INSTITUTO
METRÓPOLE
DIGITAL

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

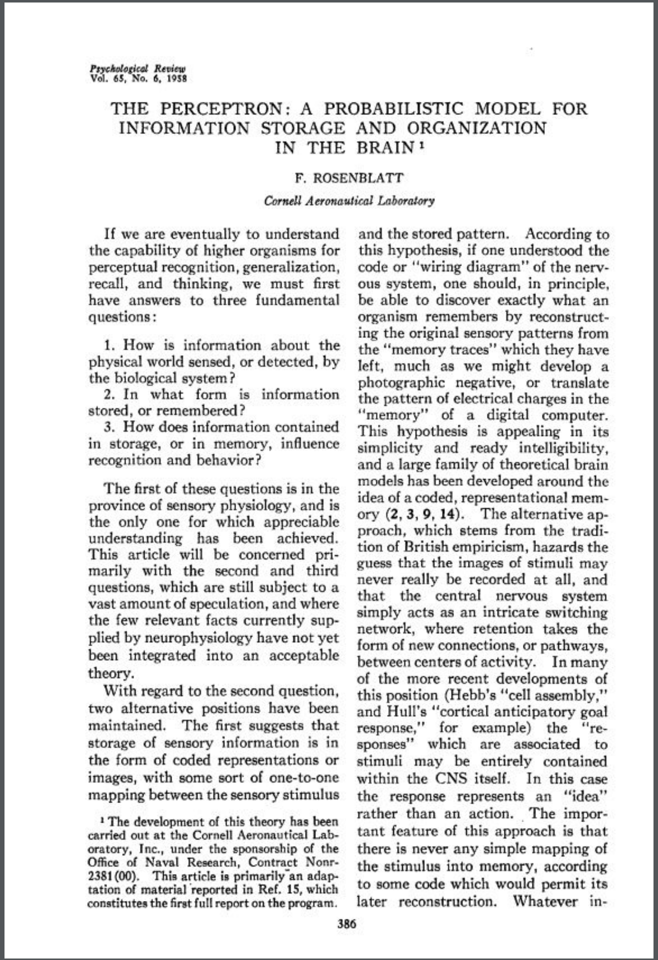
Introduction to Deep Learning and TensorFlow

Lesson #02

Table of Contents

The Perceptron

The structural building block of deep learning



Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

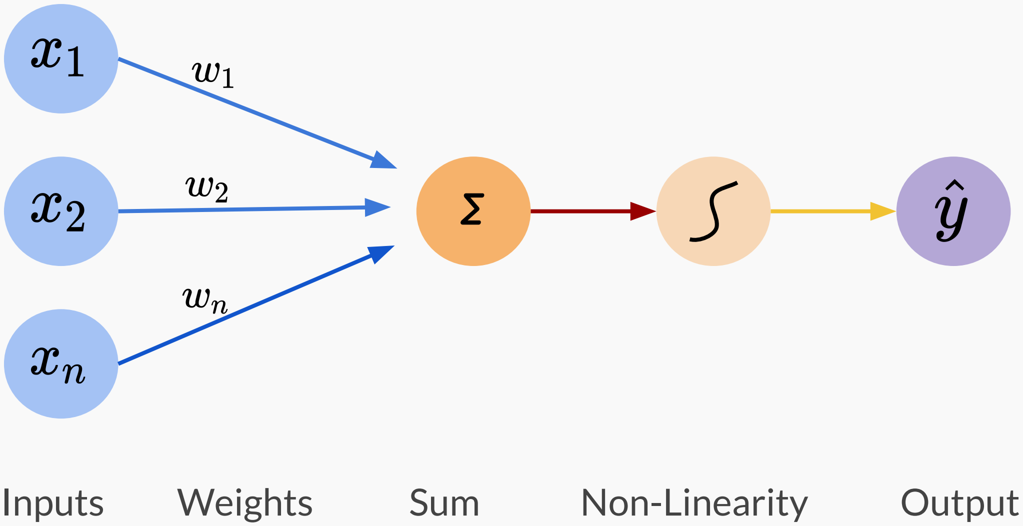
1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first of these questions is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved. This article will be concerned primarily with the second and third questions, which are still subject to a vast amount of speculation, and where the few relevant facts currently supplied by neurophysiology have not yet been integrated into an acceptable theory.

With regard to the second question, two alternative positions have been maintained. The first suggests that storage of sensory information is in the form of coded representations or images, with some sort of one-to-one mapping between the sensory stimulus and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain models has been developed around the idea of a coded, representational memory (**2**, **3**, **9**, **14**). The alternative approach, which stems from the tradition of British empiricism, hazards the guess that the images of stimuli may never really be recorded at all, and that the central nervous system simply acts as an intricate switching network, where retention takes the form of new connections, or pathways, between centers of activity. In many of the more recent developments of this position (Hebb's "cell assembly," and Hull's "cortical anticipatory goal response," for example) the "responses" which are associated to stimuli may be entirely contained within the CNS itself. In this case the response represents an "idea" rather than an action. The important feature of this approach is that there is never any simple mapping of the stimulus into memory, according to some code which would permit its later reconstruction. Whatever in-

[1] The development of this theory has been carried out at the Cornell Aeronautical Laboratory, Inc., under the sponsorship of the Office of Naval Research, Contract Nonr-2381(00). This article is primarily an adaptation of material reported in Ref. 15, which constitutes the first full report on the program.

The Perceptron: Forward Propagation



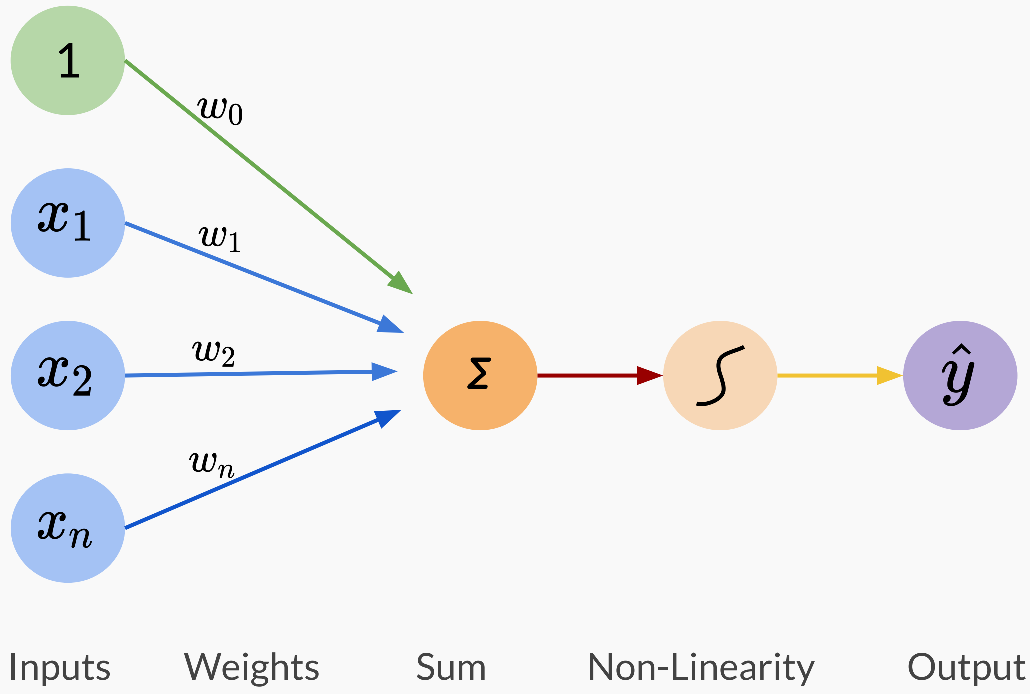
Linear combination of inputs

Output

$$\hat{y} = g\left(\sum_{i=1}^{n} x_i w_i\right)$$

Non-linear activation function

The Perceptron: Forward Propagation



Output

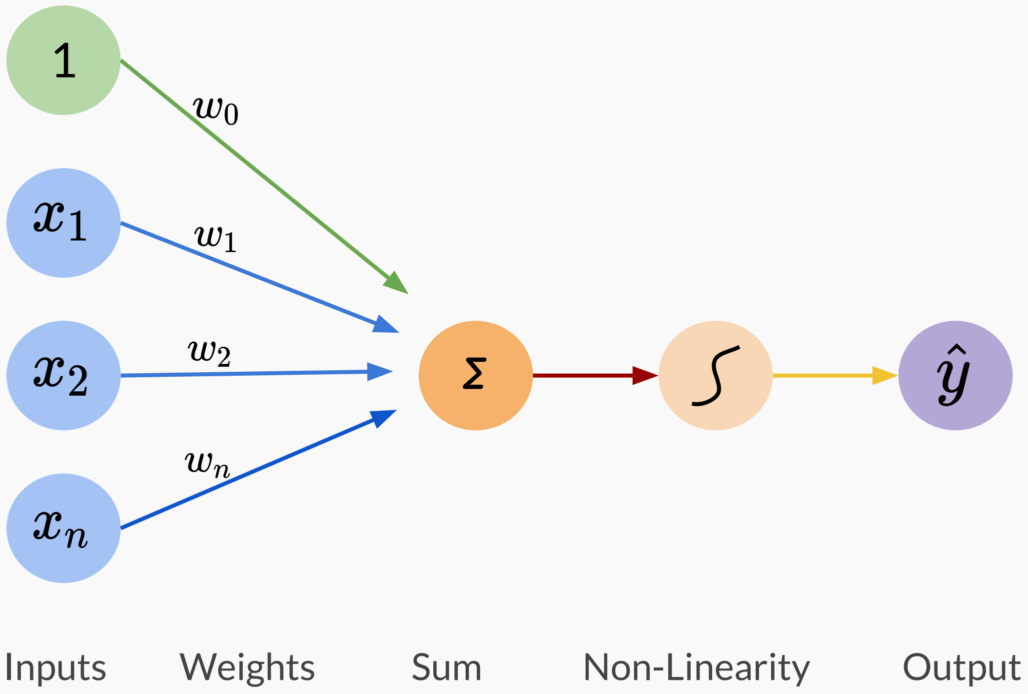
Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} x_i w_i\right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation

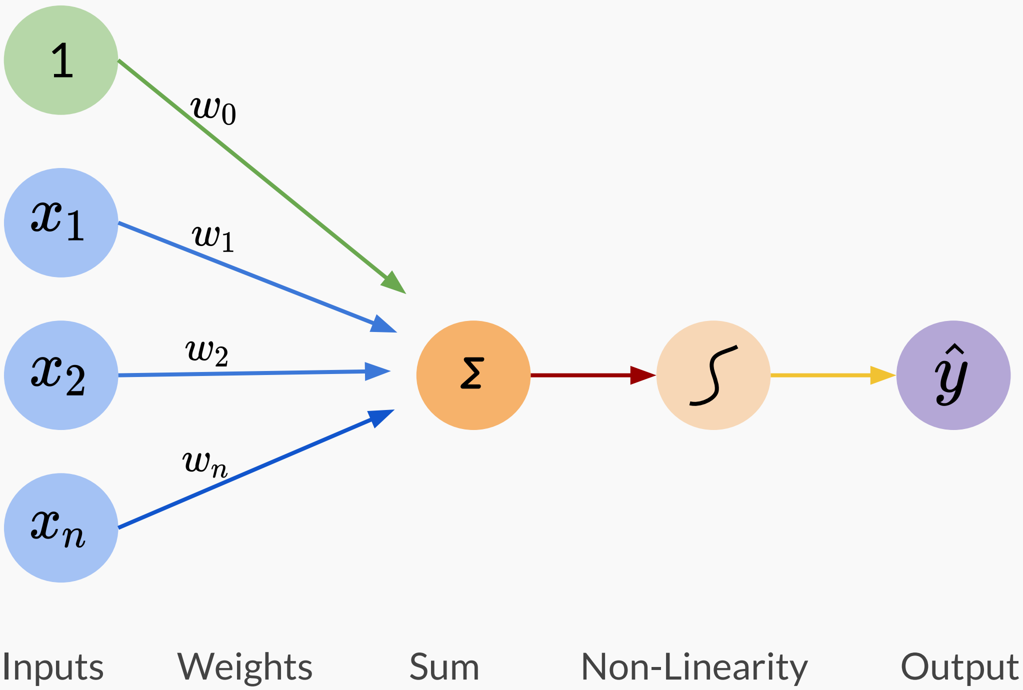


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} x_i w_i\right)$$

$$\hat{y} = g\left(w_0 + X^T W\right)$$

where: $X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ and $W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$

The Perceptron: Forward Propagation

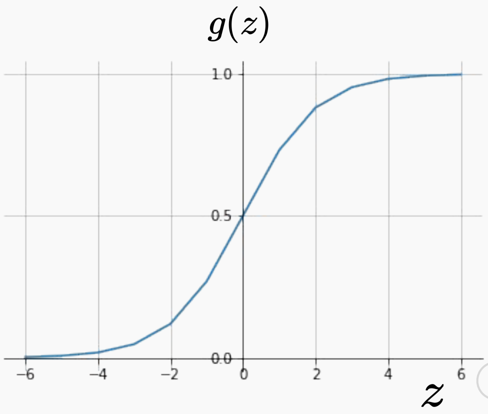


Activation Functions

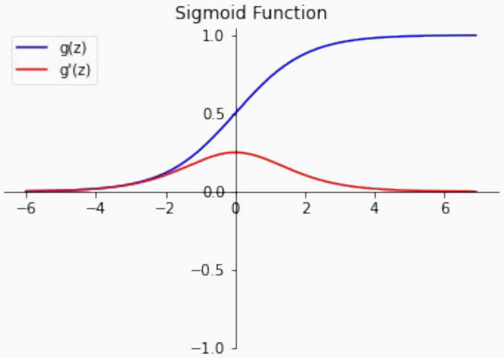
$$\hat{y} = \boxed{g}\left(w_0 + X^T W\right)$$

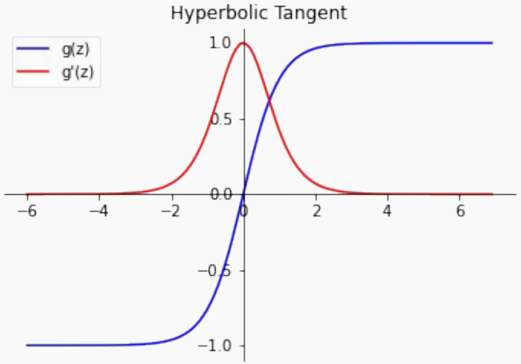
Example: sigmoid function

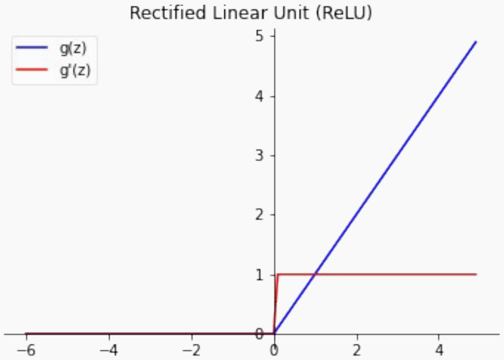
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Common Activation Functions







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

$$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$
$$g'(z) = 1 - g(z)^2$$

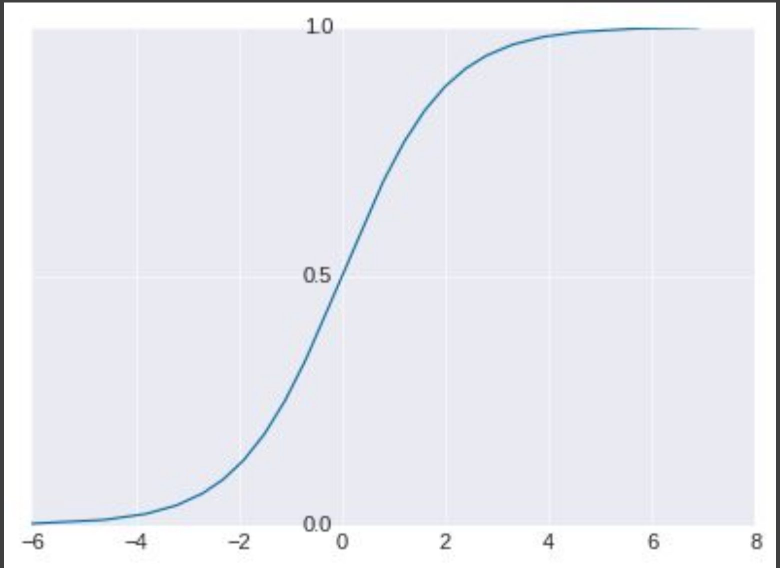
$$g(z) = max(0, z)$$
$$g'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
x = tf.constant(2, dtype=tf.float32)
tf.math.sigmoid(x).numpy()
```

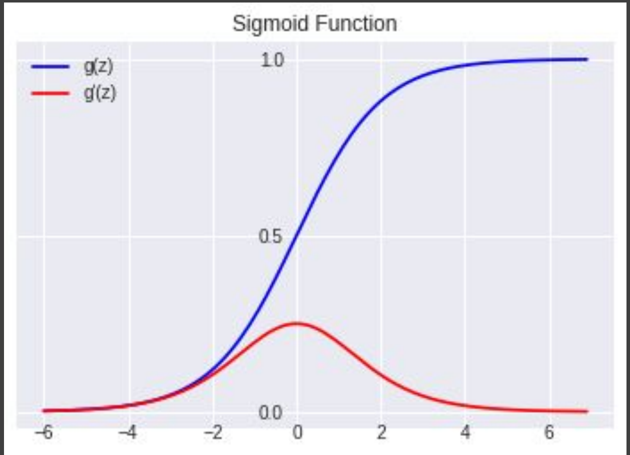
```
>> 0.8807971
```

```
# create a figure
fig, ax = plt.subplots(1,1,figsize=(6,4))

# range of values from -6 to 7
values =  tf.range(-6,7,0.1,dtype=tf.float32)

# calculate sigmoid function for all values
sigmoid_values = tf.math.sigmoid(values)

# plot values
ax.plot(values.numpy(),sigmoid_values.numpy())
```



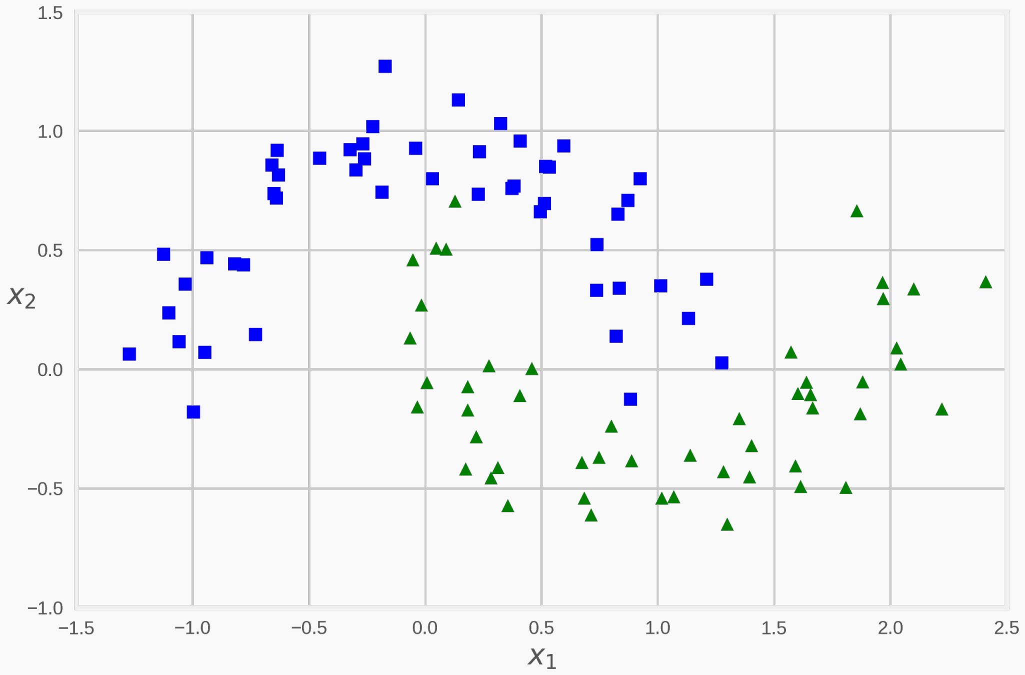
```
# create a range of values
values =  tf.range(-6,7,1,dtype=tf.float32)
# calculates the derivative of sigmoid function
with tf.GradientTape() as tape:
  # Start recording the history of operations applied to 'values'
  tape.watch(values)
  sigmoid_values = tf.math.sigmoid(values)

  # What's the gradient of `sigmoid_values` with respect to `values`?
  derivative_sigmoid = tape.gradient(sigmoid_values, values)
  print(derivative_sigmoid)
```

```
>>> tf.Tensor(
[0.00246653 0.00664812 0.01766273 0.04517666 0.10499357 0.19661194
 0.25       0.19661193 0.10499357 0.04517666 0.01766273 0.00664809
 0.00246653], shape=(13,), dtype=float32)
```



Importance of Activation Functions

The purpose of activation functions is to introduce **non-linearities** into the network

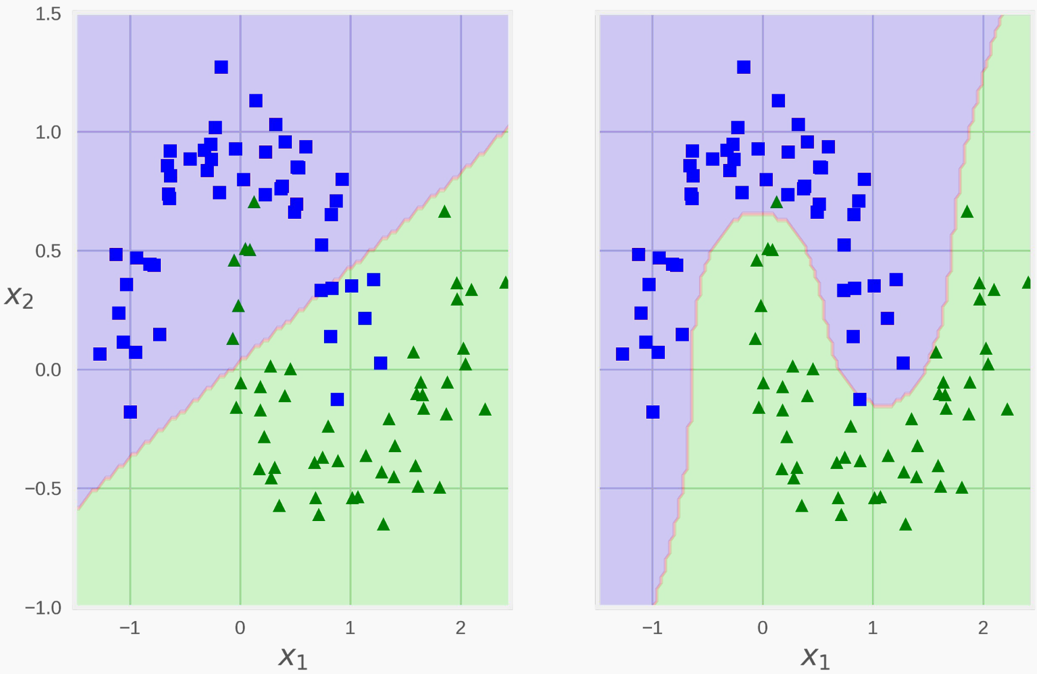


What if we wanted to build a neural network to distinguish blue vs green points?

Importance of Activation Functions

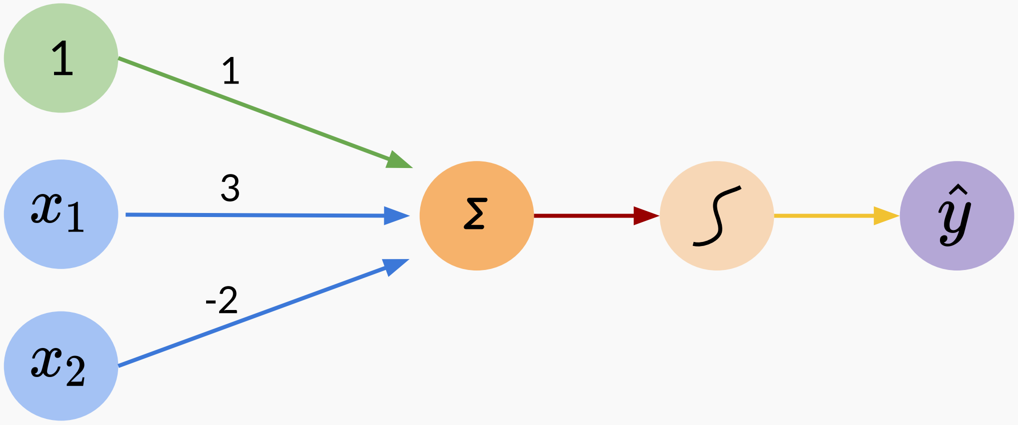
The purpose of activation functions is to introduce **non-linearities** into the network

Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

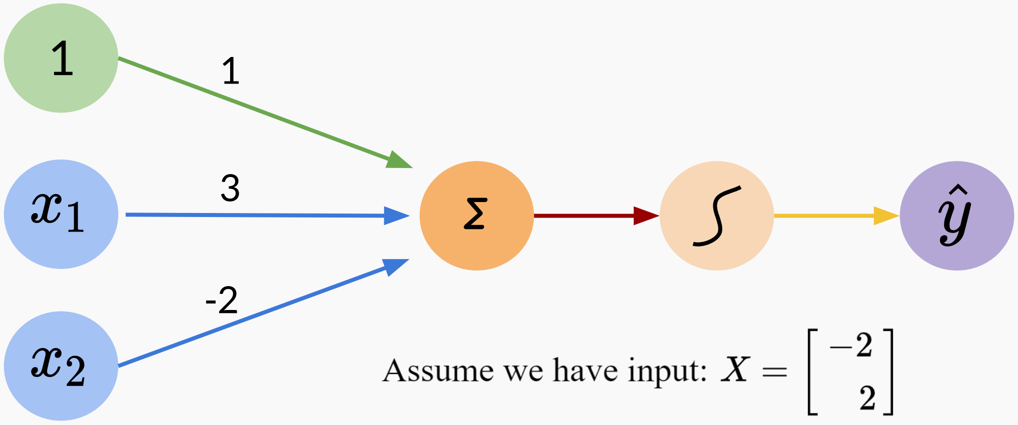
The Perceptron: Example



We have $w_0 = 1$ *and* $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\left(w_0 + X^T W\right) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ &= g\ \underbrace{(1 + 3x_1 - 2x_2)}_{\text{This is just a line in 2D!}} \end{aligned}$$

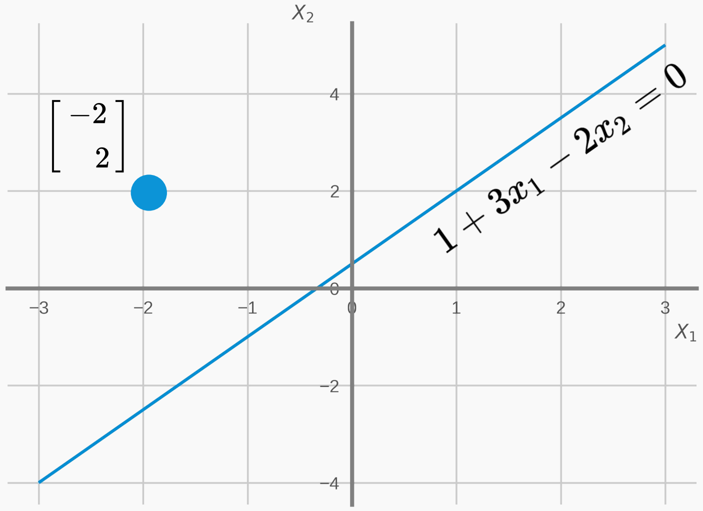
The Perceptron: Example



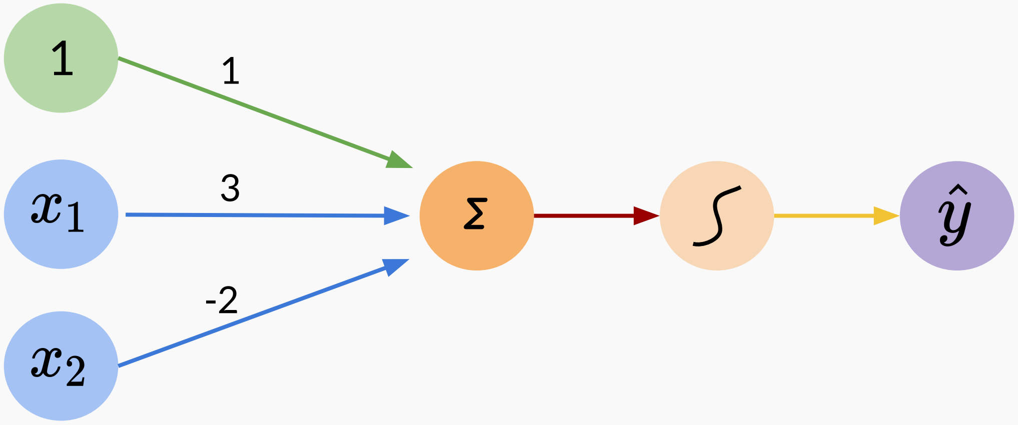
Assume we have input: $X = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\left(1 + (3 \times -2) - (2 \times 2)\right) \\ &= g(-9) \approx 0.00012338161 \end{aligned}$$

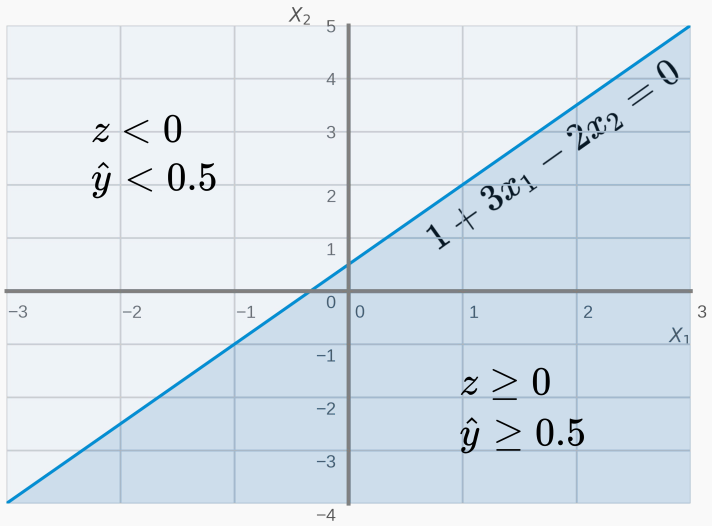
$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$



The Perceptron: Example

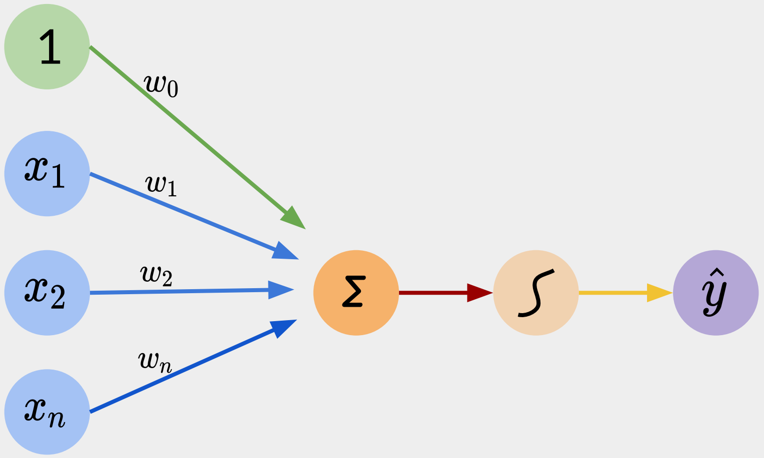


$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$



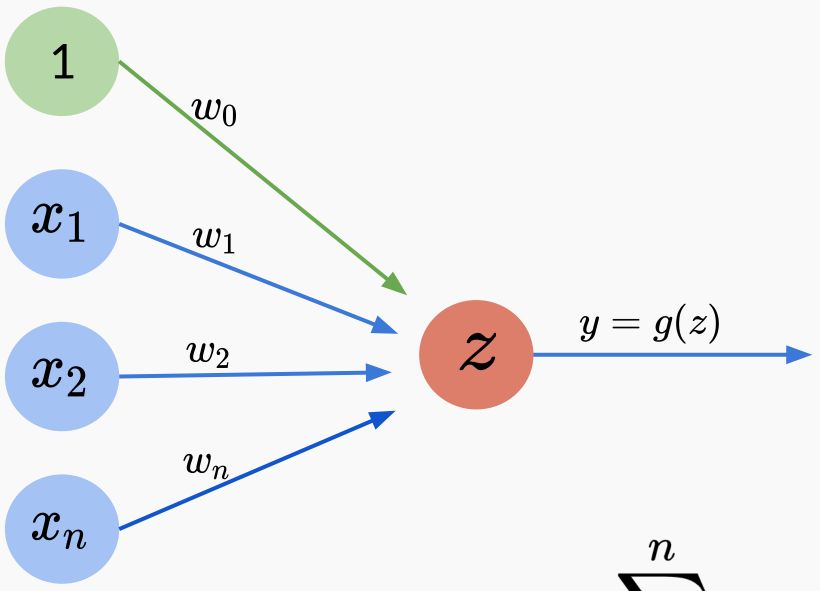
Building Neural Networks

With Perceptrons



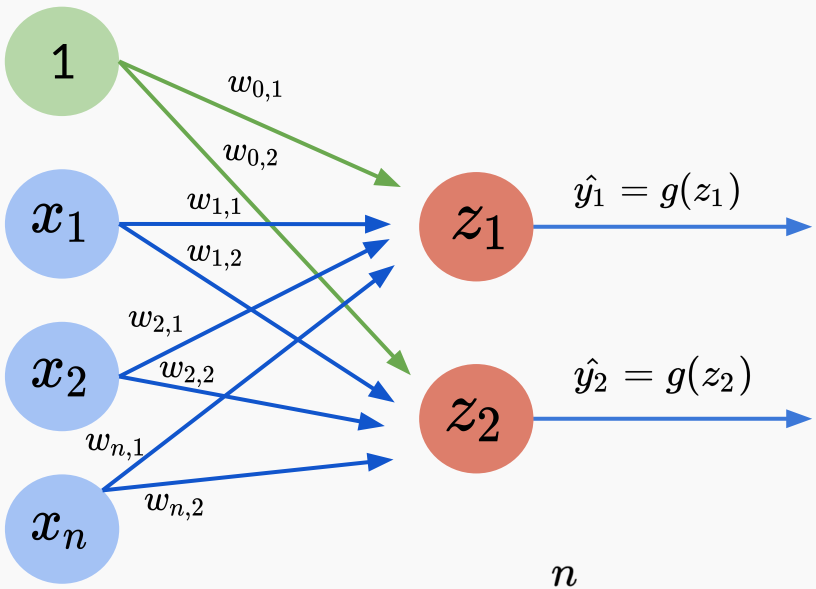
Lesson #02

The Perceptron: Simplified



$$z = w_0 + \sum_{j=1}^{n} x_j w_j$$

Multi Output Perceptron



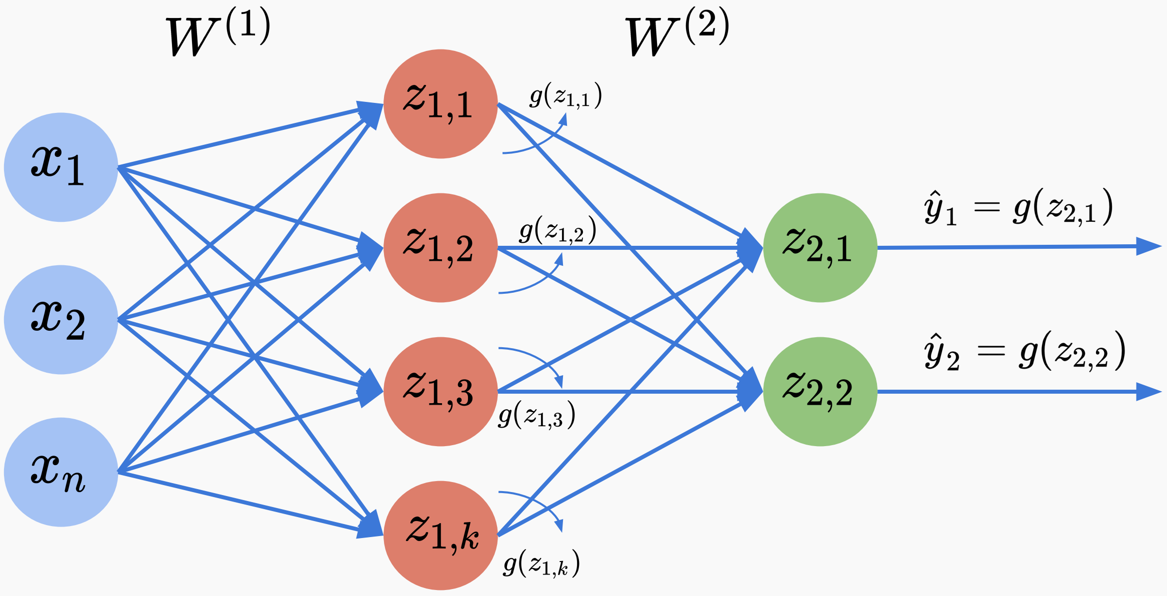
$$z_i = w_{0,i} + \sum_{j=1}^{n} x_j w_{j,i}$$

Dense layer from scratch

```python
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, units=32):
    super(MyDenseLayer, self).__init__()
    self.units = units

  def build(self, input_shape):
    input_dim = int(input_shape[-1])
    # Initialize weights and bias
    self.W = self.add_weight("weight",
                             shape=[input_dim,self.units],
                             initializer='random_normal')
    self.b = self.add_weight("bias",
                             shape=[1,self.units],
                             initializer='zeros')

  def call(self, x):
    # Forward propagation
    z = tf.matmul(x,self.W) + self.b

    # Feed through a non-linear activation
    y = tf.sigmoid(z)

    return y
```

```python
layer = tf.keras.layers.Dense(units=2, activation="sigmoid")
```

Single Hidden Layer Neural Network



$$z_{1,i} = W_{0,i}^{(1)} + \sum_{j=1}^{n} x_j w_{j,i}^{(1)}$$

$$z_{2,i} = W_{0,i}^{(2)} + \sum_{j=1}^{k} g(z_{1,j}) w_{j,i}^{(2)}$$

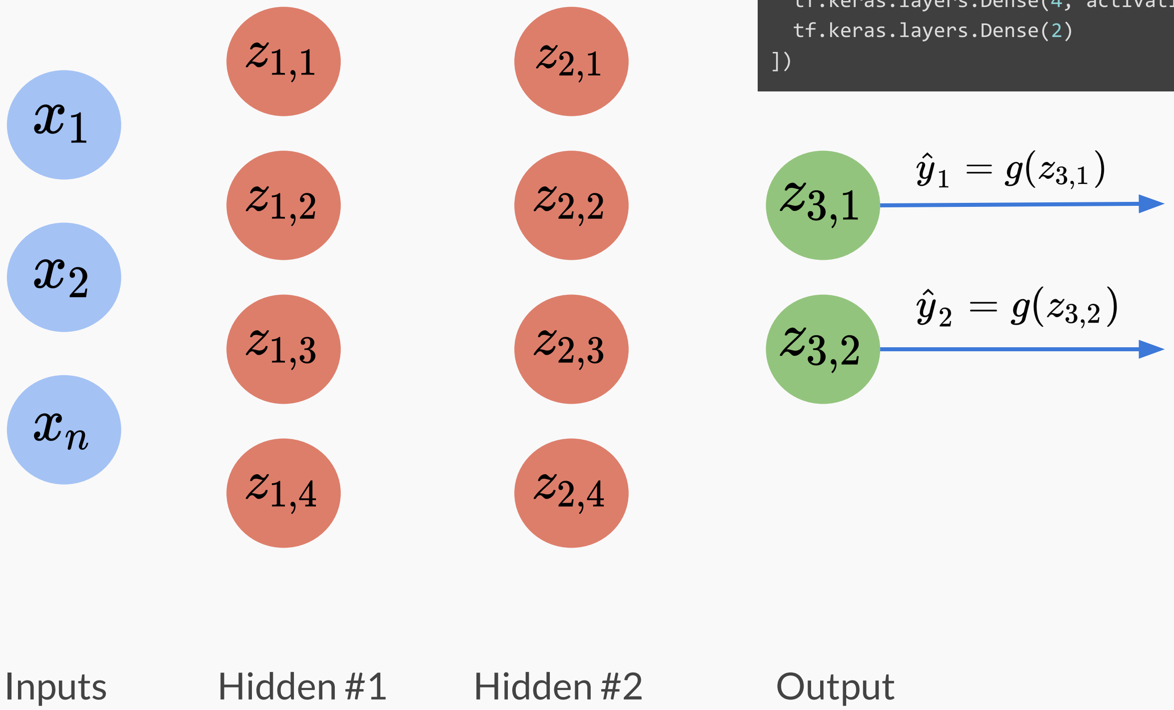
Inputs

Hidden

Output

Multi Hidden Layer Neural Network

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(4, activation=tf.nn.sigmoid),
  tf.keras.layers.Dense(4, activation=tf.nn.sigmoid),
  tf.keras.layers.Dense(2)
])
```



Applying Neural Networks

Cat vs Non Cat

Example Problem

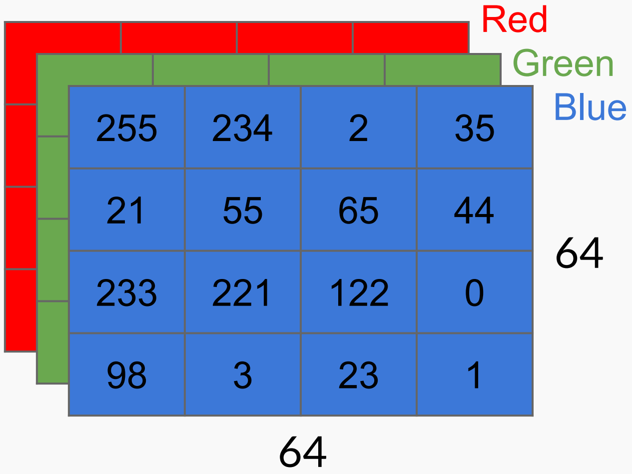
Binary Classification

64

64

Number of pixels

$$n = 64 \times 64 \times 3 = 12288$$

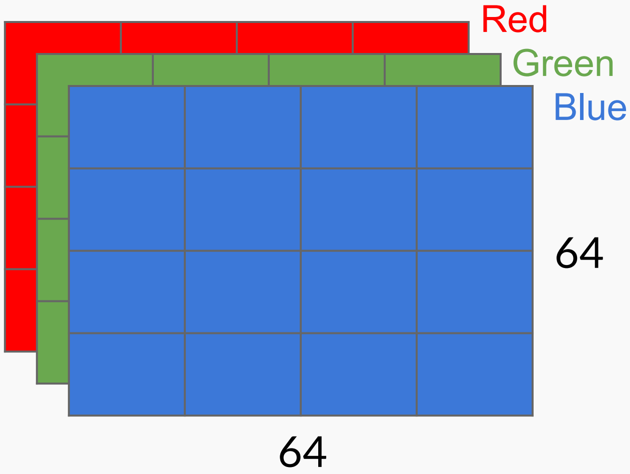


Example Problem

Binary Classification

Number of pixels

$$n = 64 \times 64 \times 3 = 12288$$



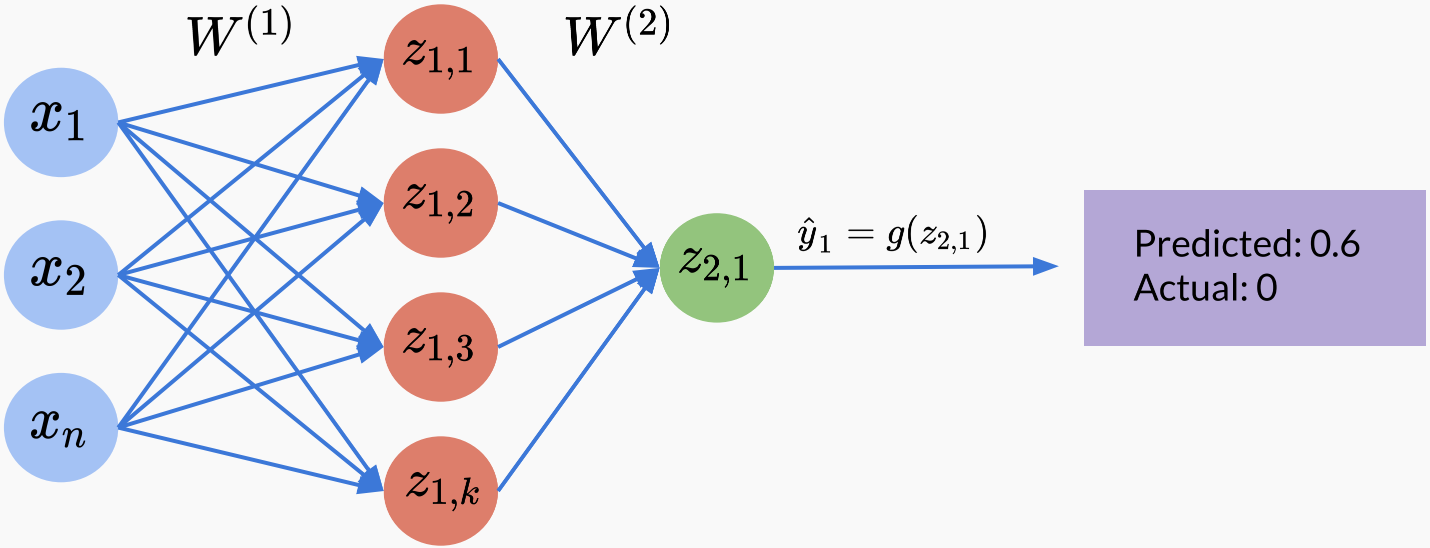
$$\begin{array}{cc} & y \\ \begin{matrix} Y^{(1)} \\ Y^{(2)} \\ Y^{(3)} \\ \vdots \\ Y^{(m)} \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \end{array}$$

$$Y^{(i)} = \begin{cases} 0 & \text{if y is not a cat} \\ 1 & \text{if y is a cat} \end{cases}$$

$$\begin{array}{cc} & \boxed{\begin{matrix} x_1 & x_2 & x_3 & \dots & x_{12288} \end{matrix}} \\ \begin{matrix} X^{(1)} \\ X^{(2)} \\ X^{(3)} \\ \vdots \\ X^{(m)} \end{matrix} & \begin{bmatrix} 25 & 34 & 2 & \dots & 37 \\ \vdots & \vdots & \vdots & & \vdots \\ \\ \vdots & \vdots & \vdots & & \vdots \\ 200 & 10 & 32 & & 3 \end{bmatrix} \end{array}$$

Quantifying Loss

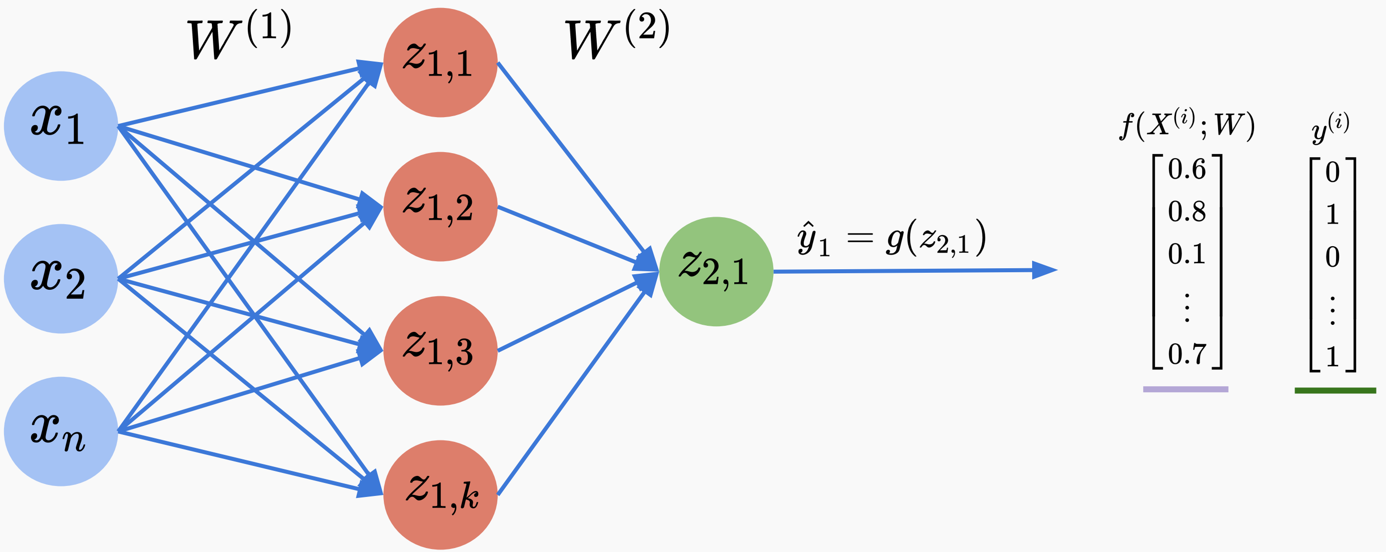
The **loss** of our network measure the cost incurred from incorrect predictions



$$\mathcal{L}(\underbrace{f(X^{(i)}; W)}_{\text{Predict}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss

The **empirical loss** measures the total loss over our entire dataset



Also known as:
- Objective function
- Cost function
- Empirical Risk

$$J(W) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\underline{f(X^{(i)};W)}, \underline{y^{(i)}} \right)$$

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1

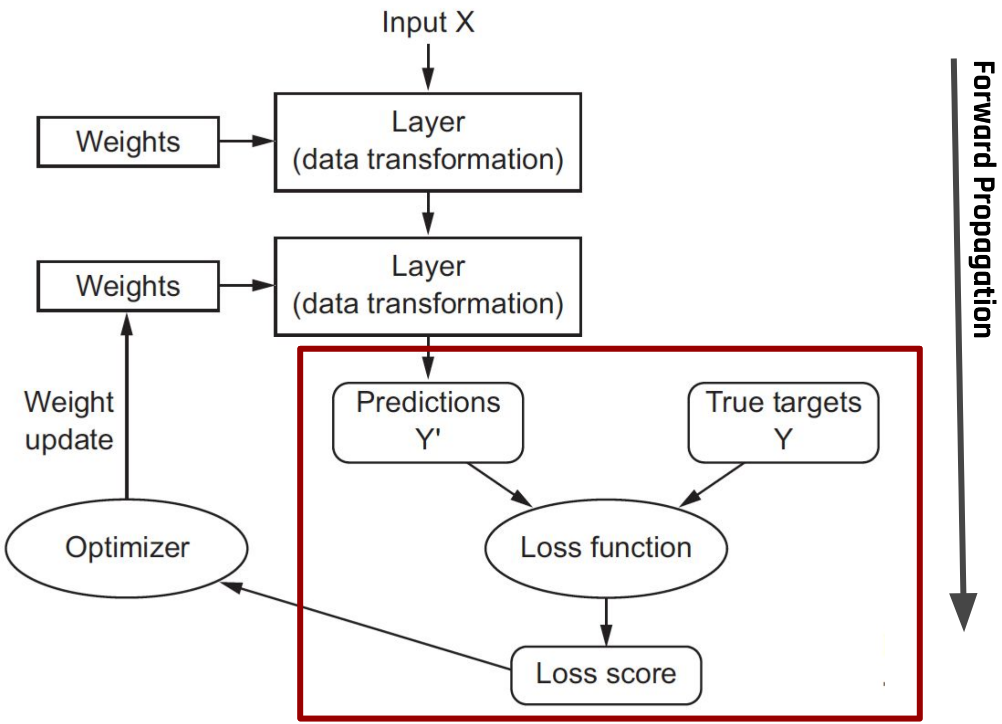
$$\mathcal{L}\left(\underline{f(X^{(i)}; W)}, \underline{y^{(i)}}\right) = \mathcal{L}\left(\underline{\hat{y}^{(i)}}, \underline{y^{(i)}}\right)$$

$$\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) = \underline{-y^{(i)}}\,\underline{log(\hat{y}^{(i)})} - \underline{(1 - y^{(i)})}\underline{log(1 - \hat{y}^{(i)})}$$

$$J(W) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

```
loss = tf.reduce_mean( tf.keras.losses.BinaryCrossentropy(y, predicted) )
```

Understanding how DL works



Adapted from @FrançoisChollet

Training Neural Networks

Next ....