Introduction to Deep Learning and TensorFlow

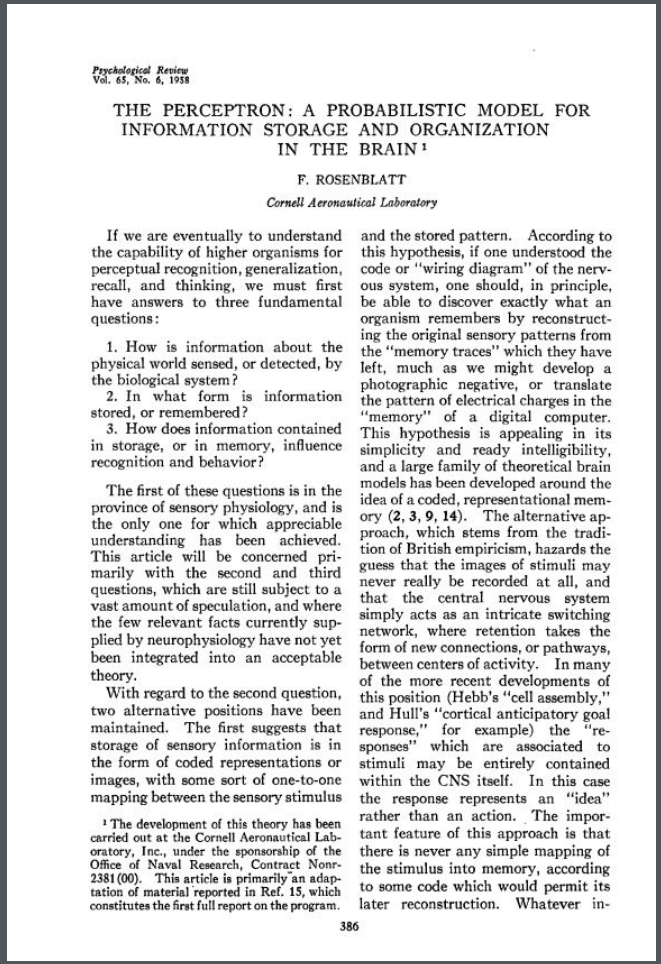
From Perceptron to Training a Neural Network

Ivanovitch Silva
ivanovitch.silva@ufrn.br

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:

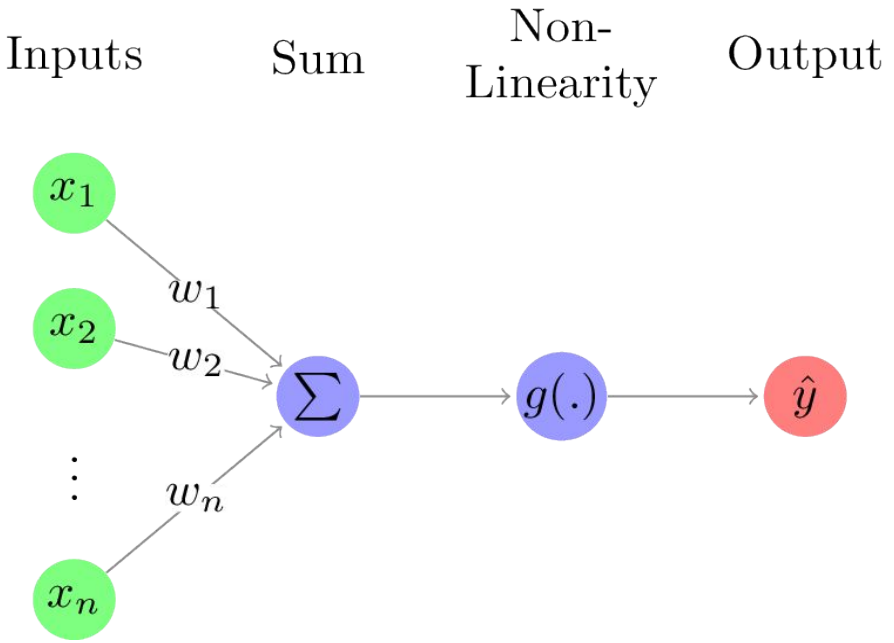
1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

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



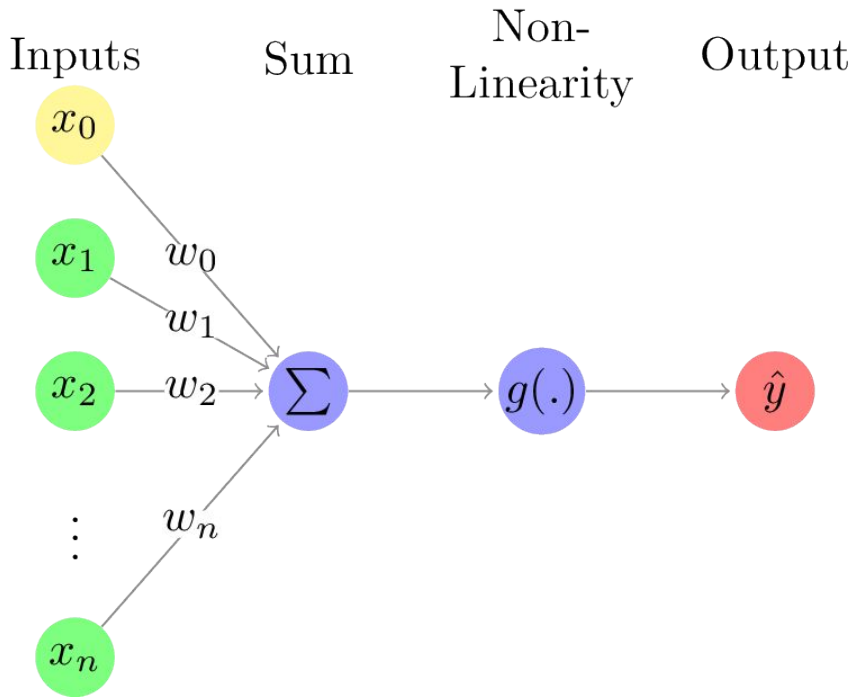
Linear combination of inputs

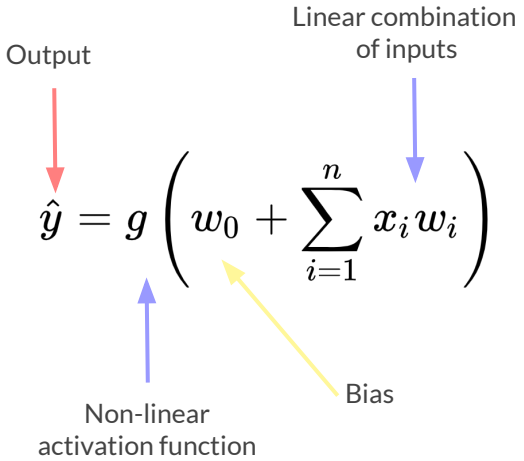
Output

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

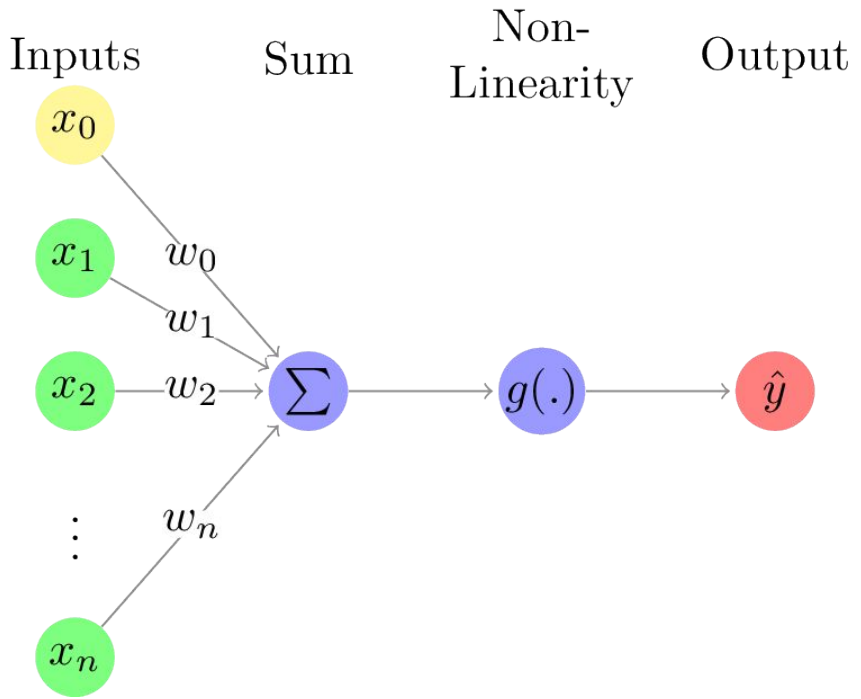
Non-linear activation function

The Perceptron: Forward Propagation





The Perceptron: Forward Propagation

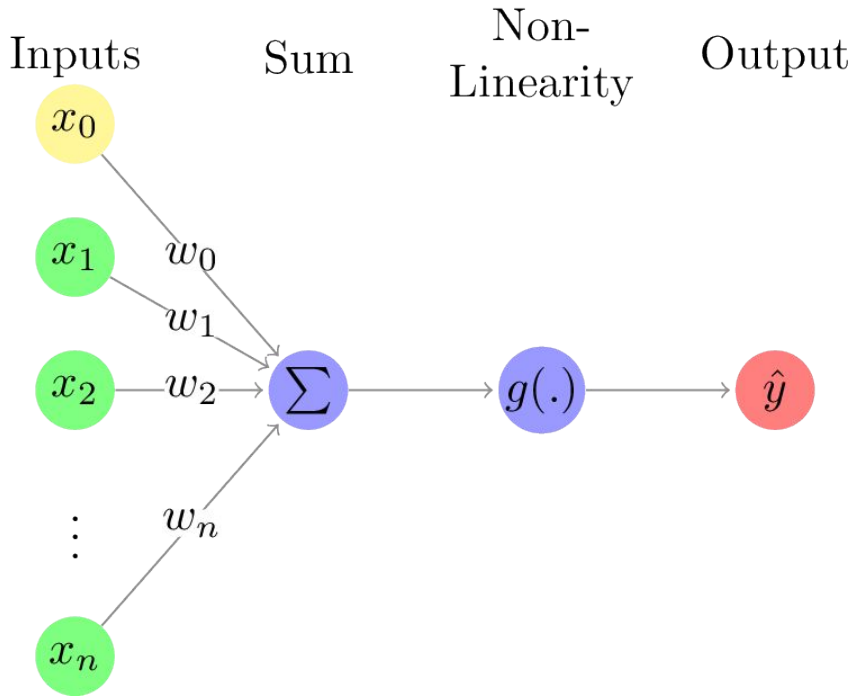


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} x_i w_i\right)$$

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

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

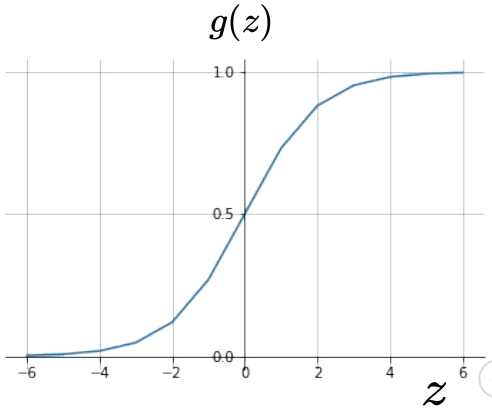


Activation Functions

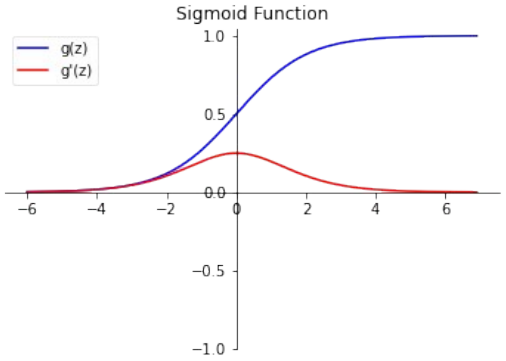
$$\hat{y} = g\left(w_0 + X^T W\right)$$

Example: sigmoid function

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

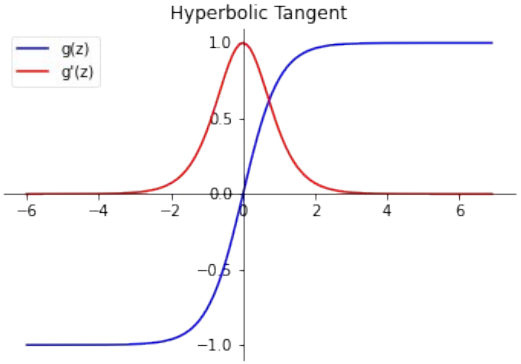


Common Activation Functions



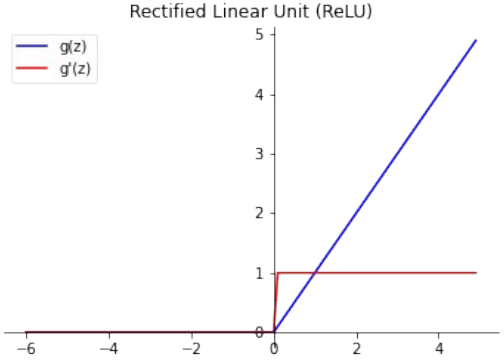
$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$



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

$$g'(z) = 1 - g(z)^2$$



$$g(z) = max(0, z)$$

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

https://arxiv.org/pdf/2005.00817.pdf

A survey on modern trainable activation functions

Andrea Apicella[1], Francesco Donnarumma[2], Francesco Isgrò[1]
and Roberto Prevete[1]
[1]Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Università di Napoli Federico II, Italy
[2]Istituto di Scienze e Tecnologie della Cognizione
CNR, Italy

0817v3 [cs.LG] 26 Jan 2021

Abstract

In neural networks literature, there is a strong interest in identifying and defining activation functions which can improve neural network performance. In recent years there has been a renovated interest of the scientific community in investigating activation functions which can be trained during the learning process, usually referred to as *trainable*, *learnable* or *adaptable* activation functions. They appear to lead to better network performance. Diverse and heterogeneous models of trainable activation function have been proposed in the literature. In this paper, we present a survey of these models. Starting from a discussion on the use of the term "activation function" in literature, we propose a taxonomy of trainable activation functions, highlight common and distinctive proprieties of recent and past models, and discuss main advantages and limitations of this type of approach. We show that many of the proposed approaches are equivalent to adding neuron layers which use fixed (**non-trainable**) activation functions and some simple local rule that constraints the corresponding weight layers.

Keywords— neural networks, machine learning, activation functions, trainable activation functions, learnable activation functions

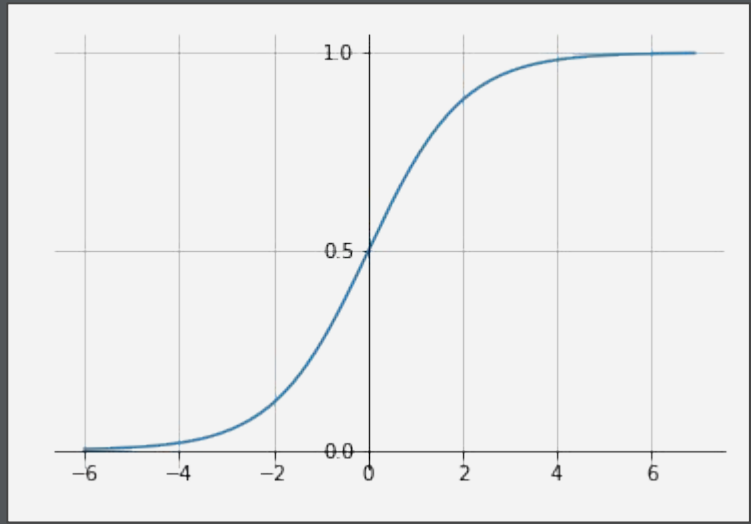
1 Introduction

```
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()
```

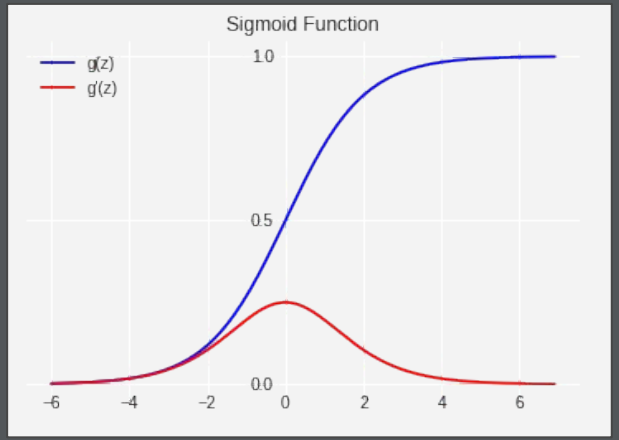
```
>> 0.8807971
```

```
# 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())
```



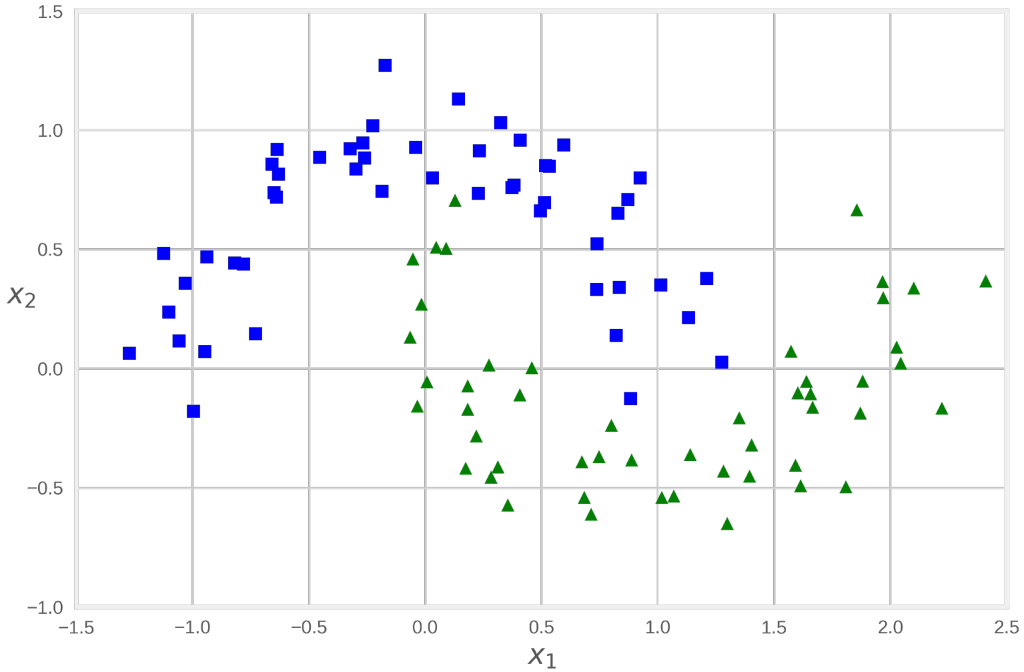
```
# create a range of values
values =  tf.range(-6,7,1,dtype=tf.float32)
# calculates the derivative of sigmoid function
with tf.GradientTape() as tape:
  # Start recording the history of operations applied to 'values'
  tape.watch(values)
  sigmoid_values = tf.math.sigmoid(values)

  # 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

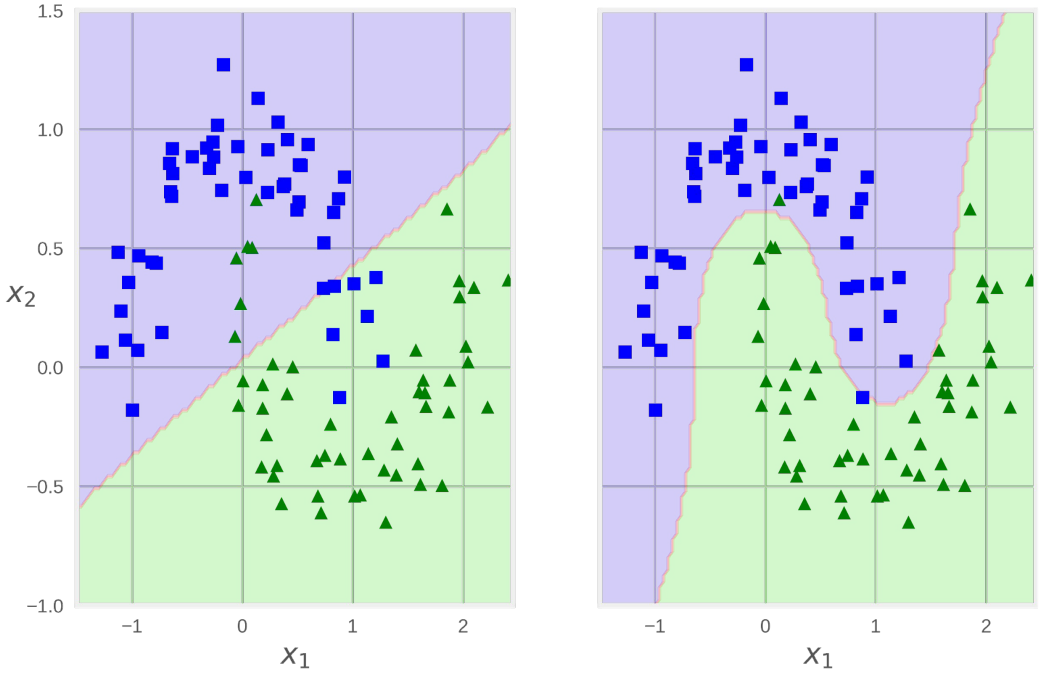


What if we wanted to build a neural network to distinguish blue vs green points?

Importance of Activation Functions

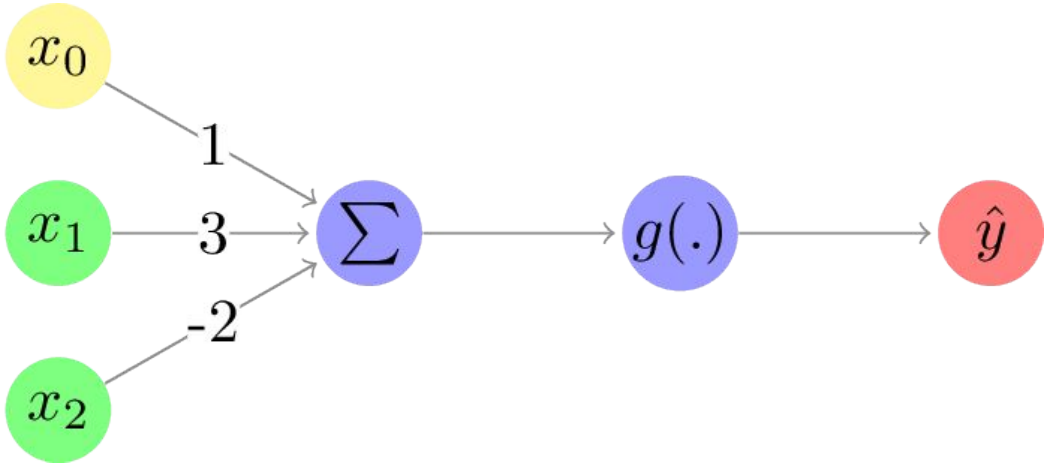
The purpose of activation functions is to introduce **non-linearities** into the network

Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



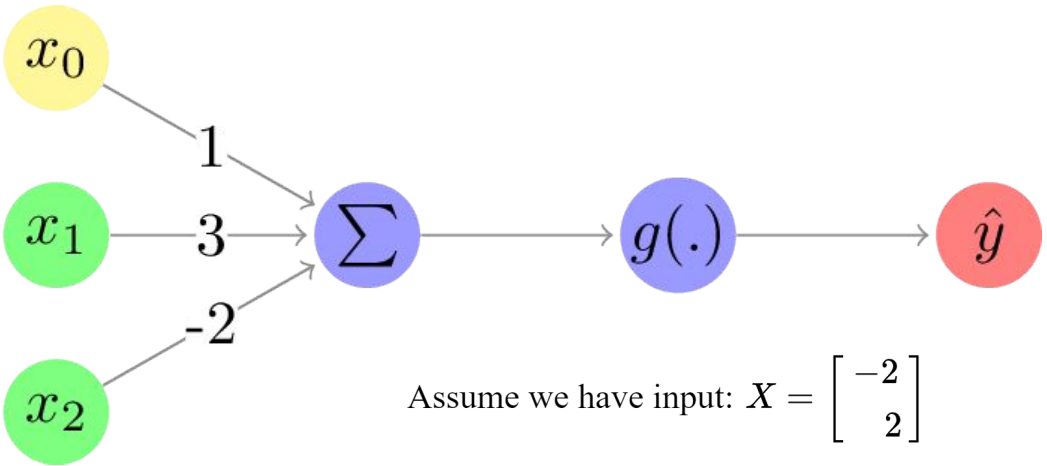
We have $w_0 = 1$ *and* $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\left(w_0 + X^T W\right) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ &= g\underbrace{\left(1 + 3x_1 - 2x_2\right)} \end{aligned}$$

This is just a line in 2D!

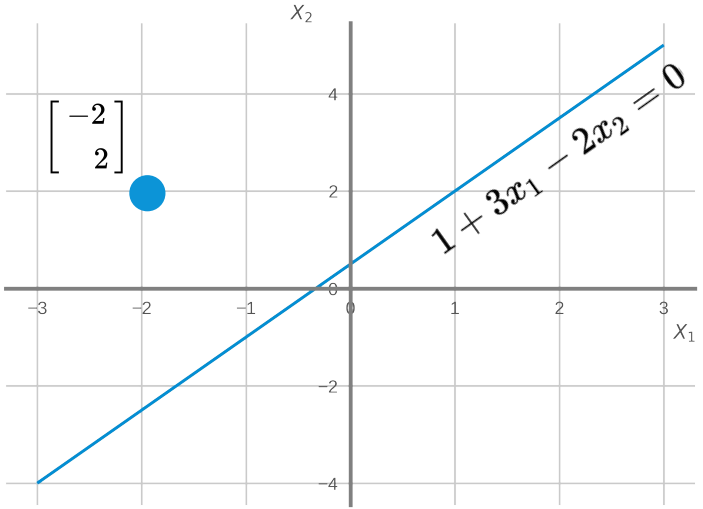
The Perceptron: Example

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



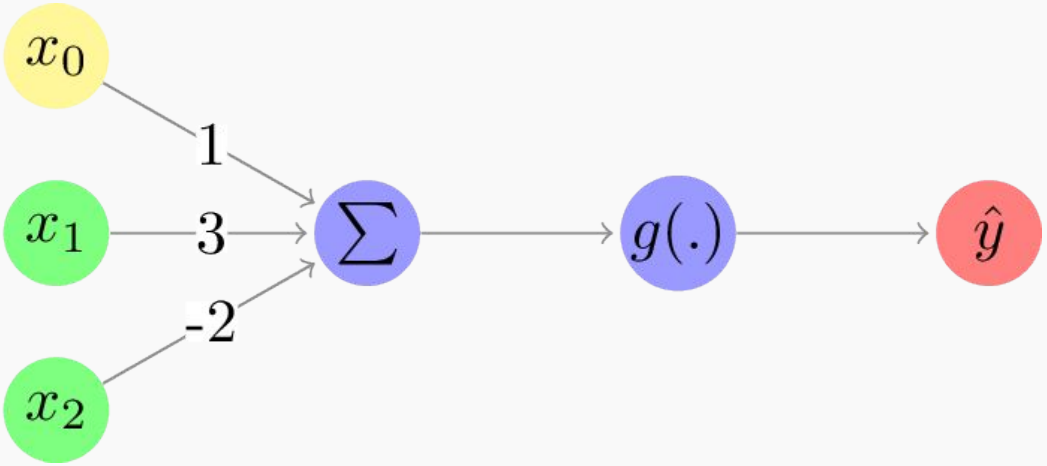
Assume we have input: $X = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$

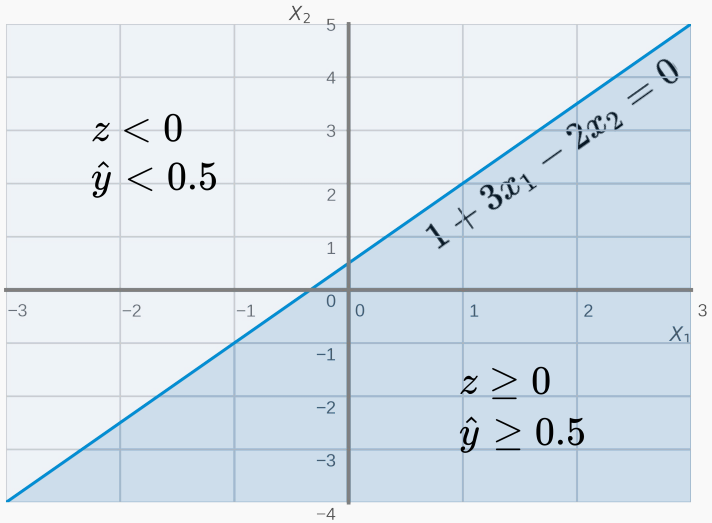
$$\begin{aligned} \hat{y} &= g\left(1 + (3 \times -2) - (2 \times 2)\right) \\ &= g\left(-9\right) \approx 0.00012338161 \end{aligned}$$



The Perceptron: Example

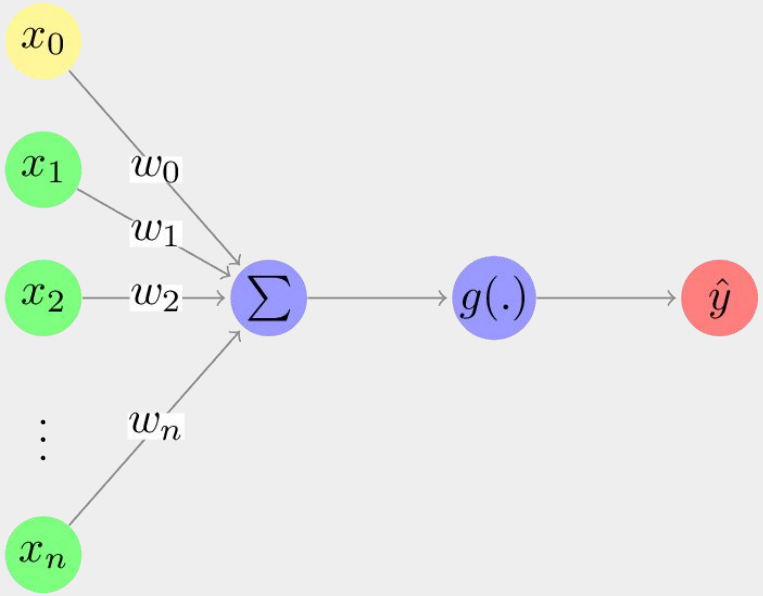
$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$



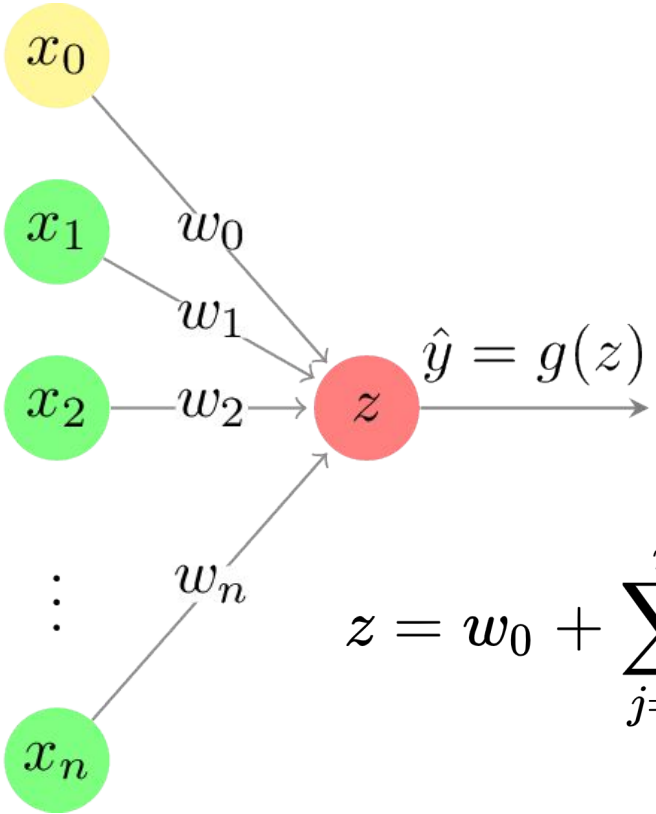


Building Neural Networks

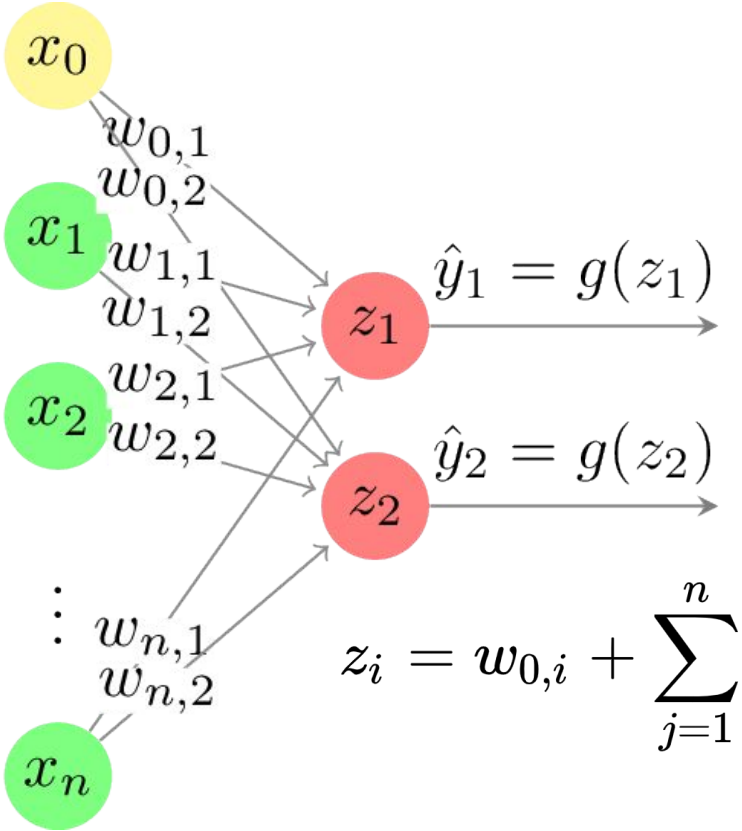
With Perceptrons



The Perceptron: Simplified vs Multi Output



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



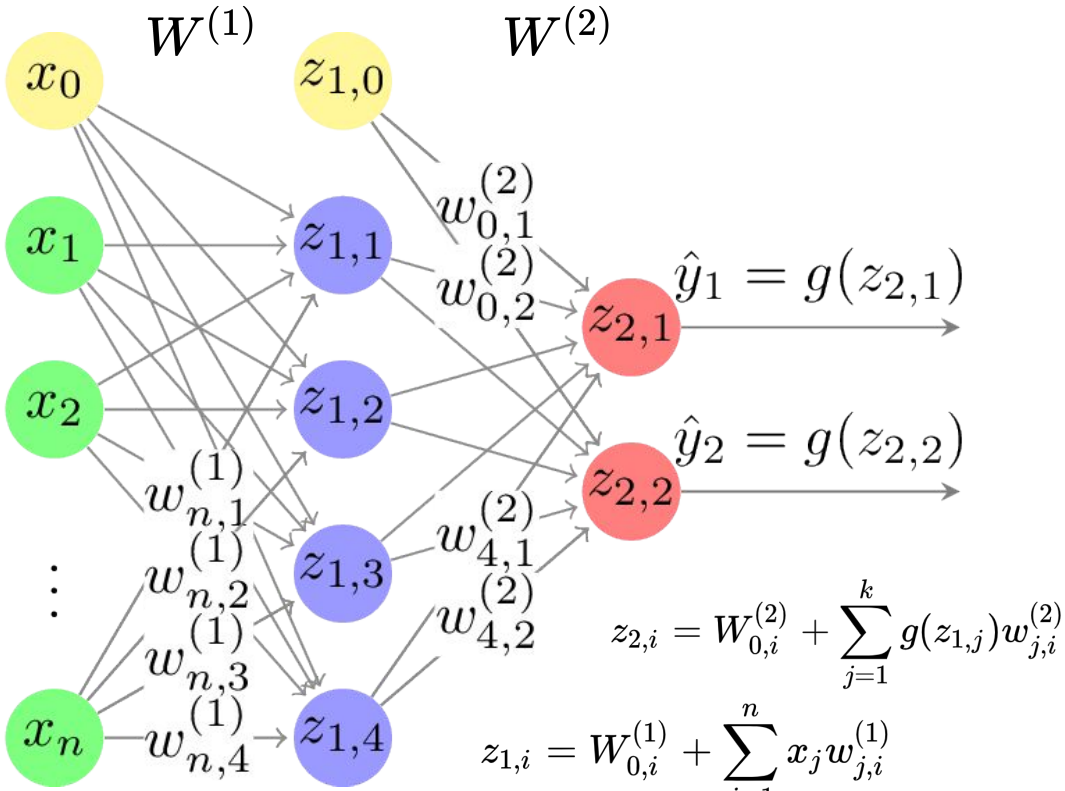
$$z_i = w_{0,i} + \sum_{j=1}^{n} x_j w_{j,i}$$

Dense layer from scratch

```
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
```

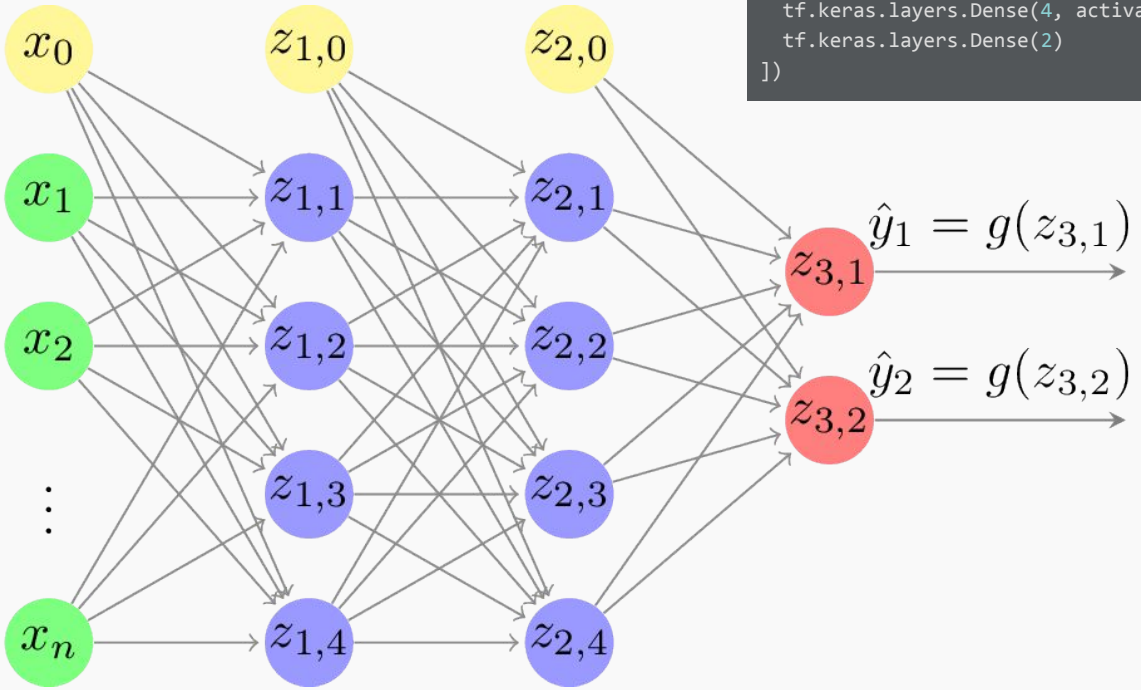
```
layer = tf.keras.layers.Dense(units=2, activation="sigmoid")
```

Single Hidden Layer Neural Network

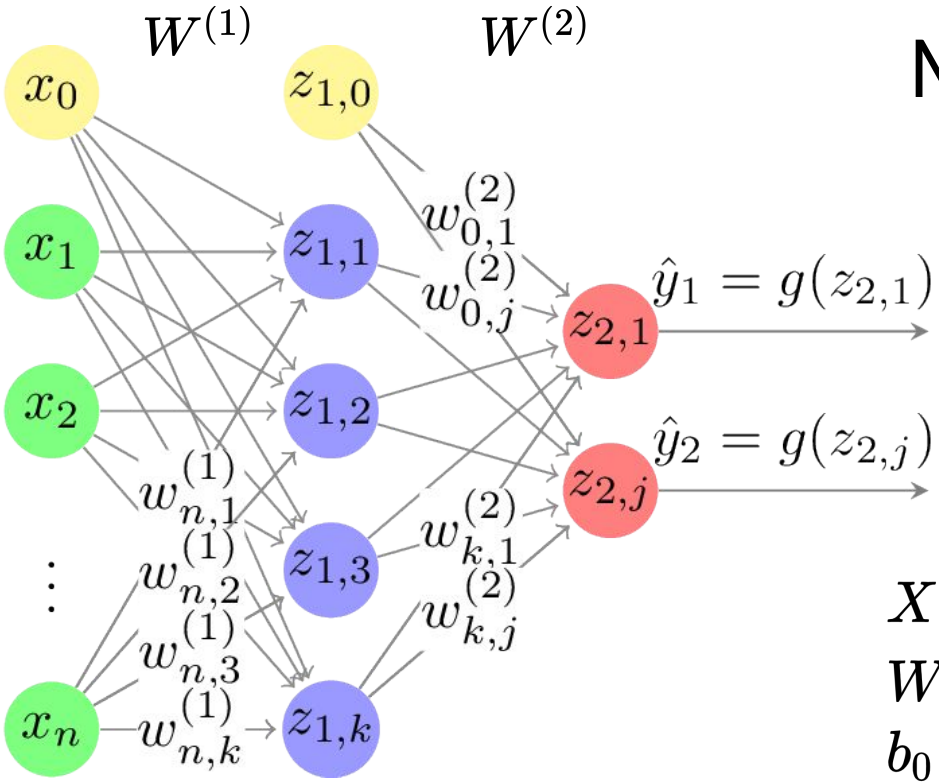


Multi Hidden Layer Neural Network

```
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)
])
```



Matrix Dimension



$z_1 = X \times W^{(1)} + b_0$
$b_0 = x_0$
$a_1 = g(z_1)$

$z_2 = a_1 \times W^{(2)} + b_1$
$b_1 = z_{1,0}$
$a_2 = \hat{y} = g(z_2)$

$X \to 1 \times n$
$W^{(1)} \to n \times k$
$b_0 \to 1 \times k$
$z_1 \to 1 \times k$
$a_1 \to 1 \times k$

$W^{(2)} \to k \times j$
$b_1 \to 1 \times j$
$z_2 \to 1 \times j$
$a_2 \to 1 \times j$

Applying Neural Networks

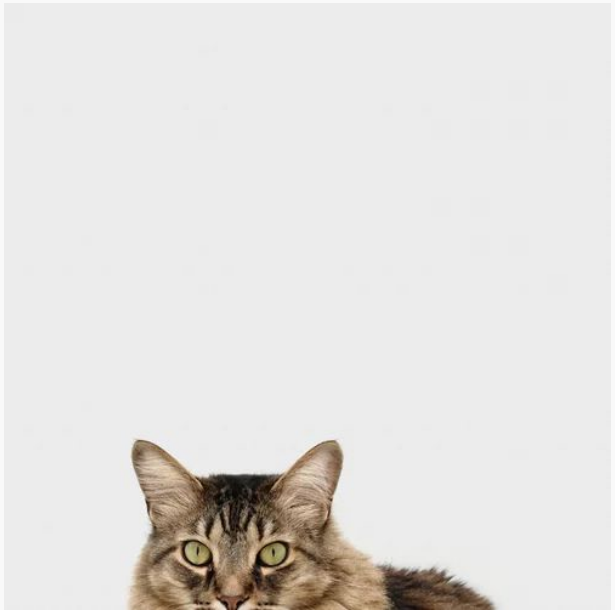
Cat vs Non Cat

Example Problem

Binary Classification

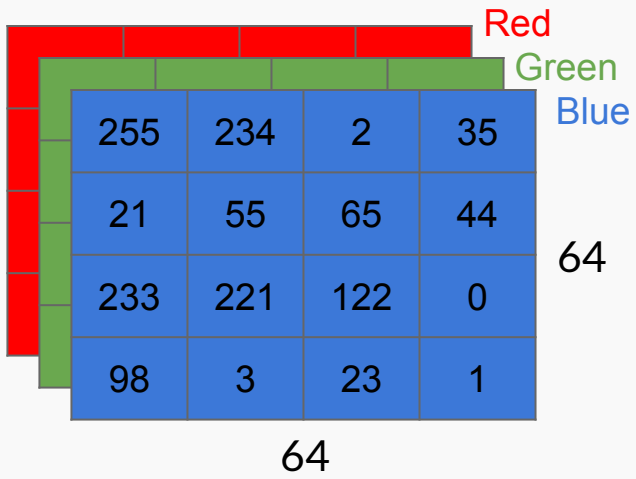
64

64



Number of pixels

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

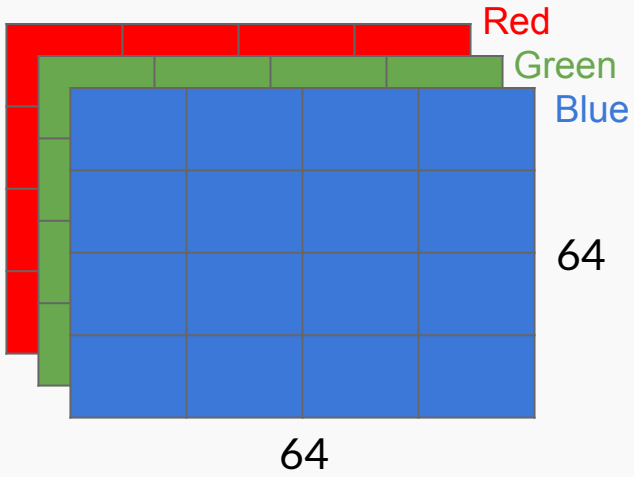


Example Problem

Binary Classification

Number of pixels

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



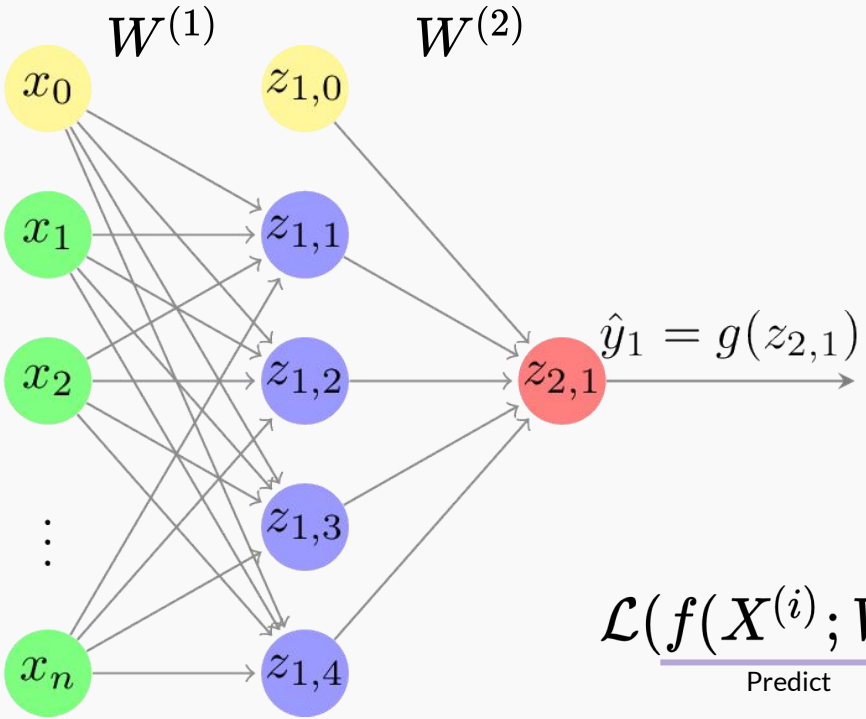
$$y = \begin{matrix} Y^{(1)} \\ Y^{(2)} \\ Y^{(3)} \\ \vdots \\ Y^{(m)} \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

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

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

Quantifying Loss

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

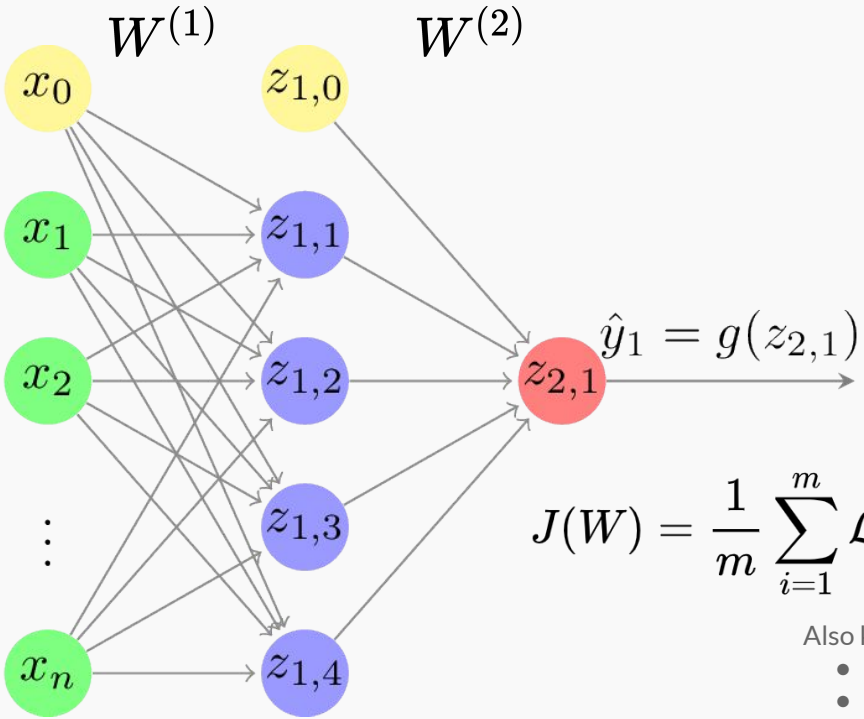


Predicted: 0.6
Actual: 0

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

Empirical Loss

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



$$f(X^{(i)}; W) \quad y^{(i)}$$

$$\begin{bmatrix} 0.6 \\ 0.8 \\ 0.1 \\ \vdots \\ 0.7 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

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

Also known as:

- Objective function
- Cost function; Empirical Risk

Binary Cross Entropy Loss

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

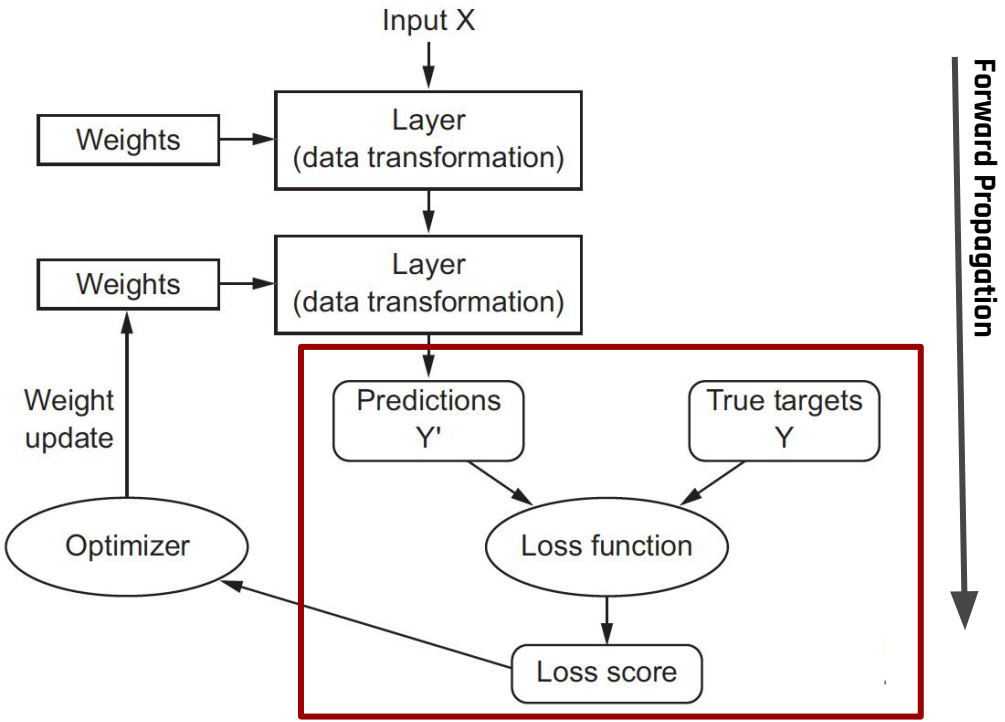
$$\mathcal{L}\left(f(X^{(i)};W), y^{(i)}\right) = \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

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

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

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

Understanding how DL works



Adapted from @FrançoisChollet

Training Neural Networks

Part #01
Loss optimization, Backprop.,
Batch size

Loss Optimization

We want to find the network weights that **achieve the lowest loss**

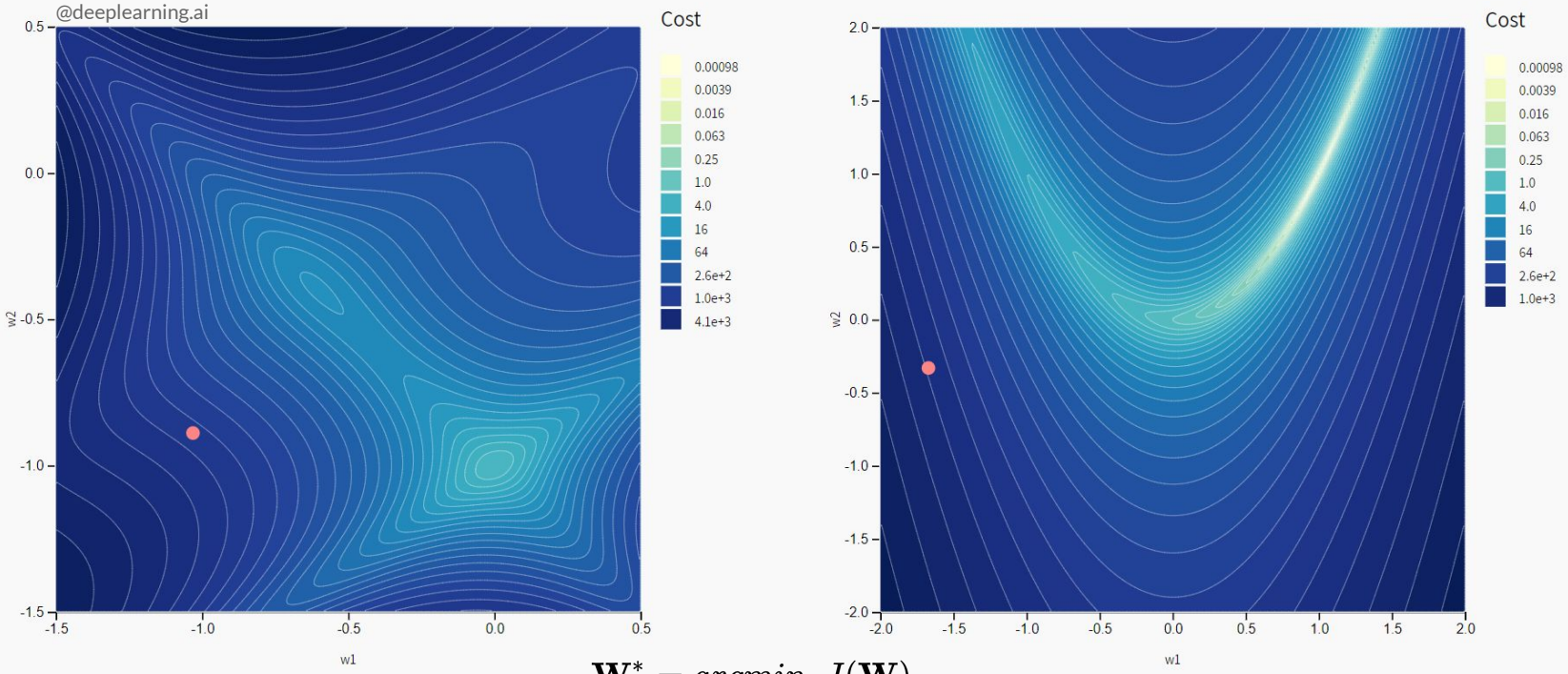
$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(f(X^{(i)}; \mathbf{W}), y^{(i)}\right)$$

$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin}\ J(\mathbf{W})$$

https://www.deeplearning.ai/ai-notes/optimization/

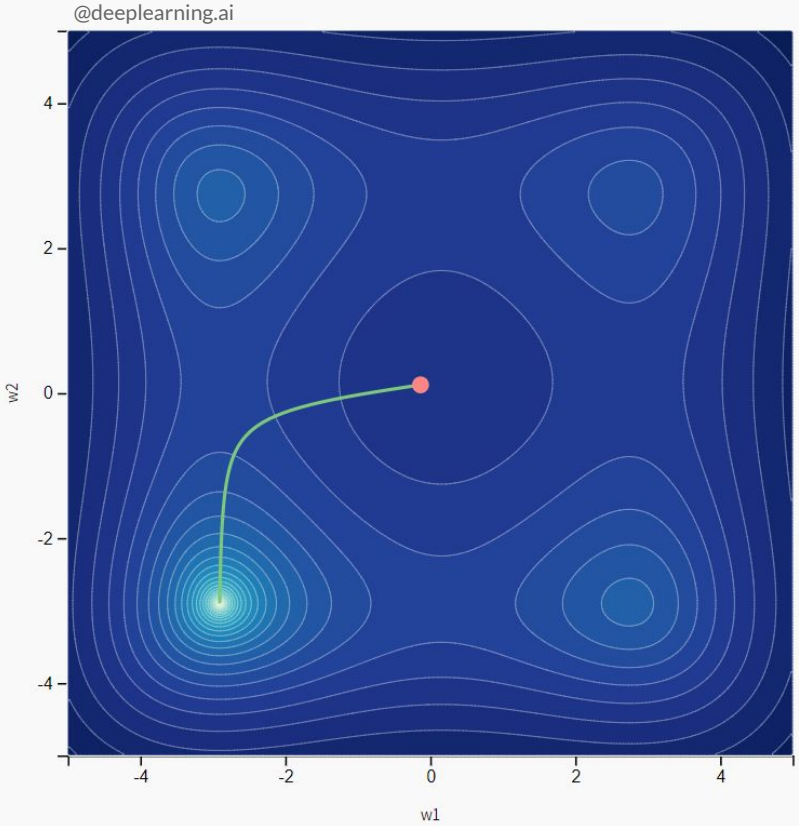
Loss Optimization

Remember: our loss is a function of the network weights!!!



$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin}\ J(\mathbf{W})$$

Loss Optimization - Gradient Descent

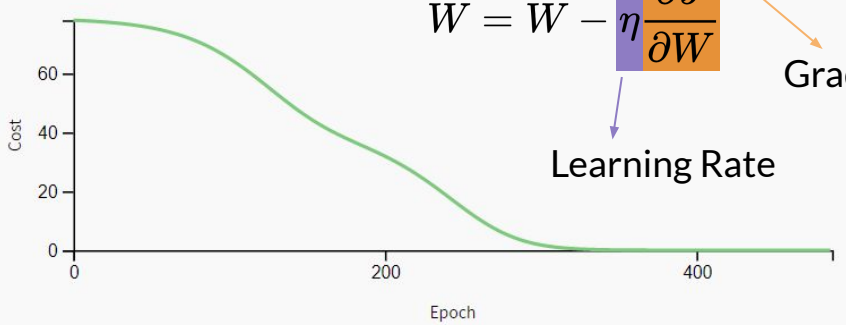


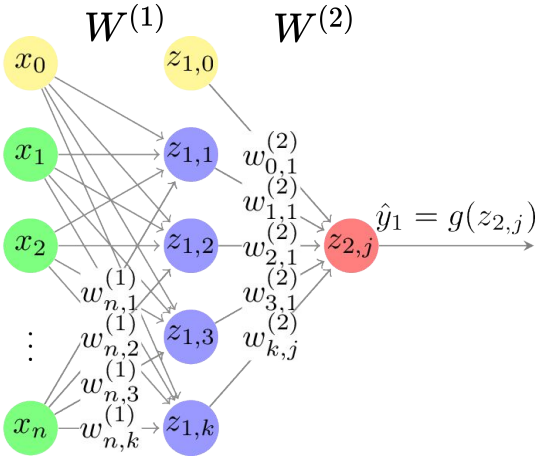
1. Initialize weights randomly
2. Compute gradient
3. Take small step in opposite direction of gradient
4. Repeat until convergence

$$W = W - \eta \frac{\partial J}{\partial W}$$

Gradient

Learning Rate





$$z_1 = X \times W^{(1)} + b_0$$
$$a_1 = g(z_1)$$
$$z_2 = a_1 \times W^{(2)} + b_1$$
$$a_2 = g(z_2) = \hat{y}$$
$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1-y) \log(1-\hat{y})$$

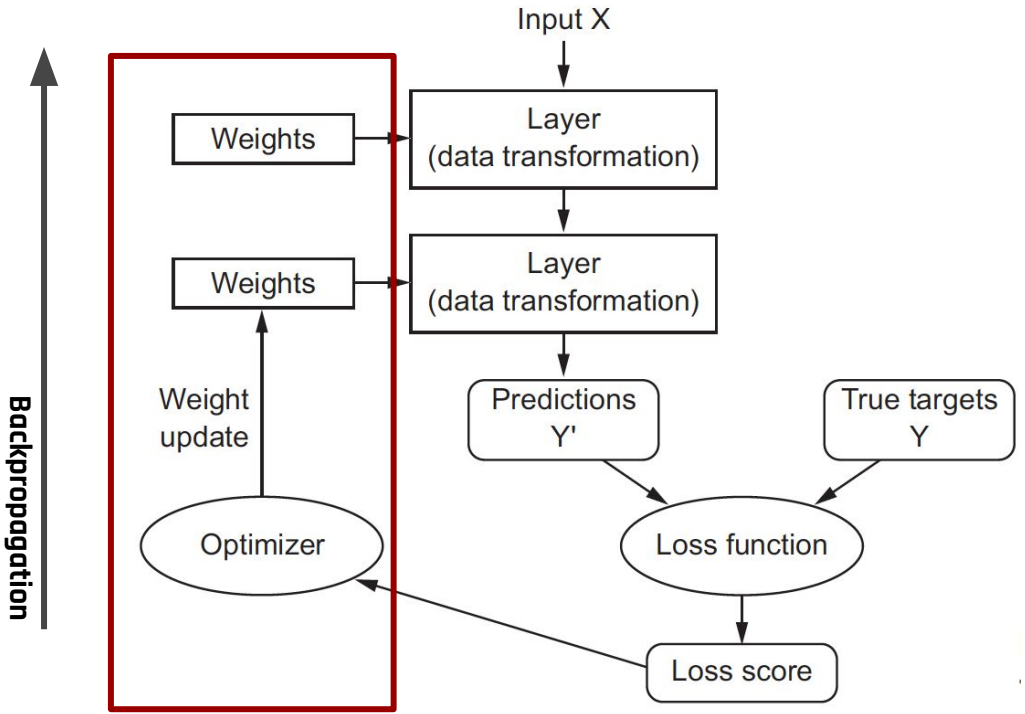
$$X \to (1 \times n)$$
$$W^{(1)} \to (n \times k)$$
$$b_0 \to (1 \times k)$$
$$z_1 \to (1 \times k)$$
$$a_1 \to (1 \times k)$$

$$W^{(2)} \to (k \times j)$$
$$b_1 \to (1 \times j)$$
$$z_2 \to (1 \times j)$$
$$a_2 \to (1 \times j)$$

Backpropagation with Pencil and Paper - See the Video

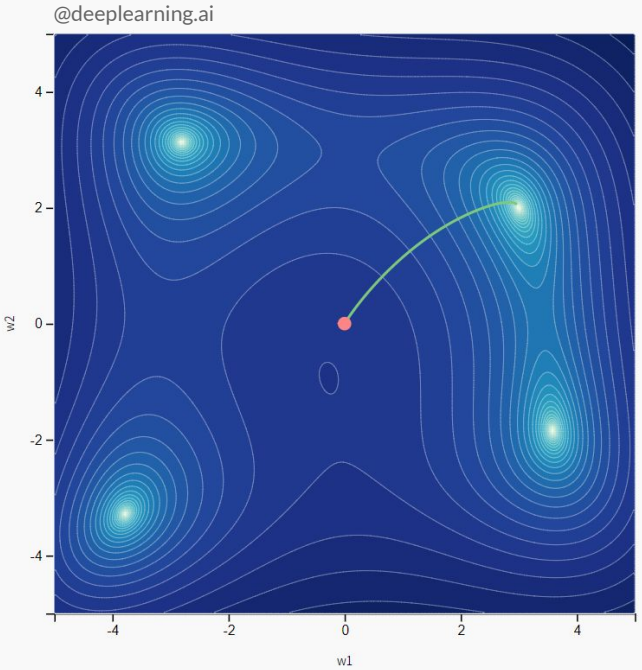
Understanding how DL works

Finding the right values of weights which minimize the error

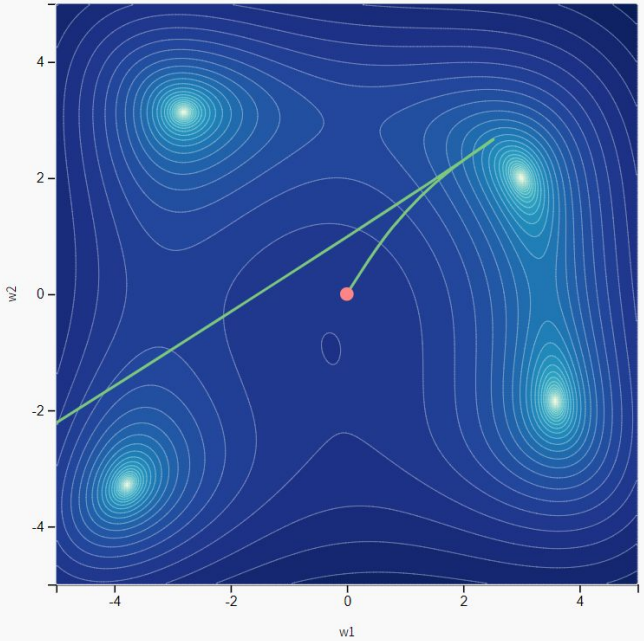


Adapted from @FrançoisChollet

Loss Functions Can Be Difficult to Optimize



Small learning rate (lr=0.001) converges slowly



Large learning rate (lr=0.1) overshoot, become unstable and diverge

How to deal with this?

Idea 1:

Try lots of different learning rates and see what works “just right”

Idea 2:

Do something smarter!!
Design a adaptive learning rate that “adapts” to the landscape

Optimization Algorithms

Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

$$W = W - \eta \boxed{?}$$

$$\beta, \beta_1, \beta_2, learning\_decay$$

TF Implementation

```
tf.keras.optimizers.SGD
```

```
tf.keras.optimizers.Adam
```

```
tf.keras.optimizers.Adadelta
```

```
tf.keras.optimizers.Adagrad
```

```
tf.keras.optimizers.RMSprop
```

Putting it all together

Mini-Batches

$1 \leq b \leq m$

Model

Loss

Evaluation Metrics

Optimizer

```
# Create a source dataset from your training data
dataset = tf.data.Dataset.from_tensor_slices((train_set_x,train_set_y))
dataset = dataset.shuffle(buffer_size=64).batch(32)

# Instantiate a simple classification model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(8, activation=tf.nn.relu, dtype='float64'),
  tf.keras.layers.Dense(8, activation=tf.nn.relu, dtype='float64'),
  tf.keras.layers.Dense(1, activation=tf.nn.sigmoid, dtype='float64')
])

# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.BinaryCrossentropy()

# Instantiate an accuracy metric.
accuracy = tf.keras.metrics.BinaryAccuracy()

# Instantiate an optimizer.
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
```

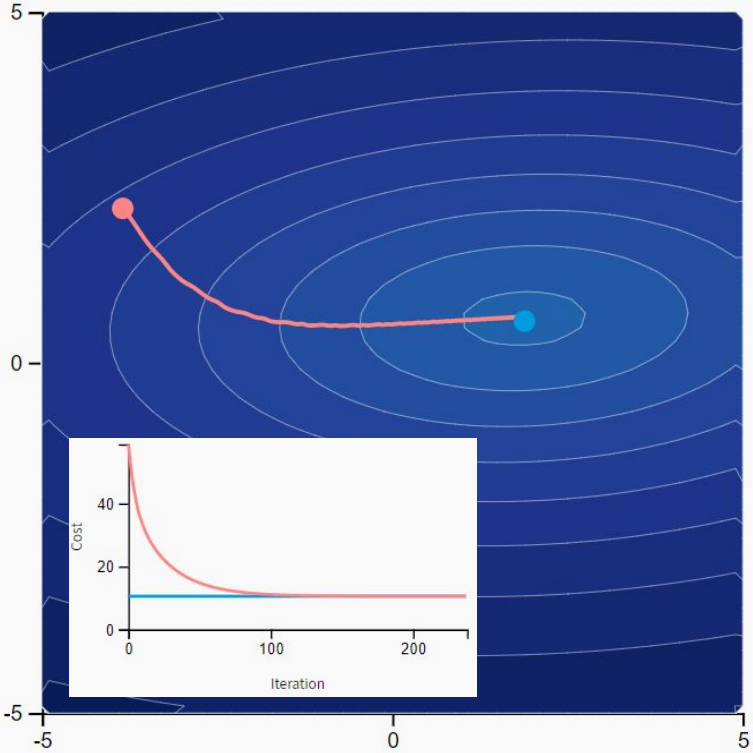
Putting it all together

```
for i in range(500):
    # Iterate over the batches of the dataset.
    for step, (x, y) in enumerate(dataset):
      # Open a GradientTape.
      with tf.GradientTape() as tape:

        # Forward pass.
        logits = model(x)

        # Loss value for this batch.
        loss_value = loss(y, logits)

      # Get gradients of loss wrt the weights.
      gradients = tape.gradient(loss_value, model.trainable_weights)

      # Update the weights of our linear layer.
      optimizer.apply_gradients(zip(gradients, model.trainable_weights))

      # Update the running accuracy.
      accuracy.update_state(y, logits)
```

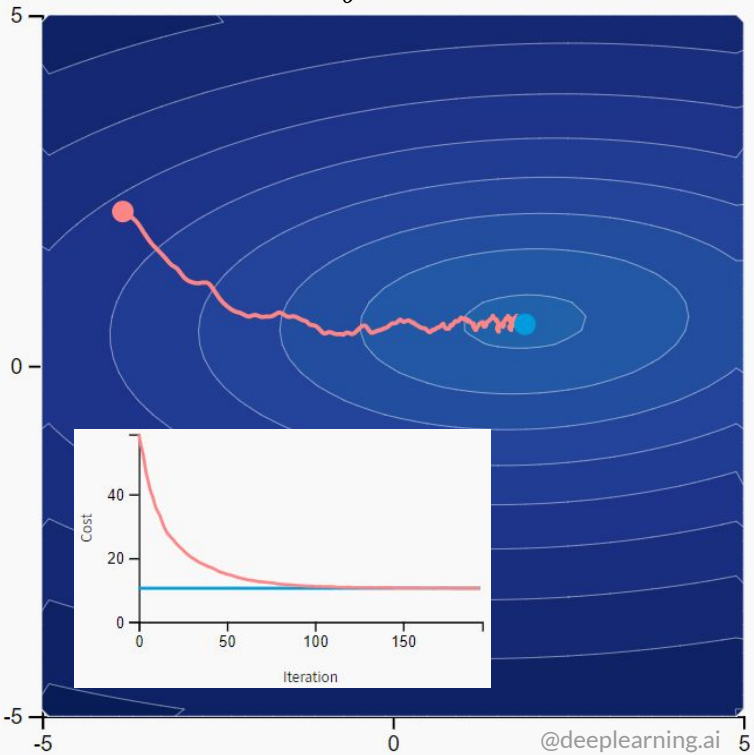
$$W = W - \eta \frac{\partial J}{\partial W}$$

Mini-Batches Challenges

$m_b = m$



$m_b = 24$



Training Neural Networks

Part #02
Exponentially weighted average,
Adaptive Learning Rate + Opt.

TRAIN A DEEP NEURAL NETWORK

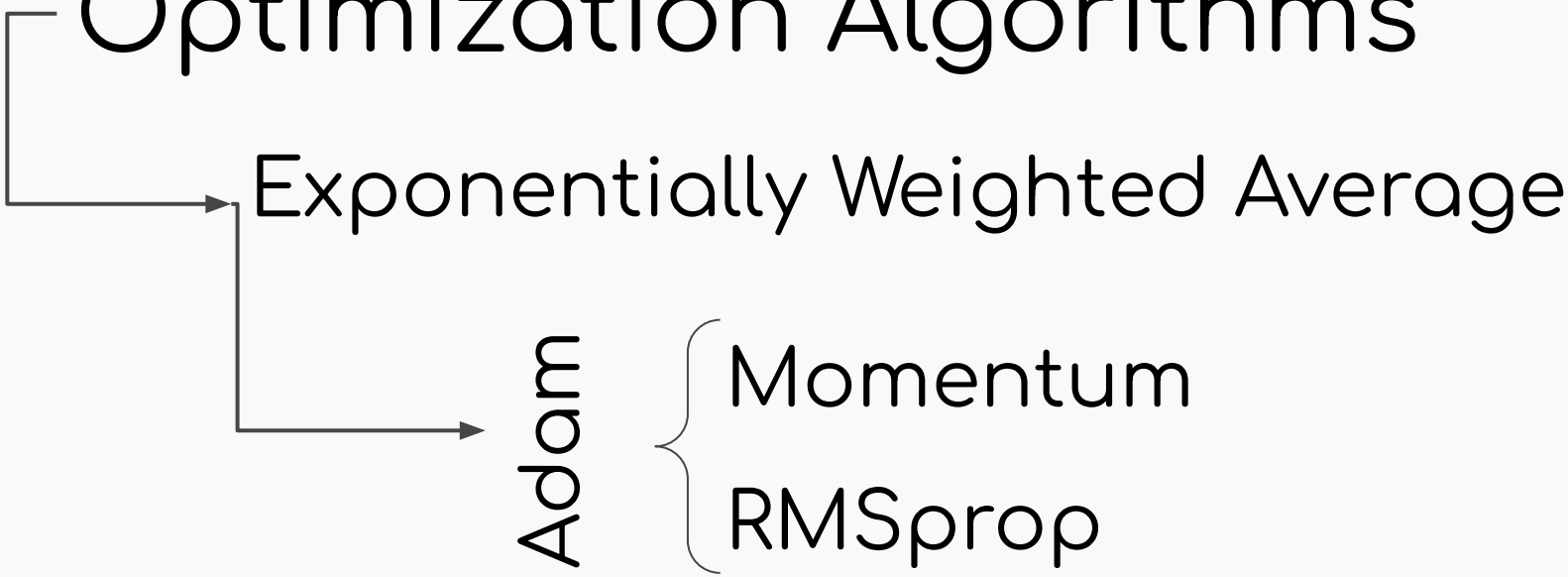
=

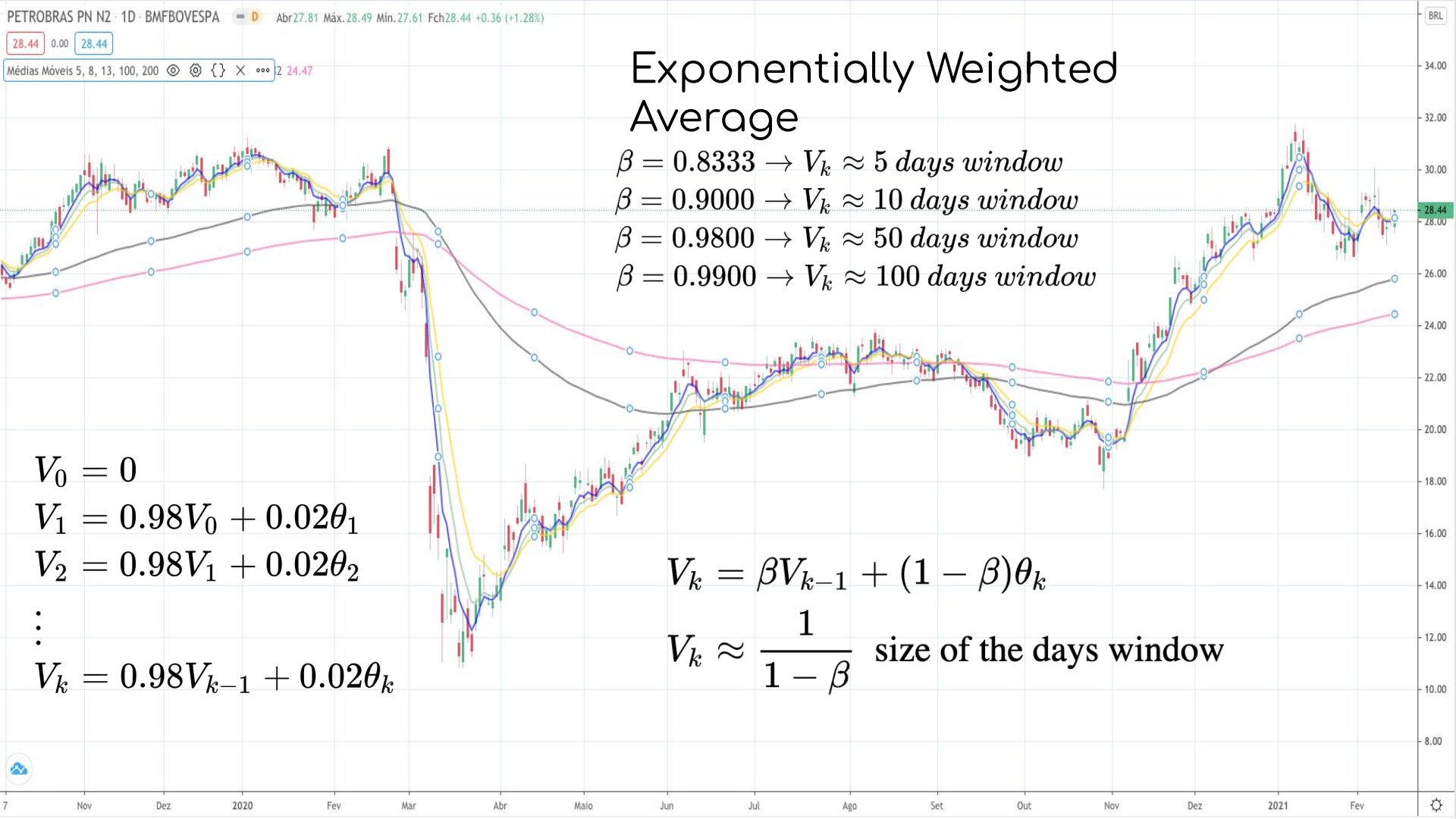
MINI-BATCH

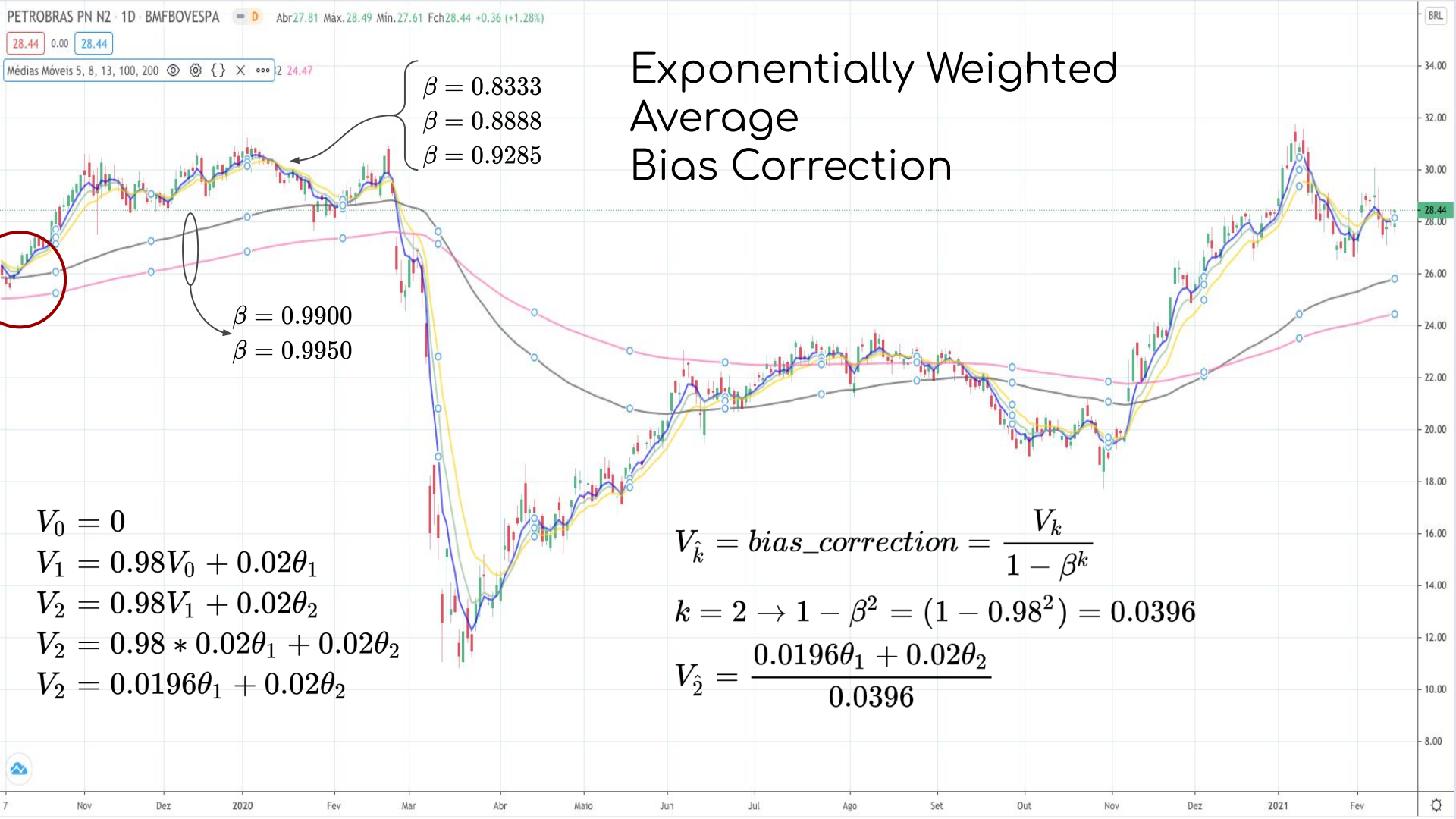
+

OPTIMIZATION ALGORITHMS

Optimization Algorithms




PETROBRAS PN N2 · 1D · BMFBOVESPA
Médias Móveis 5, 8, 13, 100, 200
Exponentially Weighted Average
$\beta = 0.8333 \rightarrow V_k \approx 5\ days\ window$
$\beta = 0.9000 \rightarrow V_k \approx 10\ days\ window$
$\beta = 0.9800 \rightarrow V_k \approx 50\ days\ window$
$\beta = 0.9900 \rightarrow V_k \approx 100\ days\ window$
$V_0 = 0$
$V_1 = 0.98V_0 + 0.02\theta_1$
$V_2 = 0.98V_1 + 0.02\theta_2$
$\vdots$
$V_k = 0.98V_{k-1} + 0.02\theta_k$
$V_k = \beta V_{k-1} + (1-\beta)\theta_k$
$V_k \approx \frac{1}{1-\beta}$ size of the days window



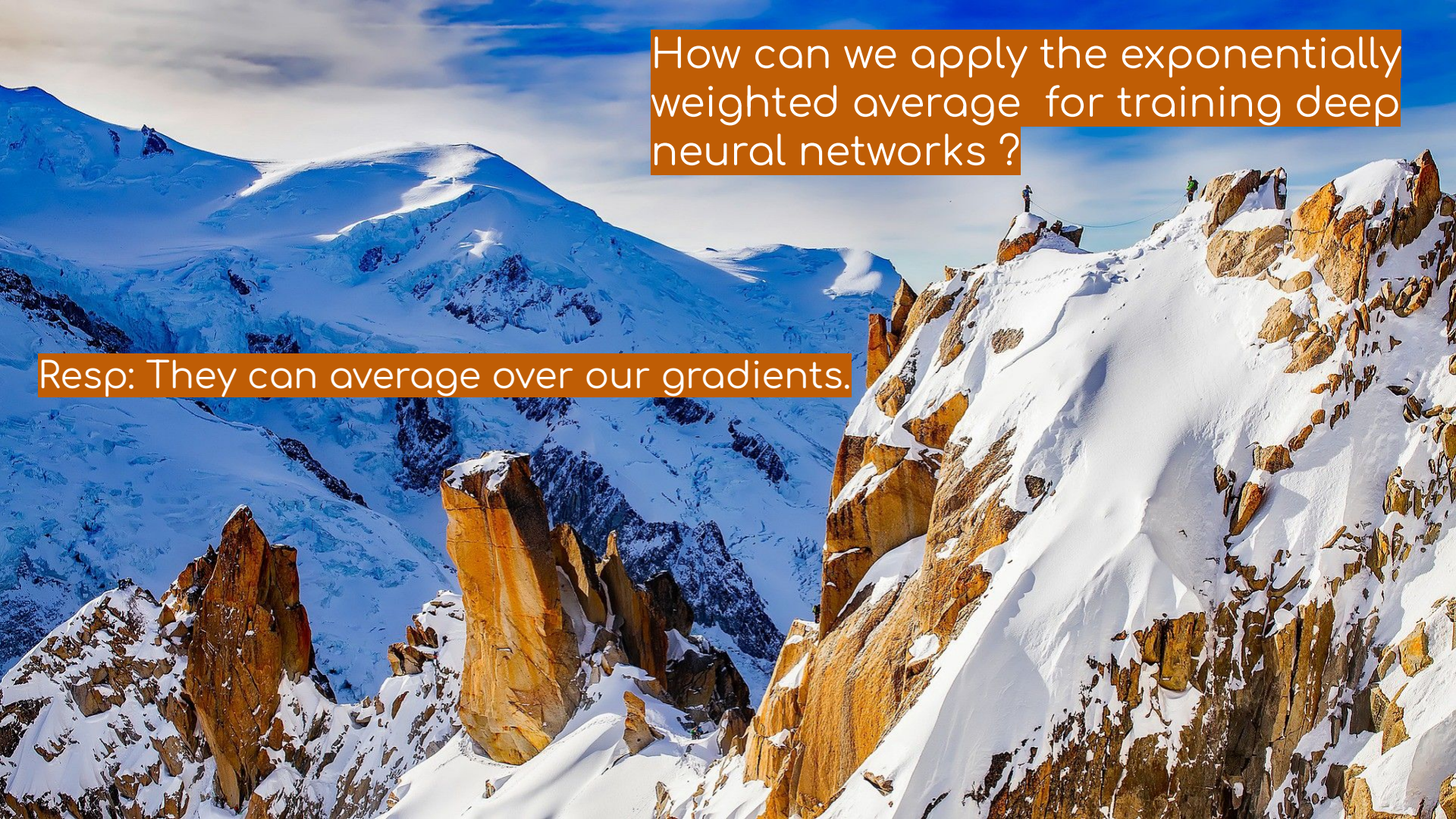
Exponentially Weighted Average Bias Correction

$$V_0 = 0$$
$$V_1 = 0.98V_0 + 0.02\theta_1$$
$$V_2 = 0.98V_1 + 0.02\theta_2$$
$$V_2 = 0.98 * 0.02\theta_1 + 0.02\theta_2$$
$$V_2 = 0.0196\theta_1 + 0.02\theta_2$$

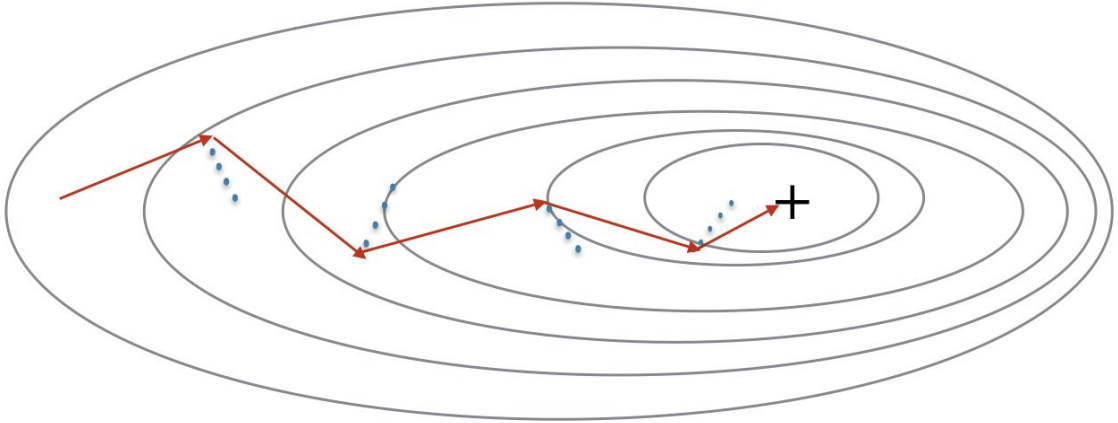
$$V_{\hat{k}} = bias\_correction = \frac{V_k}{1-\beta^k}$$

$$k = 2 \rightarrow 1 - \beta^2 = (1 - 0.98^2) = 0.0396$$

$$V_{\hat{2}} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$


How can we apply the exponentially weighted average for training deep neural networks ?
Resp: They can average over our gradients.

Gradient Descent with Momentum



- Momentum takes into account the past gradients to smooth out the update.
- Formally, this will be the exponentially weighted average of the gradient on previous steps.

Gradient Descent with Momentum

$$
\begin{aligned}
&On\ iteration\ t \\
&\quad Compute\ \ dW, db\ on\ the\ current\ mini\_batch \\
&\quad v_{dw} = \beta v_{dw} + (1-\beta) dW \\
&\quad v_{db} = \beta v_{db} + (1-\beta) db \\
&\quad W = W - \eta v_{dw} \\
&\quad b = b - \eta v_{db} \\
&Hyperparameters : \eta, \beta \qquad \beta = 0.9
\end{aligned}
$$

Gradient Descent with RMSprop

$$On\ iteration\ t$$

$$Compute\ \ dW, db\ on\ the\ current\ mini\_batch$$

$$s_{dw} = \beta s_{dw} + (1-\beta) dW^2$$

$$s_{db} = \beta s_{db} + (1-\beta) db^2$$

$$W = W - \eta \frac{dW}{\sqrt{s_{dw}} + \epsilon}$$

$$b = b - \eta \frac{db}{\sqrt{s_{db}} + \epsilon}$$

$$Hyperparameters : \eta, \beta \qquad \beta = 0.9, \epsilon = 1e^{-8}$$

Gradient Descent with Adam

On iteration t

Compute dW, db *on the current mini_batch*

$$v_{dw} = \beta_1 v_{dw} + (1-\beta_1)dW \qquad v_{db} = \beta_1 v_{db} + (1-\beta_1)db$$

$$s_{dw} = \beta_2 s_{dw} + (1-\beta_2)dW^2 \qquad s_{db} = \beta_2 s_{db} + (1-\beta_2)db^2$$

$$v_{dw}^{correct} = v_{dw}/(1-\beta_1^t) \qquad v_{db}^{correct} = v_{db}/(1-\beta_1^t)$$

$$s_{dw}^{correct} = s_{dw}/(1-\beta_2^t) \qquad s_{db}^{correct} = s_{db}/(1-\beta_2^t)$$

$$W = W - \eta \frac{v_{dw}^{correct}}{\sqrt{s_{dw}^{correct}} + \epsilon} \qquad b = b - \eta \frac{v_{db}^{correct}}{\sqrt{s_{db}^{correct}} + \epsilon}$$

$$Hyperparameters: \eta, \beta_1, \beta_2 \qquad \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e^{-8}$$

Learning Rate Decay

Idea: reduce η for each mini-bach t in order to smooth the gradient descent.

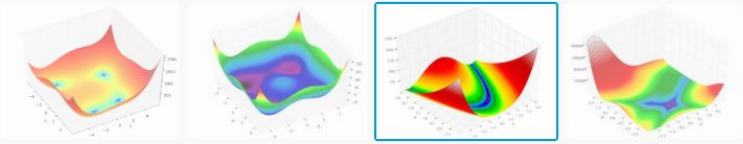
$$\eta = \frac{1}{1 + learning\_decay \times epoch\_num} \times \eta_0$$

$$\eta = 0.95 e^{epoch\_num} \eta_0$$

$$\eta = \frac{k}{\sqrt{epoch\_num}} \times \eta_0$$

1. Choose a cost landscape

Select an artificial landscape $\mathcal{J}(w_1, w_2)$.



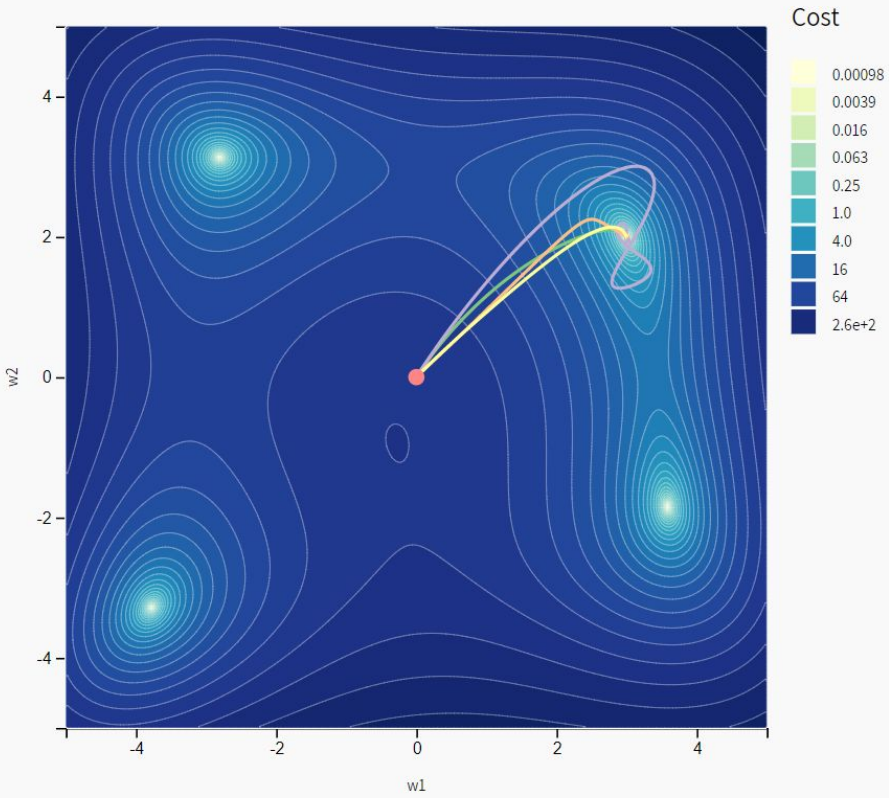
2. Choose initial parameters

On the cost landscape graph, drag the red dot to choose initial parameter values and thus the initial value of the cost.

3. Choose an optimizer

Select the optimizer(s) and hyperparameters.

Optimizer	Learning Rate	Learning Rate Decay
☑ Gradient Descent	0,001	0
☑ Momentum	0,001	0
☑ RMSprop	0,001	0
☑ Adam	0,001	0



https://www.deeplearning.ai/ai-notes/optimization/

Introduction to Deep Learning and TensorFlow

From Perceptron to Training a Neural Network

Next ....