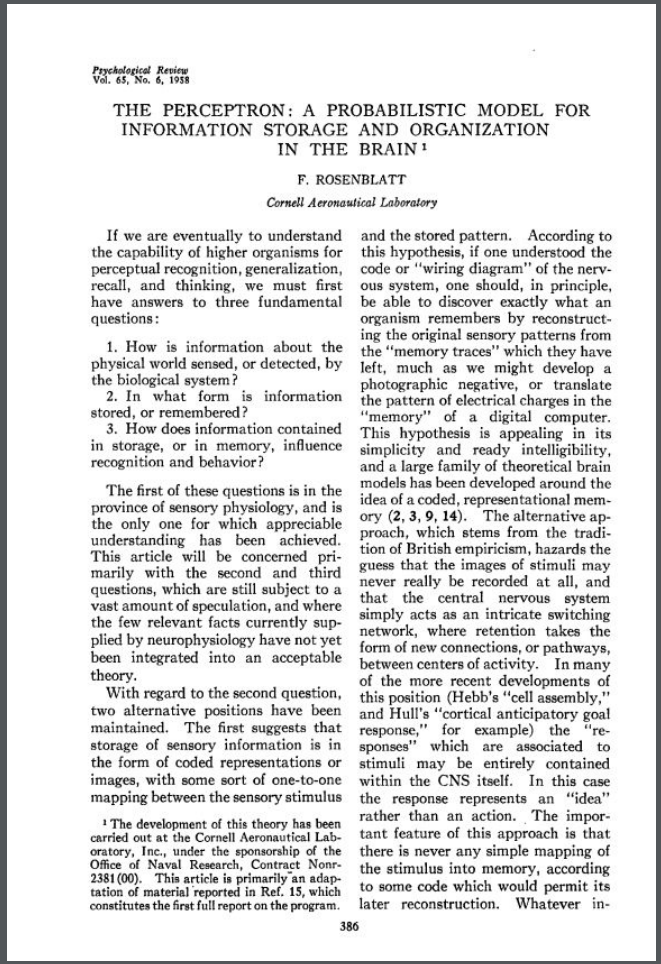
Introduction to Deep Learning and TensorFlow

Lesson #09

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:

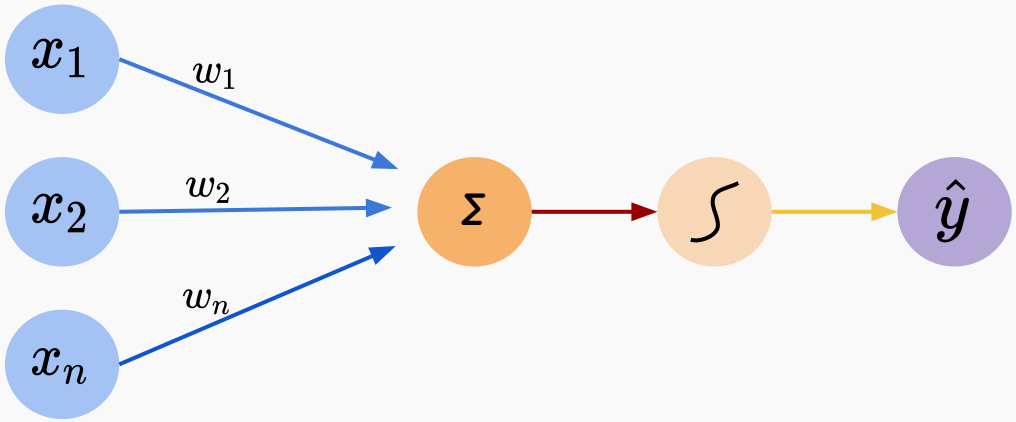
1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

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



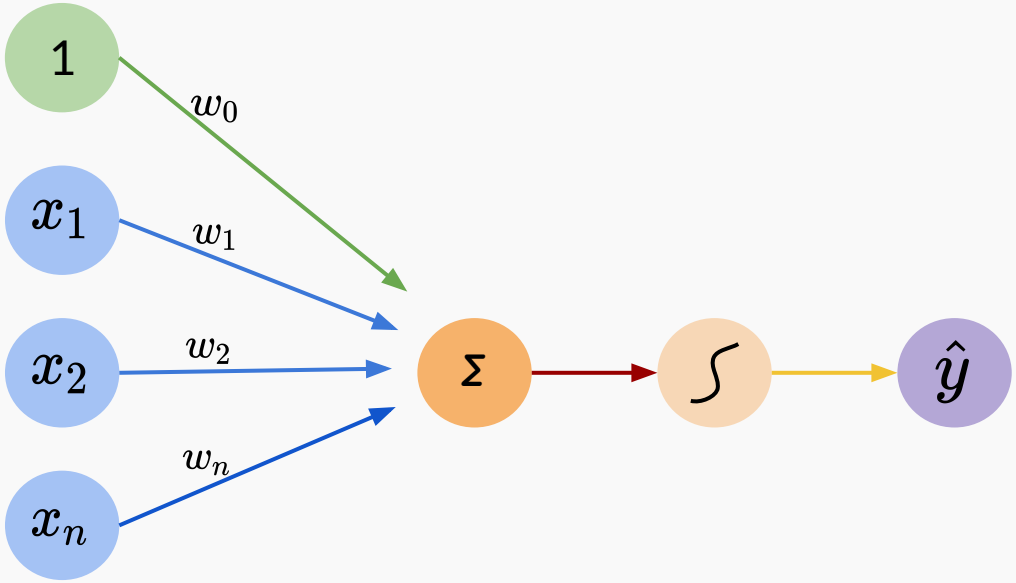
Linear combination of inputs

Output

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

Non-linear activation function

The Perceptron: Forward Propagation



Output

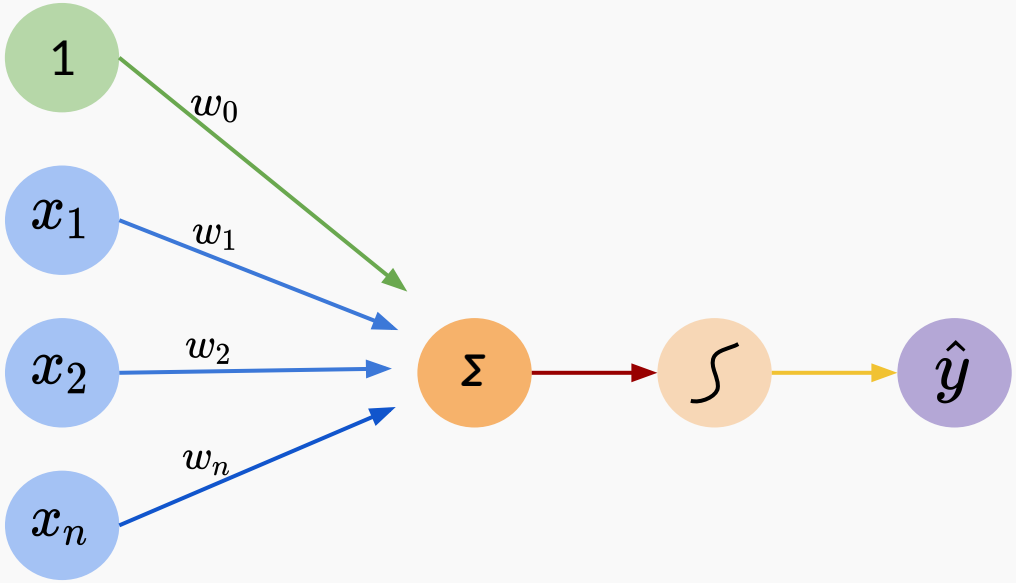
Linear combination of inputs

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

Non-linear activation function

Bias

The Perceptron: Forward Propagation

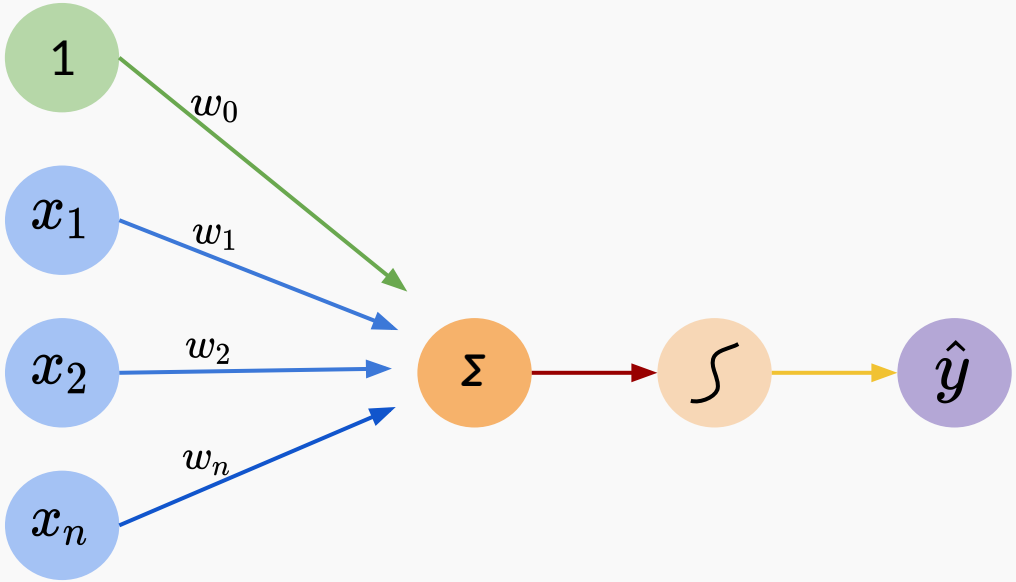


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} x_i w_i\right)$$

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

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

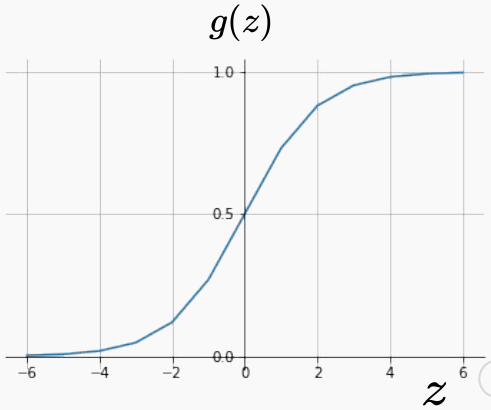


Activation Functions

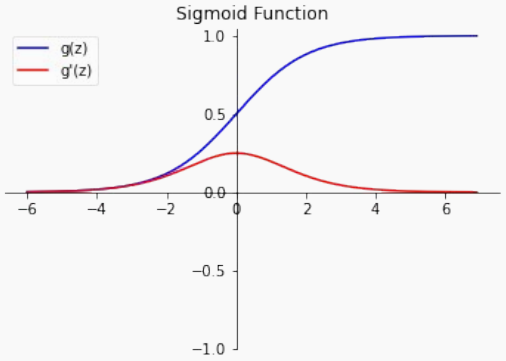
$$\hat{y} = g\left(w_0 + X^T W\right)$$

Example: sigmoid function

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

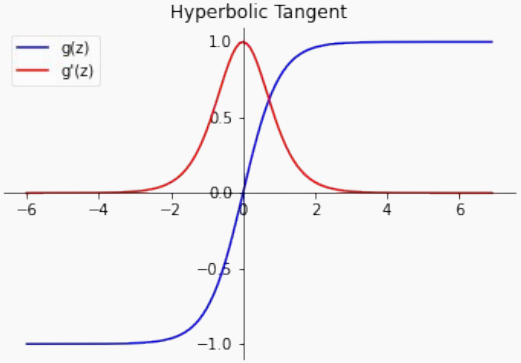


Common Activation Functions



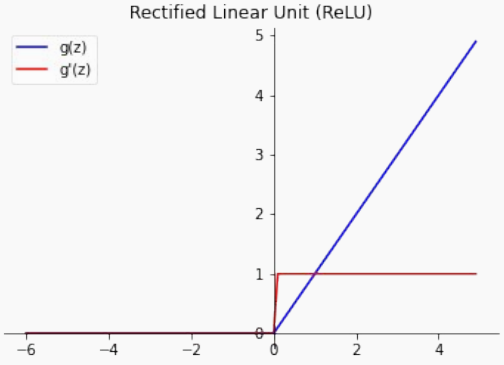
$$g(z) = \frac{1}{1 + e^{-z}}$$

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



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

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



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

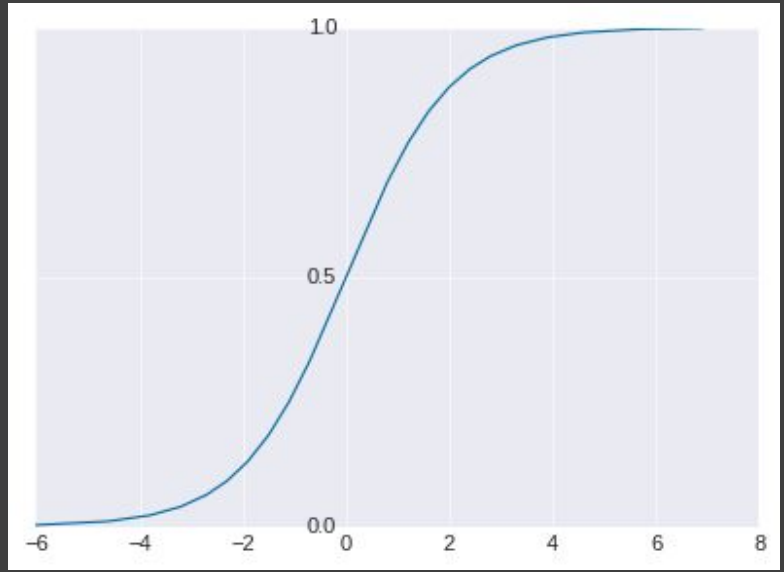
$$g'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

```
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()
```

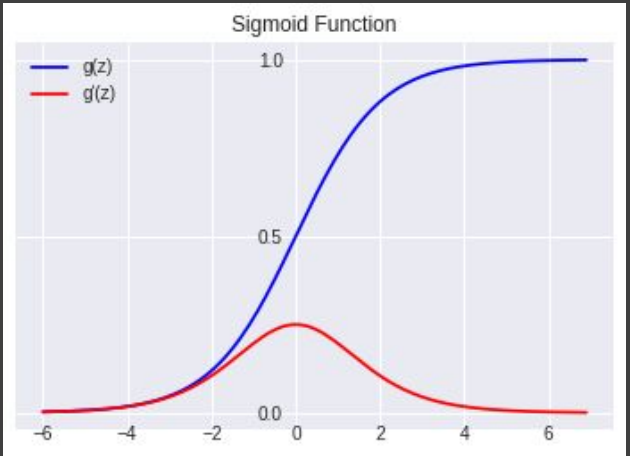
```
>> 0.8807971
```

```
# 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())
```



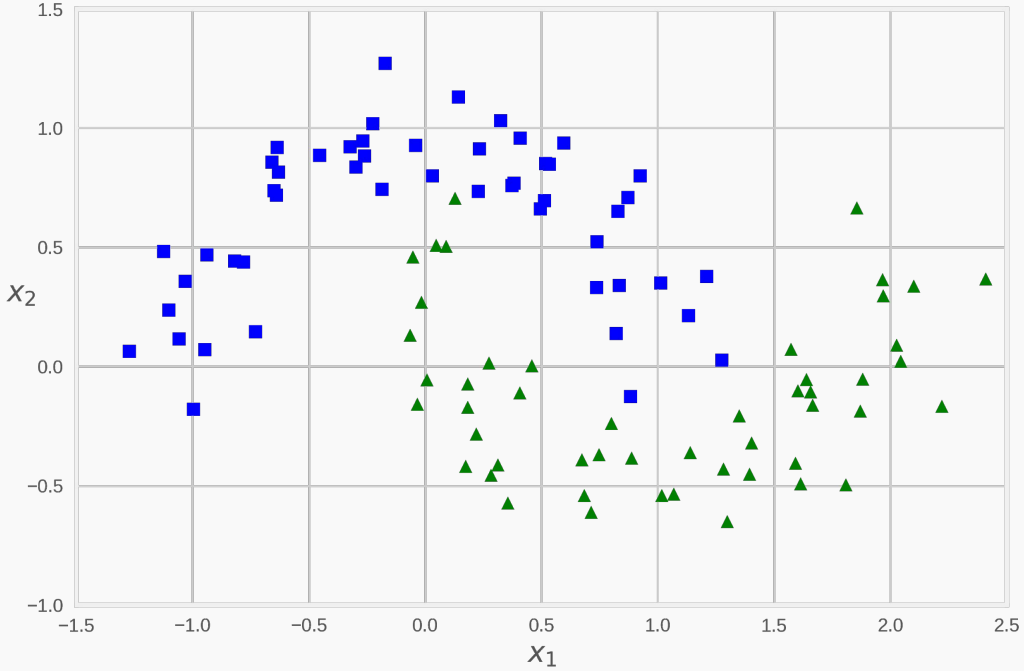
```
# create a range of values
values =  tf.range(-6,7,1,dtype=tf.float32)
# calculates the derivative of sigmoid function
with tf.GradientTape() as tape:
  # Start recording the history of operations applied to 'values'
  tape.watch(values)
  sigmoid_values = tf.math.sigmoid(values)

  # 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

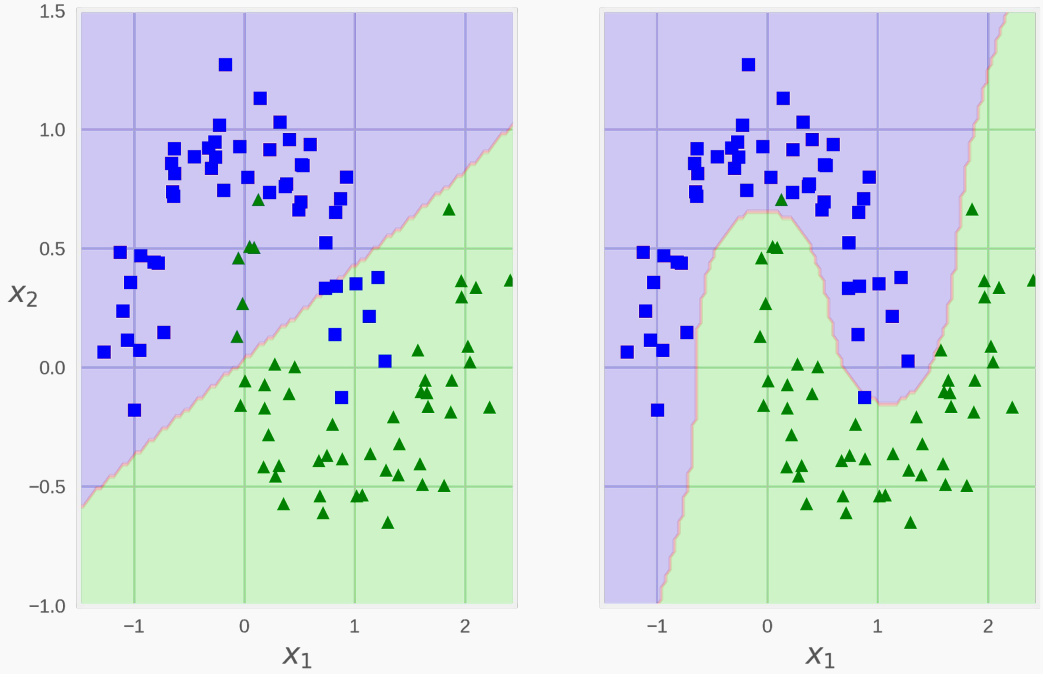


What if we wanted to build a neural network to distinguish blue vs green points?

Importance of Activation Functions

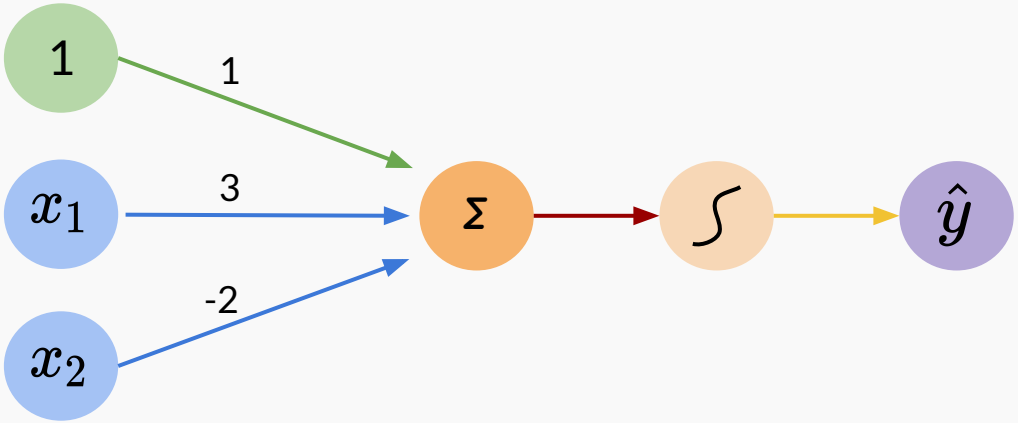
The purpose of activation functions is to introduce **non-linearities** into the network

Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

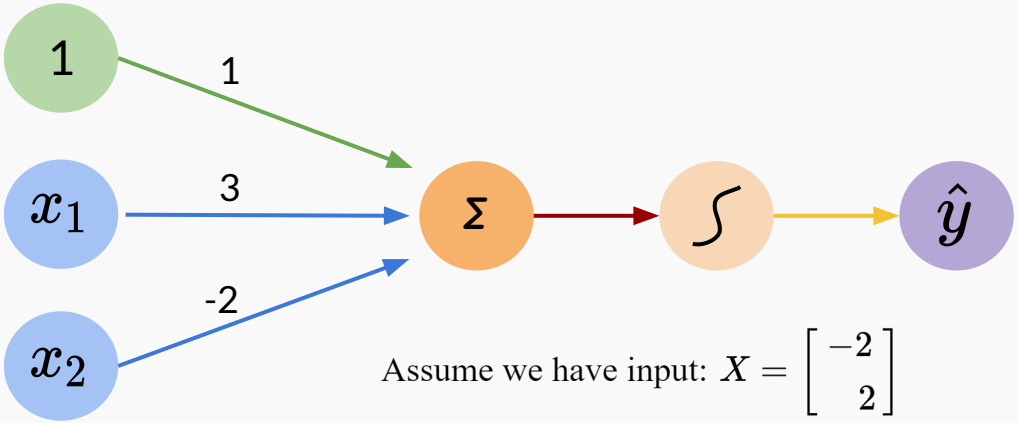


We have $w_0 = 1$ *and* $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

This is just a line in 2D!

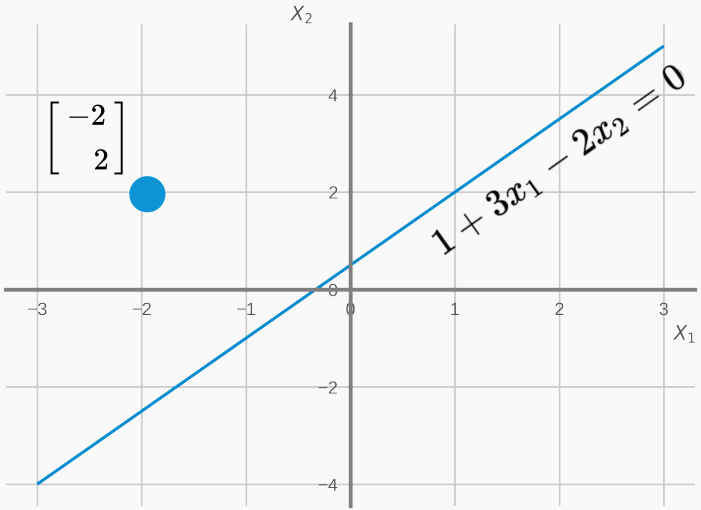
The Perceptron: Example



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

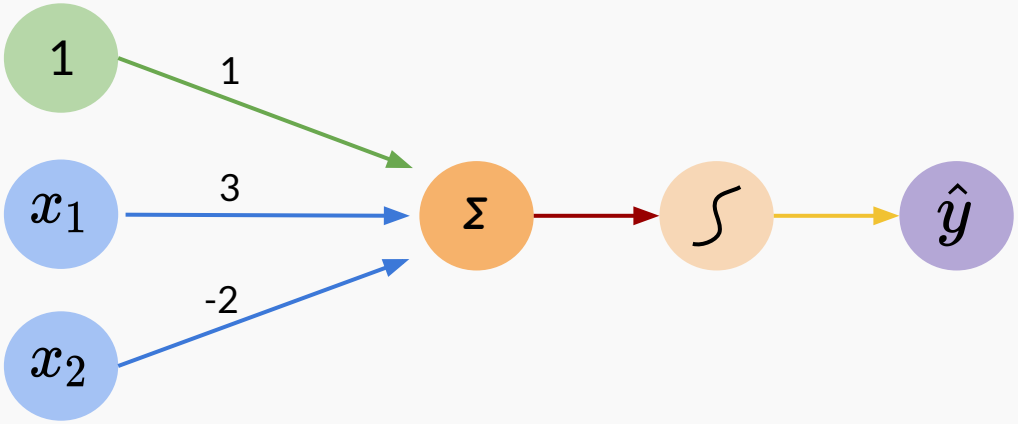
$$\begin{aligned} \hat{y} &= g\left(1 + (3 \times -2) - (2 \times 2)\right) \\ &= g\left(-9\right) \approx 0.00012338161 \end{aligned}$$

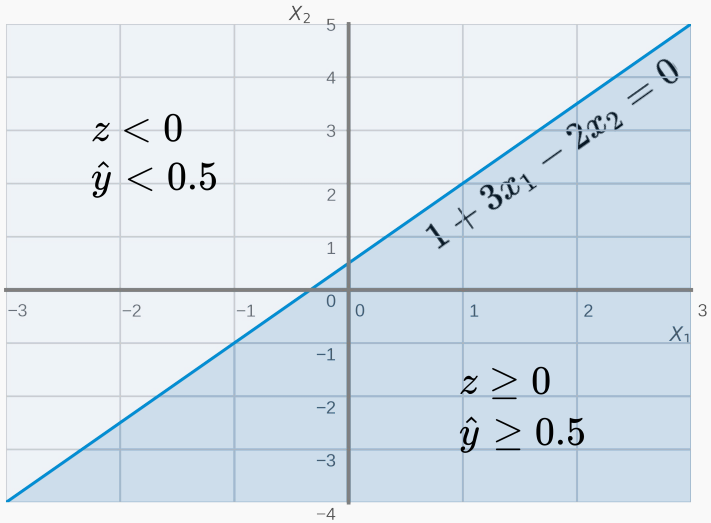
$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$



The Perceptron: Example

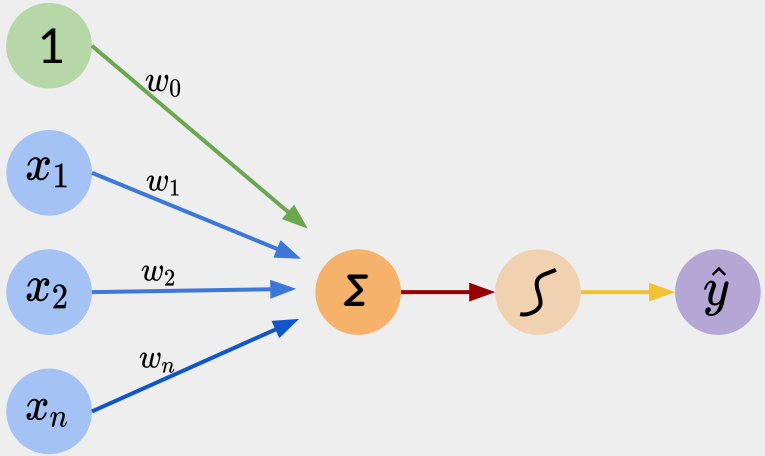
$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$





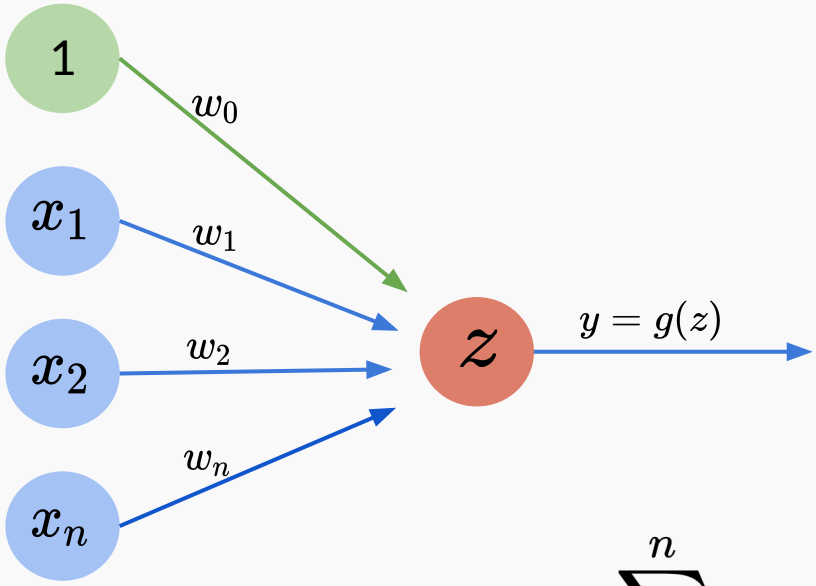
Building Neural Networks

With Perceptrons



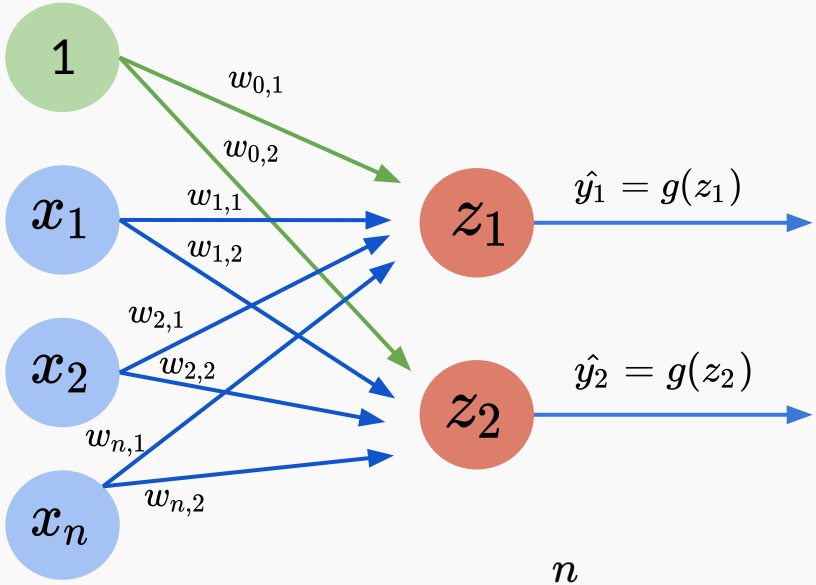
Lesson #09

The Perceptron: Simplified



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

Multi Output Perceptron



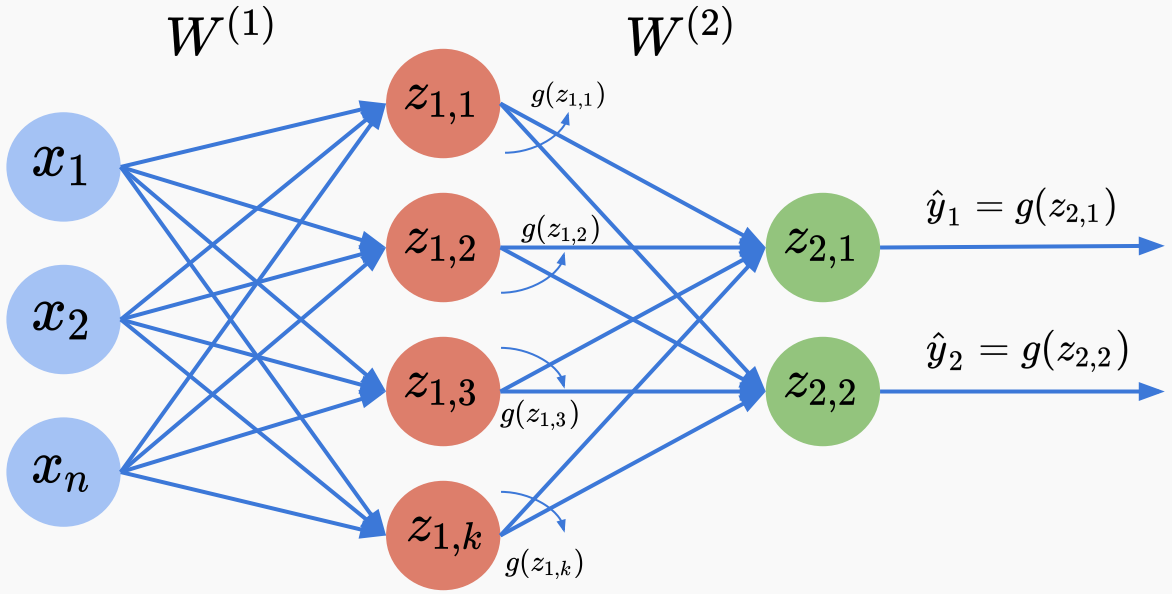
$$z_i = w_{0,i} + \sum_{j=1}^{n} x_j w_{j,i}$$

Dense layer from scratch

```
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



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

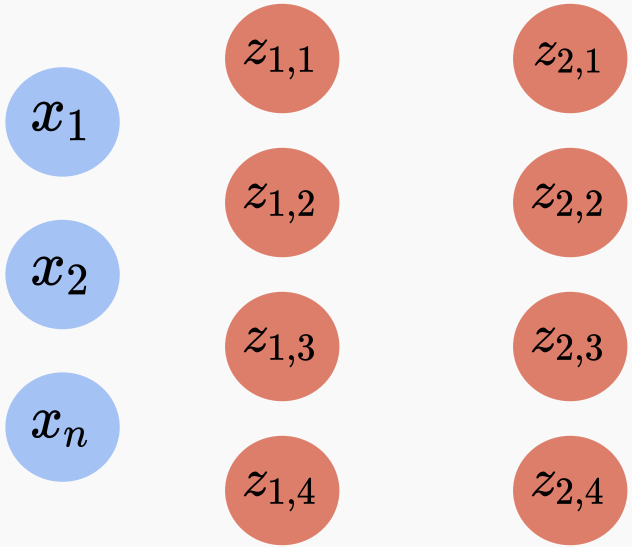
$$z_{2,i} = W_{0,i}^{(2)} + \sum_{j=1}^{k} g(z_{1,j}) w_{j,i}^{(2)}$$

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

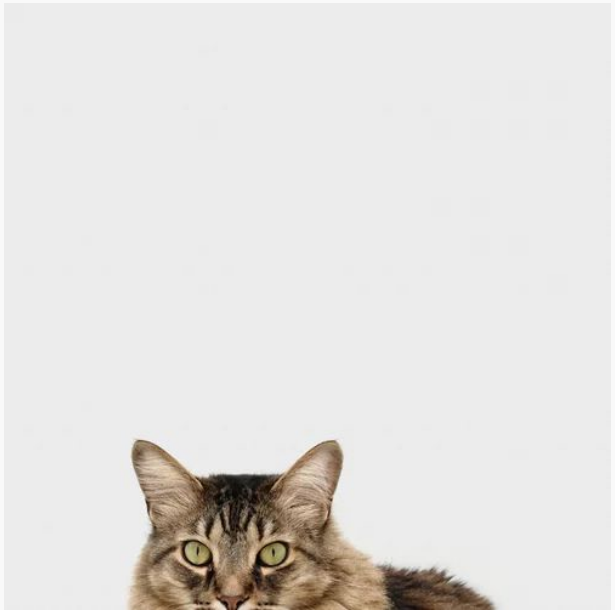
Cat vs Non Cat

Example Problem

Binary Classification

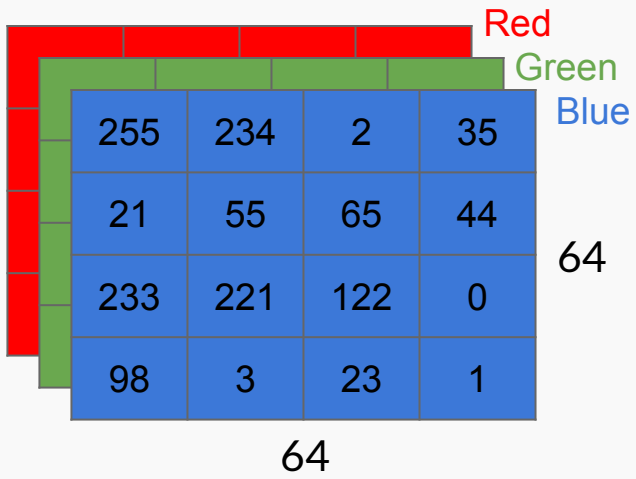
64

64



Number of pixels

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

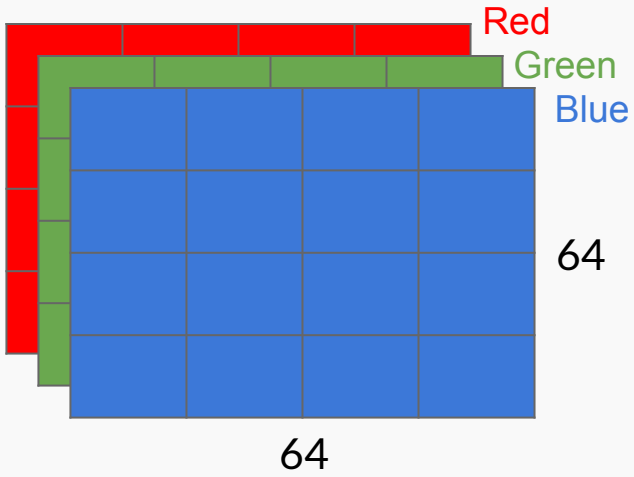


Example Problem

Binary Classification

Number of pixels

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



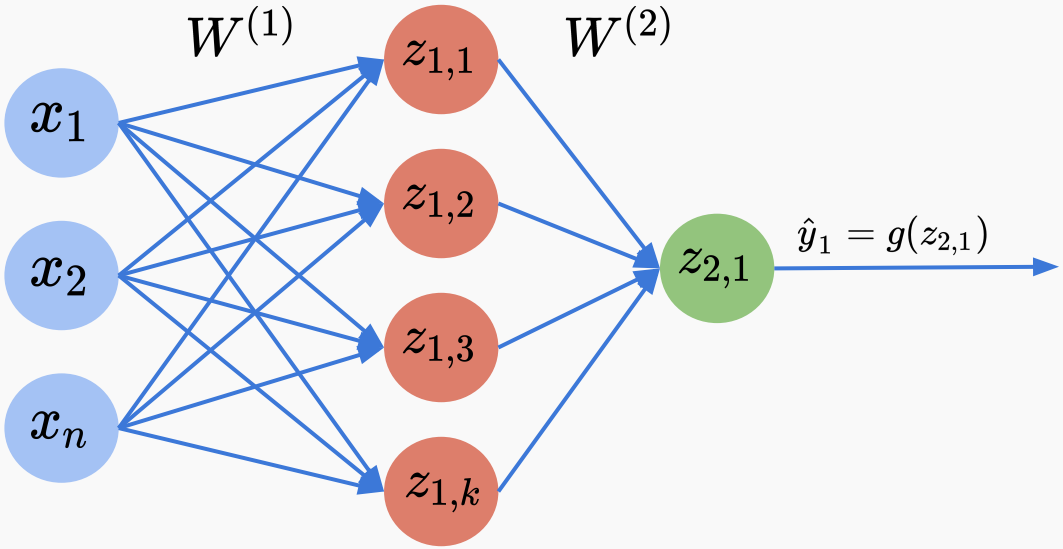
$$y = \begin{matrix} Y^{(1)} \\ Y^{(2)} \\ Y^{(3)} \\ \vdots \\ Y^{(m)} \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

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

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

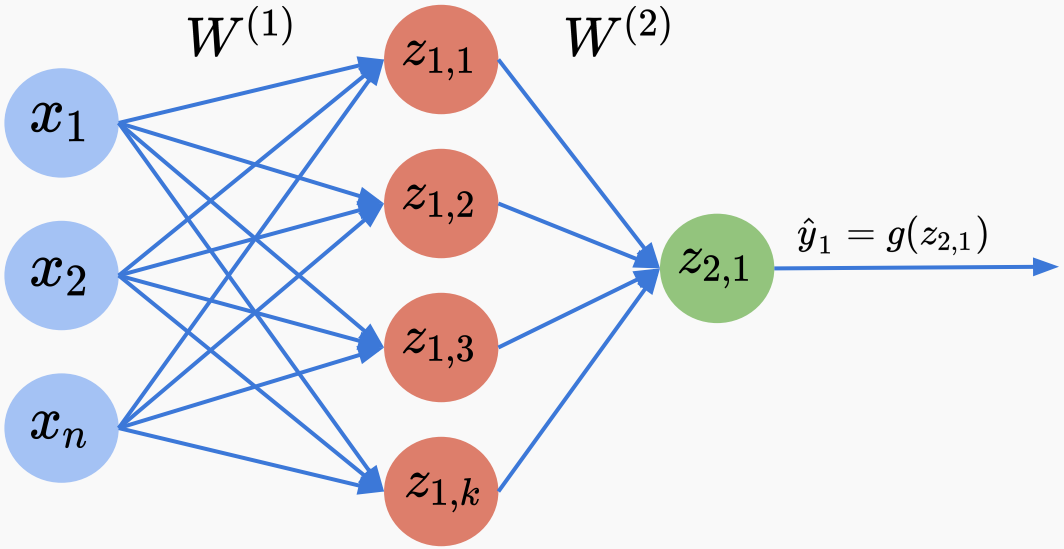
Quantifying Loss

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



Empirical Loss

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



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

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

Also known as:

- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss

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

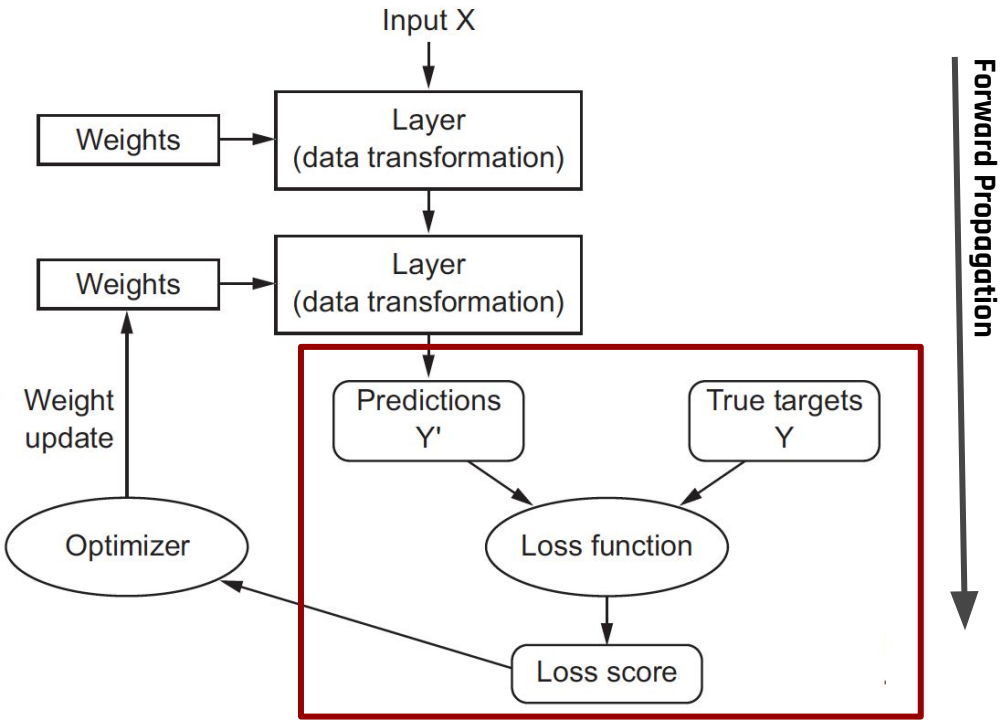
$$\mathcal{L}\left(f(X^{(i)};W), y^{(i)}\right) = \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

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

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

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

Understanding how DL works



Adapted from @FrançoisChollet

Training Neural Networks

Part #01

Loss Optimization

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

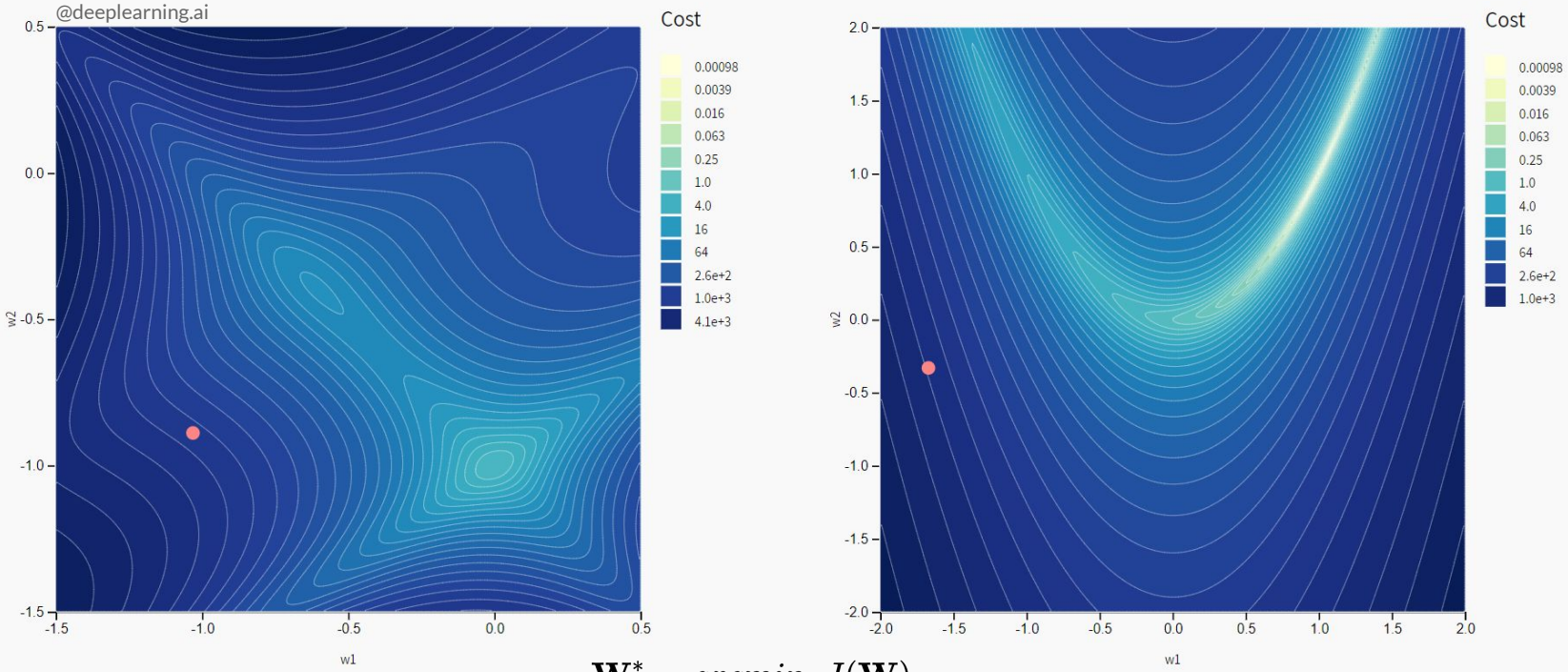
$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(f(X^{(i)}; \mathbf{W}), y^{(i)}\right)$$

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

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

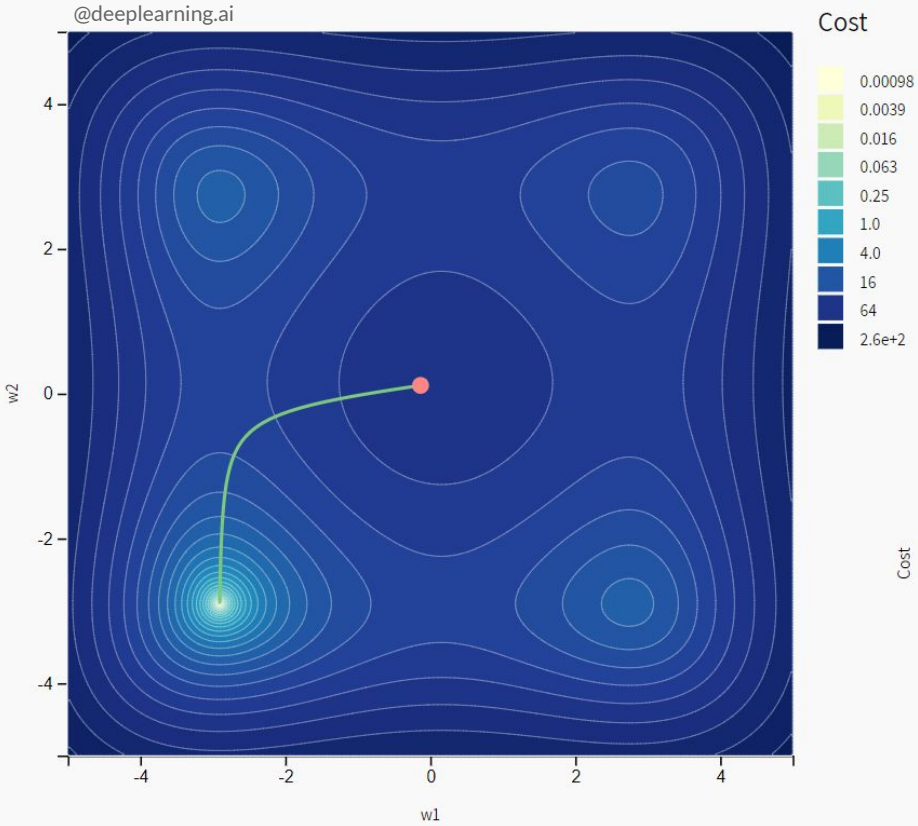
Loss Optimization

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



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

Loss Optimization - Gradient Descent

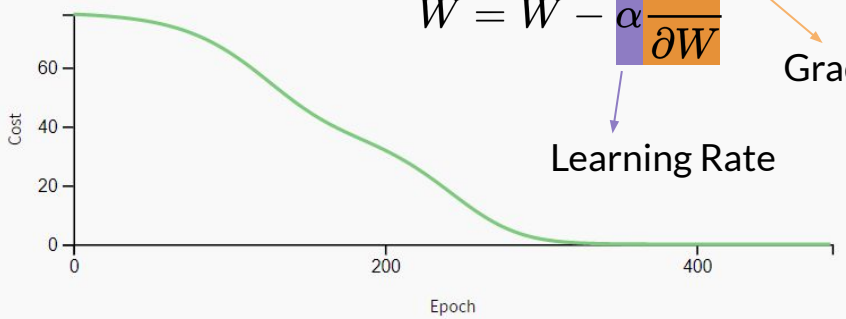


1. Initialize weights randomly
2. Compute gradient
3. Take small step in opposite direction of gradient
4. Repeat until convergence

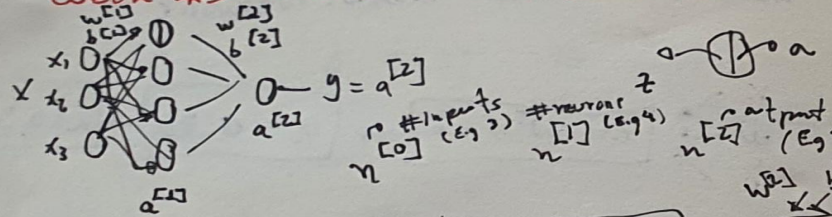
$$W = W - \alpha \frac{\partial J}{\partial W}$$

Gradient

Learning Rate



WEEK #3 Shallow Neural Networks



$$x, w^{[1]}, b^{[1]} \Rightarrow z^{[1]} = w^{[1]} x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \rightarrow z^{[2]} = w^{[2]} a^{[1]} + b^{[2]} \rightarrow a^{[2]} = \sigma(z^{[2]}) \rightarrow \mathcal{L}(a^{[2]}, y)$$

$w^{[2]}, b^{[2]}$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

(n_x = # features, m = # train instances)

$w^{[1]} = (n^{[1]}, n^{[0]})$ $\quad$ $w^{[2]} = (n^{[2]}, n^{[1]})$

$b^{[1]} = (n^{[1]}, 1)$ $\quad$ $b^{[2]} = (n^{[2]}, 1)$

$z^{[1]} = (n^{[1]}, 1)$ $\quad$ $z^{[2]} = (n^{[2]}, 1)$

$x = (n^{[0]}, 1)$ $\quad$ $a^{[2]} = (n^{[2]}, 1)$

$a^{[1]} = (n^{[1]}, 1)$

$$\mathcal{L}(a^{[2]}, y) = -y \log(a^{[2]}) - (1-y) \log(1-a^{[2]})$$

$$da^{[2]} = \frac{d\mathcal{L}}{da} = -\frac{y}{a^{[2]}} + \frac{(1-y)}{1-a^{[2]}} \quad (n^{[2]}, 1)$$

$$dz^{[2]} = \frac{d\mathcal{L}}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} = \left[-\frac{y}{a^{[2]}} + \frac{(1-y)}{1-a^{[2]}} \right] a^{[2]} \cdot (1 - a^{[2]})$$

$$\boxed{dz^{[2]} = a^{[2]} - y} \quad (n^{[2]}, 1)$$

$$dw^{[2]} = \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dw^{[2]}} = \underbrace{dz^{[2]}}_{(n^{[2]}, 1)} \cdot \underbrace{a^{[1]T}}_{(1, n^{[1]})} \quad (n^{[2]}, n^{[1]})$$

$dw^{[2]}$

$$db^{[2]} = \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{db^{[2]}} = \underbrace{dz^{[2]}}_{(n^{[2]}, 1)} \cdot \underbrace{1}_{(1, 1)} \quad (n^{[2]}, 1)$$

$db^{[2]}$

$$da^{[1]} = \frac{d\mathcal{L}}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} = \underbrace{dz^{[2]}}_{(n^{[2]}, 1)} \cdot \underbrace{w^{[2]}}_{(n^{[2]}, n^{[1]})} \quad (n^{[1]}, 1)$$

$$dz^{[1]} = \frac{d\mathcal{L}}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}} = da^{[1]} \cdot g'(z^{[1]}) = \underbrace{w^{[2]T} \cdot dz^{[2]}}_{(n^{[1]}, n^{[2]}) \; (n^{[2]}, 1)} * \underbrace{g'(z^{[1]})}_{(n^{[1]}, 1)} \quad (n^{[1]}, 1)$$

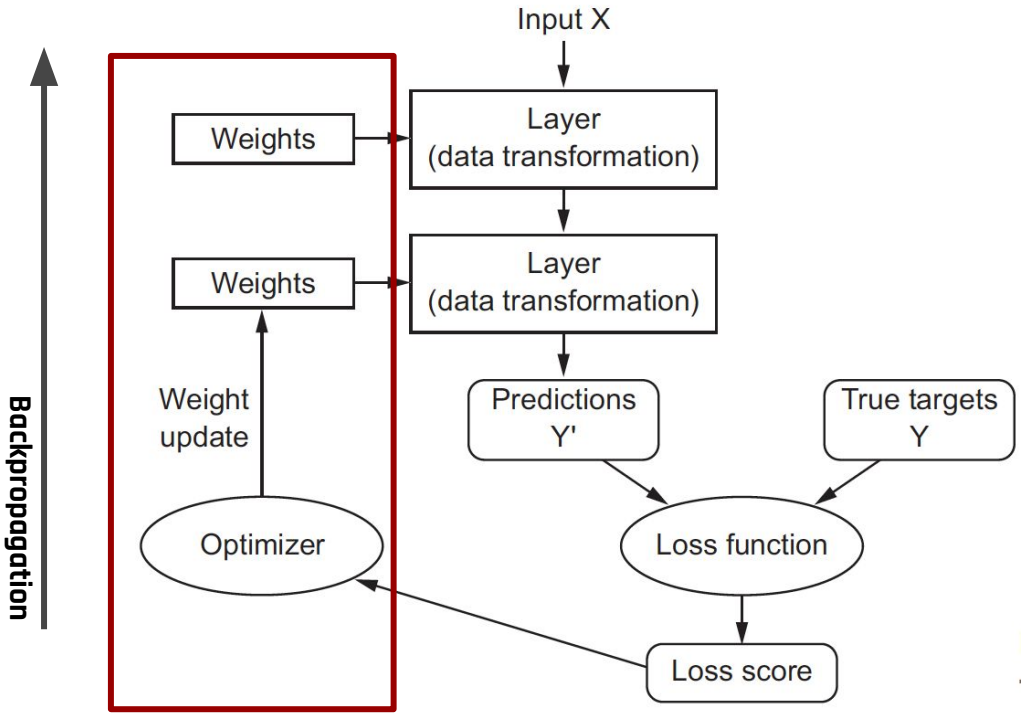
$$dw^{[1]} = \frac{d\mathcal{L}}{dz^{[1]}} \frac{dz^{[1]}}{dw^{[1]}} = \underbrace{dz^{[1]}}_{(n^{[1]}, 1)} \cdot \underbrace{x^T}_{(m, n^{[0]})} = (n^{[1]}, i) * (i, n^{[0]}) = (n^{[1]}, n^{[0]})$$

(compute)

$$= (w^{[2]T} \cdot dz^{[2]}) * g'(z^{[1]})$$

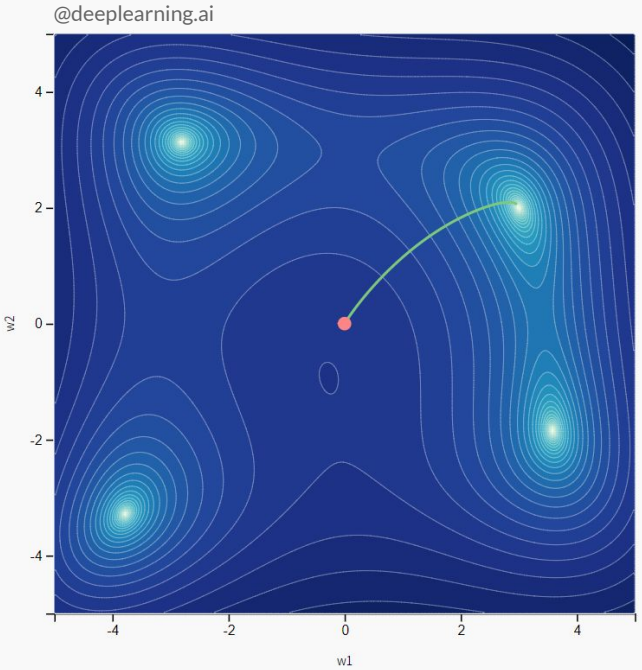
Understanding how DL works

Finding the right values of weights which minimize the error

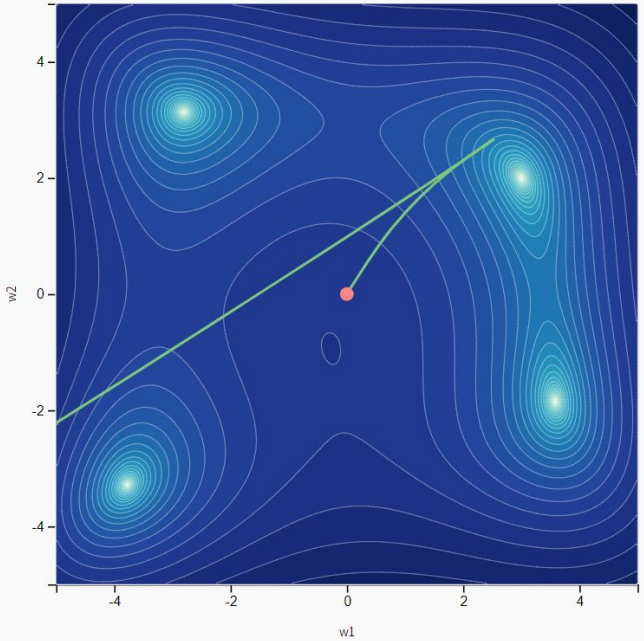


Adapted from @FrançoisChollet

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

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

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

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)
```

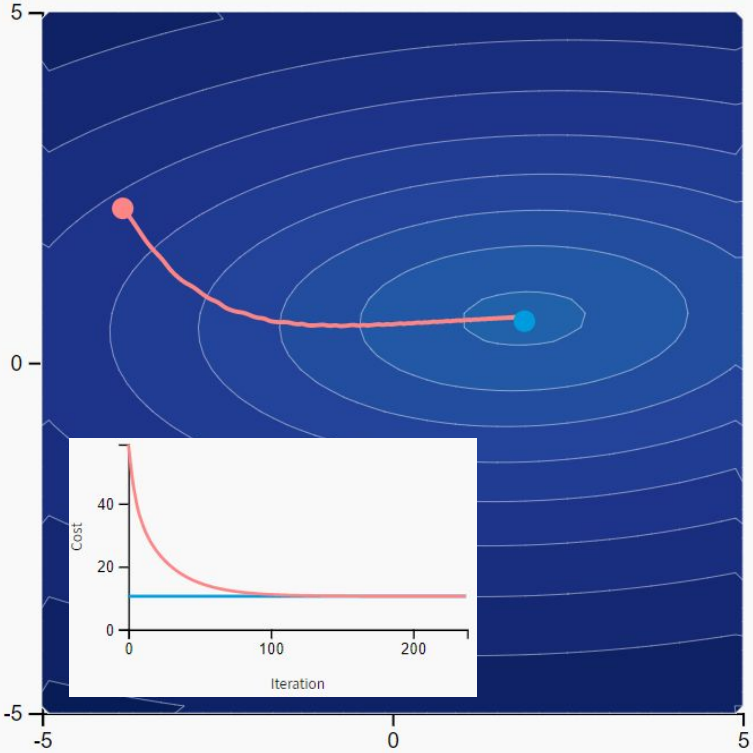
Putting it all together

```
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)
```

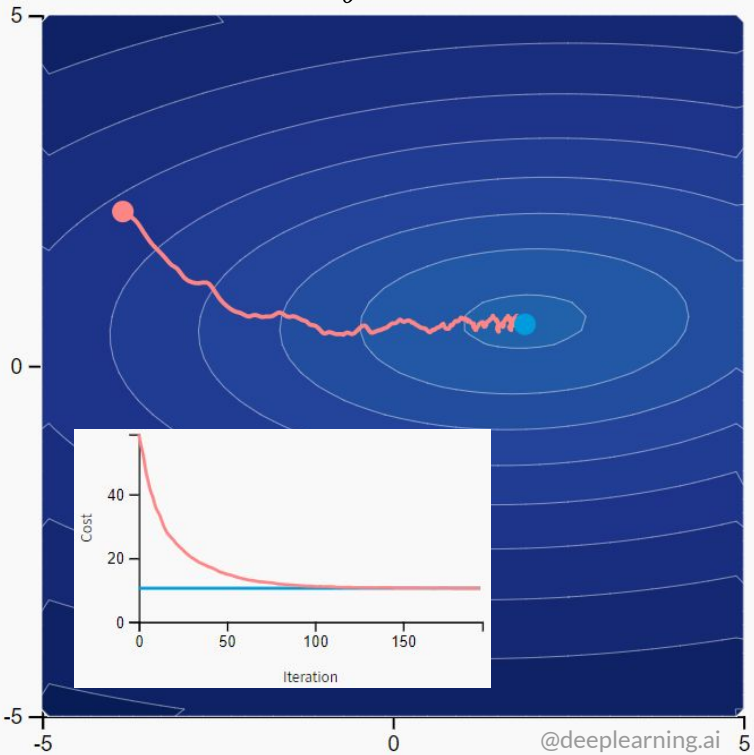
$$W = W - \alpha \frac{\partial J}{\partial W}$$

Mini-Batches Challenges

$m_b = m$



$m_b = 24$



> Lesson 09 Task 01 - TensorFlow 2.0 + Keras Crash Course.ipynb
> Lesson 09 Task 02 - Introduction to TF.ipynb

Training Neural Networks

Part #02

TRAIN A DEEP NEURAL NETWORK

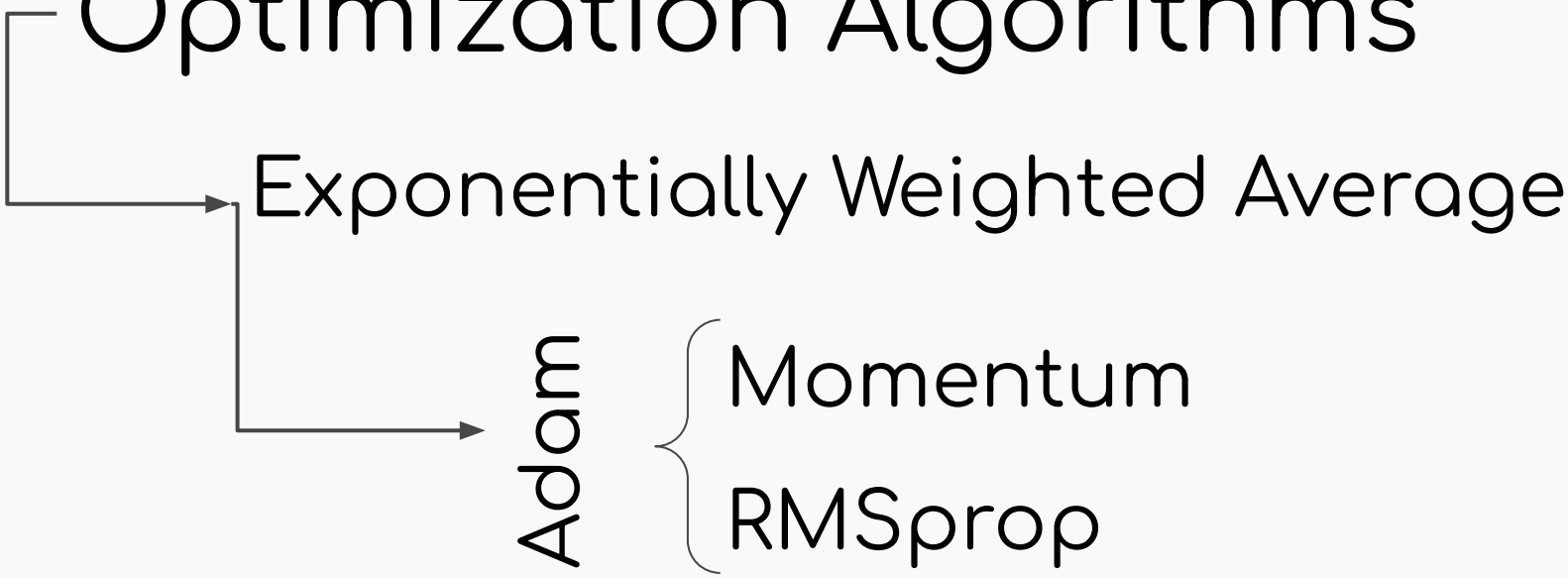
=

MINI-BATCH

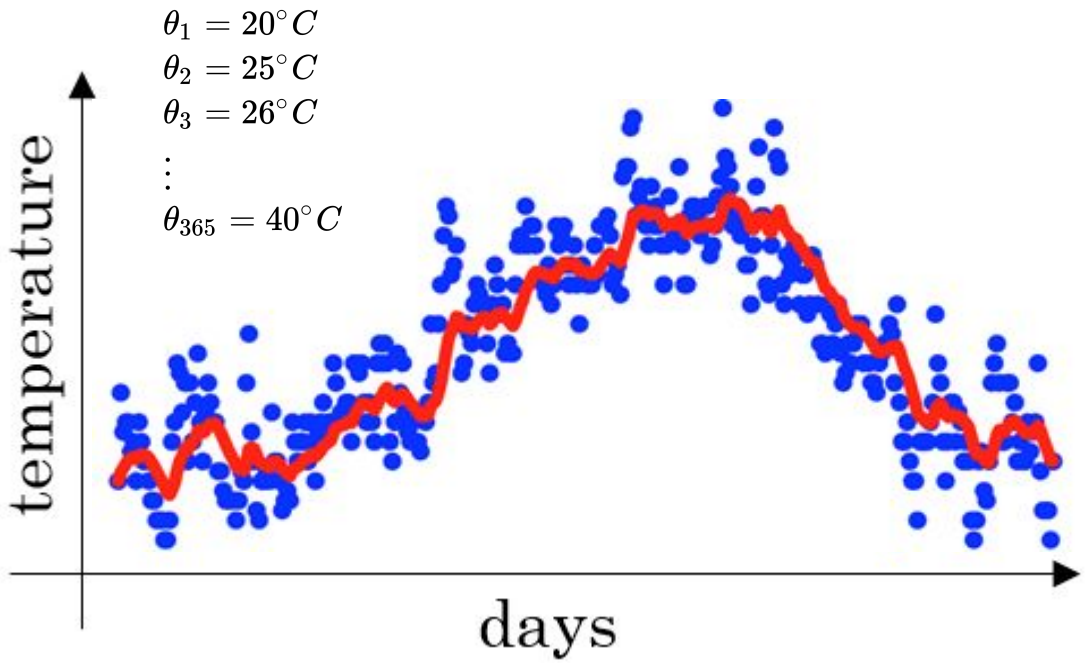
+

OPTIMIZATION ALGORITHMS

Optimization Algorithms



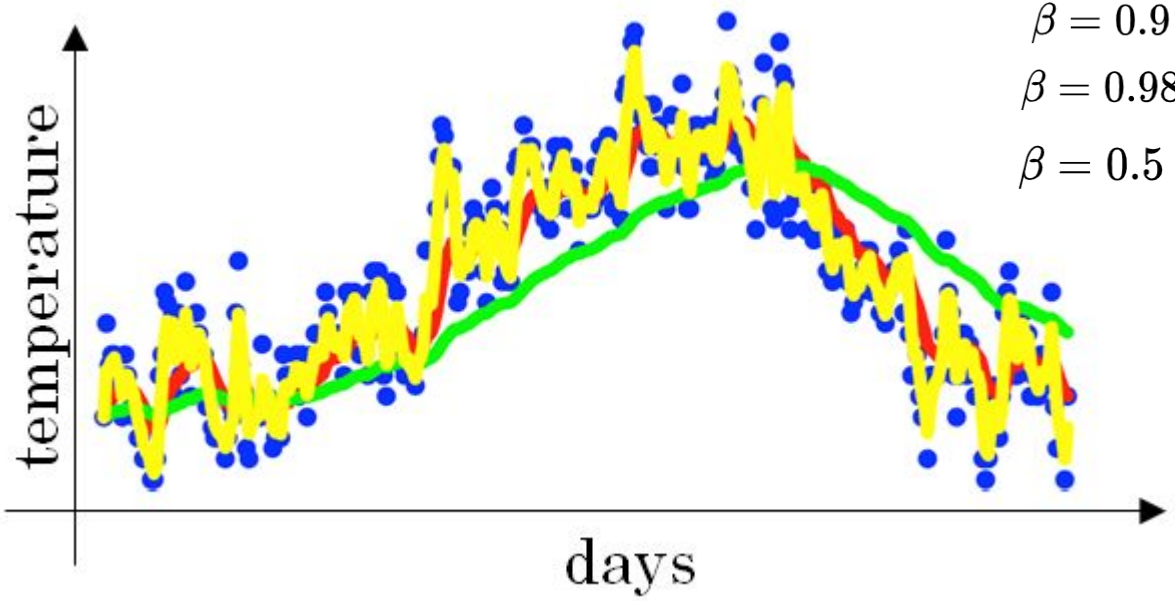
Exponentially Weighted Average



$$V_0 = 0$$
$$V_1 = 0.9V_0 + 0.1\theta_1$$
$$V_2 = 0.9V_1 + 0.1\theta_2$$
$$V_3 = 0.9V_2 + 0.1\theta_3$$
$$\vdots$$
$$V_{365} = 0.9V_{364} + 0.1\theta_{365}$$

$$V_t = 0.9V_{t-1} + 0.1\theta_t$$

Exponentially Weighted Average



$$\beta = 0.9 \rightarrow V_t \approx 10 \; days \; of \; temperature$$

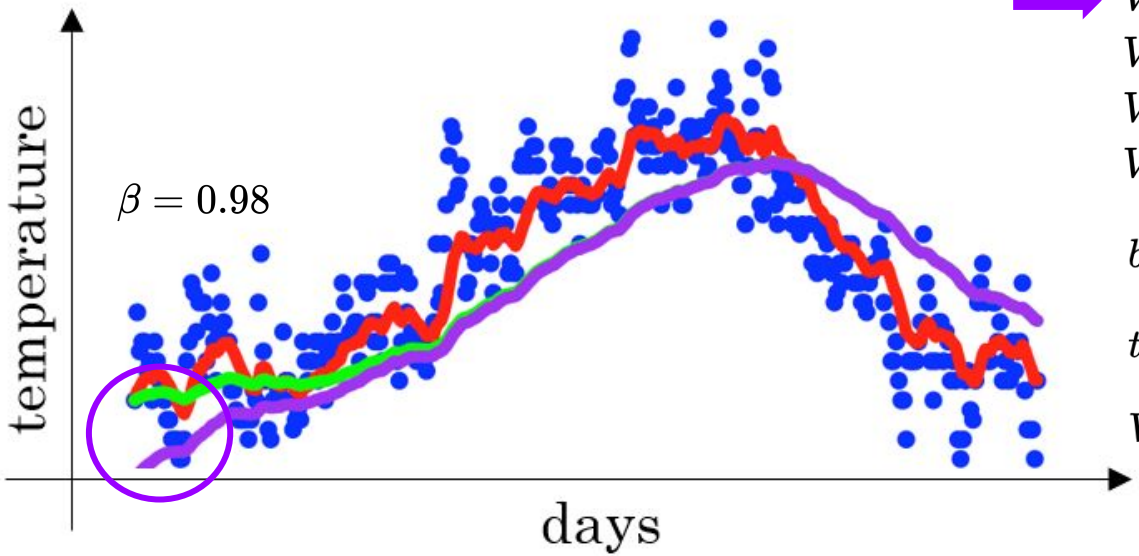
$$\beta = 0.98 \rightarrow V_t \approx 50 \; days \; of \; temperature$$

$$\beta = 0.5 \rightarrow V_t \approx 2 \; days \; of \; temperature$$

$$V_t = \beta V_{t-1} + (1-\beta)\theta_1$$

$$V_t \approx \frac{1}{1-\beta} \; days \; of \; temperature$$

Exponentially Weighted Average Bias Correction



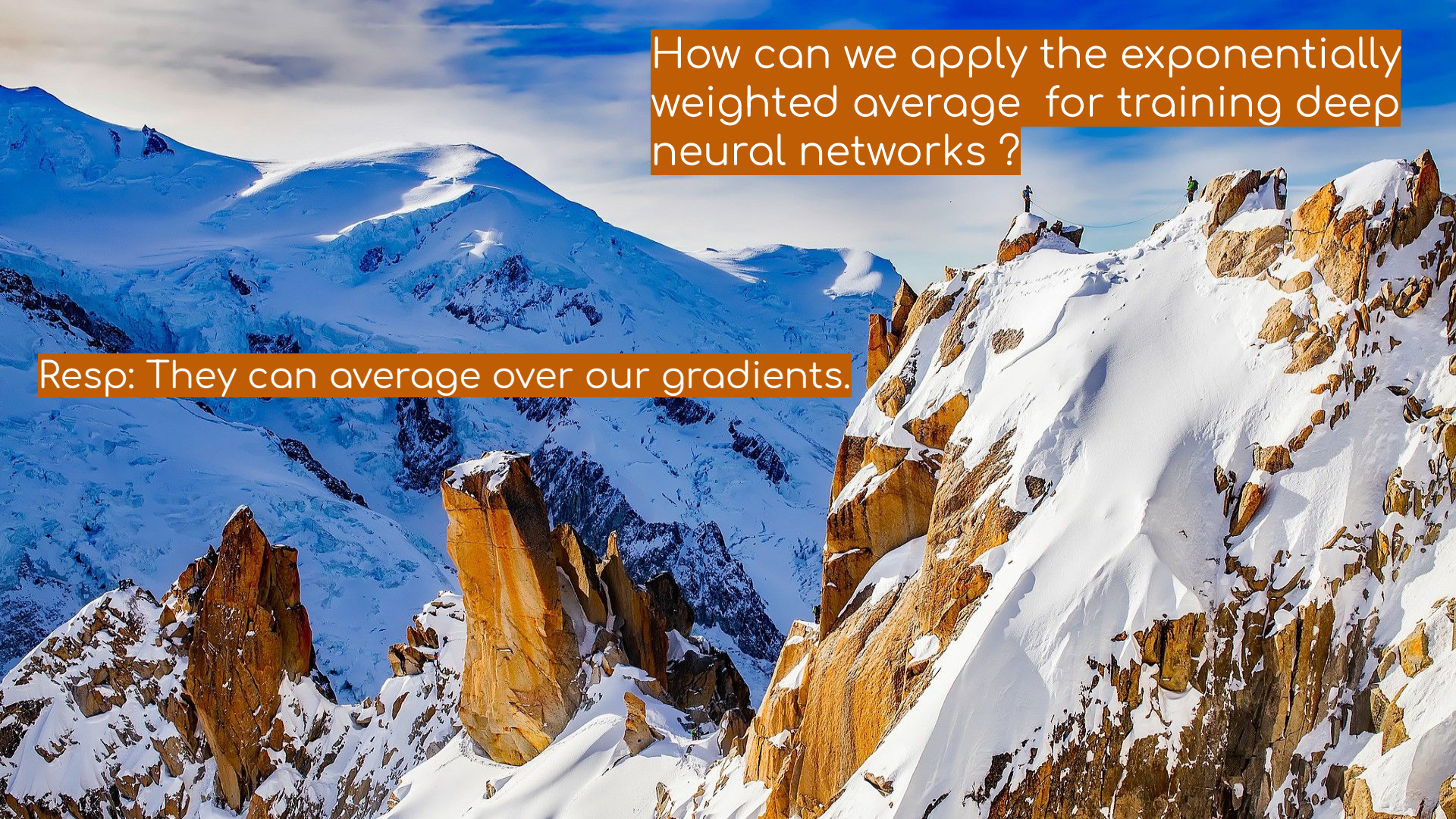
$$V_0 = 0$$
$$V_1 = 0.98V_0 + 0.02\theta_1$$
$$V_2 = 0.98V_1 + 0.02\theta_2$$
$$V_2 = 0.98 * 0.02\theta_1 + 0.02\theta_2$$
$$V_2 = 0.0196\theta_1 + 0.02\theta_2$$

$$bias\_correction = \frac{V_t}{1-\beta^t}$$

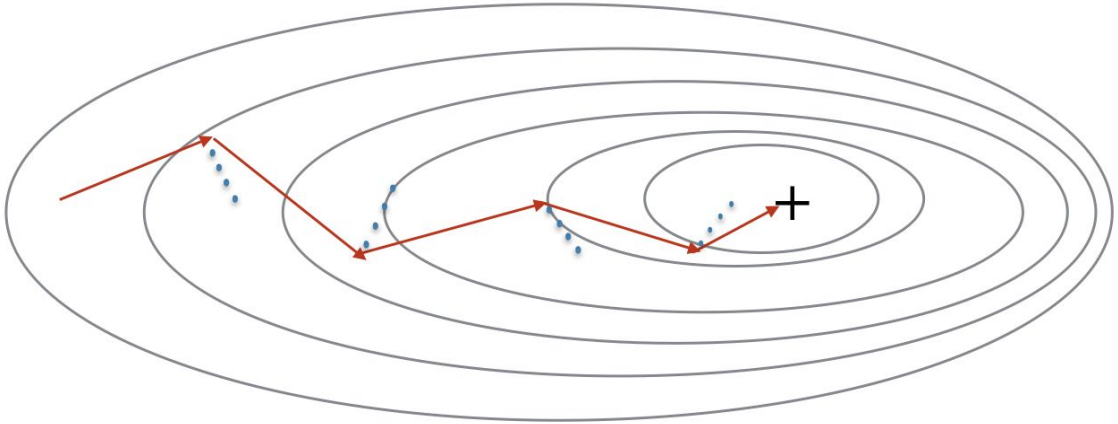
$$t = 2 \rightarrow 1 - \beta^2 = (1 - 0.98^2) = 0.0396$$

$$V_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$


PETROBRAS PN N2 · D · BMFBOVESPA O29.51 H29.68 L29.03 C29.14 −0.78 (−2.61%)
EMA 50 close 29.37
SELL 29.08
0.09
BUY 29.17
Closed Delayed
32.00
31.50
31.00
30.50
30.00
29.50
29.14
28.50
28.00
27.50
27.00
26.50
26.00
25.50
25.00
24.50
24.00
23.50
23.00
22.50
E
D
E
D
D
E
Jul
Aug
Sep
Oct
Nov
Dec
2020
Feb
Mar


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

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

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

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β $\qquad\qquad$ $\beta = 0.9$

Gradient Descent with RMSprop

On iteration t:

Compute dW, db on the current mini-batch

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

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

$$W = W - \alpha \frac{dW}{\sqrt{s_{dW}} + \varepsilon} \qquad b = b - \alpha \frac{db}{\sqrt{s_{db}} + \varepsilon}$$

$\varepsilon = 10\text{^}(-8)$

Hyperparameters: α, β $\qquad\qquad$ $\beta = 0.9$

Gradient Descent with Adam

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

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

$$W = W - \alpha \frac{v_{dW}^{correct}}{\sqrt{s_{dW}^{correct}} + \varepsilon} \qquad b = b - \alpha \frac{v_{db}^{correct}}{\sqrt{s_{db}^{correct}} + \varepsilon}$$

$\varepsilon = 10\text{^}(-8)$

Hyperparameters: α, β_1, β_2 $\beta_1 = 0.9, \beta_1 = 0.999$

Learning Rate Decay

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

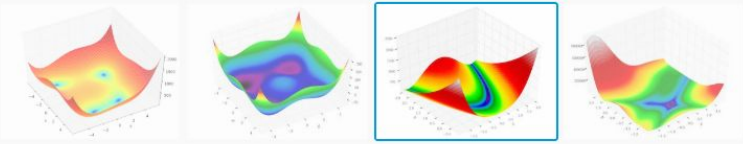
$$\alpha = \frac{1}{1 + learning\_decay * epoch\_num} * \alpha_0$$

$$\alpha = 0.95\ e^{epoch\_num}\ \alpha_0$$

$$\alpha = \frac{k}{\sqrt{epoch\_num}} * \alpha_0$$

1. Choose a cost landscape

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



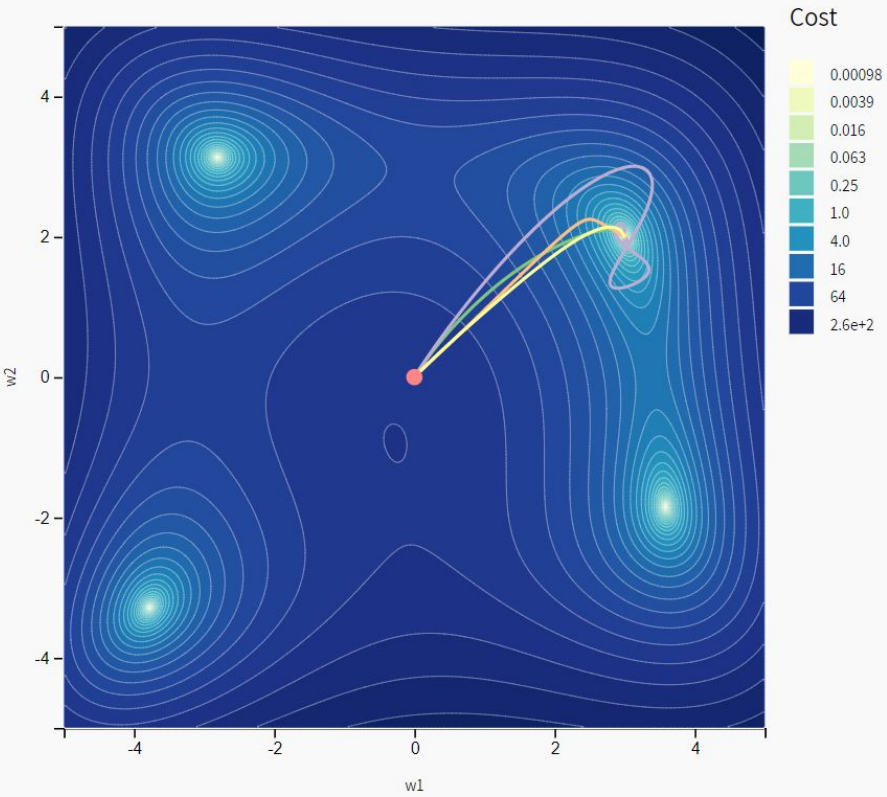
2. Choose initial parameters

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

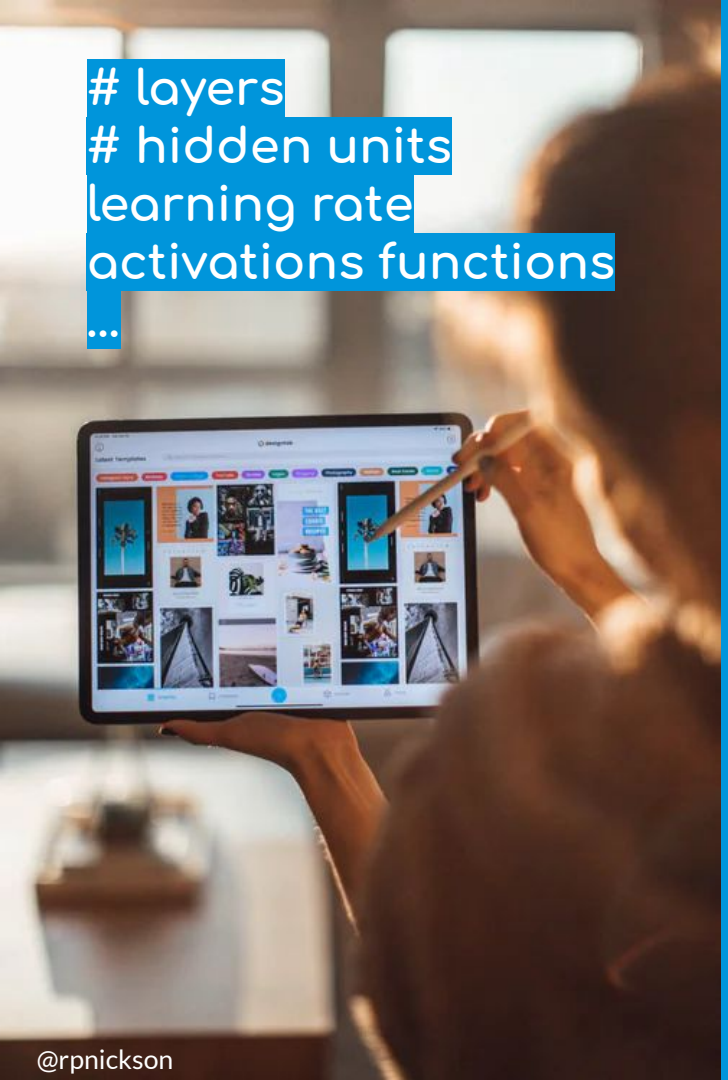


https://www.deeplearning.ai/ai-notes/optimization/

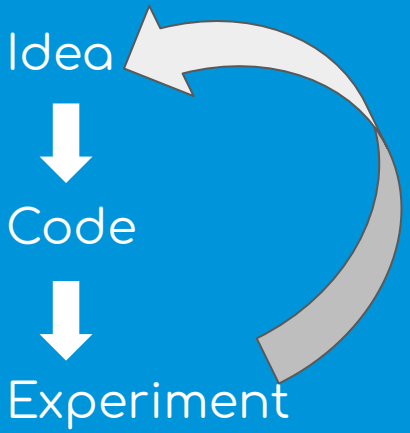
Lesson 09 Task 03 - Optimization Methods.ipynb

Neural Networks in Practice #01

Splitting Data & Regularization



Applied NN is a highly iterative process



Train - Dev - Test Sets

Making good choices in how you set up your training, development, and test sets can make a huge difference in helping you quickly find a good high performance neural network.

Holdout
Cross-Validation
Validation
Development

Previous ML era

- 70/30
- 60/20/20

Big Data era

- 98/1/1
- 99.5/0.25/0.25
- 99.5/0.4/0.1

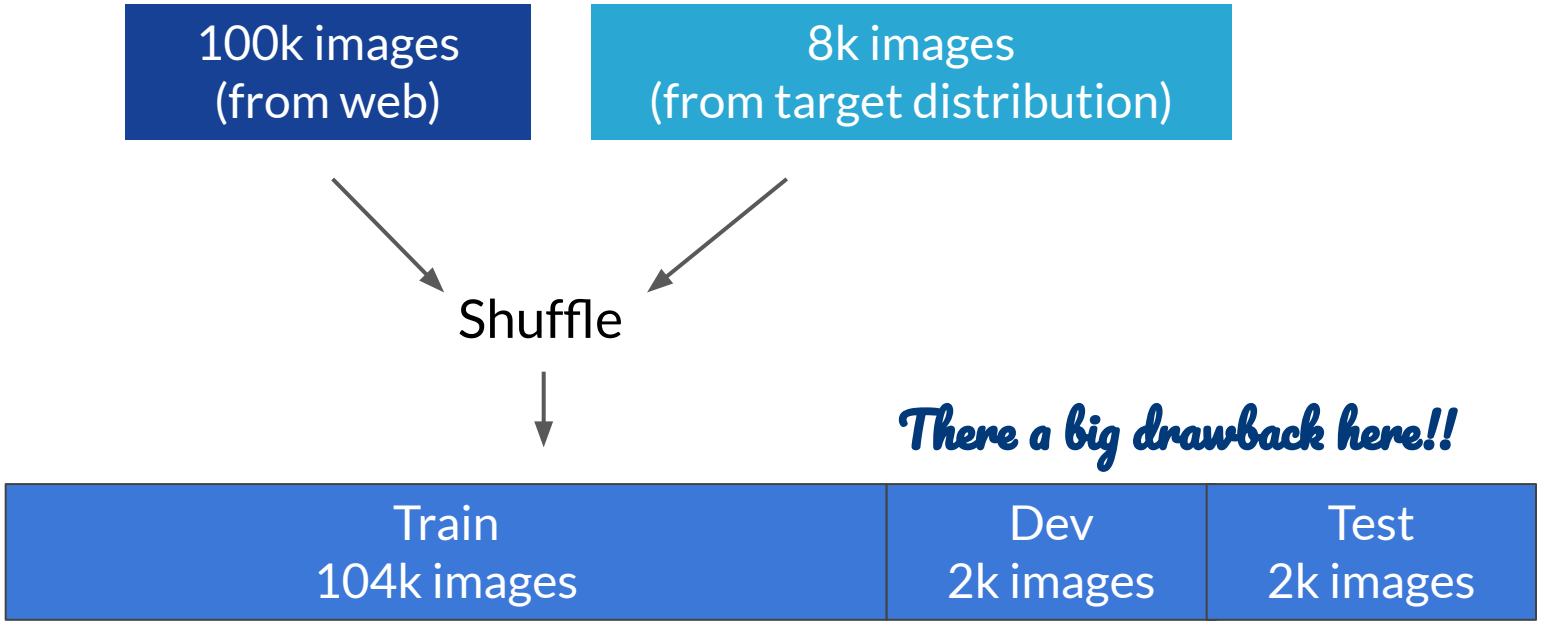
Mismatched train/test distribution

Scenario: say you are building a cat-image classifier application that determines if an image is of a cat or not. The application is intended for users in rural areas who can take pictures of animals by their mobile devices for the application to classify the animals for them.

Scraped from Web Pages
100k images

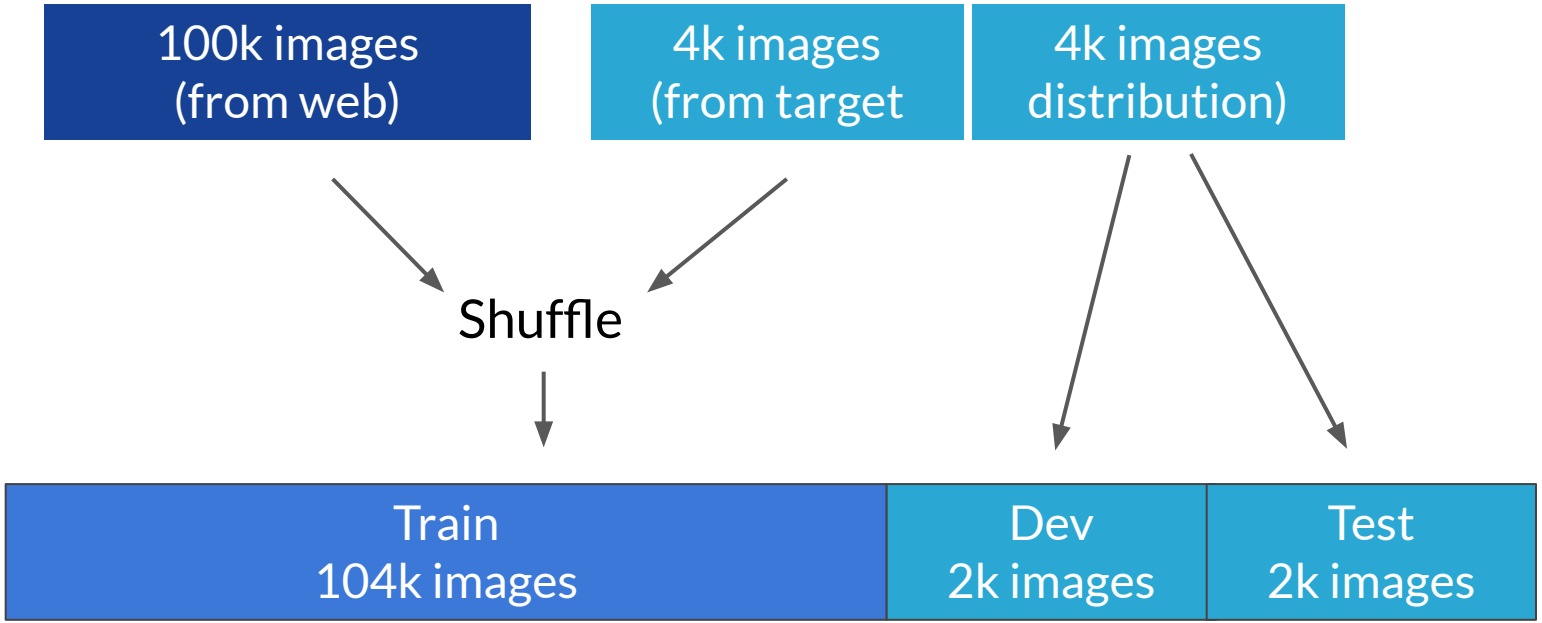
Collected from Mobile Devices
<<target distribution>>
8k images

A possible option: shuffling the data



only 148 images come from the target distribution

A better option

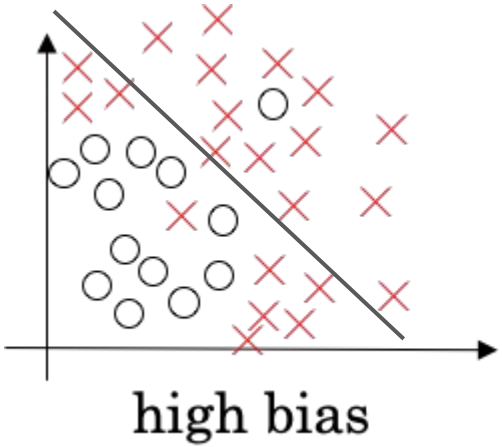


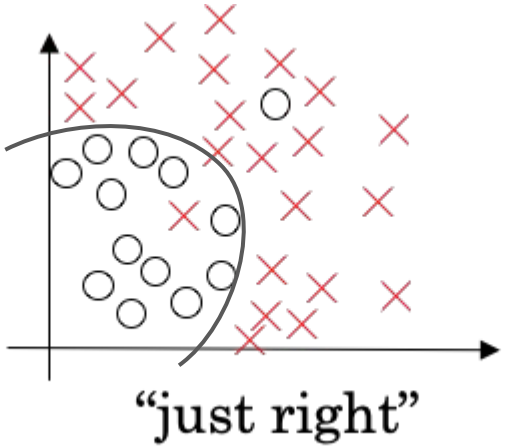
Rule of the thumb

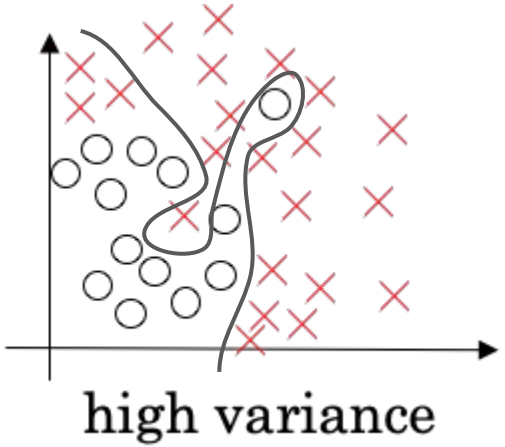
>> make sure that the dev and test sets come from the same distribution

Not having a test set might be okay. (Only dev set)

Bias vs Variance







Underfitting

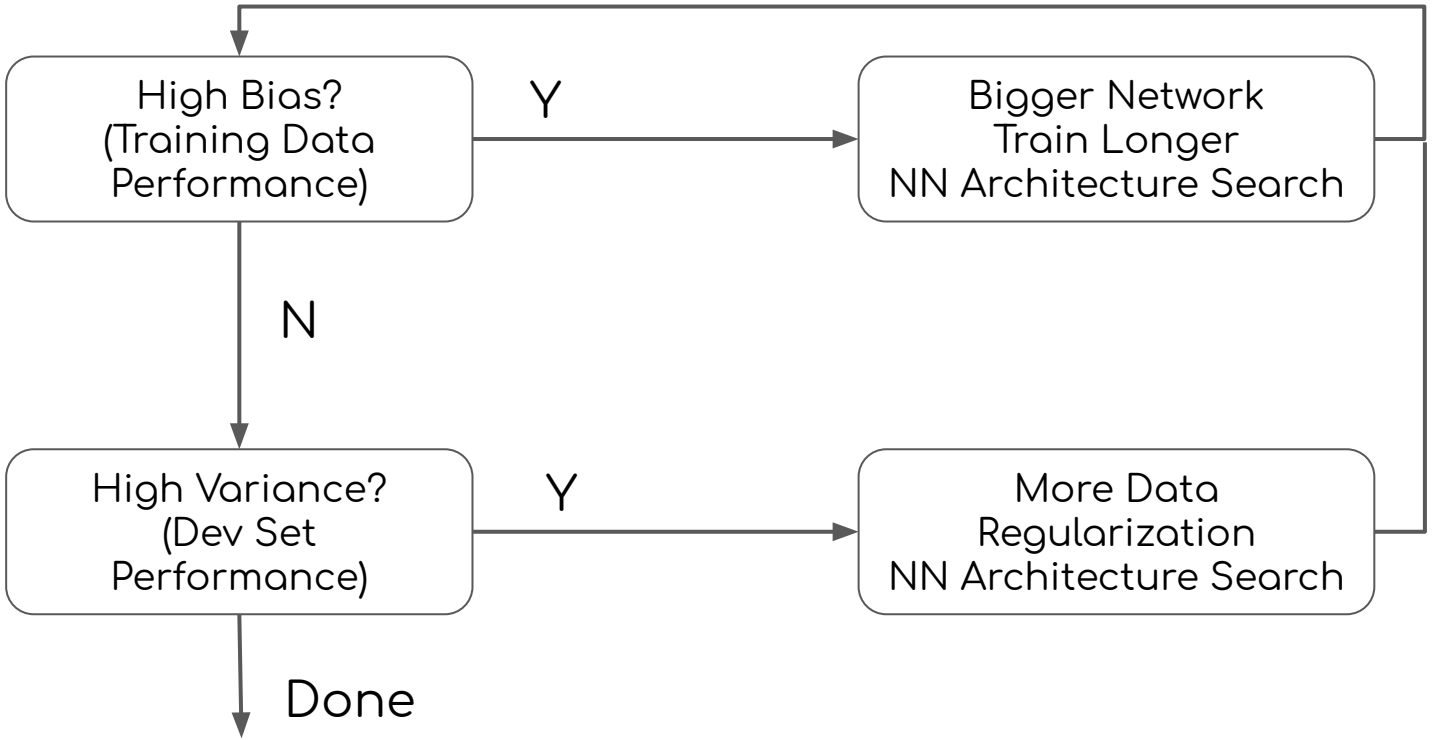
Overfitting

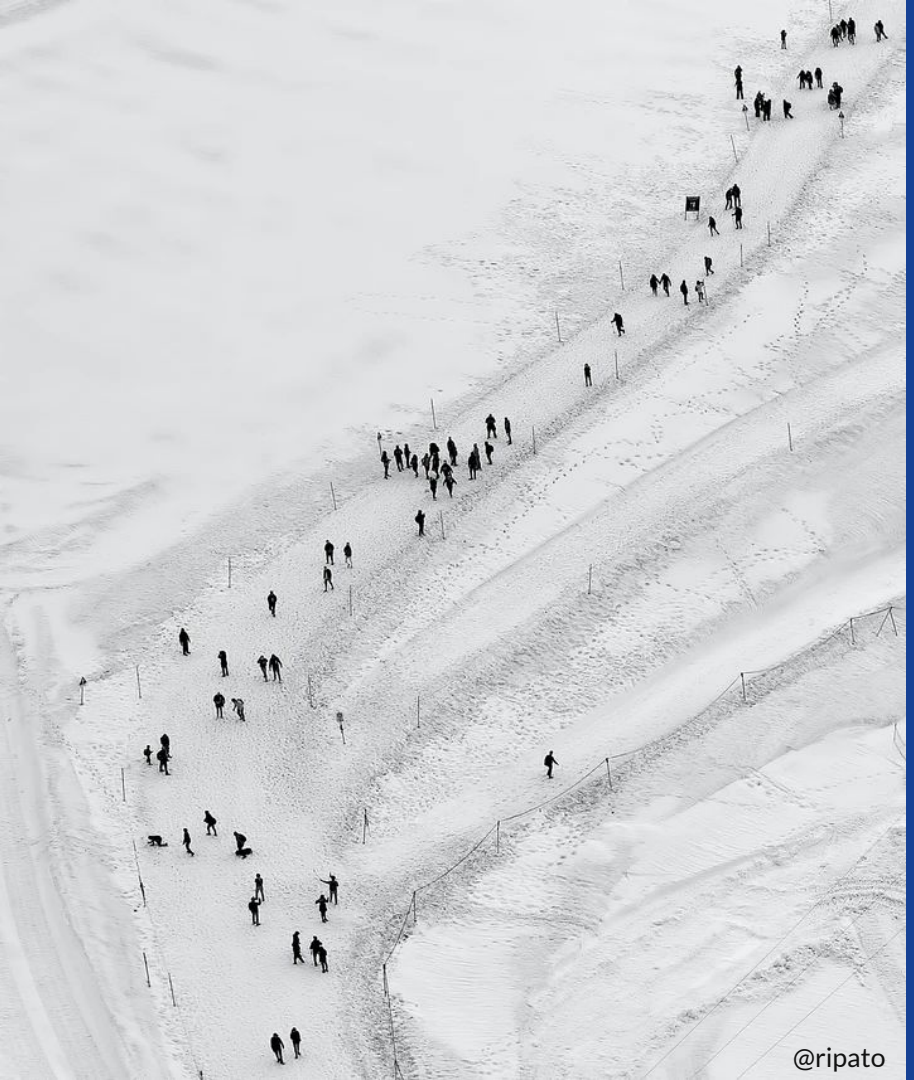
Bias vs Variance

Cat Classification

	Scenario #01	Scenario #02	Scenario #03	Scenario #04
Train Set Error	1%	15%	15%	0.5%
Dev Set Error	16%	16%	30%	1%
	Low Bias High Variance	High Bias Low Variance	High Bias High Variance	Low Bias Low Variance

Basic Recipe for Machine Learning

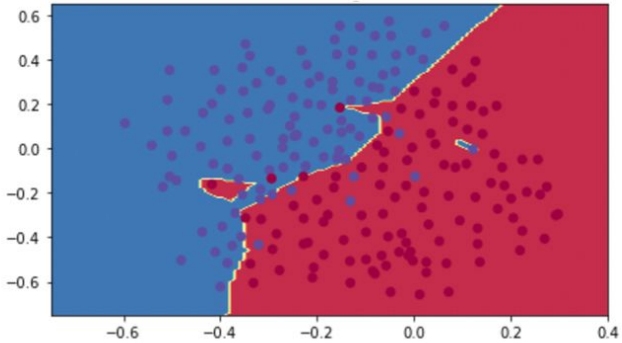




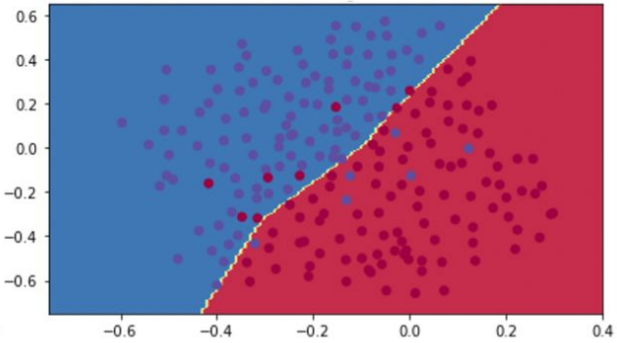
Regularizing your Neural Network

What if we penalize complexity?

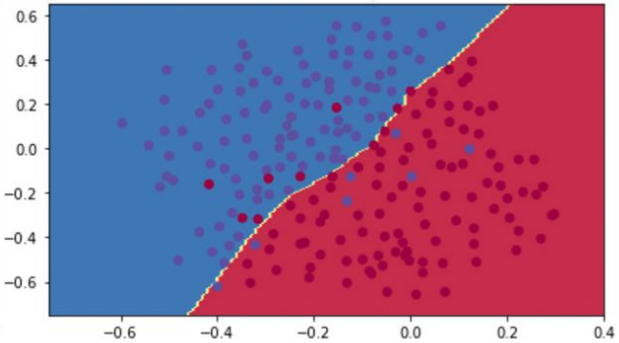
Model without regularization



Model with regularization



Model with dropout



It is very important that you regularize your model properly because it could dramatically improve your results

L2 Regularization (Frobenius Norm)

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

L2 Regularization (Frobenius Norm)

Impact on Gradient Descent

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

$$\frac{\partial J}{\partial W^{[l]}} = dW^{[l]} = \{from\ backprop.\} + \frac{\lambda}{m}W^{[l]}$$

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$$

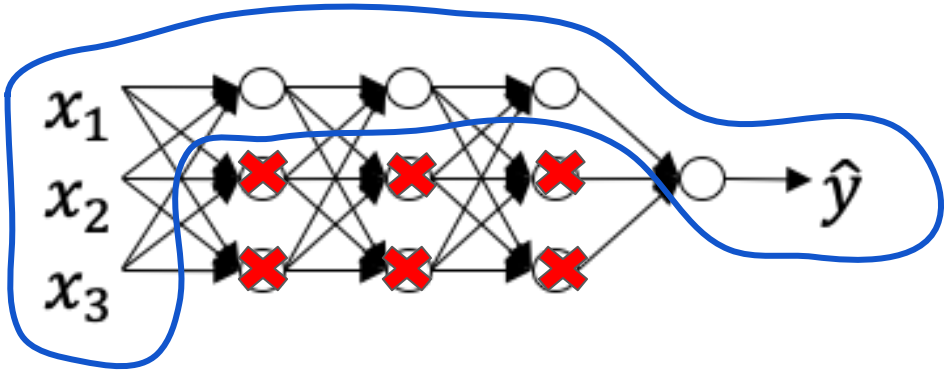
$$W^{[l]} = W^{[l]} - \alpha\left[\{from\ backprop.\} + \frac{\lambda}{m}W^{[l]}\right]$$

$$W^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right)W^{[l]} - \alpha\left[\{from\ backprop.\}\right.$$

"Weight Decay"

How does regularization prevent overfitting?

Intuition #01

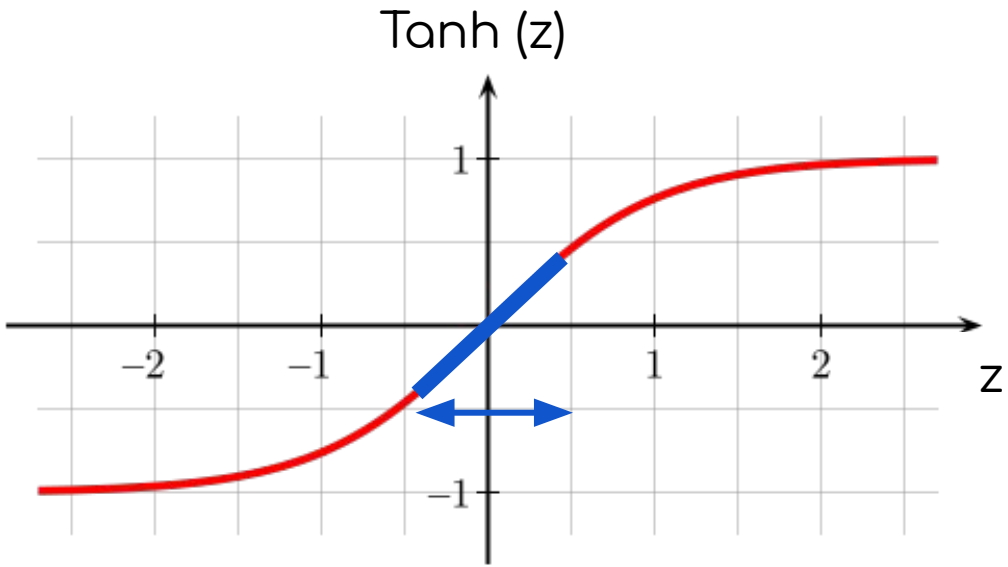


$$W^{[l]} = (1 - \frac{\alpha\lambda}{m})W^{[l]} - \alpha\,[\{from\ backprop.\}$$

$$\lambda \uparrow \quad \Rightarrow \quad W^{[l]} \approx 0$$

How does regularization prevent overfitting?

Intuition #02



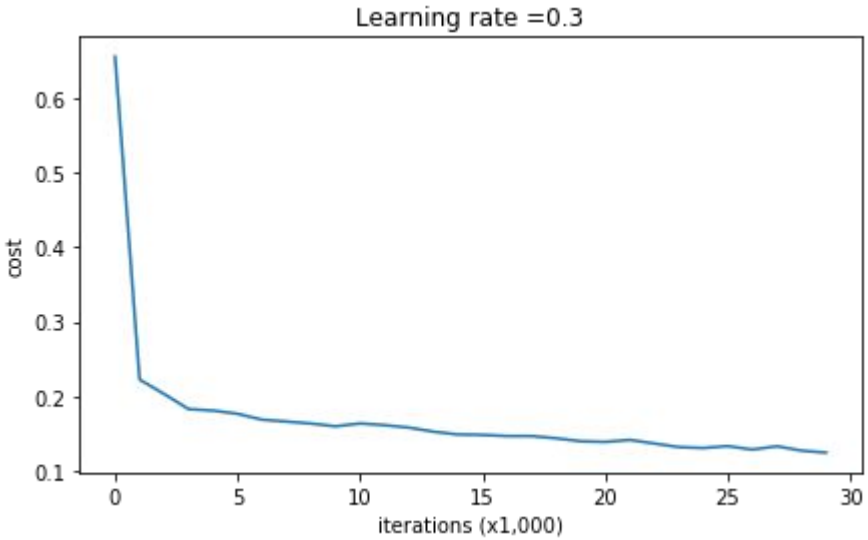
$$\lambda \uparrow \quad \Rightarrow \quad W^{[l]} \approx 0$$

$$Z = \alpha^{[l-1]} W^{[l]} + b^{[l]}$$

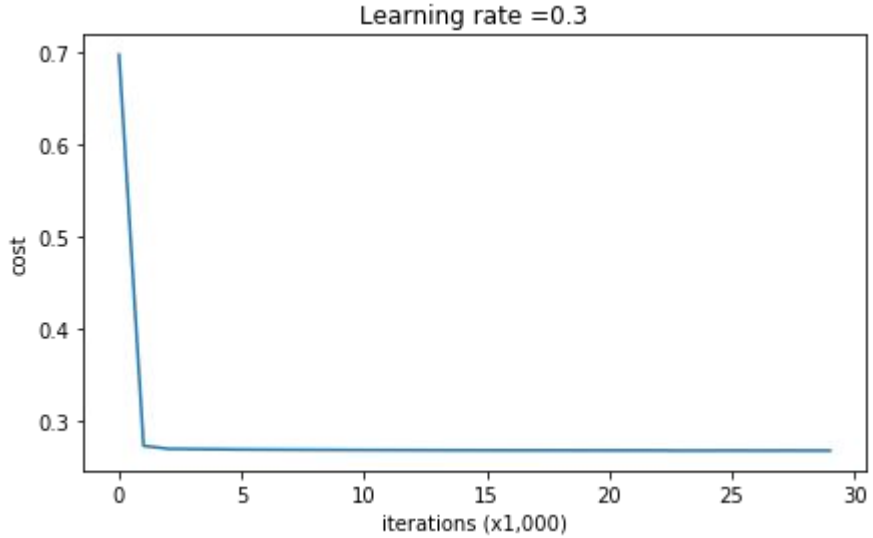
How does regularization prevent overfitting?

Intuition #03

Without Regularization



With Regularization

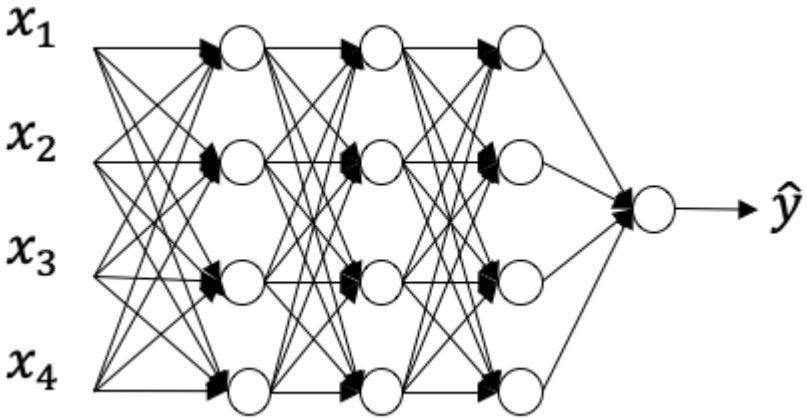


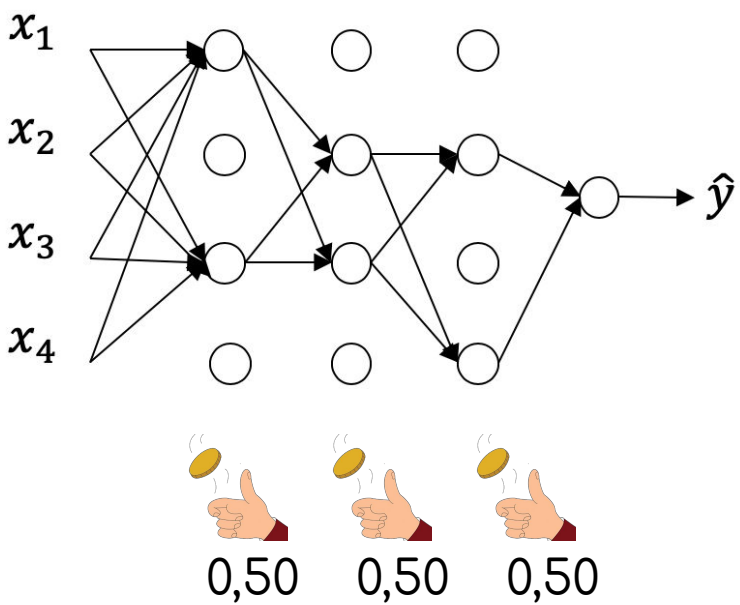
Putting it all together

```
model = tf.keras.Sequential([
                            tf.keras.layers.Dense(20, activation=tf.nn.relu),
                            tf.keras.layers.Dense(3,activation=tf.nn.relu,
                                                  kernel_regularizer=tf.keras.regularizers.l2(l=0.01)),
                            tf.keras.layers.Dense(1, activation = tf.nn.sigmoid)
                            ])
```

Dropout Regularization

You implement Dropout regularization only while training the network. You do not apply it while running your testing data through it as you do not want any randomness in your predictions





Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

Implementing Dropout ("Inverted Dropout")

Illustrate with layer "l" = 3

$$\begin{cases} keep\_prob & = 0.8 \\ 1 - keep\_prob & = 0.2 \end{cases}$$

$$d3 = np.randon.rand(a3.shape[0], a3.shape[1]) < keep\_prob$$

$$a3 = np.multiply(a3, d3)$$

$$a3 /= keep\_prob$$

$$Z^{[4]} = \alpha^{[3]} W^{[4]} + b^{[4]}$$

It is necessary not to impact the value of Z.

Applying dropout for a input

```
import tensorflow as tf
import numpy as np

tf.random.set_seed(0)
layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
data = np.arange(10).reshape(5, 2).astype(np.float32)
print(data)

[[0. 1.]
 [2. 3.]
 [4. 5.]
 [6. 7.]
 [8. 9.]]
```

keep_prob=0.8

```
outputs = layer(data, training=True)
print(outputs)

tf.Tensor(
[[ 0.    1.25]
 [ 2.5   3.75]
 [ 5.    6.25]
 [ 7.5   8.75]
 [10.    0.  ]], shape=(5, 2), dtype=float32)
```

output = output /keep_prob

Putting it all together

```
model_dropout = tf.keras.Sequential([
                            tf.keras.layers.Dense(20, activation=tf.nn.relu),
                            tf.keras.layers.Dropout(0.3),
                            tf.keras.layers.Dense(3, activation=tf.nn.relu),
                            tf.keras.layers.Dropout(0.3),
                            tf.keras.layers.Dense(1, activation = tf.nn.sigmoid)
                            ])
```

Rule of the thumb

>> You implement Dropout regularization only while training the network.

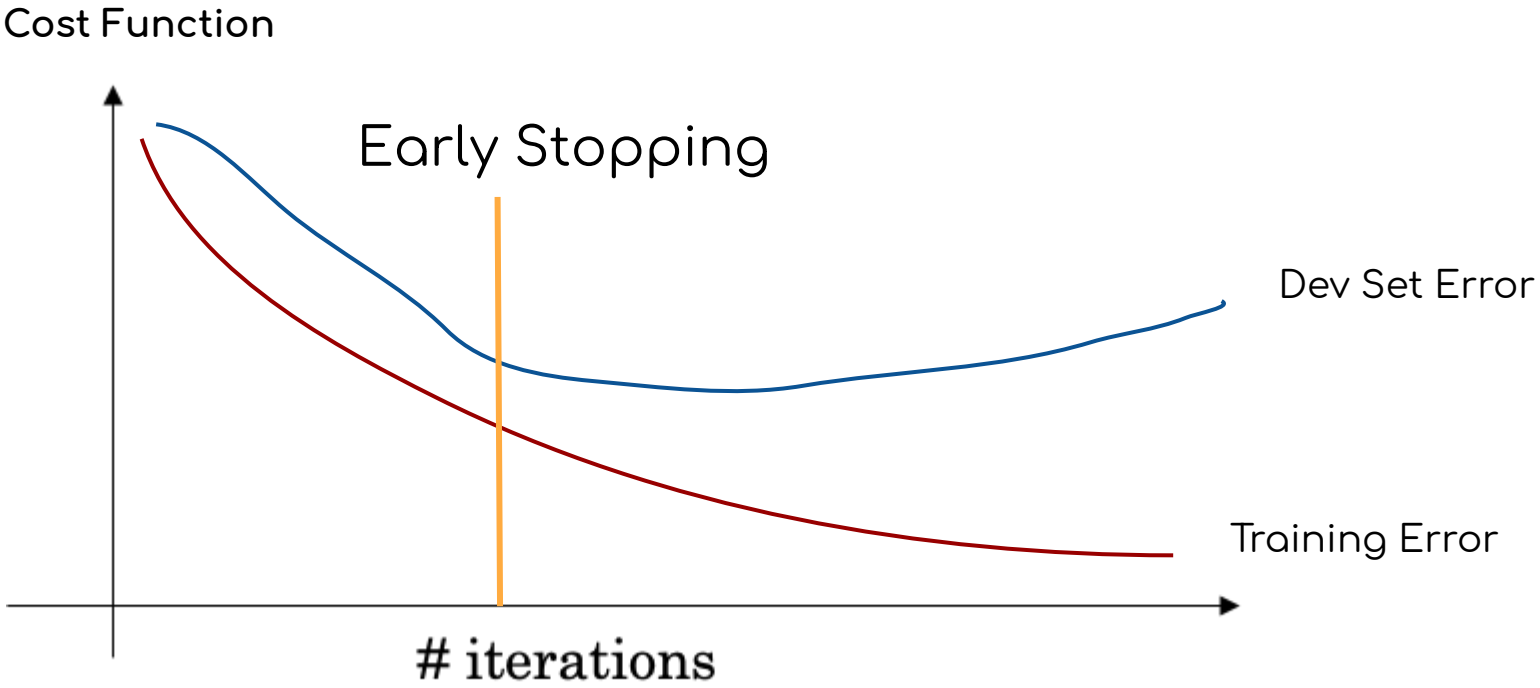
>> You do not apply it while running your testing data through it as you do not want any randomness in your predictions.

Other Regularization Methods

Original Data

Data Augmentation

Other Regularization Methods



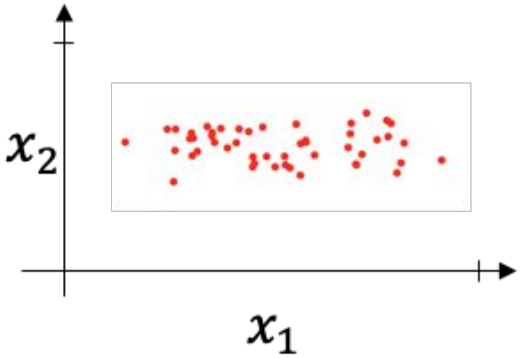
Lesson 09 Task 04 - Regularization.ipynb

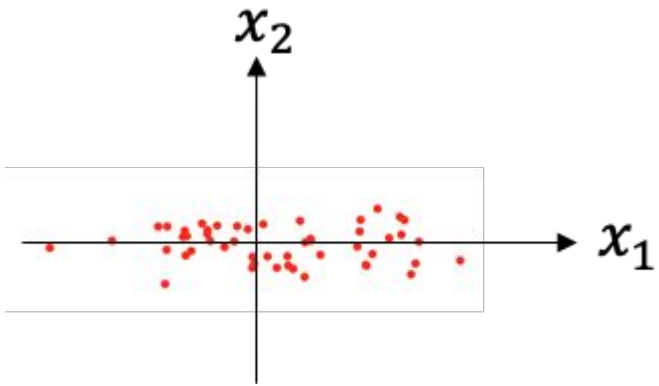
Neural Networks in Practice #02

Normalize Inputs,
Vanishing/Exploding Gradients
and Weight Initialization

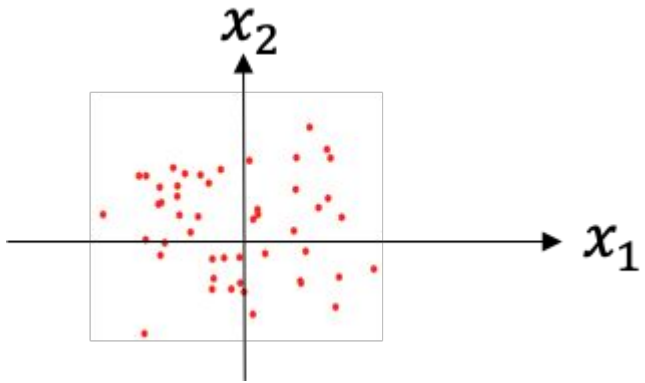
Normalizing Training Sets

Problem: x_1 and x_2 *have different scales*





$$x - \mu$$



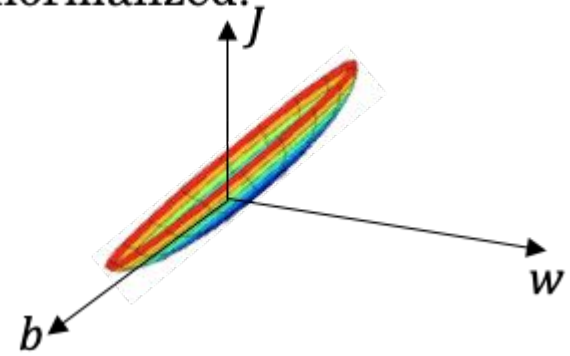
$$z\_score = \frac{x - \mu}{\sigma}$$

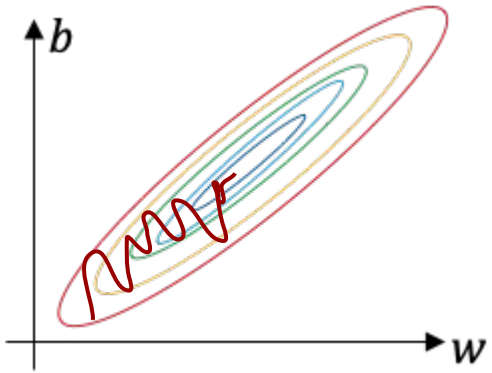
Rule of the thumb

>> Use the same μ and σ (train) to normalize the test set.

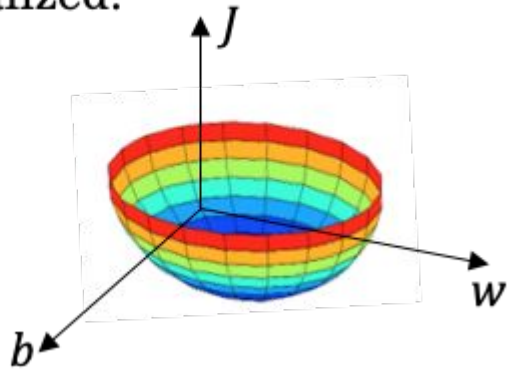
Why normalize inputs?

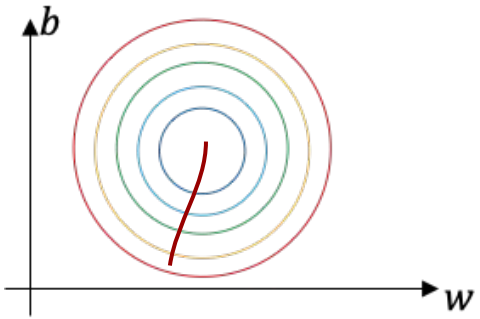
Unnormalized:



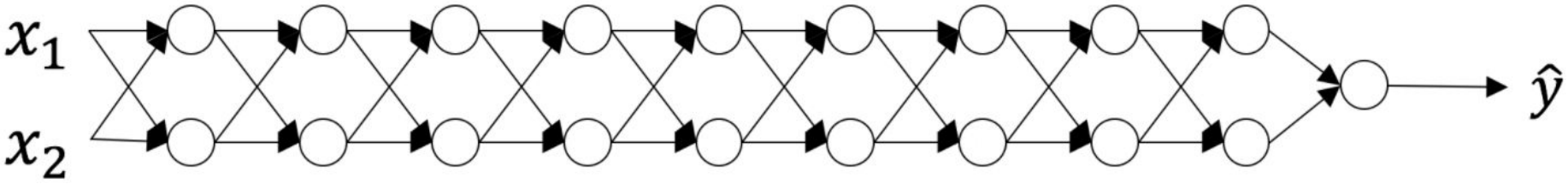


Normalized:



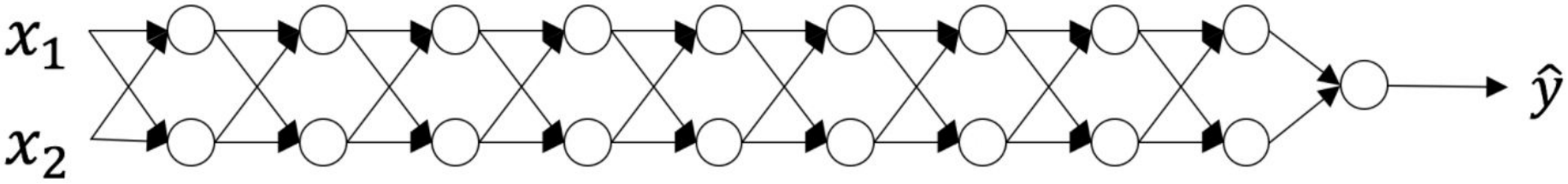


Vanishing/Exploding Gradients



At every iteration of the optimization loop (forward, cost, backward, update), we observe that backpropagated gradients are either amplified or minimized as you move from the output layer towards the input layer.

Vanishing/Exploding Gradients

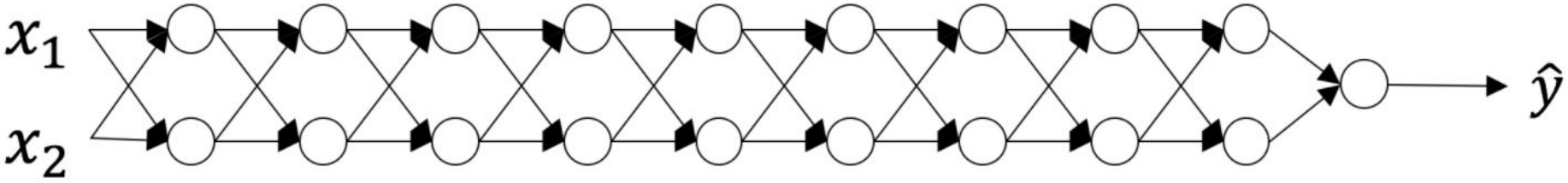


Assumptions

$$g^{[l]}(Z^{[l]}) = Z^{[l]}$$
$$b^{[l]} = 0$$
$$W^{[1]} = W^{[2]} = \ldots = W^{[L-1]} = W$$

$$Z^{[1]} = xW^{[1]}$$
$$a^{[1]} = g^{[1]}(Z^{[1]}) = Z^{[1]}$$
$$Z^{[2]} = a^{[1]}W^{[2]}$$
$$a^{[2]} = g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]}$$
$$\vdots$$
$$\hat{y} = a^{[L]} = xW^{l-1}W^{[L]}$$

Vanishing/Exploding Gradients

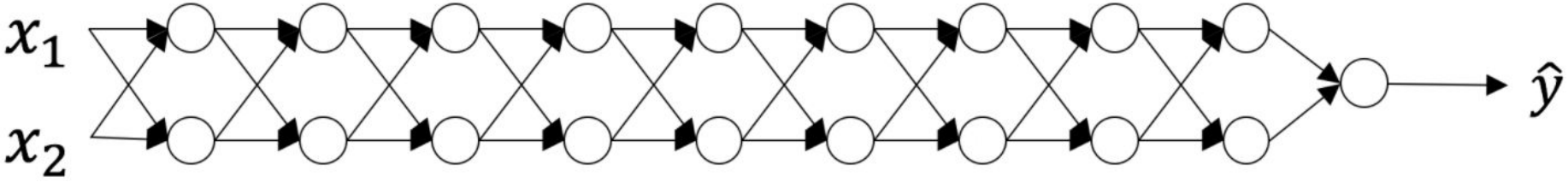


Case 1: A too-large initialization leads to exploding gradients

$$W^{[1]} = W^{[2]} = \cdots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

$$\begin{aligned} Z^{[1]} &= xW^{[1]} \\ a^{[1]} &= g^{[1]}(Z^{[1]}) = Z^{[1]} \\ Z^{[2]} &= a^{[1]}W^{[2]} \\ a^{[2]} &= g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]} \\ &\vdots \\ \hat{y} &= a^{[L]} = xW^{l-1}W^{[L]} \end{aligned}$$

Vanishing/Exploding Gradients

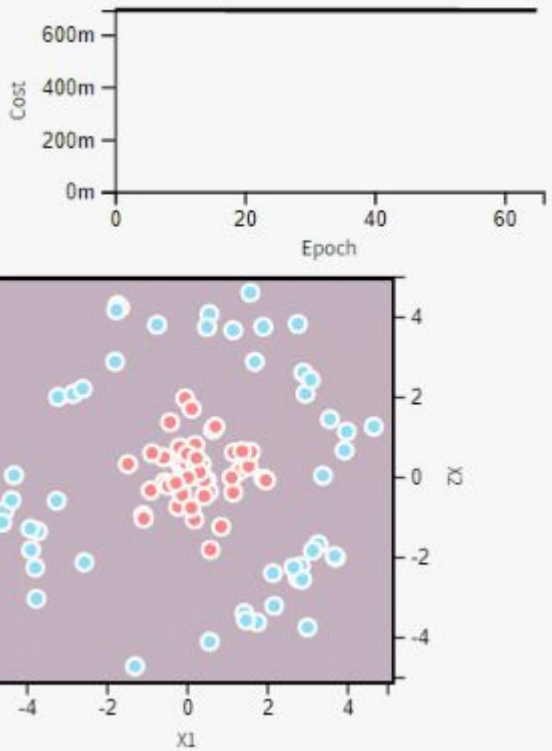


Case 2: A too-small initialization leads to vanishing gradients

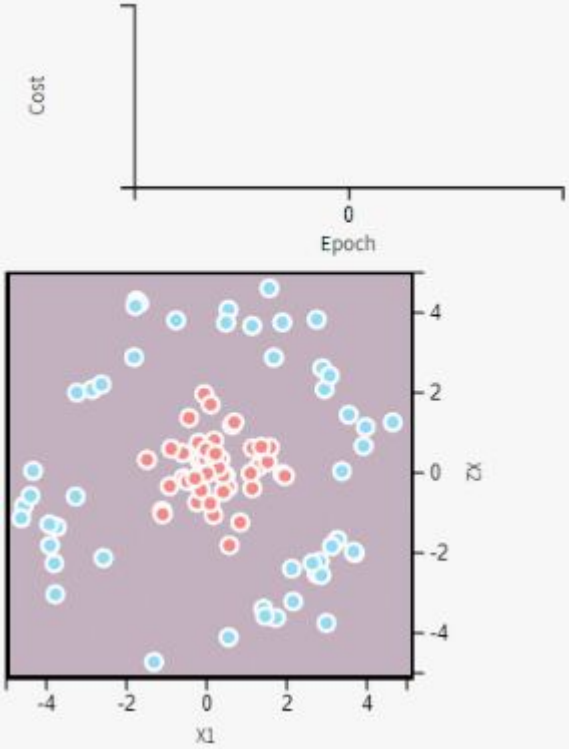
$$W^{[1]} = W^{[2]} = \cdots = W^{[L-1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

$$\begin{aligned}
Z^{[1]} &= xW^{[1]} \\
a^{[1]} &= g^{[1]}(Z^{[1]}) = Z^{[1]} \\
Z^{[2]} &= a^{[1]}W^{[2]} \\
a^{[2]} &= g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]} \\
&\vdots \\
\hat{y} &= a^{[L]} = xW^{l-1}W^{[L]}
\end{aligned}$$

Weight initialization Too Small



Weight initialization Too Large



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

How to find appropriate initialization values?

Weight Initialization

To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:

1. The mean of the activations should be zero.
2. The variance of the activations should stay the same across every layer.

$$W^{[l]} = np.random.rand(n^{[l-1]}, n^{[l]}) \times factor$$

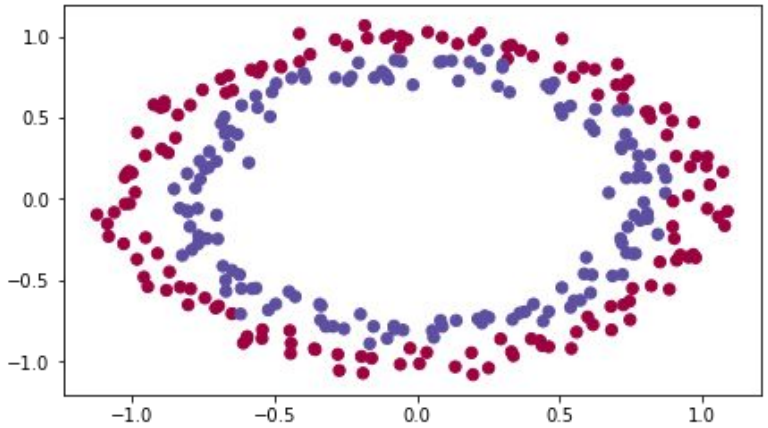
How to find appropriate initialization values?

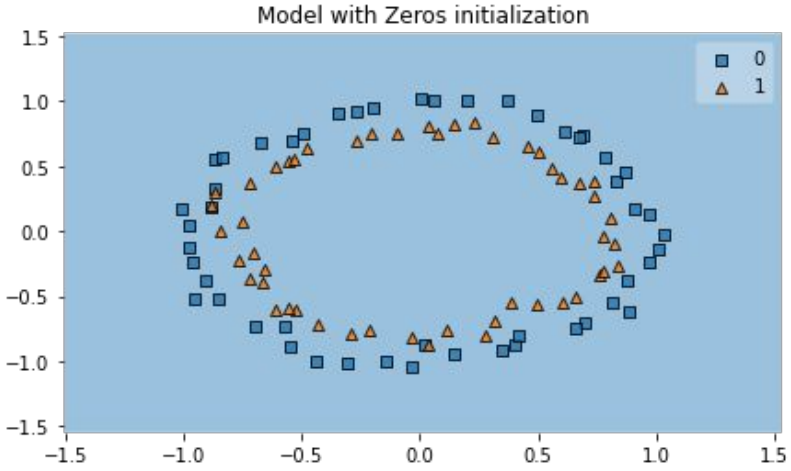
Weight Initialization

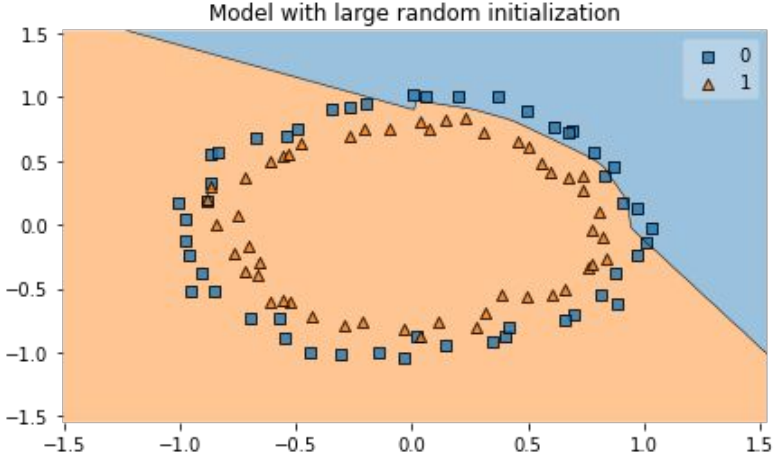
$$W^{[l]} = np.random.rand(n^{[l-1]}, n^{[l]}) \times factor$$

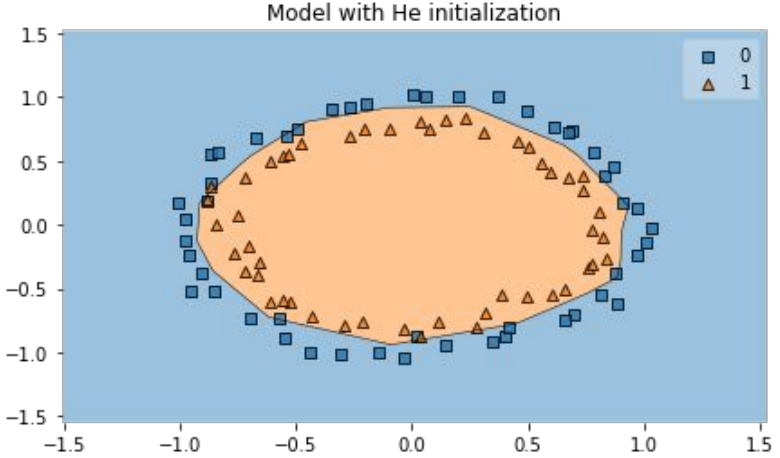
Activation Func.	Relu	Tanh
Factor	$\sqrt{\frac{2}{n^{[l-1]}}}$	$\sqrt{\frac{1}{n^{[l-1]}}}$ or $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
Author	He at al.	Xavier et al. or Yoshua Bengio et al.

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


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Model with Zeros initialization
0
1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Model with large random initialization
0
1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Model with He initialization
0
1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Lesson 09 Task 05 - Weight Initialization.ipynb

UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Next ....

Lesson #09