Machine Learning and Transportation - Neural Networks & Deep Neural Networks

Peter Chen

Shanghai University of Engineering Science

December 7-15, 2019

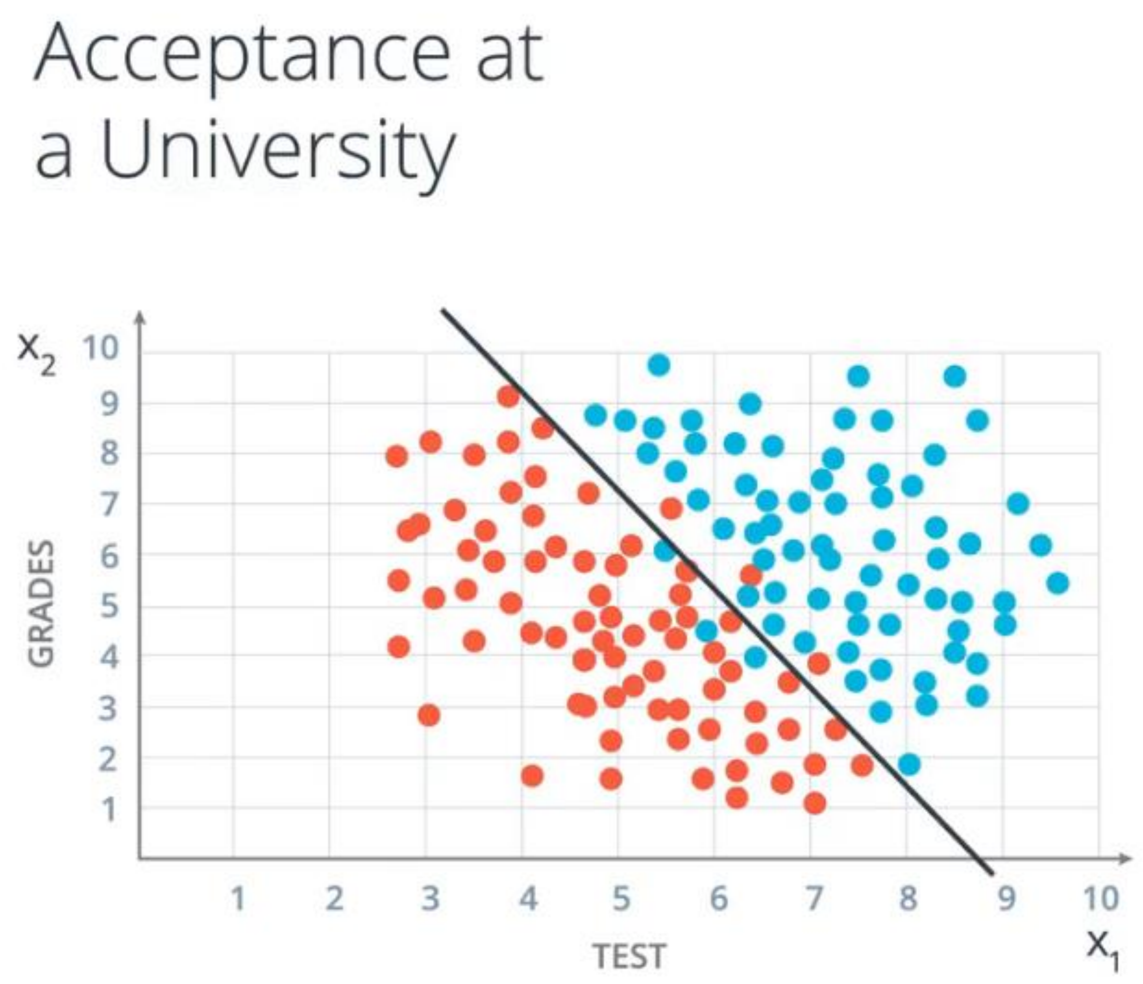
Neural Networks

The following lessons contain introductory and intermediate material on neural networks, building a neural network from scratch, using TensorFlow, and Convolutional Neural Networks:

- **Neural Networks**
- **TensorFlow**
- **Deep Neural Networks**
- **Convolutional Neural Networks**

Neural Networks

Linear Boundaries



BOUNDARY:

A LINE

$w_1x_1 + w_2x_2 + b = 0$

$Wx + b = 0$

$W = (w_1, w_2)$

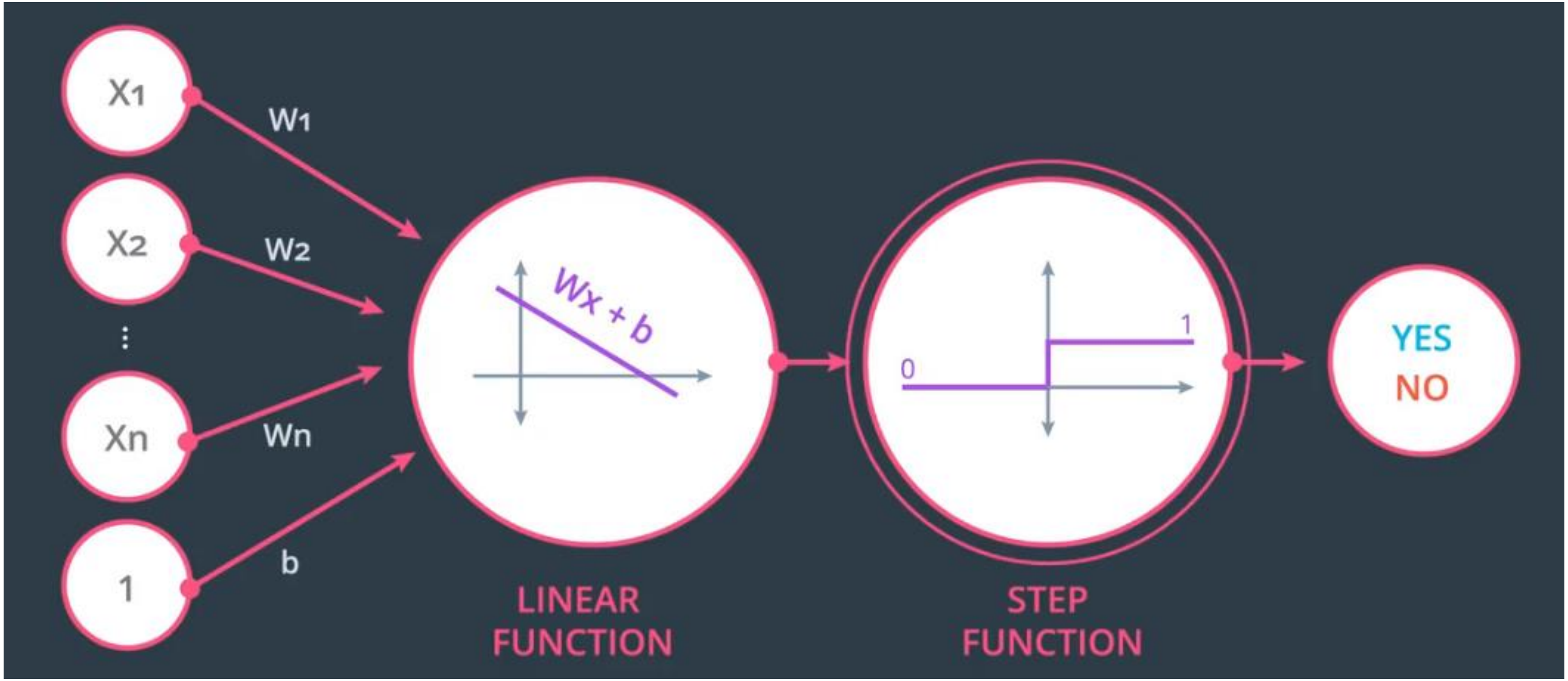
$x = (x_1, x_2)$

y = label: 0 or 1

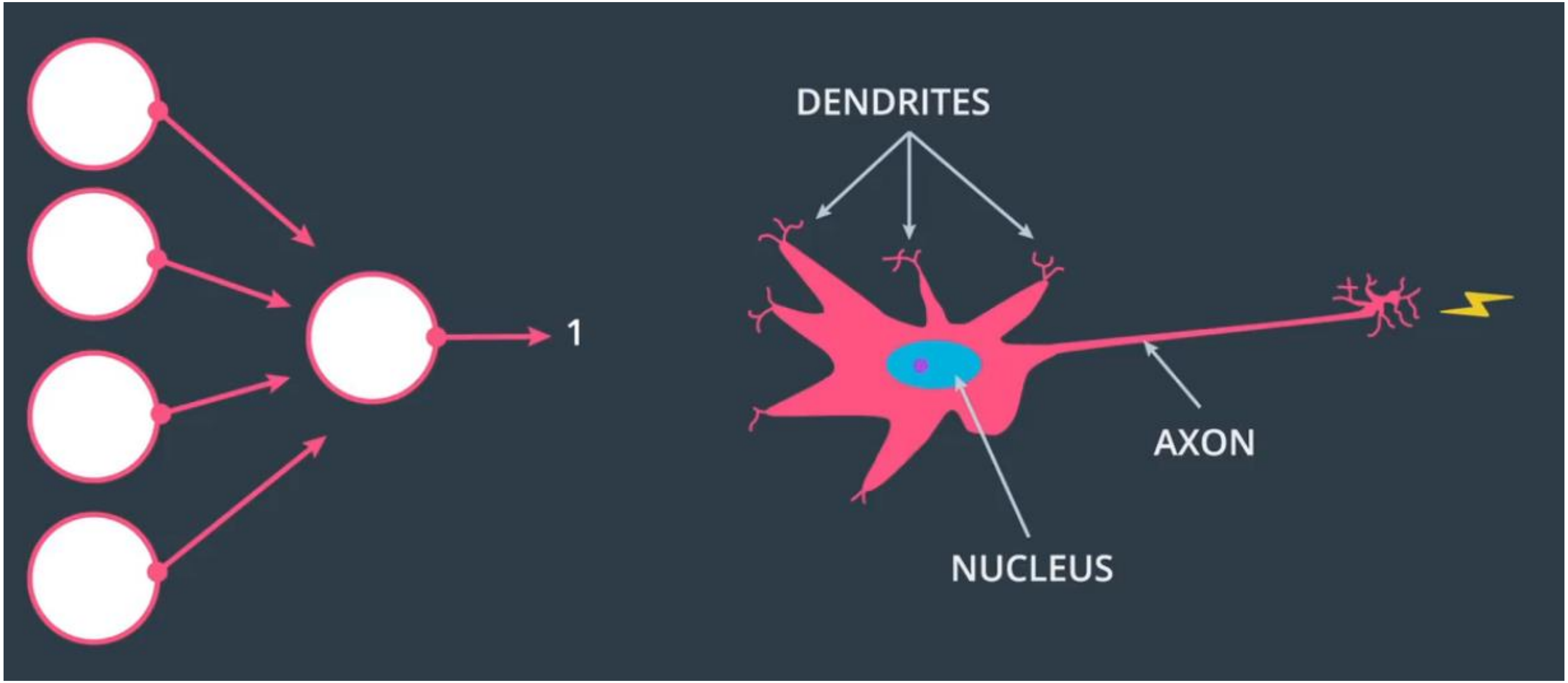
PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

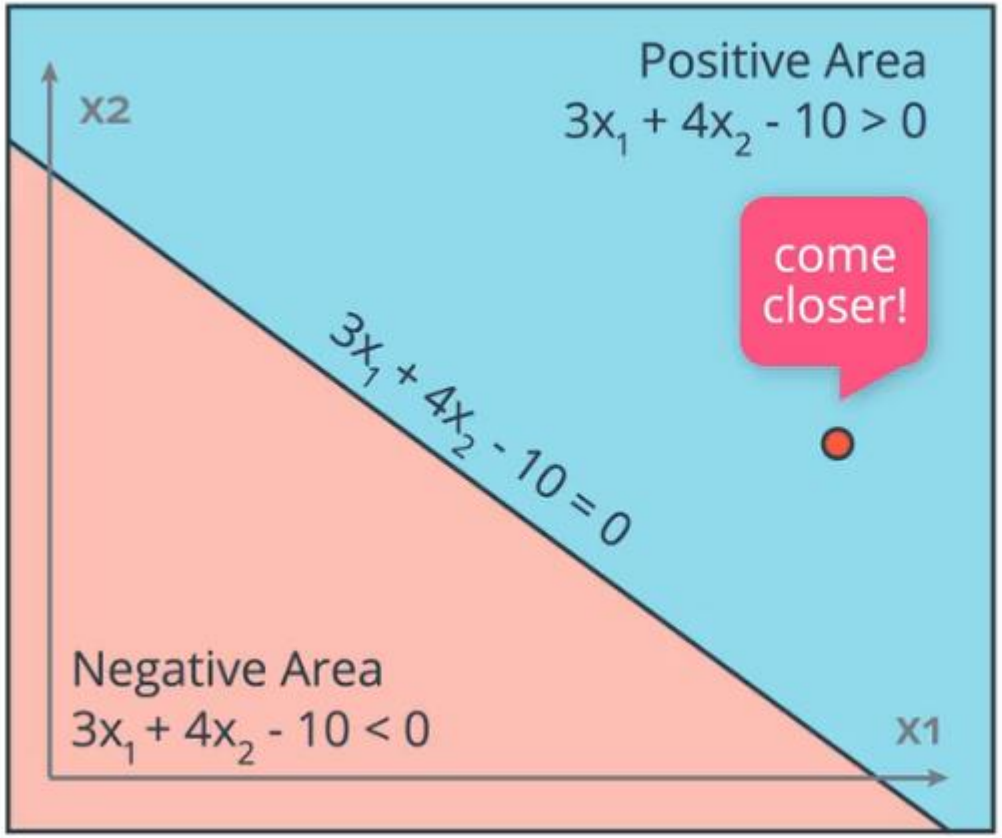
Perceptron


X1
X2
Xn
1
W1
W2
Wn
b
Wx + b
0
1
YES
NO
LINEAR FUNCTION
STEP FUNCTION

Perceptron



Learning Rate



LINE: $3x_1 + 4x_2 - 10 = 0$

POINT: (4,5)

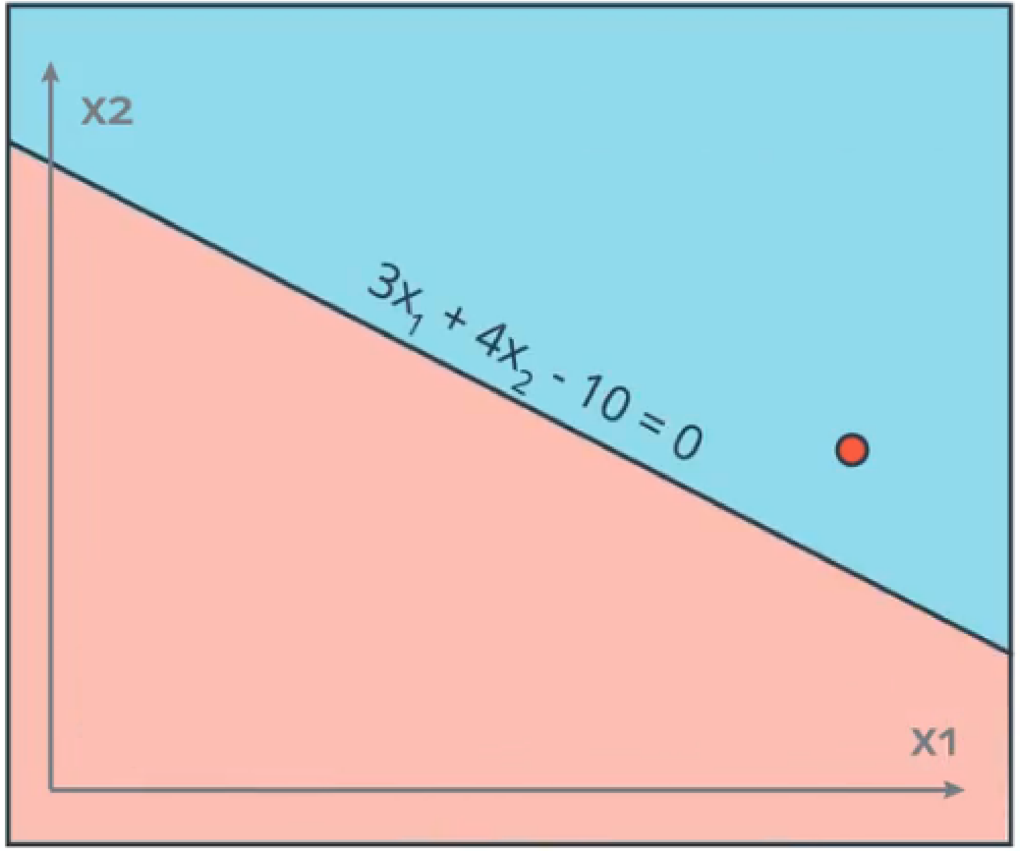
LEARNING RATE: 0.1

	3	4	-10
−	0.4	0.5	0.1
	2.6	3.5	-10.1

NEW LINE

$2.6x_1 + 3.5x_2 - 10.1 = 0$

Learning Rate



LINE: $3x_1 + 4x_2 - 10 = 0$

POINT: (4,5)

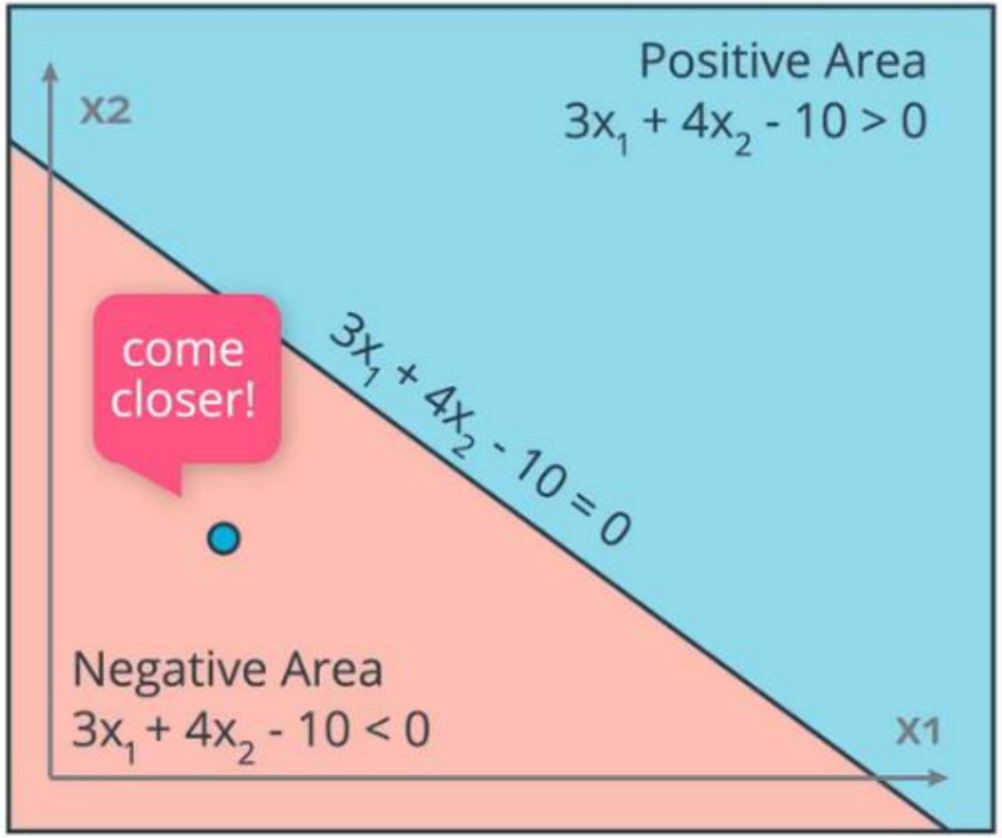
LEARNING RATE: 0.1

$$
\begin{array}{rrrr}
 & 3 & 4 & -10 \\
- & 0.4 & 0.5 & 0.1 \\
\hline
 & 2.6 & 3.5 & -10.1
\end{array}
$$

NEW LINE

$2.6x_1 + 3.5x_2 - 10.1 = 0$

Learning Rate



LINE: $3x_1 + 4x_2 - 10 = 0$

POINT: (1,1)

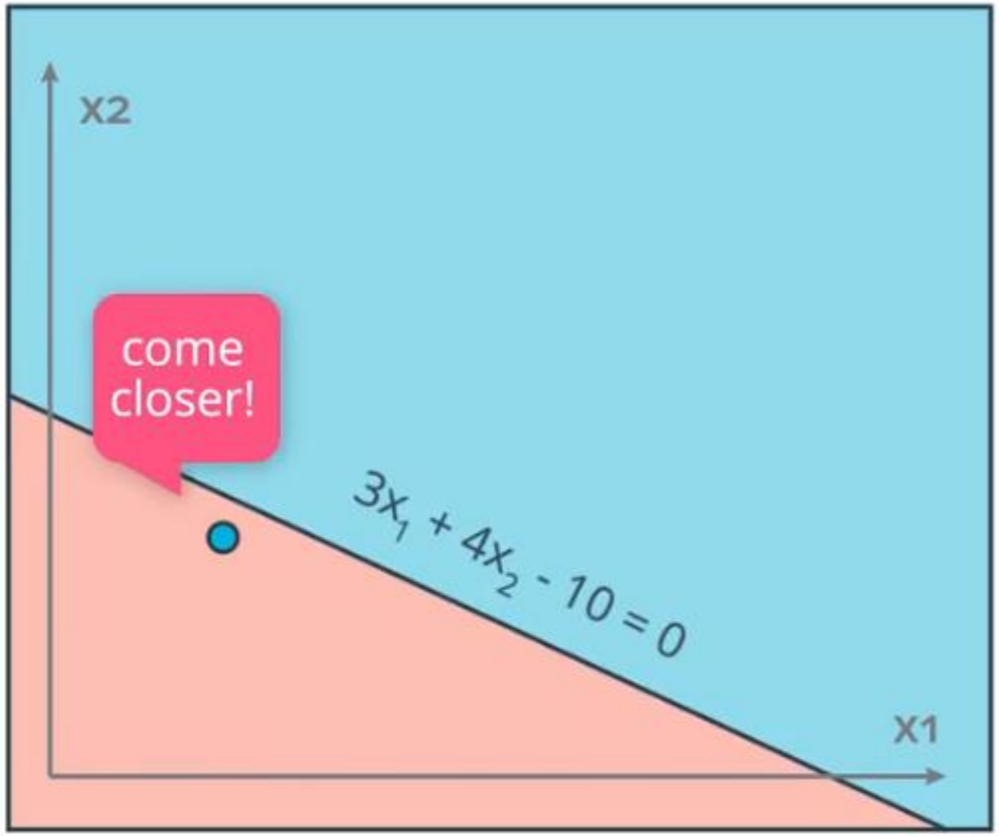
LEARNING RATE: 0.1

	3	4	-10
+	0.1	0.1	0.1
	3.1	4.1	-9.9

NEW LINE

$3.1x_1 + 4.1x_2 - 9.9 = 0$

Learning Rate



LINE: $3x_1 + 4x_2 - 10 = 0$

POINT: (1,1)

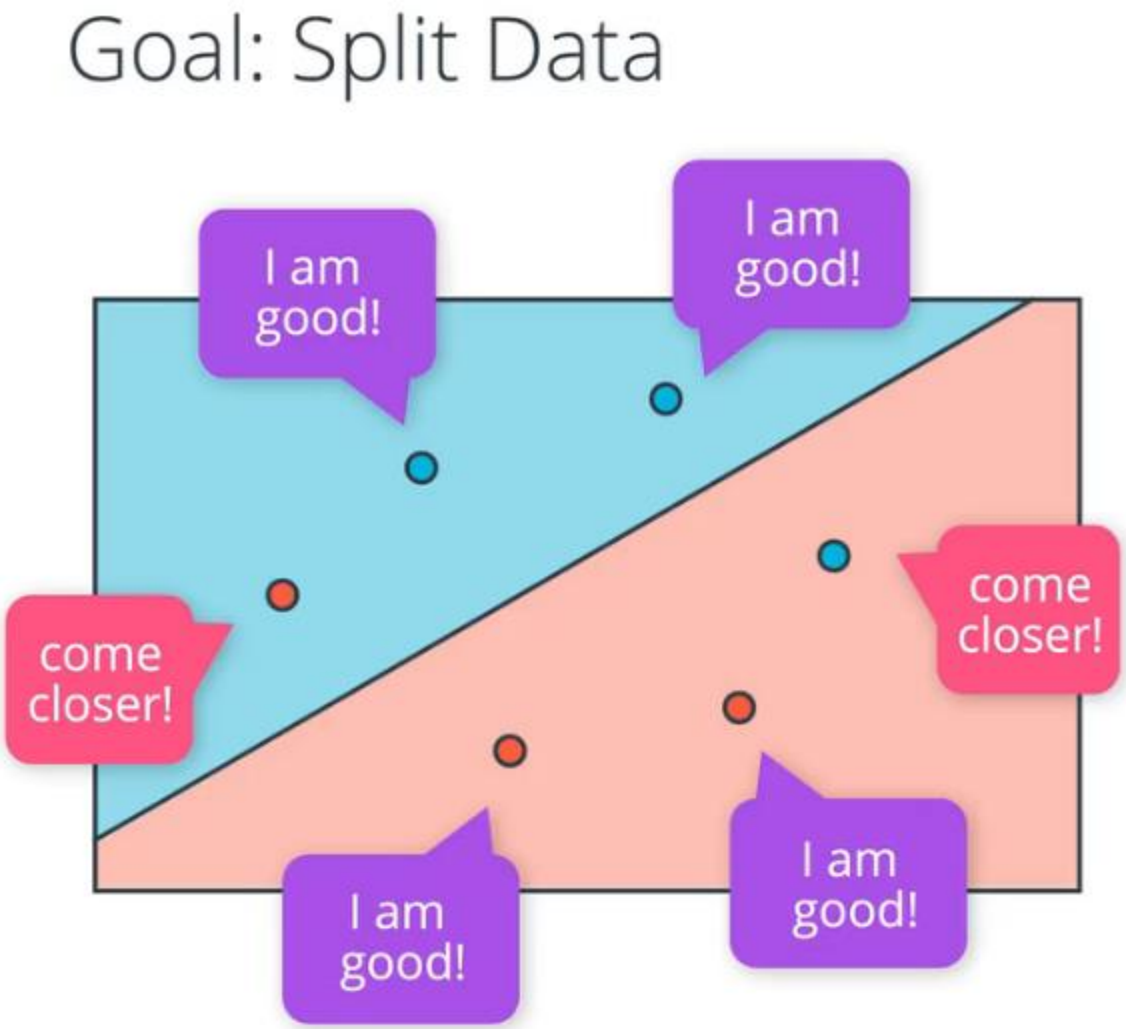
LEARNING RATE: 0.1

	3	4	-10
+	0.1	0.1	0.1
	3.1	4.1	-9.9

NEW LINE

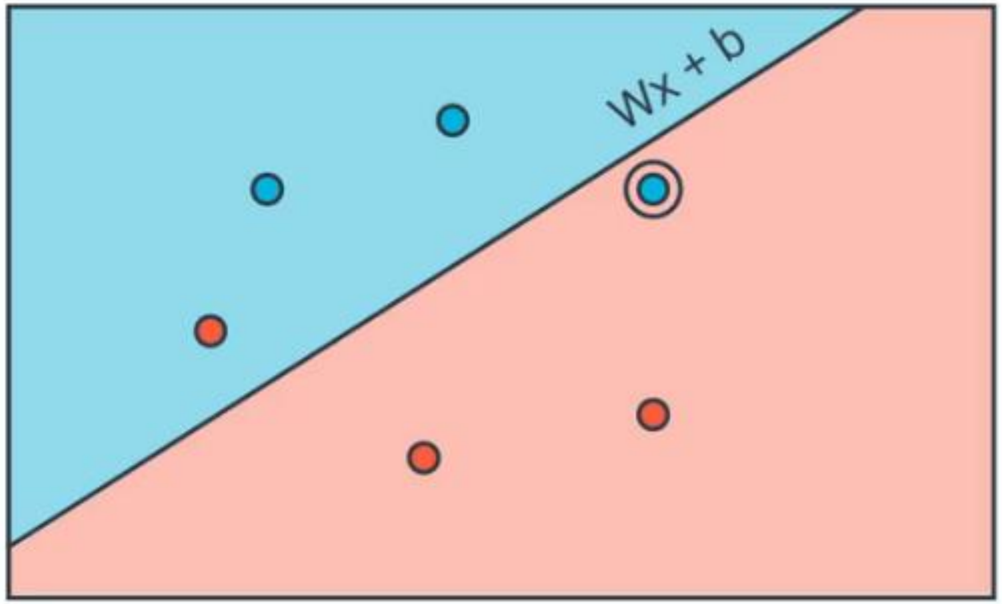
$3.1x_1 + 4.1x_2 - 9.9 = 0$

Perceptron Algorithm



Perceptron Algorithm

Perceptron Algorithm



1. Start with random weights: w_1, …, w_n, b

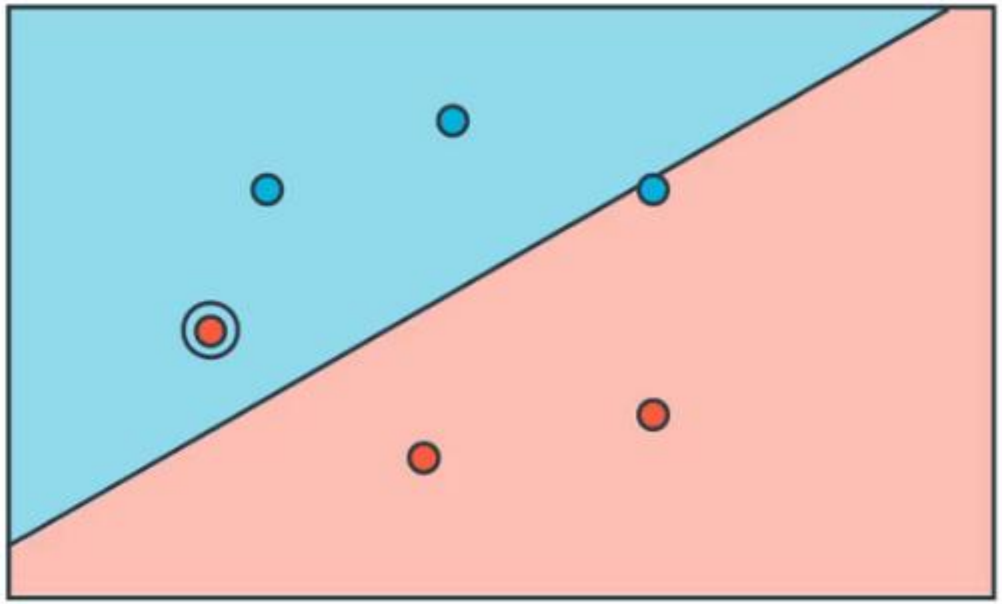
2. For every misclassified point (x_1,…,x_n):

   2.1. If prediction = 0:

   - For i = 1 …n
     - Change $w_i + \alpha\, x_i$
   - Change b to b + α

Perceptron Algorithm

Perceptron Algorithm

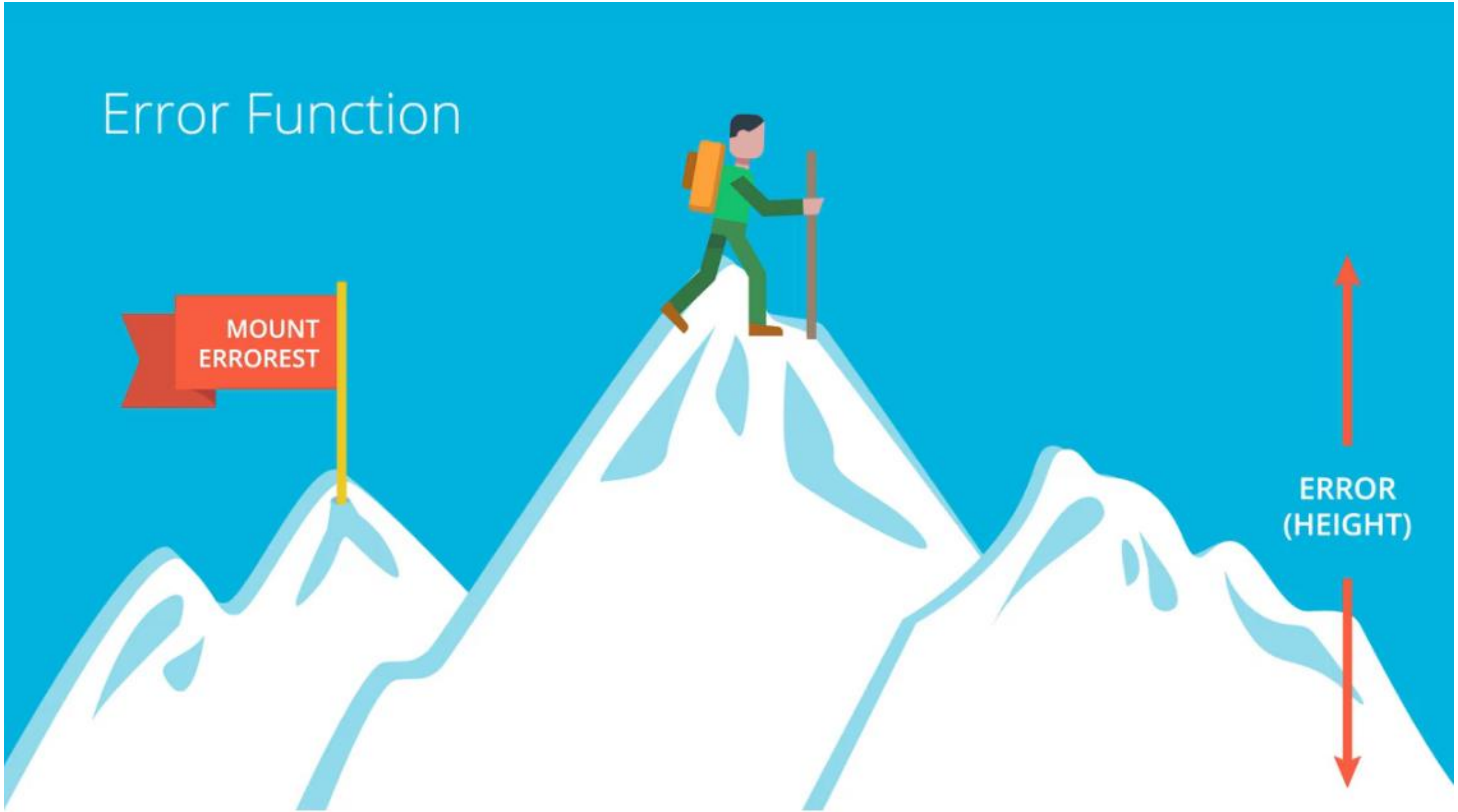


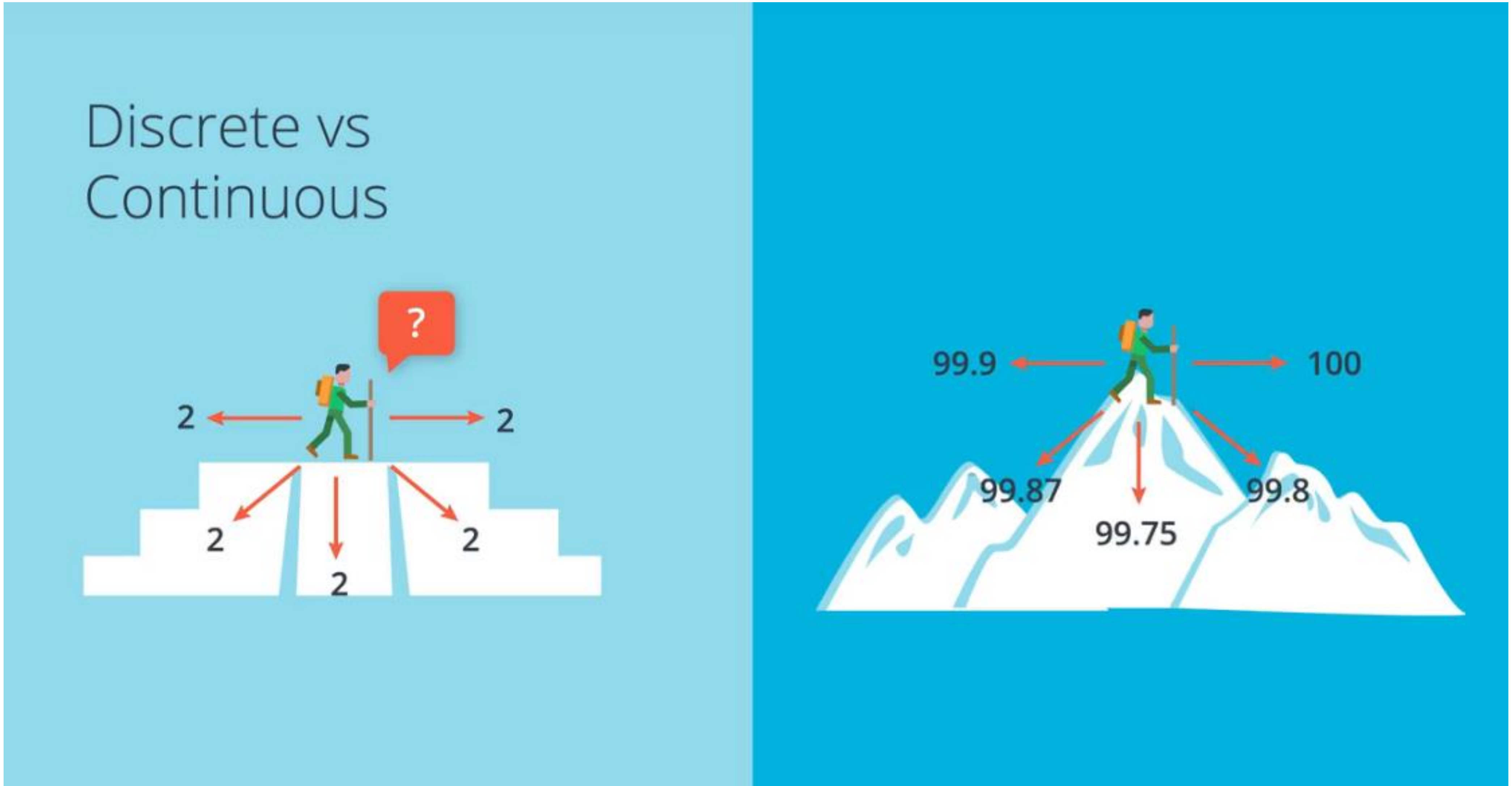
1. Start with random weights: $w_1, \ldots, w_n, b$

2. For every misclassified point $(x_1,\ldots,x_n)$:

   2.1. If prediction = 0:
   - For i = 1 ...n
     - Change $w_i + \alpha x_i$
   - Change b to $b + \alpha$

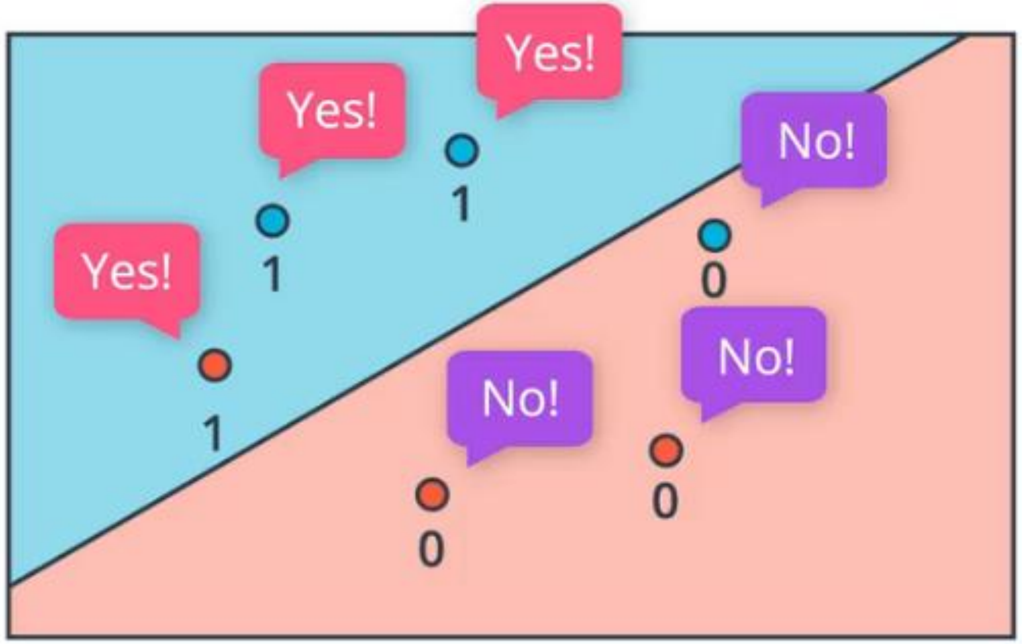
   2.2. If prediction = 1:
   - For i = 1 ...n
     - Change $w_i - \alpha x_i$
   - Change b to $b - \alpha$


Error Function
MOUNT
ERROREST
ERROR
(HEIGHT)

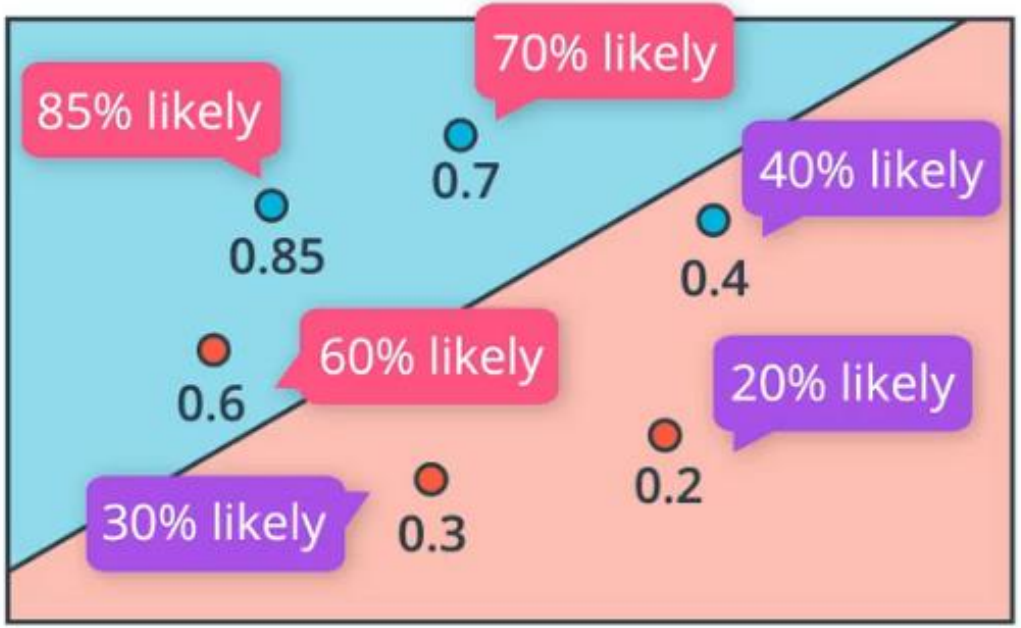


Discrete vs
Continuous
?
2
2
2
2
2
99.9
100
99.87
99.75
99.8

Predictions

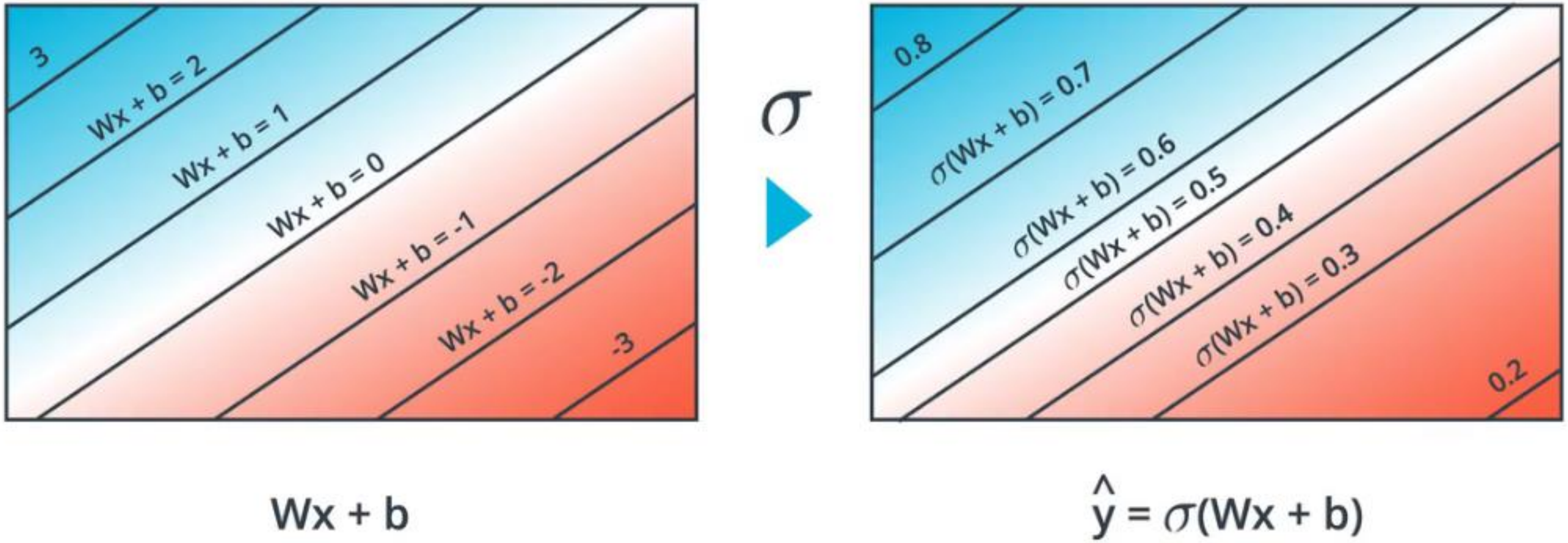


DISCRETE

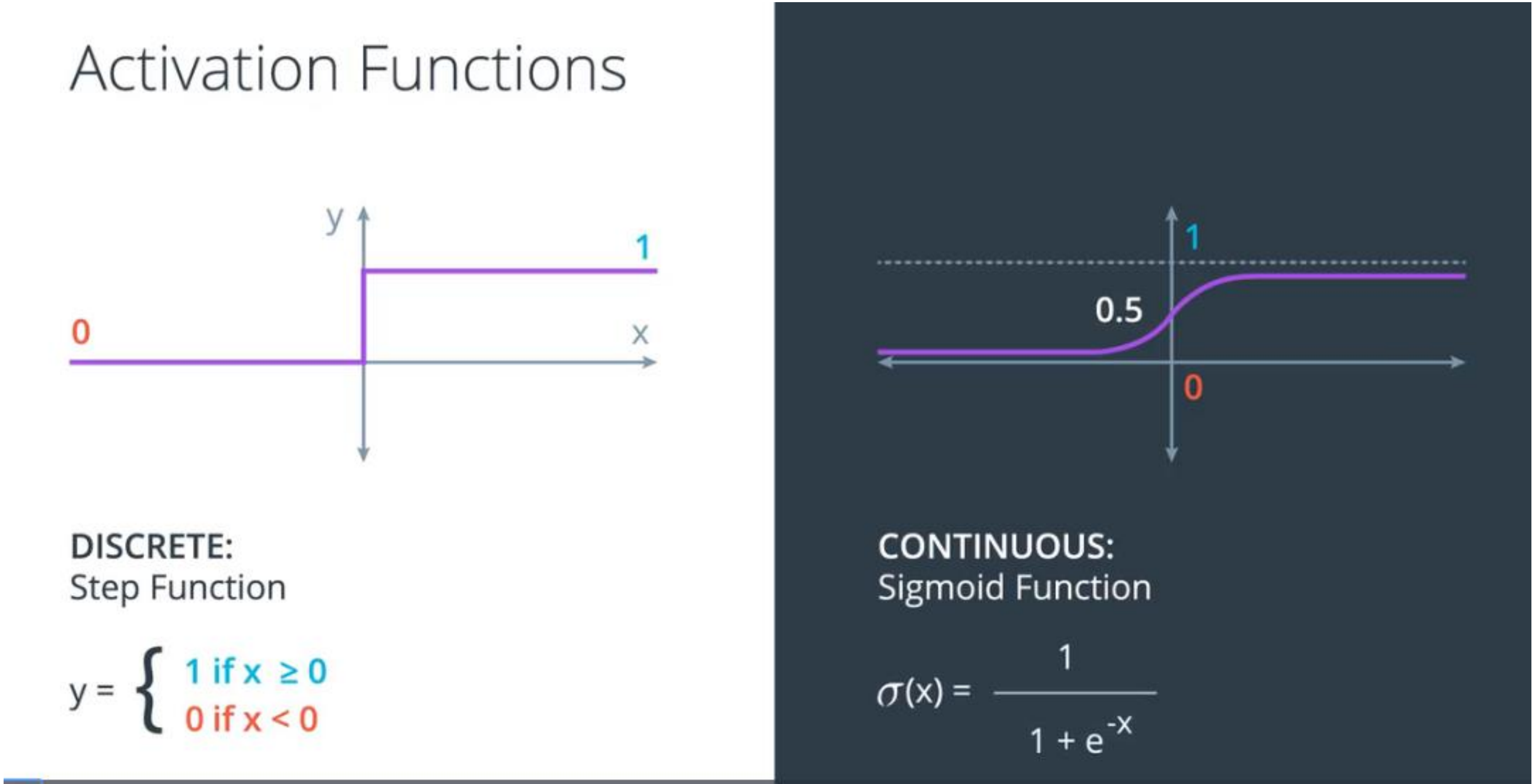


CONTINUOUS

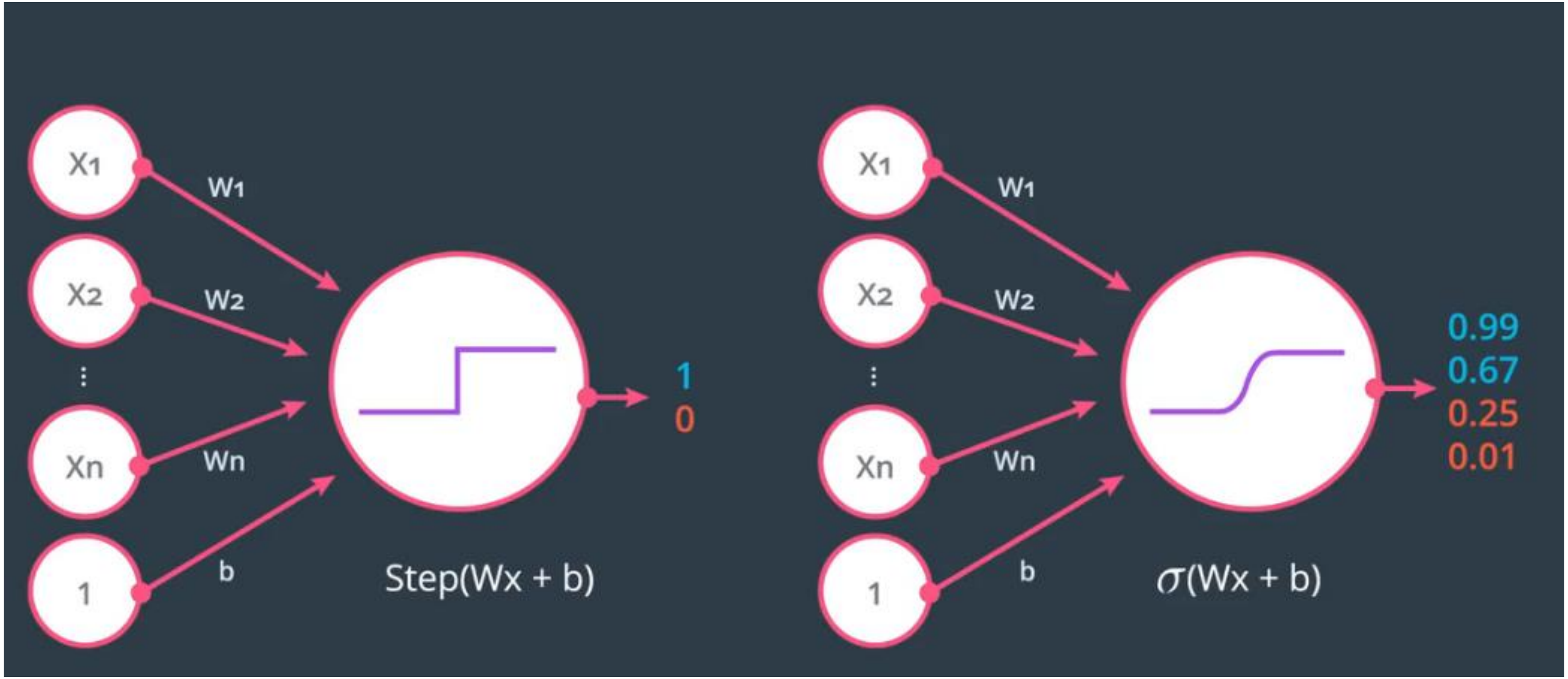
Predictions



Activation Functions



Perceptron



X1
X2
⋮
Xn
1
W1
W2
Wn
b
Step(Wx + b)
1
0
X1
X2
⋮
Xn
1
W1
W2
Wn
b
σ(Wx + b)
0.99
0.67
0.25
0.01

Softmax Function

Softmax Function

LINEAR FUNCTION SCORES:

$Z_1, \ldots, Z_n$

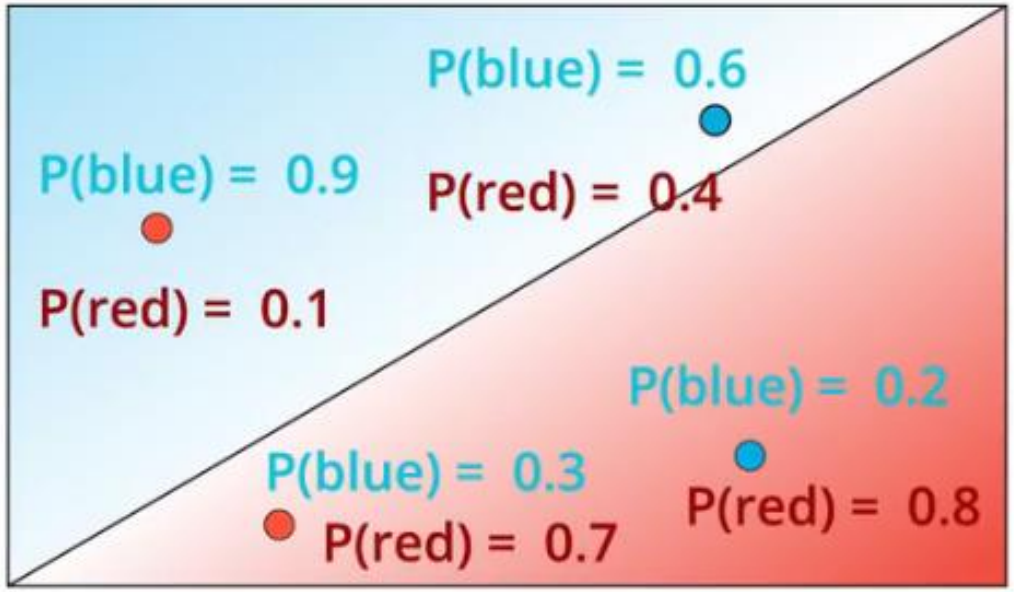
$$P(\text{class } i) = \frac{e^{Z_i}}{e^{Z_1} + \ldots + e^{Z_n}}$$

QUESTION

Is Softmax for n=2 values the same as the sigmoid function?

Maximum Likelihood

Probability

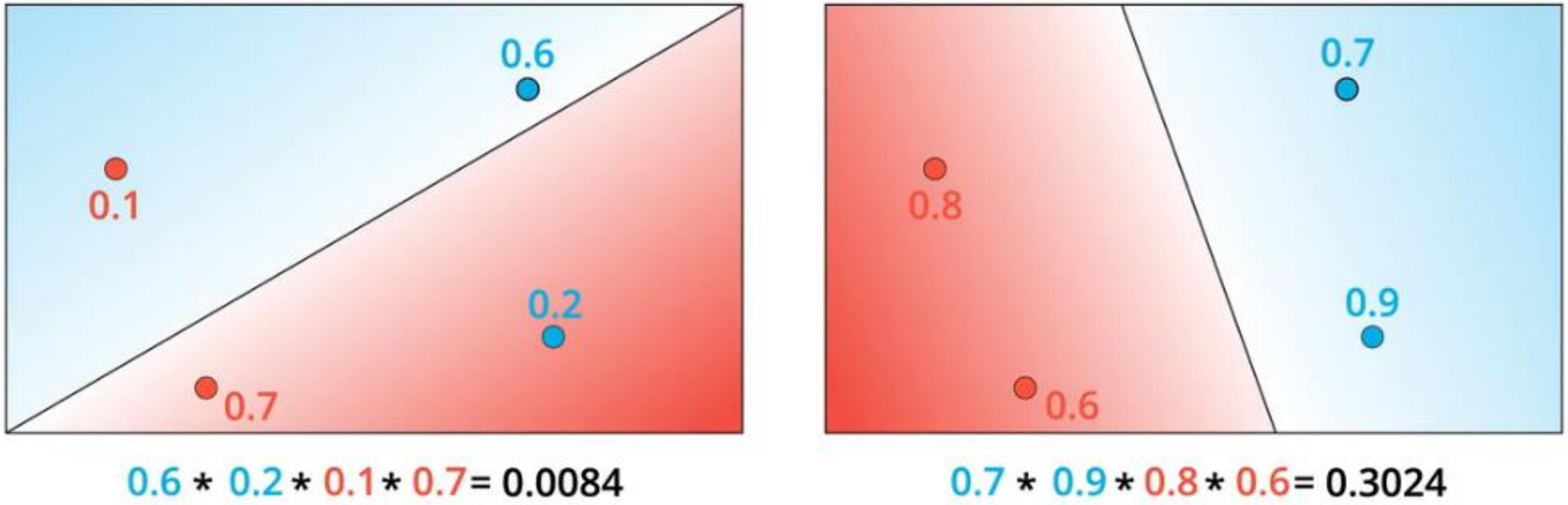


$\hat{y} = \sigma(Wx+b)$

P(blue) $= \sigma(Wx+b)$

Maximum Likelihood

Probability



Maximizing Probabilities

Products

$0.6 * 0.2 * 0.1 * 0.7 = 0.0084$

$\ln(0.6) + \ln(0.2) + \ln(0.1) + \ln(0.7)$

$-0.51 \quad -1.61 \quad -2.3 \quad -0.36$

$-\ln(0.6) - \ln(0.2) - \ln(0.1) - \ln(0.7) = 4.8$

$0.51 \quad 1.61 \quad 2.3 \quad 0.36$

$0.7 * 0.9 * 0.8 * 0.6 = 0.3024$

$\ln(0.7) + \ln(0.9) + \ln(0.8) + \ln(0.6)$

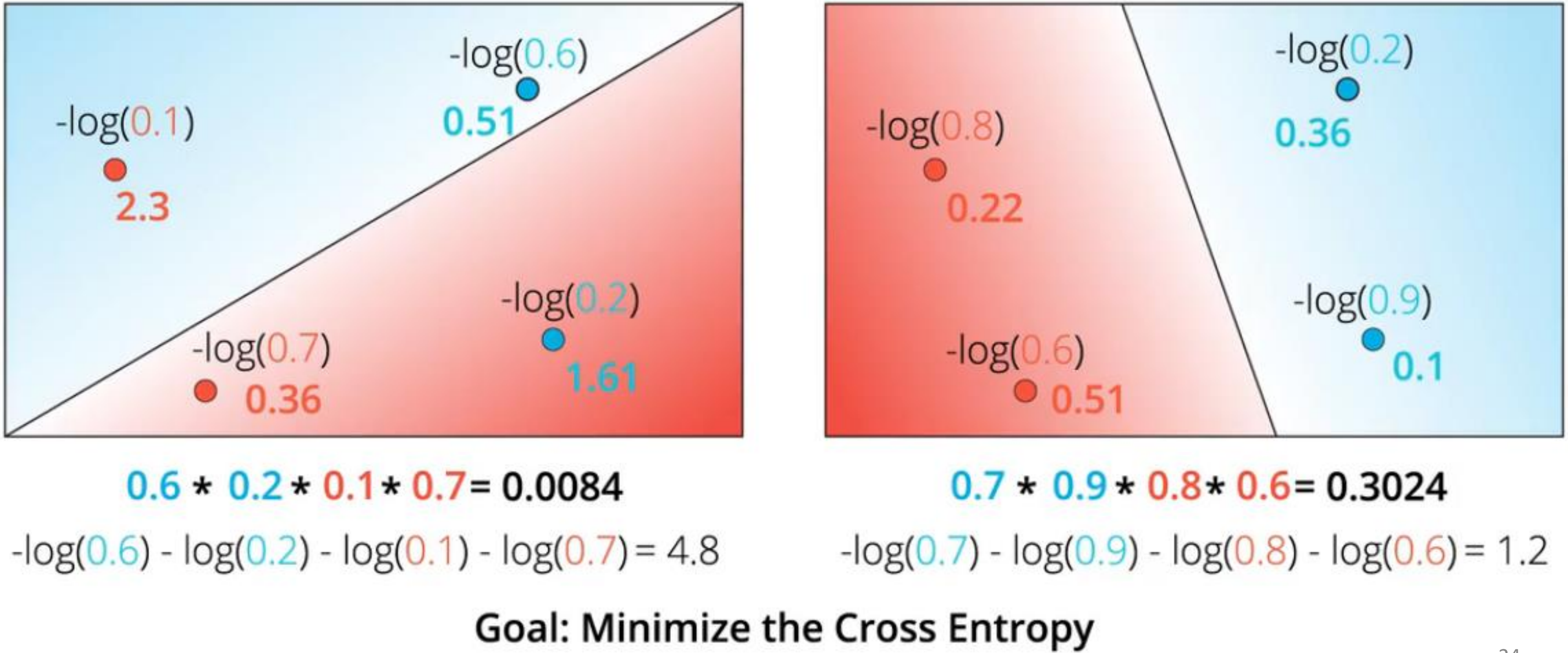
$-0.36 \quad -0.1 \quad -.22 \quad -0.51$

$-\ln(0.7) - \ln(0.9) - \ln(0.8) - \ln(0.6) = 1.2$

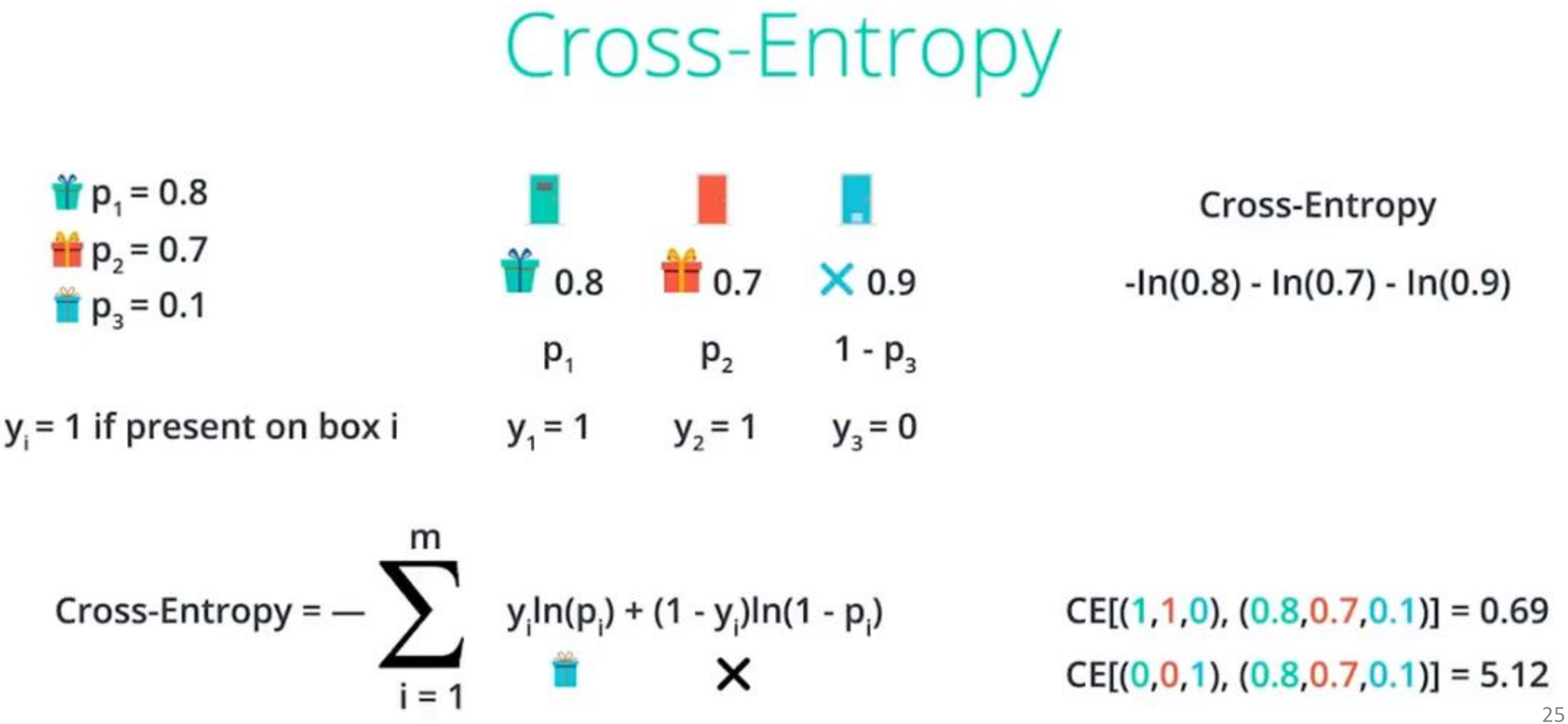
$0.36 \quad 0.1 \quad .22 \quad 0.51$

Cross Entropy

Cross Entropy

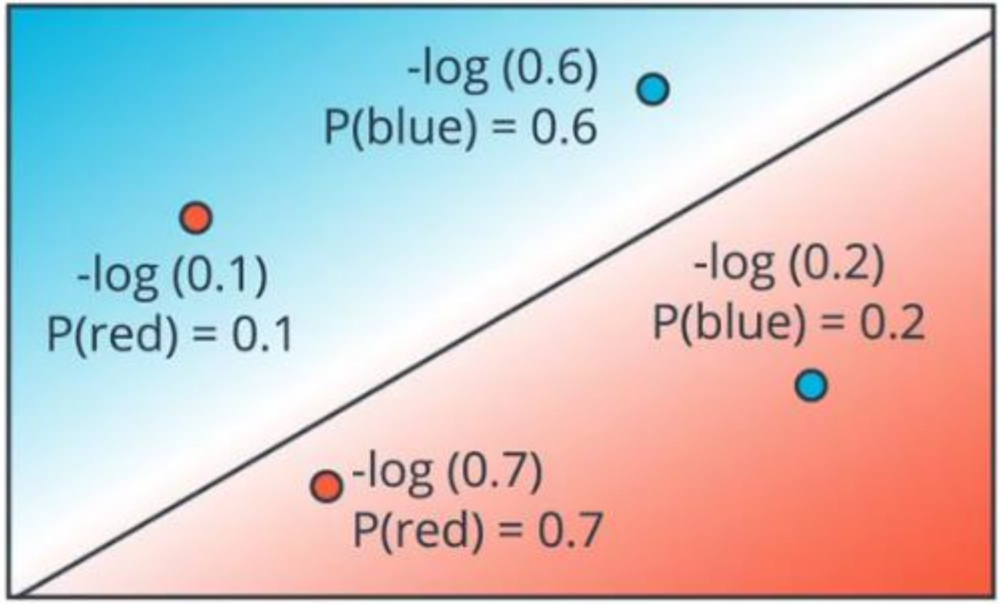


Cross Entropy



Logistic Regression

Error Function



$-\log(0.6) - \log(0.2) - \log(0.1) - \log(0.7) = 4.8$

0.51 1.61 2.3 0.36

If $y = 1$

$P(\text{blue}) = \hat{y}$

$\text{Error} = -\ln(\hat{y})$

If $y = 0$

$P(\text{red}) = 1 - P(\text{blue}) = 1 - \hat{y}$

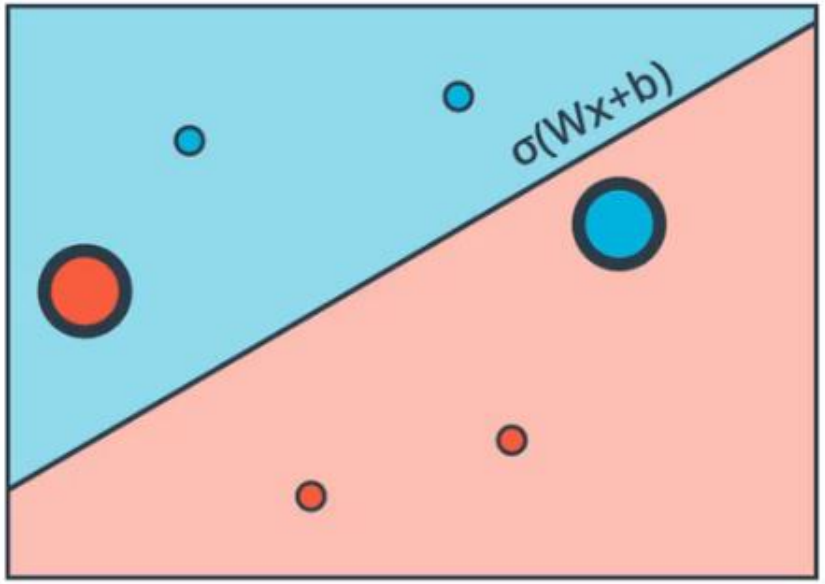
$\text{Error} = -\ln(1 - \hat{y})$

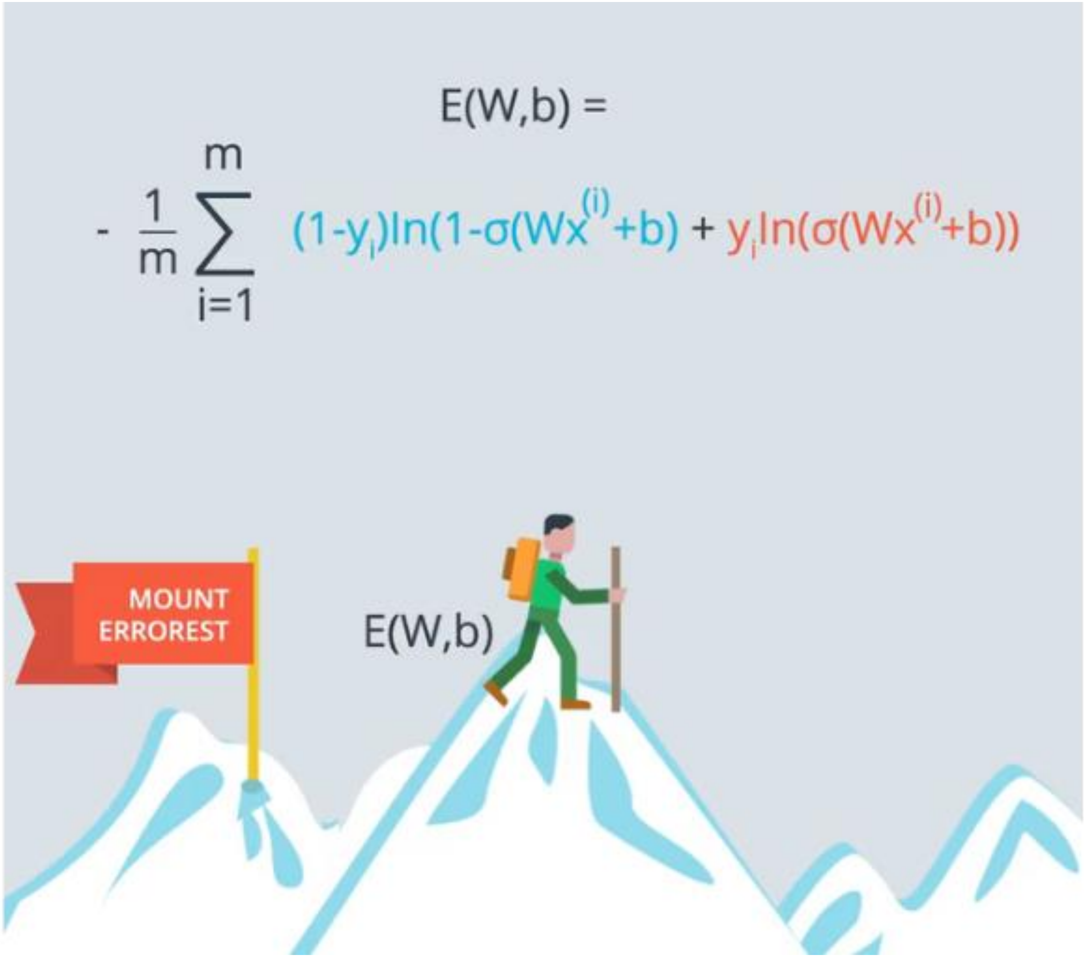
$\text{Error} = -(1-y)(\ln(1-\hat{y})) - y\ln(\hat{y})$

$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)(\ln(1-\hat{y}_i)) + y_i\ln(\hat{y}_i)$$

Logistic Regression

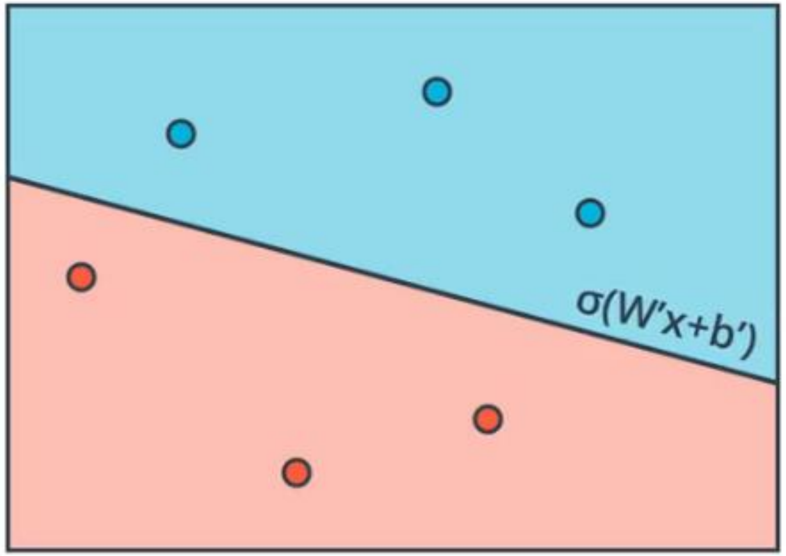
Goal: Minimize
Error Function

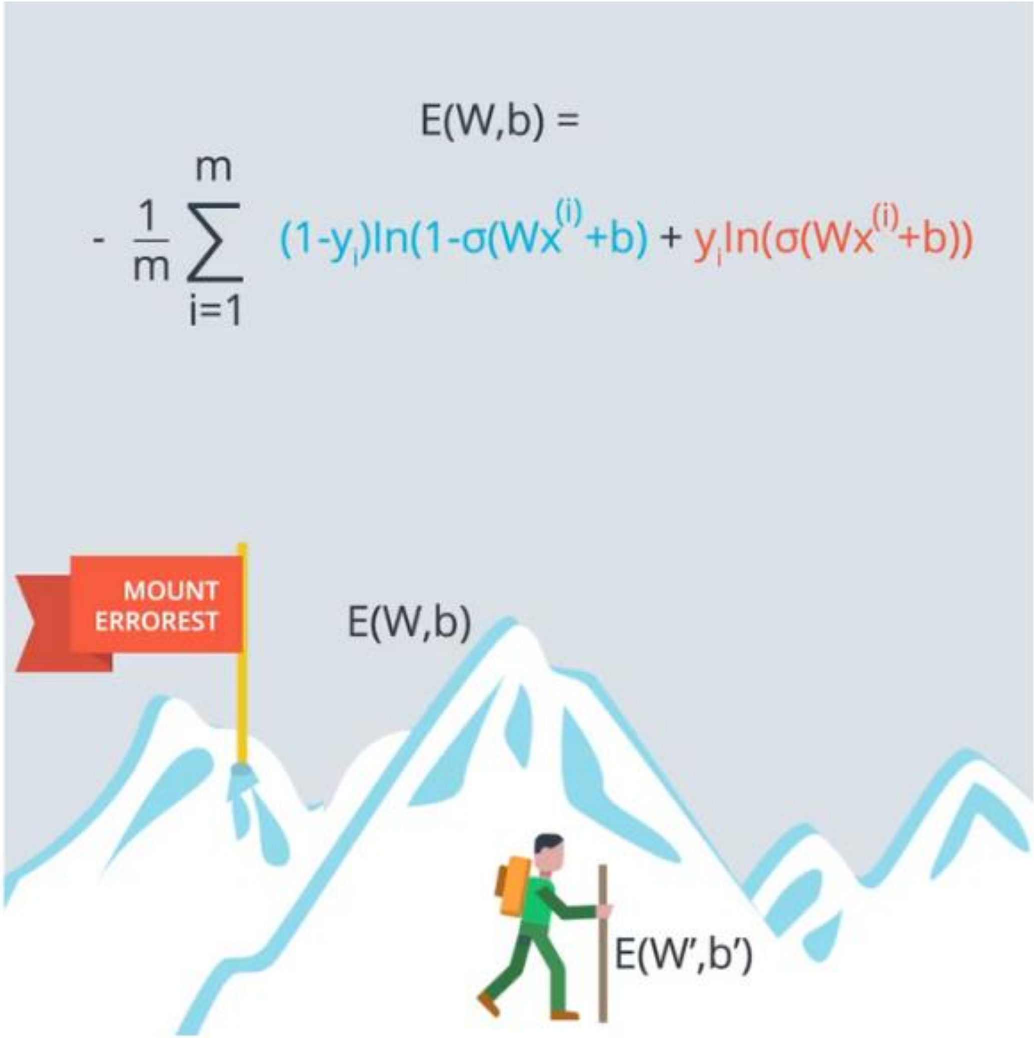




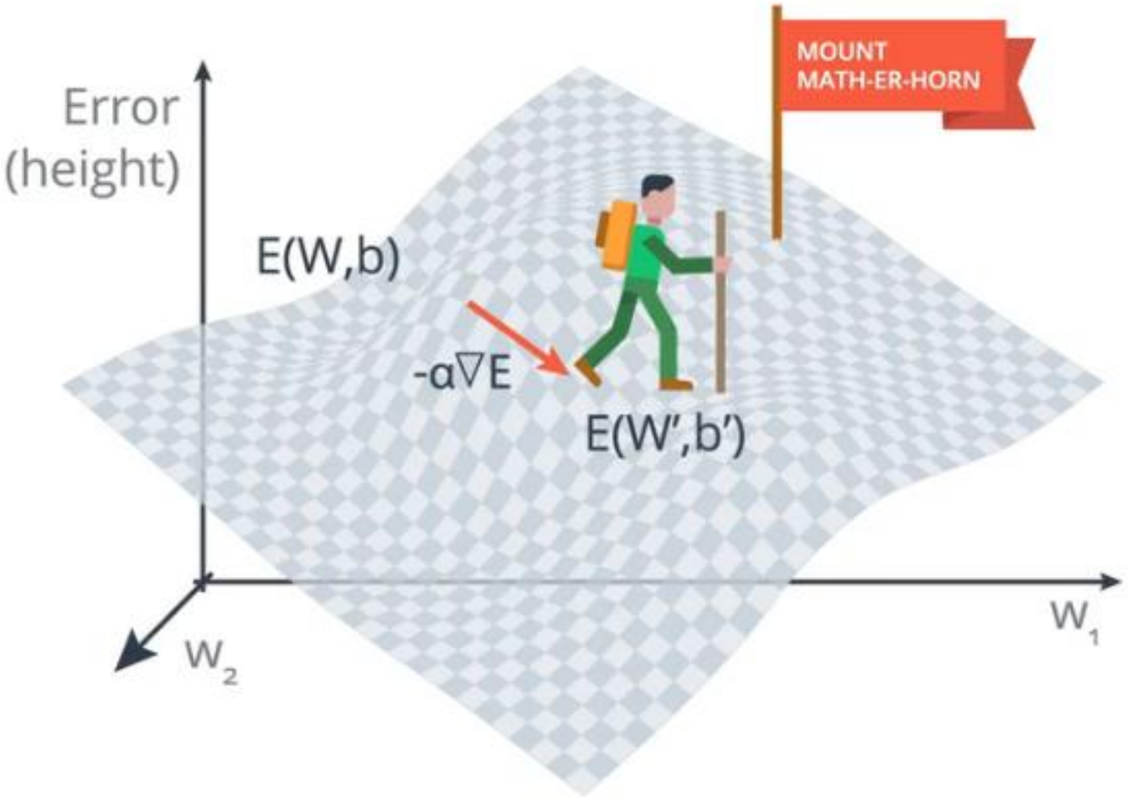
Logistic Regression

Goal: Minimize Error Function





Gradient Descent



$\hat{y} = \sigma(Wx+b) \longleftarrow$ Bad

$\hat{y} = \sigma(w_1x_1 + \dots + w_nx_n + b)$

$\nabla E = (\partial E/\partial w_1, \dots, \partial E/\partial w_n, \partial E/\partial b)$

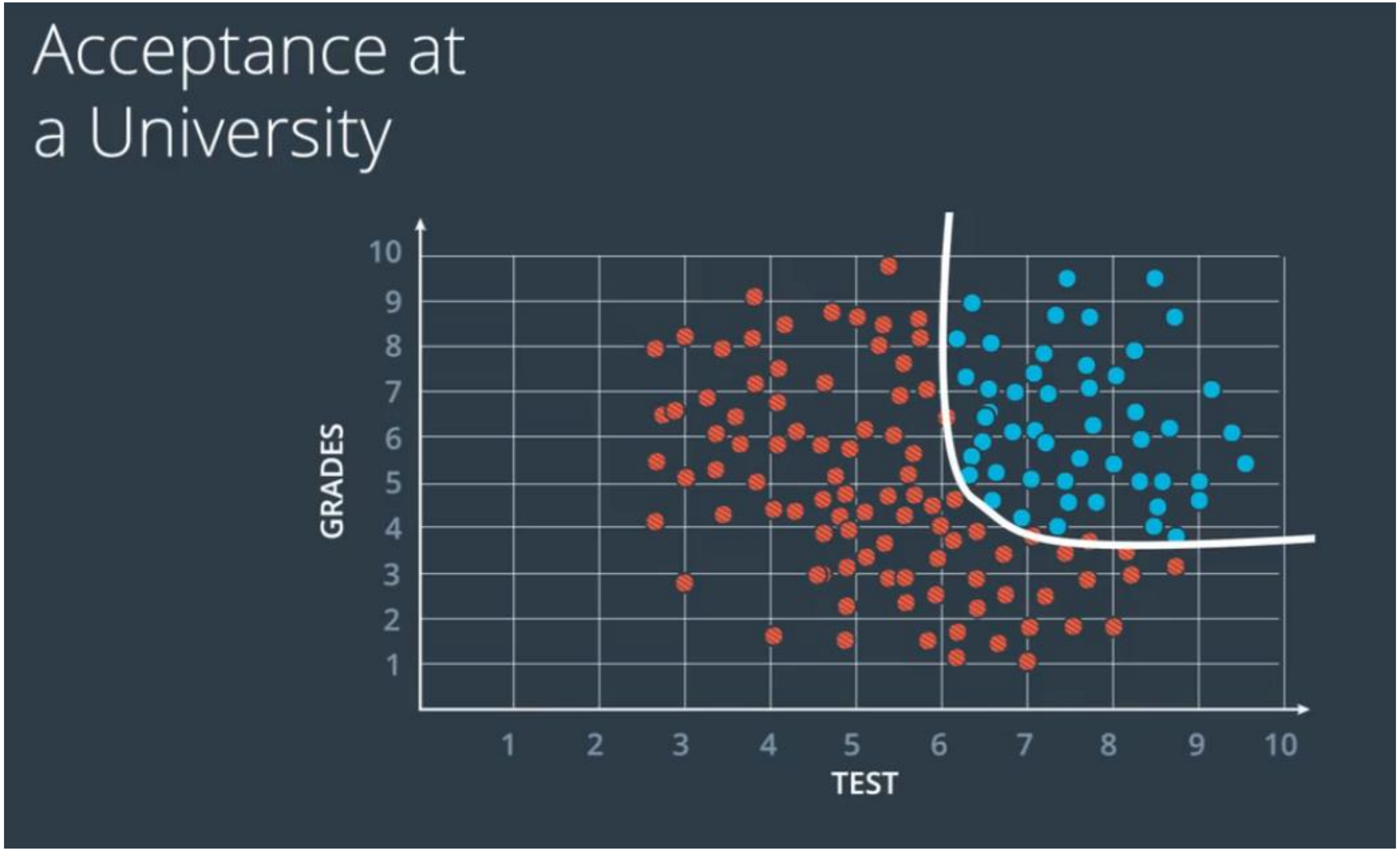
$\alpha = 0.1$ (learning rate)

$w_i' \longleftarrow w_i - \alpha\, \partial E/\partial w_i$

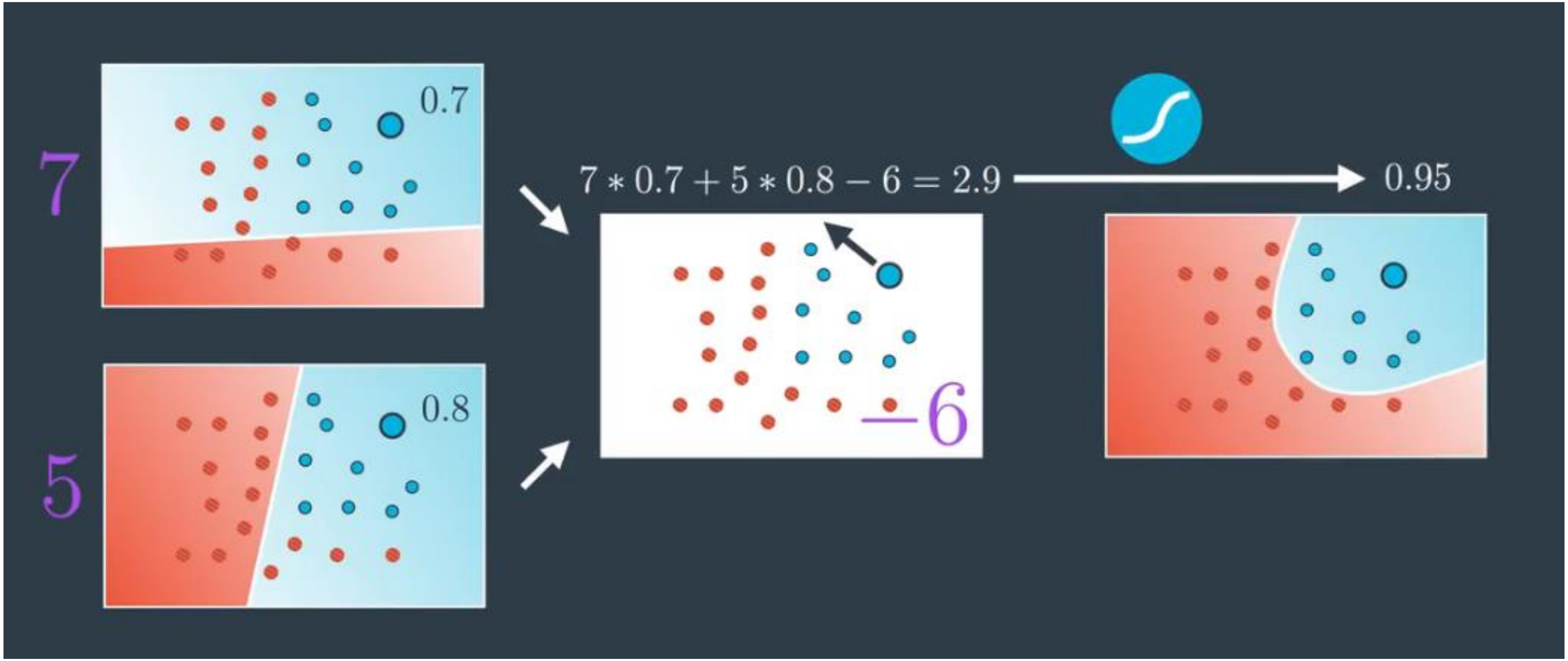
$b' \longleftarrow b - \alpha\, \partial E/\partial b$

$\hat{y} = \sigma(W'x+b') \longleftarrow$ Better

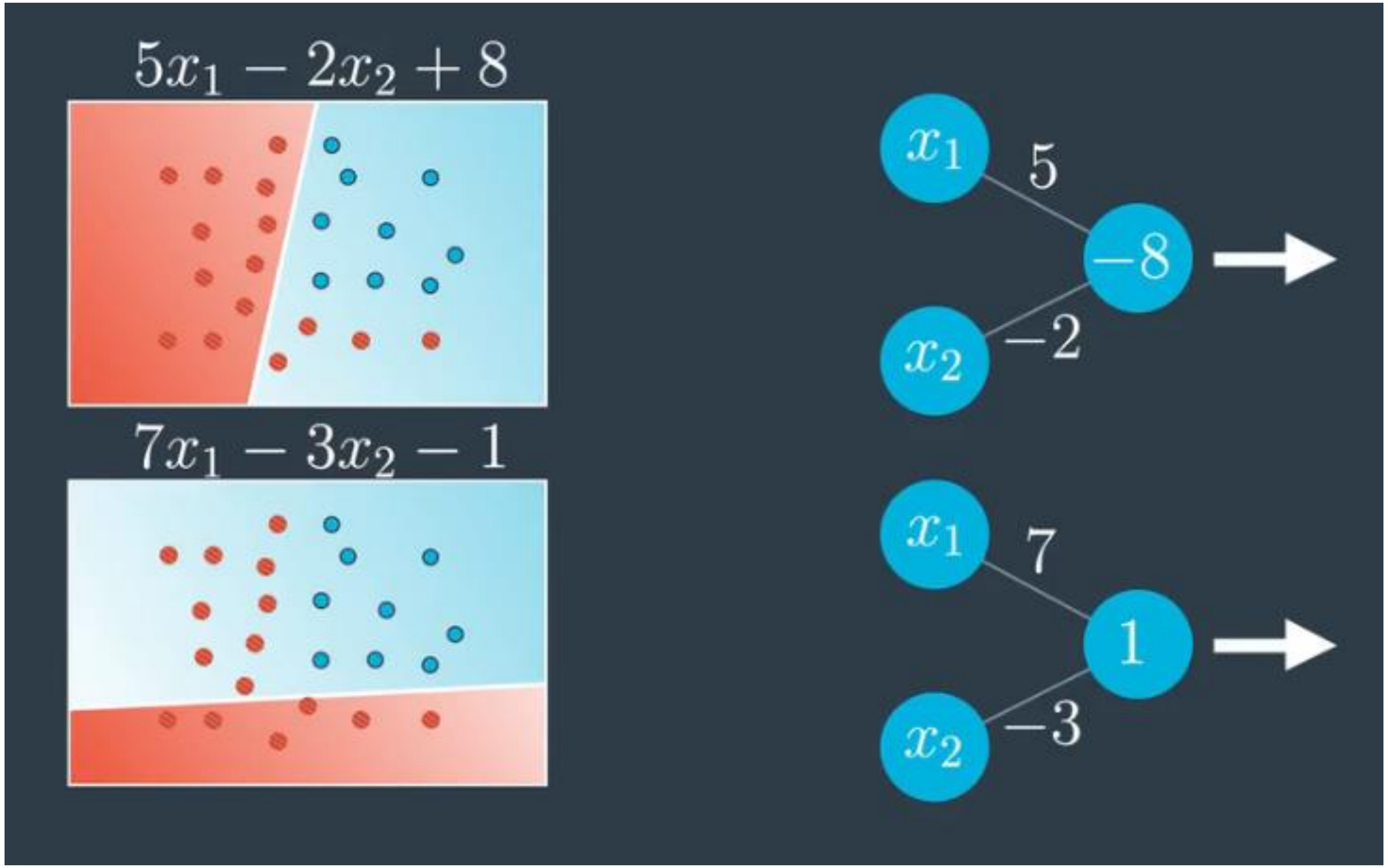
Non-Linear Models



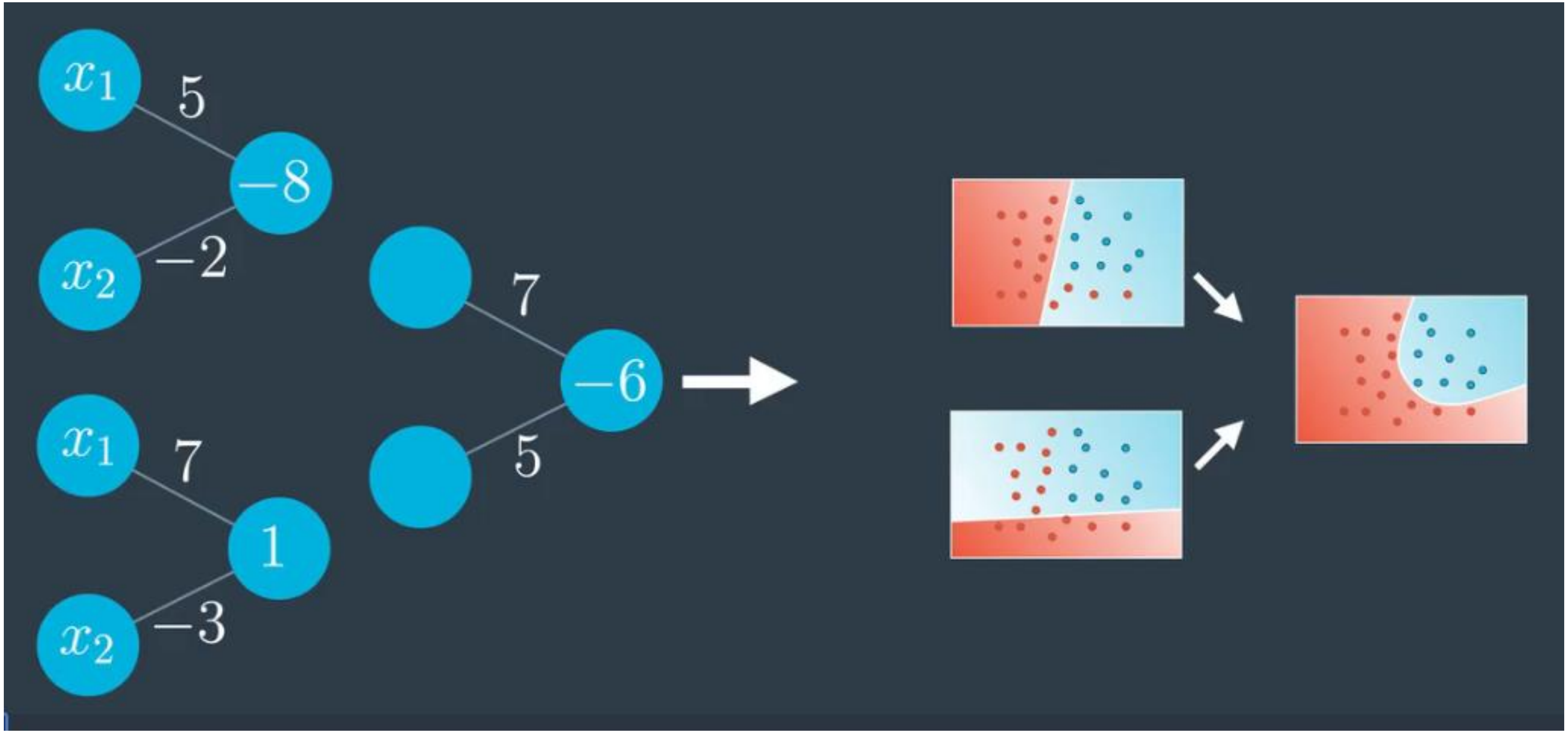
Neural Network Architecture



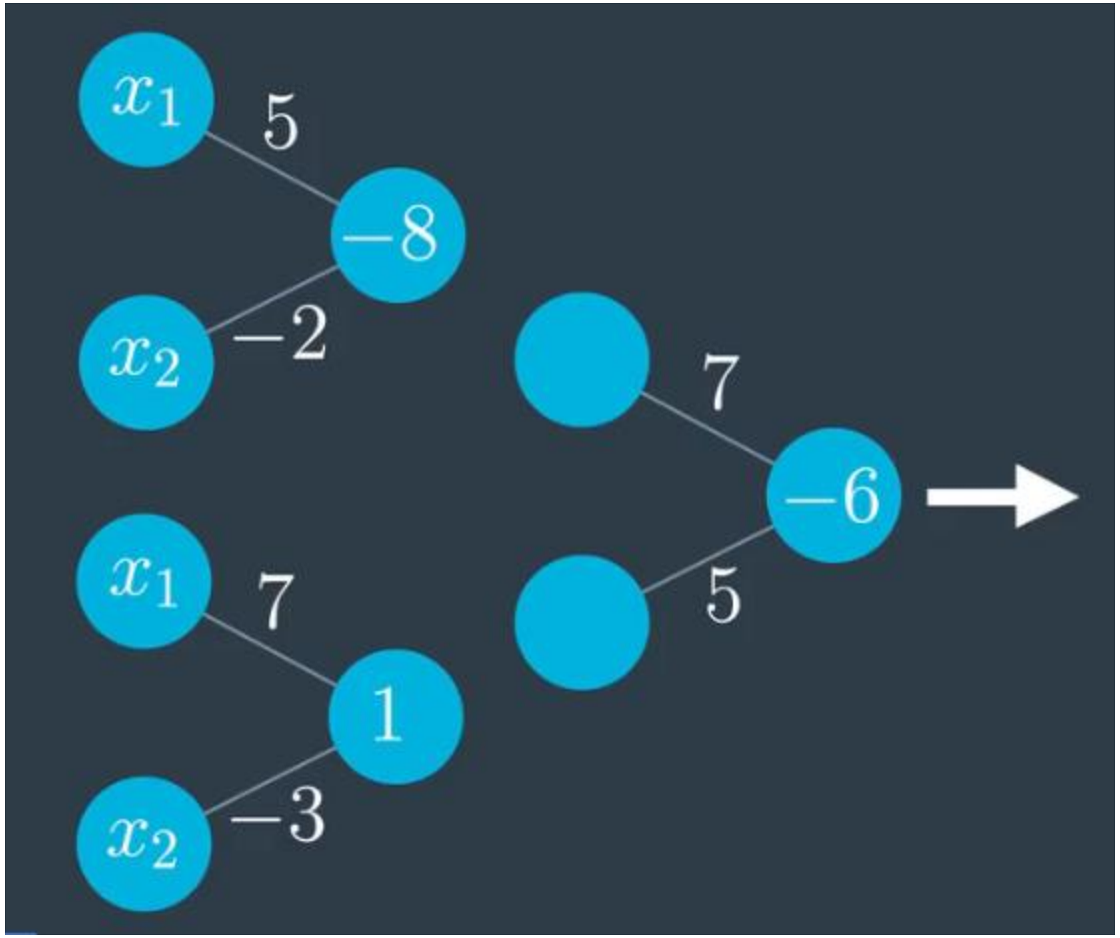
Neural Network Architecture

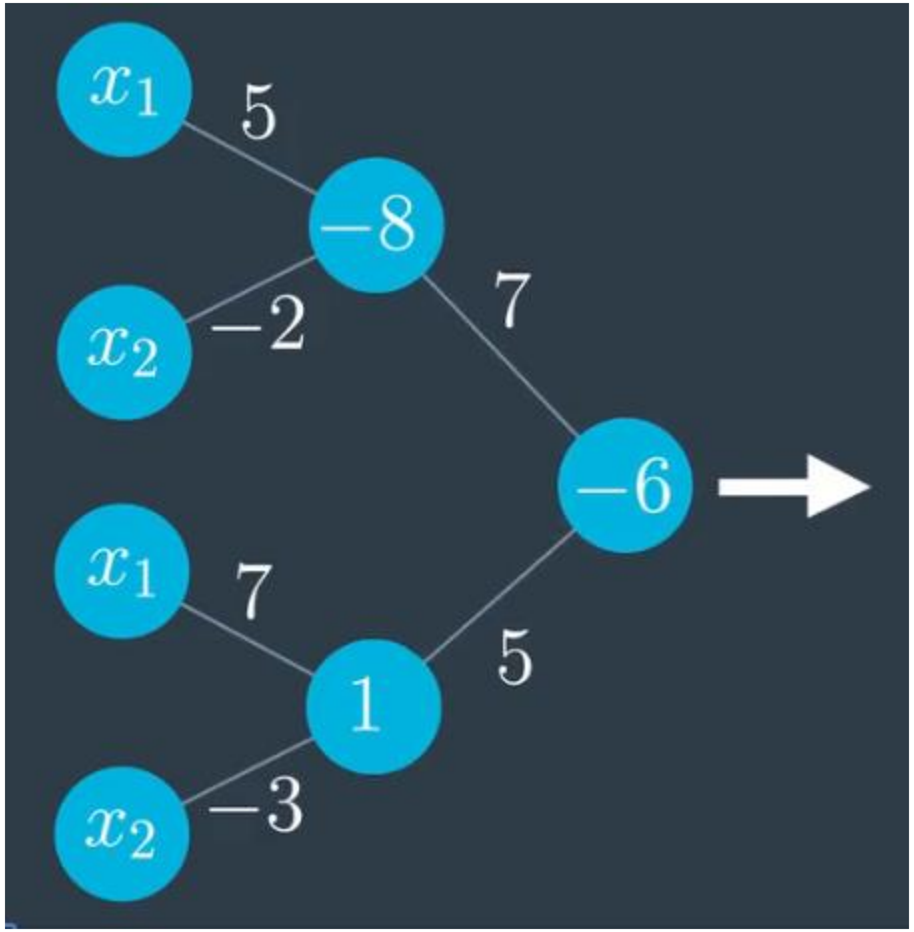


Neural Network Architecture

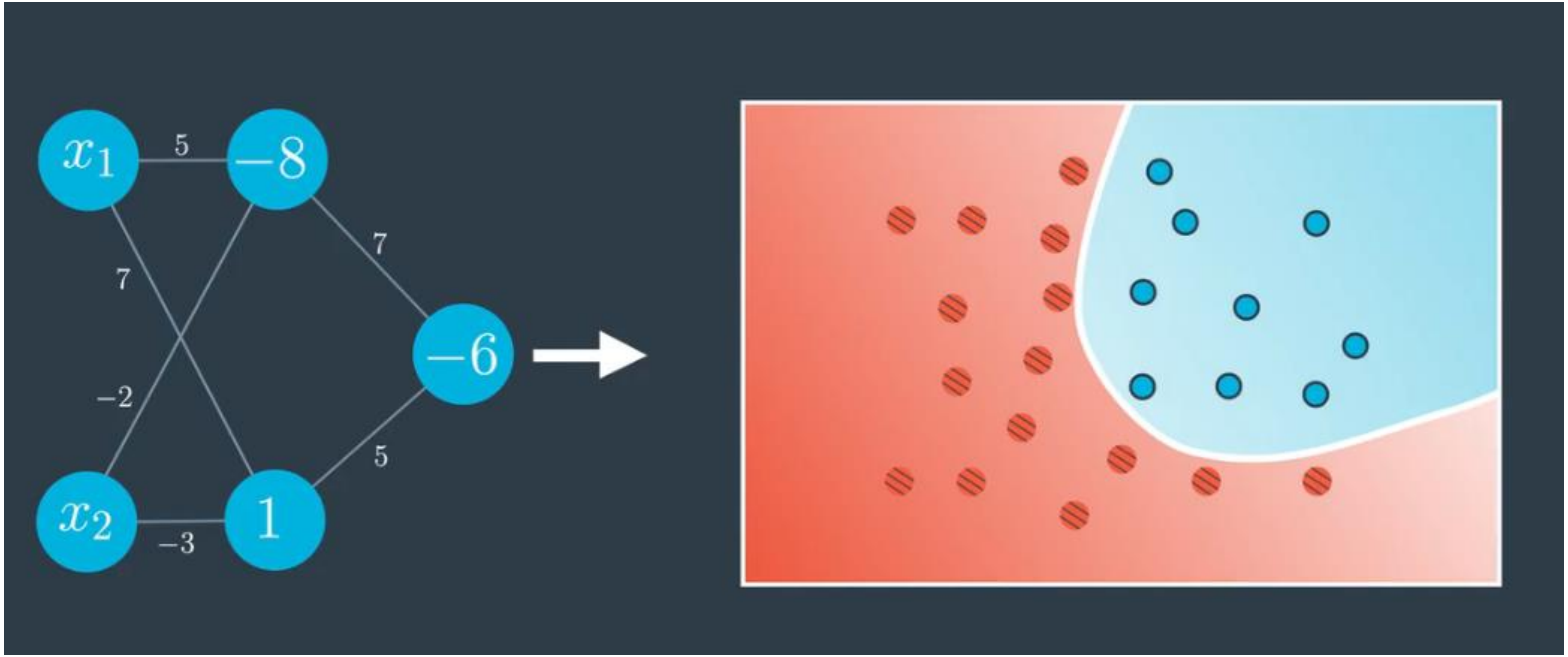


Neural Network Architecture

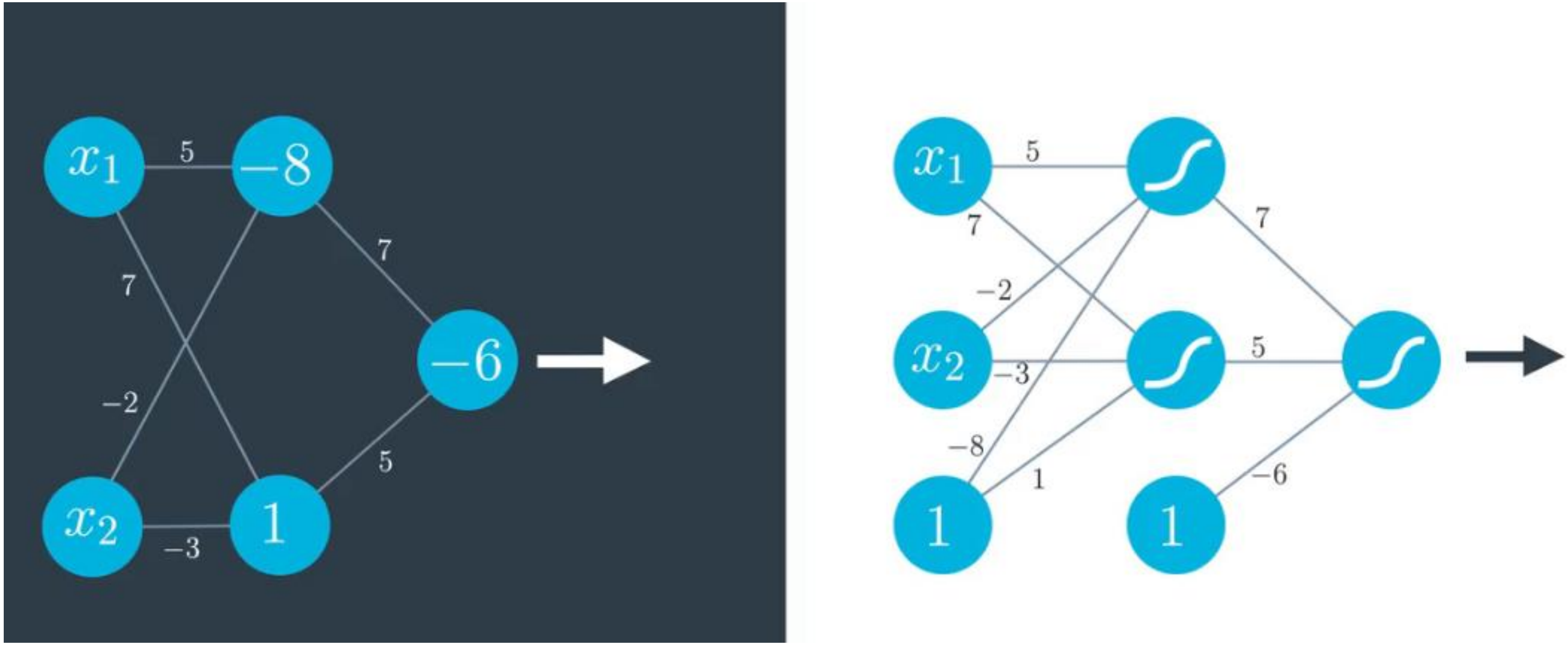




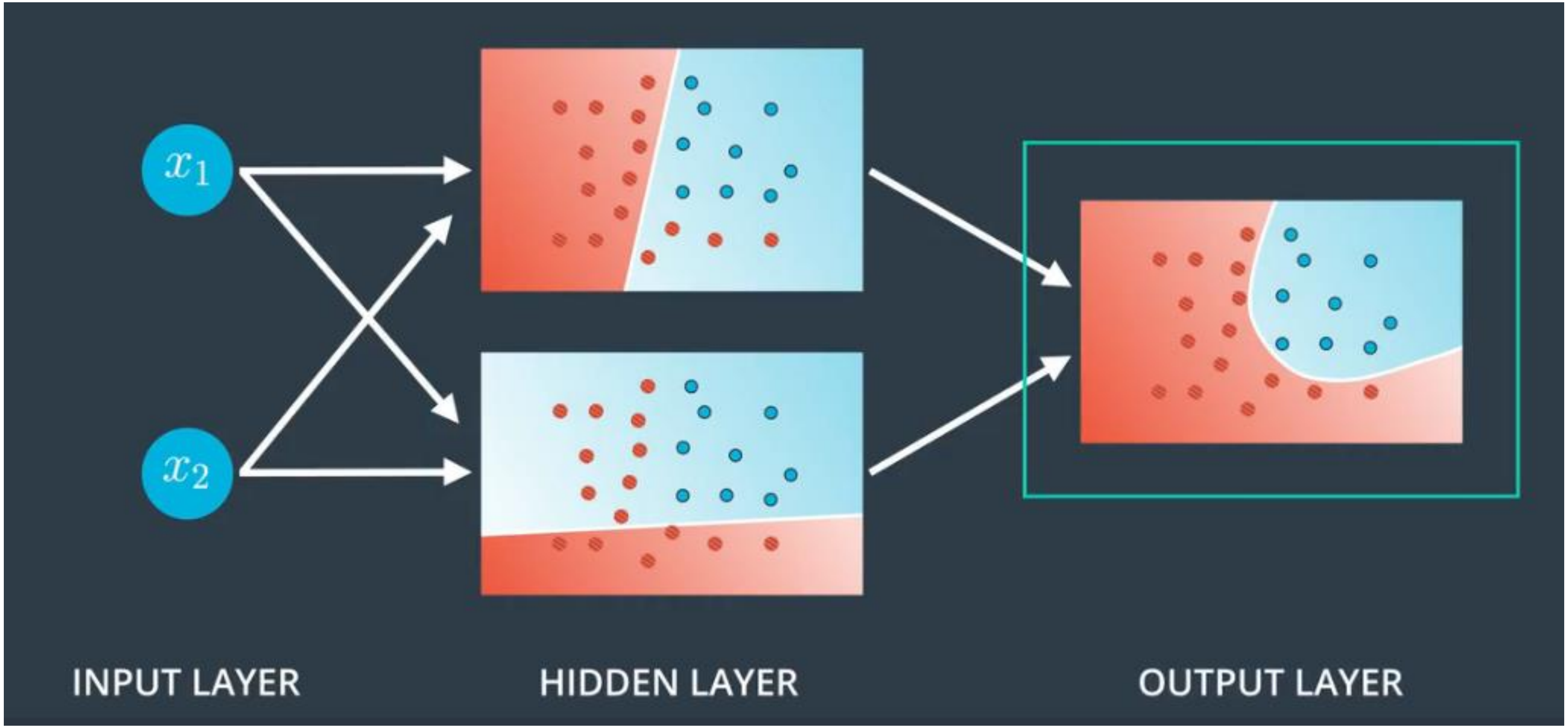
Neural Network Architecture



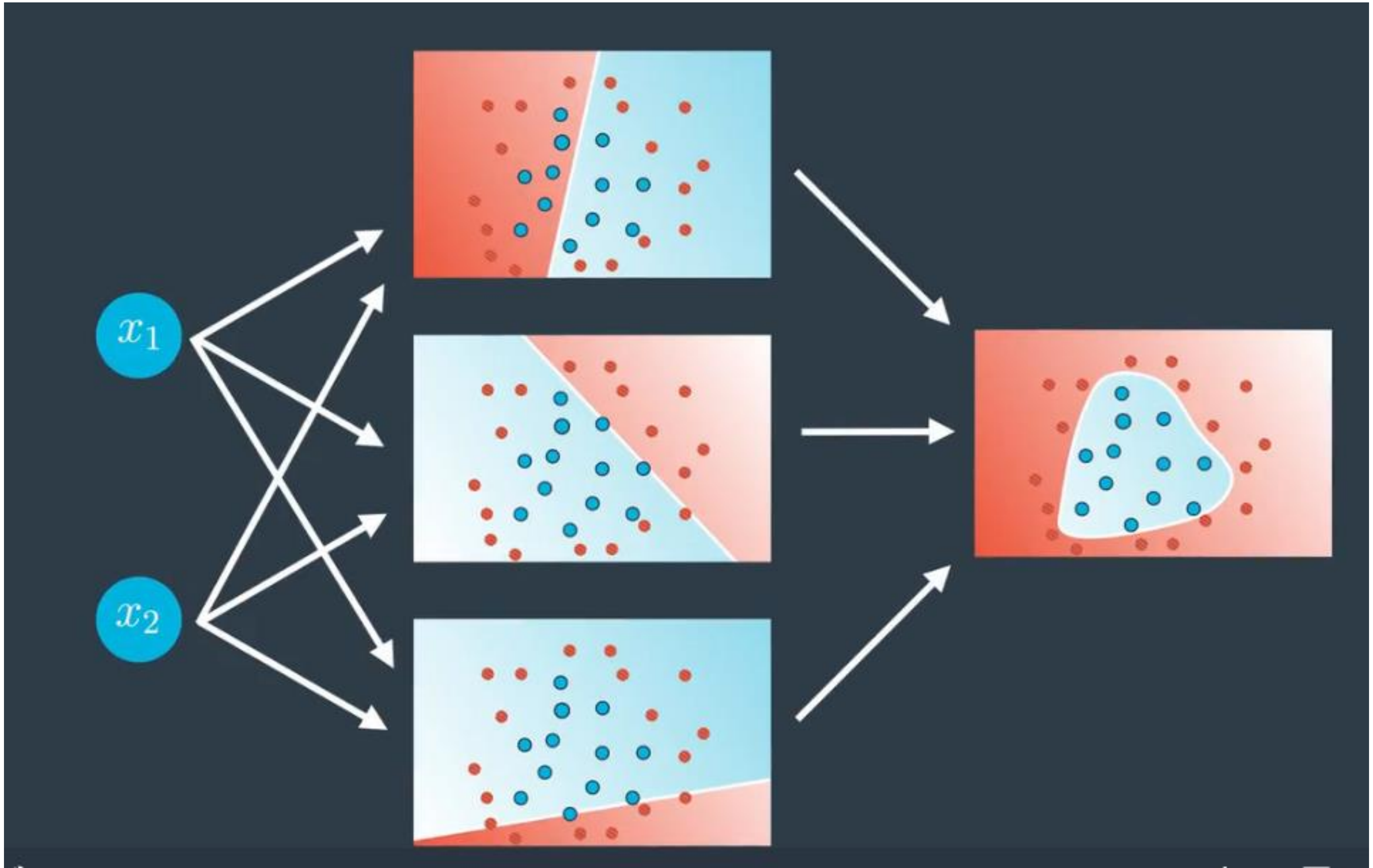
Neural Network Architecture



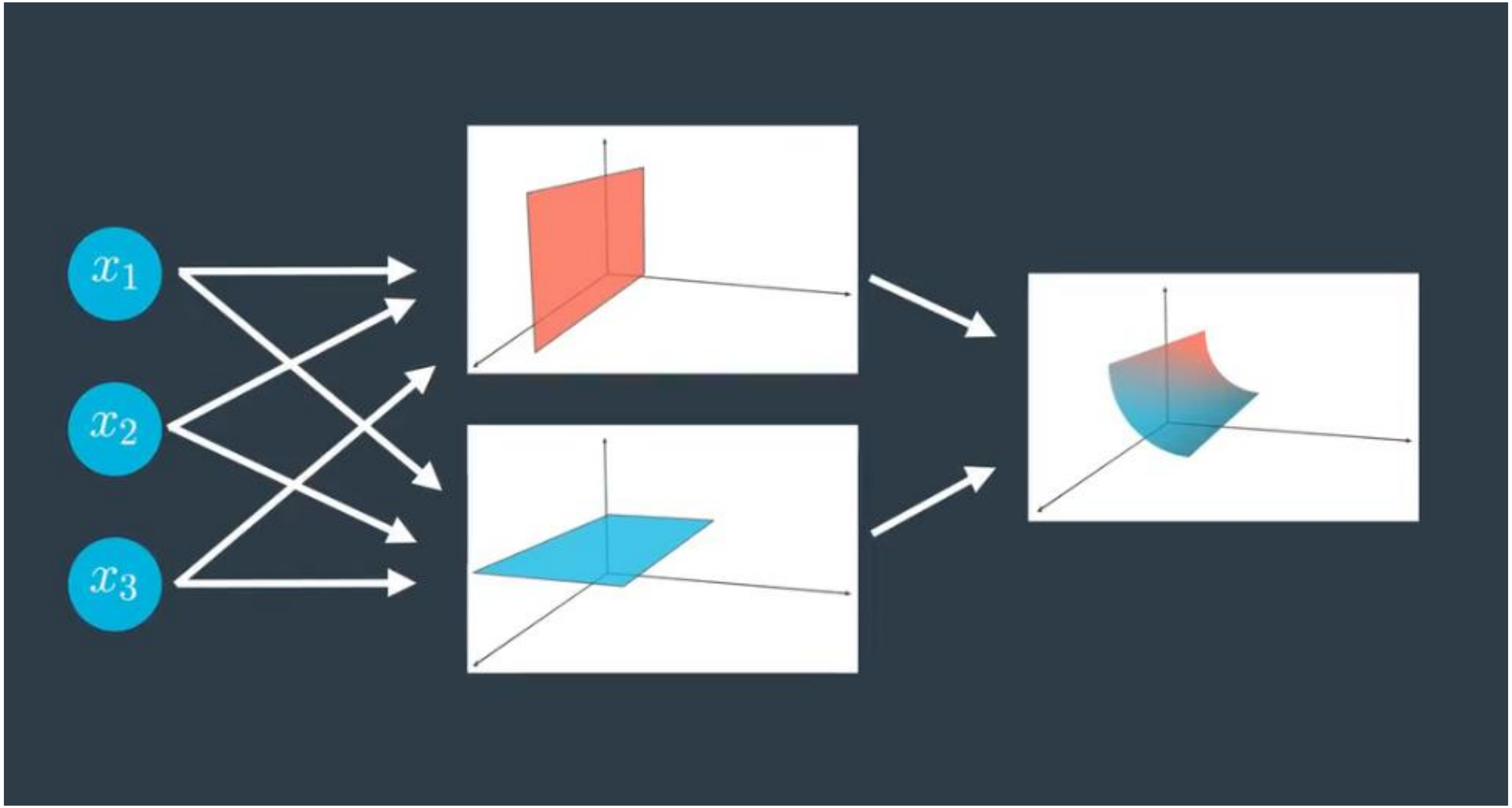
Neural Network Architecture



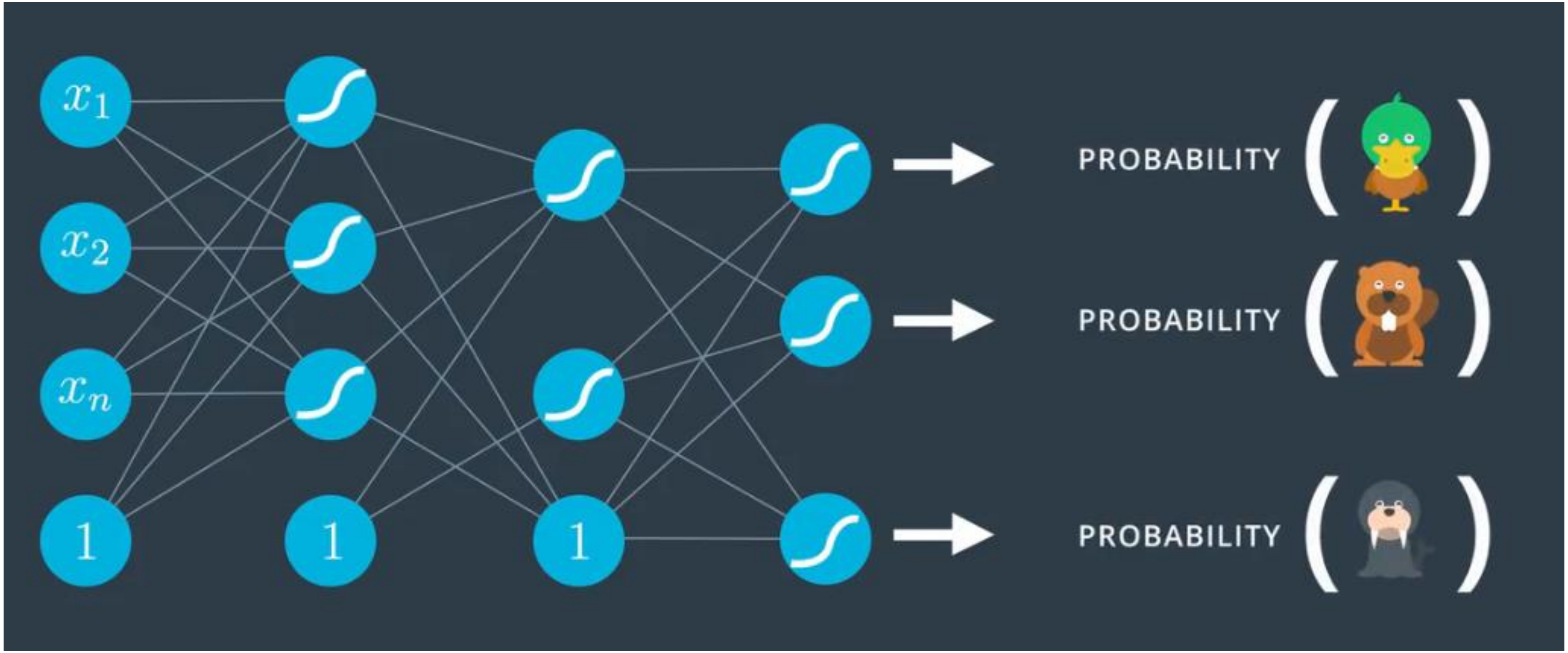
Neural Network Architecture



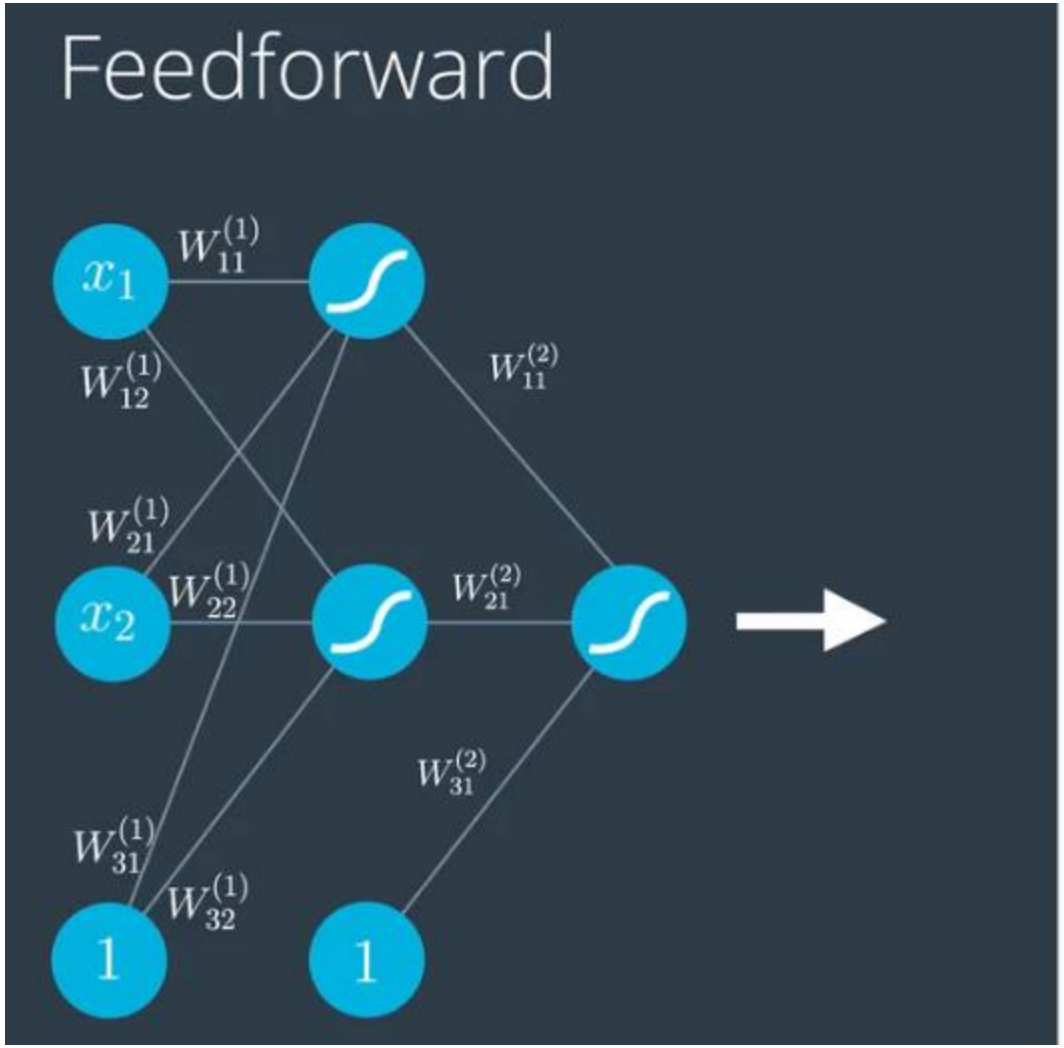
Neural Network Architecture



Neural Network Architecture



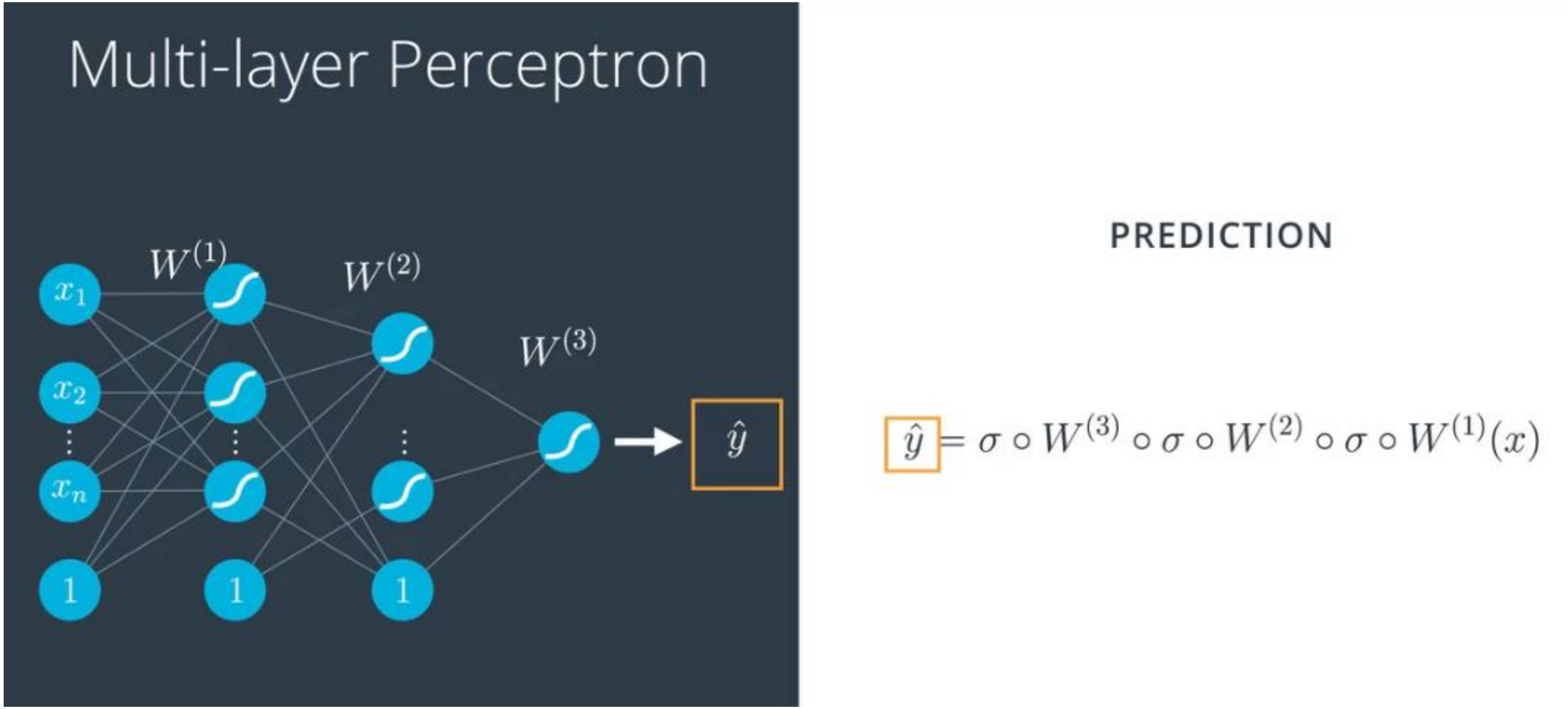
Feedforward



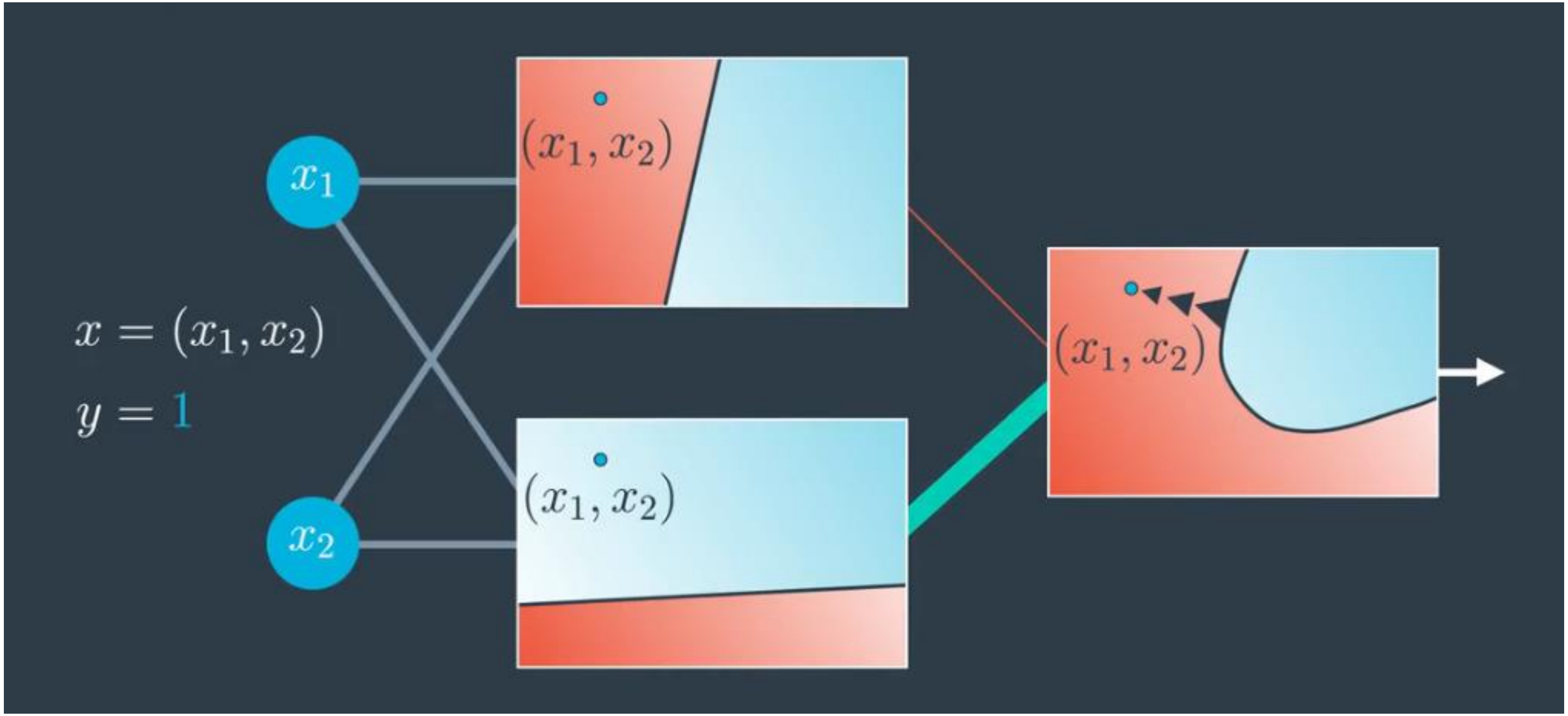
$$\hat{y} = \sigma \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

$$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

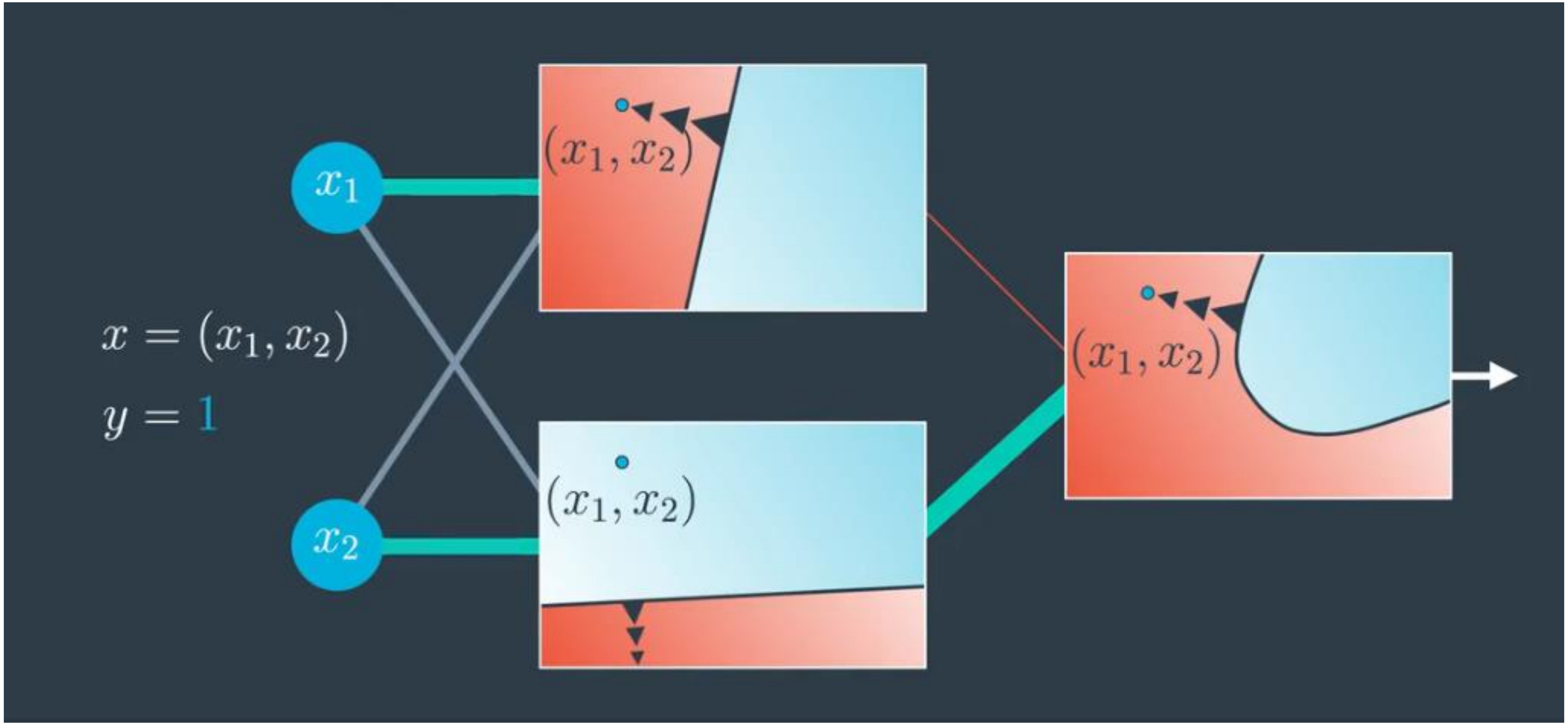
Feedforward



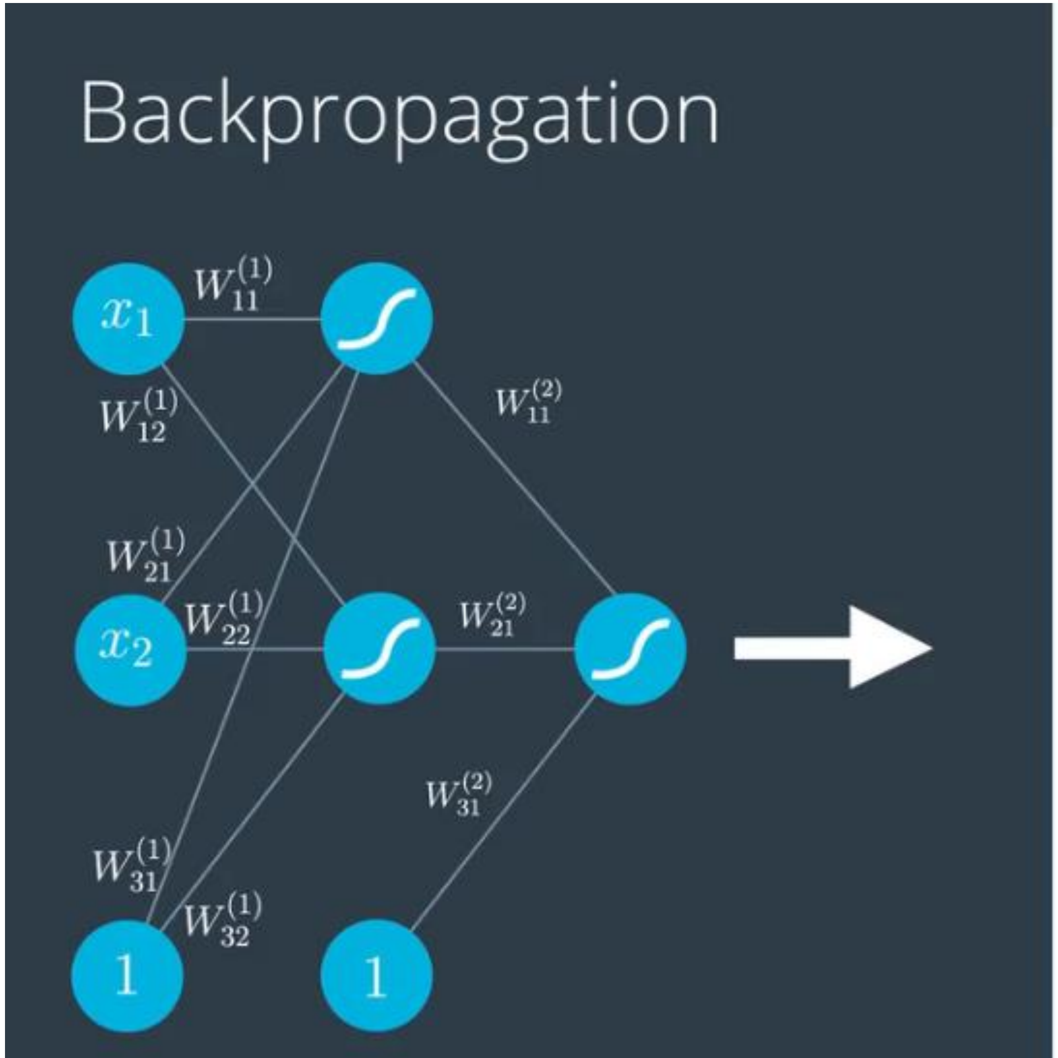
Backpropagation



Backpropagation



Backpropagation



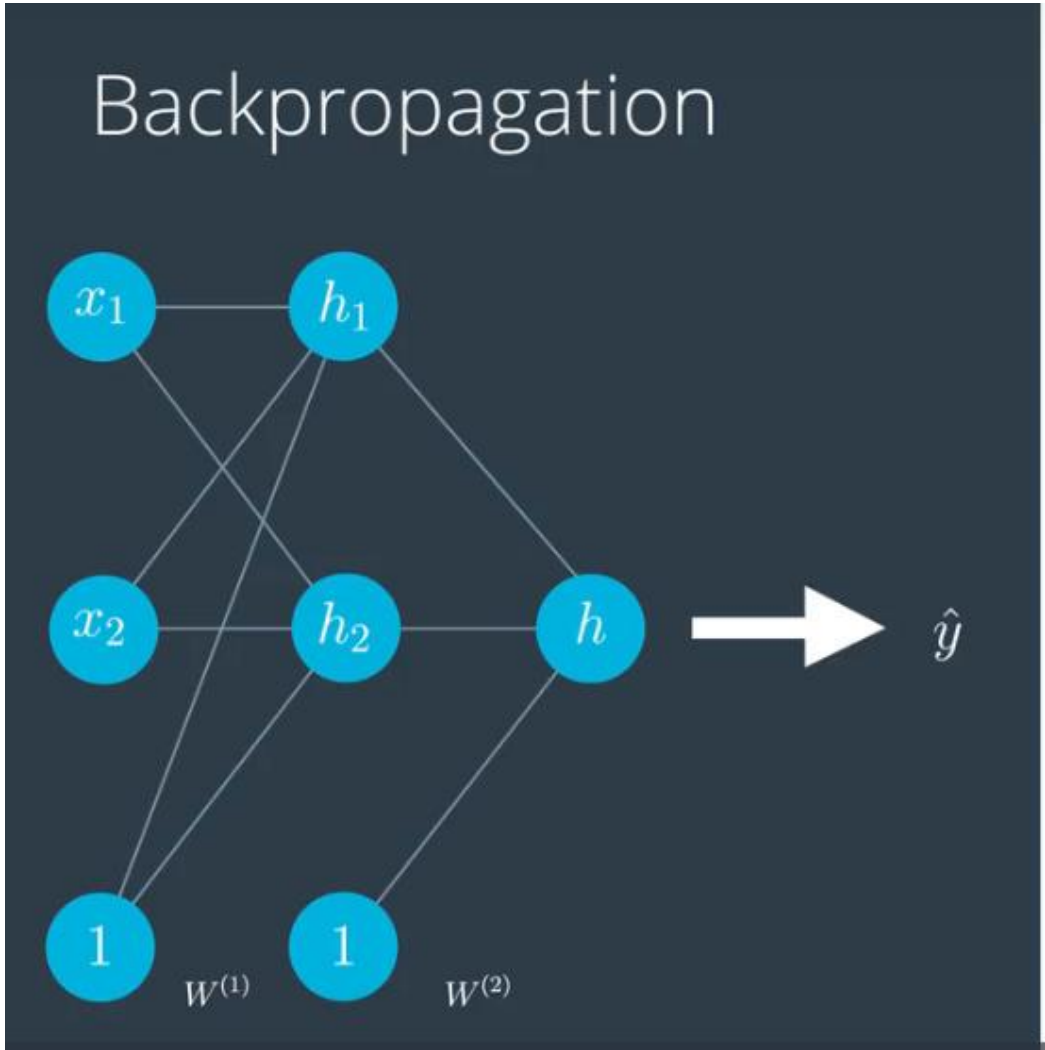
$$\hat{y} = \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

$$W^{(1)} = \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix}$$

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{11}^{(2)}} \\ \frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{21}^{(2)}} \\ \frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{31}^{(2)}} \end{pmatrix}$$

$$W_{ij}^{\prime(k)} \leftarrow W_{ij}^{(k)} - \alpha \frac{\partial E}{\partial W_{ij}^{(k)}}$$

Backpropagation



$$E(W) = -\frac{1}{m}\sum_{i=1}^{m} y_i ln(\hat{y}_i) + (1-y_i)ln(1-\hat{y}_i)$$

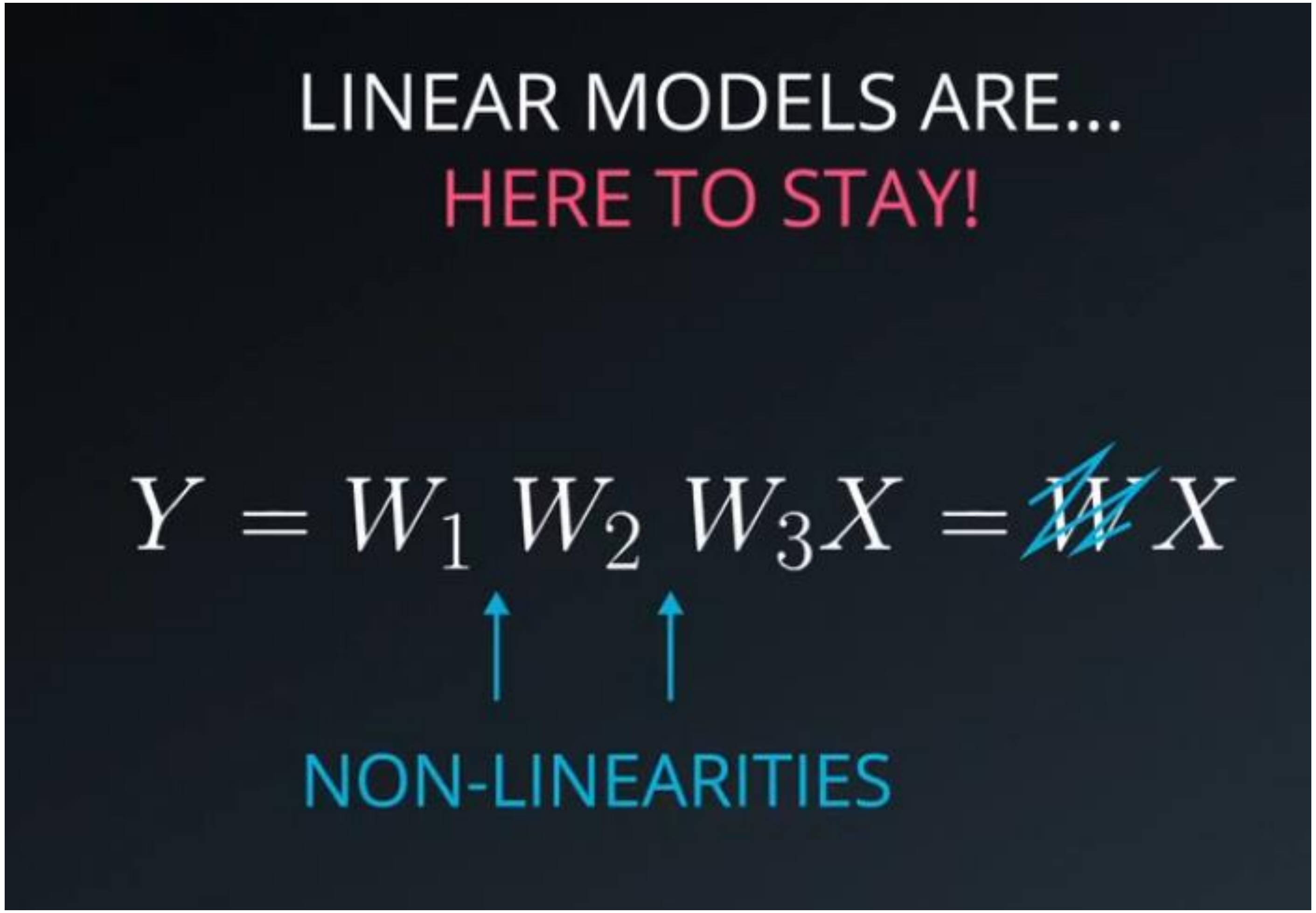
$$E(W) = E(W_{11}^{(1)}, W_{12}^{(1)}, \ldots, W_{31}^{(2)})$$

$$\nabla E = \left(\frac{\partial E}{\partial W_{11}^{(1)}}, \ldots, \frac{\partial E}{\partial W_{31}^{(2)}}\right)$$

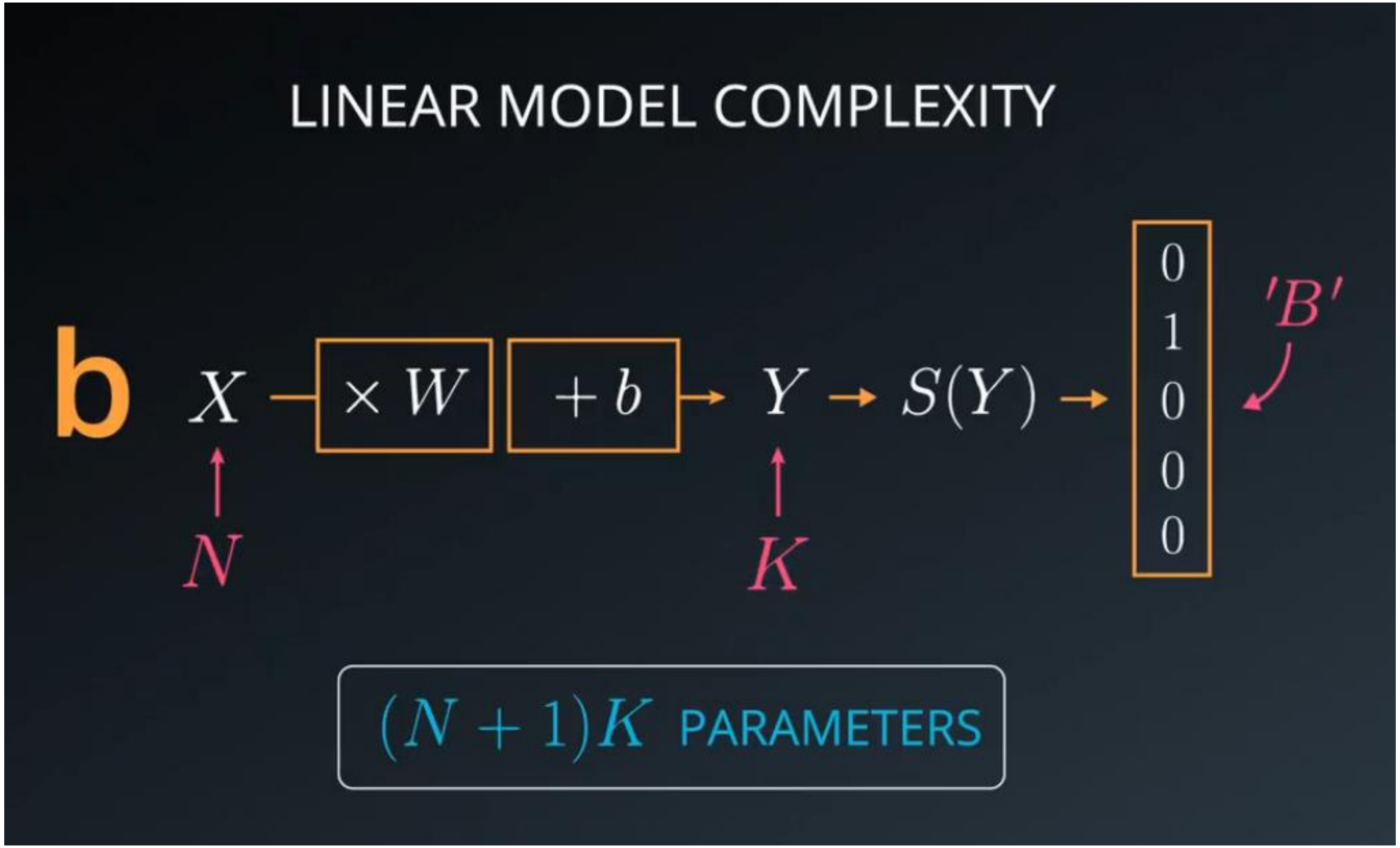
$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$

Deep Neural Networks

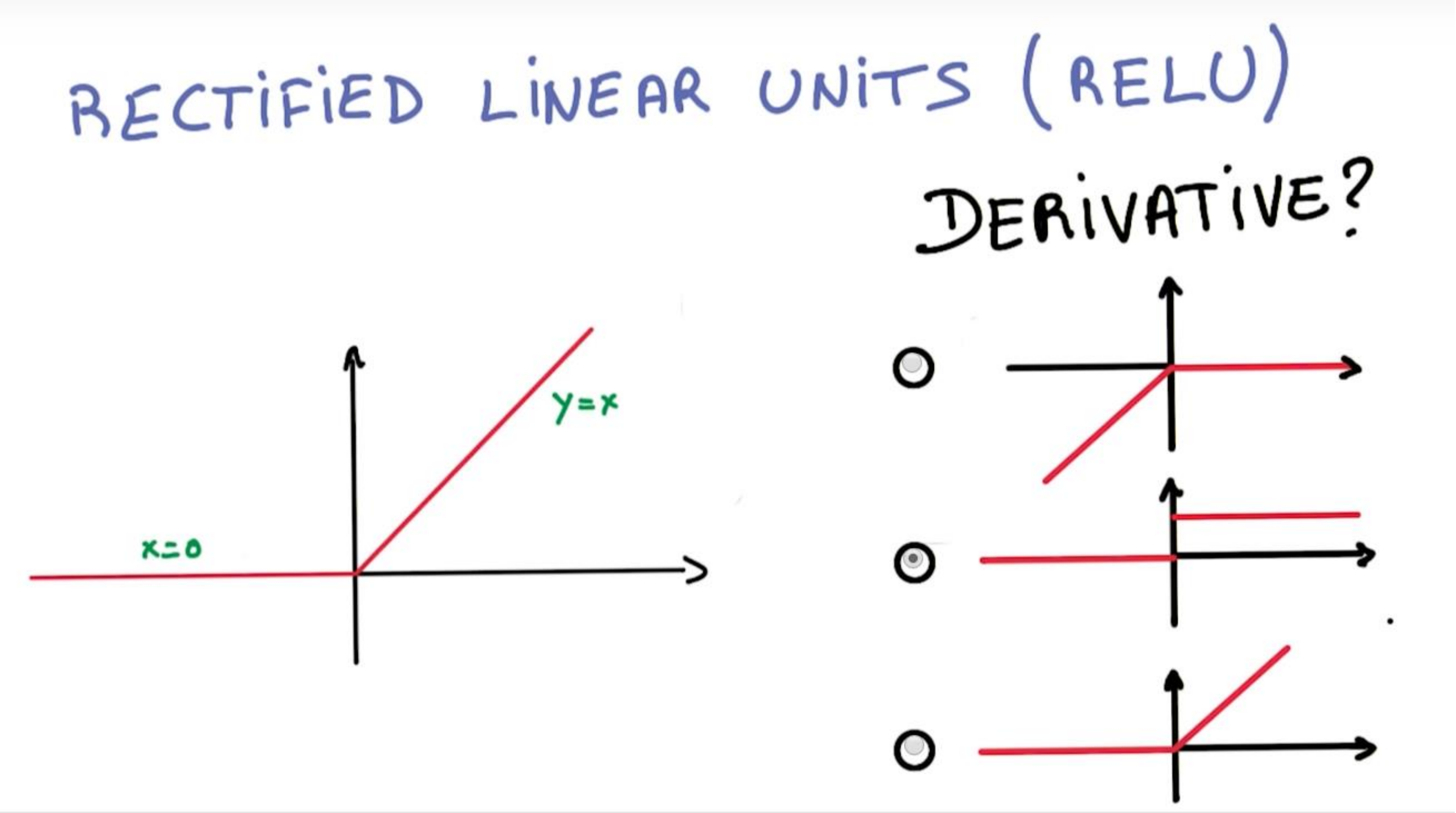
Linear Models are Limited



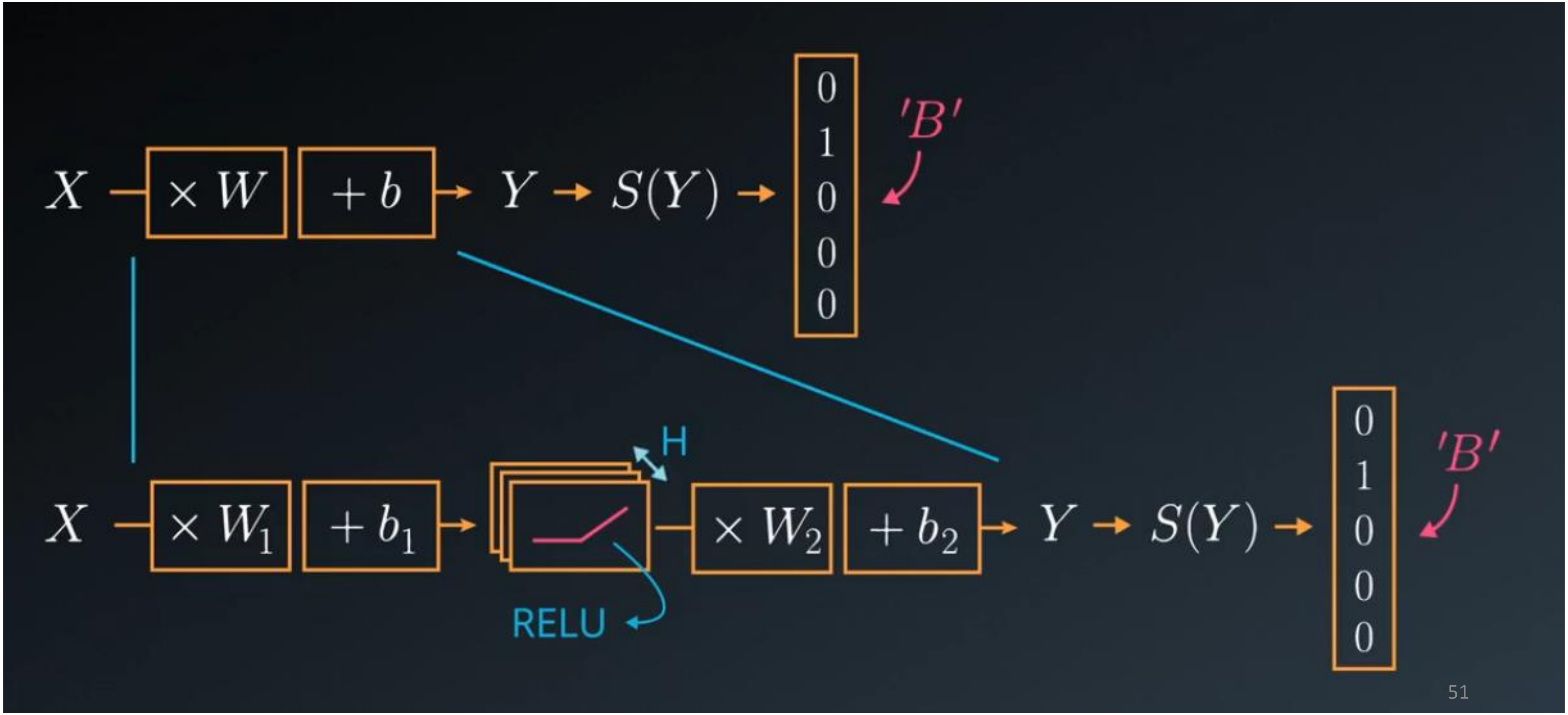
Linear Models are Limited



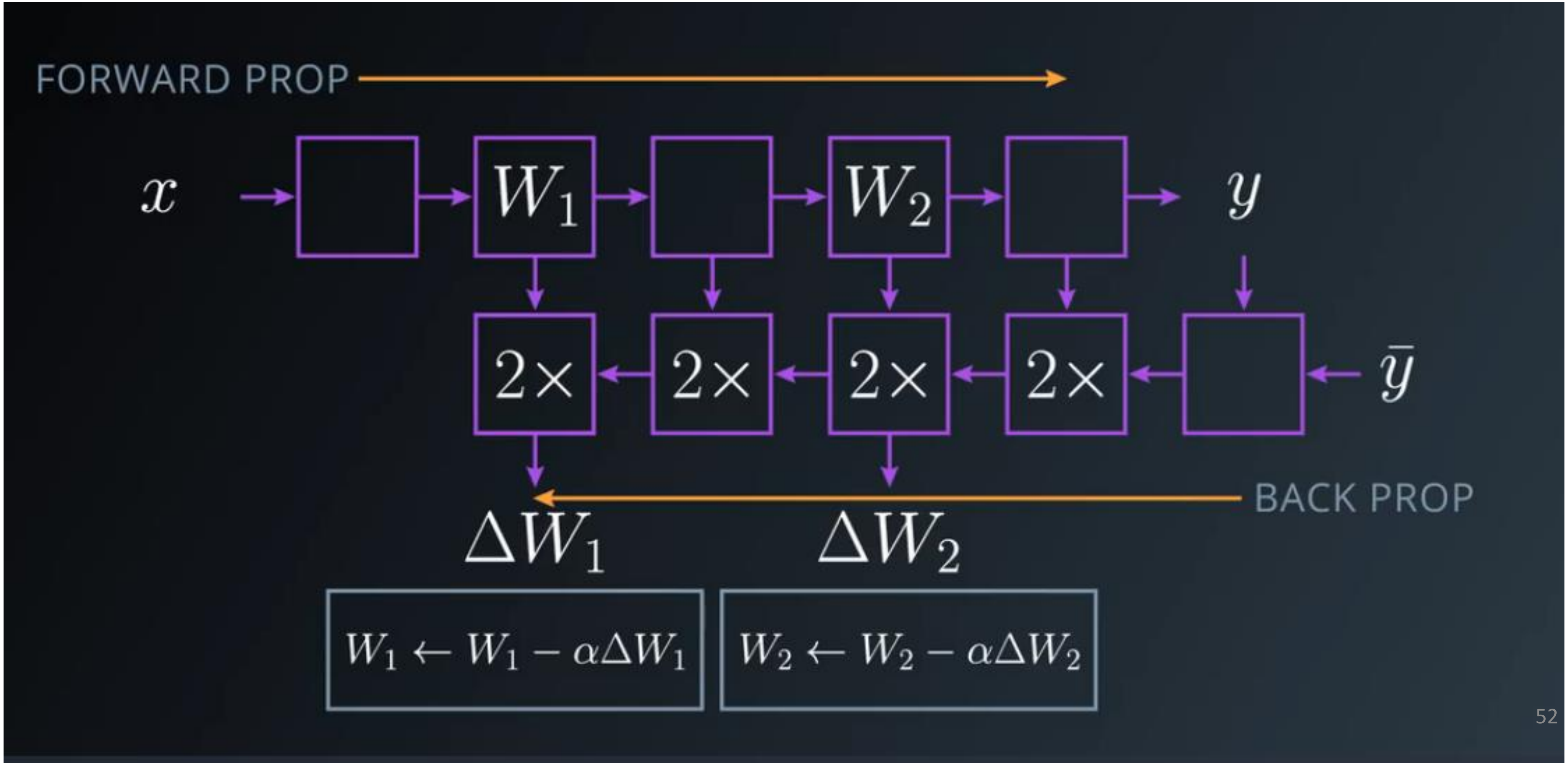
Rectified Linear Units (RELU)



Network of ReLUs



Backprop



FORWARD PROP
x
W_1
W_2
y
$2\times$
$2\times$
$2\times$
$2\times$
$\bar{y}$
BACK PROP
ΔW_1
ΔW_2
$W_1 \leftarrow W_1 - \alpha \Delta W_1$
$W_2 \leftarrow W_2 - \alpha \Delta W_2$

Deep Neural Network in TensorFlow

You've seen how to build a logistic classifier using TensorFlow. Now you're going to see how to use the logistic classifier to build a deep neural network.

Step by Step

In the following walkthrough, we'll step through TensorFlow code written to classify the letters in the MNIST database. If you would like to run the network on your computer, the file is provided **here**. You can find this and many more examples of TensorFlow at **Aymeric Damien's GitHub repository**.

Code

TensorFlow MNIST

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)
```

You'll use the MNIST dataset provided by TensorFlow, which batches and One-Hot encodes the data for you.

Learning Parameters

```
import tensorflow as tf

# Parameters
learning_rate = 0.001
training_epochs = 20
batch_size = 128  # Decrease batch size if you don't have enough memory
display_step = 1

n_input = 784  # MNIST data input (img shape: 28*28)
n_classes = 10  # MNIST total classes (0-9 digits)
```

The focus here is on the architecture of multilayer neural networks, not parameter tuning, so here we'll just give you the learning parameters.

Hidden Layer Parameters

```
n_hidden_layer = 256 # layer number of features
```

The variable `n_hidden_layer` determines the size of the hidden layer in the neural network. This is also known as the width of a layer.

Weights and Biases

```
# Store layers weight & bias
weights = {
    'hidden_layer': tf.Variable(tf.random_normal([n_input, n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_hidden_layer, n_classes]))
}
biases = {
    'hidden_layer': tf.Variable(tf.random_normal([n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Deep neural networks use multiple layers with each layer requiring it's own weight and bias. The `'hidden_layer'` weight and bias is for the hidden layer. The `'out'` weight and bias is for the output layer. If the neural network were deeper, there would be weights and biases for each additional layer.

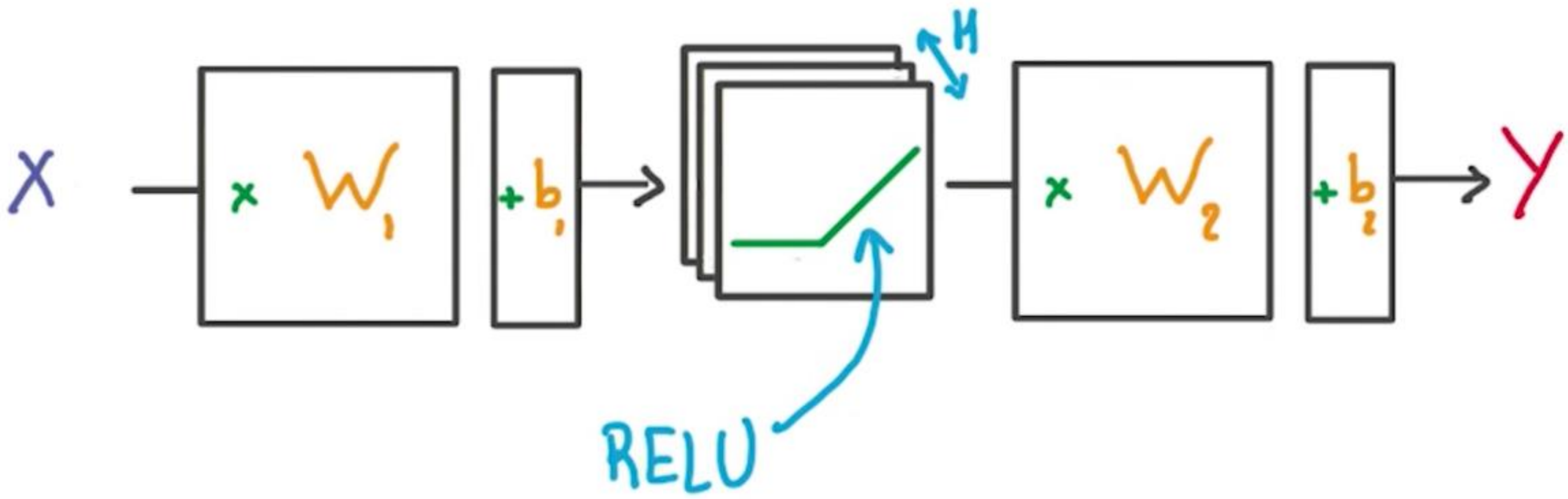
Input

```
# tf Graph input
x = tf.placeholder("float", [None, 28, 28, 1])
y = tf.placeholder("float", [None, n_classes])

x_flat = tf.reshape(x, [-1, n_input])
```

The MNIST data is made up of 28px by 28px images with a single channel. The `tf.reshape()` function above reshapes the 28px by 28px matrices in `x` into row vectors of 784px.

Multilayer Perceptron



```
# Hidden layer with RELU activation
layer_1 = tf.add(tf.matmul(x_flat, weights['hidden_layer']),\
    biases['hidden_layer'])
layer_1 = tf.nn.relu(layer_1)
# Output layer with linear activation
logits = tf.add(tf.matmul(layer_1, weights['out']), biases['out'])
```

You've seen the linear function `tf.add(tf.matmul(x_flat, weights['hidden_layer']), biases['hidden_layer'])` before, also known as `xw + b`. Combining linear functions together using a ReLU will give you a two layer network.

Optimizer

```
# Define loss and optimizer
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)
```

This is the same optimization technique used in the Intro to TensorFLow lab.

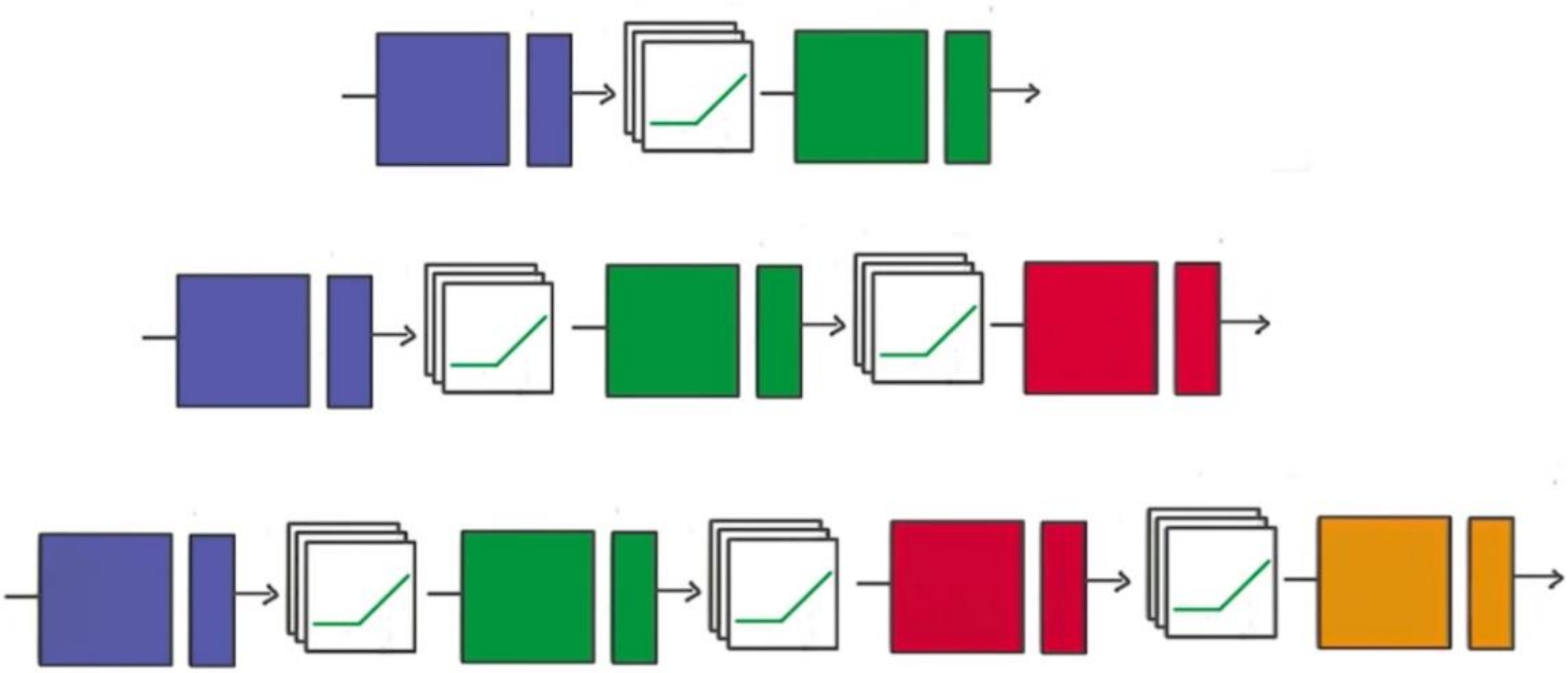
Session

```
# Initializing the variables
init = tf.global_variables_initializer()


# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    for epoch in range(training_epochs):
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            # Run optimization op (backprop) and cost op (to get loss value)
            sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
```

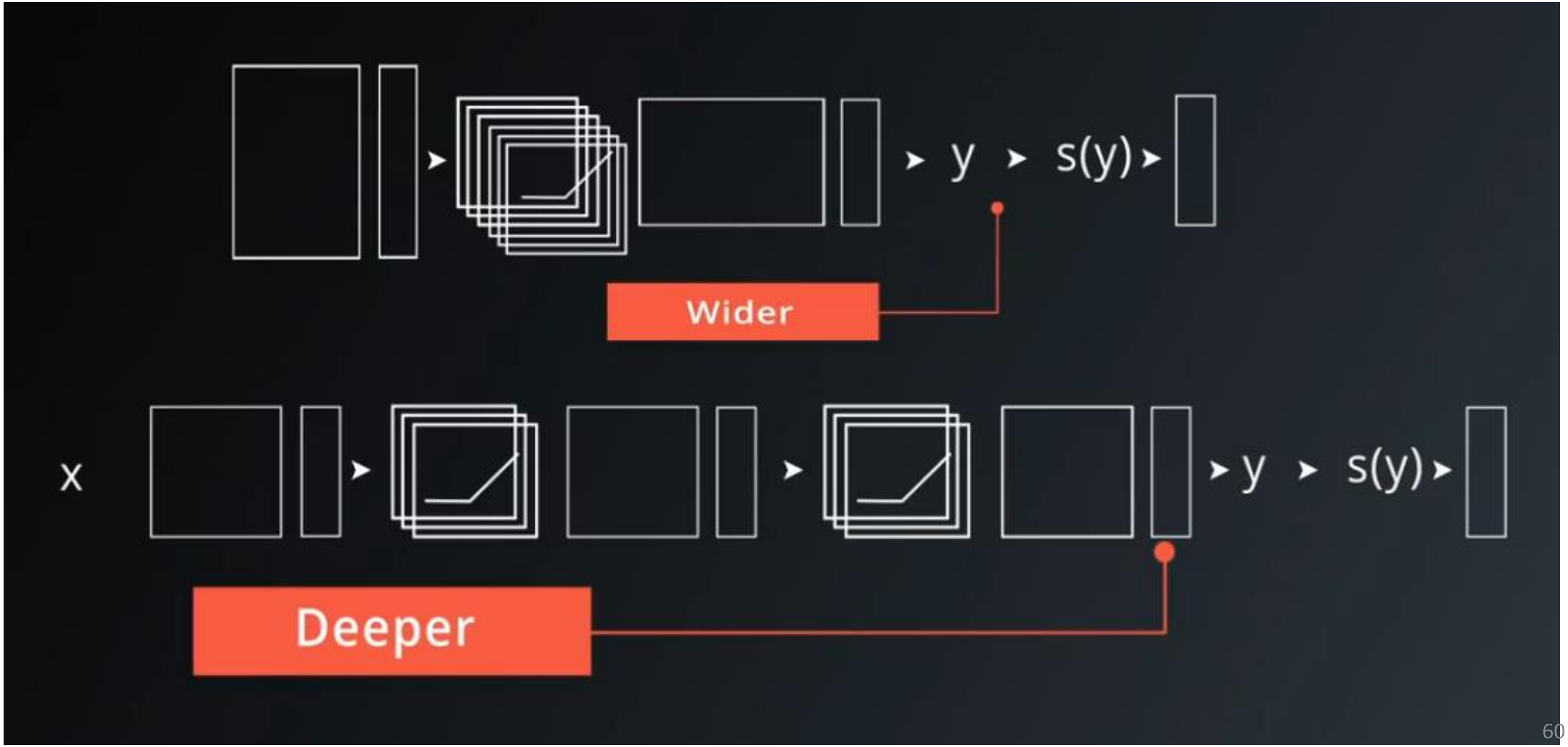
The MNIST library in TensorFlow provides the ability to receive the dataset in batches. Calling the `mnist.train.next_batch()` function returns a subset of the training data.

Deeper Neural Network

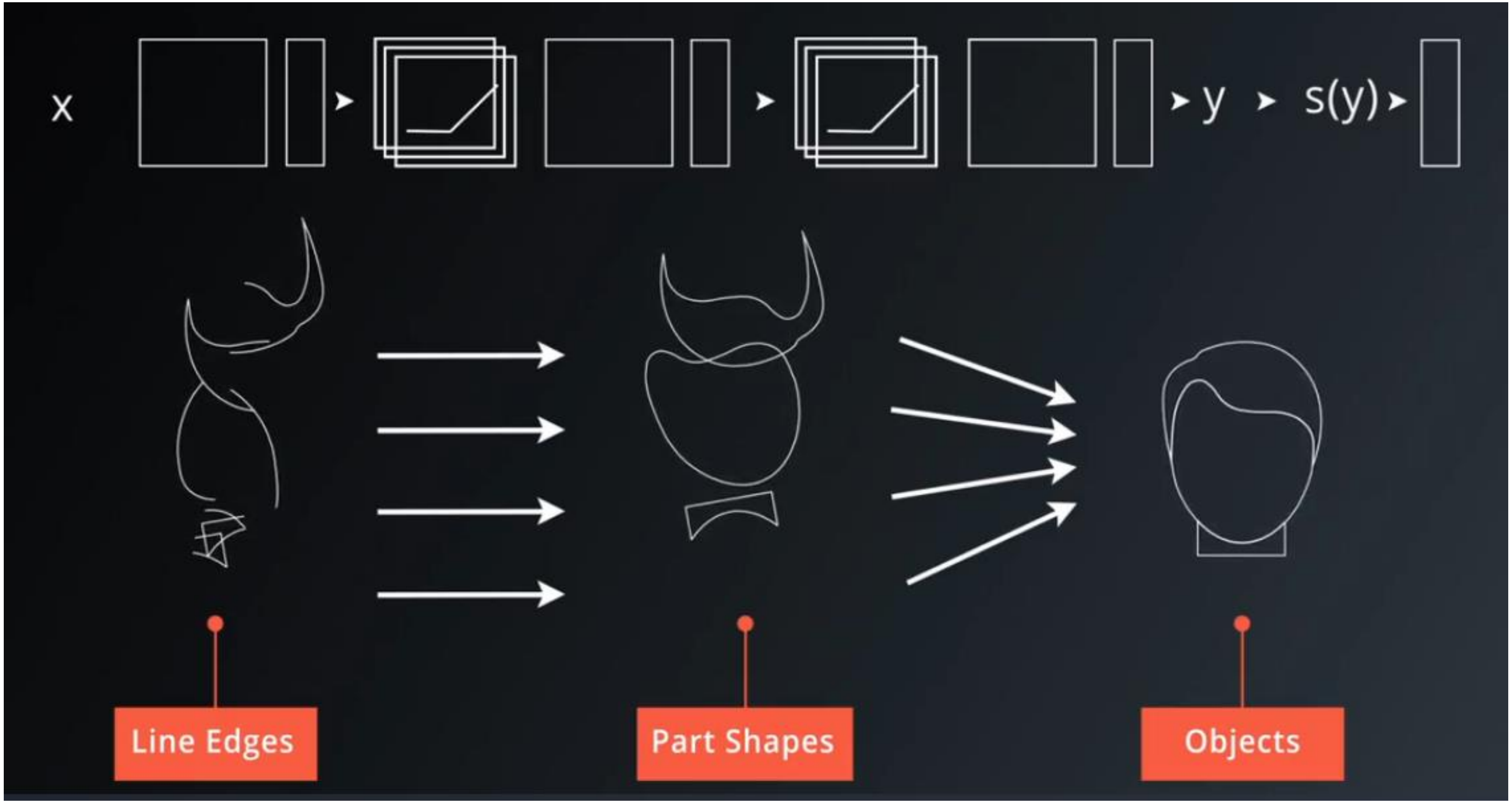


That's it! Going from one layer to two is easy. Adding more layers to the network allows you to solve more complicated problems.

Training a Deep Learning Network



Training a Deep Learning Network



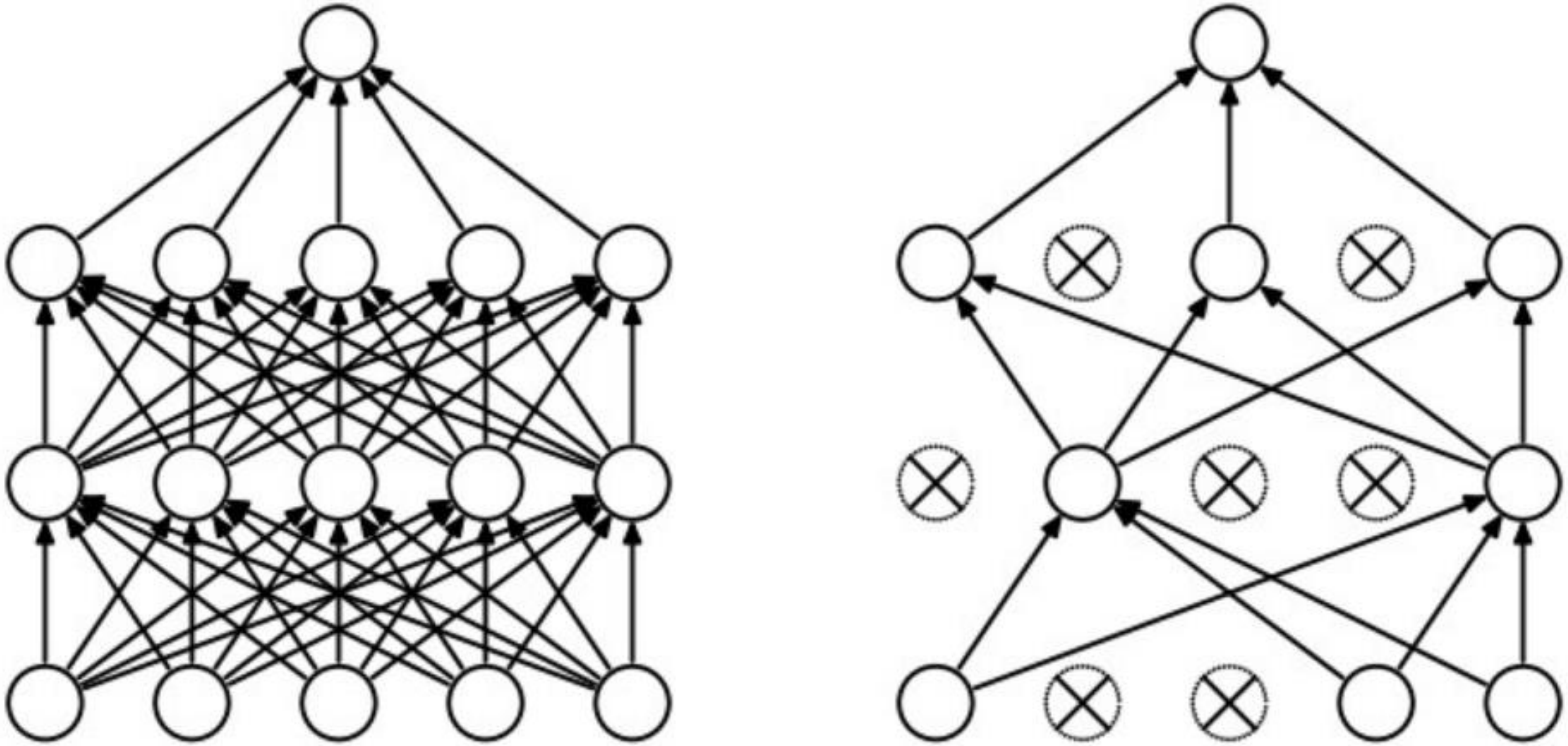
Save and Restore TensorFlow Models

- See pdf file

Finetuning: Loading the Weights and Biases into a New Model

- See pdf file

TensorFlow Dropout



(a) Standard Neural Net

(b) After applying dropout.

Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from

Dropout is a regularization technique for reducing overfitting. The technique temporarily drops units (artificial neurons) from the network, along with all of those units' incoming and outgoing connections. Figure 1 illustrates how dropout works.

TensorFlow provides the `tf.nn.dropout()` function, which you can use to implement dropout.

Let's look at an example of how to use `tf.nn.dropout()`.

```
keep_prob = tf.placeholder(tf.float32) # probability to keep units

hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)

logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
```

The code above illustrates how to apply dropout to a neural network.

The `tf.nn.dropout()` function takes in two parameters:

1. `hidden_layer`: the tensor to which you would like to apply dropout
2. `keep_prob`: the probability of keeping (i.e. *not* dropping) any given unit

`keep_prob` allows you to adjust the number of units to drop. In order to compensate for dropped units, `tf.nn.dropout()` multiplies all units that are kept (i.e. *not* dropped) by `1/keep_prob`.

During training, a good starting value for `keep_prob` is `0.5`.

During testing, use a `keep_prob` value of `1.0` to keep all units and maximize the power of the model.