



Introduction to Deep Learning and TensorFlow

From Perceptron to Training a Neural
Network

01

The Perceptron

The structural building block of
deep learning

02

Building Neural Networks

Single and multi output
perceptron, single and multi
hidden layers

03

Applying Neural Networks

Cat vs Non Cat

04

Training Neural Networks #01

Loss optimization, gradient
descent, mini-batches

05

Training Neural Networks #02

Optimization algorithms

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 BRAIN¹

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:

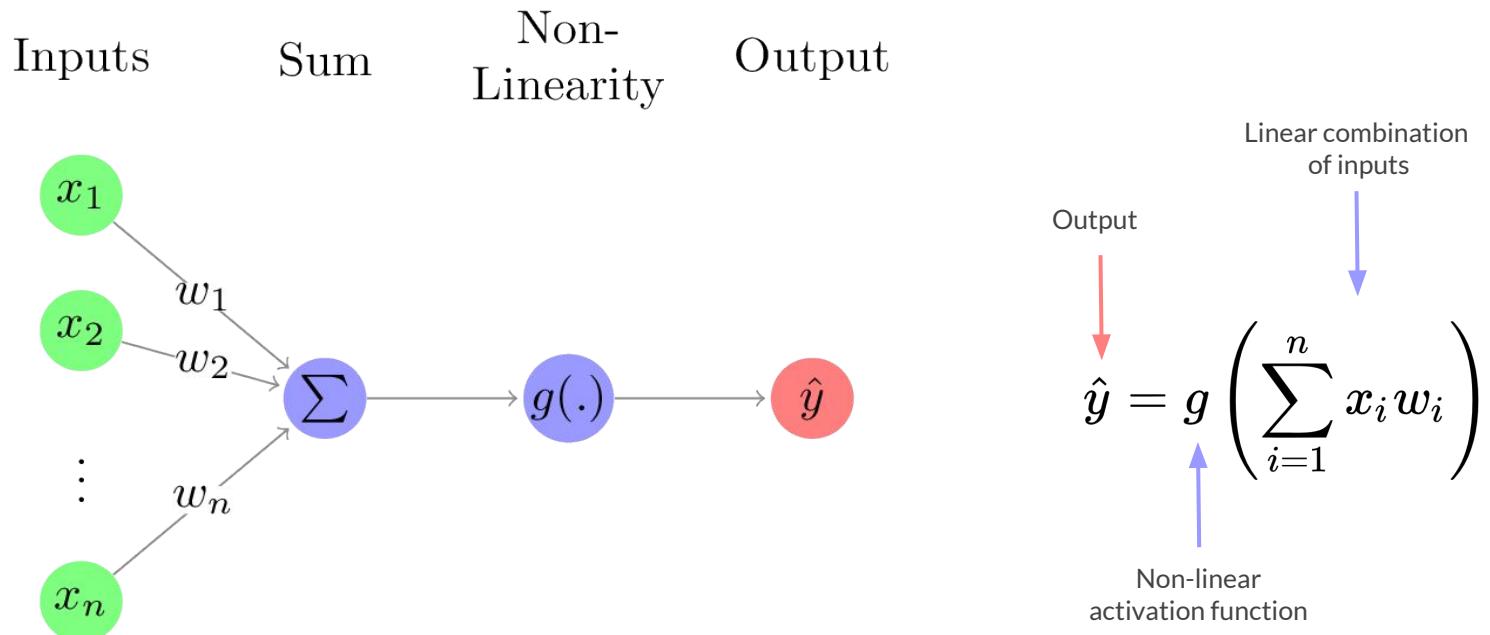
1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

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.

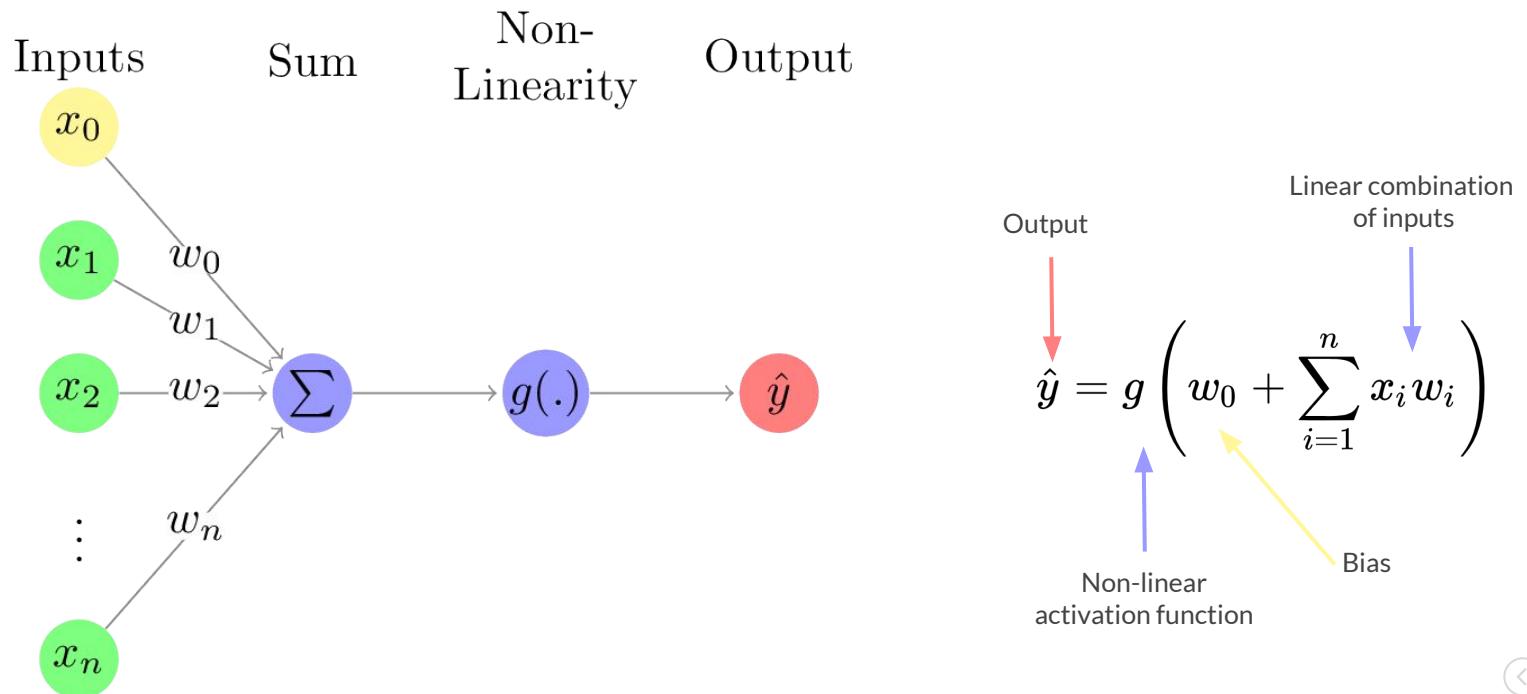
With regard to the second question, two alternative positions have been maintained. The first suggests that storage of sensory information is in the form of coded representations or images, with some sort of one-to-one mapping between the sensory stimulus

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.

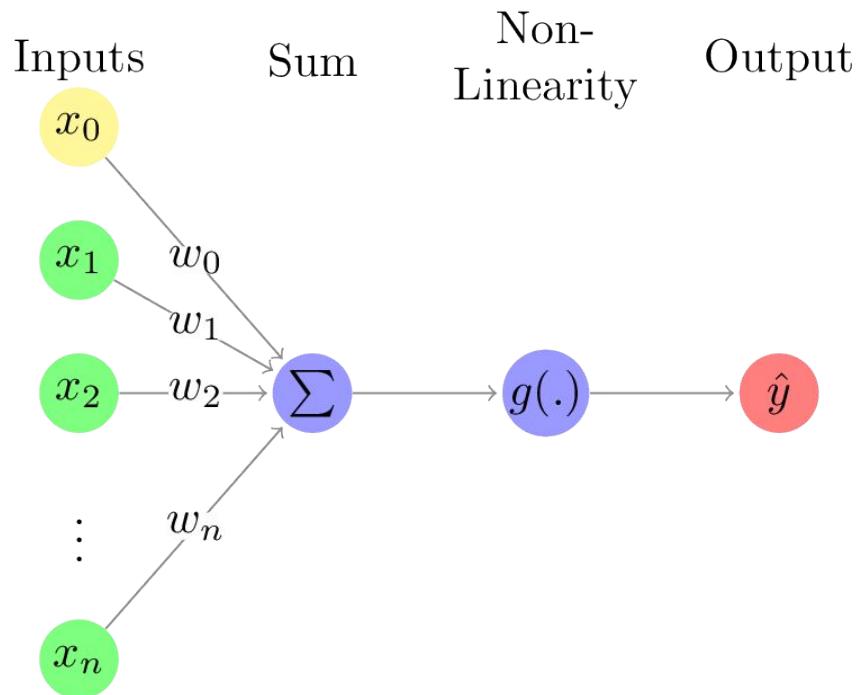
The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



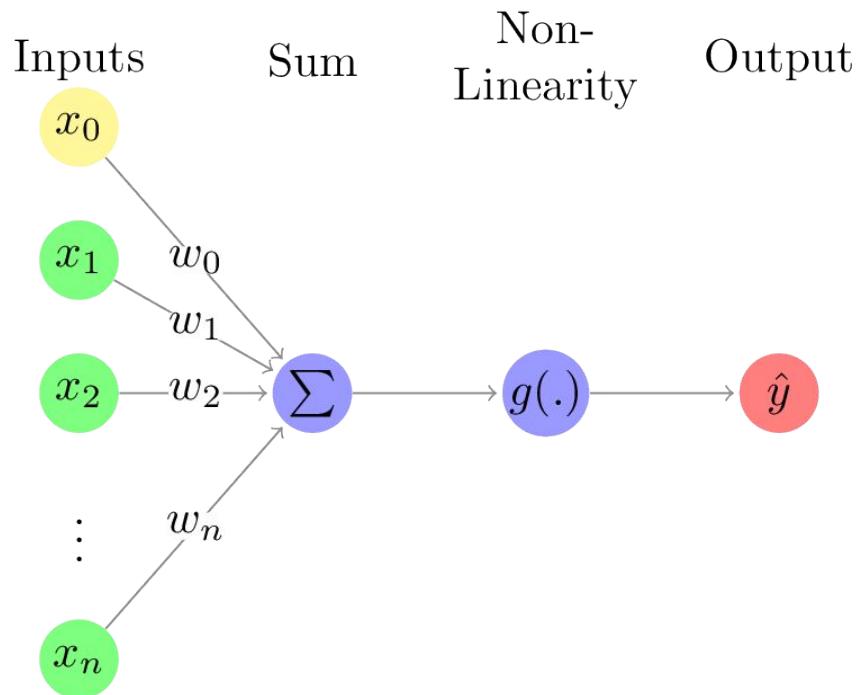
$$\hat{y} = g \left(w_0 + \sum_{i=1}^n x_i w_i \right)$$

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

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

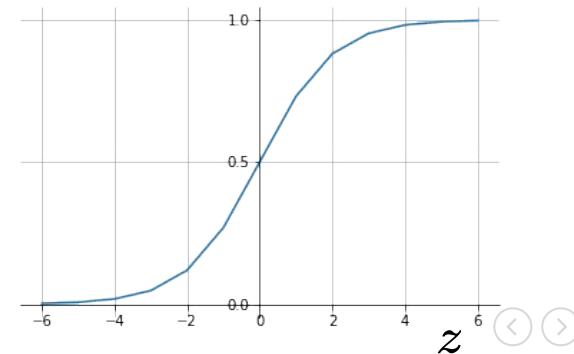


$$\hat{y} = g(w_0 + X^T W)$$

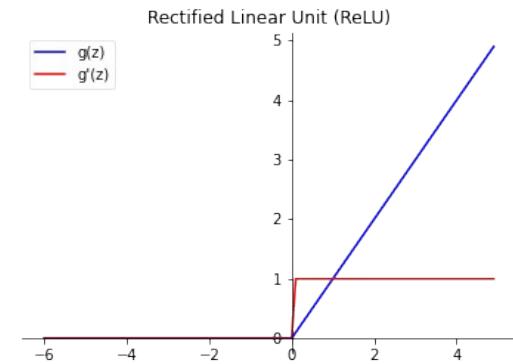
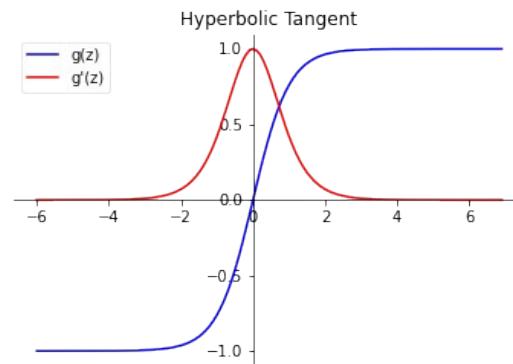
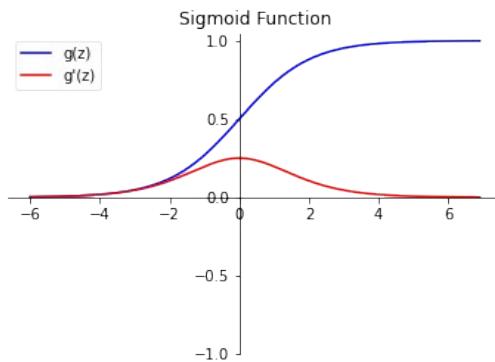
Example: sigmoid function

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

$$g(z)$$



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 Apicella¹, Francesco Donnarumma², Francesco Isgrò¹
and Roberto Prevete¹

¹Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Università di Napoli Federico II, Italy

²Istituto di Scienze e Tecnologie della Cognizione
CNR, Italy

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

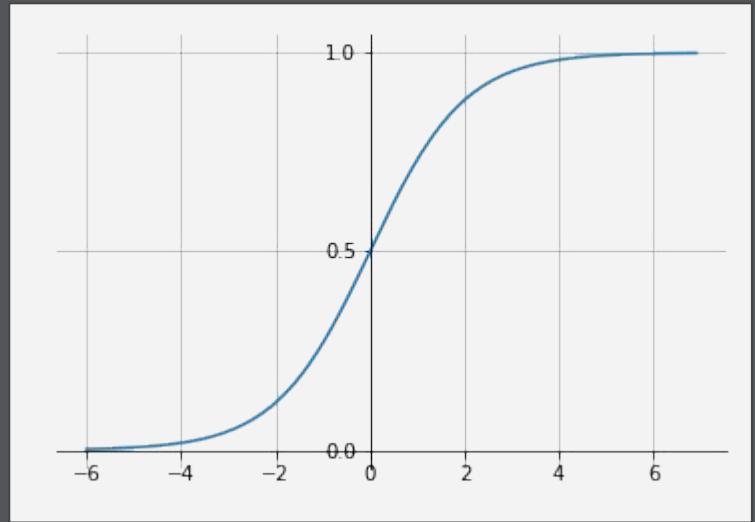


```
import tensorflow as tf  
import numpy as np  
import matplotlib.pyplot as plt
```

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

```
>> 0.8807971
```

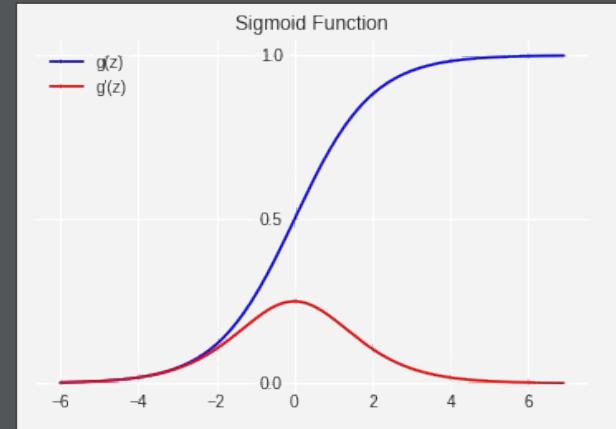
```
# create a figure  
fig, ax = plt.subplots(1,1,figsize=(6,4))  
  
# range of values from -6 to 7  
values = tf.range(-6,7,0.1,dtype=tf.float32)  
  
# calculate sigmoid function for all values  
sigmoid_values = tf.math.sigmoid(values)  
  
# plot values  
ax.plot(values.numpy(),sigmoid_values.numpy())
```



```
# 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)

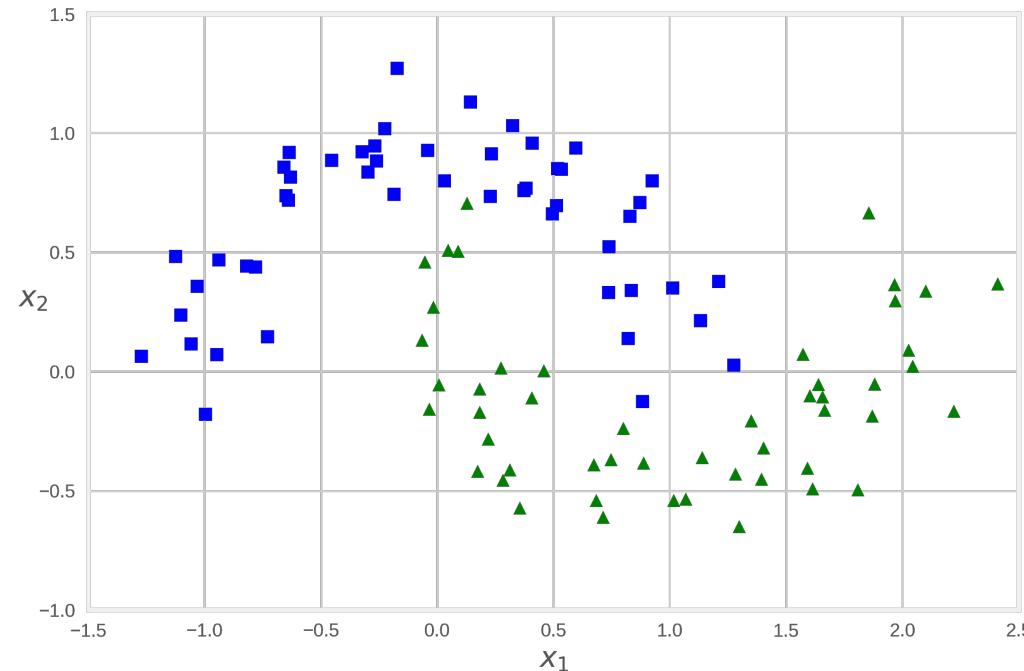
# What's the gradient of `sigmoid_values` with respect to `values`?
derivative_sigmoid = tape.gradient(sigmoid_values, values)
print(derivative_sigmoid)
```

```
>>> 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



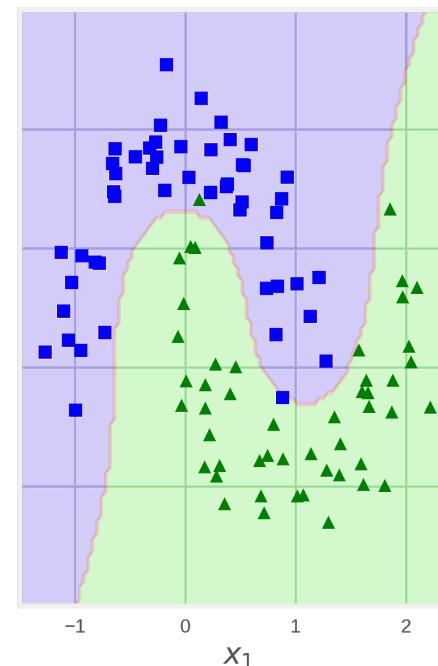
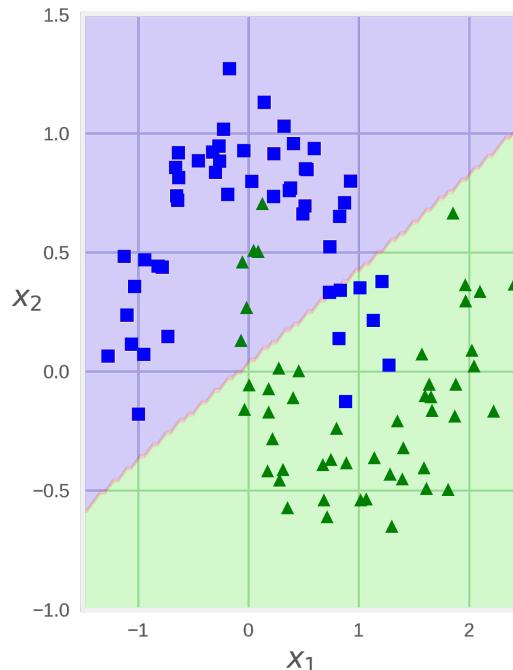
What if we wanted to build a neural network to distinguish blue vs green points?



Importance of Activation Functions

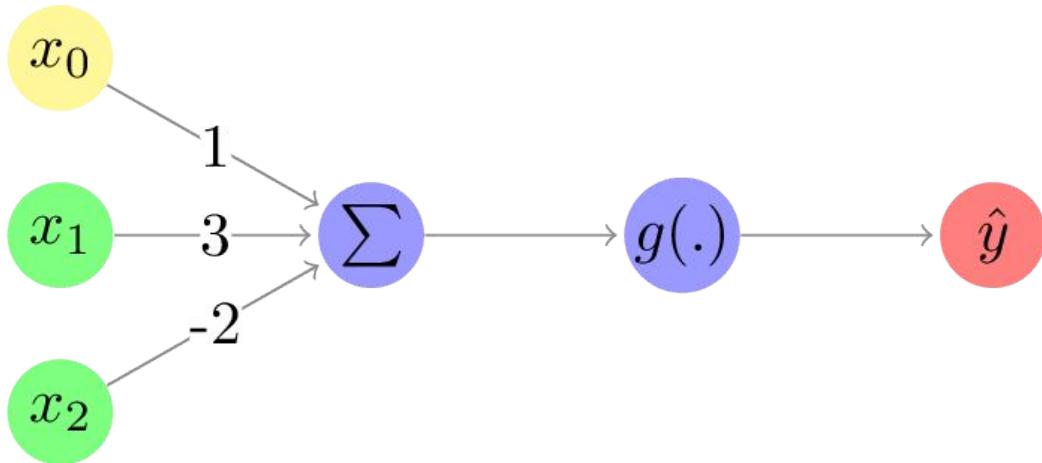
The purpose of activation functions is to introduce **non-linearities** into the network

Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

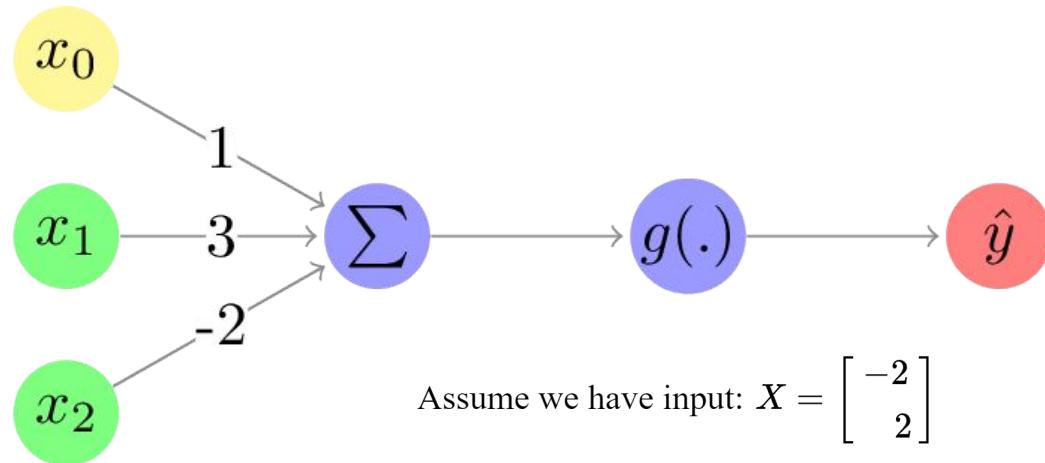


We have $w_0 = 1$ and $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

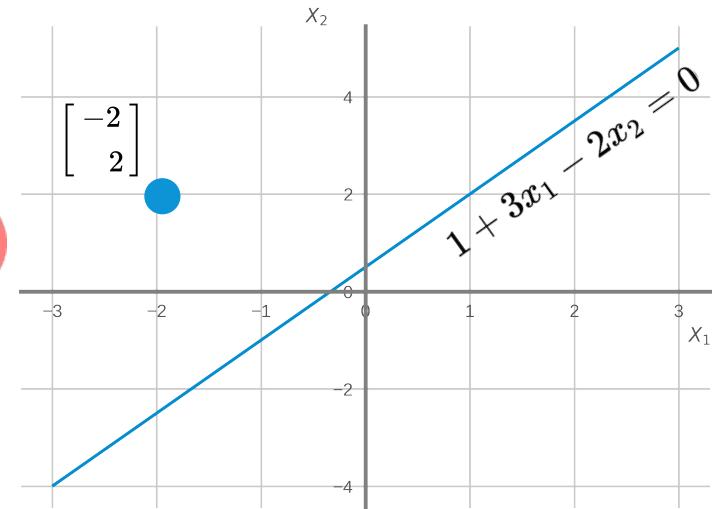
This is just a line in 2D!

The Perceptron: Example



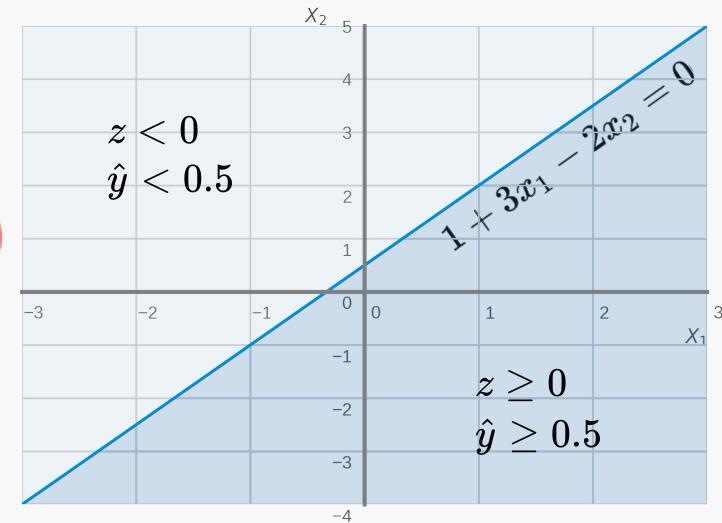
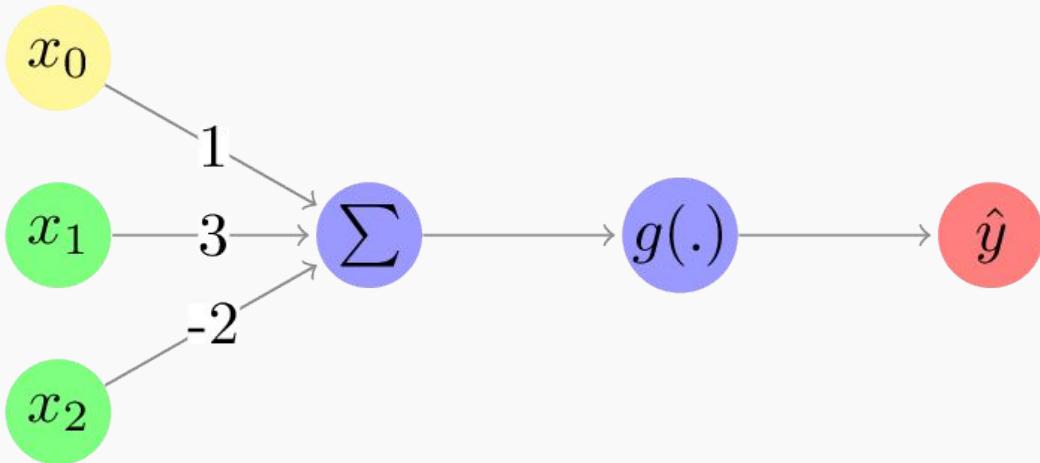
Assume we have input: $X = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$

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



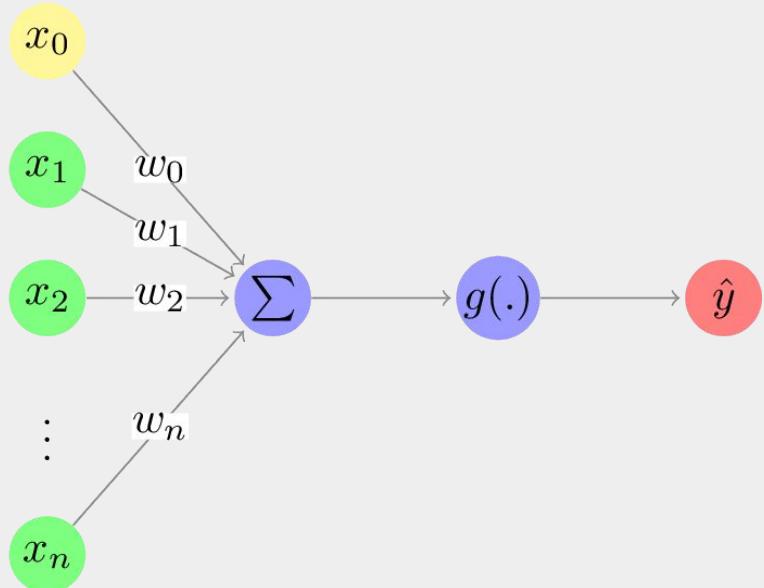
The Perceptron: Example

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

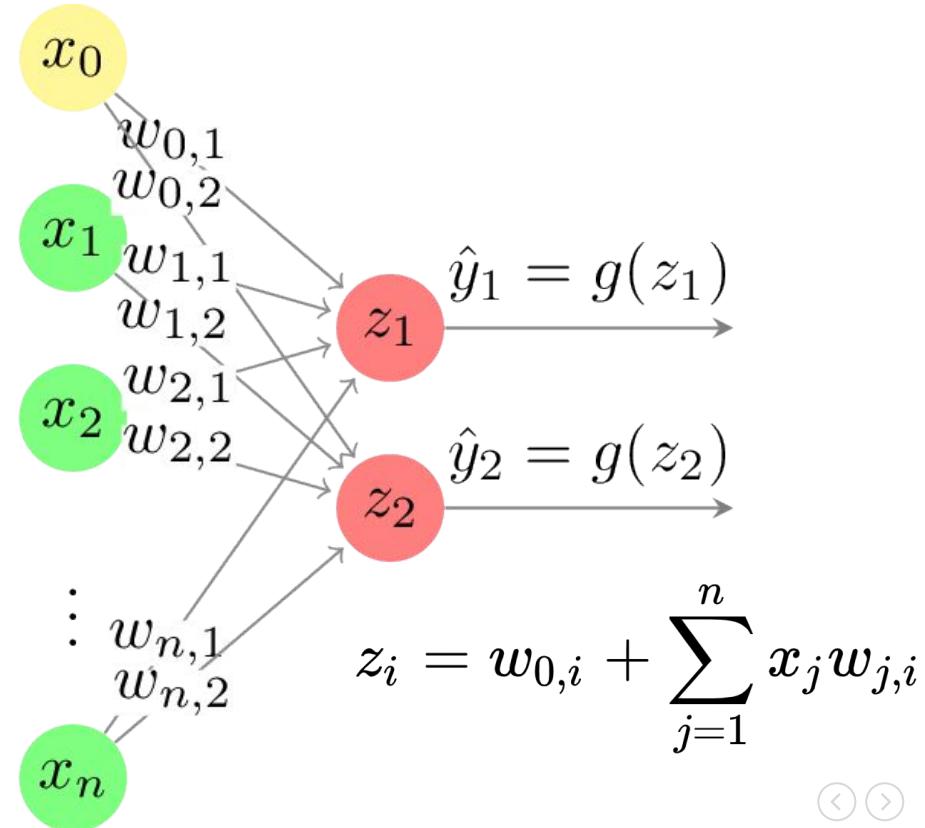
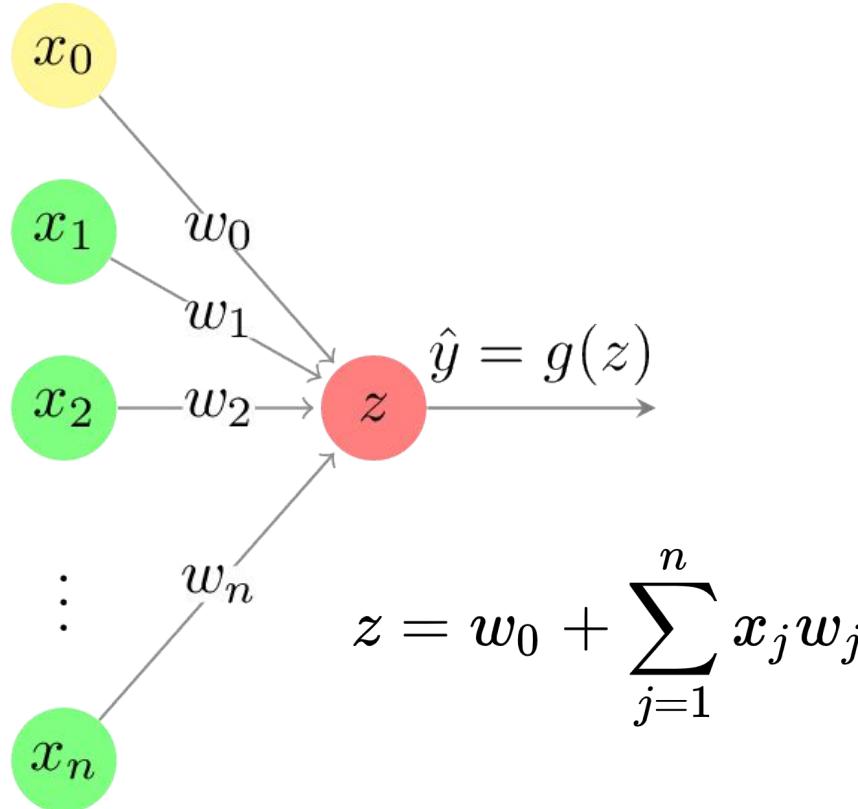


Building Neural Networks

With Perceptrons



The Perceptron: Simplified vs Multi Output



```
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
```

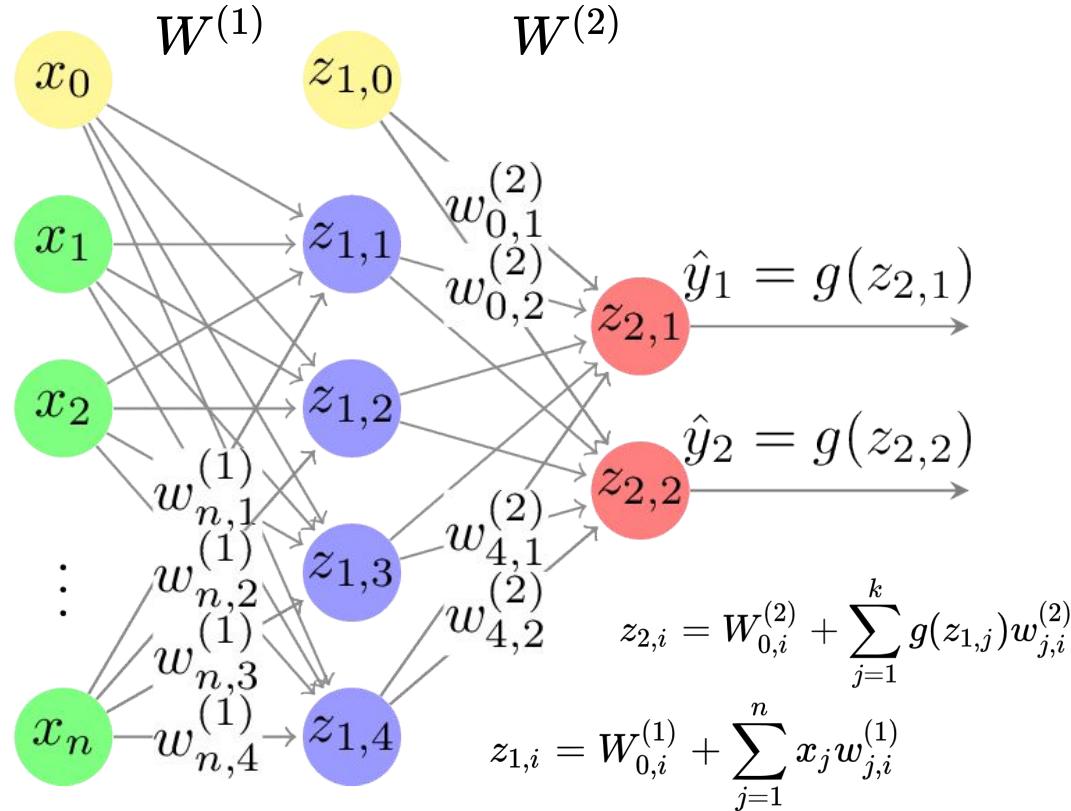
Dense layer from scratch



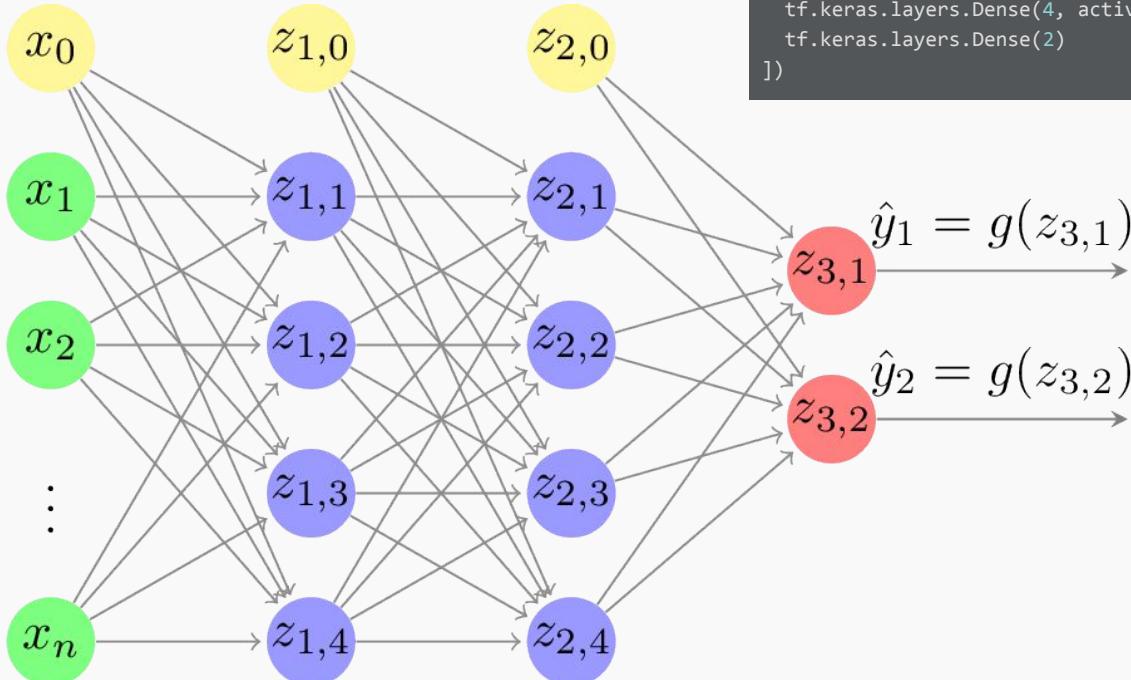
```
layer = tf.keras.layers.Dense(units=2, activation="sigmoid")
```



Single Hidden Layer Neural Network



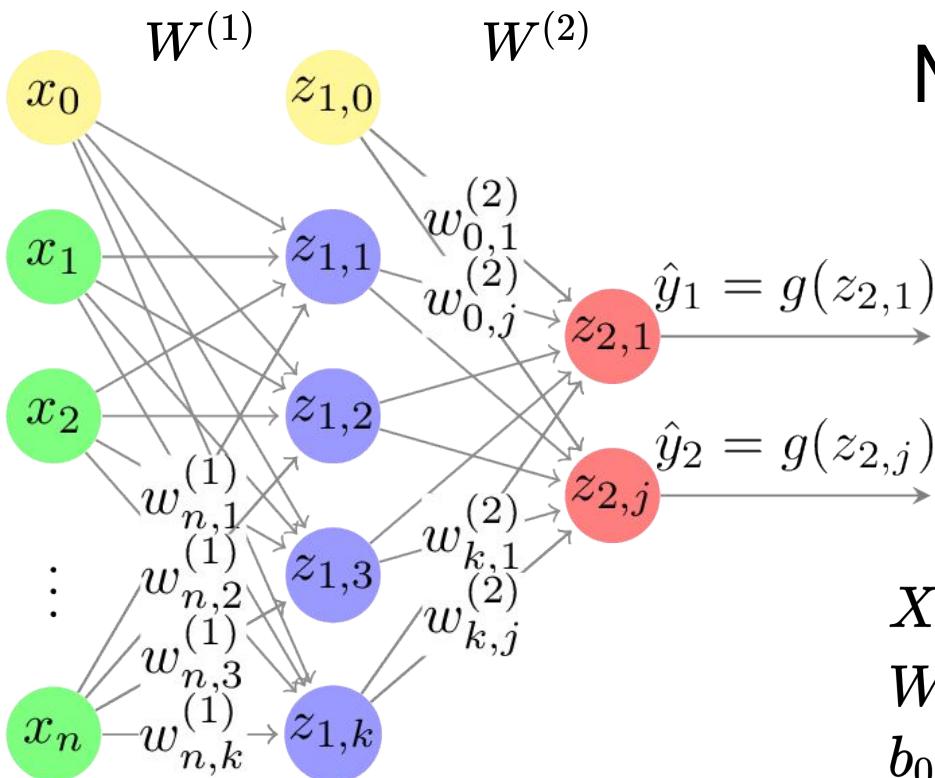
Multi Hidden Layer Neural Network



```
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



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

$$b_0 = x_0$$

$$a_1 = g(z_1)$$

$$z_2 = a_1 \times W^{(2)} + b_1$$

$$b_1 = z_{1,0}$$

$$a_2 = \hat{y} = g(z_2)$$

$$X \rightarrow 1 \times n$$

$$W^{(1)} \rightarrow n \times k$$

$$b_0 \rightarrow 1 \times k$$

$$z_1 \rightarrow 1 \times k$$

$$a_1 \rightarrow 1 \times k$$

$$W^{(2)} \rightarrow k \times j$$

$$b_1 \rightarrow 1 \times j$$

$$z_2 \rightarrow 1 \times j$$

$$a_2 \rightarrow 1 \times j$$

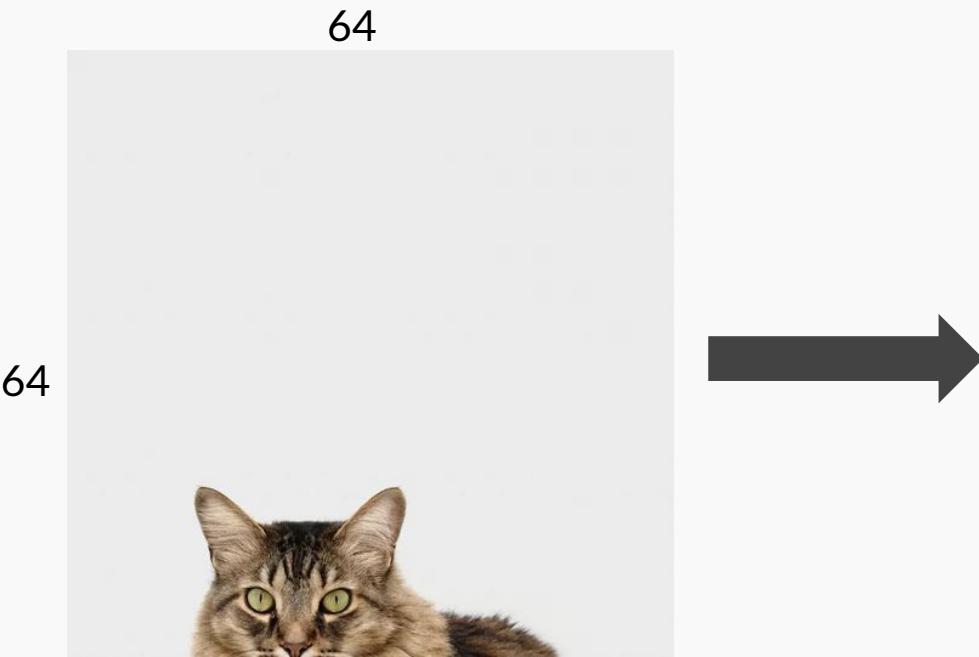
Applying Neural Networks

Cat vs Non Cat



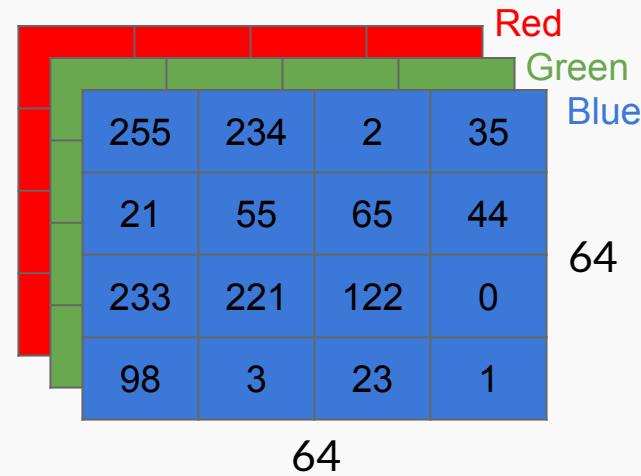
Example Problem

Binary Classification



Number of pixels

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

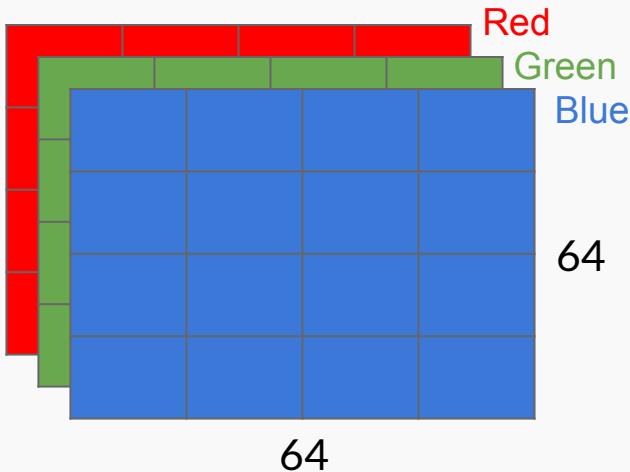


Example Problem

Binary Classification

Number of pixels

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



$$\begin{matrix} & y \\ Y^{(1)} & \left[\begin{matrix} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \end{matrix} \right] \\ Y^{(2)} & \\ Y^{(3)} & \\ \vdots & \\ Y^{(m)} & \end{matrix}$$

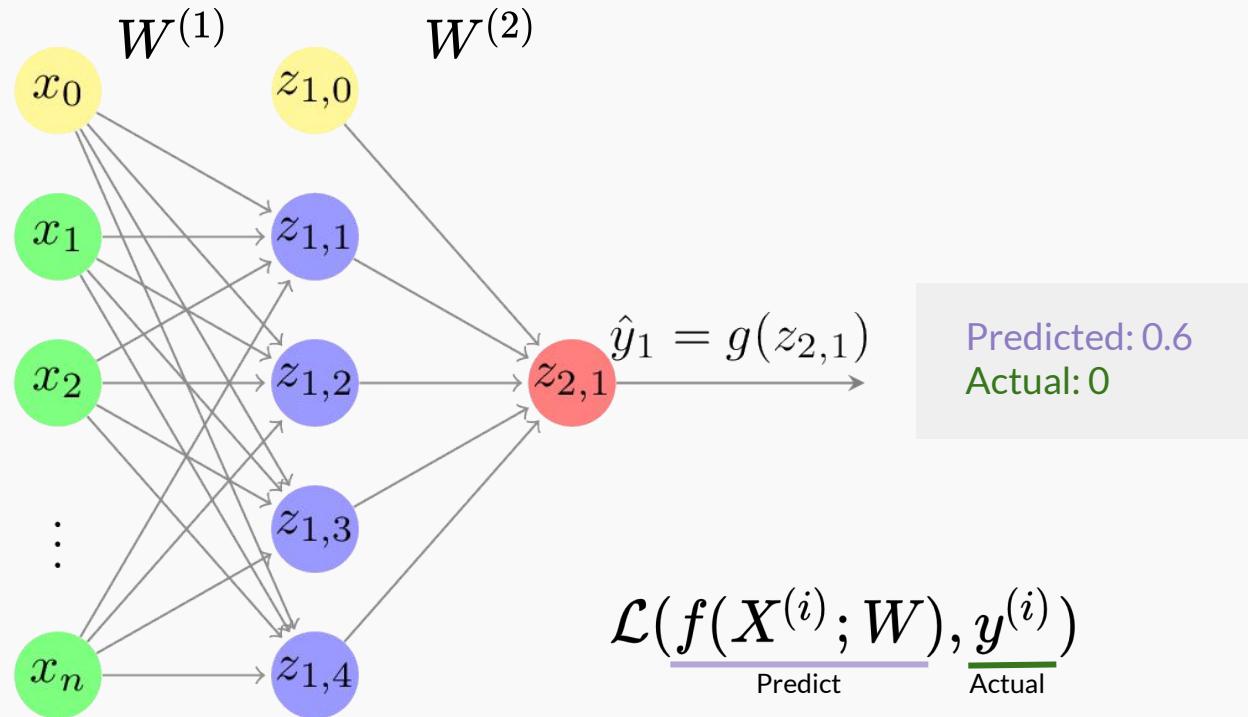
$$Y^{(i)} = \begin{cases} 0 & \text{if } y \text{ is not a cat} \\ 1 & \text{if } y \text{ is a cat} \end{cases}$$

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



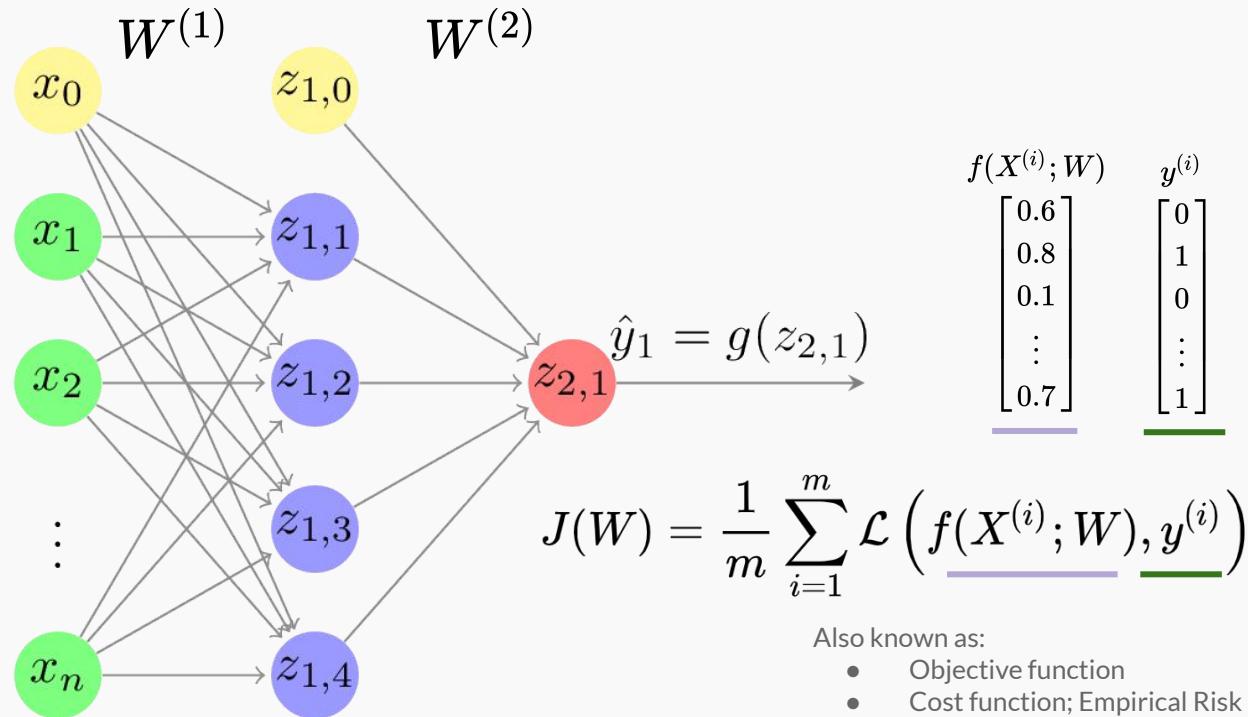
Quantifying Loss

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



Empirical Loss

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



Binary Cross Entropy Loss

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

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

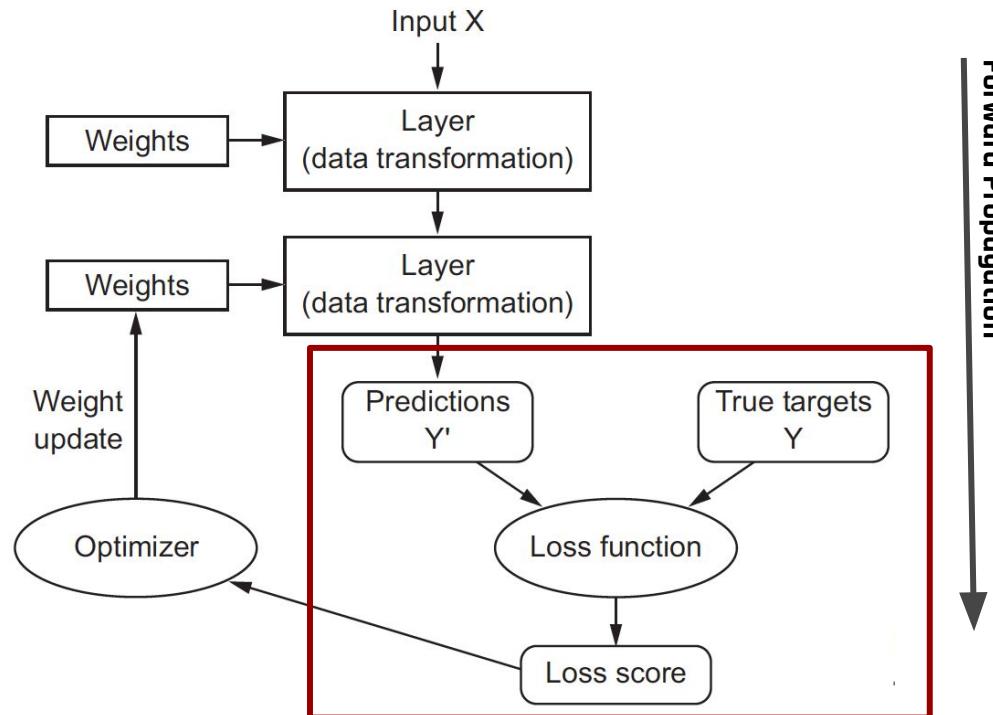
$$\mathcal{L} \left(\underline{\hat{y}^{(i)}}, \underline{y^{(i)}} \right) = -\underline{y^{(i)}} \log(\underline{\hat{y}^{(i)}}) - (1 - \underline{y^{(i)}}) \log(1 - \underline{\hat{y}^{(i)}})$$

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

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



Understanding how DL works



Training Neural Networks

Part #01
Loss optimization, Backprop.,
Batch size



Loss Optimization

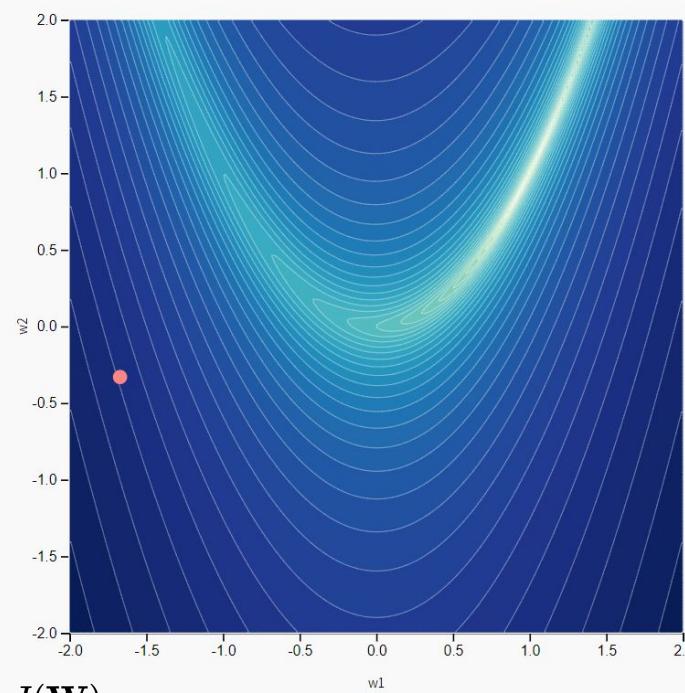
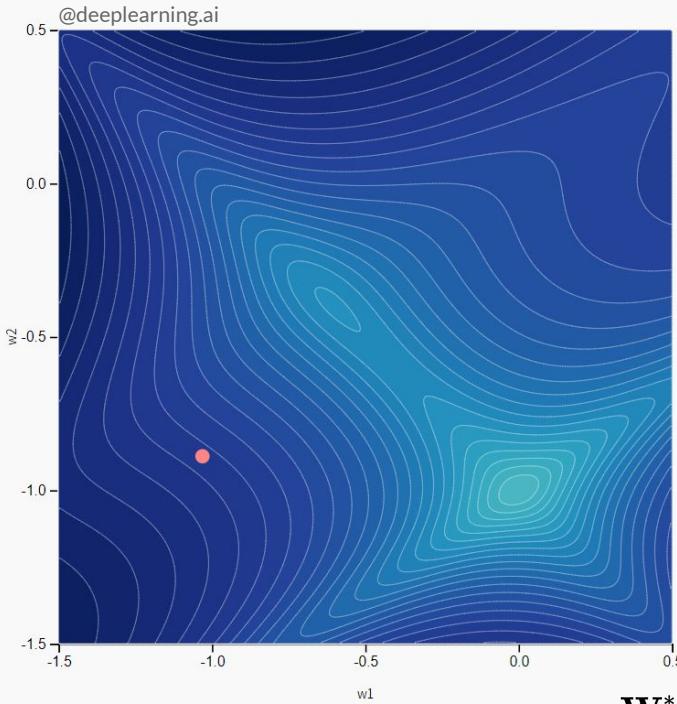
We want to find the network weights that achieve the lowest loss

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

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

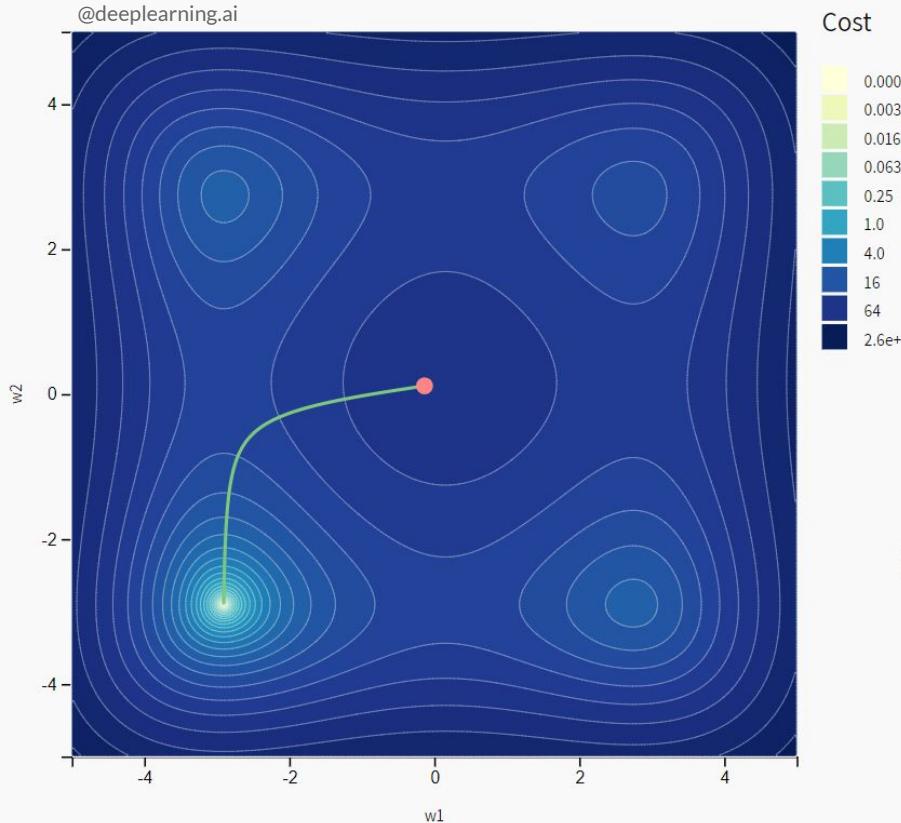
Loss Optimization

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

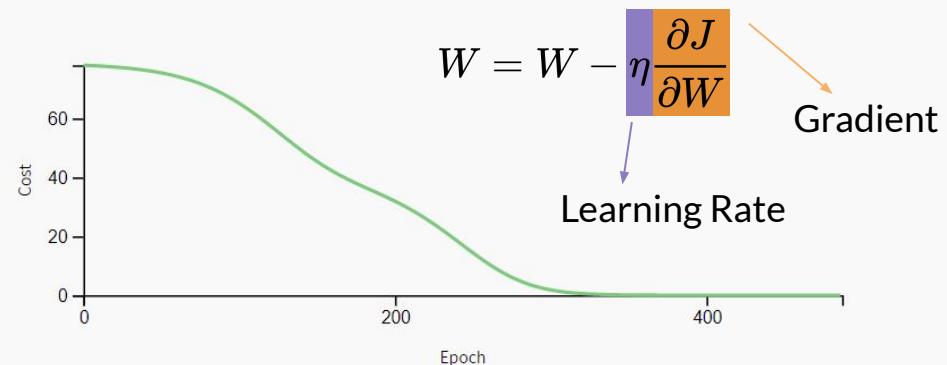


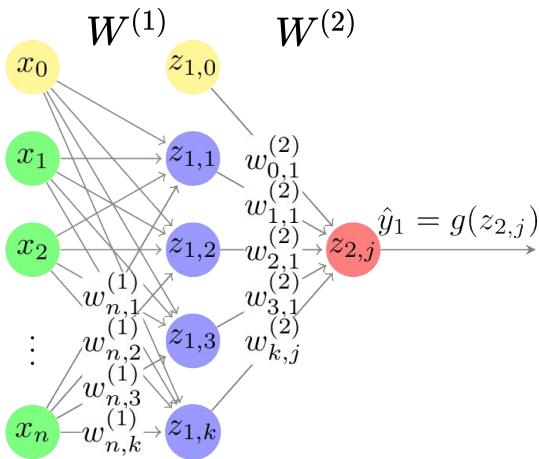
$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

Loss Optimization - Gradient Descent



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





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

$$a_1 = g(z_1)$$

$$z_2 = a_1 \times W^{(2)} + b_1$$

$$a_2 = g(z_2) = \hat{y}$$

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$X \rightarrow (1 \times n)$$

$$W^{(1)} \rightarrow (n \times k)$$

$$b_0 \rightarrow (1 \times k)$$

$$z_1 \rightarrow (1 \times k)$$

$$a_1 \rightarrow (1 \times k)$$

$$W^{(2)} \rightarrow (k \times j)$$

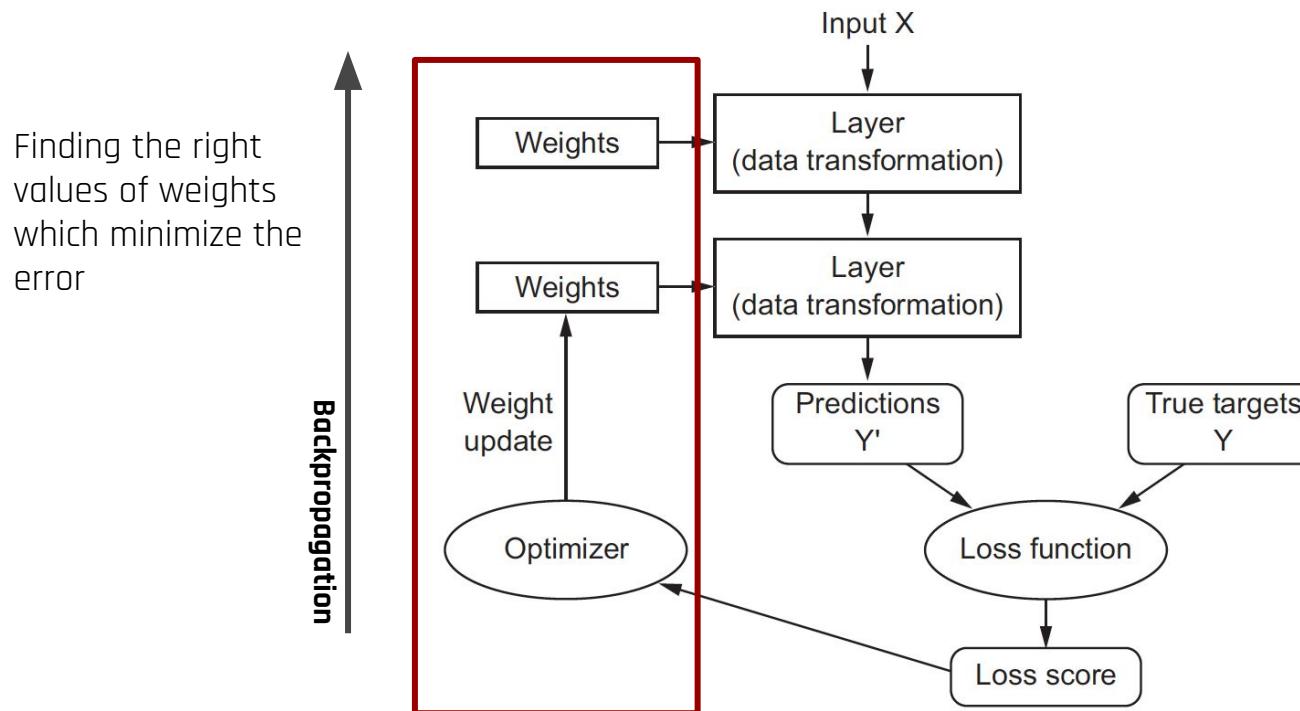
$$b_1 \rightarrow (1 \times j)$$

$$z_2 \rightarrow (1 \times j)$$

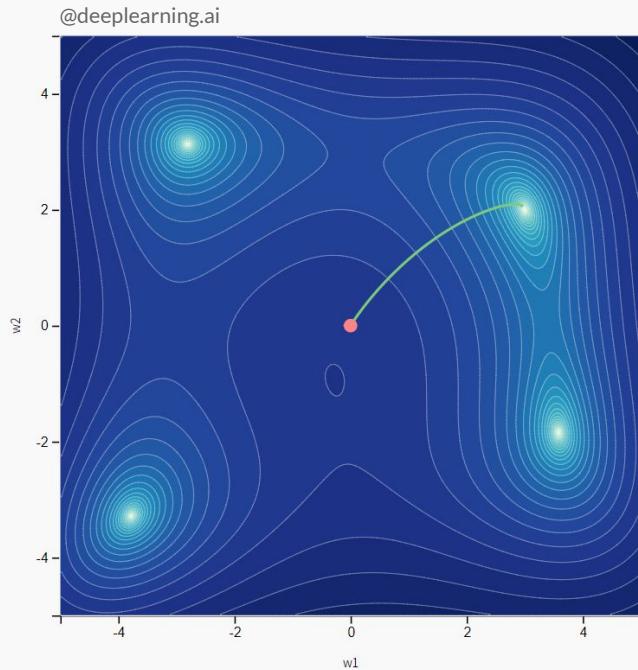
$$a_2 \rightarrow (1 \times j)$$

Backpropagation with Pencil and Paper - See the Video

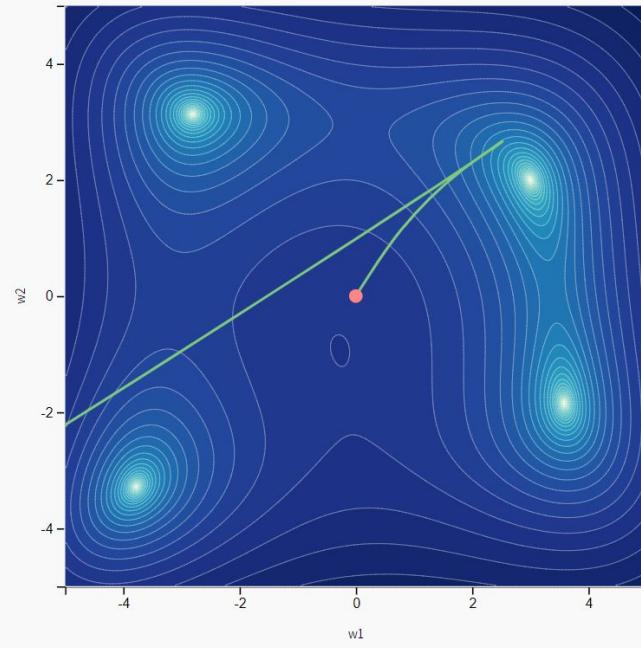
Understanding how DL works



Loss Functions Can Be Difficult to Optimize



Small learning rate ($\text{lr}=0.001$)
converges slowly



Large learning rate ($\text{lr}=0.1$) 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

$$W = W - \eta$$

?

$\beta, \beta_1, \beta_2, learning_decay$

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
 $1 \leq b \leq m$



```
# 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)
```

Model



```
# 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')
])
```

Loss



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

Evaluation Metrics



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

Optimizer



```
# 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)

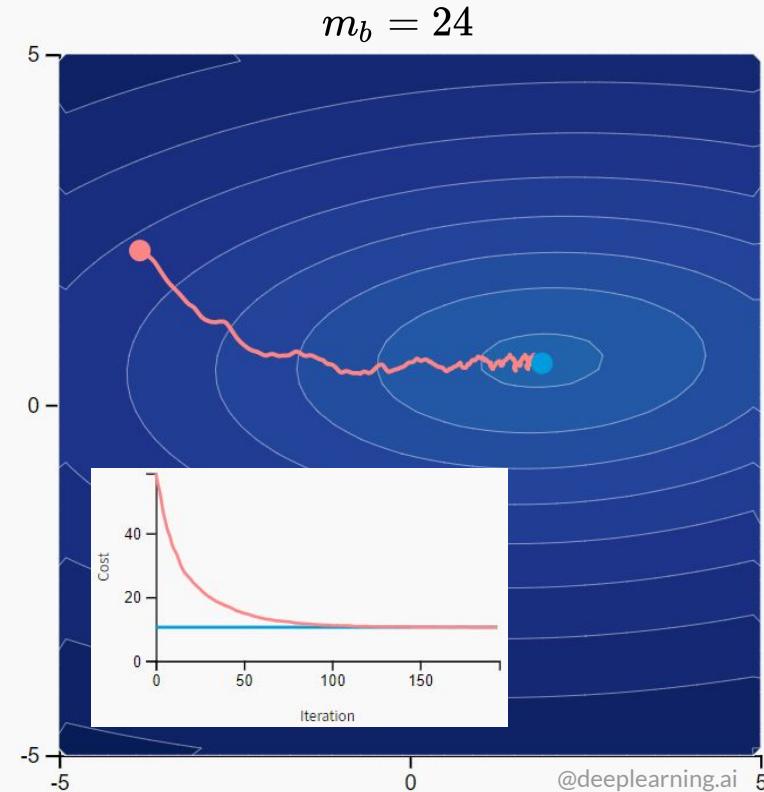
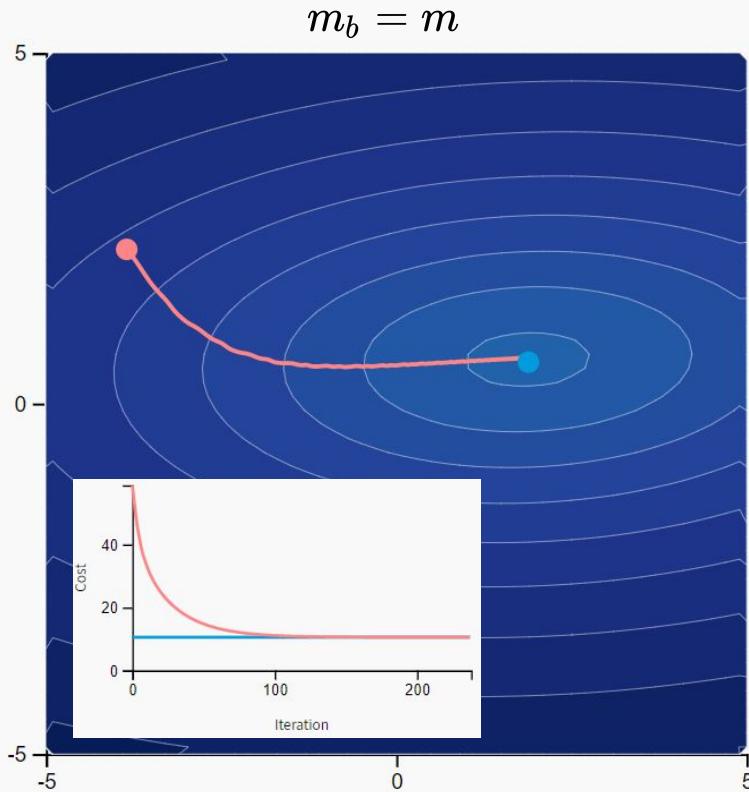
            # Update the weights of our linear layer.
            optimizer.apply_gradients(zip(gradients, model.trainable_weights))

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

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



Mini-Batches Challenges



Training Neural Networks

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



TRAIN A DEEP NEURAL NETWORK

=

MINI-BATCH

+

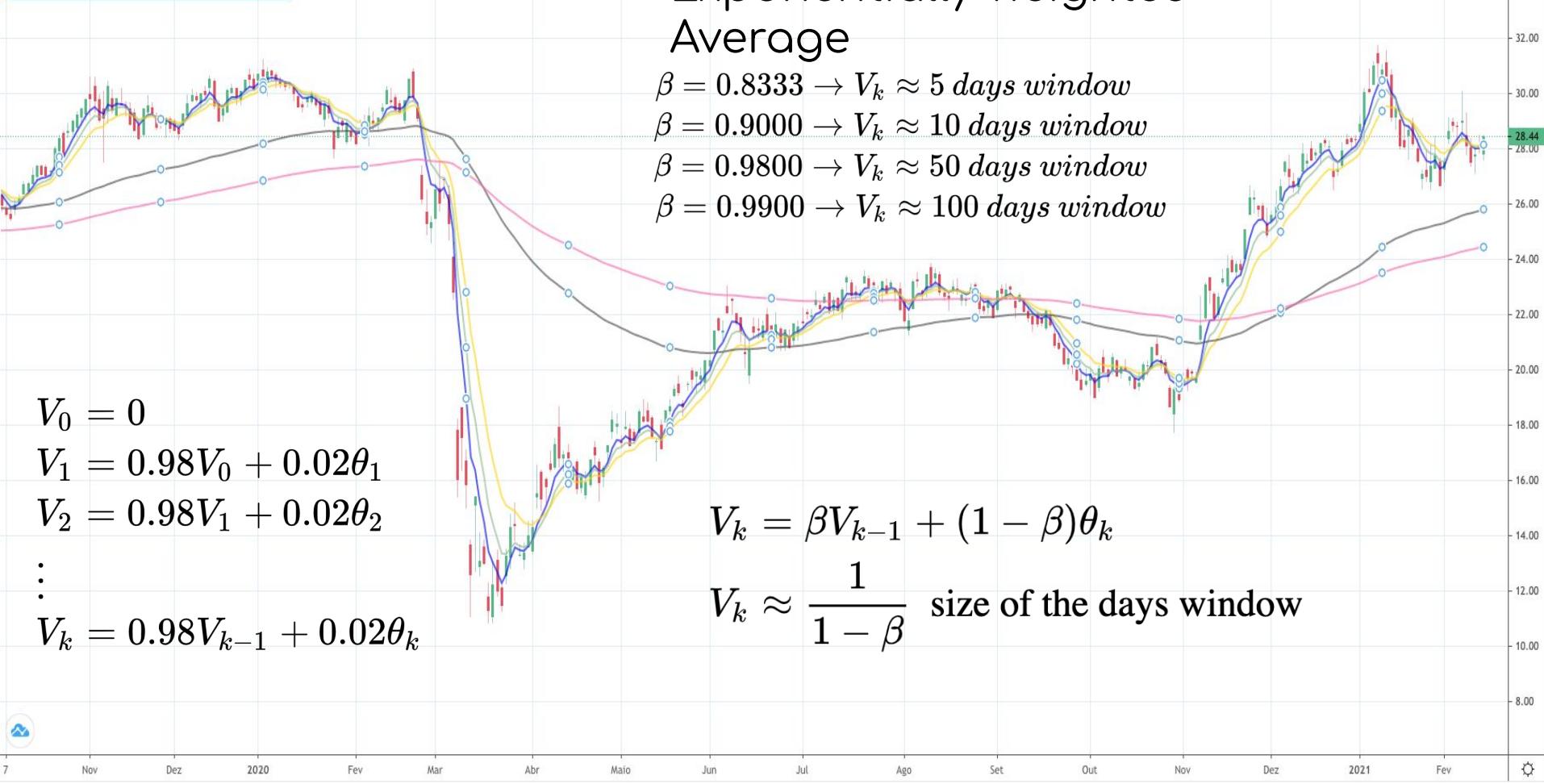
OPTIMIZATION ALGORITHMS



Optimization Algorithms

Exponentially Weighted Average

Adam { Momentum
RMSprop



Exponentially Weighted Average

$\beta = 0.8333 \rightarrow V_k \approx 5$ days window

$\beta = 0.9000 \rightarrow V_k \approx 10 \text{ days window}$

$\beta = 0.9800 \rightarrow V_{k_i} \approx 50 \text{ days window}$

$\beta = 0.9900 \rightarrow V_k \approx 100$ days window

$$V_0 = 0$$

$$V_1 = 0.98V_0 + 0.02\theta_1$$

$$V_2 = 0.98V_1 + 0.02\theta_2$$

•
•
•

$$V_k = 0.98V_{k-1} + 0.02\theta_k$$

$$V_k = \beta V_{k-1} + (1 - \beta) \theta_k$$

$$V_k \approx \frac{1}{1 - \beta} \text{ size of the days window}$$

28.44 0.00 28.44

M  dias M  veis 5, 8, 13, 100, 200

@ {} X 2 24.47



Exponentially Weighted Average Bias Correction

$$V_0 = 0$$

$$V_1 = 0.98V_0 + 0.02\theta_1$$

$$V_2 = 0.98V_1 + 0.02\theta_2$$

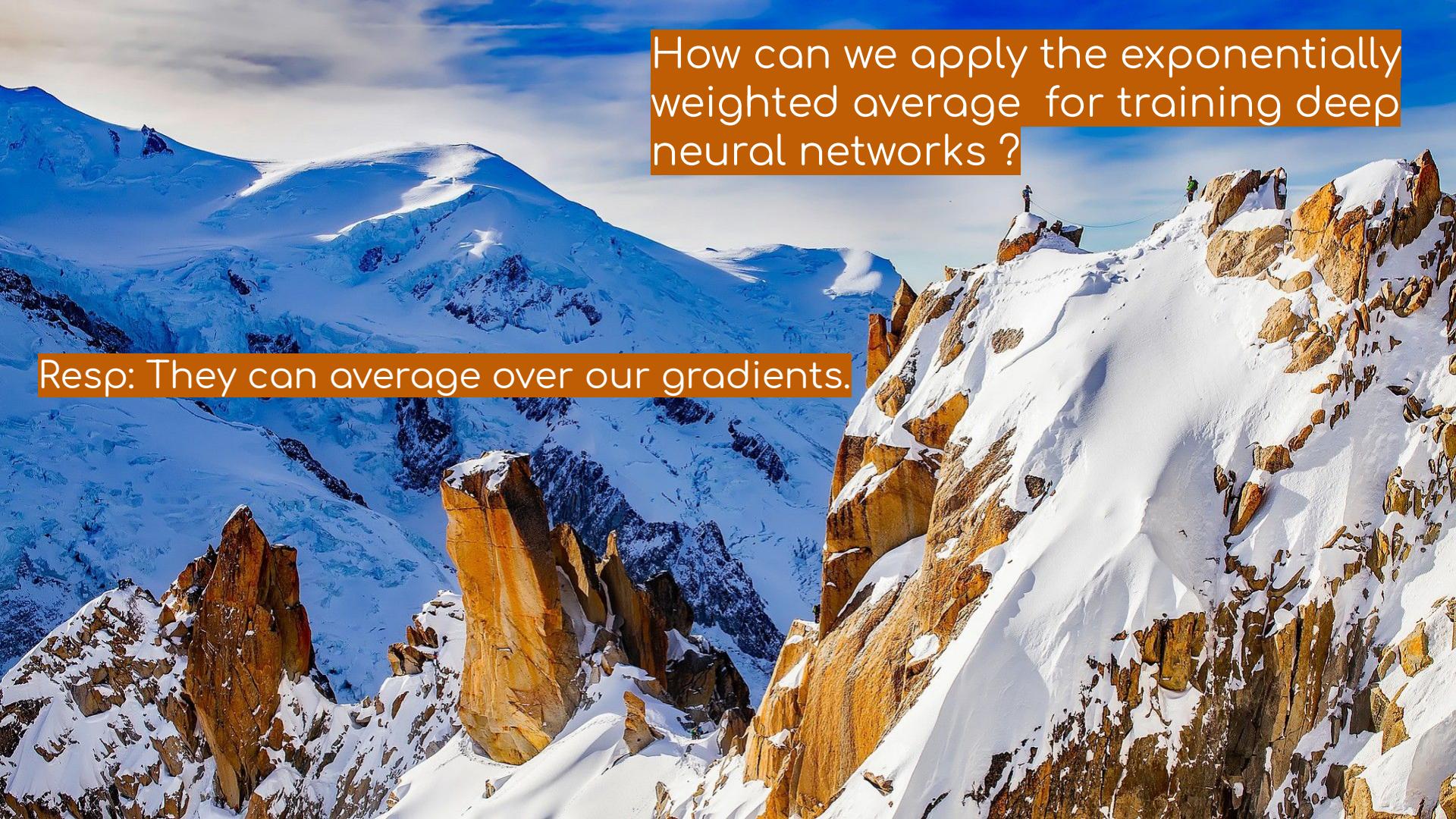
$$V_2 = 0.98 * 0.02\theta_1 + 0.02\theta_2$$

$$V_2 = 0.0196\theta_1 + 0.02\theta_2$$

$$\hat{V}_k = \text{bias_correction} = \frac{V_k}{1 - \beta^k}$$

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

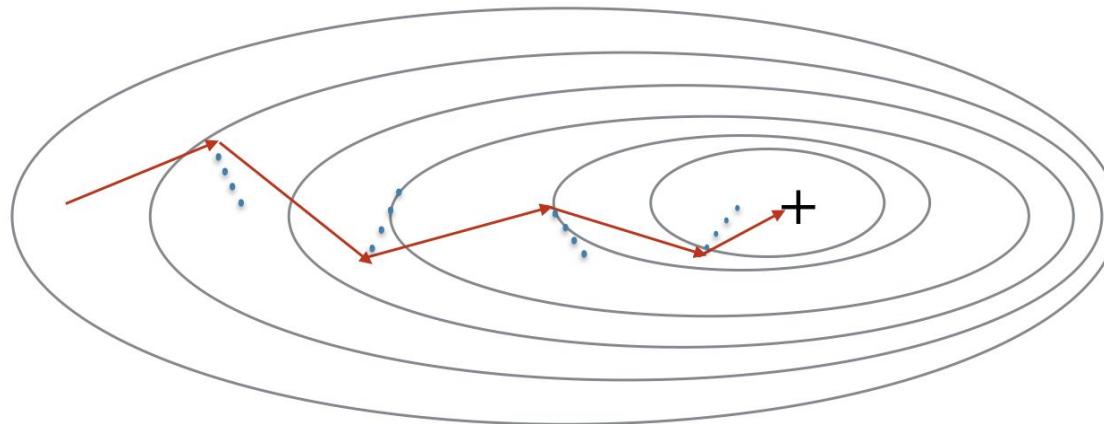
$$\hat{V}_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$



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

On iteration t

Compute dW, db on the current mini_batch

$$v_{dw} = \beta v_{dw} + (1 - \beta) dW$$

$$v_{db} = \beta v_{db} + (1 - \beta) db$$

$$W = W - \eta v_{dw}$$

$$b = b - \eta v_{db}$$

Hyperparameters : η, β $\beta = 0.9$

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 : η, β $\beta = 0.9, \epsilon = 1e^{-8}$

Gradient Descent with Adam

On iteration t

Compute dW, db on the current mini_batch

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

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

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

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

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

Hyperparameters : η, β_1, β_2 $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e^{-8}$



Learning Rate Decay

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

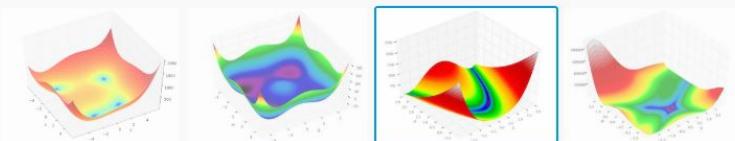
$$\eta = \frac{1}{1 + learning_decay \times epoch_num} \times \eta_0$$

$$\eta = 0.95e^{epoch_num} \eta_0$$

$$\eta = \frac{k}{\sqrt{epoch_num}} \times \eta_0$$

1. Choose a cost landscape

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



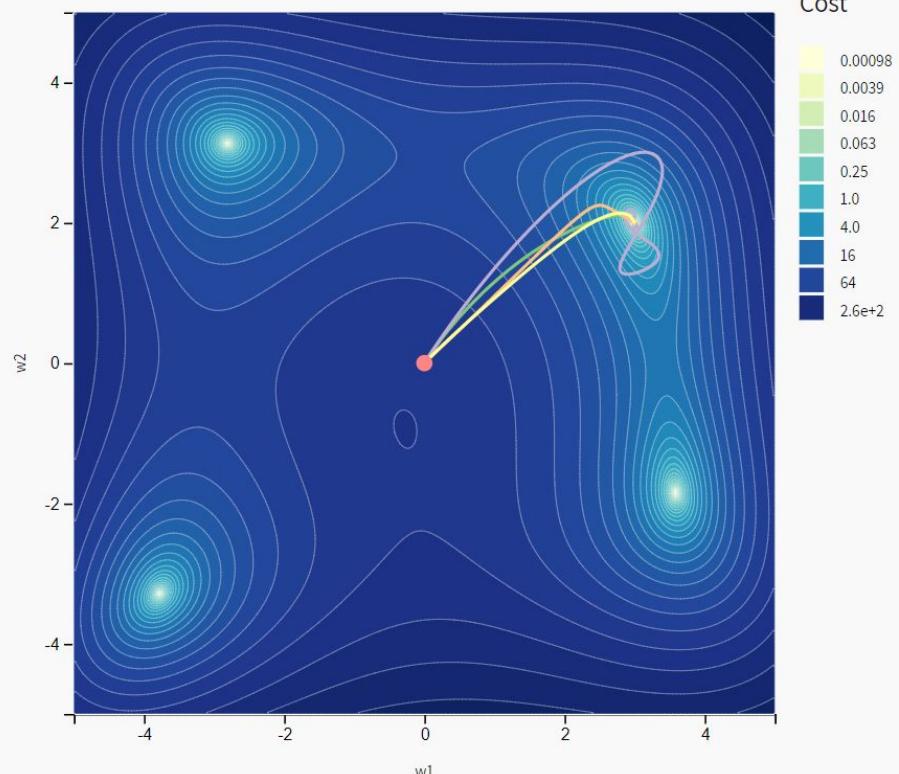
2. Choose initial parameters

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
<input checked="" type="checkbox"/> Gradient Descent	0,001	0
<input checked="" type="checkbox"/> Momentum	0,001	0
<input checked="" type="checkbox"/> RMSprop	0,001	0
<input checked="" type="checkbox"/> 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