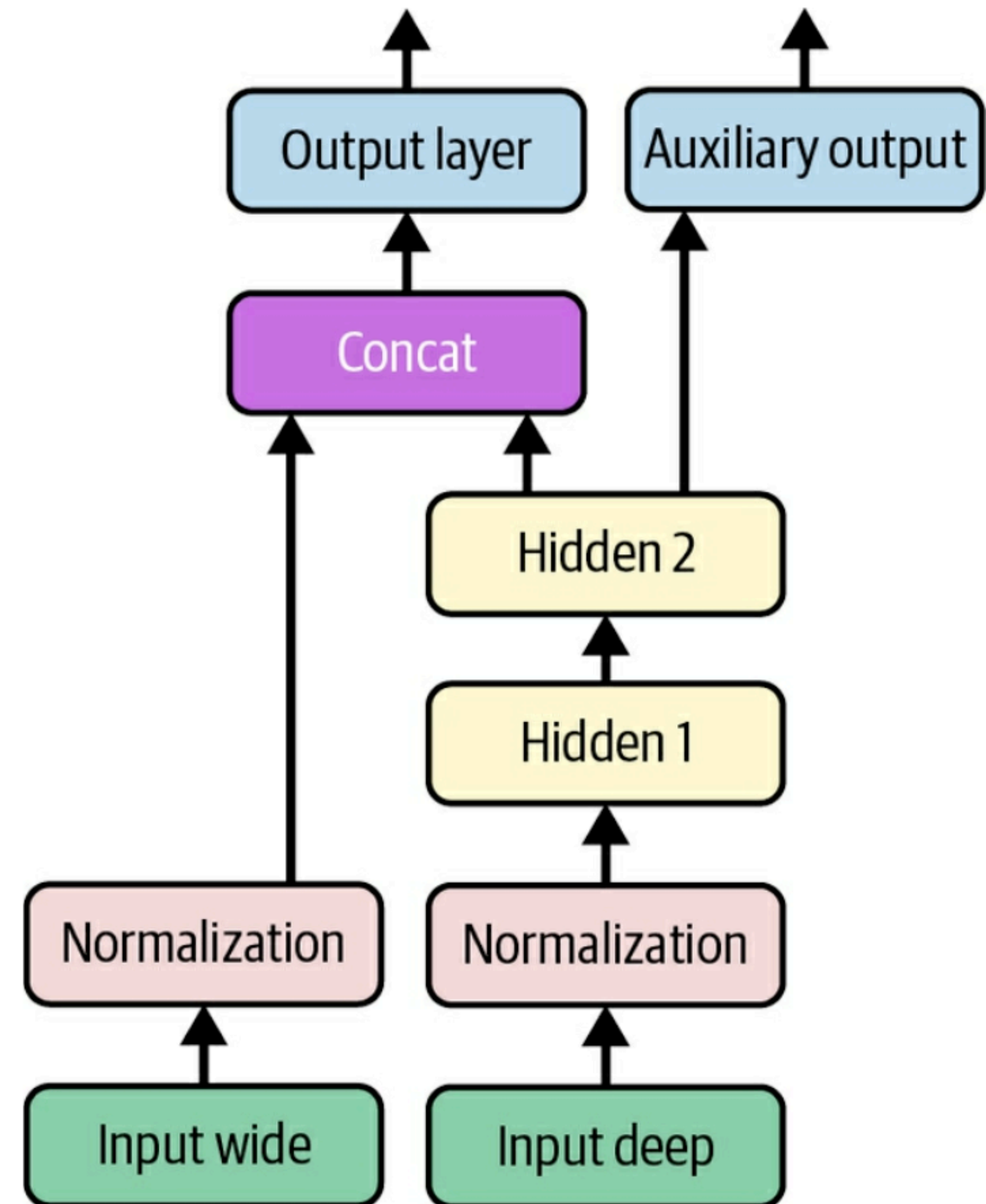
- Some inputs go wide, others go deep

```
input_wide = tf.keras.layers.Input(shape=[5])  # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6])  # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```



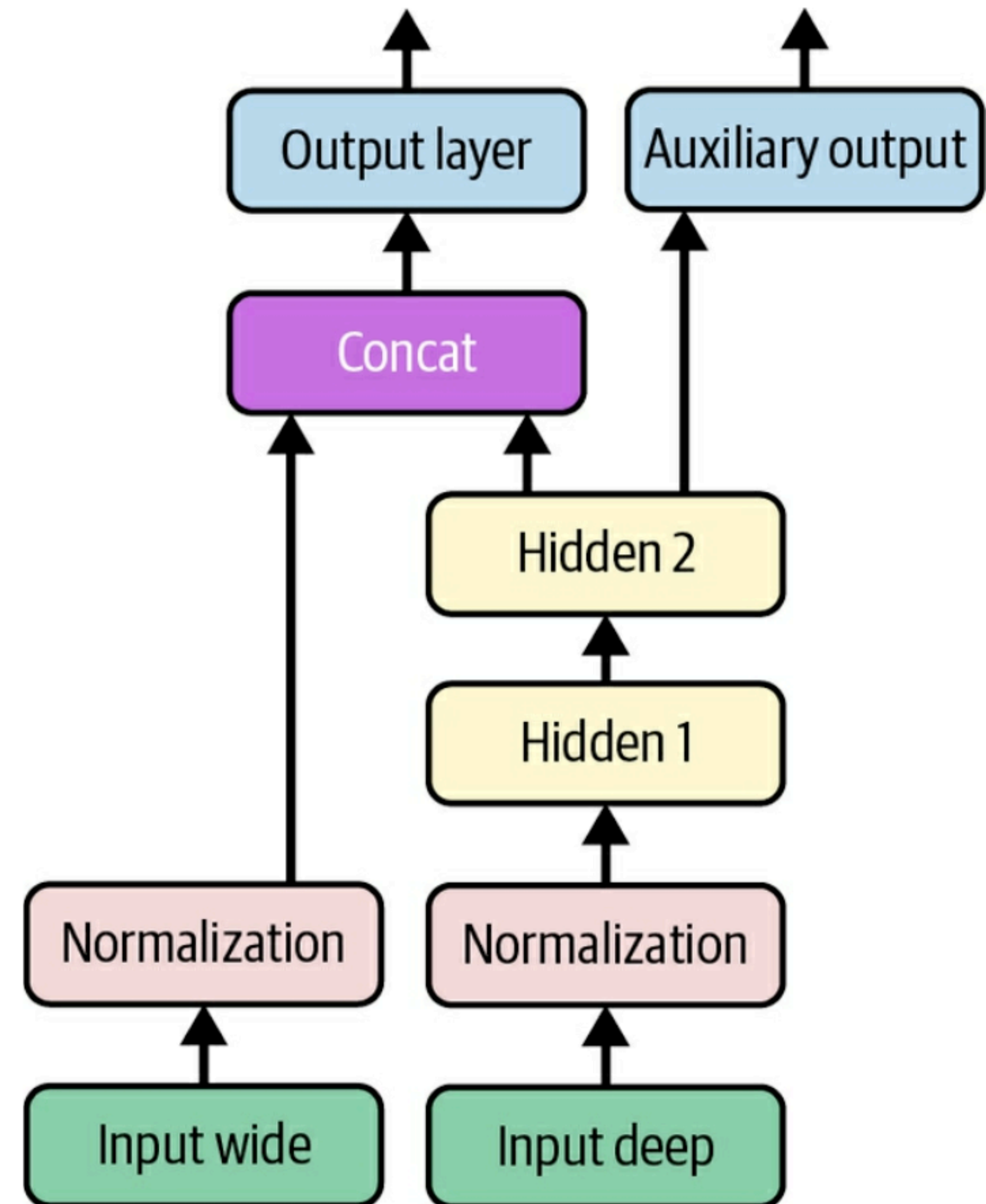
When to Use Multiple Outputs

- The task may demand it, such as locating and classifying the main object in a picture
  - This is both regression and classification
- You may have multiple independent tasks on the same data
  - One network is often better than several, because it can learn features that are useful across tasks



When to Use Multiple Outputs

- For regularization
  - The auxiliary output can ensure that the underlying part of the network learns something useful on its own



Multiple Outputs

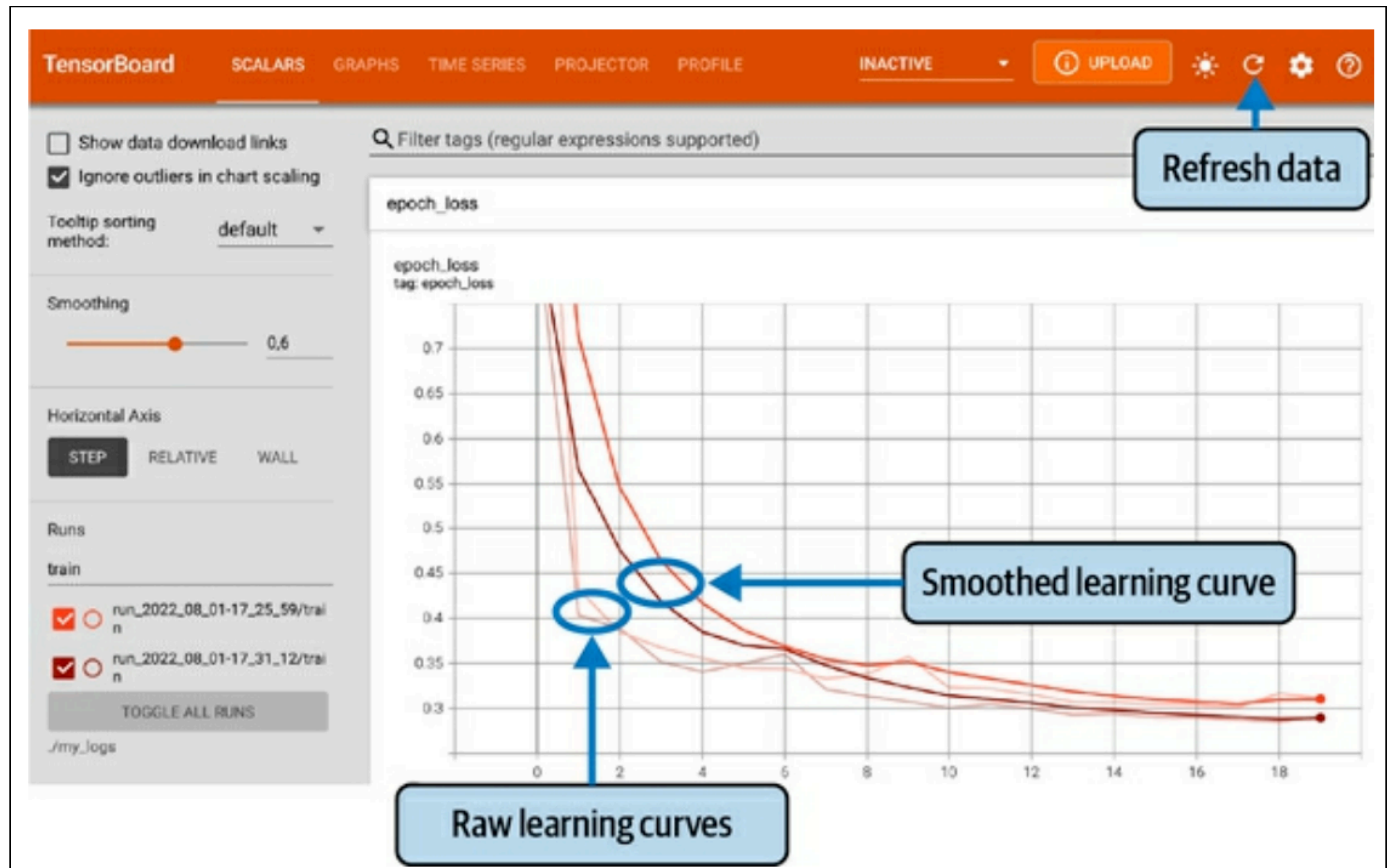
- Each output needs its own loss function
- The data needs labels for each output

Dynamic Models

- Keras can be used to make models without a fixed structure
- Including **for** loops, **if** statement, etc.

Using TensorBoard for Visualization

- Can view learning curves, statistics, find speed bottlenecks, and more



Fine-Tuning Neural Network Hyperparameters

Hyperparameters

- A basic MLP has:
  - Number of layers
  - Number of neurons
  - Activation functions
  - Weight initialization logic
  - Optimizer
  - Learning rate
  - Batch size
  - etc.

Tuning Strategies

- Scikit-learn offers grid search and randomized search options
- **Keras Tuner** integrates with TensorBoard
  - Optimizes hyperparameters using SGD or Adam (like SGD but varies the learning rate)

Number of Hidden Layers

- A MLP with one hidden layer can theoretically model anything
- But for complex problems, deep models have a higher ***parameter efficiency***
    - Models with fewer neurons
- Because, like human brains, one layer finds low-level components like edges
- Higher levels look at larger-scale features
- Highest level finds whole meaningful shapes, like faces

Transfer learning

- A new model can start from pre-trained lower levels
- Image classification and speech recognition models typically use dozens or hundreds of layers
- But rarely are trained from scratch
  - You reuse parts of a pre-trained network that performs a similar task

Number of Neurons per Hidden Layer

- Input and output layers are set by the problem
  - MNIST has 28 x 28 inputs and 10 outputs
- Old way: hidden layers in a pyramid shape
  - More neurons at the lower layers
  - For MNIST, 300, 200, 100
- But using the same number of neurons in each layer seems to work better, and is now the standard
- One way to select numbers:
  - Gradually increase the # of neurons per payer and the # of layers until you get overfitting

Stretch Pants

- Start with more layers and neurons than you need
- Use early stopping and other regularization to prevent excessive overfitting
- This avoids the problem of "bottleneck" layers
  - Too weak to represent the data
  - Information is lost and cannot be recovered

Learning Rate

- The most important hyperparameter
- Optimal learning rate is half the maximum learning rate
  - Above that, the model diverges
- One way: train the model for a few iterations at a very low learning rate like 10^{-5}
  - Gradually increase the rate to a large value like 10
  - Find the value where loss starts to rise dramatically
  - Use a rate 1/10 of that rate

Other Parameters

- **Optimizer** -- discussed in later chapters
- **Batch size** -- unclear, some people prefer large batches like 8,192 to fill GPU RAM, others prefer batches less than 32
- **Activation function** -- ReLU is good for hidden layers
- **Number of iterations** -- Just use early stopping instead

Kahoot!

Ch 10c