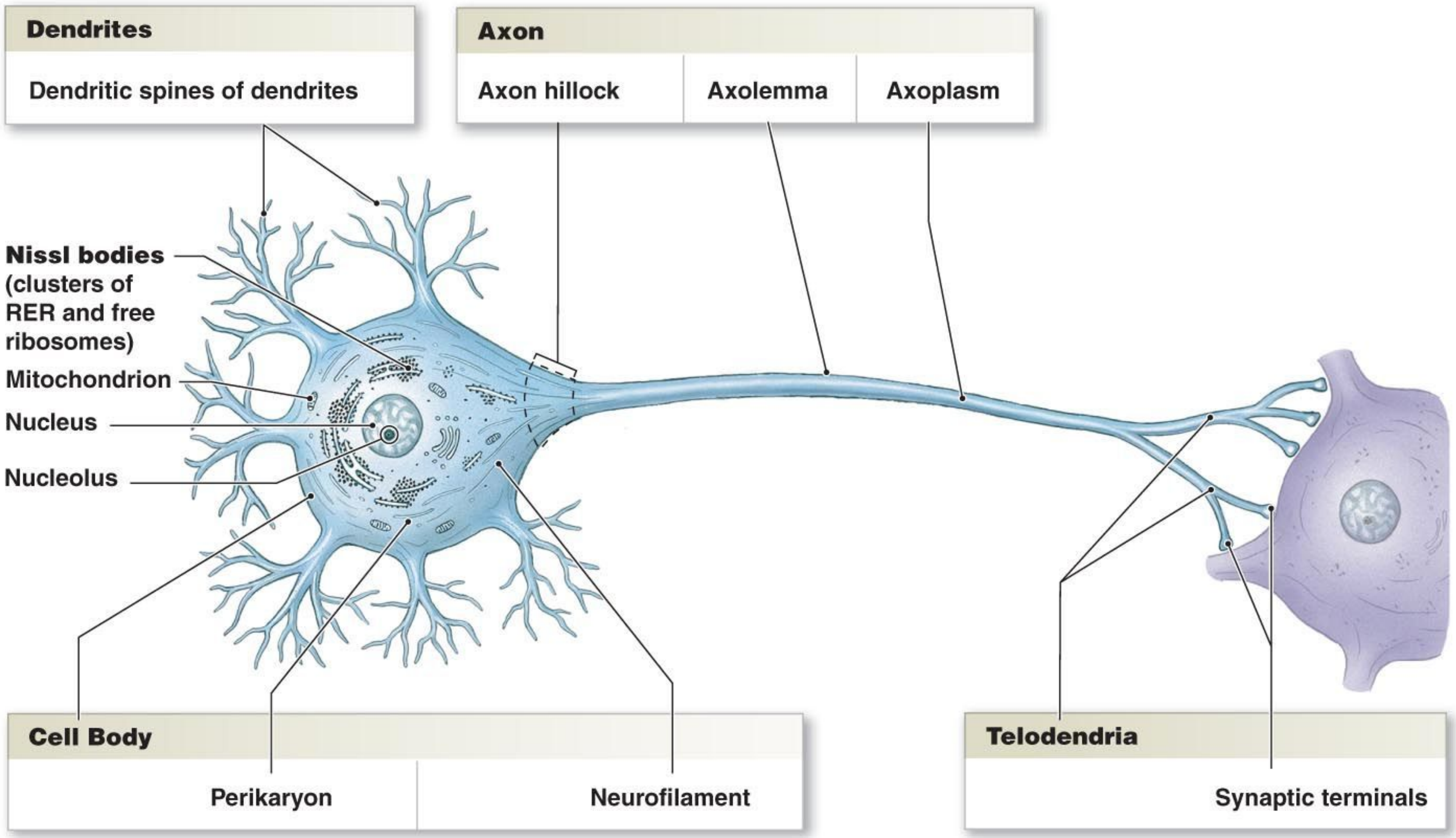
Neural Network

Dr. Dongchul Kim

Neural Network

- A neural network is a method in AI that teaches computers to process data in a way that is inspired by the **human brain**.
- It is a type of machine learning process, called deep learning, that uses **interconnected nodes (neurons) in a layered structure** that resembles the human brain.

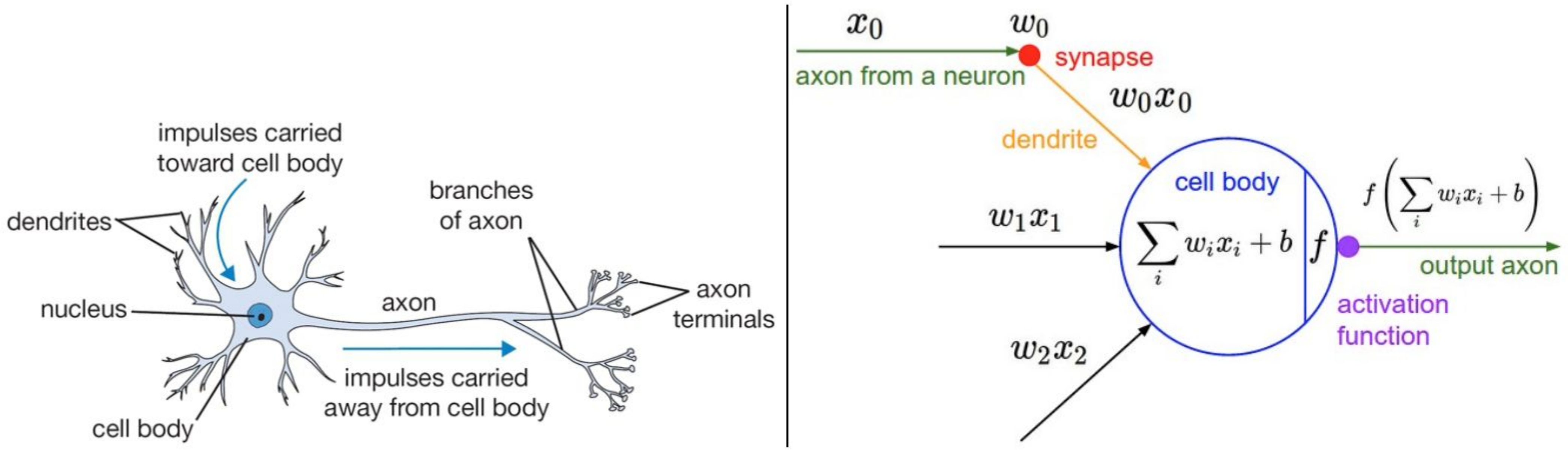
A diagrammatic view of a representative neuron



© 2011 Pearson Education, Inc.

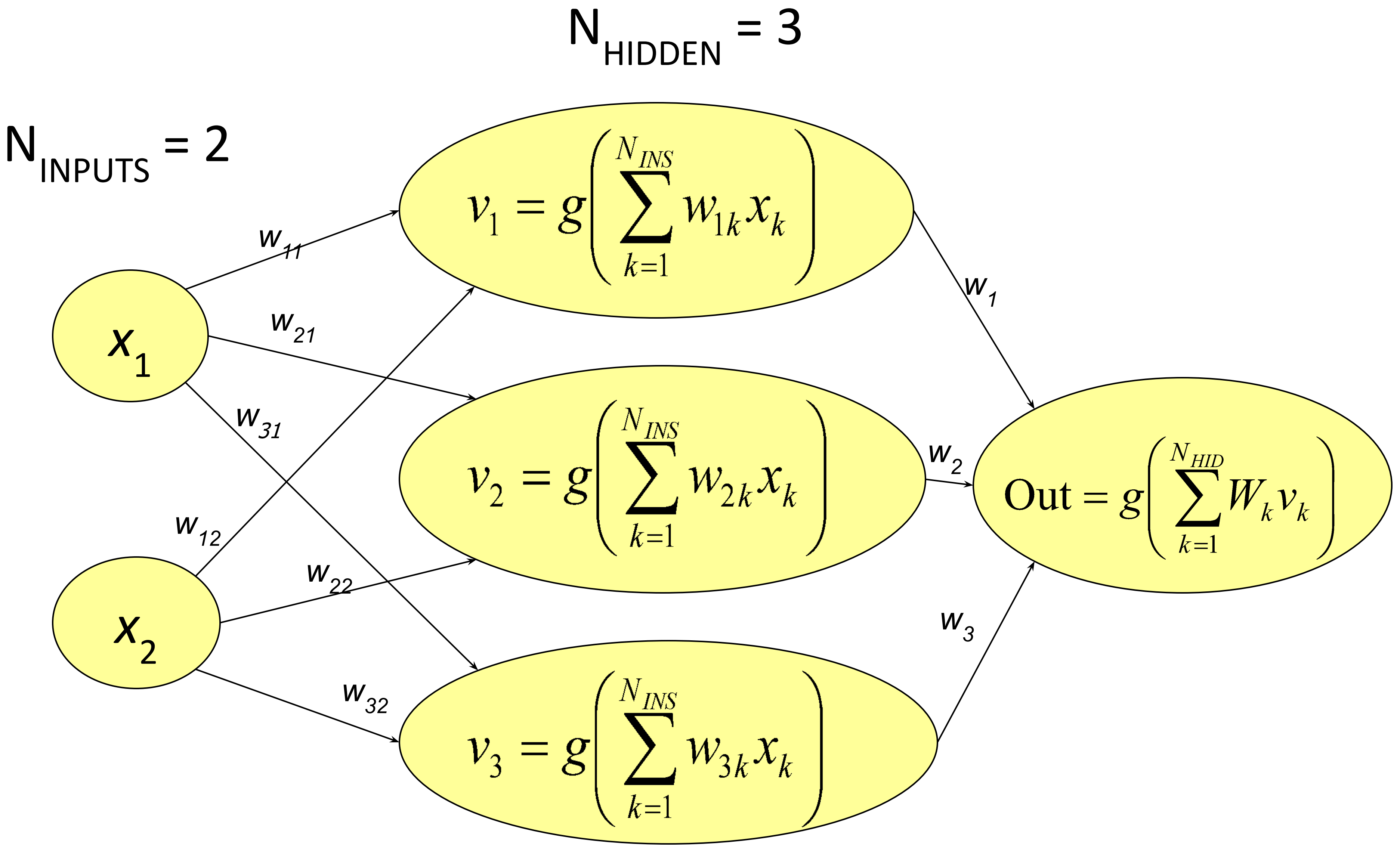
Activation Function

- the activation function of a node defines the output of that node given an input or set of inputs.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

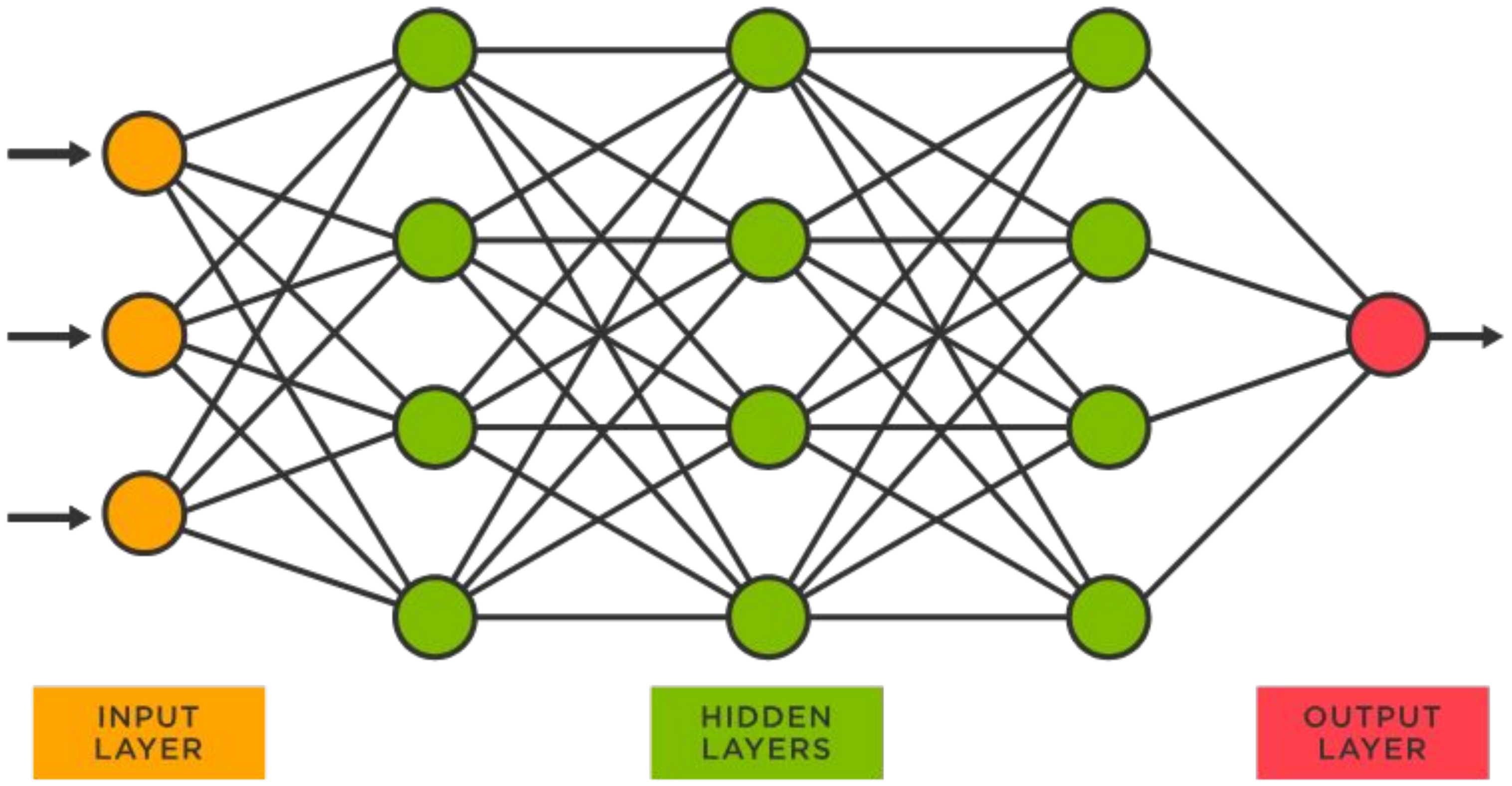
A 1-HIDDEN LAYER NET



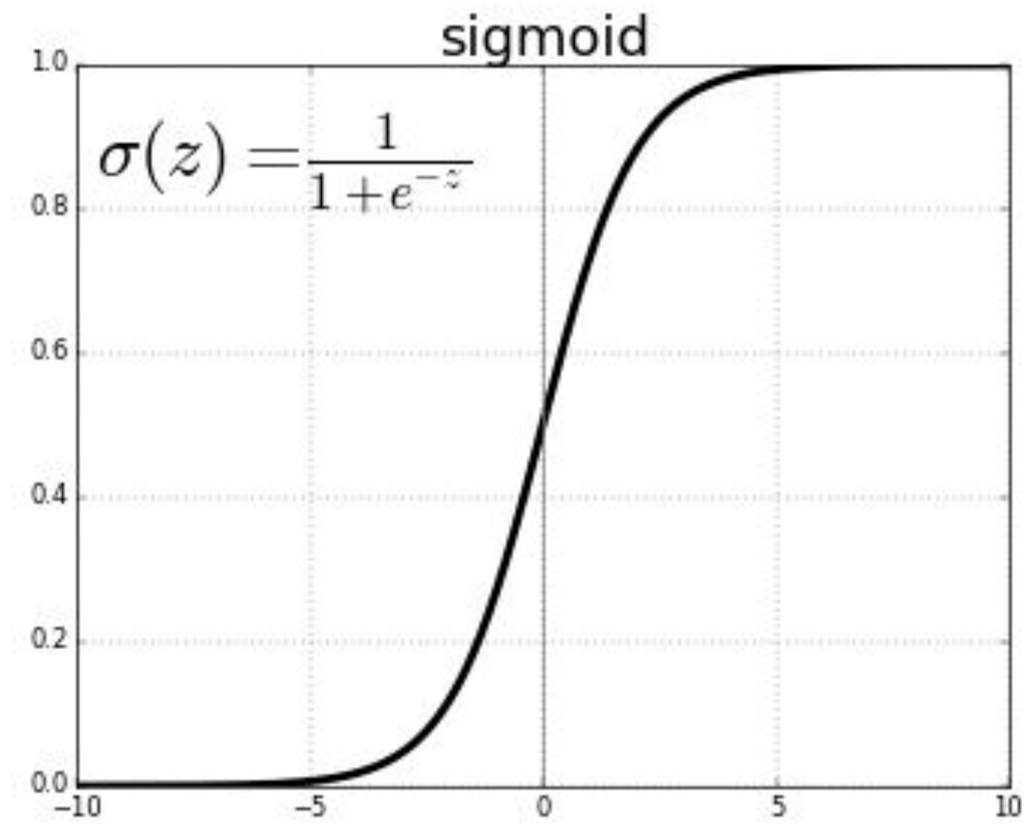
$N_{HIDDEN} = 3$
$N_{INPUTS} = 2$
x_1
x_2
w_{11}
w_{21}
w_{31}
w_{12}
w_{22}
w_{32}
$v_1 = g\left(\sum_{k=1}^{N_{INS}} w_{1k} x_k\right)$
$v_2 = g\left(\sum_{k=1}^{N_{INS}} w_{2k} x_k\right)$
$v_3 = g\left(\sum_{k=1}^{N_{INS}} w_{3k} x_k\right)$
W_1
W_2
W_3
$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$

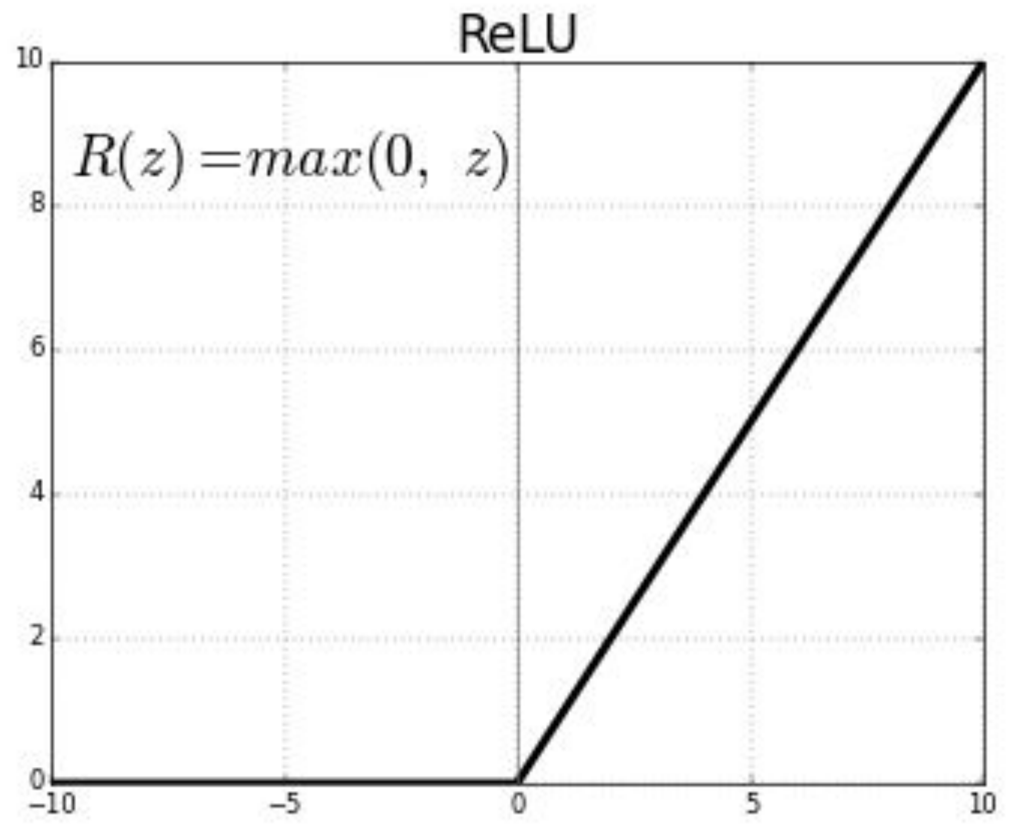
Copyright © 2001, 2003, Andrew W. Moore

Neural Networks (Multi-layer Perceptron)



Activation Function





Sigmoid function

The sigmoid function transforms the output of a linear function into a nonlinear form **ranging between 0 and 1**. It is primarily used to probabilistically represent classification problems like logistic regression.

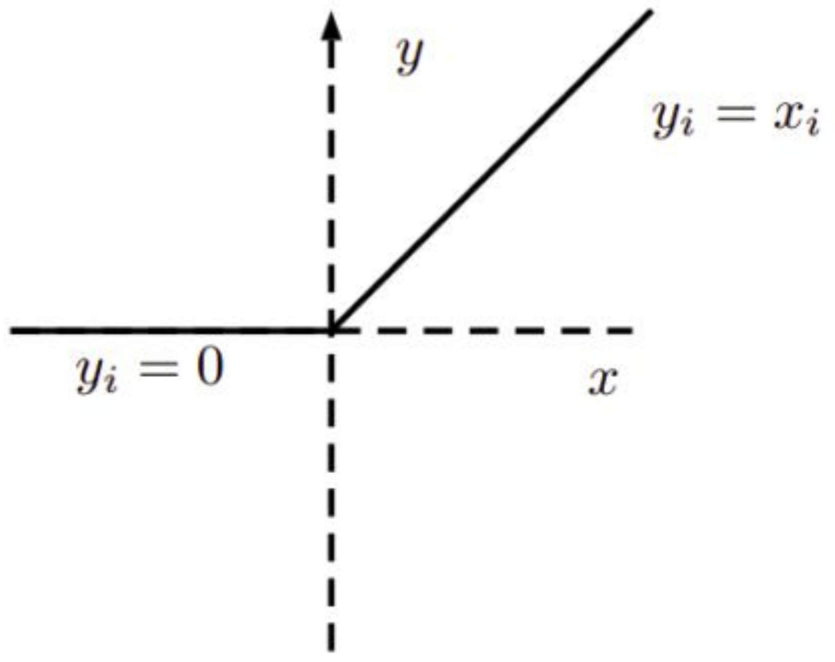
Despite its popularity in the past, it's not frequently used in deep learning models due to the '**vanishing gradient problem**' that arises as the depth of the model increases. We'll delve into the details of the vanishing gradient issue at a later stage.

Relu (rectified linear unit) function

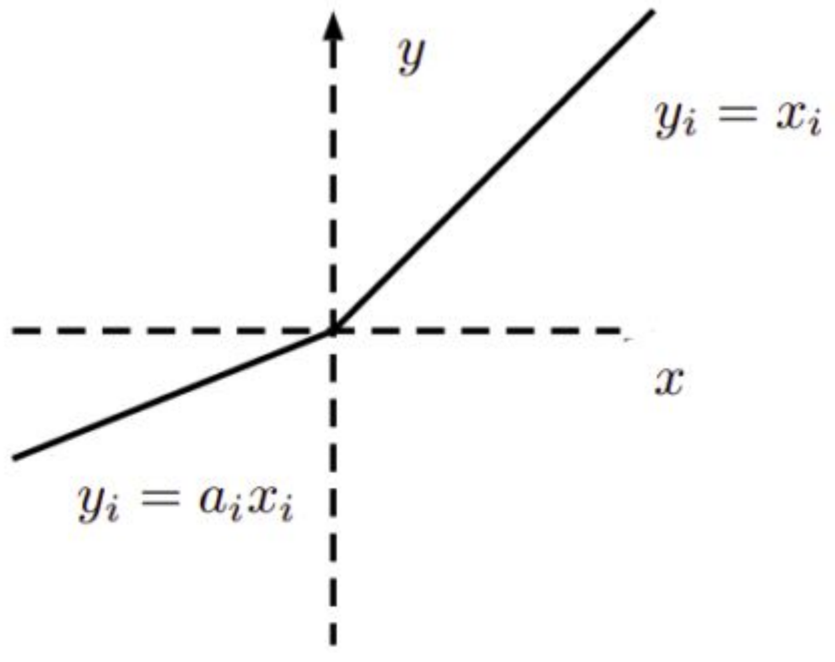
The Rectified Linear Unit (ReLU) function, which is actively employed in recent times, **outputs 0 when the input 'x' is negative, and 'x' when the input is positive**. Its attributes of not impacting gradient descent and subsequently leading to faster learning, along with mitigating the vanishing gradient problem, are notable benefits.

Typically, the ReLU function finds its use in hidden layers. One challenge it presents is that its output is always 0 when receiving negative input values, which could potentially hamper its learning capacity. To address this issue, the Leaky ReLU function is employed.

Leaky ReLU



ReLU



Leaky ReLU

Leaky ReLU

The Leaky ReLU function is a variation of the ReLU activation function. For positive input values, it behaves just like the standard ReLU. However, for negative inputs, instead of producing a zero output, it returns a small, non-zero output, thereby "leaking" a bit of information and keeping the neurons from 'dying'.

Softmax function

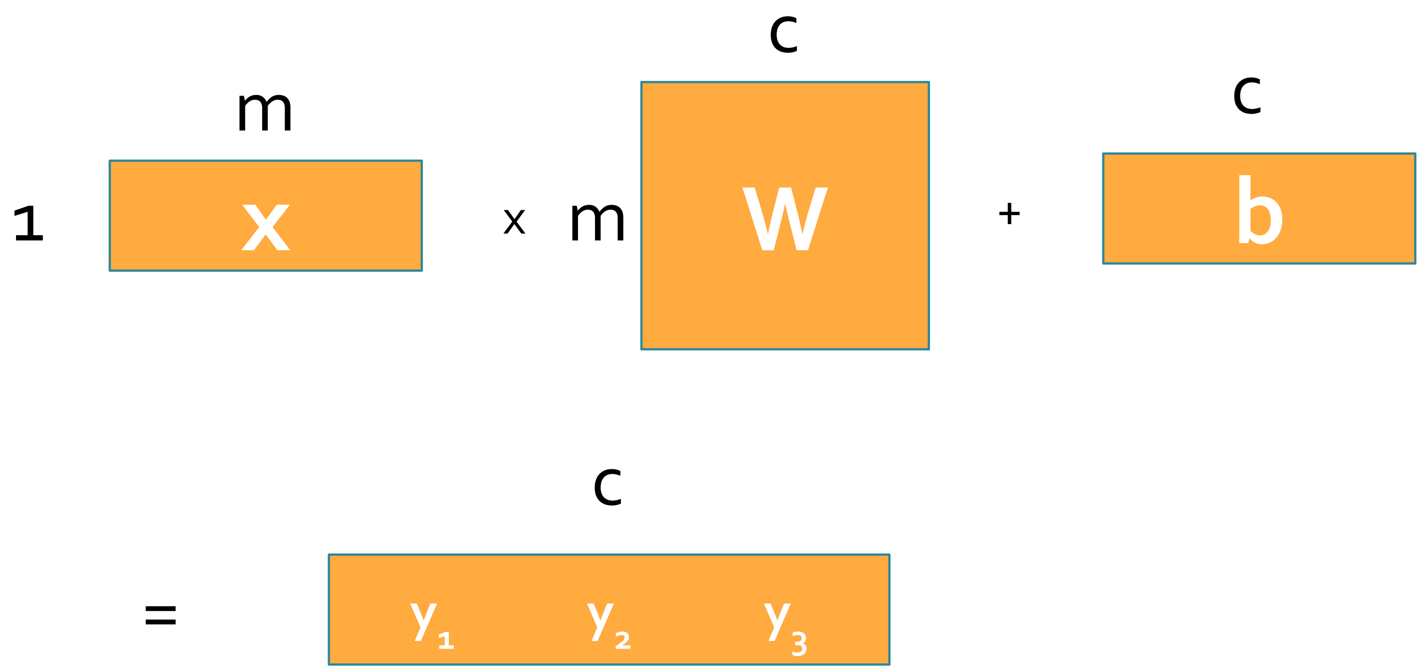
The softmax function normalizes input values so they are outputted within the range of 0 to 1, ensuring that the sum of these outputs always equals 1. This function is **commonly used as the activation function for output nodes in deep learning.** Its mathematical formula is as follows.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ	= softmax
$\vec{z}$	= input vector
e^{z_i}	= standard exponential function for input vector
K	= number of classes in the multi-class classifier
e^{z_j}	= standard exponential function for output vector
e^{z_j}	= standard exponential function for output vector

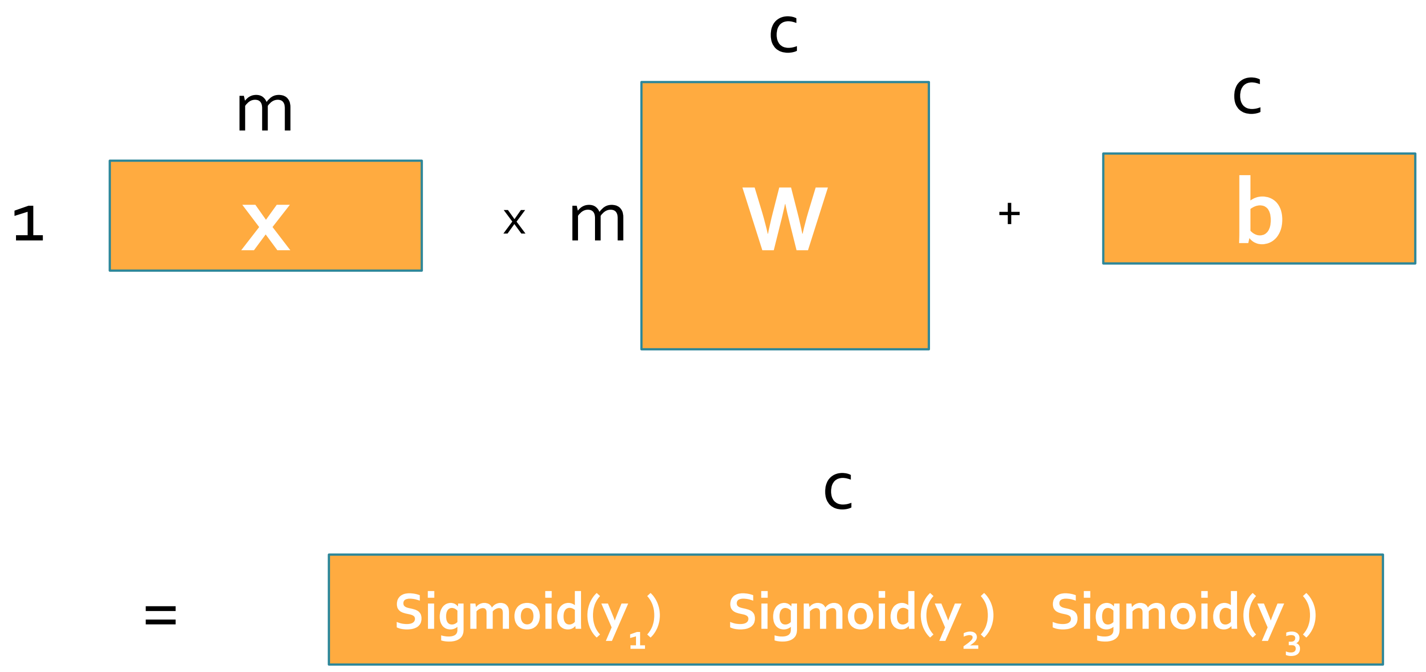
Linear Classifier

- Mutliclass
    - *c* is # of class
    - *m* is # of feature

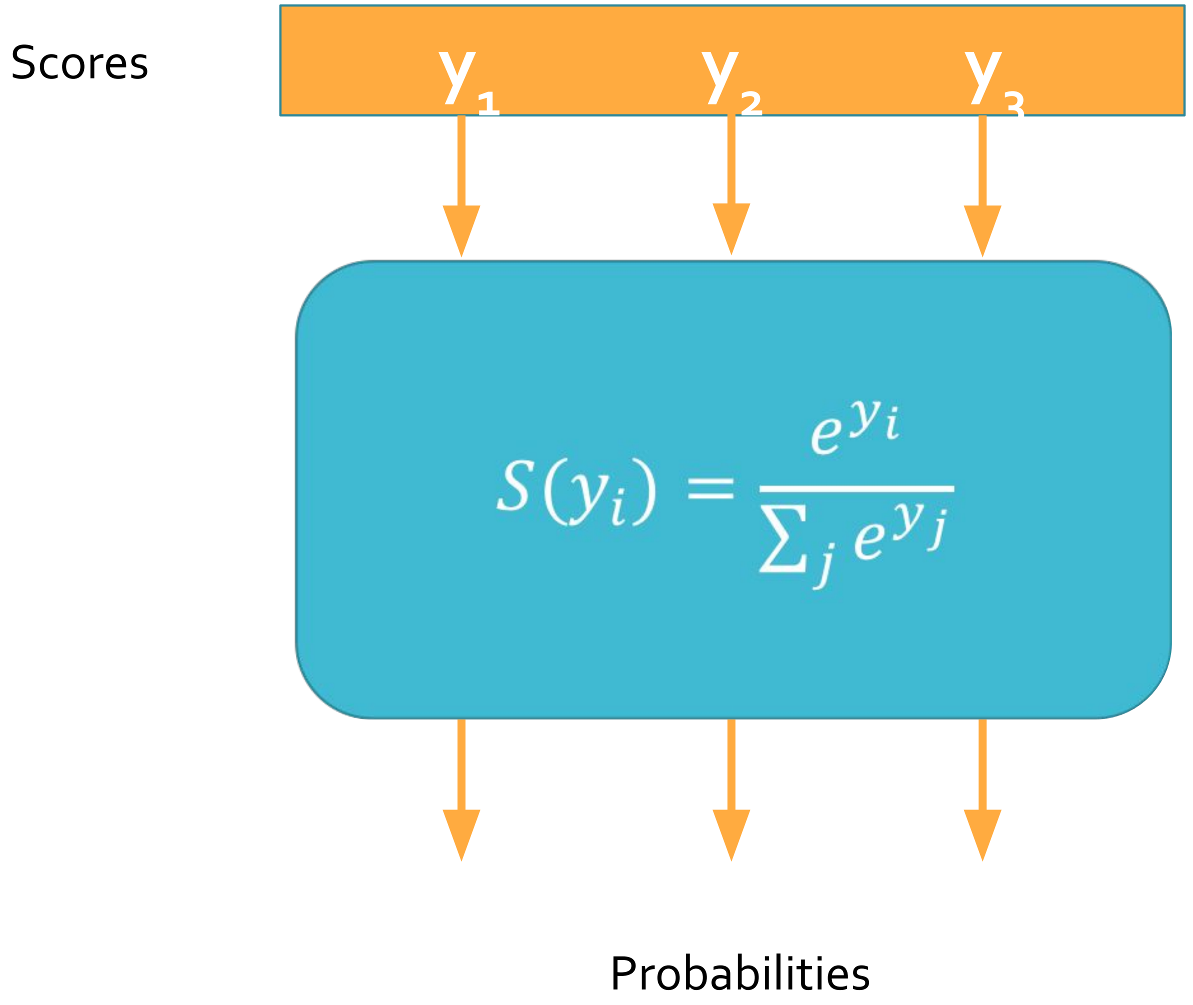


Logistic Regression

- Mutliclass
  - *c* is # of class
  - *m* is # of feature

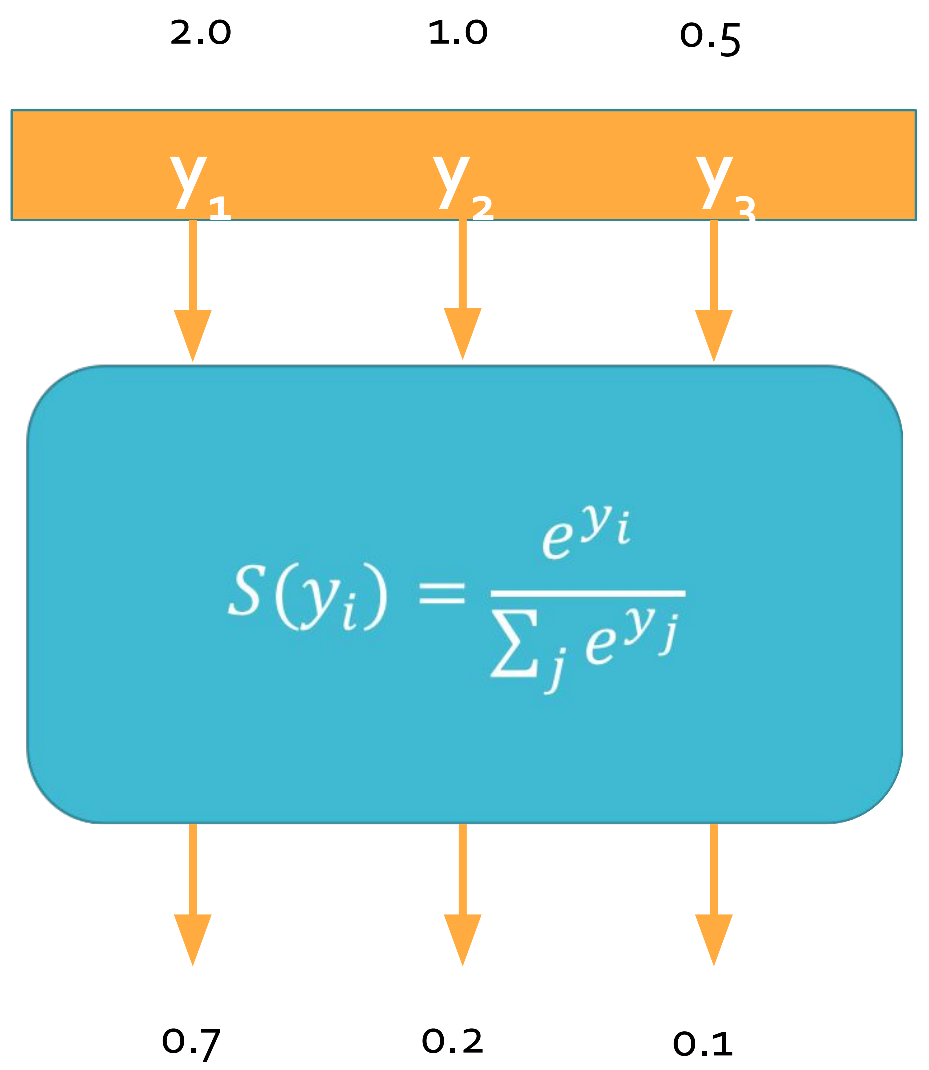


Softmax



$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Softmax



Cost function

Cost(Loss) function

A loss function quantifies the **disparity between the actual data and the predictions** rendered by a model, with the weights of the model fine-tuned through training. A larger value of this loss function indicates a less accurate prediction, while a value close to zero implies a minimal difference between the predicted and actual data.

As previously discussed, the weight updates in the model are guided by the method of *gradient descent*, which leverages the instantaneous gradient of the loss function. The **resultant adjusted weights enable the model to yield more precise predictions**. Consequently, the magnitude of the loss function diminishes with the progression of the learning process.

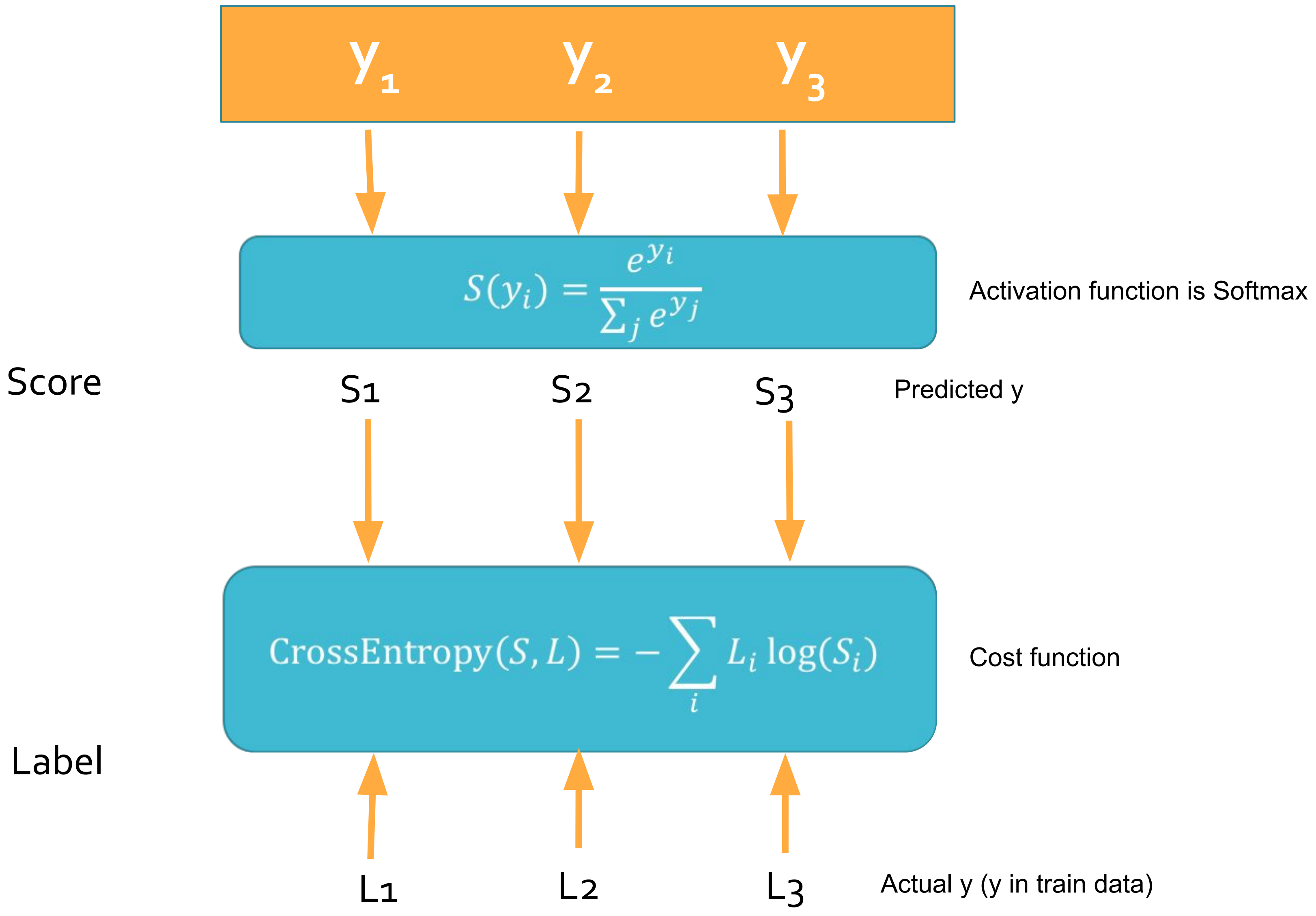
In practice, we utilize two specific loss functions: **Mean Squared Error** (MSE) and **Cross-Entropy**.

Cost function for **Binary class**

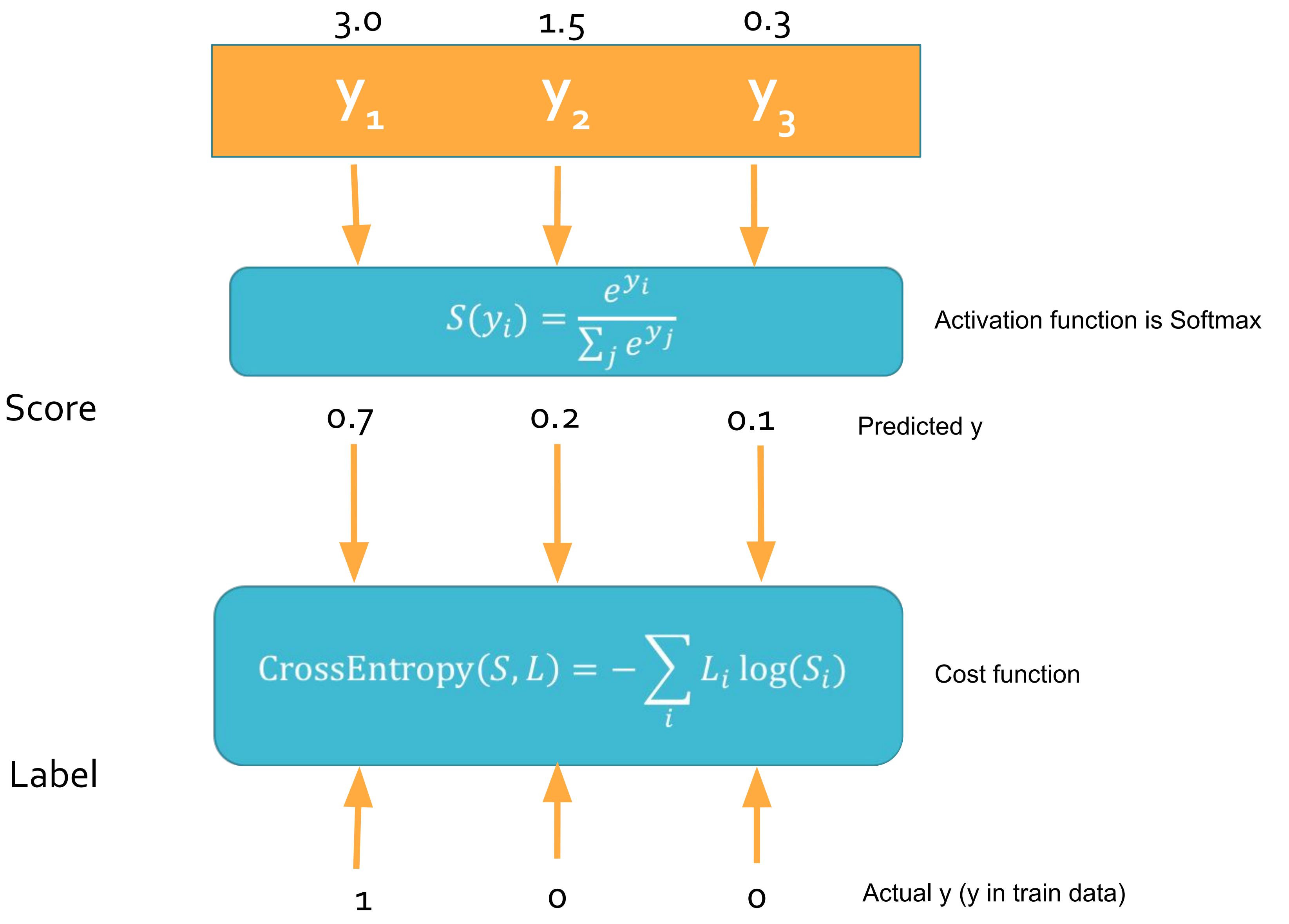
When activation function is sigmoid function,

$$Cost(W,b) = -y\log(\underbrace{H(x)}_{\text{Predicted y}}) \quad -(1-\underbrace{y}_{\text{Actual y (y in train data)}})\log(1-H(x))$$

Cost function for Softmax (**Multiclass**)


y_1 y_2 y_3
$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$
Activation function is Softmax
Score
S1 S2 S3
Predicted y
$\text{CrossEntropy}(S, L) = -\sum_i L_i \log(S_i)$
Cost function
Label
L1 L2 L3
Actual y (y in train data)

Cost function for Softmax(**Multiclass**)



3.0
1.5
0.3
y_1
y_2
y_3
$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$
Activation function is Softmax
Score
0.7
0.2
0.1
Predicted y
$\text{CrossEntropy}(S, L) = -\sum_i L_i \log(S_i)$
Cost function
Label
1
0
0
Actual y (y in train data)

Training - Feedforward and Backpropagation

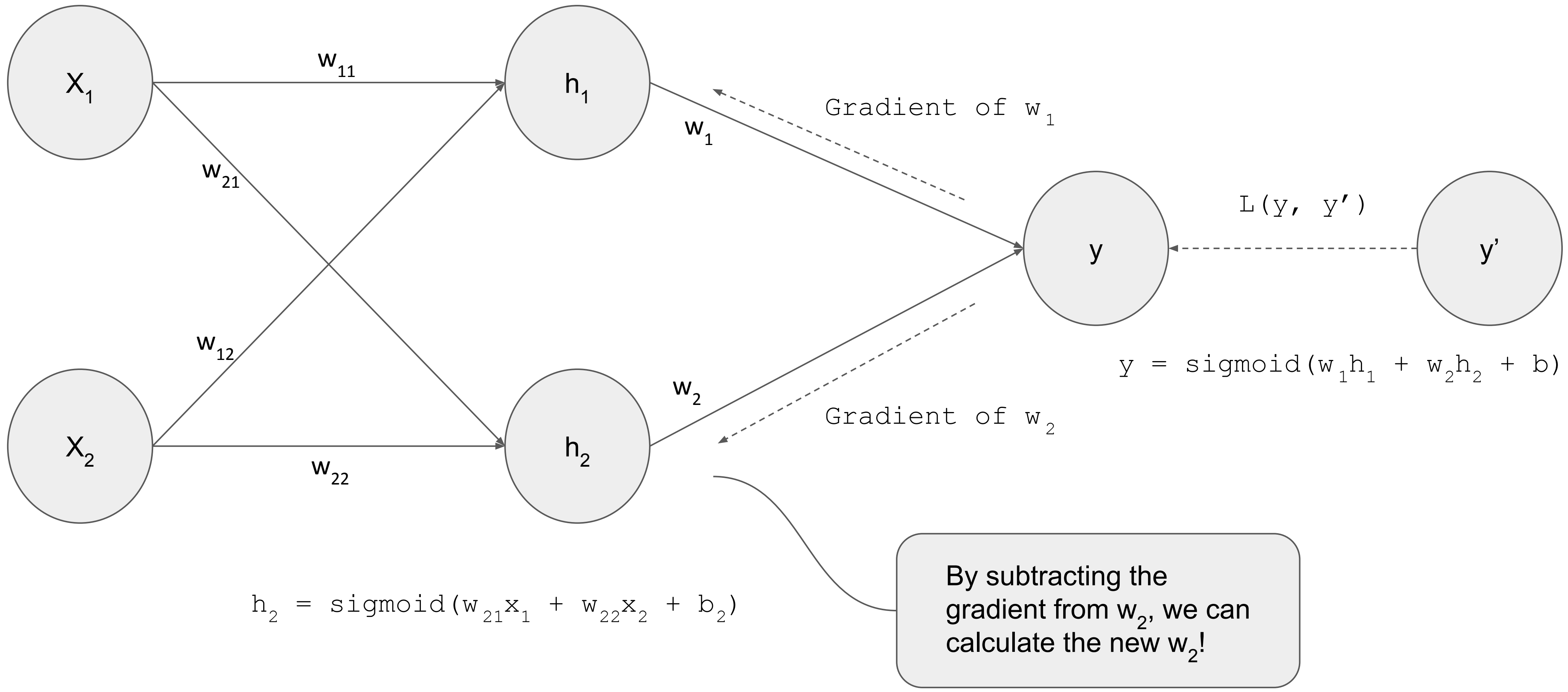
The process of training in deep learning can be broadly divided into two steps: feedforward and backpropagation.

During the **feedforward** step, the training data is input into the network, and it traverses through the entire architecture to compute the predictions based on the given data. Specifically, each neuron applies transformations (weighted sums and activation functions) to the information received from the previous layer's neurons and sends this output to the neurons in the next layer. Once the values have been propagated through all the layers, the output value is calculated at the final output layer.

Training - Feedforward and Backpropagation

The **backpropagation** step follows feedforward. Here, the difference (or error) between the computed output value (the predicted value) and the actual target value, y, is calculated through a loss function. The goal is to minimize this error. To achieve this, the weights (w) in the network are updated using an optimization algorithm, such as gradient descent. The changes made to the weights in the output layer propagate back to the previous hidden layers, with the weights in each hidden layer being adjusted in turn. This error propagation continues until the input layer, leading to an adjustment of the weight values across all layers in the network.

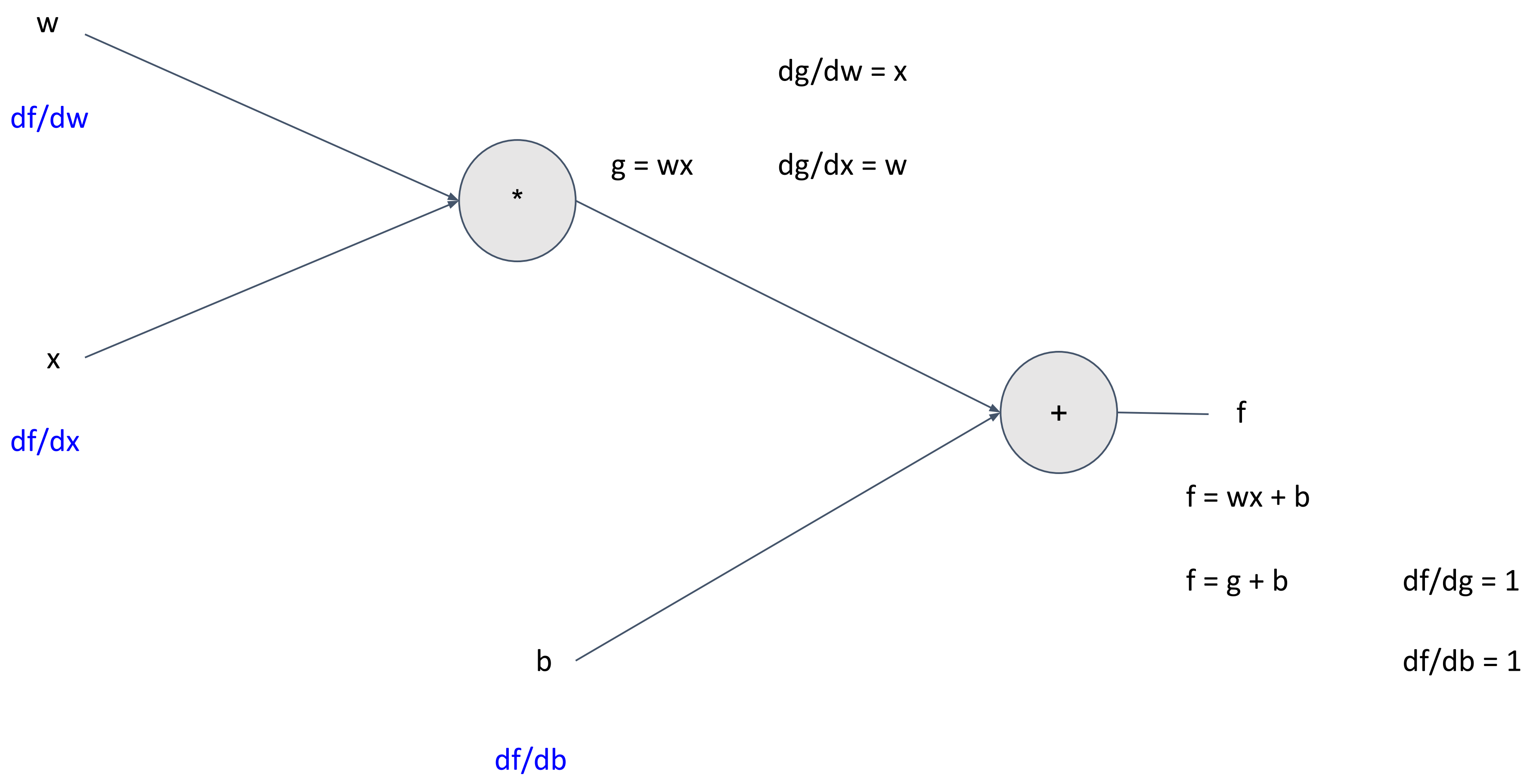
$$h_1 = \mathrm{sigmoid}(w_{11}x_1 + w_{12}x_2 + b_1)$$

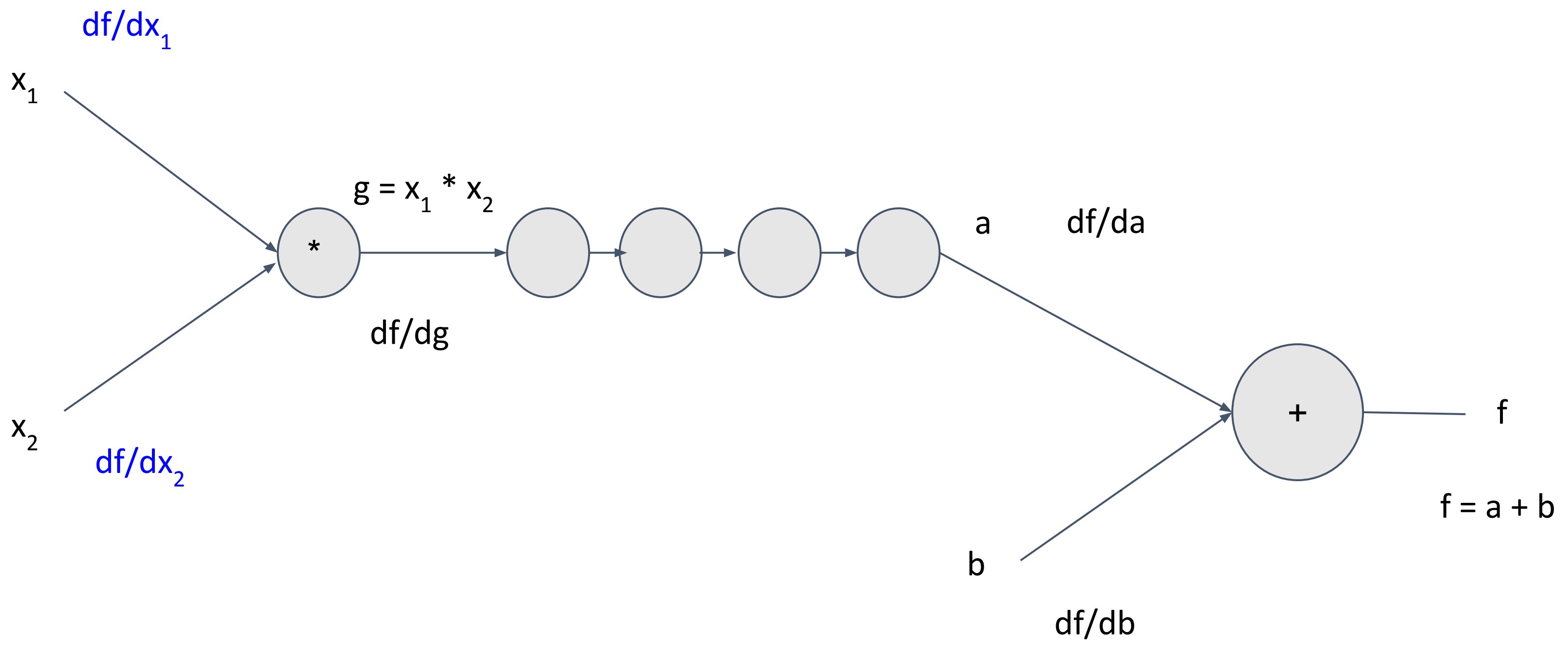


By subtracting the gradient from w_2, we can calculate the new w_2!

$$y = \mathrm{sigmoid}(w_1h_1 + w_2h_2 + b)$$

$$h_2 = \mathrm{sigmoid}(w_{21}x_1 + w_{22}x_2 + b_2)$$

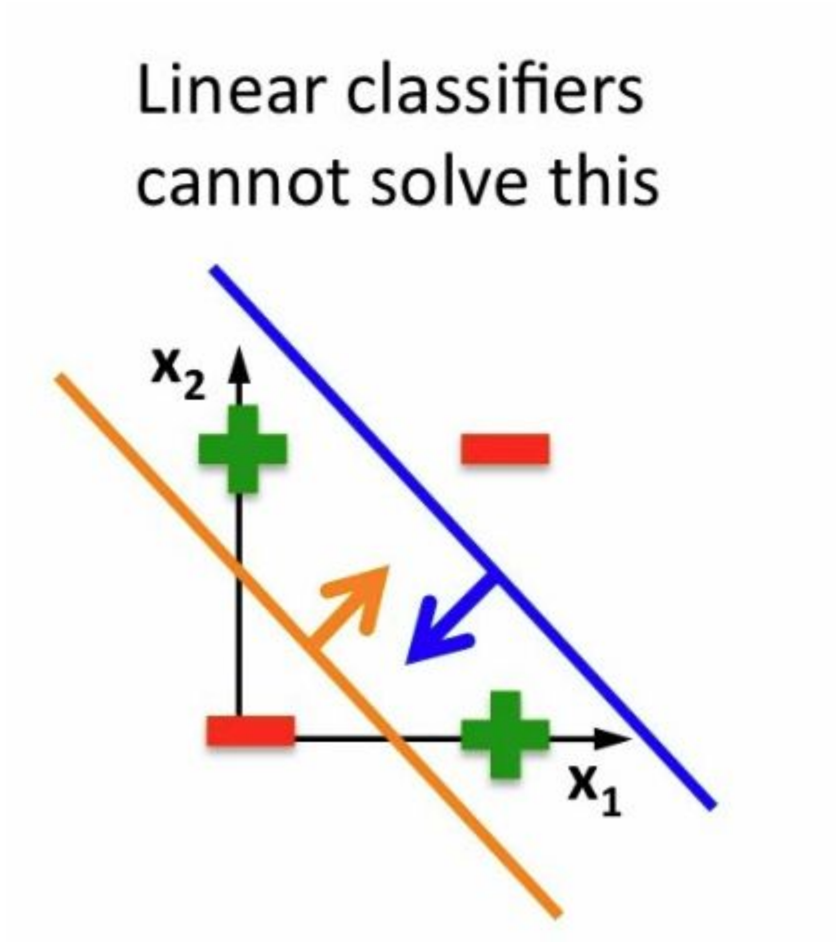

w
df/dw
x
df/dx
*
g = wx
dg/dw = x
dg/dx = w
+
f
f = wx + b
f = g + b
df/dg = 1
df/db = 1
b
df/db


df/dx$_1$
x$_1$
g = x$_1$ * x$_2$
*
a
df/da
df/dg
+
f
x$_2$
df/dx$_2$
f = a + b
b
df/db

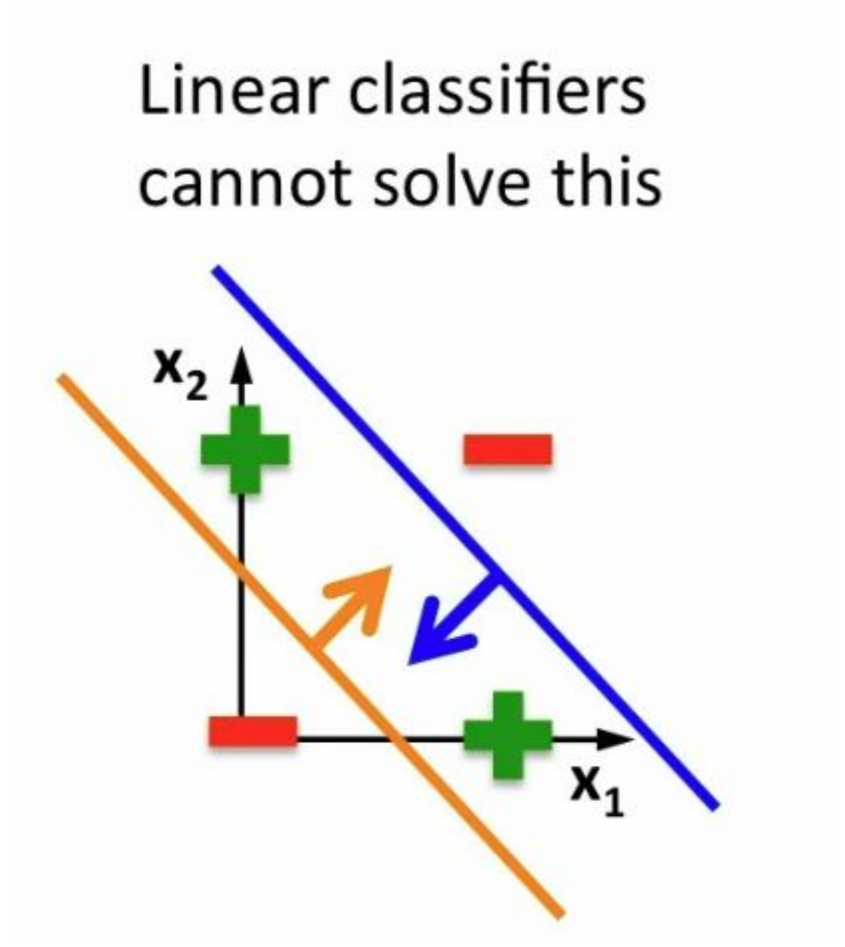
Why Neural Net?

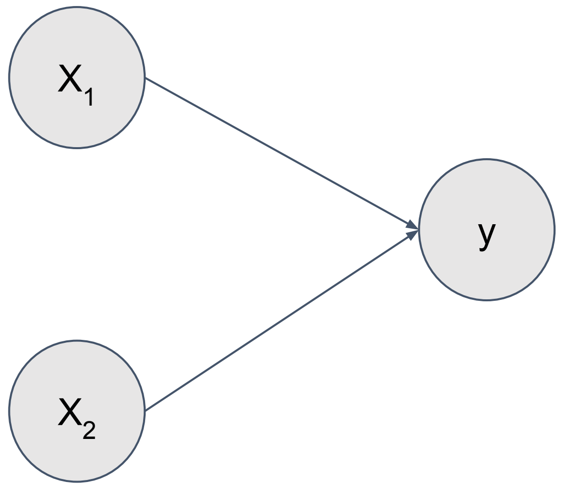
With a hidden layer **vs** Without a hidden layer

XOR problem



Logistic Regression (Non-linear classifier)

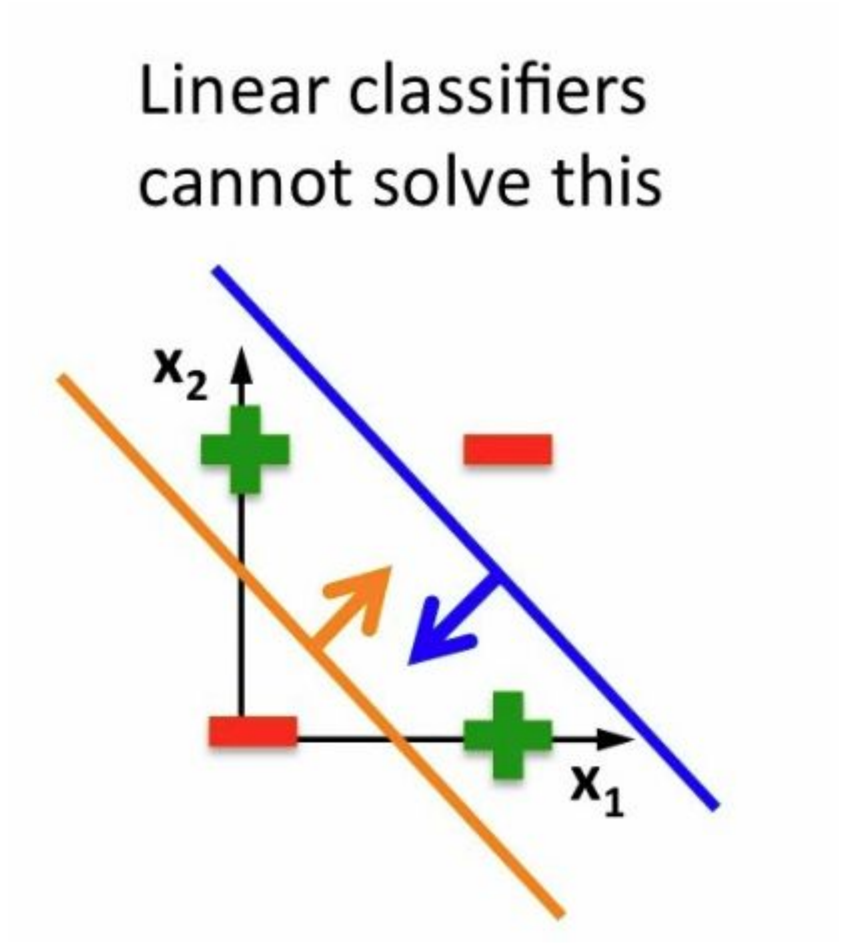


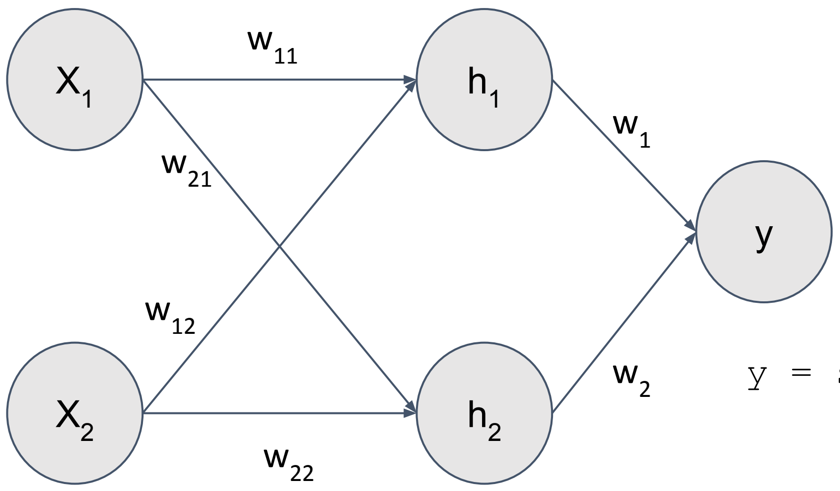


$y = \text{sigmoid}(w_1x_1 + w_2x_2 + b)$

Equivalent to Neural net **without any hidden layers**

Neural Net (Non-linear classifier)





$h_1 = \text{sigmoid}(w_{11}x_1 + w_{12}x_2 + b_1)$

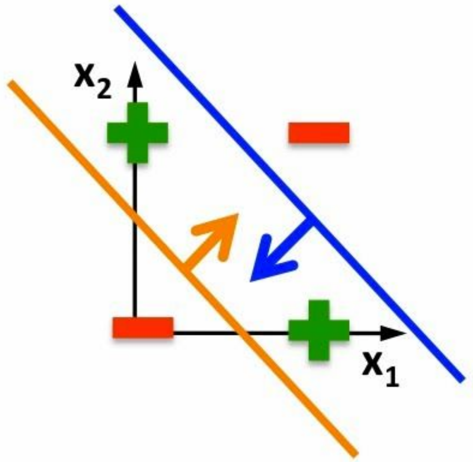
$y = \text{sigmoid}(w_1h_1 + w_2h_2 + b)$

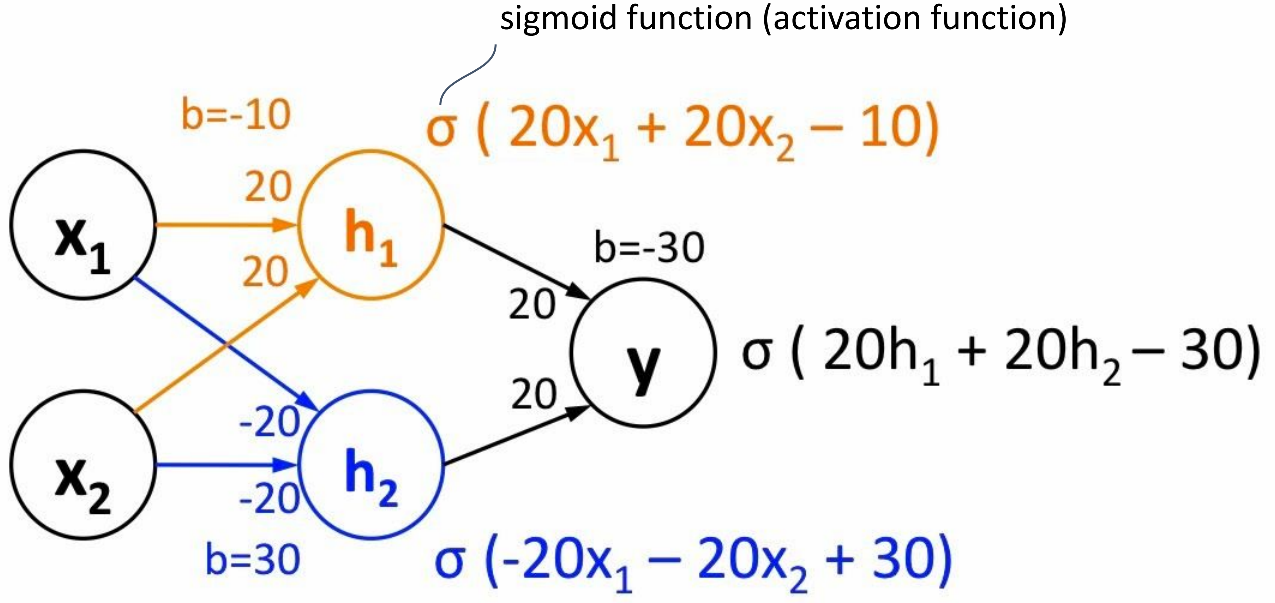
$h_2 = \text{sigmoid}(w_{21}x_1 + w_{22}x_2 + b_2)$

Neural net with a hidden layer

Solving XOR with a Neural Net

Linear classifiers cannot solve this



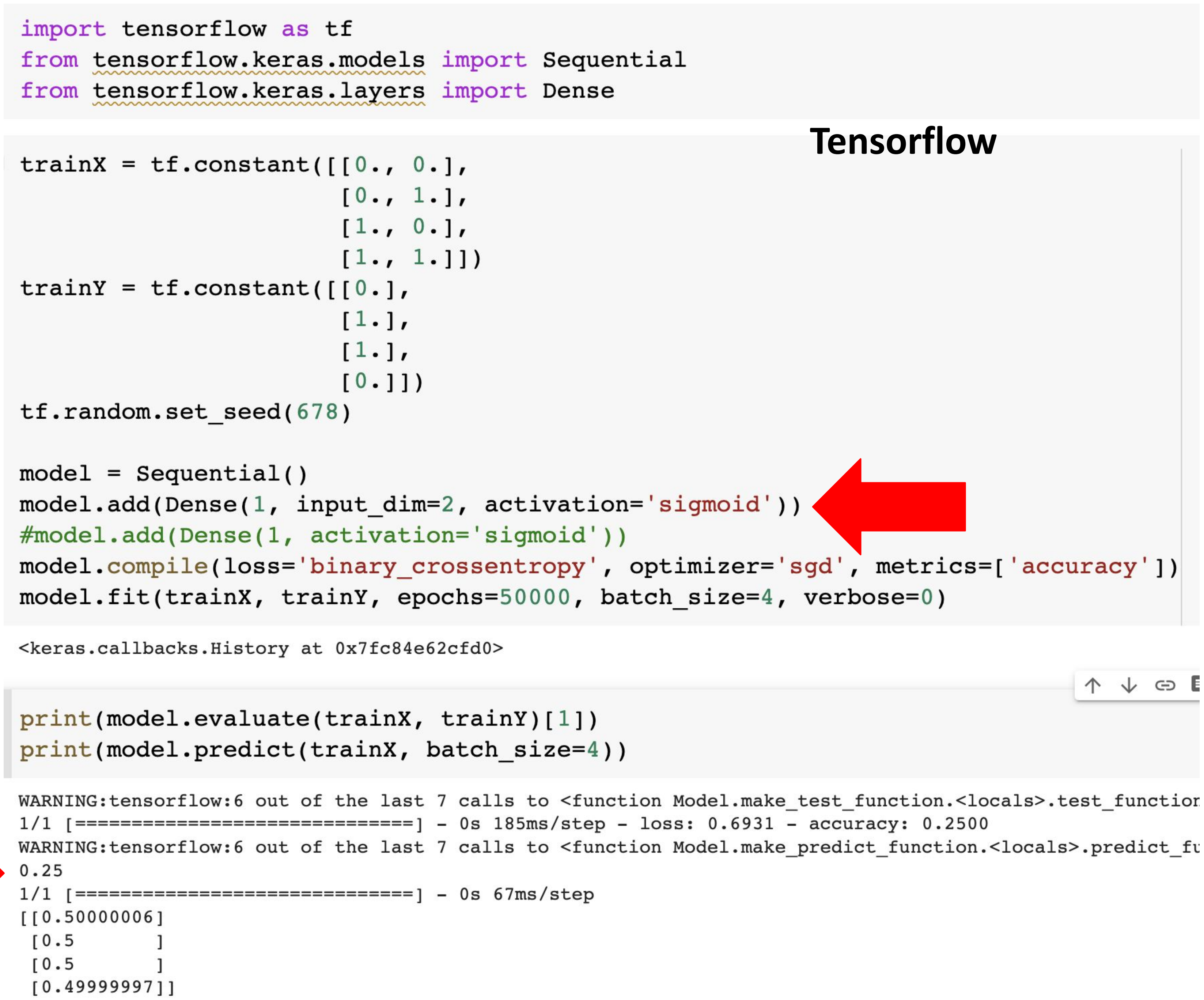


sigmoid function (activation function)
b=-10
$\sigma\,(20x_1 + 20x_2 - 10)$
x_1
x_2
20
20
h_1
-20
-20
h_2
b=30
$\sigma\,(-20x_1 - 20x_2 + 30)$
b=-30
20
20
y
$\sigma\,(20h_1 + 20h_2 - 30)$

$\sigma(20*0 + 20*0 - 10) \approx 0$	$\sigma\,(-20*0 - 20*0 + 30) \approx 1$	$\sigma\,(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$	$\sigma\,(-20*1 - 20*1 + 30) \approx 0$	$\sigma\,(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*0 + 20*1 - 10) \approx 1$	$\sigma\,(-20*0 - 20*1 + 30) \approx 1$	$\sigma\,(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$	$\sigma\,(-20*1 - 20*0 + 30) \approx 1$	$\sigma\,(20*1 + 20*1 - 30) \approx 1$

Copyright © 2014 Victor Lavrenko

Tensorflow



```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

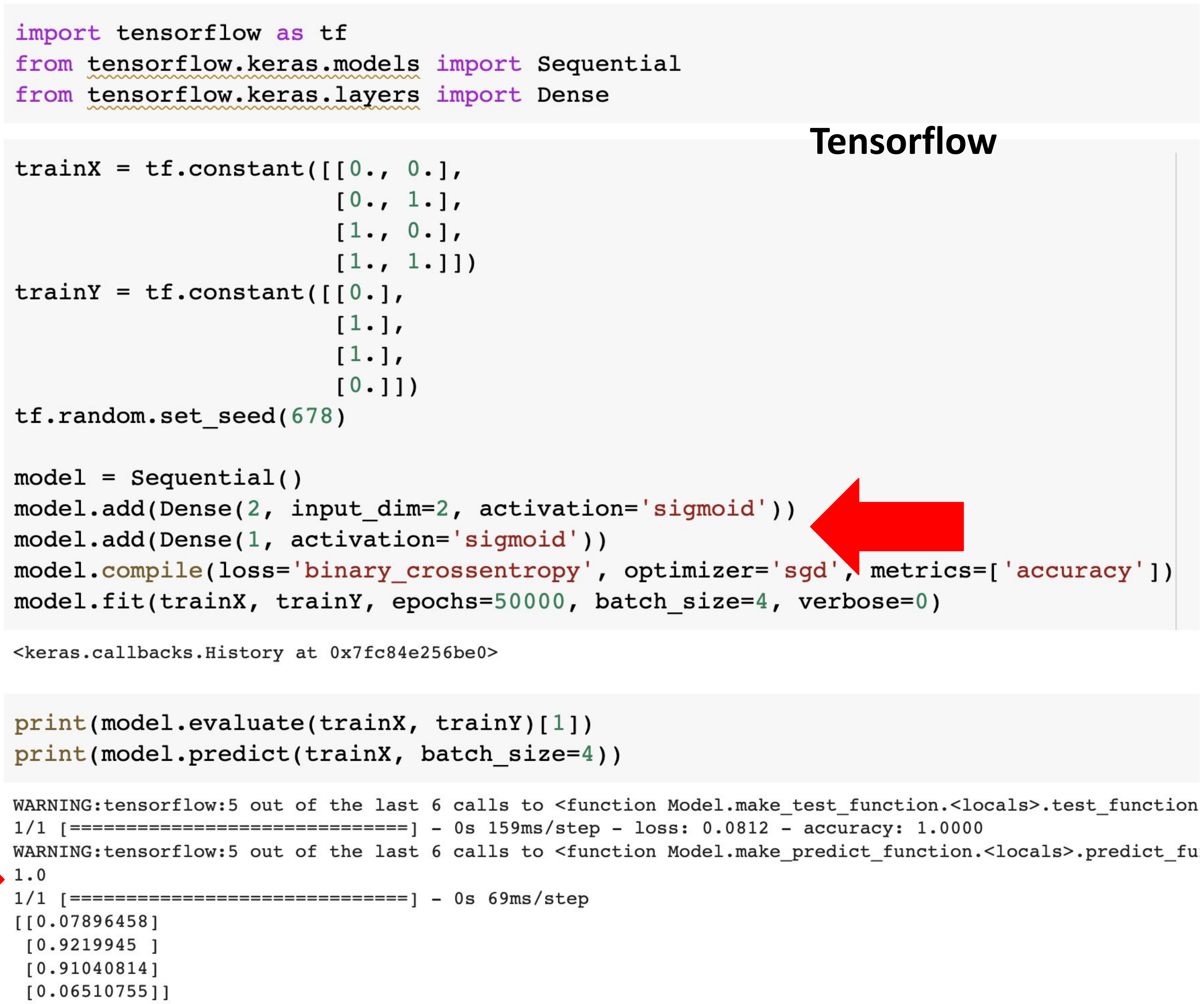
```python
trainX = tf.constant([[0., 0.],
                      [0., 1.],
                      [1., 0.],
                      [1., 1.]])
trainY = tf.constant([[0.],
                      [1.],
                      [1.],
                      [0.]])
tf.random.set_seed(678)

model = Sequential()
model.add(Dense(1, input_dim=2, activation='sigmoid'))
#model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
model.fit(trainX, trainY, epochs=50000, batch_size=4, verbose=0)
```

```
<keras.callbacks.History at 0x7fc84e62cfd0>
```

```python
print(model.evaluate(trainX, trainY)[1])
print(model.predict(trainX, batch_size=4))
```

```
WARNING:tensorflow:6 out of the last 7 calls to <function Model.make_test_function.<locals>.test_functior
1/1 [==============================] - 0s 185ms/step - loss: 0.6931 - accuracy: 0.2500
WARNING:tensorflow:6 out of the last 7 calls to <function Model.make_predict_function.<locals>.predict_fu
0.25
1/1 [==============================] - 0s 67ms/step
[[0.50000006]
 [0.5       ]
 [0.5       ]
 [0.49999997]]
```

Tensorflow



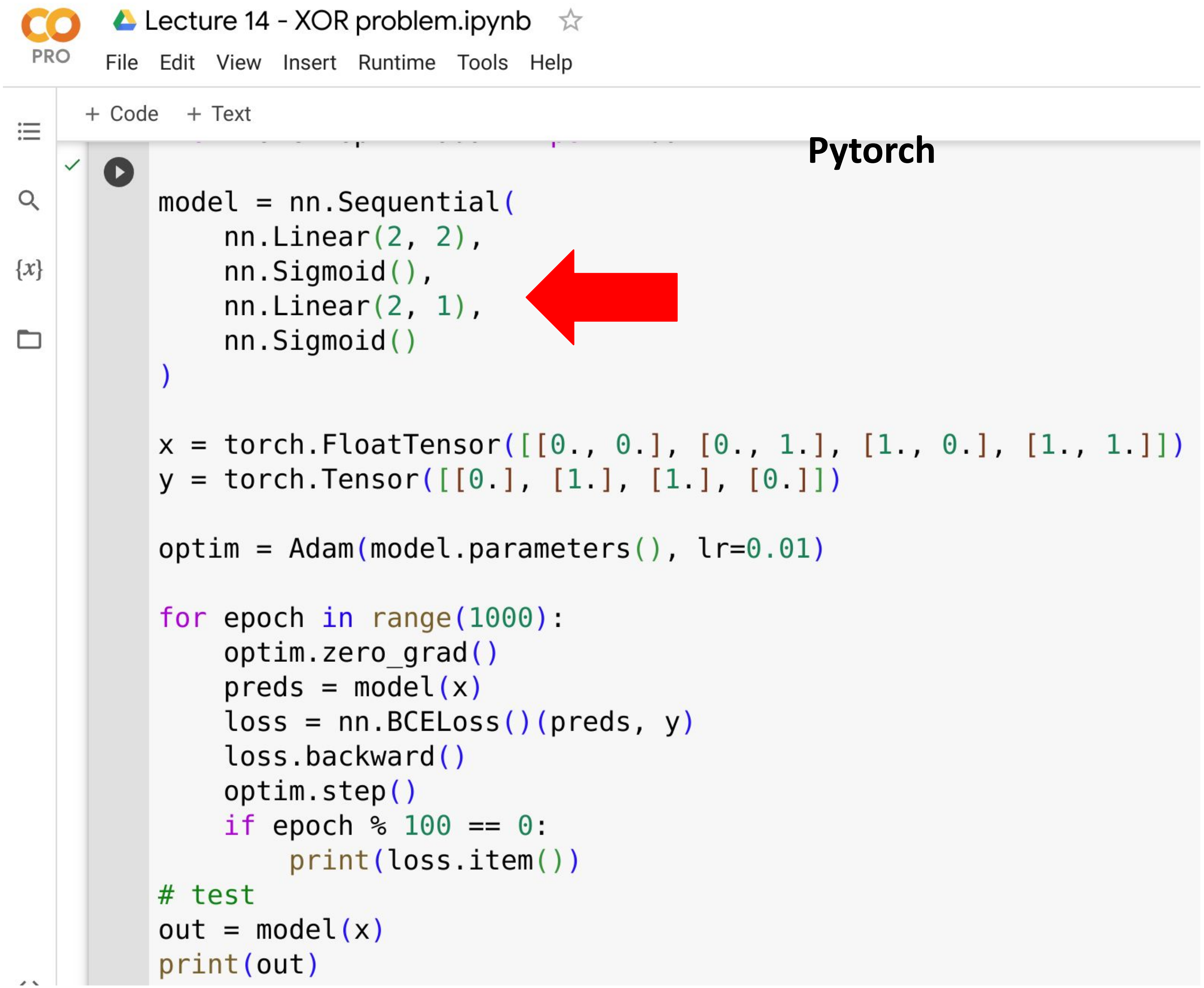
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
trainX = tf.constant([[0., 0.],
                      [0., 1.],
                      [1., 0.],
                      [1., 1.]])
trainY = tf.constant([[0.],
                      [1.],
                      [1.],
                      [0.]])
tf.random.set_seed(678)

model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
model.fit(trainX, trainY, epochs=50000, batch_size=4, verbose=0)
```

```
<keras.callbacks.History at 0x7fc84e256be0>
```

```
print(model.evaluate(trainX, trainY)[1])
print(model.predict(trainX, batch_size=4))
```

```
WARNING:tensorflow:5 out of the last 6 calls to <function Model.make_test_function.<locals>.test_function
1/1 [==============================] - 0s 159ms/step - loss: 0.0812 - accuracy: 1.0000
WARNING:tensorflow:5 out of the last 6 calls to <function Model.make_predict_function.<locals>.predict_fu
1.0
1/1 [==============================] - 0s 69ms/step
[[0.07896458]
 [0.9219945 ]
 [0.91040814]
 [0.06510755]]
```

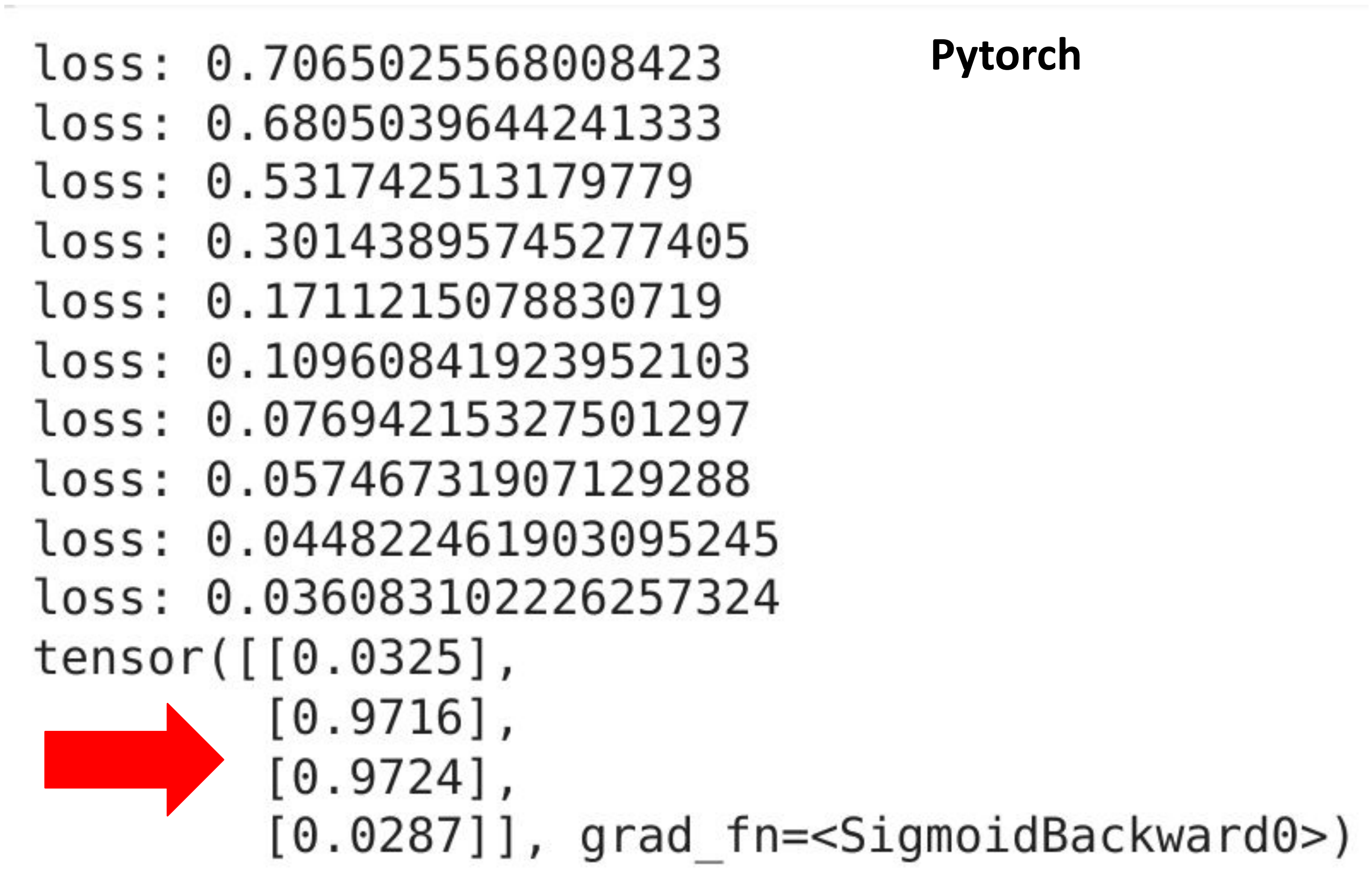


Lecture 14 - XOR problem.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text

Pytorch

```python
model = nn.Sequential(
    nn.Linear(2, 2),
    nn.Sigmoid(),
    nn.Linear(2, 1),
    nn.Sigmoid()
)

x = torch.FloatTensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
y = torch.Tensor([[0.], [1.], [1.], [0.]])

optim = Adam(model.parameters(), lr=0.01)

for epoch in range(1000):
    optim.zero_grad()
    preds = model(x)
    loss = nn.BCELoss()(preds, y)
    loss.backward()
    optim.step()
    if epoch % 100 == 0:
        print(loss.item())
# test
out = model(x)
print(out)
```

Pytorch



```
loss: 0.7065025568008423
loss: 0.680503964241333
loss: 0.531742513179779
loss: 0.3014389574527405
loss: 0.1711215078830719
loss: 0.1096084192395210З
loss: 0.07694215327501297
loss: 0.05746731907129288
loss: 0.044822461903095245
loss: 0.036083102226257324
tensor([[0.0325],
        [0.9716],
        [0.9724],
        [0.0287]], grad_fn=<SigmoidBackward0>)
```