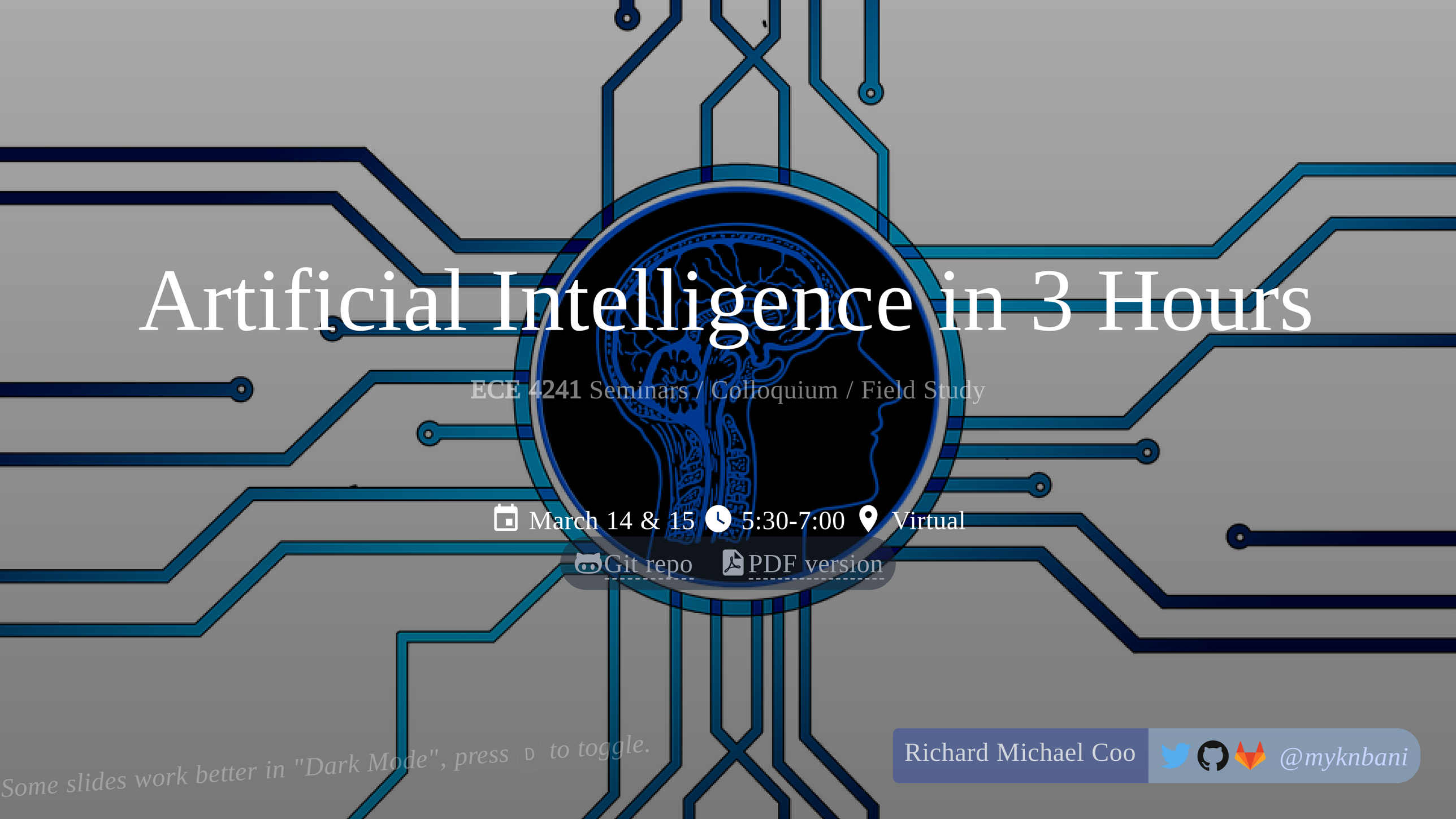


Artificial Intelligence in 3 Hours

ECE 4241 Seminars / Colloquium / Field Study

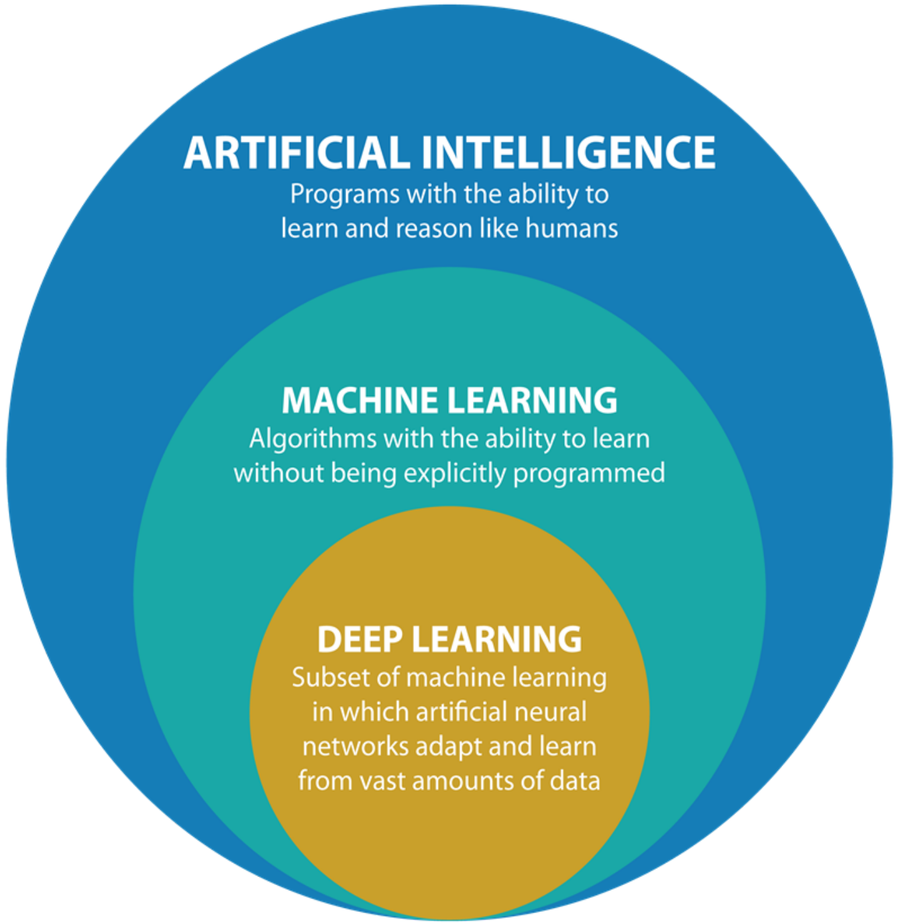
March 14 & 15 5:30-7:00 Virtual

Git repo PDF version

Some slides work better in "Dark Mode", press D to toggle.

Richard Michael Coo @myknbani

Which is impossible



HUGE field, *so let's try...*

100% online
Start instantly and learn at your own schedule.

Approx. 61 hours to complete

English
Subtitles: Arabic, French, Portuguese (European), Chinese (Simplified), Italian, Vietnamese, German, Russian, English, Hebrew, Spanish, Hindi, Japanese

Machine Learning
Stanford University

Go To Course

Intermediate Level
Some related experience required.

Approximately 4 months to complete
Suggested pace of 4 hours/week

DeepLearning.AI TensorFlow Developer Professional Certificate
DeepLearning.AI

Enroll for Free
Starts Mar 2

~~Artificial Intelligence in 3 Hours~~

~~Machine Learning in 3 Hours~~

Experiencing

Neural Networks in 3 Hours

ECE 4241 Seminars / Colloquium / Field Study

March 14 & 15 5:30-7:00 Virtual

Git repo PDF version

Some slides work better in "Dark Mode", press D to toggle.

Richard Michael Coo @myknbani

Day 1

❝ The power ⚡ of a single neuron ❞

- Theory ⏳ 5m
  - machine learning
  - regression vs classification
  - neural networks
- Webinar + Workshops
  - Hello  Jupyter ⏳ 10m
  - Hello  NumPy ⏳ 5m
  - Hello  TensorFlow ⏳ 5m
  - linear regression ⏳ 35m
    - includes: loss functions & optimizers
  - logistic regression ⏳ 25m
  - loading and saving ML models ⏳ 5m

Day 2

🧠🧠🧠 MMMOOOOAAAARRR neurons

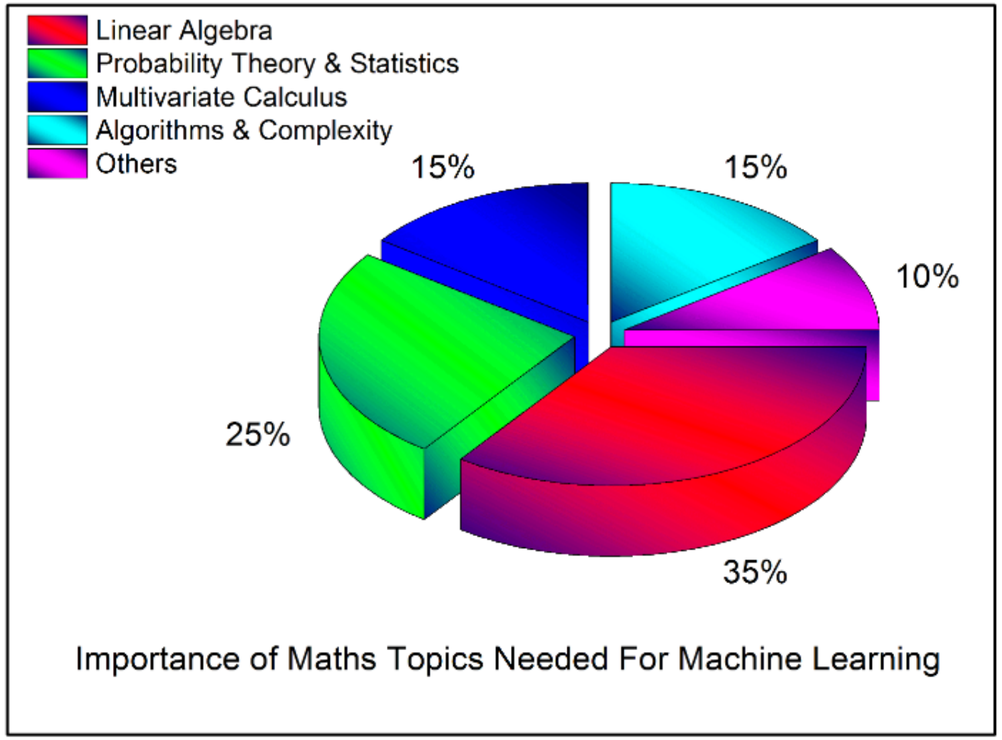
- More intense 💪 webinar + workshops
  - Vanilla neural networks
    - intuition on approximating functions ⏳ 10m
    - handwritten digit recognition ⏳ 15m
    - experiments + Fashion MNIST ⏳ 15m
    - Video: ALVINN ⏳ 2m
  - convolutional neural networks (CNN)
    - convolutional layers ⏳ 10m
    - pooling layers ⏳ 10m
    - training with your own images ⏳ 20m
    - Video: Cassava Plant Disease Detection ⏳ 3m
- Final Advice ⏳ 5m

Assumptions

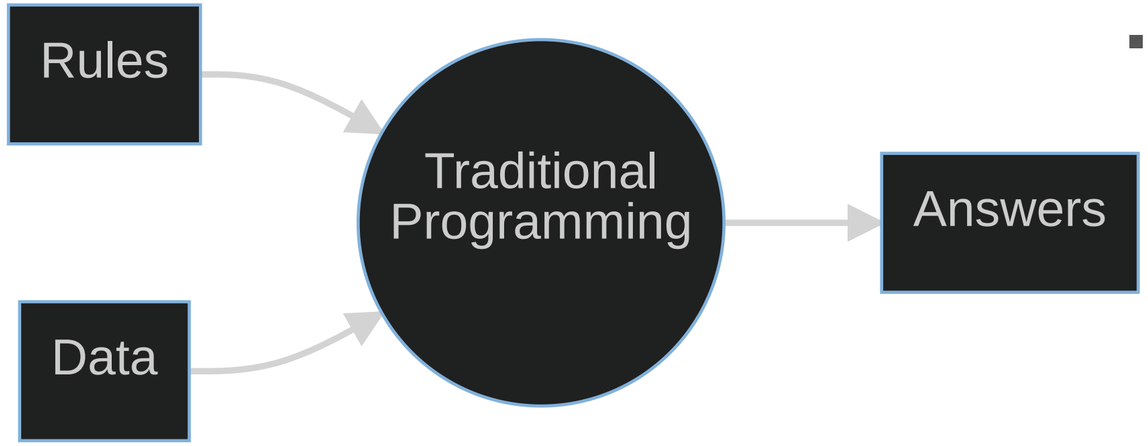
You can code in any language, preferably

- variables
- control structures (`if-else`, loops)
- sequences (strings, lists, tuples)
  - destructuring is a ➕
  - knowing dictionaries and `**kwargs` is a ➕
- functions
- OOP basics
  - getters, operator overloading is a ➕

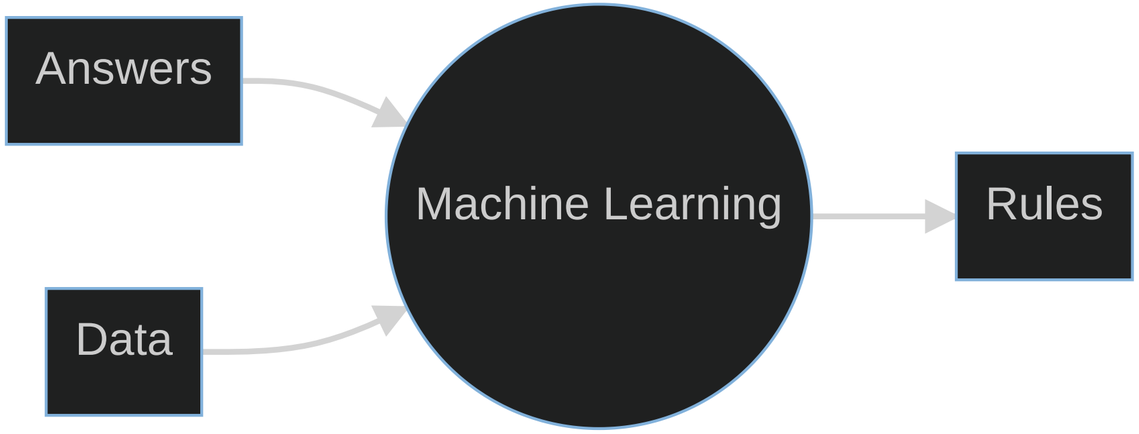
You are not allergic to



Machine Learning



A new paradigm



- 💎 HARD 😱: code some complicated logic that tells apart a dog or a cat in a picture
- ✌️ *"EASY"* ✌️: give **tons** of examples of dogs and cats, and have the machine discover patterns

> Field of study that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel *(1959)*

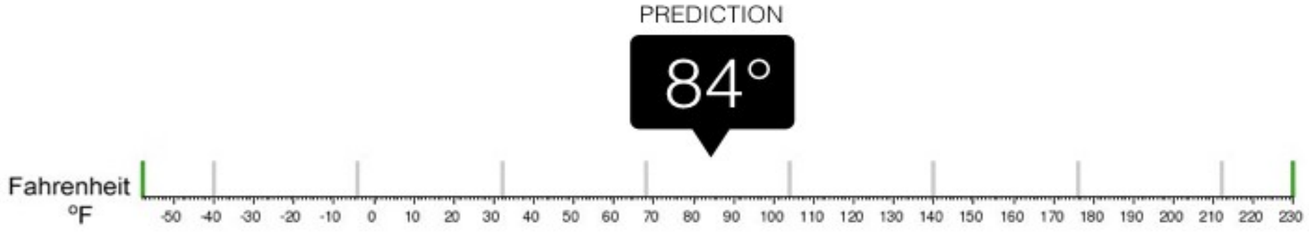
> A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, **improves with experience E.**

Tom Mitchell *(1998)*

Regression vs Classification

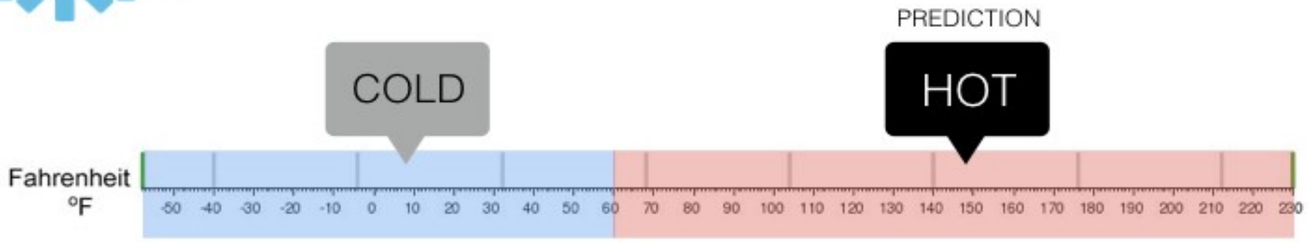
Regression

What is the temperature going to be tomorrow?

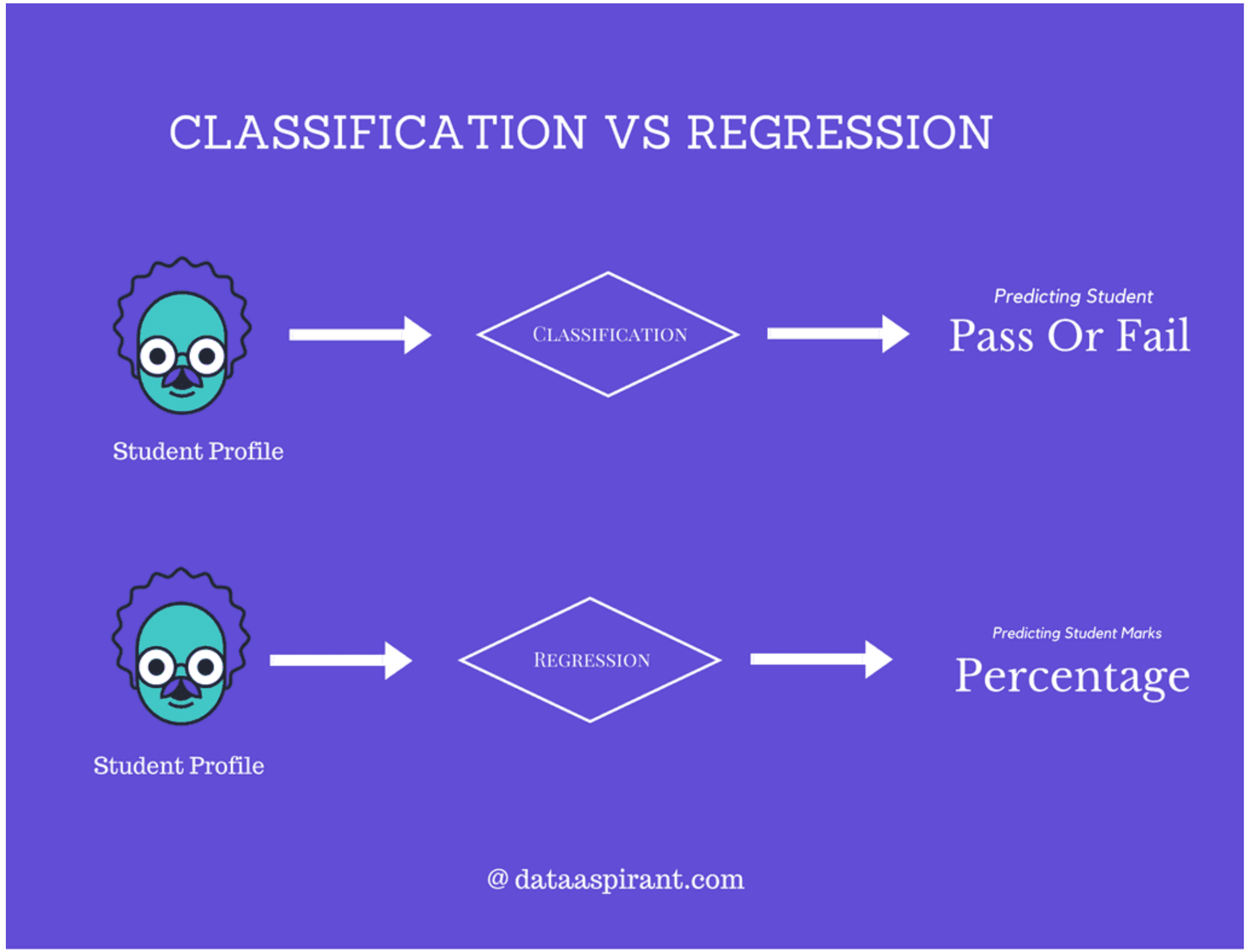


Classification

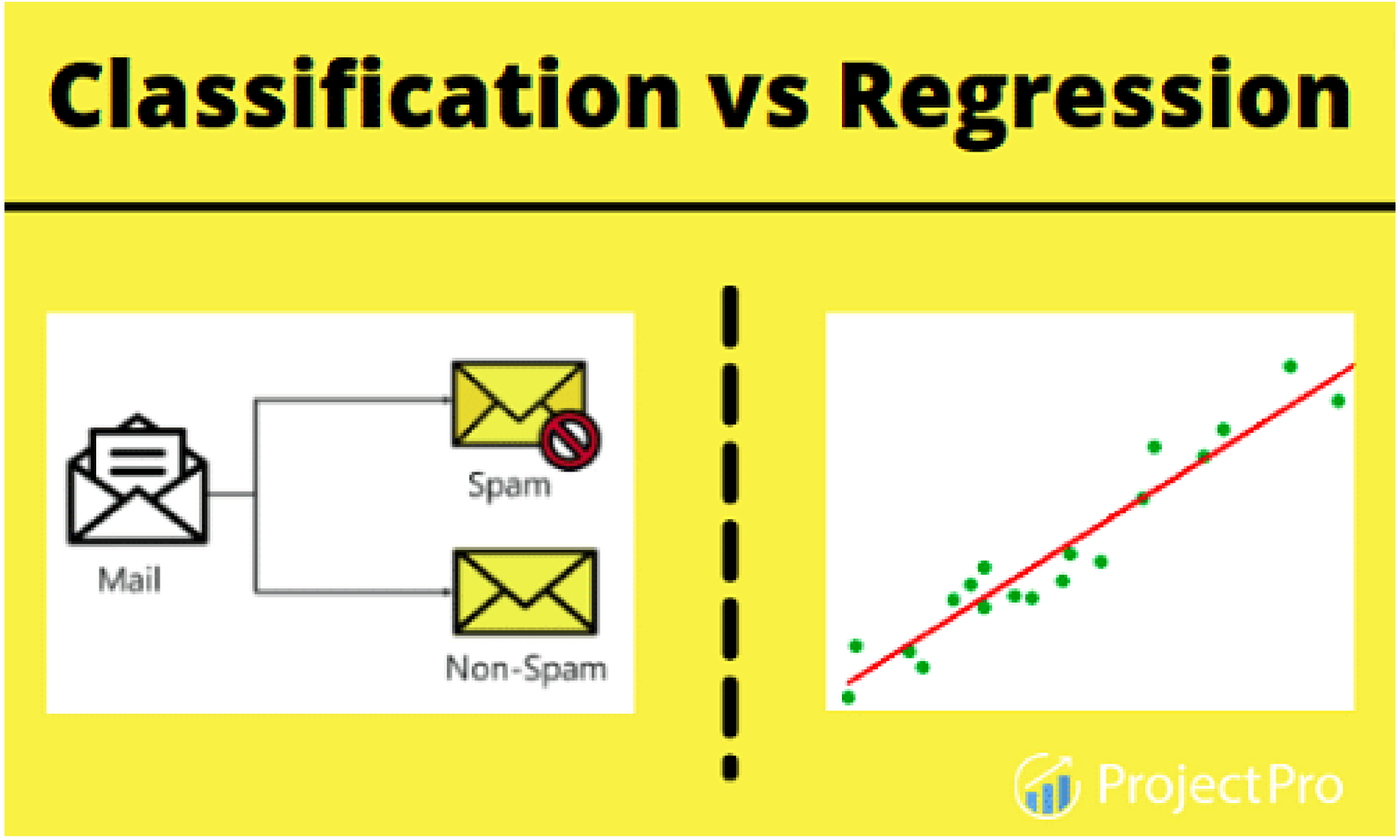
Will it be Cold or Hot tomorrow?



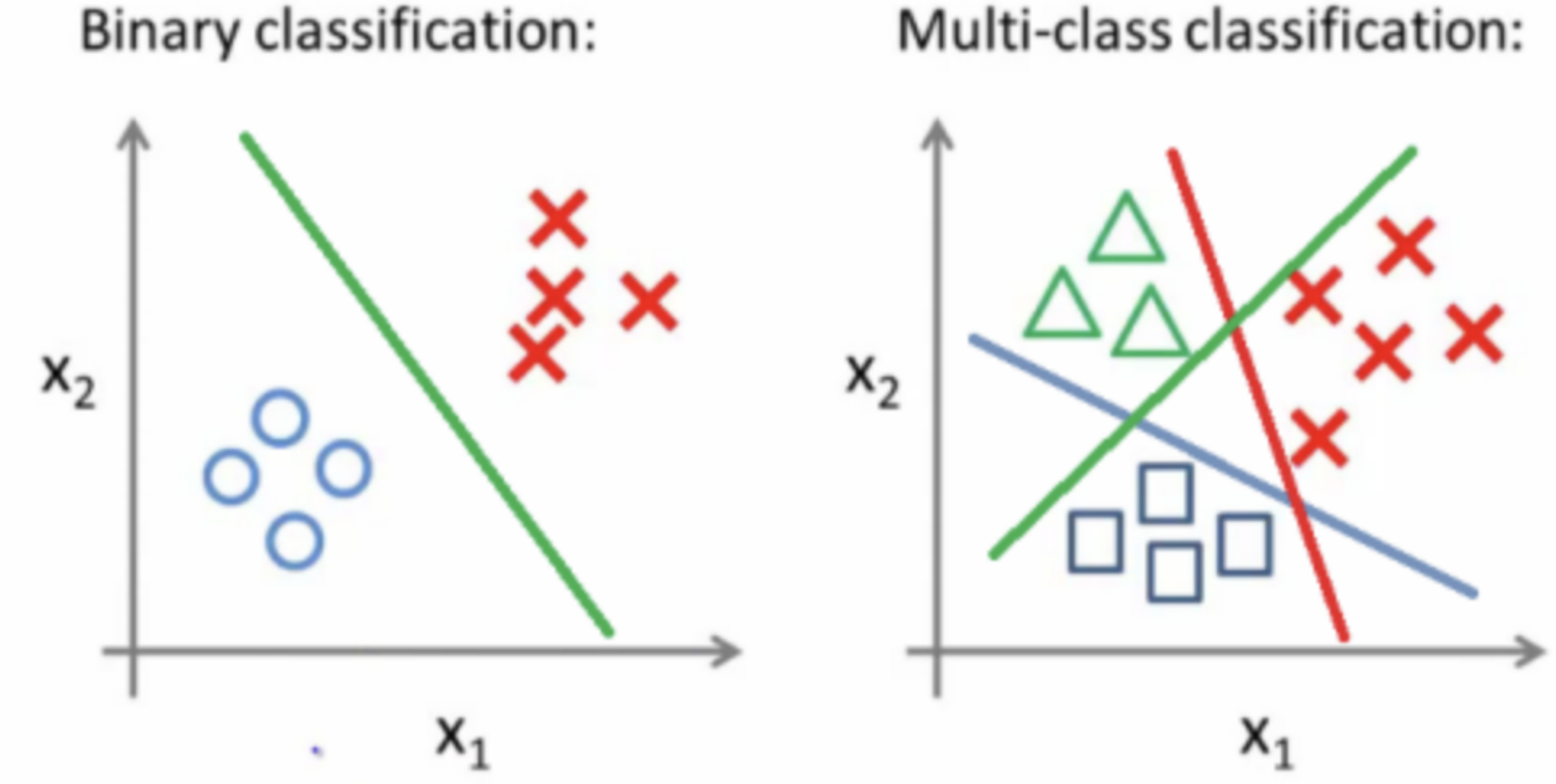
Regression vs Classification



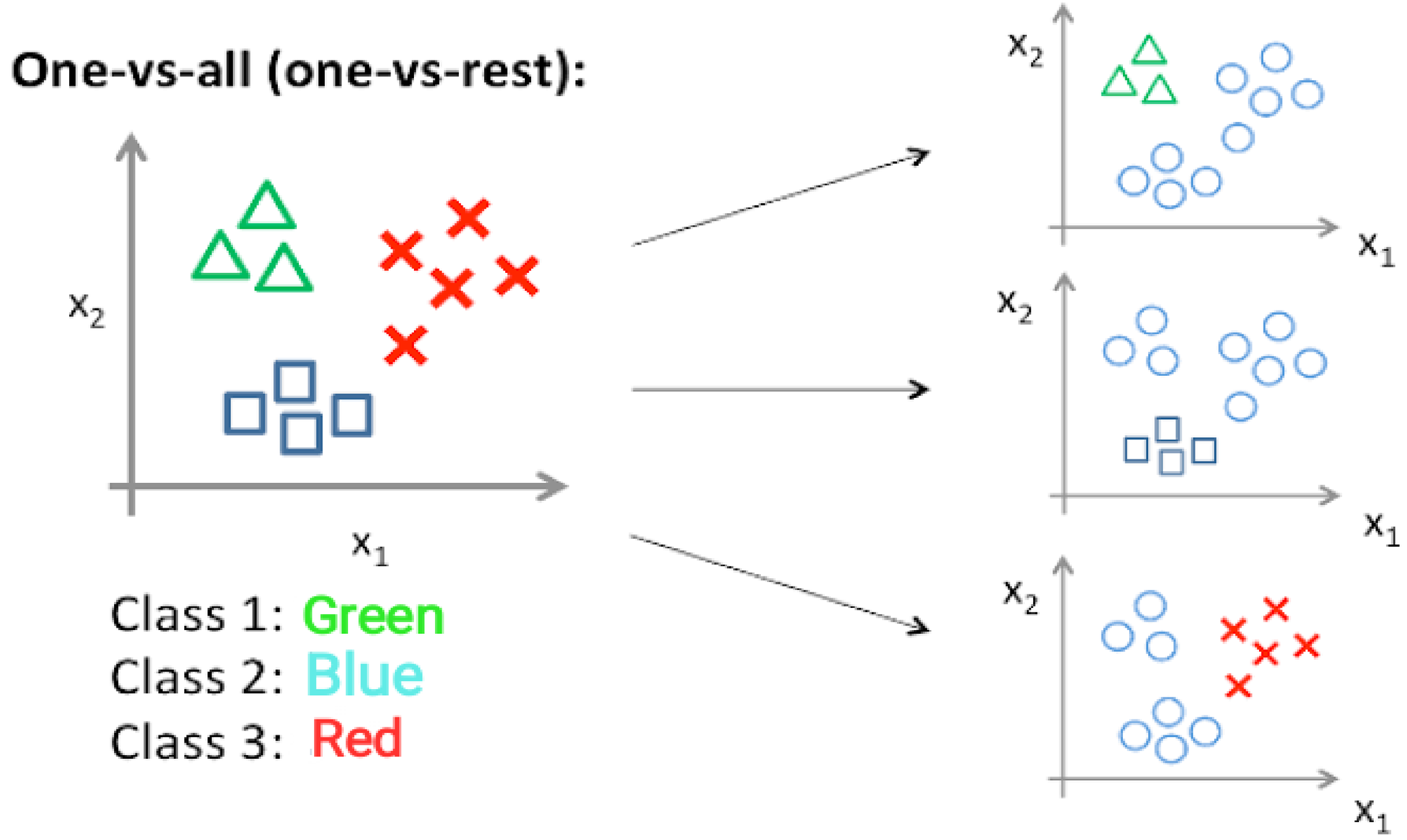
Regression vs Classification



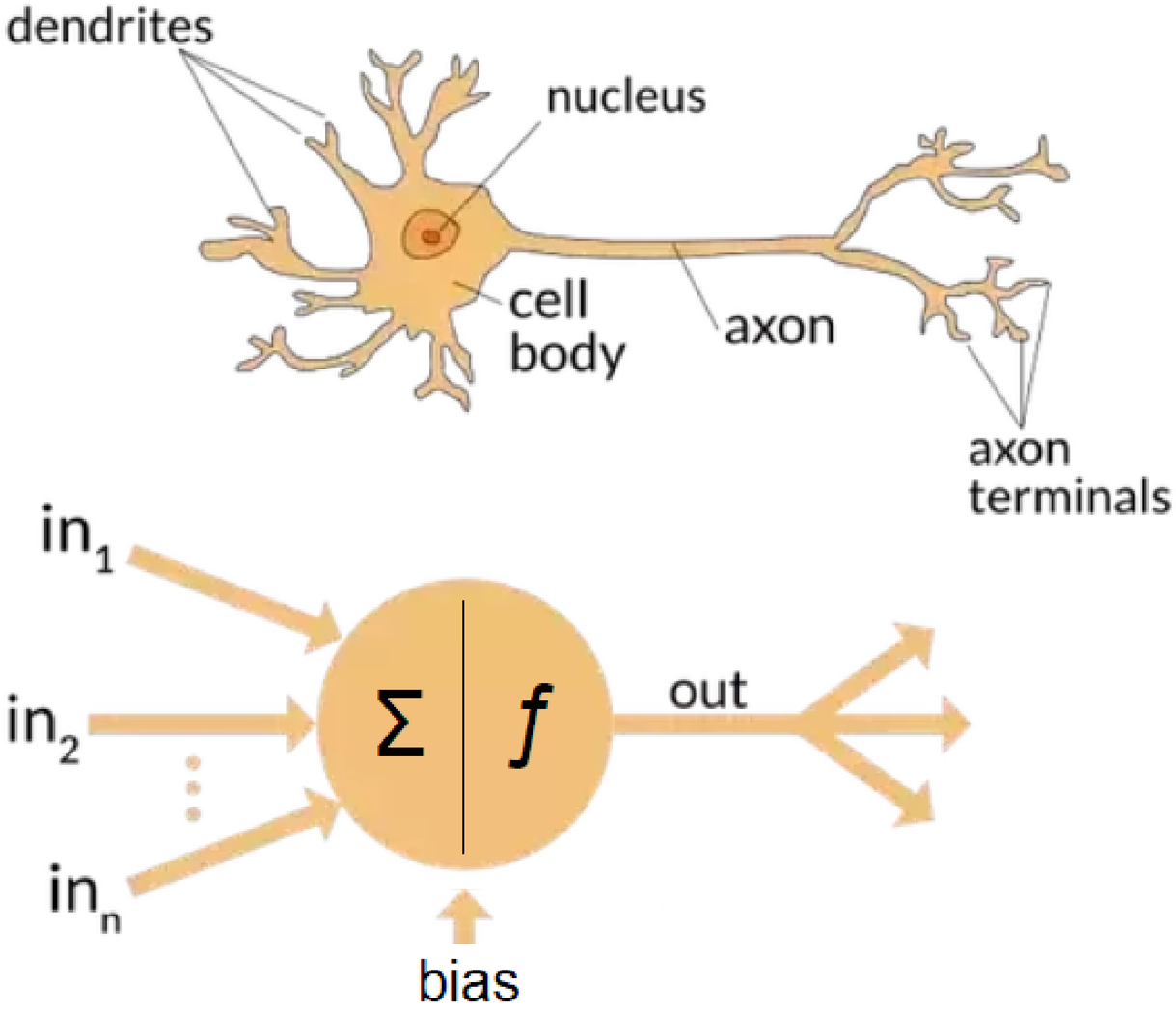
Binary vs Multi-class Classification



Multi-class Classification

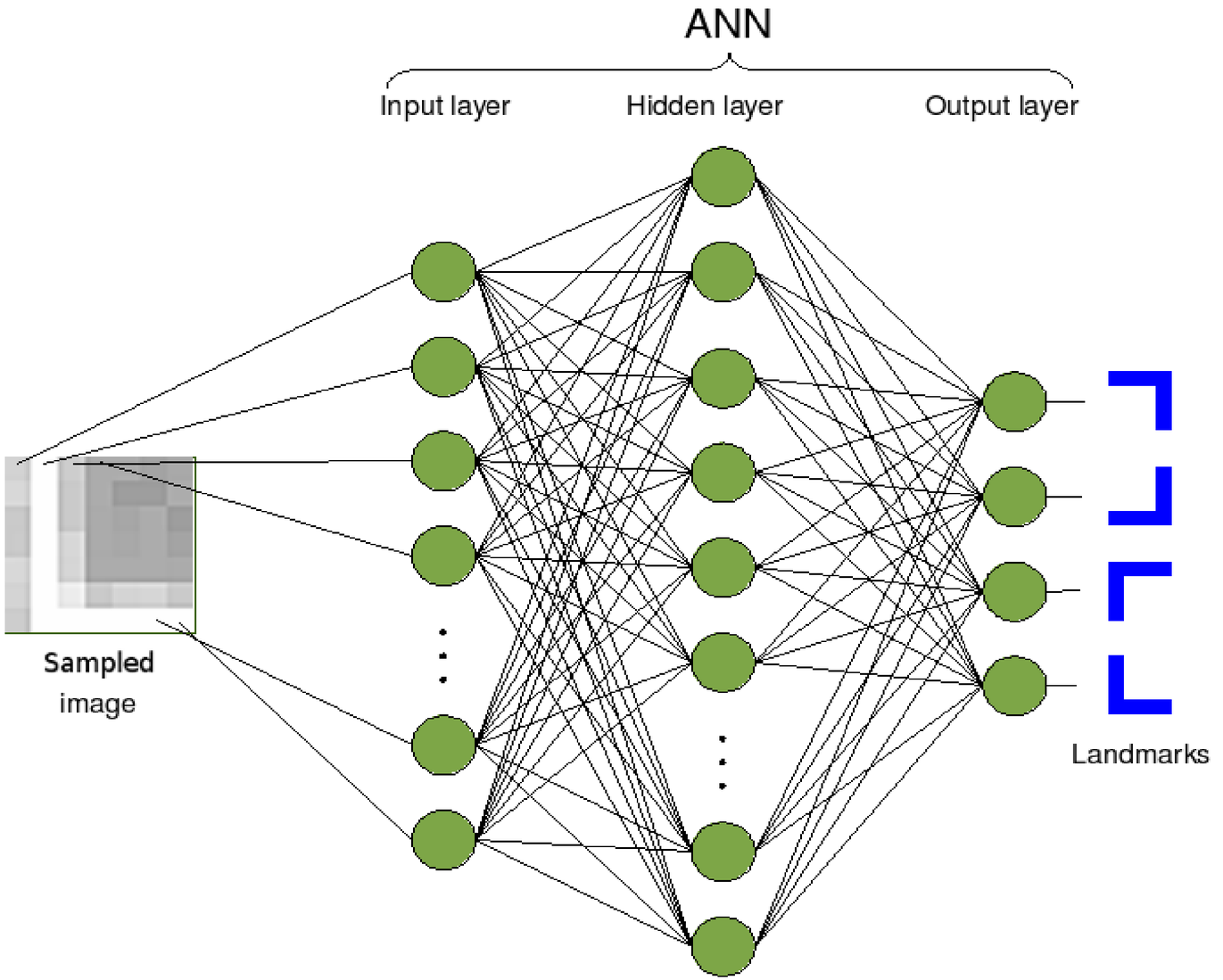


Artificial Neurons



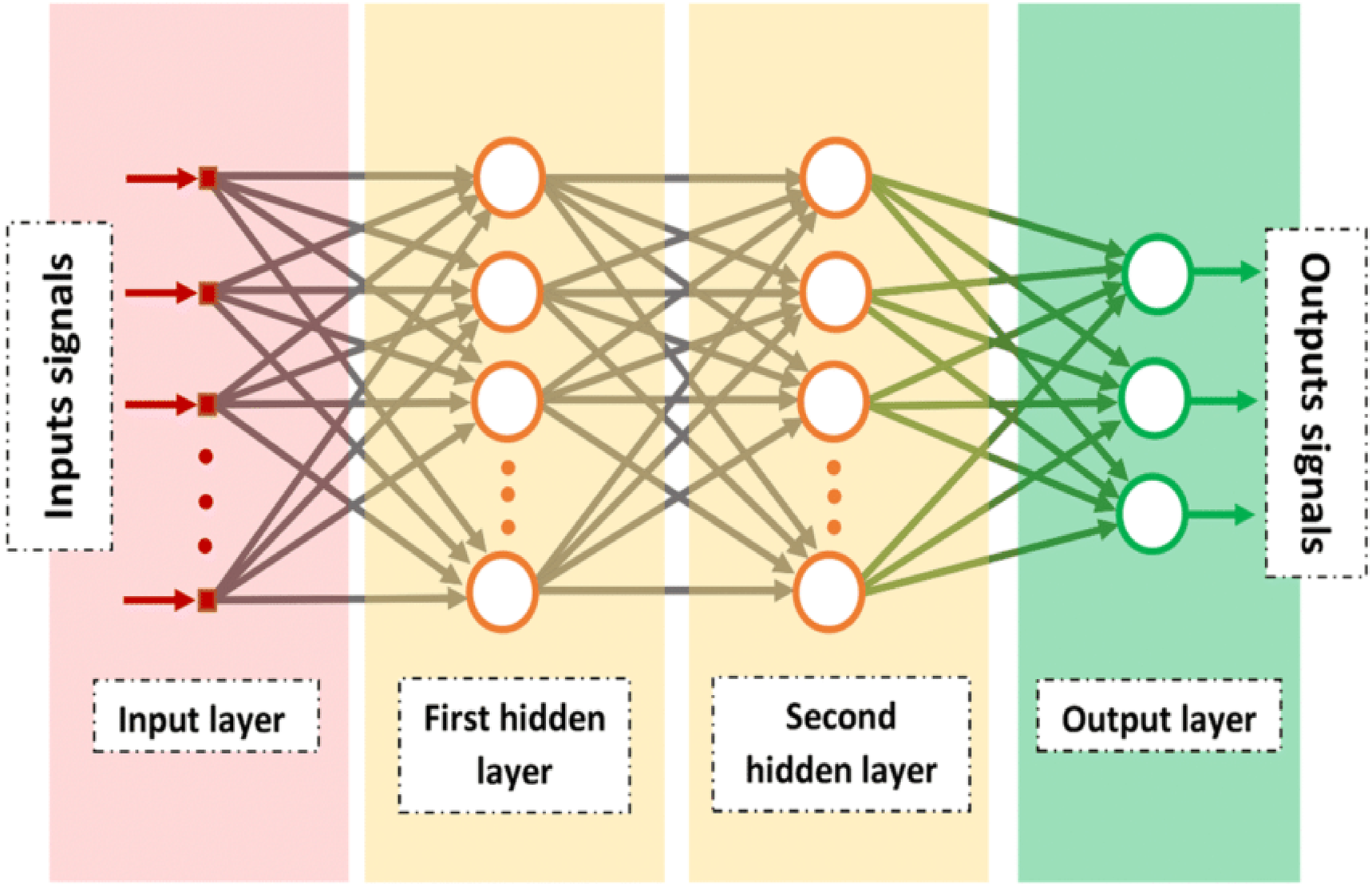
Artificial Neural Networks

1 hidden layer



Artificial Neural Networks

2 hidden layers

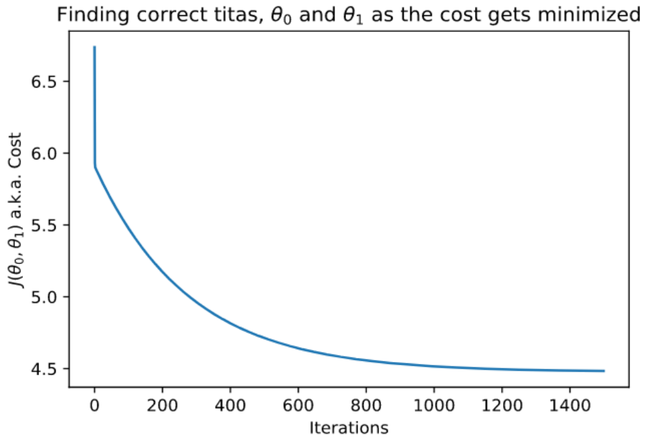


Jupyter Notebooks

- Jupyter notebooks can illustrate the exploration and analysis process step by step and allows easy switching between
  - explanatory text in Markdown format
    - can contain equations $\theta = (X^T X)^{-1} \cdot (X^T y)$
  - code

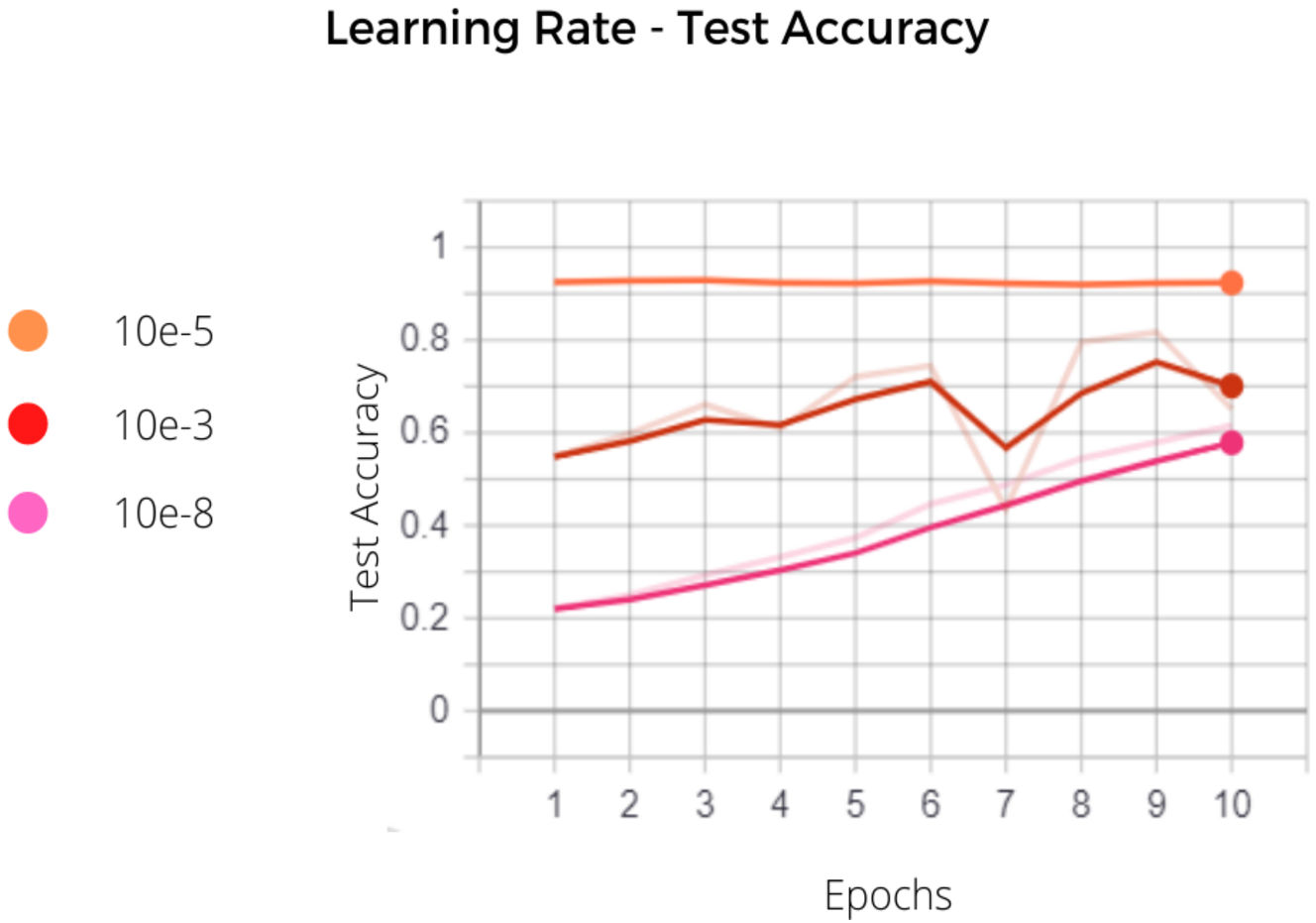
```
def normal_equation(X, y):
    return np.linalg.inv(X.T @ X) @ X.T @ y
    #              3×3           @ 3×47 @ 47×1
```

  - graphs / visualization



Jupyter Notebooks

- Document your thoughts and experiments.
  - e.g. experiments to find out which learning rate α is *"just right"* for our model



Jupyter Notebooks

- Capture results as the part of the notebook.

```
df = pd.read_sql_query(sql, connection)
df[0:20]
```

	Player	age	salary	minutes	PER	pts	ast	reb
0	Russell Westbrook	28.0	28530608.0	7854.0	29.100000	6322.0	2248.0	1978.0
1	Boban Marjanovic	28.0	7000000.0	801.0	28.650000	488.0	30.0	324.0
2	Stephen Curry	28.0	34682550.0	7951.0	28.033333	6274.0	1669.0	1124.0
3	Kevin Durant	28.0	25000000.0	5561.0	27.800000	4270.0	771.0	1280.0
4	LeBron James	32.0	33285709.0	7996.0	26.800000	5617.0	1671.0	1621.0
5	James Harden	27.0	28299399.0	9053.0	26.433333	6949.0	2083.0	1619.0
6	Chris Paul	31.0	24599495.0	7198.0	26.133333	4114.0	2139.0	990.0
7	Kawhi Leonard	25.0	18868625.0	6887.0	25.166667	4468.0	607.0	1384.0
8	DeMarcus Cousins	26.0	18063850.0	9189.0	24.860000	7053.0	1088.0	3088.0
9	Hassan Whiteside	27.0	23775506.0	5780.0	24.833333	2913.0	93.0	2435.0
10	Jimmy Butler	27.0	19301070.0	7796.0	22.566667	4516.0	950.0	1207.0
11	Blake Griffin	27.0	29512900.0	5602.0	22.533333	3534.0	824.0	1298.0
12	Damian Lillard	26.0	26153057.0	8295.0	22.333333	5623.0	1458.0	1048.0

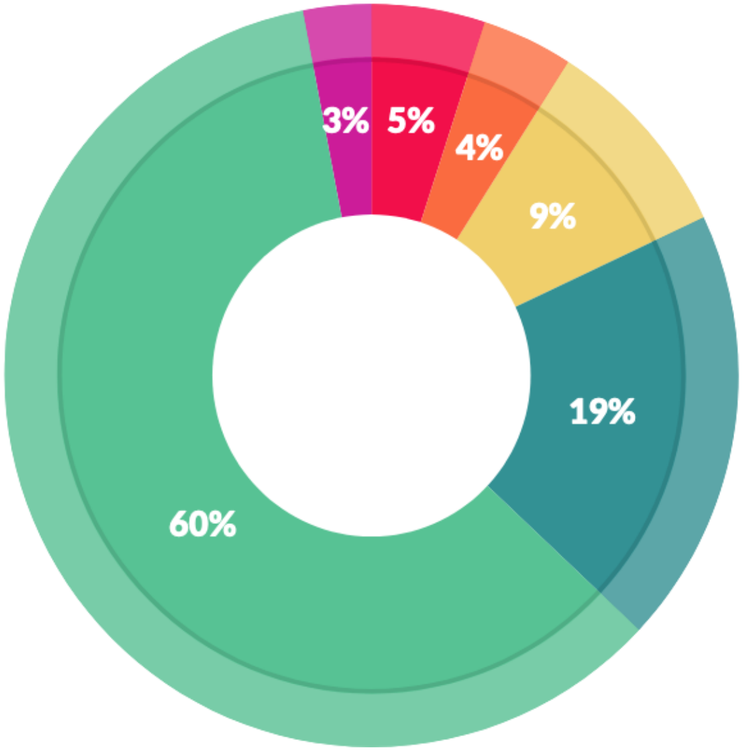
- Re-run only a small portion of your code

- Share your work with a peer

Garbage-in, garbage-out: real-world data needs preprocessing

How a Data Scientist Spends Their Day

Here's where the popular view of data scientists diverges pretty significantly from reality. Generally, we think of data scientists building algorithms, exploring data, and doing predictive analysis. That's actually not what they spend most of their time doing, however.

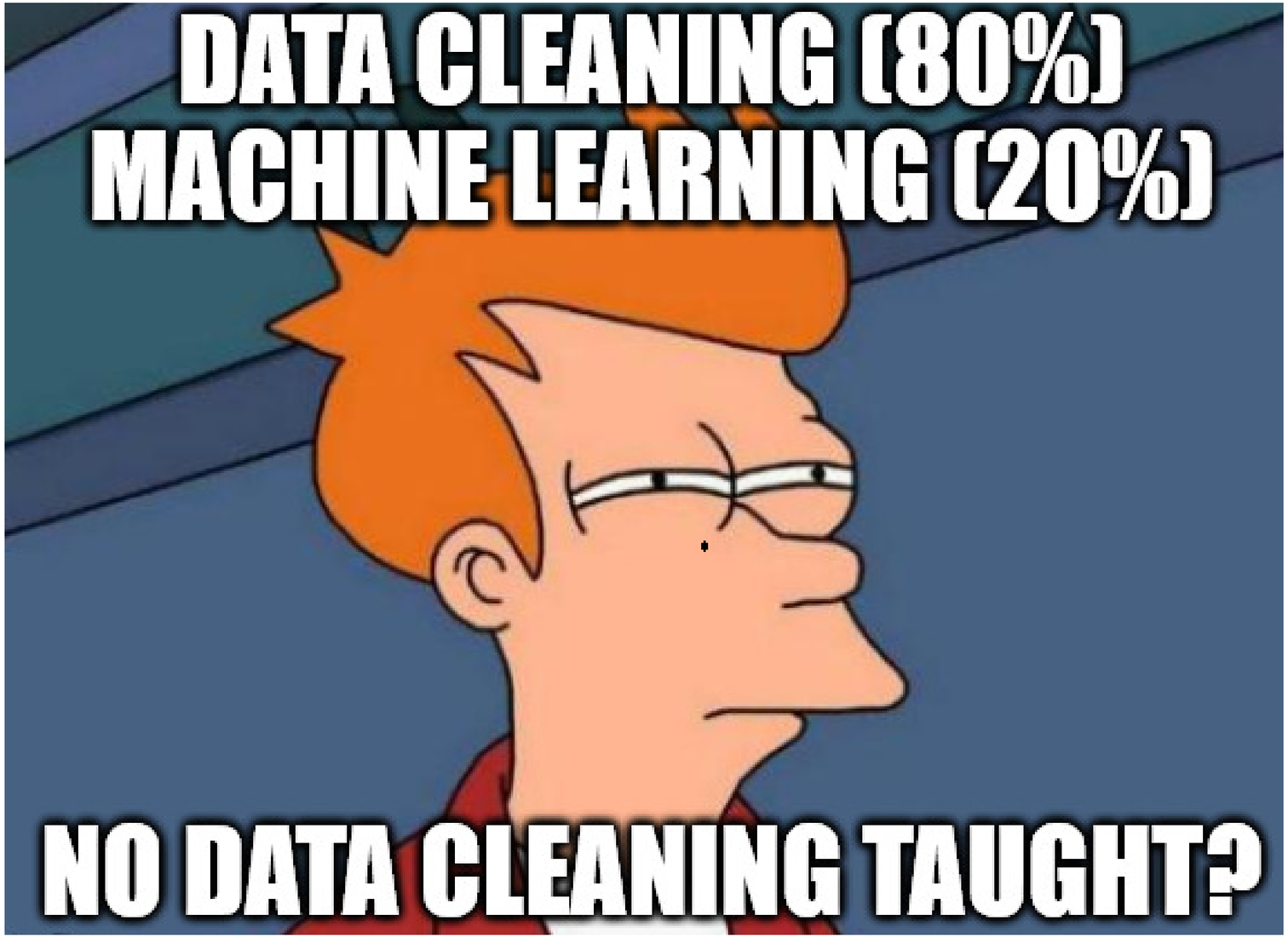


What data scientists spend the most time doing

- *Building training sets: 3%*
- *Cleaning and organizing data: 60%*
- *Collecting data sets; 19%*
- *Mining data for patterns: 9%*
- *Refining algorithms: 4%*
- *Other: 5%*

Source: survey by CrowdFlower 2016
https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf

Notebooks allow us to explore and preprocess data and images before jumping to building our ML models



Google Colab

Google "Colaboratory"

- ☁️ cloud-based, zero configuration required
  - you don’t need a Python environment installed
- Free access to *"datacenter"* GPUs
  - faster, and more VRAM than budget gaming laptops
  - GTX 1080 is only around 15% faster on average
- Easy sharing
- Google Drive integration
- Only supports Python

> Colab focuses on supporting Python and its ecosystem of third-party tools. We’re aware that users are interested in support for other Jupyter kernels (eg R or Scala). We would like to support these, but don’t yet have any ETA.

VSCode

- can work offline if your net sucks
- need to download, install Python and friends
- need to use your own hardware
- julia R JS can use any language

🧡 Colab GPUs, but also 💙 your own VSCode setup?

Hello NumPy

"Numerical Python"

- mathematical functions
- random number generators
- **inear algebra** routines
- n-dimensional arrays
  - generic enough to represent matrices and vectors

4x2 matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

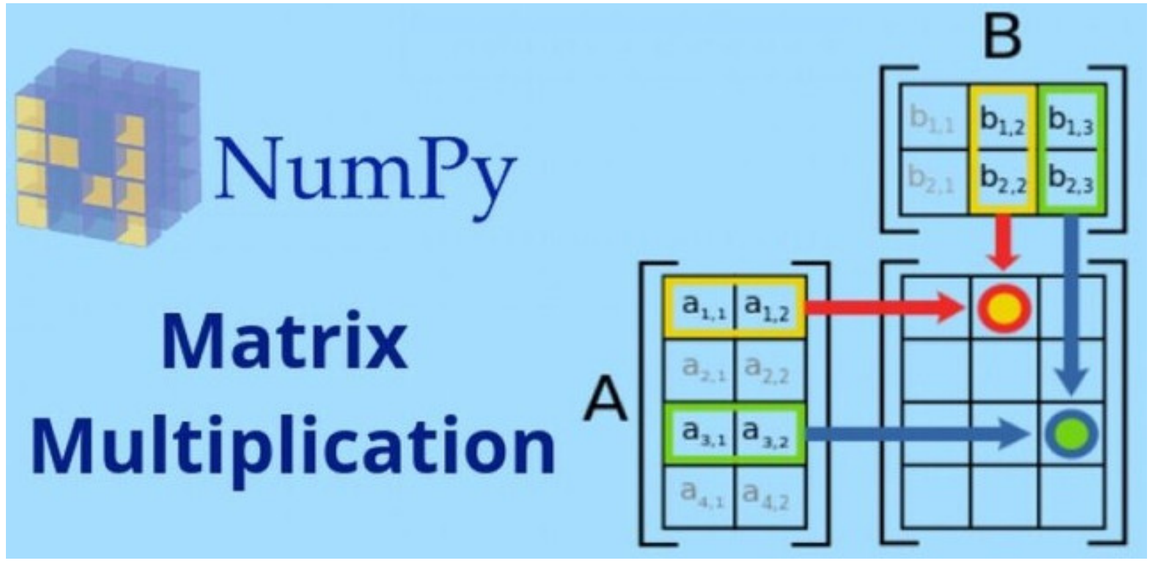
```
A = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8]
])
```

4-element vector (a.k.a. 4x1 matrix)

$$v = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

```
v = np.array([
    [1],
    [2],
    [3],
    [4]
]) # written to look like a vector for readability

# hate all the []? transpose from a 1×4 matrix
v = np.array([[
    1,
    2,
    3,
    4
]]).T
# note we just wrote each number on a separate line
# for readability of our code. [1, 2, 3, 4] IS A ROW
```

Matrix libs are multithreaded



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

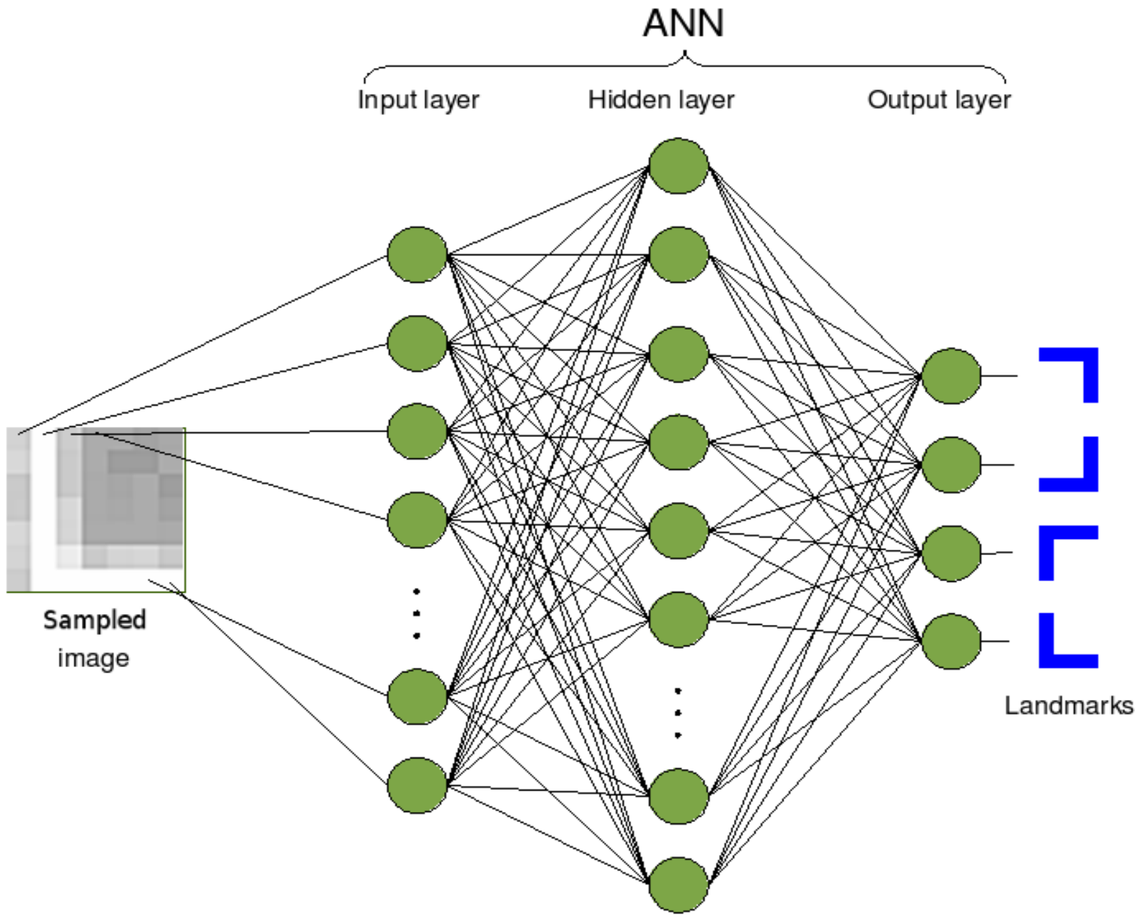
$$1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58$$

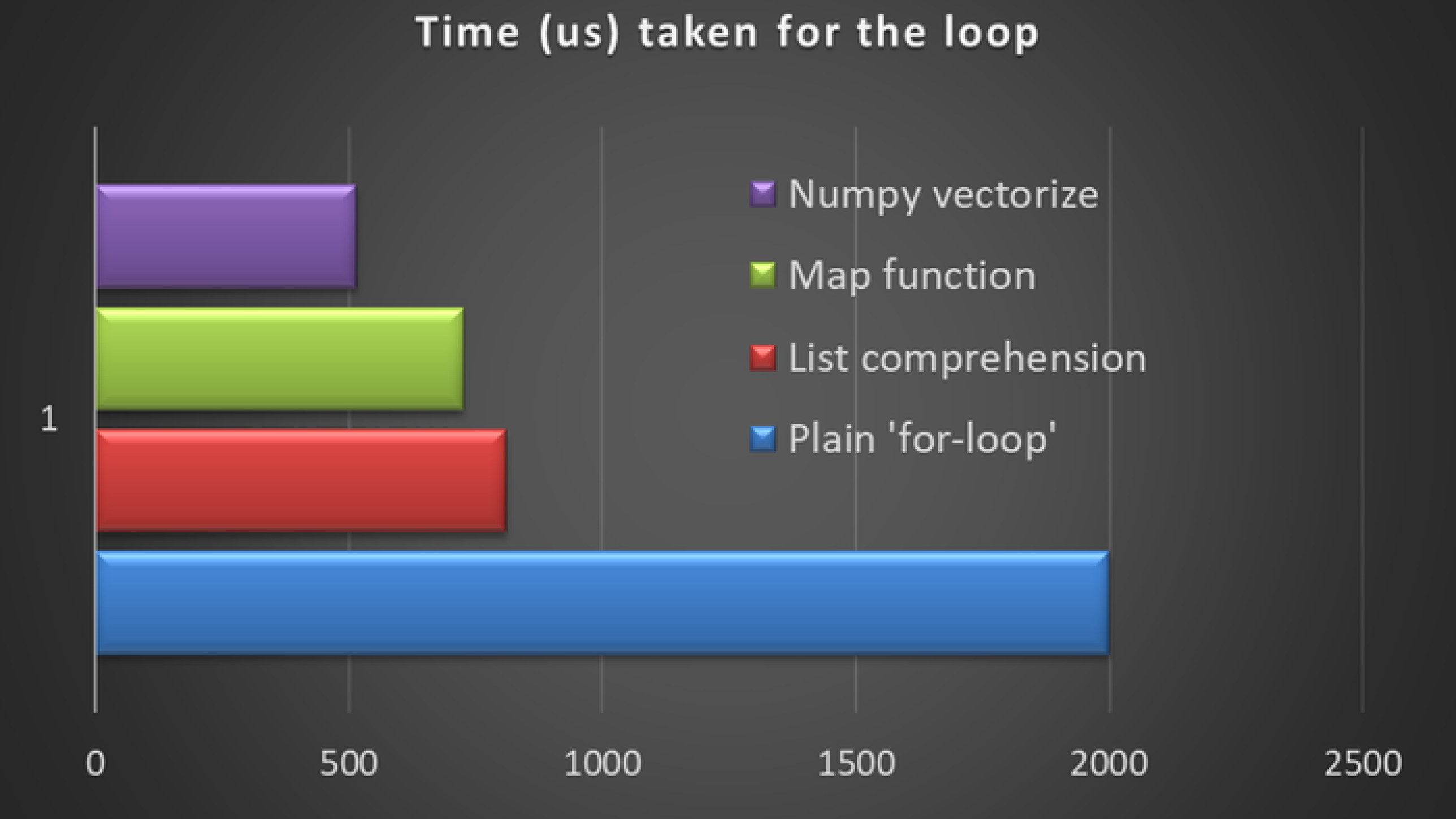
- Matrices of numerical libraries like NumPy can take advantage of multiple CPU cores
- Libraries like TensorFlow can also use the GPU
  - **GPUs** have **more** cores
  - on and , on only
- on the images, differently-colored operations can be done **in parallel**, rather than one at a time

Machines need lots of data

just like some of us need lots of examples to learn well

- ⚠️ **Spoiler alert**: Example for image classification
  - image pixels are **flattened** such that they become a single ROW in a matrix
    - an image of 100x100 pixels = 10k inputs if graysale, 30k if colored (RGB)
  - if you train with 30,000 pictures of cats and 30,000 pictures of dogs, then
    - that's 60,000 rows *(1 row per training sample)*
  - $60,000 \times 10,000$ matrix? *Take that EMath 2200!!*
  - Each neuron (*green circle*) is also a matrix 😁😁
  - Many operations are safe to do in parallel
    - e.g. addition, multiplication, summation




Time (us) taken for the loop
Numpy vectorize
Map function
List comprehension
Plain 'for-loop'
1
0
500
1000
1500
2000
2500

How To Speed Up Your Loop In Python

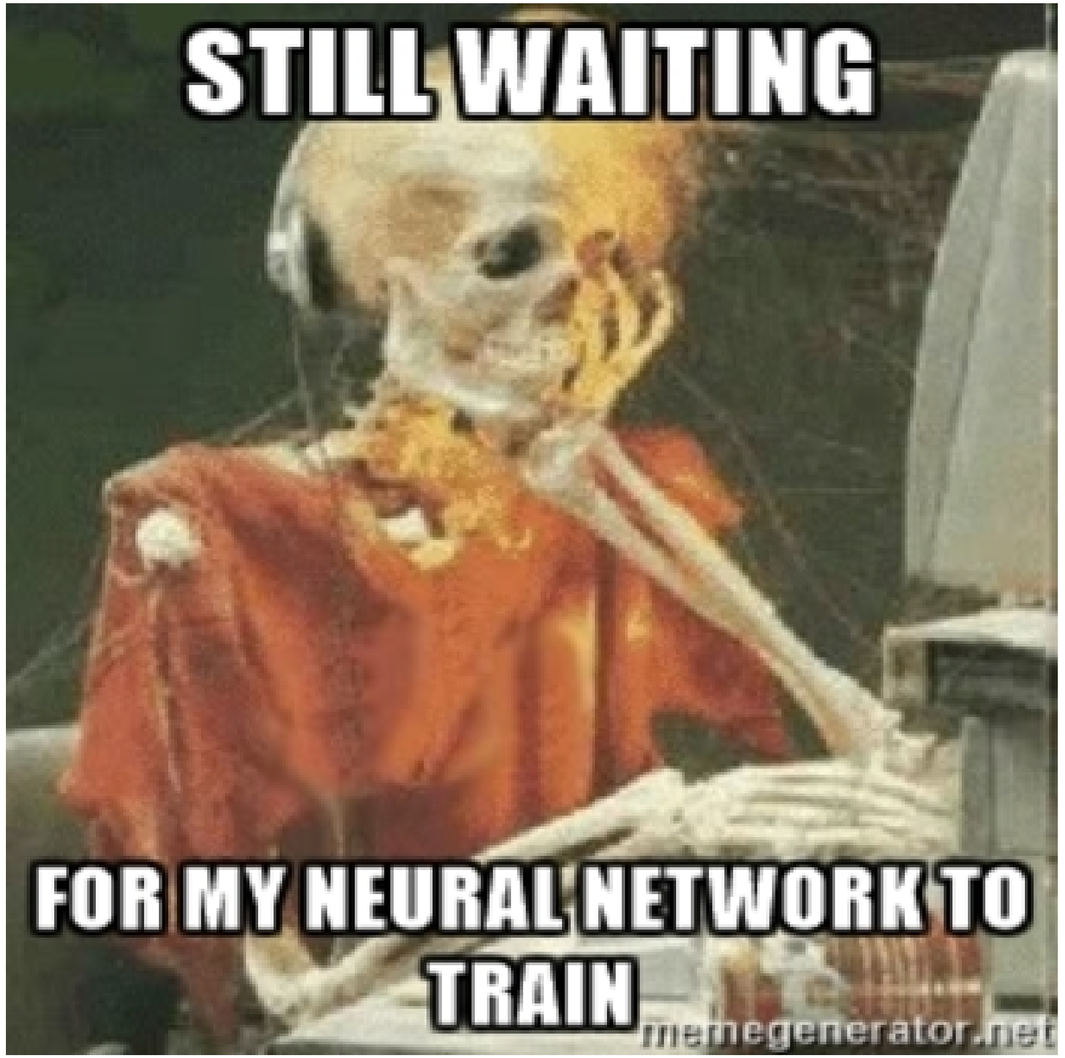
t
roseconds
p
40 DataFrame

rage
iseconds
p
40 DataFrame

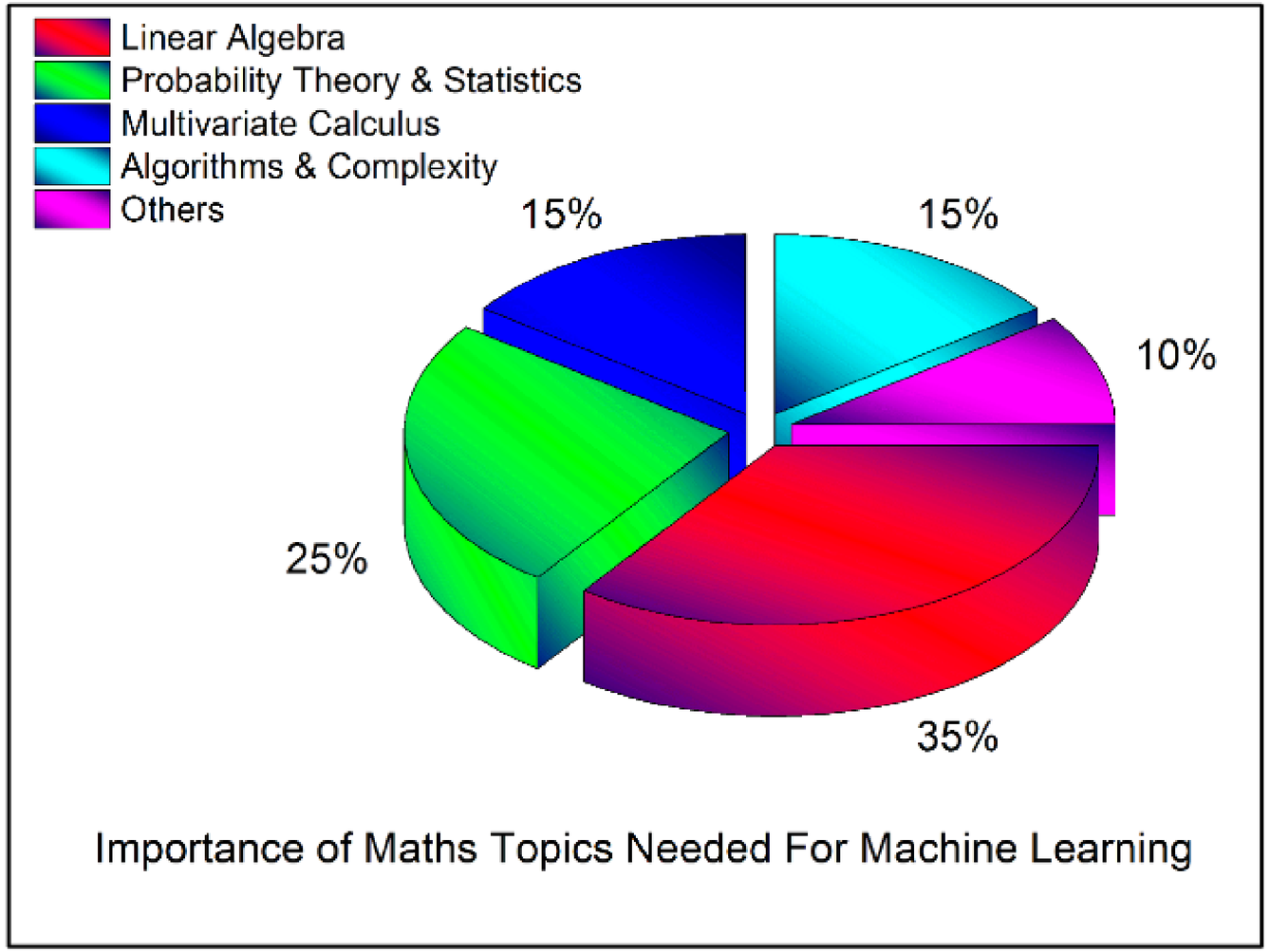
w
onds
p
40 DataFrame

Time
each l

Multi~~tasking~~threading is fast 🏃🐎🏇🐇🚀



Linear Algebra and GPUs makes machine learning feasible



especially

on

LARGE

datasets

Fun fact: linear algebra and GPUs also power game dev

Some NumPy Operations

Convert Python lists to Numpy arrays

```
matrix = np.array([
    [i, i * 3, i ** 2] for i in range(1, 8 + 1)
])
```

```
array([[ 1,  3,  1],    # 0
       [ 2,  6,  4],    # 1
       [ 3,  9,  9],    # 2
       [ 4, 12, 16],    # 3
       [ 5, 15, 25],    # 4
       [ 6, 18, 36],    # 5
       [ 7, 21, 49],    # 6
       [ 8, 24, 64]])   # 7
```

Slicing: get the first rows

```
matrix[0:3]
```

```
array([[1, 3, 1],
       [2, 6, 4],
       [3, 9, 9]])
```

Slicing: get all rows, but only the first two columns

```
matrix[:, 0:2]
```

```
array([[ 1,  3],
       [ 2,  6],
       [ 3,  9],
       [ 4, 12],
       [ 5, 15],
       [ 6, 18],
       [ 7, 21],
       [ 8, 24]])
```

Slicing: get the last column, first 5 rows

```
# note the extra [square brackets]
matrix[0:5, [-1]]

# positive index also works, but sometimes harder
matrix[0:5, [2]]
```

```
array([[ 1],
       [ 4],
       [ 9],
       [16],
       [25]])
```

TensorFlow

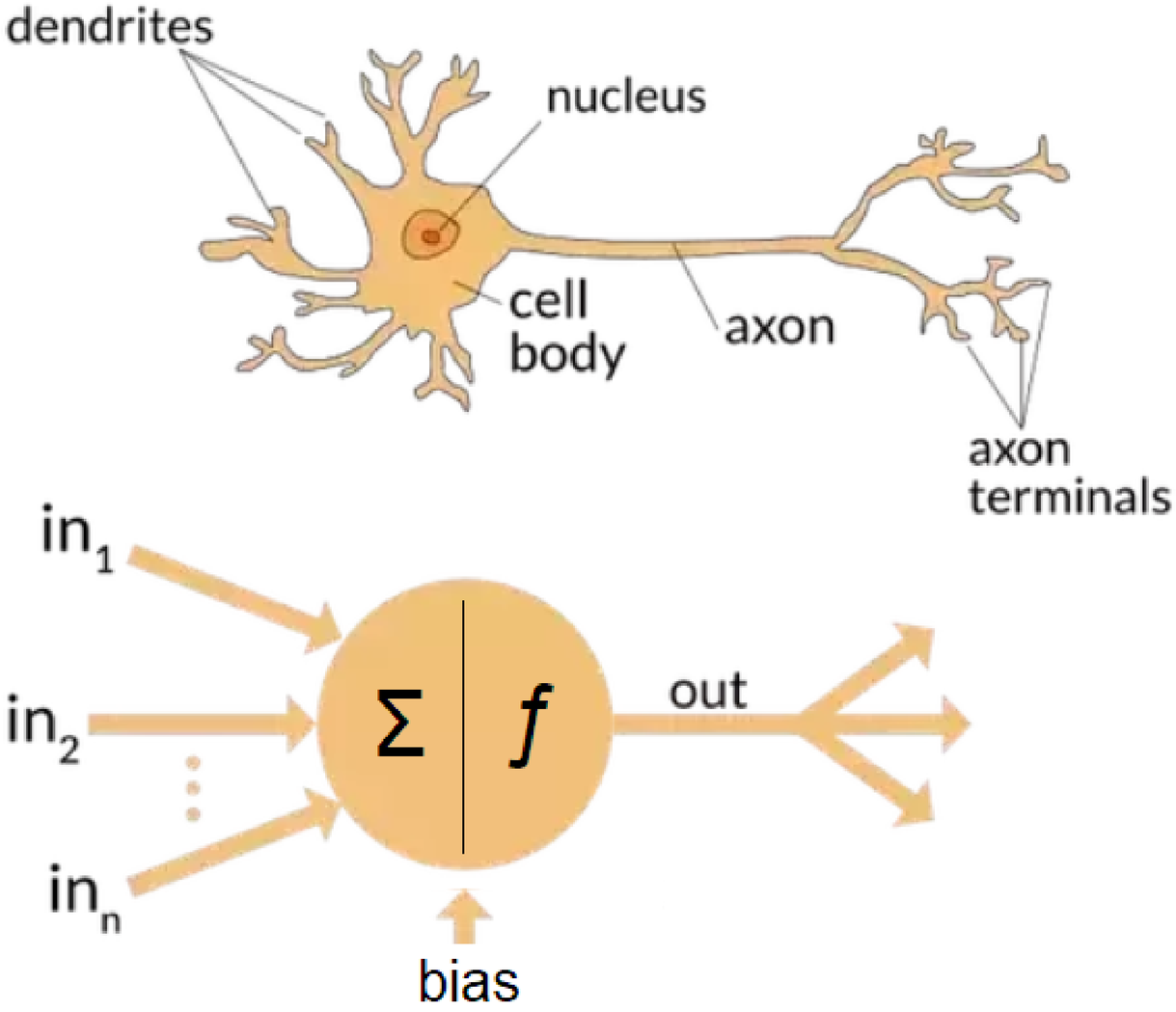
From Wikipedia:

- free and open-source software library for machine learning and artificial intelligence.
- developed by the Google Brain team
- can be used across a range of tasks but has a particular focus on training and inference of **deep neural networks**
- usable in many different languages
  - primarily Python
  - has bindings for other mainstream languages 
- TF 2.0 is tightly integrated with Keras
  - common to see code like

```
tf.keras.models.Sequential
```

  - high-level **neural networks** API, written in 🐍
  - supports rapid experimentation, prototyping, and has a developer-friendly API
    - 👎 comes up with the cost of losing access to the inner details of TensorFlow

Recall: An Artificial Neuron



Hello Tensorflow

```
import tensorflow as tf
import numpy as np
from tensorflow import keras

training_data = np.array([
    [ 1,  3],
    [ 2,  5],
    [ 3,  7],
    [ 4,  9],
    [ 5, 11],
    [ 7, 15],
    [ 8, 17],
    [ 9, 19],
    [11, 23]
])
```

- If you observe carefully:
  - the relationship between columns 1 and 2 is $2x + 1$
- Can a machine possibly figure out this relationship?
  - nowhere in our code is this relationship mentioned

```
# separate matrix into input (x) and output (y)
x = training_data[:, [0]]
y = training_data[:, [1]]

# just one unit, an output neuron and one input
model = keras.Sequential([
    keras.layers.Dense(units=1, input_shape=[1])
])
model.compile(
    loss=keras.losses.MeanSquaredError(),
    optimizer=keras.optimizers.SGD(learning_rate=.02)
)

# train our model
model.fit(x, y, epochs=500)

# test with values it's never seen before
model.predict(np.array([
    [13],
    [-5]
]))

# produces the following.  that's pretty close!!
# array([[27.00273 ],
#        [-9.005974]], dtype=float32)
```

NumPy Array, Numpy Slice …and…panic 🤯🤯

From the Keras docs:

> **The Sequential model**, which is very straightforward (a simple *sequential* list of layers), but is limited to single-input, single-output stacks of layers (as the name gives away).

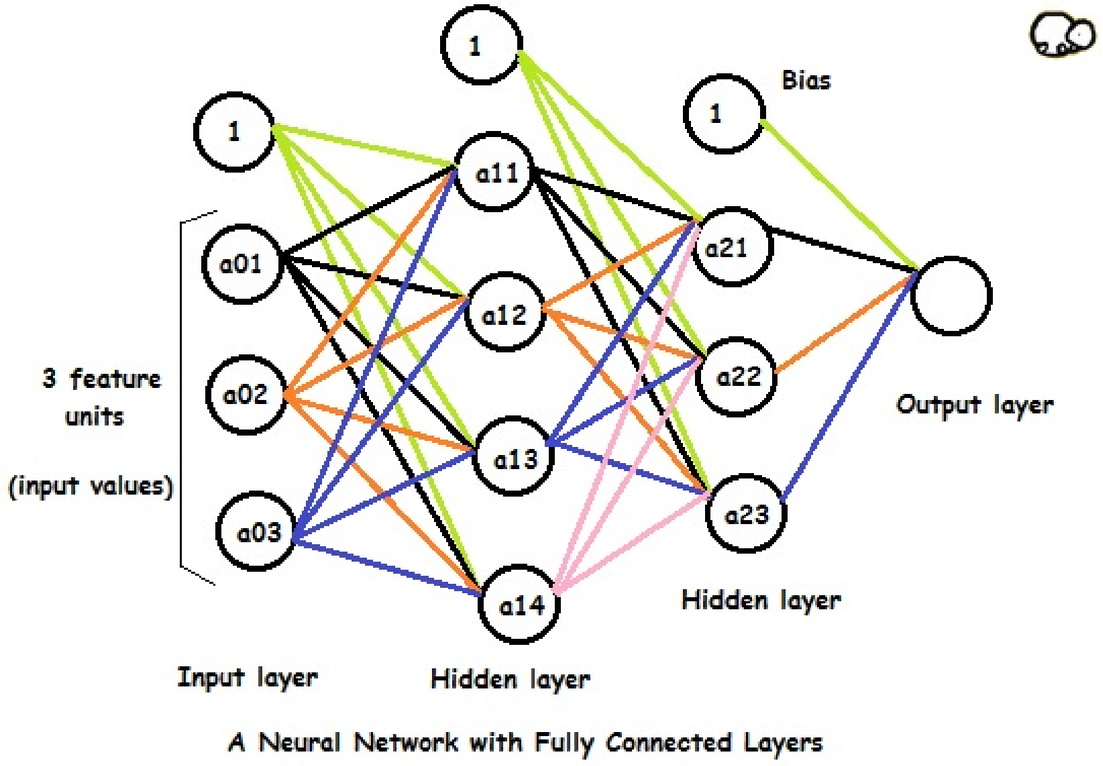
There are two other models, which is out of scope for the workshop.

3 ways to
create a Machine Learning model
with Keras and TensorFlow 2.0

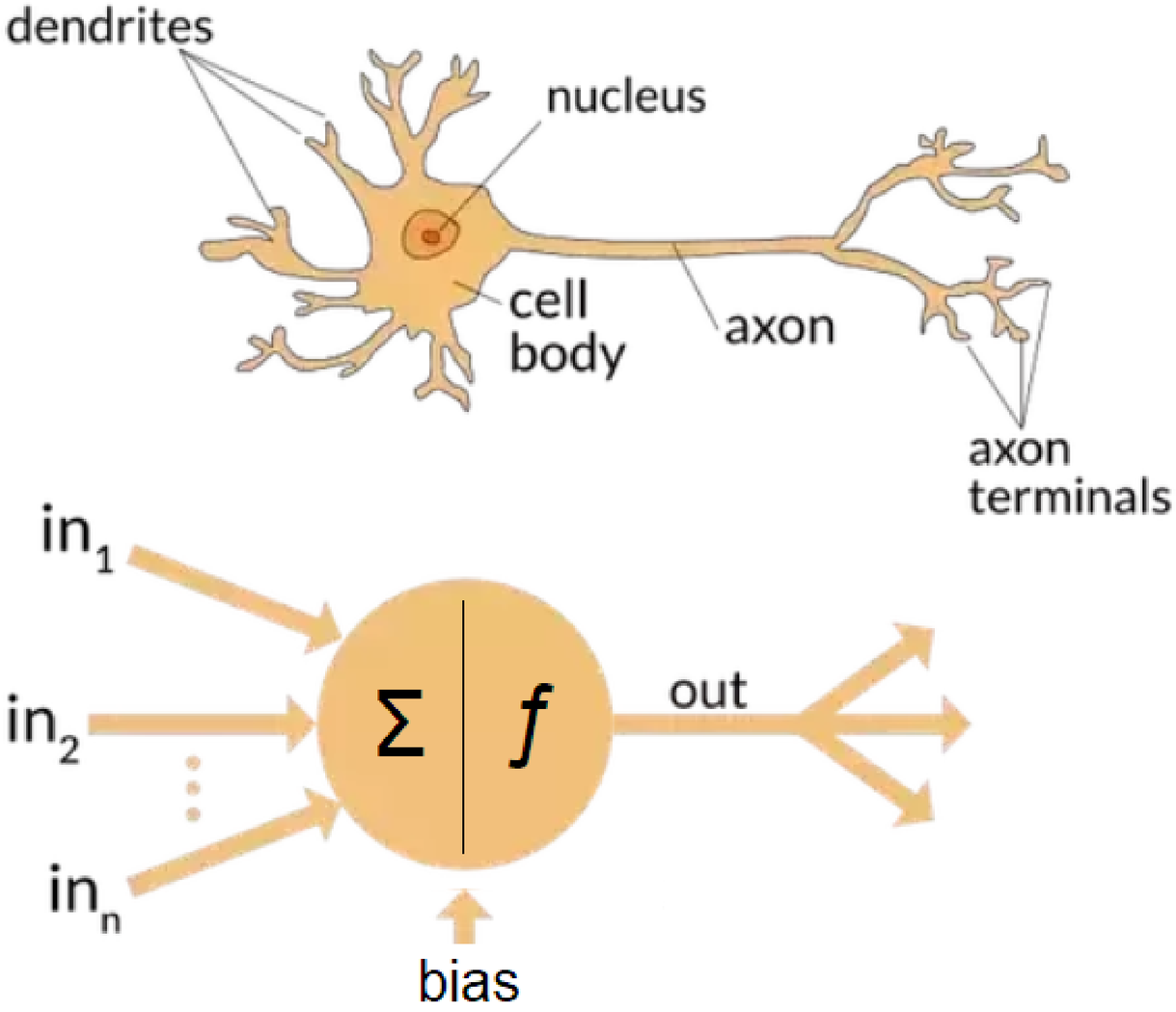
1. Sequential Model
2. Functional API
3. Model Subclassing

From Wikipedia:

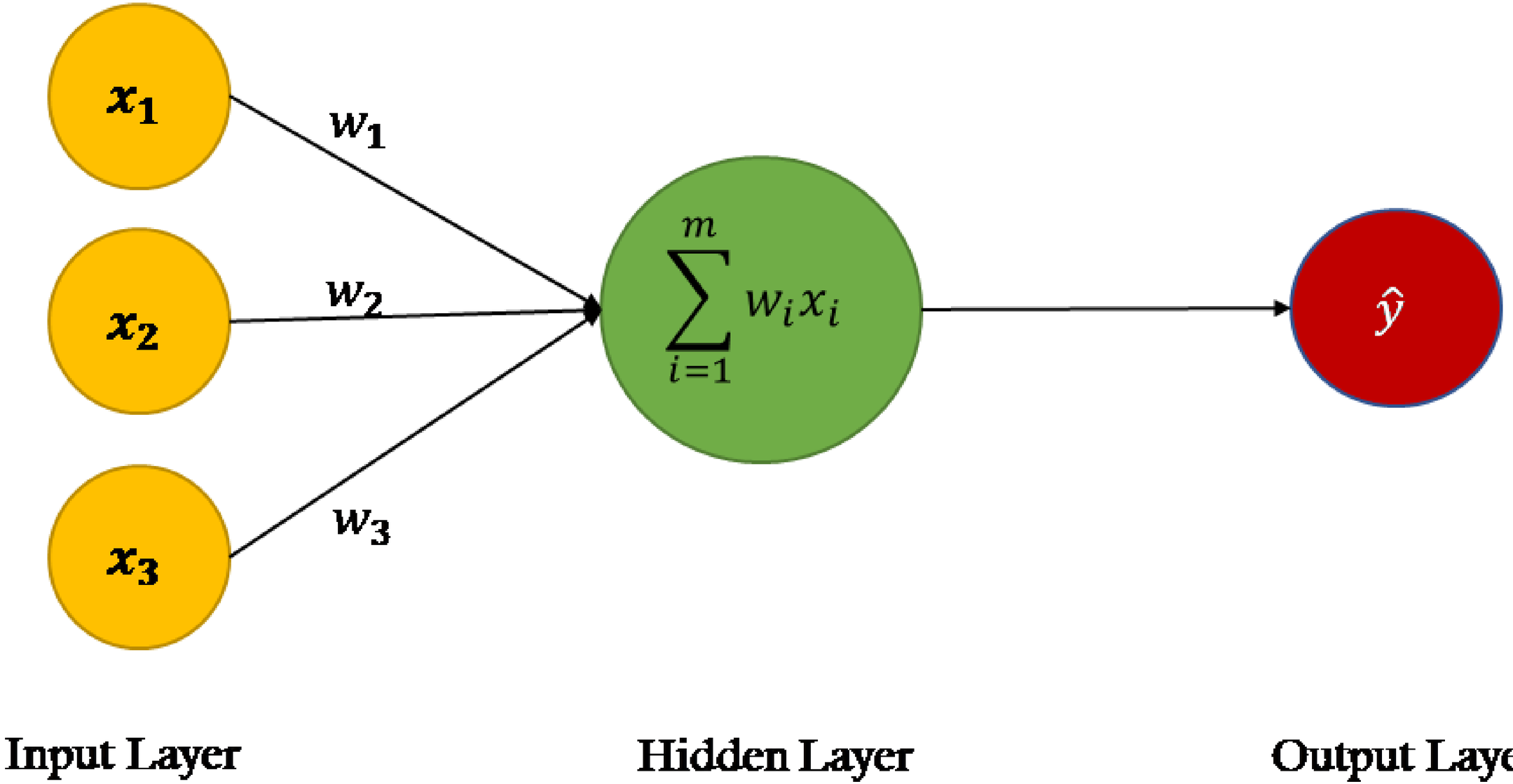
> **Dense layer**, also called ***fully-connected layer***, refers to the layer whose inside neurons **connect to every neuron** in the preceding layer.



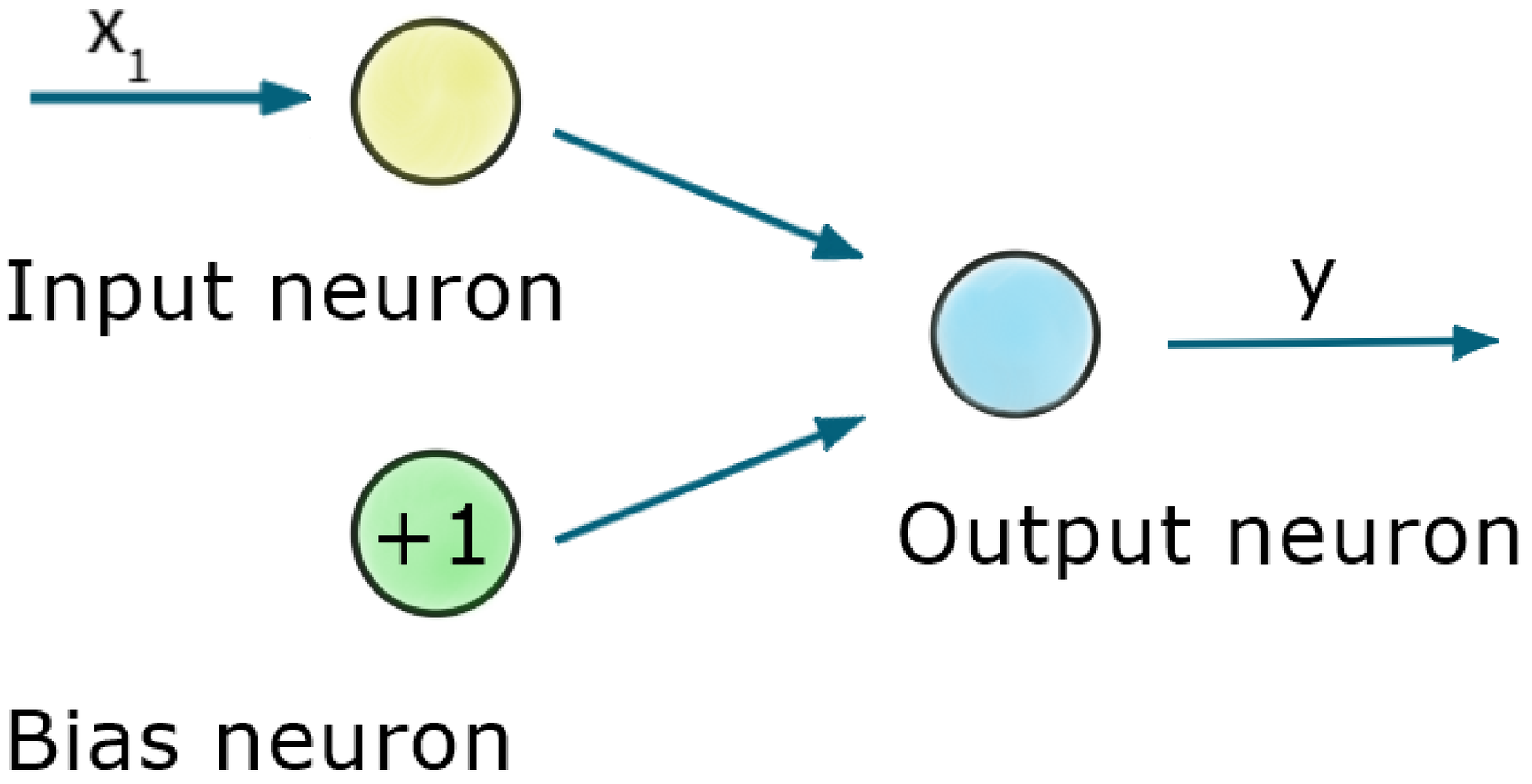
Recall: An Artificial Neuron



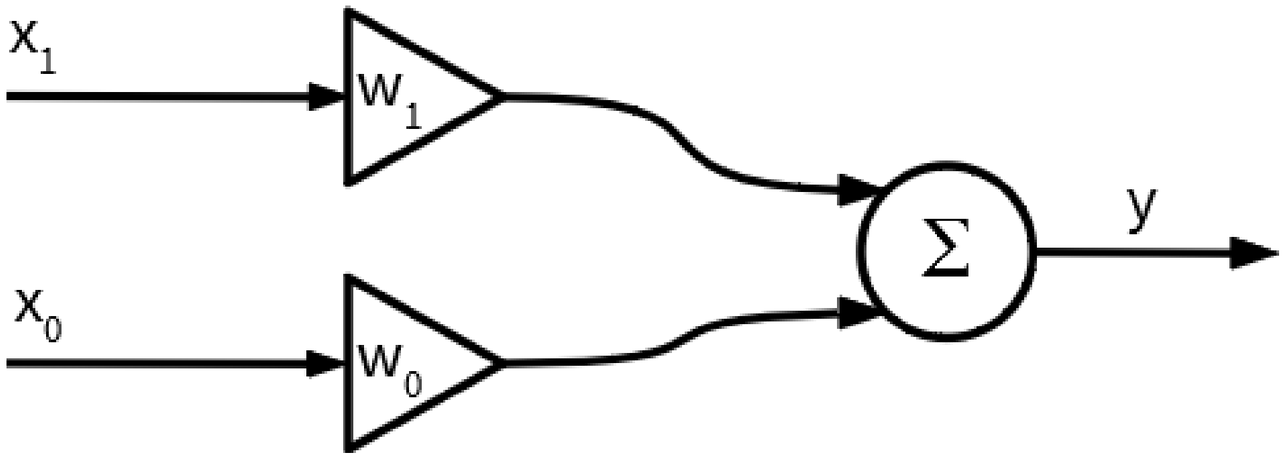
The Power of a Single Neuron



The Power of a Single Neuron



The Power of a Single Neuron



let's represent bias as x_0, and **let x_0 be always equal to** 1, now our output neuron is able to compute

$$y = \sum_{i=0}^{1} w_i x_i$$

$$y = w_1 x_1 + w_0 x_0$$

$$y = w_1 x_1 + w_0 \cdot 1$$

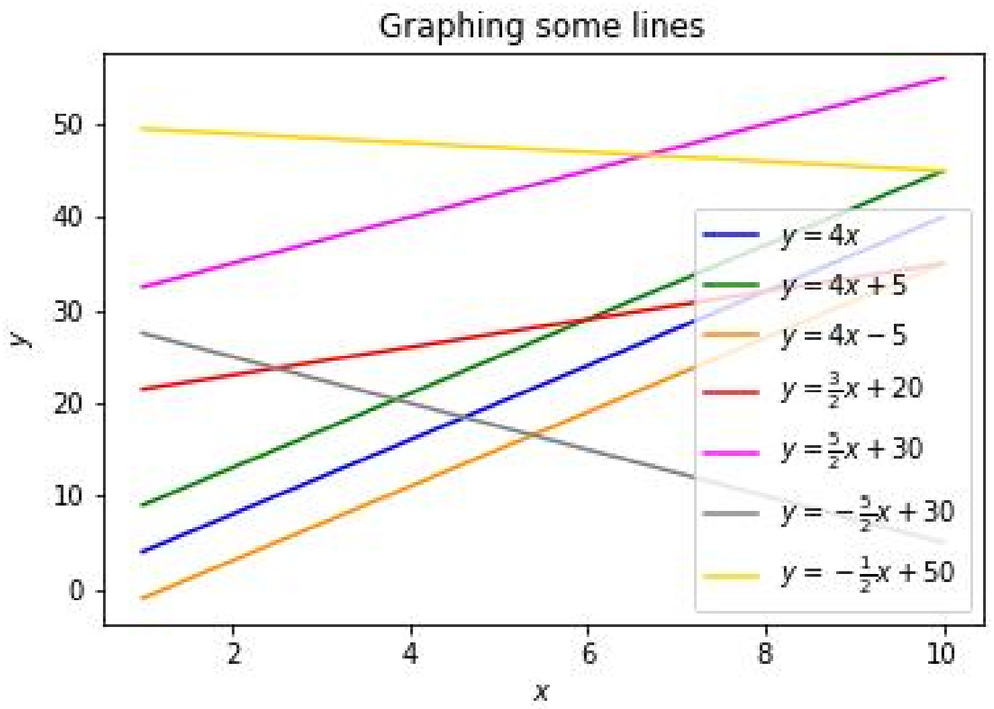
$y = w_1 x_1 + w_0$ 👉👉 *What does this equation remind you of?*

$$y = mx + b$$

Take a closer look

$$y = w_1 x_1 + w_0$$

Our single neuron can approximate a **line**ar function!

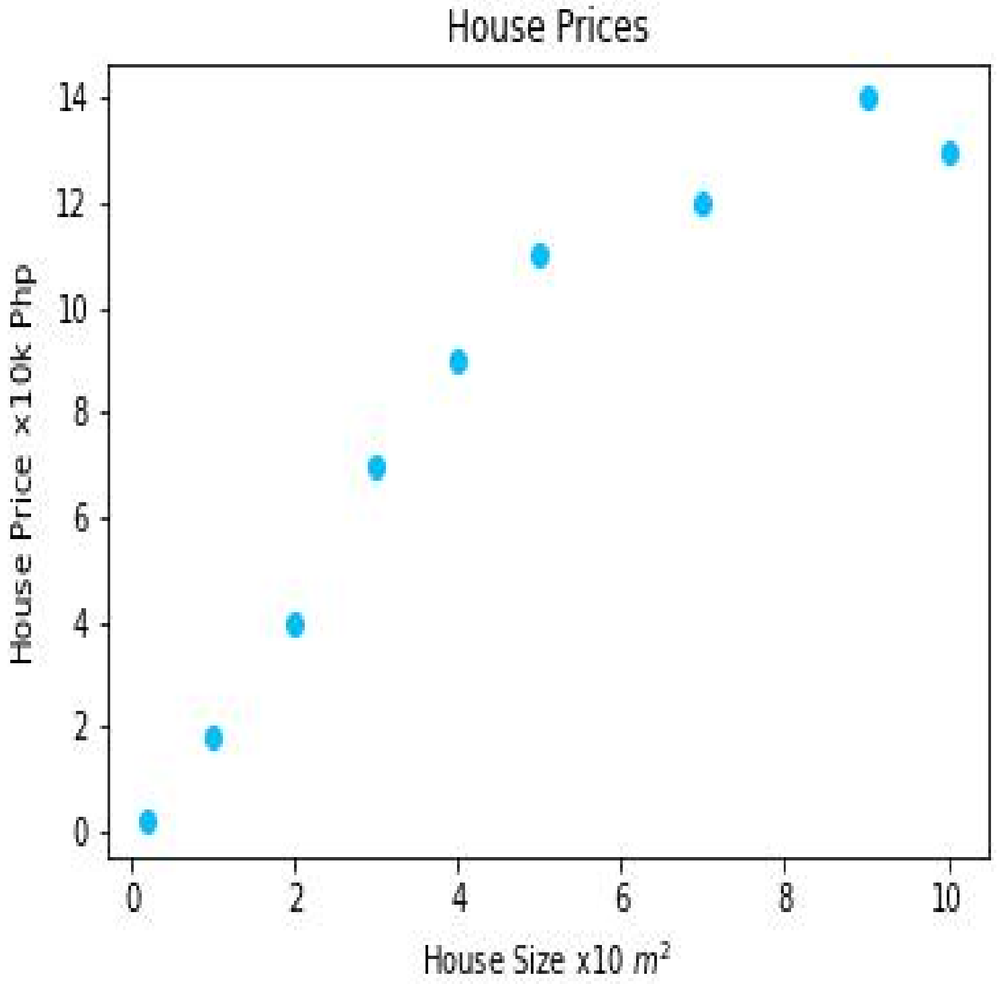


Predicting House Prices: A great idea 💡

1. start with random values for w_0 and w_1
2. determine how wrong the guessed w_0 and w_1 are
   - predict the house price $\hat{y}$ with the equation containing guessed w_0 and w_1
   - $\hat{y} = w_1 x_1 + w_0$, where x_1 is the house size
   - compare how far is the predicted value $\hat{y}$ with the **actual** house price y
   - keep doing this for all data in the training set `#1`, `#2`, `#3`, `#4`, etc…
     - $\hat{y}^{(1)} = w_1 x_1^{(1)} + w_0$, then compare how far is $\hat{y}^{(1)}$ to $y^{(1)}$
     - $\hat{y}^{(2)} = w_1 x_1^{(2)} + w_0$, then compare how far is $\hat{y}^{(2)}$ to $y^{(2)}$
     - $\hat{y}^{(3)} = w_1 x_1^{(3)} + w_0$, then compare how far is $\hat{y}^{(3)}$ to $y^{(3)}$
     - $\hat{y}^{(4)} = w_1 x_1^{(4)} + w_0$, then compare how far is $\hat{y}^{(4)}$ to $y^{(4)}$
   - average those error values
3. make another guess, repeat Step 2
   - 🙏 hopefully with some ~~magic~~ 🌈✨ **math** the new w_0 and w_1 become better guesses

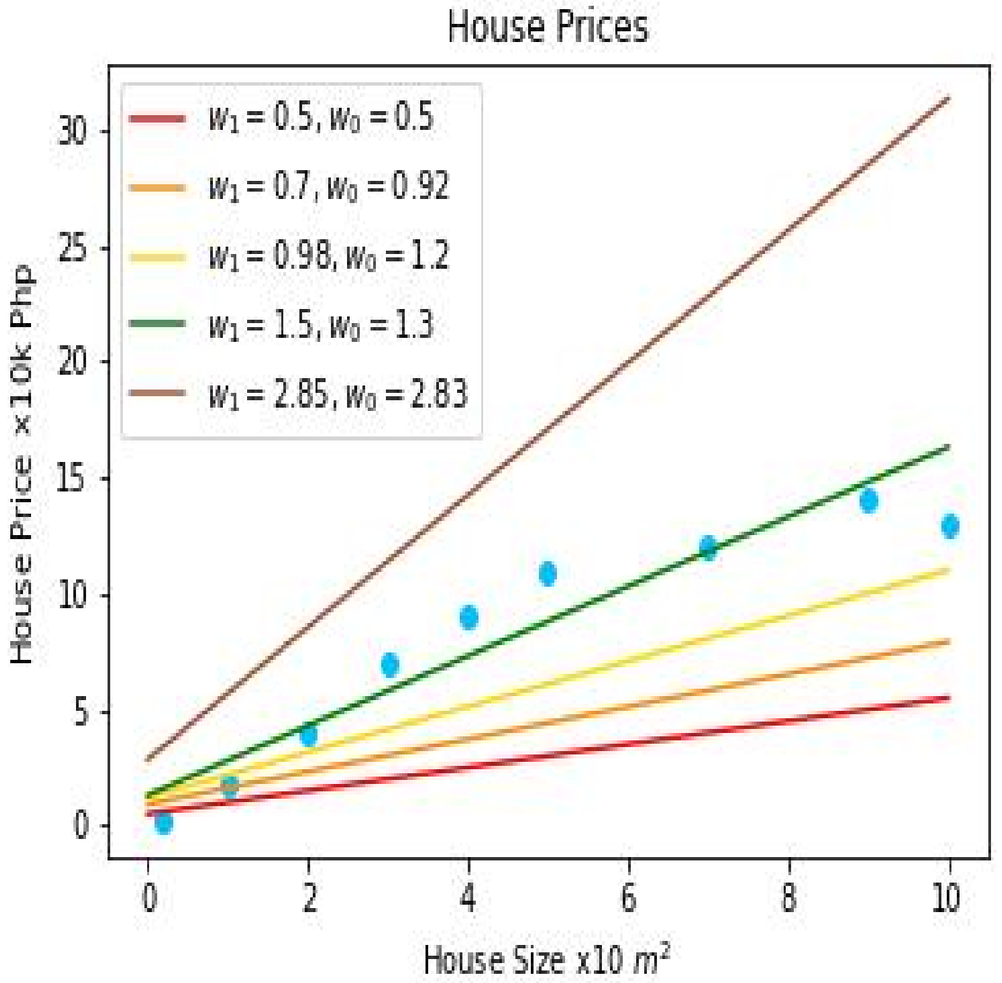
The idea: in visual form

Size in m^2	Price in ₱ ×10k
0.2	0.2
1	1.8
2	4
3	7
5	11
7	12
10	13
9	14



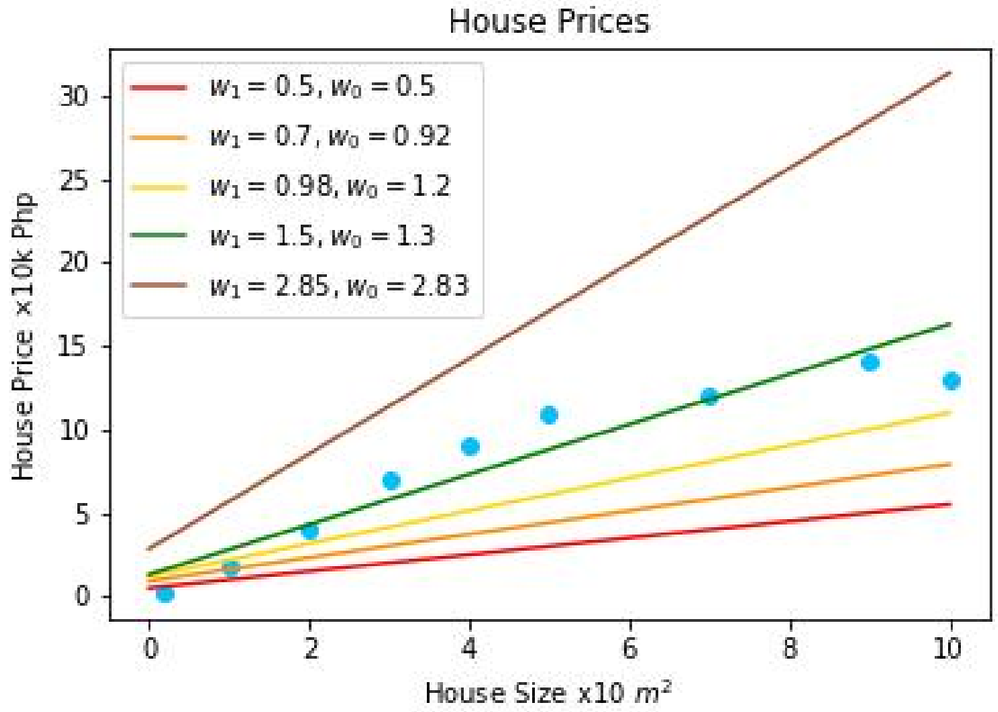
The idea: in visual form

Size in m^2	Price in ₱ ×10k
0.2	0.2
1	1.8
2	4
3	7
5	11
7	12
10	13
9	14



Loss functions

- `Simplest definition 👉` **Loss functions** measure how far an estimated value is from its true value.
  - also called **cost function**, or sometimes even **error function**
- What function can we use to determine how bad our predictions $\hat{y}$ are?
  - absolute value any good?



Mean-square error (MSE)

```
loss=keras.losses.MeanSquaredError()
```

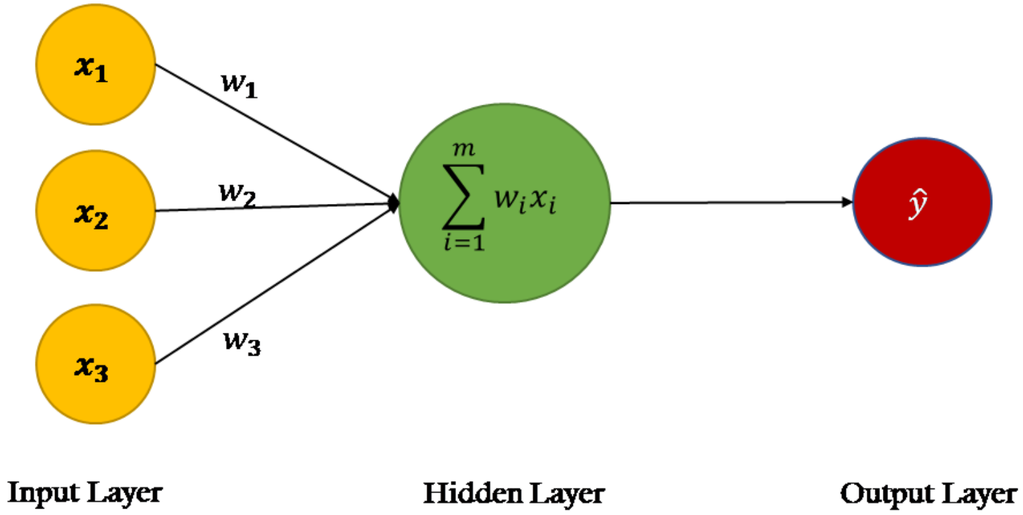
- Squaring values also make them positive, and **penalizes bigger errors.**

  $MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$ where $\hat{y}_i = w_1 x_{1i} + w_0$

- Some mathematicians *"halve"* this mean, which is still proportional for a very cool reason 😉 $\frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$

y	$\hat{y}$	$\lvert y - \hat{y} \rvert$	$(y - \hat{y})^2$
4	4	0	0
4	3	1	1
4	-3	1	1
4	7	3	9
4	-8	4	16

Unvectorized?



- also notice that:

$$\sum_{i=0}^{3} w_i x_i = w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ x_0^{(3)} & x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ \vdots & \vdots & \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} & x_2^{(m)} & x_3^{(m)} \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$\hat{y} = Xw$$

- and finally $\frac{1}{2m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$
  - is vectorized as: $\frac{1}{2m} (\hat{y} - y)^T (\hat{y} - y)$
  - also equivalent to: $\frac{1}{2m} (Xw - y)^T (Xw - y)$

Vectorized sum of squares

If the last slide is 😵‍💫🌠🥲, they're the same!

- Sum of squares, unvectorized

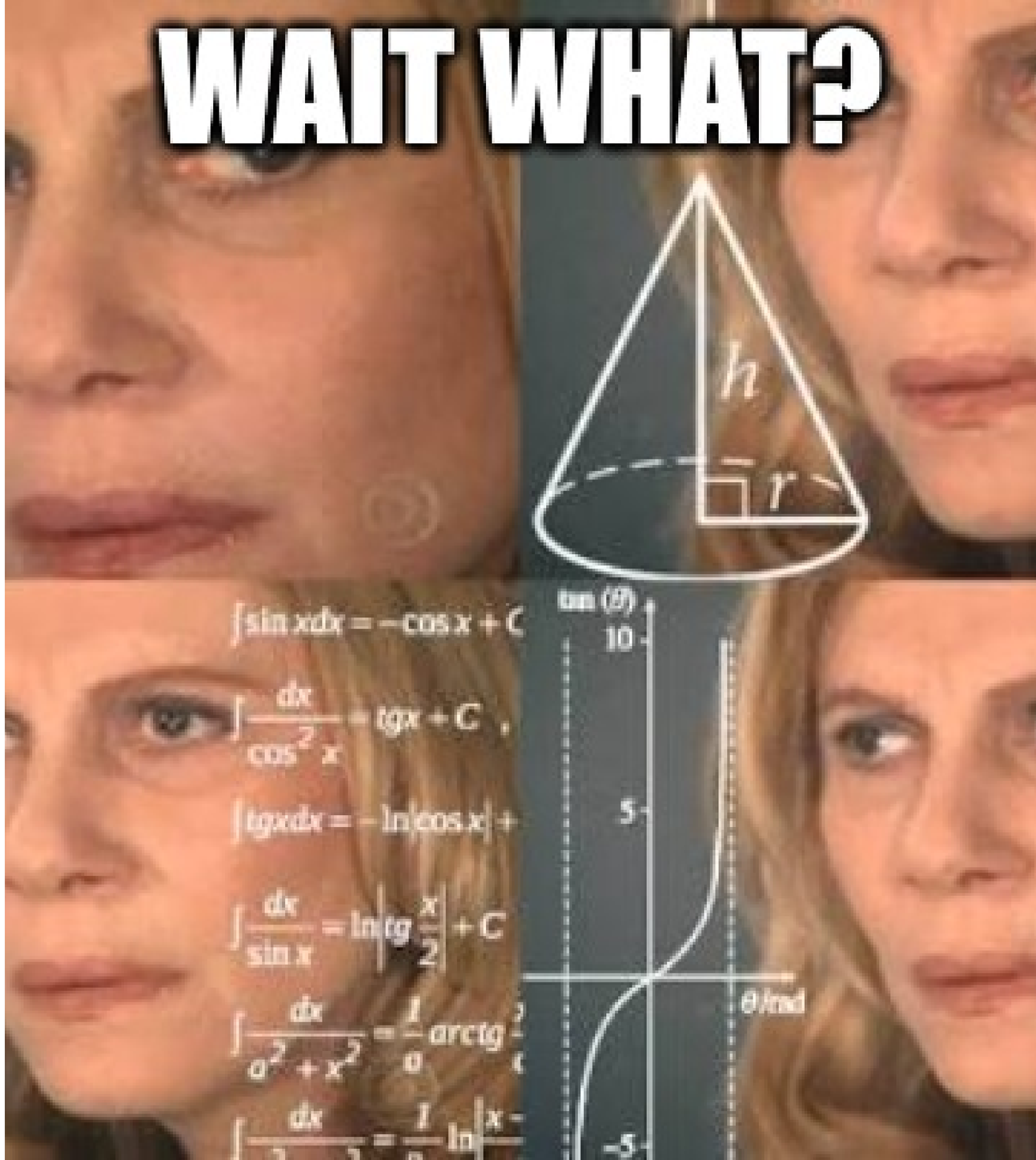
$$\sum_{i=1}^{4} i^2 = 1^2 + 2^2 + 3^2 + 4^2 = 30$$

- Sum of squares of all elements in vector v

$$= v^T v$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$= 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 + 4 \cdot 4 = 30$$



Unvectorized 🐢

$$Cost(w_1, w_0) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

where $\hat{y} = w_1 x_{1i} + w_0$

```
1   def cost(x1, y, w1, w0)
2       total = 0
3       m = len(x1)
4
5       for i in range(m):
6           y_hat_i = w1 * x1[i] + w0
7           total += (y_hat_i - y[i]) ** 2
8
9       return total * 0.5 * m
```

- sequential operations / single-threaded
- **will not scale**
- writing your own custom loss functions in 🔶 requires vectorization: ~~ ***have a "vectorized mentality"*** 😄

Vectorized 🐇

$$Cost(w) = \frac{1}{2m} (Xw - y)^T (Xw - y)$$

```
1   def cost(X, y, w):
2       m = X.shape[0]
3       return 0.5 * m * (X @ w - y).T @ (X @ w - y)
```

- parallel operations
- `X` is a matrix, `y` and `w` are vectors
- **BONUS**: more succinct code
  - almost looks like the equation
- BTW: in Python
  - `@` - **THE** matrix multiplication (*usually* ✓)
  - `*` - element-wise multiplication (*usually* ✗)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw & bx \\ cy & dz \end{bmatrix}$$

Congratulations on on knowing your first loss function!!

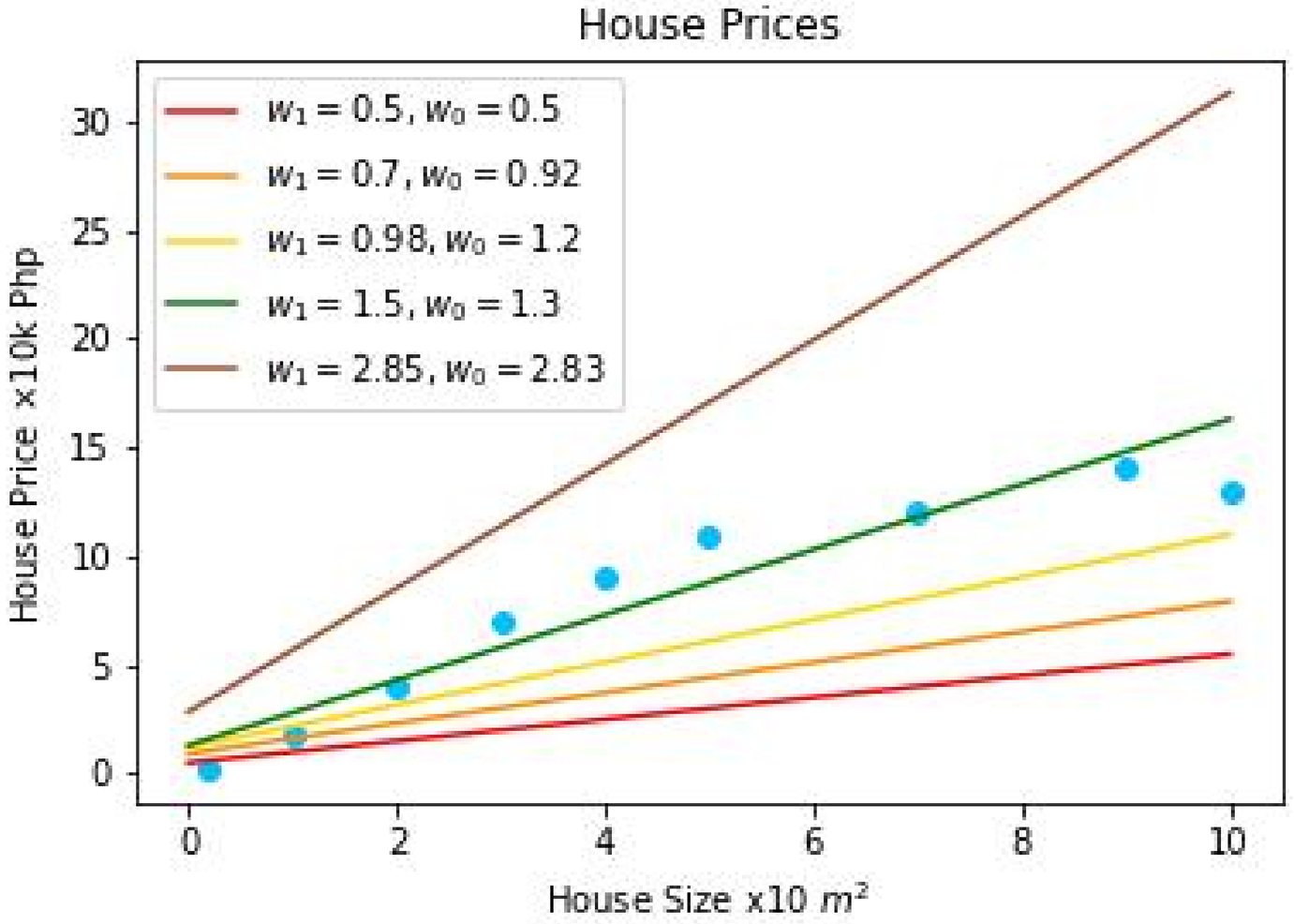
Remember its name: Mean Squared Error

Also remember and UNDERSTAND the how and why.

Even if you can't remember either formula, TF / Keras will remember it for you.

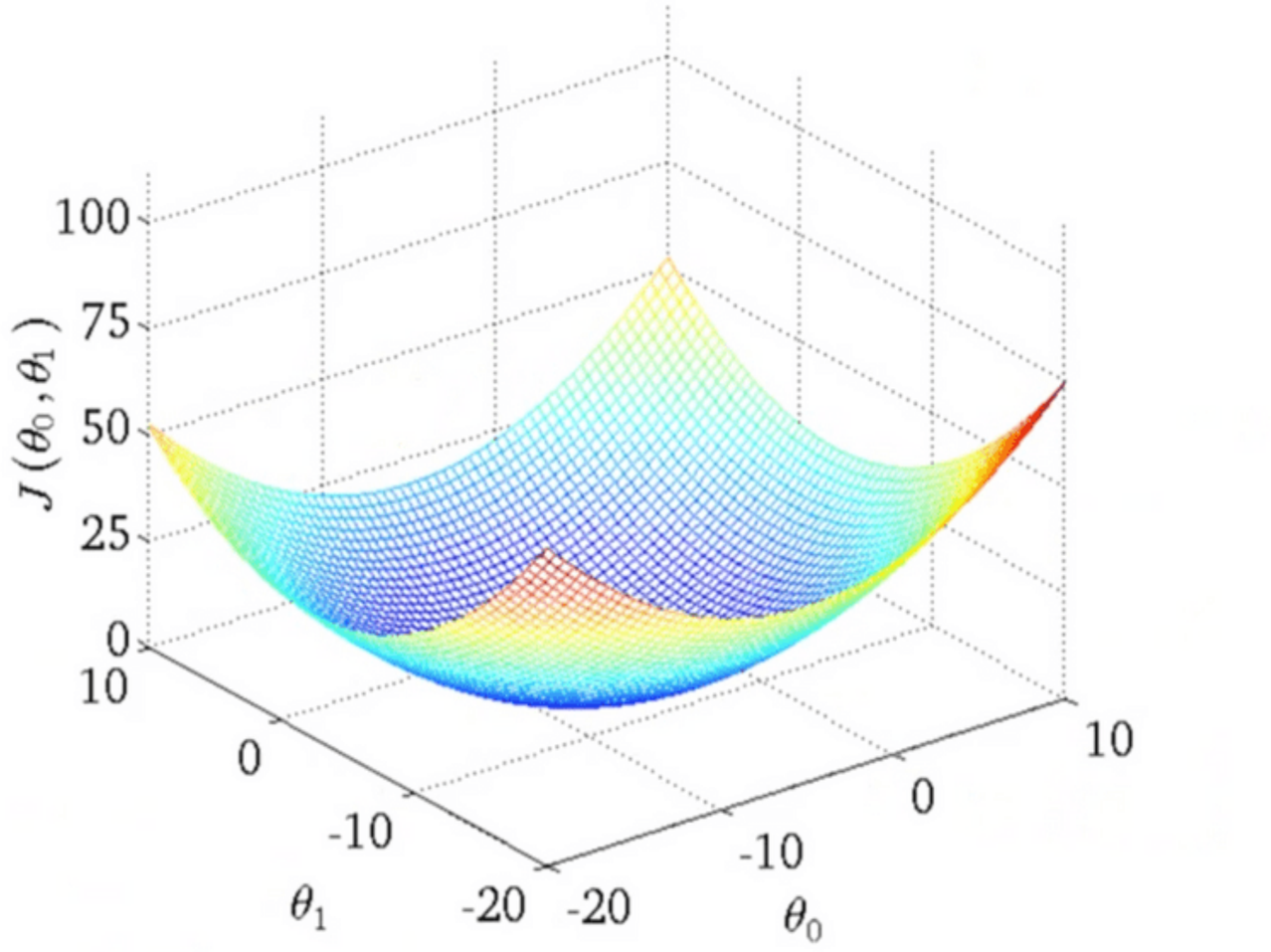
Unvectorized: $\frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$ Vectorized: $\frac{1}{2m}(Xw - y)^T(Xw - y)$

💡 Remember the idea: we want the best fit line



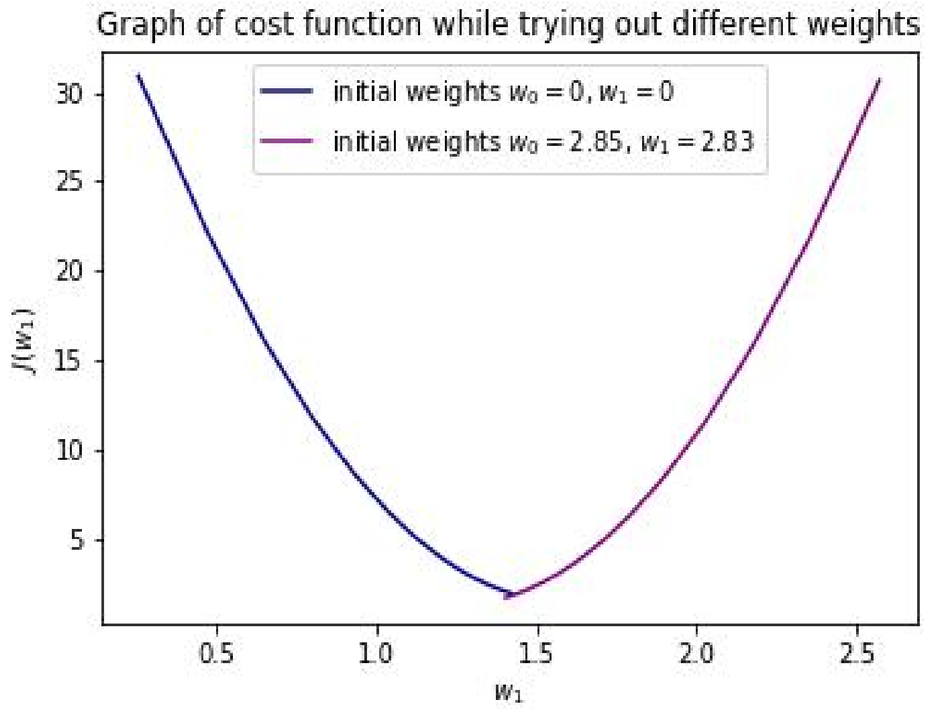
Loss function plot with 2 weights

w_0 and w_1 are the same as θ_0 ad θ_1, the graph will ***look something like this***



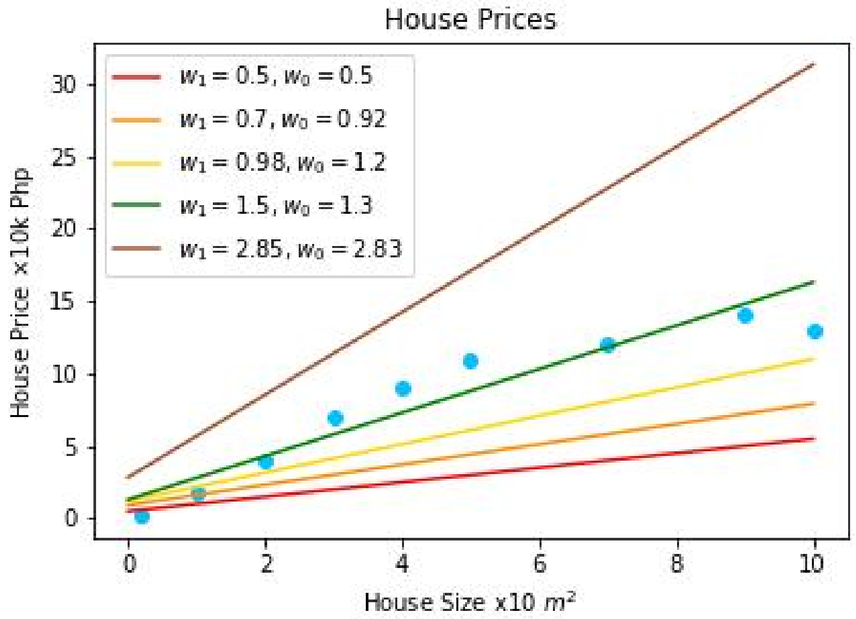
Graph of the loss function

Paraboloid? let's remove θ_0 for a simpler 2D parabola



- What do you see?
  - The lowest cost is when w_1 is around **1.5**
  - $w_1 = 1.5, w_0 = 1.3$ seems to be a good guess

w_1	w_0	MSE: $\hat{y} = (Xw - y)^T(Xw - y)$
0.5	0.5	18.864
0.7	0.92	7.079
0.98	1.2	2.542
1.5	1.3	1.397



Gradient Descent

or *"How to get the weights with the lowest cost?"*

- Parabolas have both constantly changing slopes and a ***minimum*** point.
- The ***minimum*** point occurs where the **slope is zero**
  - that’s also the point where the slope begins to increase on the other side
- the ***minimum*** occurs at the point where the **derivative** of the equation **is equal to zero**

How does it work?

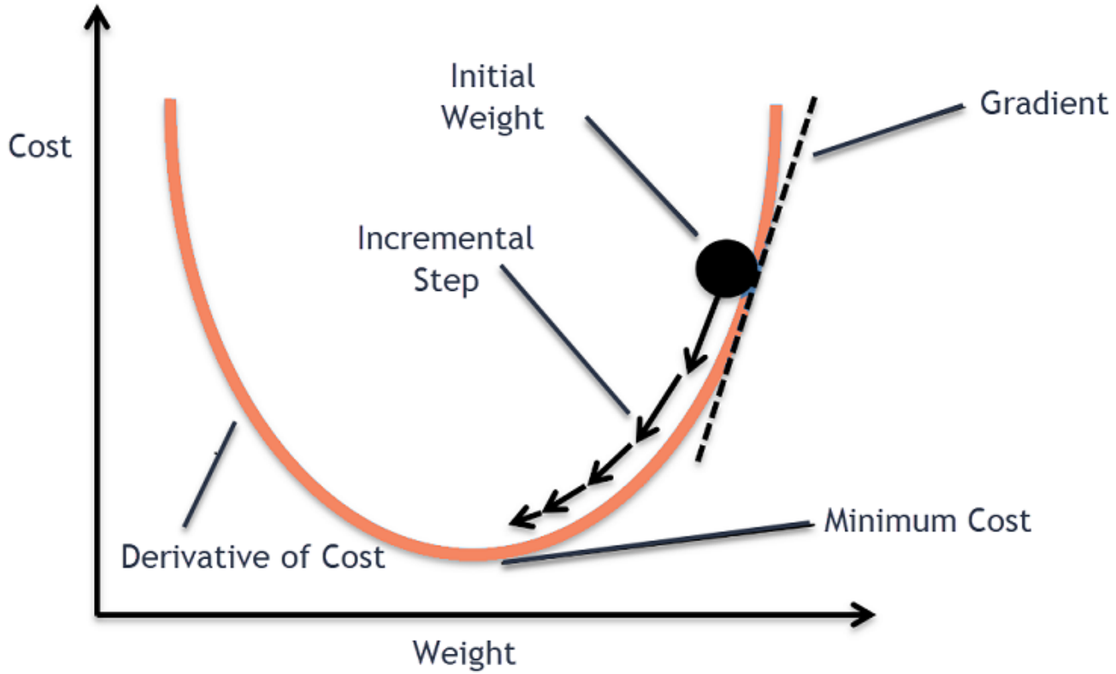
1. Calculates the partial derivative of each variable at the point in question
2. Combines the partial derivatives with respect to each variable to identify the direction of most negative slope across the vector space
3. Steps in the direction of the most negative slope
4. Repeats the process until it finds the minimum point.

Gradient Descent

- goal is to **minimize** the loss function $f(w)$:

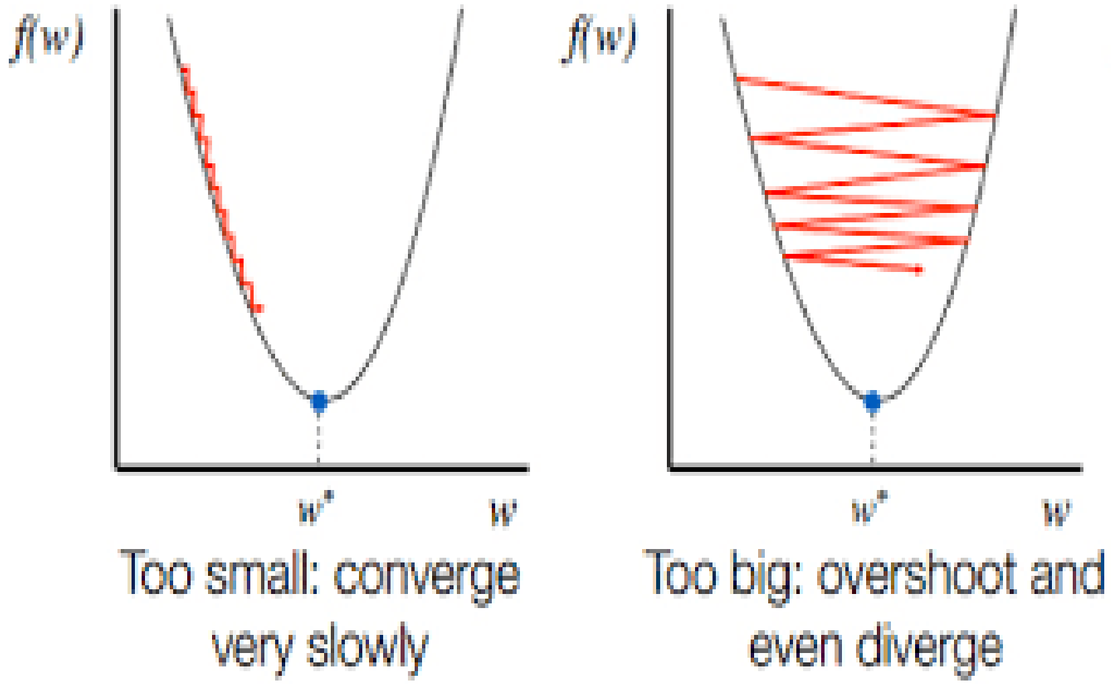
`start with a "random-ish" ` w `` vector`
`repeat for several iterations:`

$$w = w - \alpha \frac{\partial}{\partial w} f(w)$$



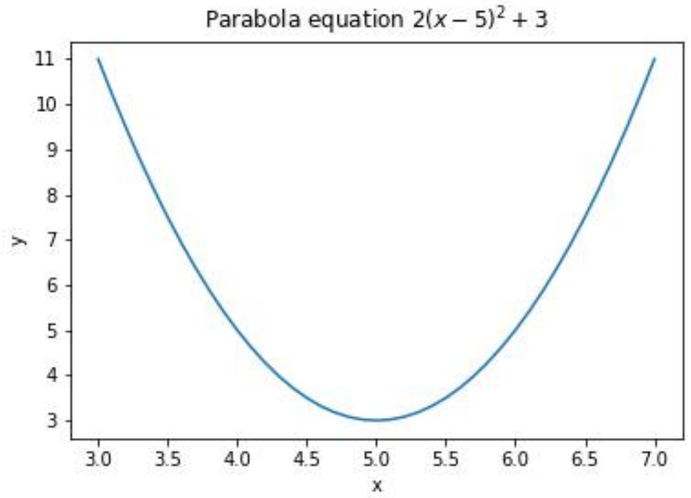
Learning rate

- α stands for the **learning rate** (a.k.a. ***step size***)
  - too big and you'll overshoot the minimum
  - too small and it'll slow down your model's training
  - α has to be *"just right"*, i.e. find the *"sweet spot"*



Gradient Descent Code

- Let’s try it on a parabola
  - Say the equation is $y = 2(x-5)^2 + 3$
  - Derivative is: $\frac{dy}{dx} = 4(x-5)$



  - We can see that when x is 5, y is minimized
  - And true, setting $\frac{dy}{dx} = 0$, we get
    - $4(x-5) = 0$; thus x should be 5

```python
def gradient_descent(initial_x):
    x = initial_x
    learning_rate = 0.1

    for i in range(1000):
        derivative = 4 * (x - 5)
        x -= learning_rate * derivative

    return x
```

Testing it on `initial_x = 0`, and also `10`, we get:

```python
gradient_descent(0) # returns 4.999999999999999
gradient_descent(10) # returns 5.000000000000001
```


Yay! Pretty close don’t you think?

Calculus makes 💻 machines learn from - and minimize - their *"errors"* ‼️

❓ For Php50 load: If we increase the iterations of the loop, will it go from the `minimum` and walk upwards the parabola? **WHY?**

Hyperparameters

- set / ***tuned*** by **us** - *the machine learning practitioners*
- parameters whose values control the learning process
- these values **cannot** be changed by the ML model
- we've already learned **two**
  - learning rate α

```
SGD(learning_rate=0.02)
```

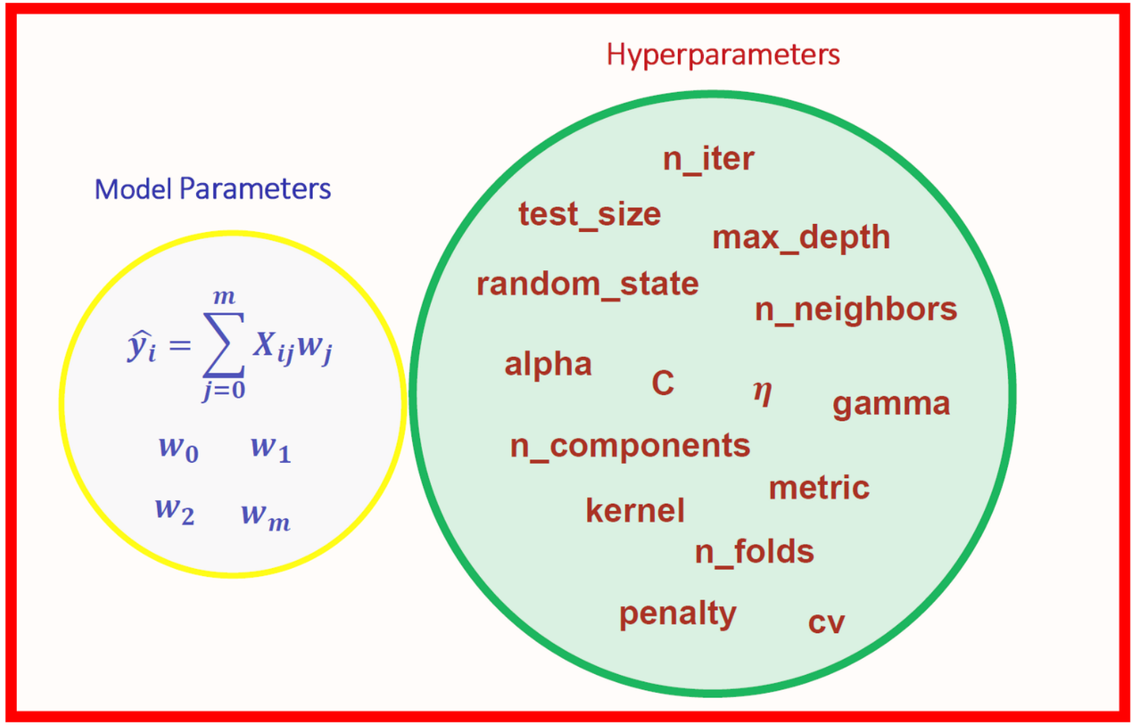
  - number of iterations (epochs)

```
model.fit(x, y, epochs=500)
```

- other hyperparams we'll learn today / tomorrow:
  - train-test split ratio
  - number of hidden layers in our NN
  - number of units (neurons) in each NN layer
  - filter size and pooling size in our CNNs

Parameters

- **learned** by the machine based on our training data
- the model can change these values
- example: the weights w of a neural network



Stochastic Gradient Descent (SGD)

```
optimizer=keras.optimizers.SGD(learning_rate=0.02)
```

- based on the assumption that the errors are additive.
  - The error at point one can be added to the error at point two, which can be added to the error at point three, etc.
- model doesn't need to calculate the error at all points simultaneously - ***a computationally expensive process***
- calculate the error at each point individually, and sum them together.
- this assumption is ***almost*** always true. ⚠️ *Still, it's best practice to ensure that it's true in your problem.*

TL:DR -- This saves dramatic amounts of computation time.

> 👎 The downside of stochastic gradient descent is that it doesn't find the true minimum as reliably. Instead it has a tendency to get extremely close, then circle the minimum forever. It basically sacrifices some accuracy for speed.

 Tensorflow only has SGD, and doesn't have BGD built-in *(many beginner tutorials teach the simpler BGD)*.

> **Batch gradient descent** (BGD) is a type of gradient descent which ***processes all the training examples*** for each iteration of gradient descent. But if the number of training examples is large, then **batch gradient descent is computationally very expensive.**

Congratulations on on knowing your first optimizer!!

Remember its acronym: SGD

Also remember and UNDERSTAND the how and why.

You don't even have to know the partial derivatives of loss functions like MSE

If interested, it's $\frac{\partial}{\partial w} J(w) = \frac{1}{m} X^T (Xw - y)$ long derivation of unfactorized variant: $2X^T X\theta - 2X^T y$

👩‍💻 You do it

Size in m^2	Price in ₱ ×10k
0.2	0.2
1	1.8
2	4
3	7
5	11
7	12
10	13
9	14

🏃‍♂️🏃‍♀️ Ready ...

02:00 05:00 10:00 15:00

- replace the $y = 2x + 1$ training set with the *"house prices"* you see on the left
  - more like *"bahay-kubo"* prices with 10 m^2 as the largest
- predict the prices of house sizes it's never seen before:
  - try **8** m^2 and **12** m^2 for example
- **[Optional]**: Graph the line that best fits all data points.
  - 💡 Examine `model.weights` to get w_1 and w_0.
- **[Optional]**: Graph the costs, and see the curve flatten.
  - 💡 `history = model.fit( ... )` and graph `history.epoch` against `history.history['loss']`

Congratulations on your first machine learning algorithm!

Linear Regression

Specifically, univariate linear regression

All that with a single neuron in Tensorflow

Multivariate is ***almost*** the same

- to predict house price based on 3 **features**: the area (x_1), number of bedrooms (x_2), and age of the house (x_3)…
  - it's still the same formula: $\hat{y} = Xw$, but for 5 training samples, now our X and w might look like

$$\hat{y} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ x_0^{(3)} & x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ x_0^{(4)} & x_1^{(4)} & x_2^{(4)} & x_3^{(4)} \\ x_0^{(5)} & x_1^{(5)} & x_2^{(5)} & x_3^{(5)} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

- but this introduces a *"problem"* that the values are in different ranges
  - house size could range between 22 to 300 sq.m.
  - number of bedrooms could be 1 to 5
  - age could be 1 to 115 🐶🏠

Feature Scaling

Quoting from Analytics Vidhya:

> The presence of feature value X in the formula will affect the step size of the gradient descent. The difference in ranges of features will cause different step sizes for each feature. To ensure that the gradient descent **moves smoothly** towards the minima and that the steps for gradient descent are updated at the same rate for all the features, we scale the data before feeding it to the model.
>
> **Having features on a similar scale can help the gradient descent *converge more quickly towards the minima*.**

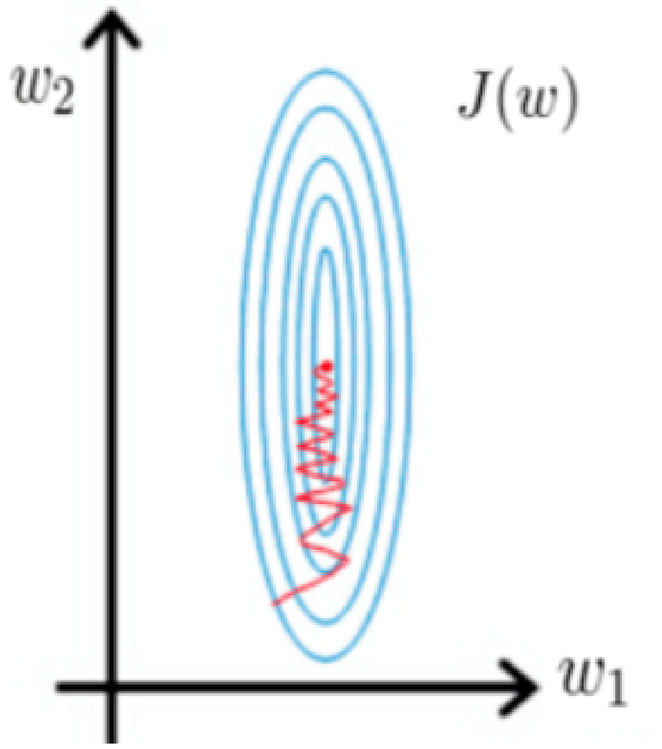
Quoting from the Tensorflow tutorial on Basic Regression:

> **It is good practice to normalize features that use different scales and ranges.**
>
> One reason this is important is because the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs.
>
> **Although a model might converge without feature normalization, normalization makes training much more stable.**

Feature Scaling

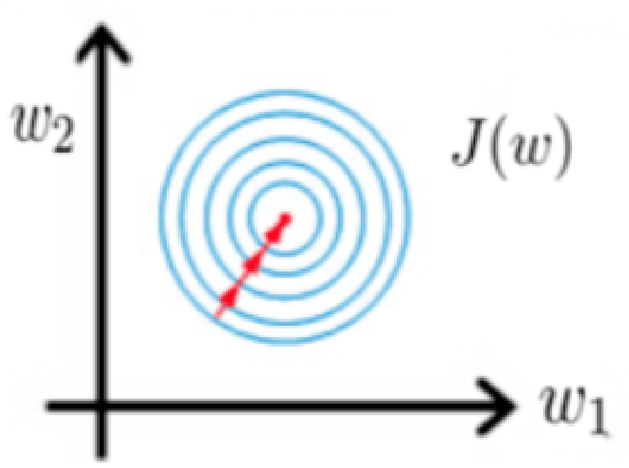
Gradient descent without scaling

$x_1 >> x_2$



Gradient descent after scaling variables

$0 \leq x_1 \leq 1$
$0 \leq x_2 \leq 1$



Feature Normalization

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as **Min-Max scaling.**

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

- the lowest value will produce $X_{min} - X_{min}$ in the numerator, and will become 0.
- the highest value will produce $\frac{X_{max} - X_{min}}{X_{max} - X_{min}}$ and will be scaled to 1.

```
# say our X matrix is:
X = np.array([
    [1000, 5, 70],
    [2000, 6, 90],
    [2500, 2, 95],
])
```

Using Scikit-learn: *ohnoes, another library?* 😅

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(X) # get min and max, store internally
scaler.transform(X) # scale!!
```

Using NumPy only and vectorization powers 💪💪:

```
# get min & max column-wise (axis=0 means X-axis)
X_min = np.min(X, axis=0) # array([1000, 2, 70])
X_max = np.max(X, axis=0) # array([2500, 6, 95])
X_norm = (X - X_min) / (X_max - X_min)
```

They'll produce the same matrix!

```
array([[0.   , 0.75 , 0.   ],
       [0.667, 1.   , 0.8  ],
       [1.   , 0.   , 1.   ]])
```

👩‍💻 Your turn! 😱 *already?* 🐎

I prepared a notebook that contains **156** ‼️ training data, just to spare the participants a lot of 🖐⌨🖐 pain.

The notebook’s link will be sent via chat, or just ⌨ ⌨ type it out:

https://bit.ly/ece4241-ml-01

- *If you have extra time, you can also experiment what happens if feature scaling is off.*

 Congratulations on knowing your first preprocessing task!

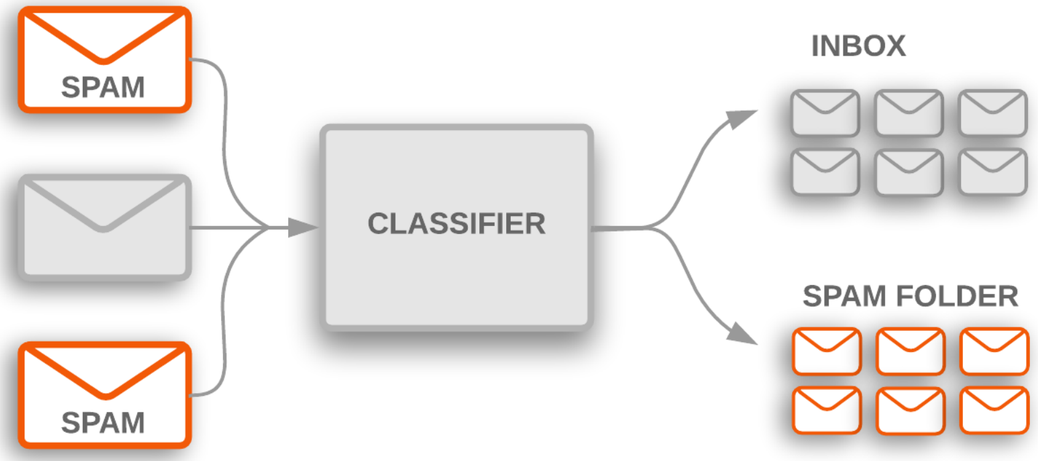
Feature Normalization

Also good job on enhancing linear regression with one more variable!

All that, STILL with a single neuron in Tensorflow

Classification

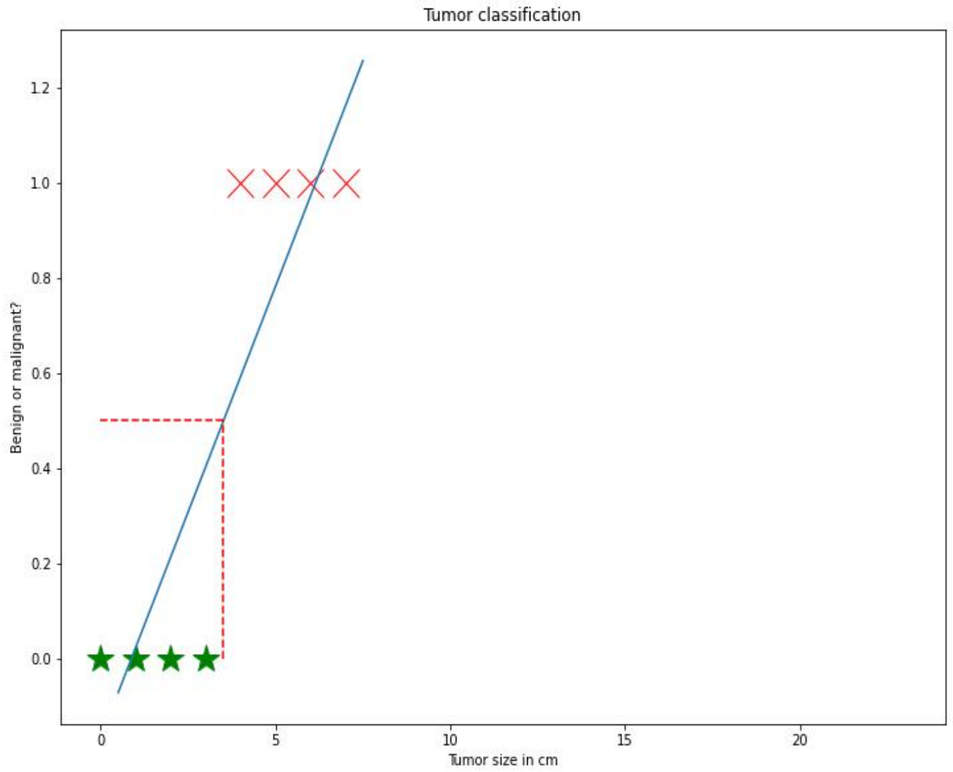
- Examples:
  - Email: **spam** or **not spam**
    - label $y = 1$ if spam, $y = 0$ if not spam
  - Transaction: **fraudulent** or **not fraudulent**
    - label $y = 1$ if fraudulent, $y = 0$ if not
  - Tumor: **malignant** or **benign** (not malignant)
    - label $y = 1$ if malignant, $y = 0$ if benign
- Basically:
  - $y = 1$ if data belongs to *"positive class"*
    - e.g. malignant, `<something>`
  - $y = 0$ if data belongs to *"negative class"*
    - e.g. NOT malignant, `NOT <something>`



- 😟😵 Don't be confused:
  - While *"fraudulent"*, *"spam"*, and *"malignant"* are NEGATIVE words in everyday use, in ML they belong to the ***positive class*** because their label is 1.
  - If we label *benign* with 1 instead, and *malignant* 0,
    - then `benign` belongs to the *positive class*
    - `NOT benign` belongs to the *negative class*
      - 👆 *"malignant"*

Can we use linear regression – for classification?

- Say this is our data for tumor size vs. malignant/benign
- line fits perfectly

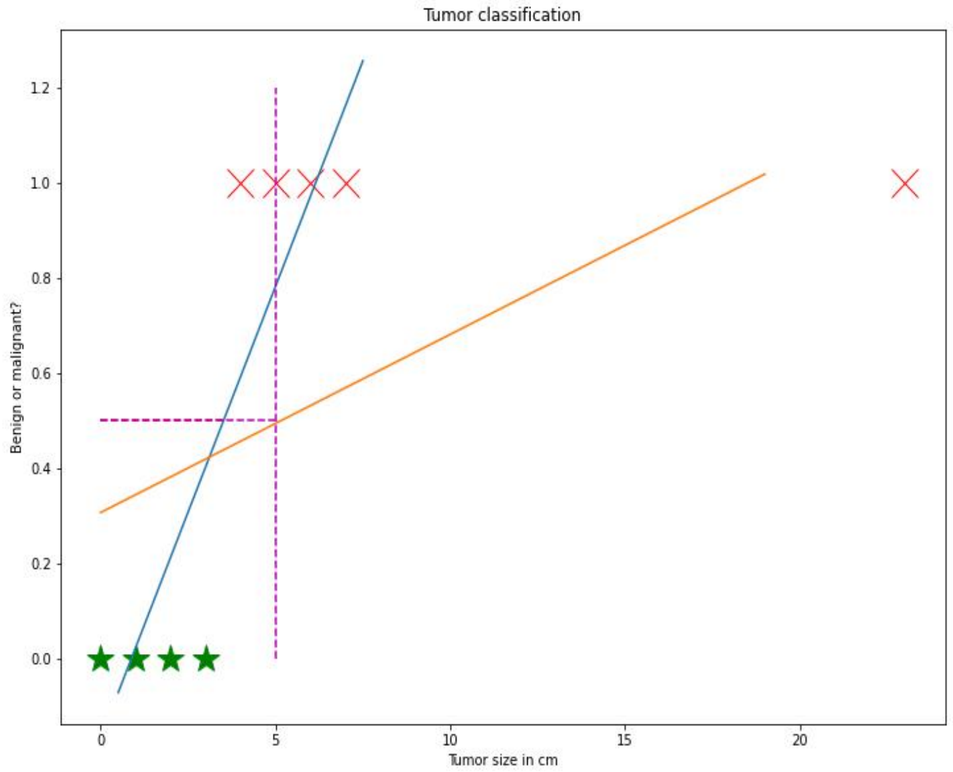


- predict `0 (benign)` if $y < 0.5$
- predict `1 (malignant)` if $y \geq 0.5$

Tumor Size	y	$y >= 0.5$?	Prediction
1.1	0.04	false	benign
3.4	0.48	false	benign
4.0	0.60	true	malignant
4.3	0.65	true	malignant

We can’t use linear regression for classification

- Now we add more training data, a tumor size of **23!!**



Tumor Size	y	$y >= 0.5$?	Prediction
1.1	0.36	false	benign
3.4	0.46	false	benign
4.0	0.48	false	**benign**
4.3	0.49	false	**benign**

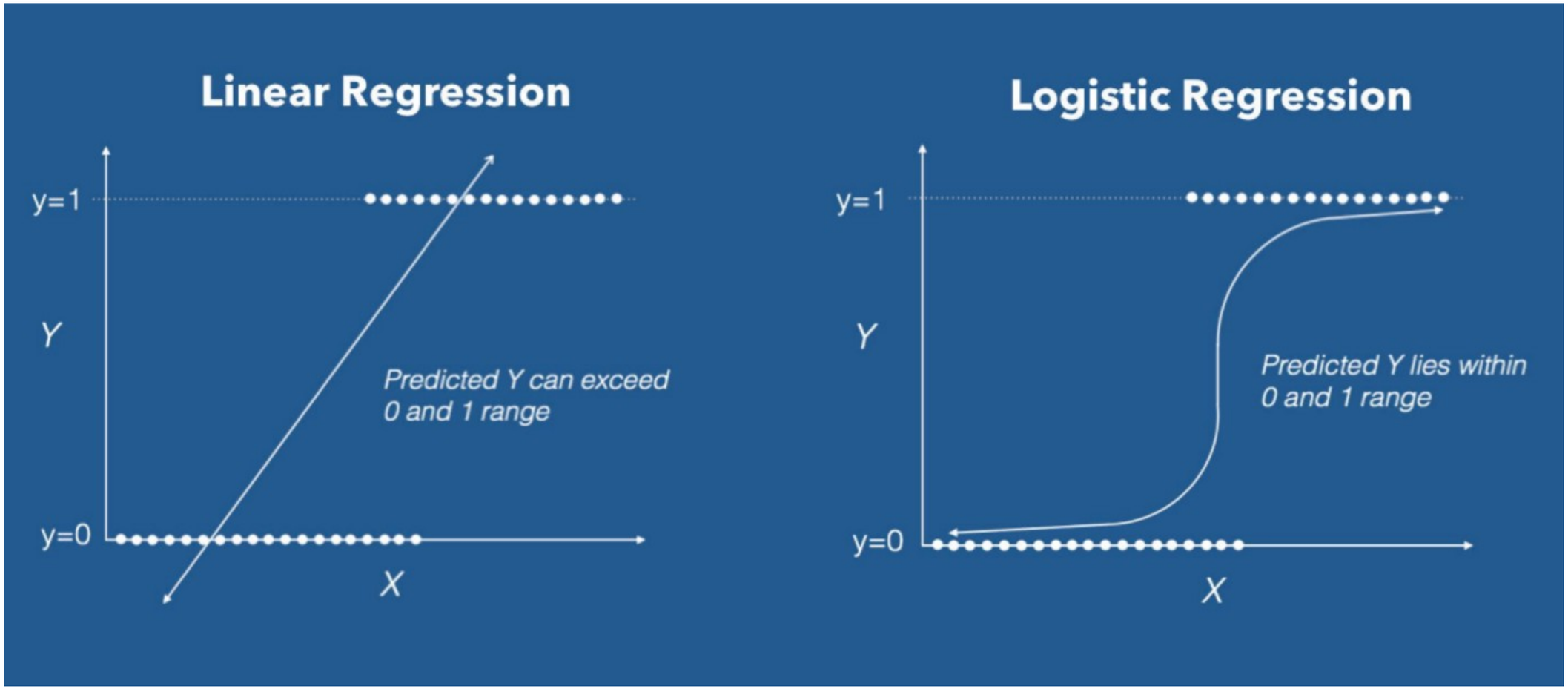
- what’s formerly classified as malignant is now benign!!

🤦 Reason #1: additional training data will shift the *"best-fit"* line

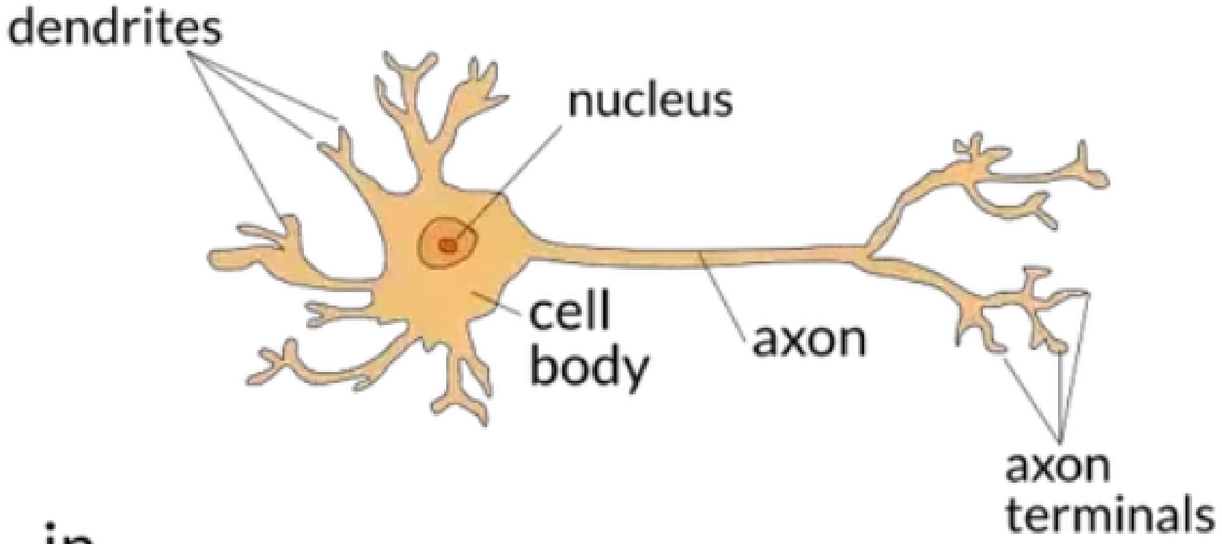
We can’t use linear regression for classification

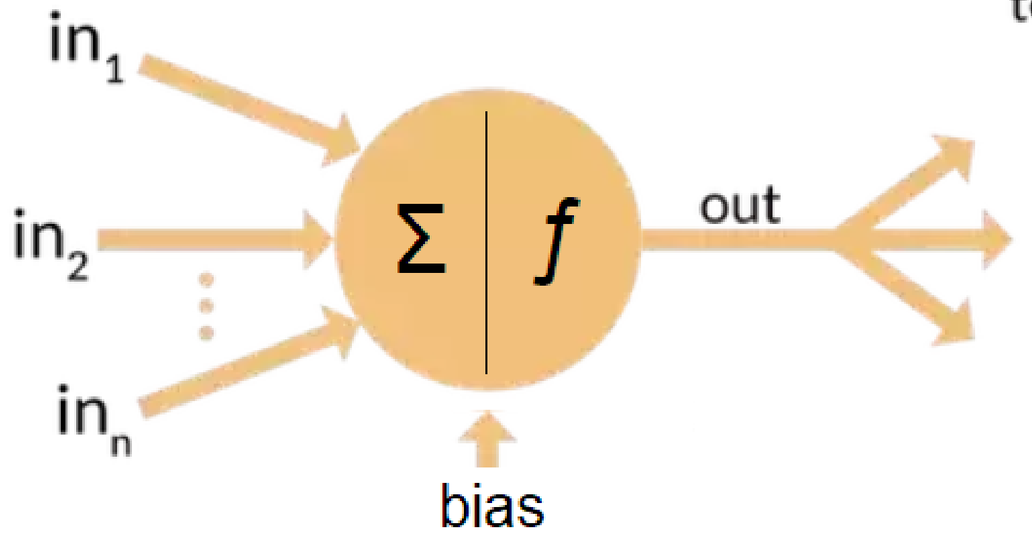
🤦 Reason #2: y ranges from $-\infty$ to $+\infty$

- What will be our *"cutoff"* in predicting 1 or 0?



Our artificial neuron has another superpower!





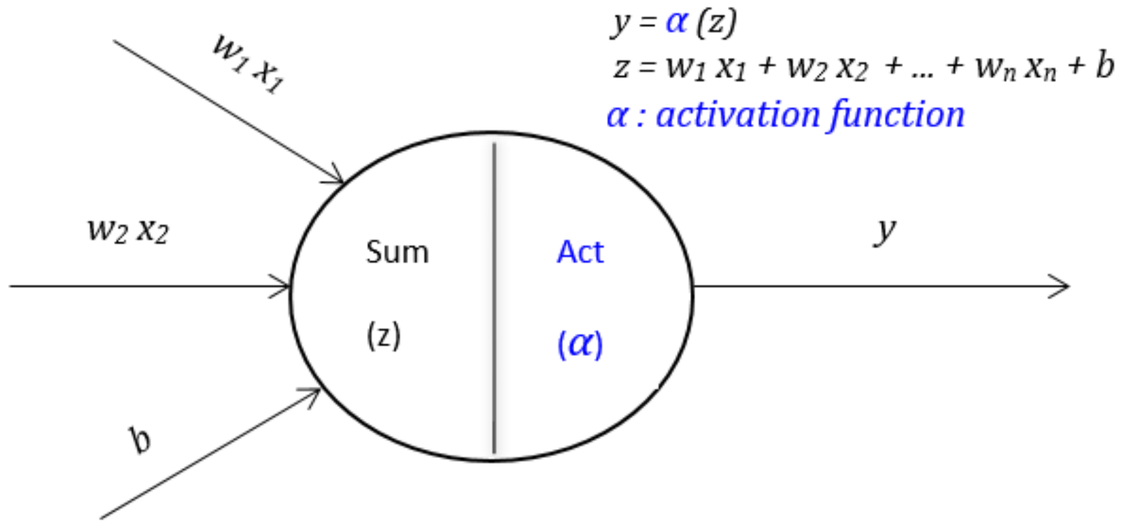
#1 Σ

#2 f

Activation Functions

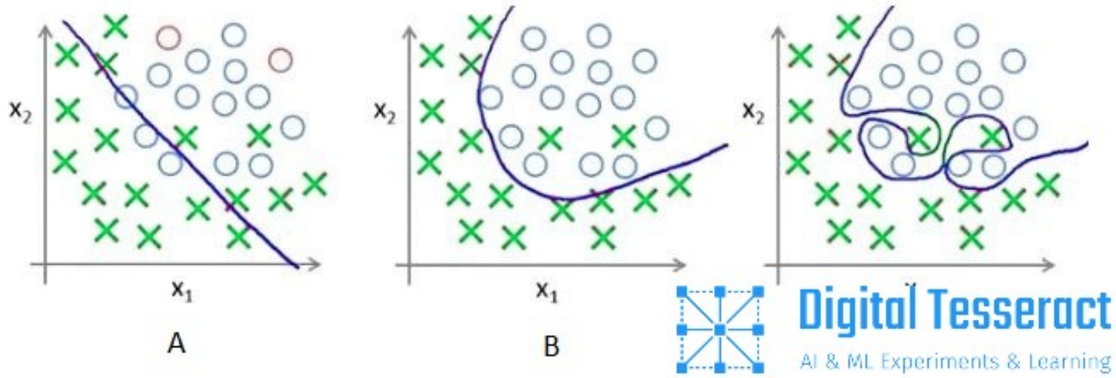
When our brain is fed with a lot of information simultaneously, it tries hard to understand and classify the information into “useful” and “not-so-useful” information.

We need a similar mechanism to ***" 💡light-up neurons"*** for classifying incoming information as “useful” or “less-useful” in the case of **NNs.**

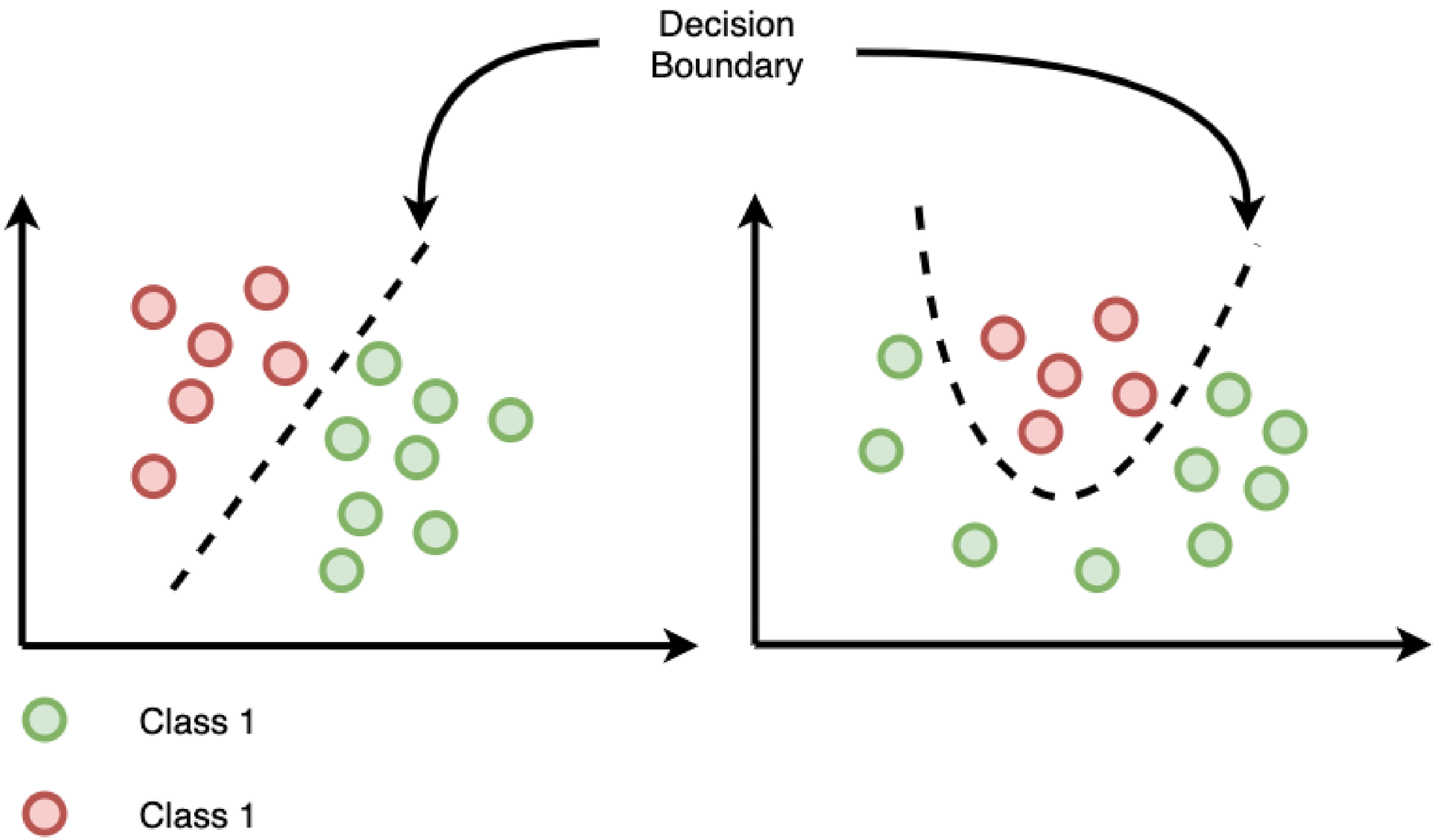


- Activation functions determine the output of **neural networks** like yes or no. It maps the resulting values to a certain range: **0 to 1** or **-1 to 1**, etc.
- The need for these activation functions includes converting the linear input signals and models into non-linear output signals, which aids the learning of high order polynomials for deeper networks.
  - ⚠️ A neural network without an activation function is **essentially just a linear regression model.**

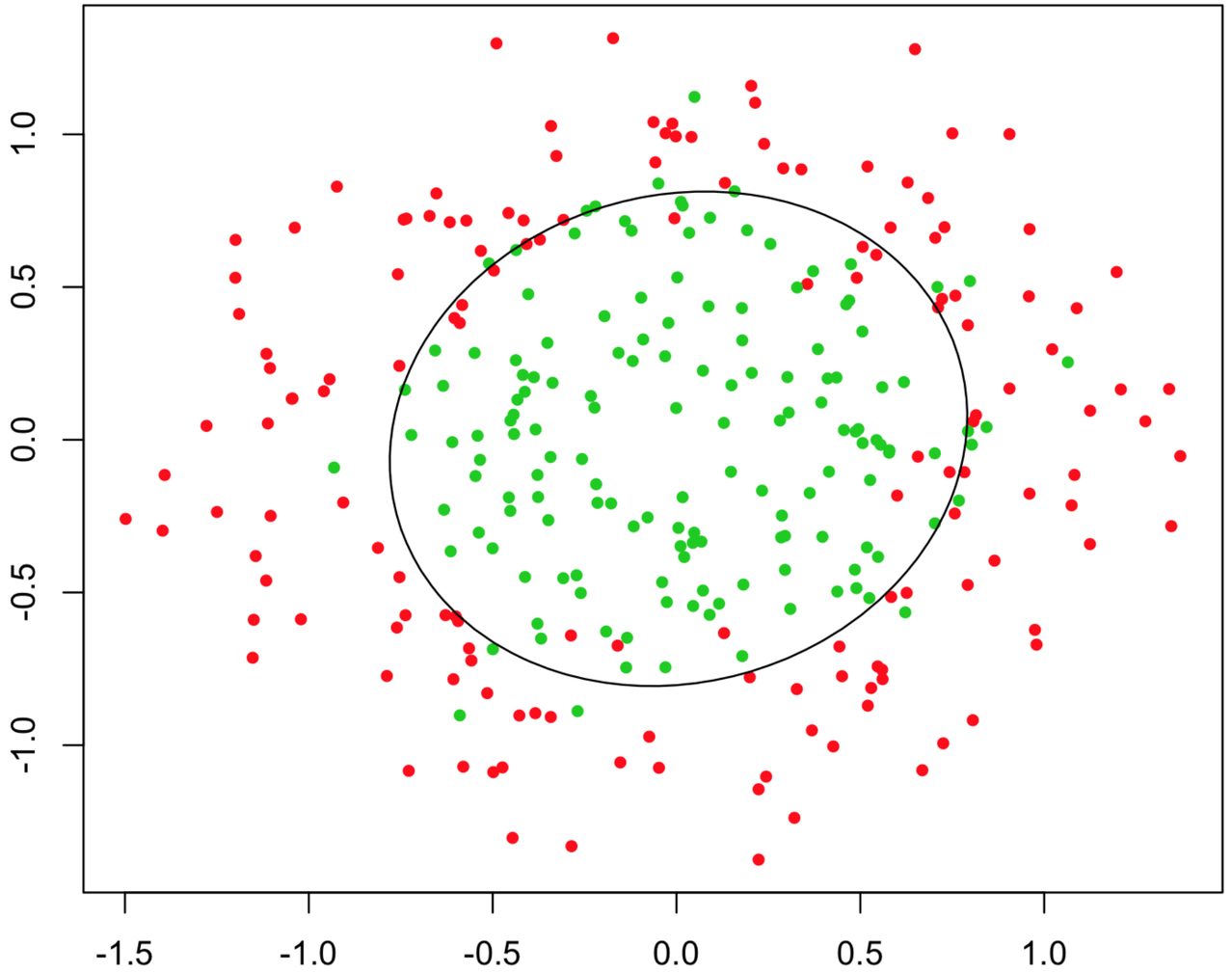
❝ Linear functions are only single-grade polynomials that render the neuron to act as a linear regression model. No matter how many linear functions we stack, we will always get a linear function as an output. Hence activation of the linear combiner enables us to **create complex decision boundaries** by using a combination of multiple neurons. ❞



Linear vs non-linear boundaries



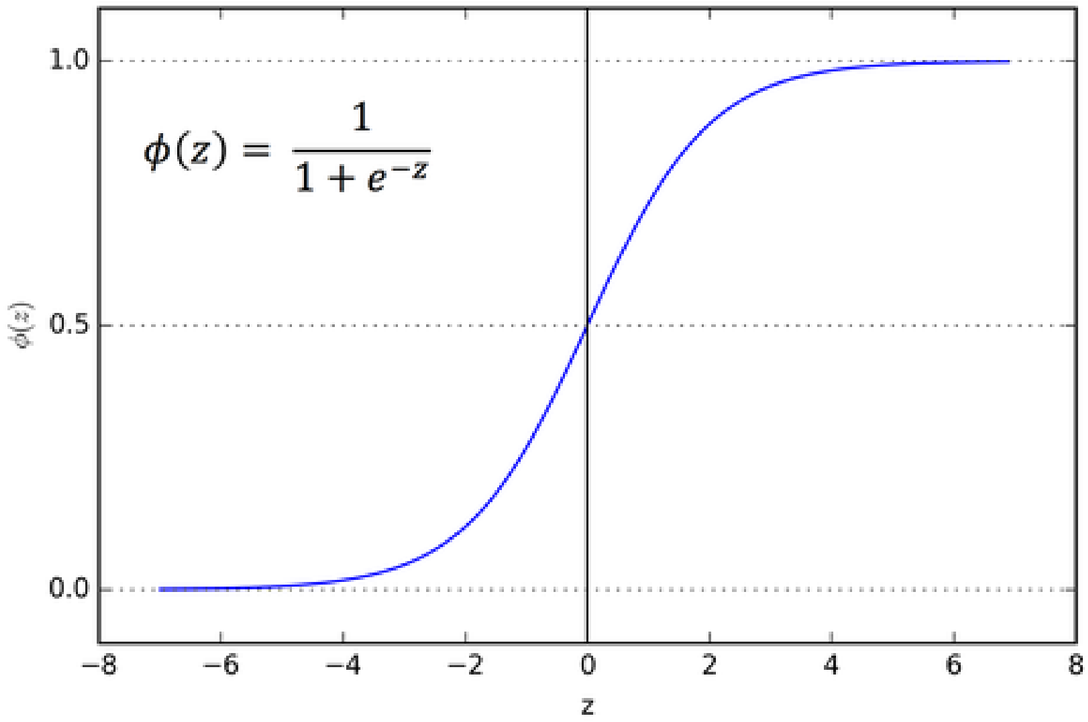
A circular boundary



Sigmoid function

💡 $z = Xw$: there are other activation functions that can transform z, so a *"temporary variaable"* is helpful.

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



z	$\sigma(z)$
-10	4.5397×10^{-5}
-8	0.0003353501
-4	0.0179862010
-0.5	0.3775406688
0	0.5
0.5	0.6224593312
1	0.7310585786
5	0.9933071491
9	0.9998766054

- Interpret $\sigma(z)$ as probability of y being a 1.


That would be 15 dollars, Sigmoid!
Thanks for the ride, Dino!
Here are 0.999999694097773 dollars!!

Congratulations on knowing your first activation function!

Remember its name: Sigmoid

Also remember and UNDERSTAND the how and why

Even if you can't remember the formula, TF / Keras will 😉

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

Loss $\mathcal{f}$: MSE still 👌?

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(\frac{1}{1+e^{-Xw}} - y^{(i)})^2$$

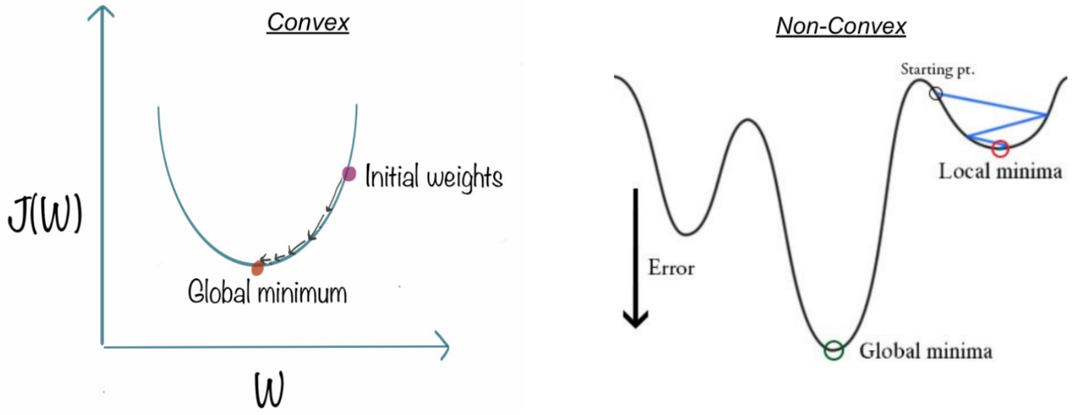
- 🤦 Nope 01: The more obvious reason:
  - It doesn't ***strongly penalize*** misclassifications
  - even for a perfect mismatch

y	$\hat{y}$	$y - \hat{y}$	$(y - \hat{y})^2$
1	0	1	1
1	0.2	0.8	0.64
1	0.9	0.1	0.01

- 🤦 Nope 02: The less-obvious reason:

👈 that's a **non-convex function**, *just research the reason why* ✌ ✌

If we try to use the cost function of the linear regression in 'Logistic Regression' then it would be of no use as it would end up being a **non-convex function with many local minimums**, in which it would be ***very difficult to minimize the cost value and find the global minimum.***



Cross entropy cost function with **sigmoid** function gives a **convex curve with *one local / global minima***.

Binary Cross Entropy Loss

👇 This is the only part you need to understand

- For a single training sample:

$\hat{y} = \sigma(Xw) = \frac{1}{1+e^{-Xw}}$

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

y (0 or 1 only)	$\hat{y}$ (0 to 1 range)	Cost
1	0.99999	almost 0
1	0.7	0.35667
1	0.1	2.30259
1	0.000001	13.81551
0	0.000001	almost 0
0	0.3	0.35667
0	0.9	2.30259
0	0.999999	13.81551

👇👇 *No need to absorb for workshop* 😹

`Just here for completeness`

- For the entire training set, as usual - average of costs

Unvectorized

$$J(w) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(\hat{y}^{(i)}) - (1-y^{(i)}) \log(1-\hat{y}^{(i)})$$

Vectorized

$h = \sigma(Xw)$

$J(w) = -\frac{1}{m} \cdot (y^T \log(h) + (1-y)^T \log(1-h))$

Partial Derivative of vectorized $J(w)$: Derivation if interested

$\frac{\partial}{\partial w} J(w) = \frac{1}{m} X^T [\sigma(Xw) - y]$

Congratulations on knowing your second loss function!

Remember its name: Binary Cross Entropy

Also remember and UNDERSTAND the how and why

Even if you can't remember the formulas and derivatives, TF / Keras will ☺

```
model.compile( # note lowercase 👇 'e'
    loss=keras.losses.BinaryCrossentropy()
)
```

Building a Binary Classifier

It's not much different from linear regression. We just change a couple of things:

1. we now **add** an activation function to our **NEURON**
   - *actually ALL neurons in the entire layer* will have the **same activation function**
   - only that right now, our layer has only one neuron

```
keras.layers.Dense(
    units=1,
    input_shape=[2],
    activation='sigmoid' # 👈 ADD THIS
)
```

2. change the loss function

```
- loss=keras.losses.MeanSquaredError(),
+ loss=keras.losses.BinaryCrossentropy(),
```

3. ✏️ a f to turn $< 50\%$ probability to 0, $\geq 50\%$ to 1

```
def predict(normalized_input):
    # model.predict returns 0 to 1, due to sigmoid
    return model.predict(normalized_input) >= 0.5
```

4. predict using your shiny new function

```
predict(np.array([[38, 75]]))
```

Full example (NN part only)

```
model = keras.Sequential([
    keras.layers.Dense(
        units=1,
        input_shape=[2],
        activation='sigmoid' # string, not object
    )
])
model.compile(  # note lwoercase 👇 `e`
    loss=keras.losses.BinaryCrossentropy(),
    optimizer=keras.optimizers.SGD()
)
```

👩‍💻 Your turn to make a classifier!

- I prepared a notebook that contains **100** training samples, each has 2 exam scores and their admission results:
  - 1 if admitted, 0 if not admitted.
- The machine should classify scores it’s never seen before, in a similar pattern to the training set.

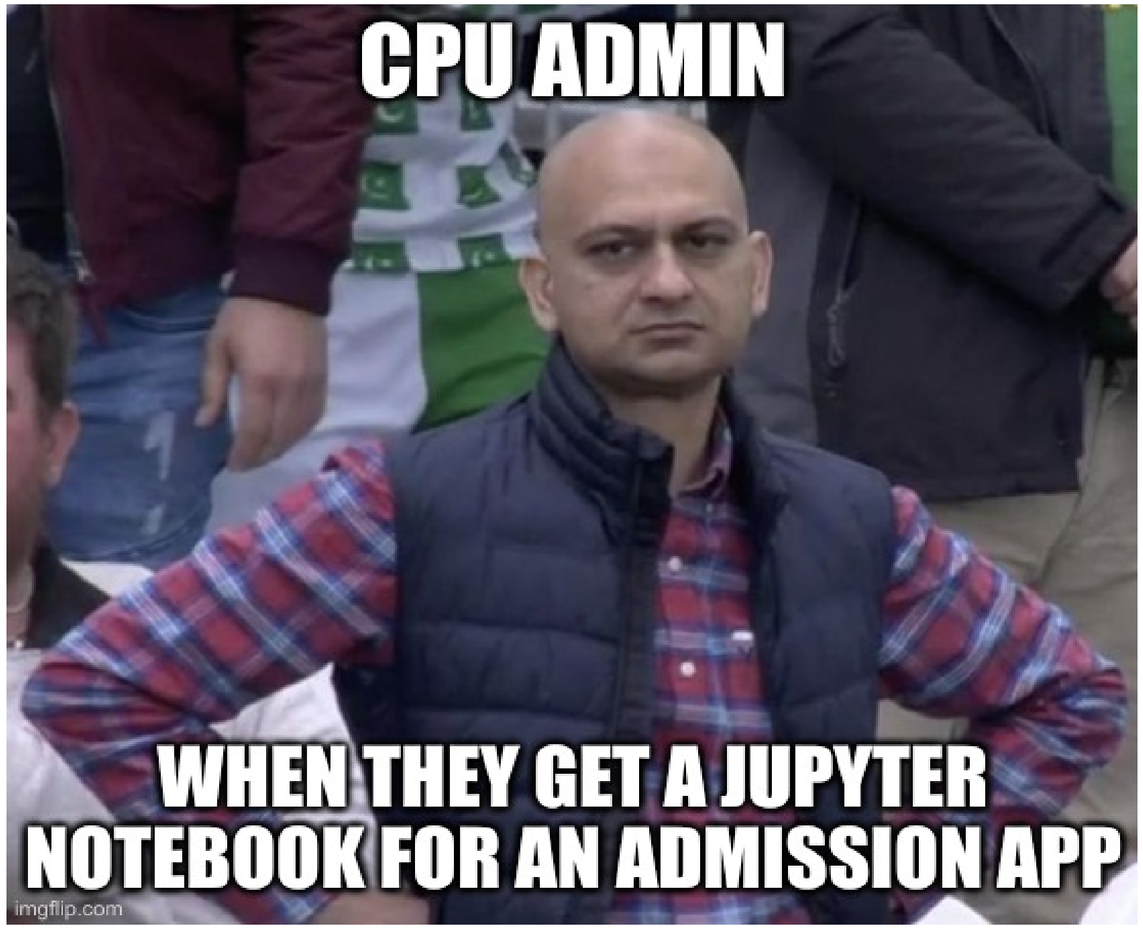
The notebook’s link will be sent via chat, or just ⌨️⌨️🖐️type it out:🖐️

https://bit.ly/ece4241-ml-02

Ready ...

02:00 05:00 10:00 15:00

notebook is STILL a 📓 *"notebook"*, and acts like one



- while you ***can*** extract a `.py` file from a notebook with all Ⓜ️ markdown cells as comment,
- you also don't want **to retrain again** when your script **exits.**

Loading and Saving Models

1. If happy with the accuracy, save as:
   - `admission` - TF native format (SavedModel)
   - `admission.h5` - HDF5 format *differences w/ ☝*
   - others formats friendlier to other languages *e.g. JSON*

```
model.save('admission.h5')
```

2. If you feature-scaled, save your MinMaxScaler as well:

```
import joblib
# some other imports ...

scaler = MinMaxScaler()
# fit, transform ...
joblib.dump(scaler, 'admission-scaler.gz')

# 👇👇 only for notebooks on Google Colab
from google.colab import files
files.download('admission.h5')
files.download('admission-scaler.gz')
```

In your vanilla Python app:

```
import numpy as np
import tensorflow as tf
import joblib   # pip install joblib on your PC

from tensorflow import keras
from sklearn.preprocessing import MinMaxScaler

model = keras.models.load_model('admission.h5')
exam1 = float(input("Enter score for Exam 1: "))
exam2 = float(input("Enter score for Exam 2: "))

scaler = joblib.load('admission-scaler.gz')
scaled_inputs = scaler.transform(np.array([
    [exam1, exam2]
]))

# this will be a 1×1 MATRIX containing True/False
result = model.predict(scaled_inputs) ⩾ 0.5

# a 1×1 matrix is a SCALAR! get it with bool()
should_admit = bool(result) # won't work if not 1×1

print('Admitted' if should_admit else 'Sorry 😭')
```

Congratulations on being able to load and save ML model!

and writing your first ML-powered app.

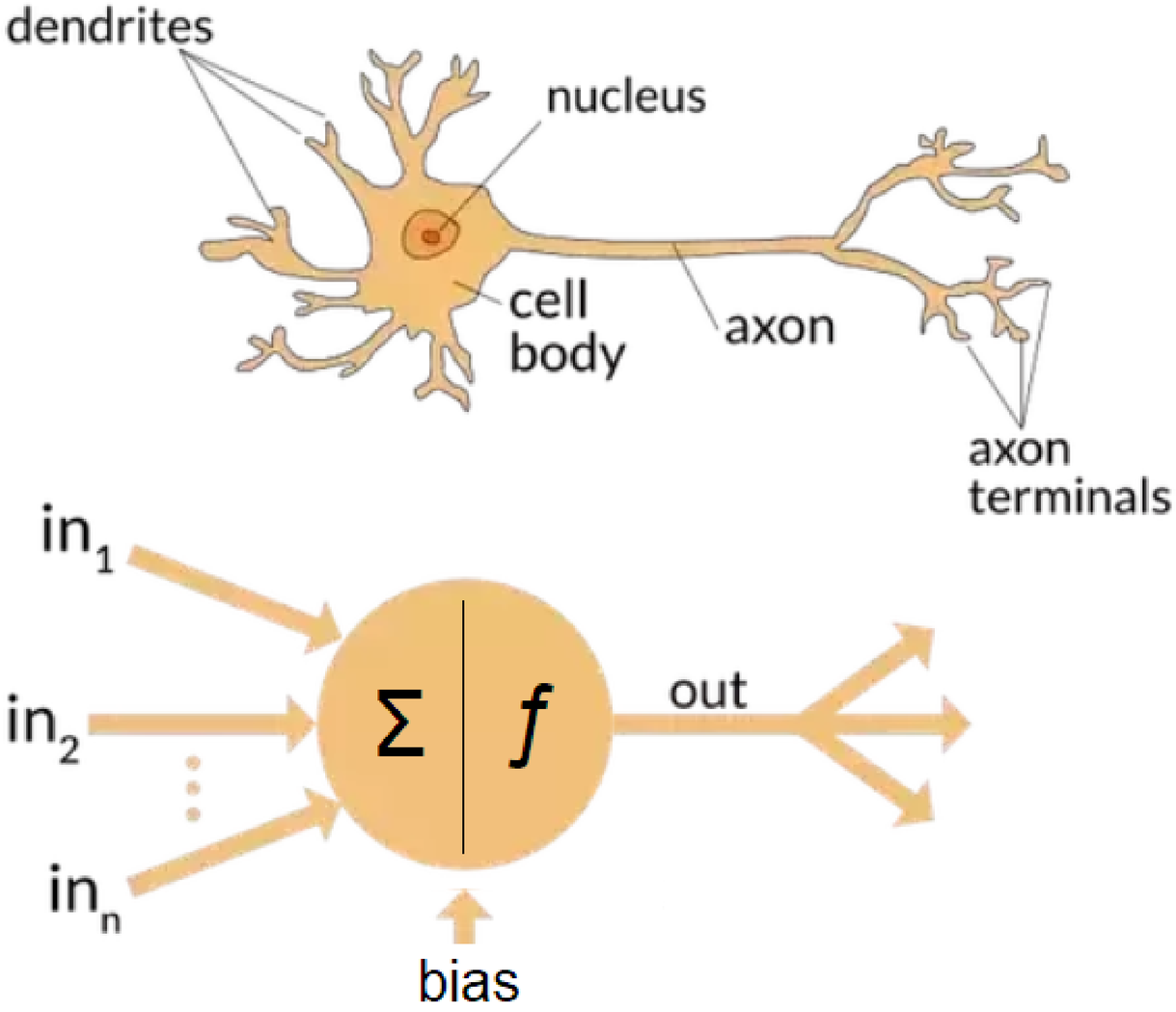
In addition, you even saved and loaded a MinMaxScaler object containing min and max values!

Otherwise we'll have to load the training set in our app, then fit and transform 😫

And that ends 🌛 🗓️ **Day 1!!**

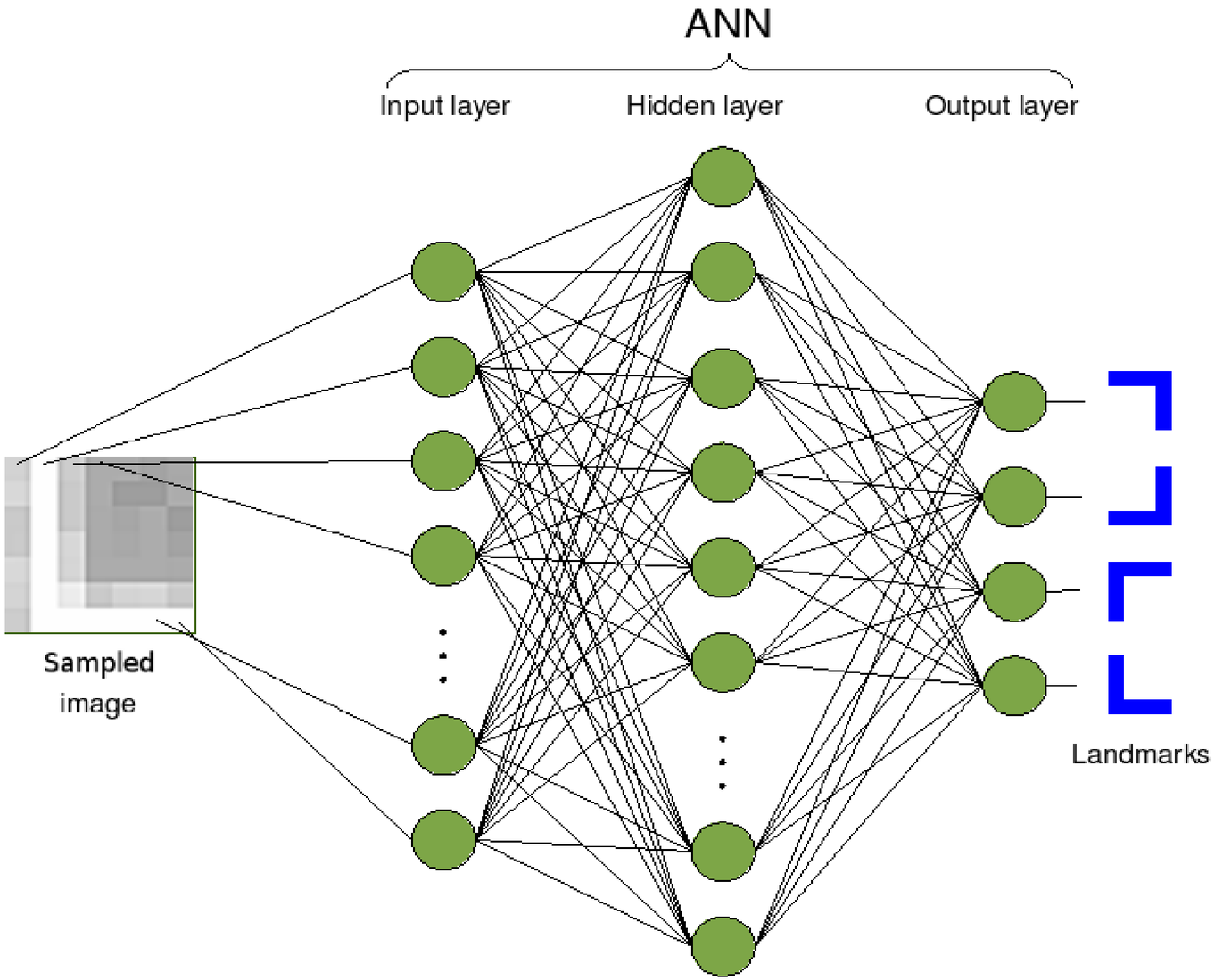
Welcome to ☀️ Day 2

Recall: An Artificial Neuron



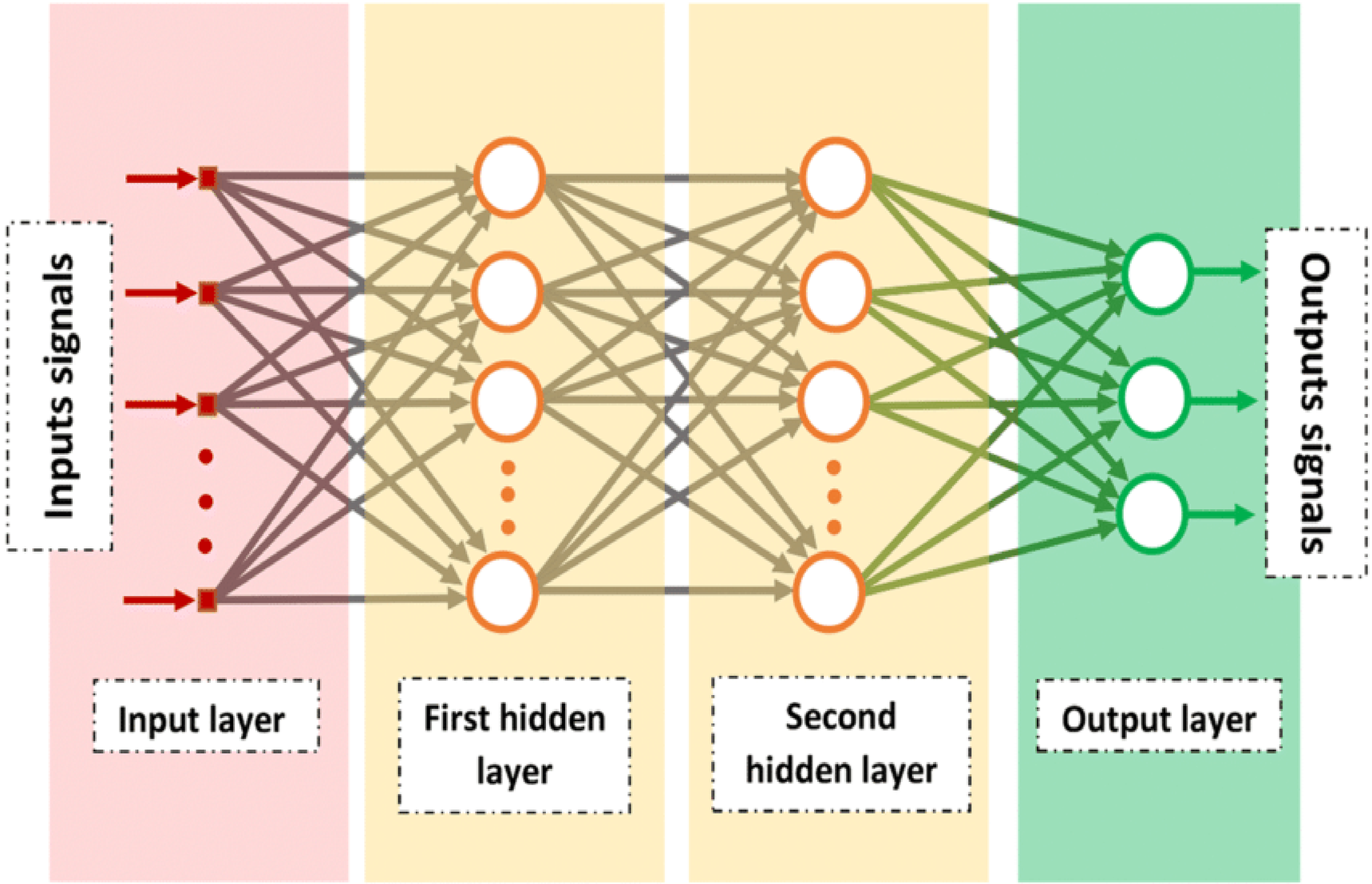
Artificial Neural Networks

1 hidden layer

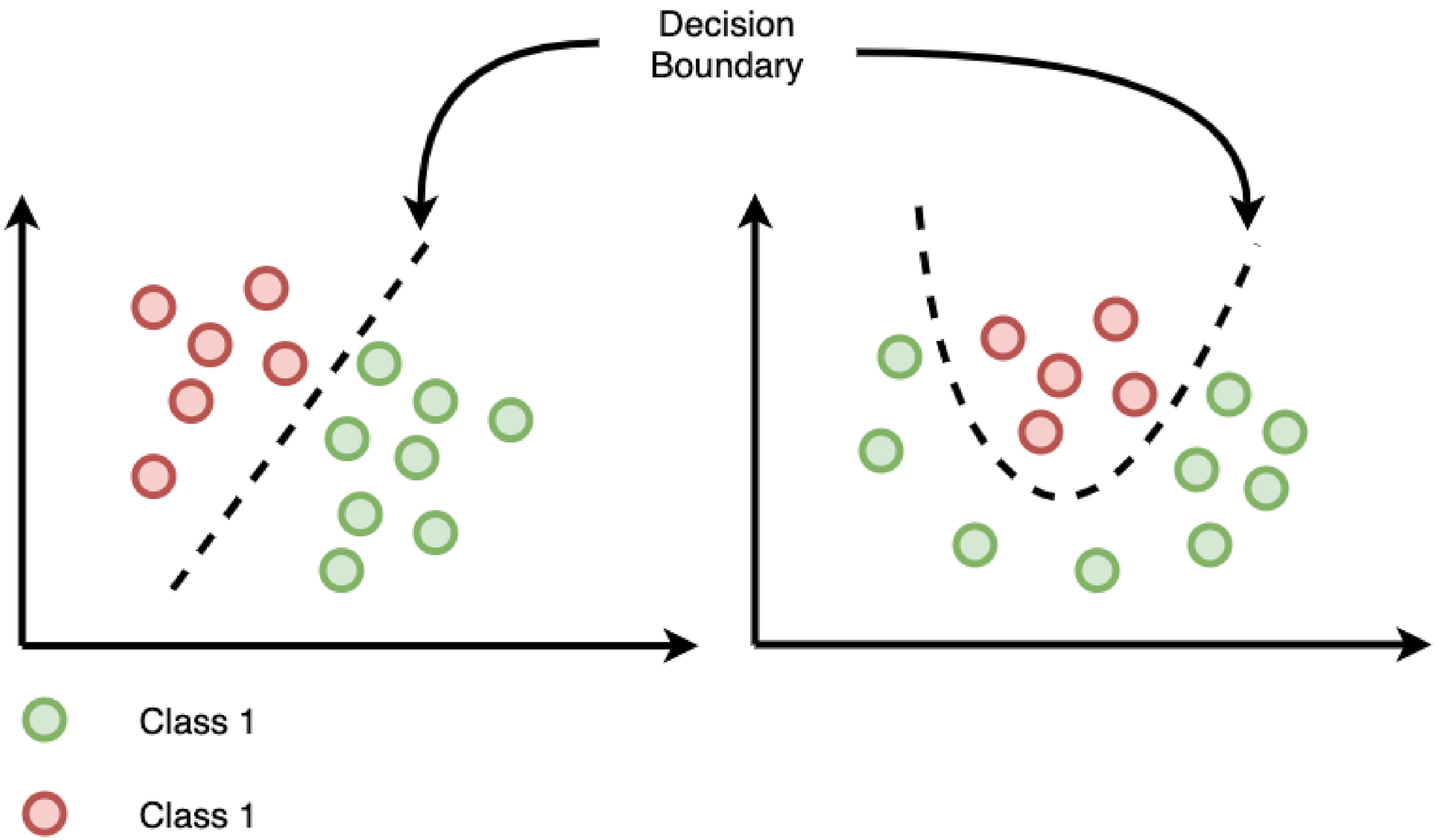


Artificial Neural Networks

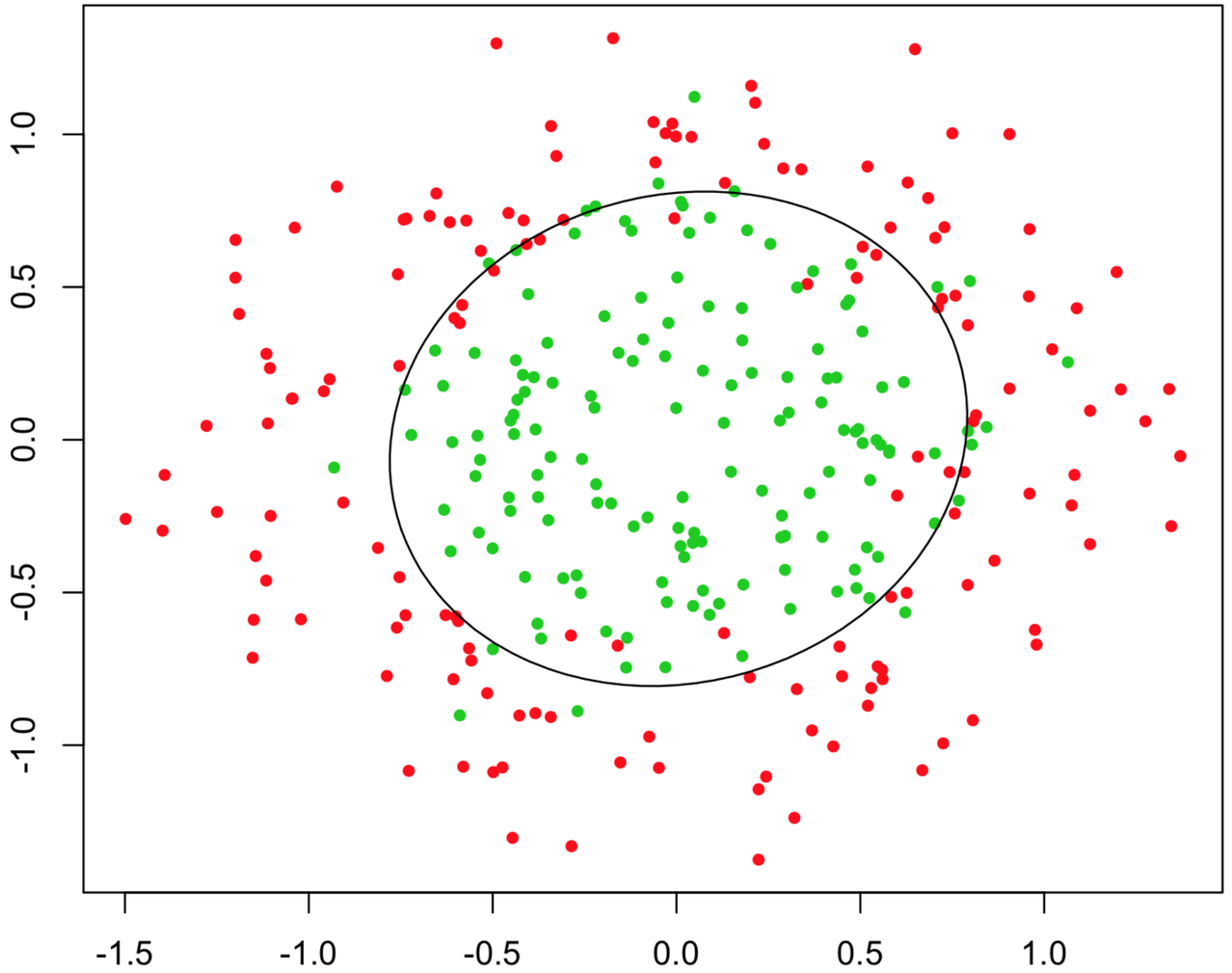
2 hidden layers



Linear vs non-linear boundaries



A circular boundary



The **Universal Approximation Theorem** tells us that Neural Networks has a kind of universality i.e. no matter what $f(x)$ is, ***there is a network that can approximately approach the result and do the job!*** This result holds for any number of inputs and outputs.

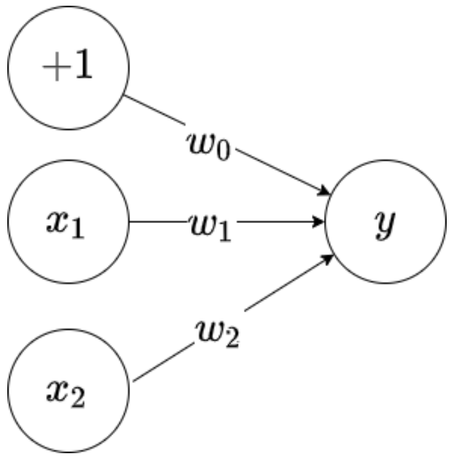
The ***universal approximation theorem*** says that **neural networks can approximate any function**. Now, this is powerful. Because, what this means is that **any task** that can be thought of as a function computation, can be performed/computed/**approximated** by neural networks.

- Almost any task that neural network does today is a function computation:
  - language translation
  - caption generation
  - speech to text
  - image classification
  - etc.

Intuition and Examples for Universal Approx. Theorem

Boolean Logic is not $y = mx + b$ right? Single neuron can approximate AND, NAND, and OR **functions** via σ

x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	x_1 AND x_2 $\sigma(z)$	$z = w_0 + w_1 x_1 + w_2 x_2$	x_1 NAND x_2 $\sigma(z)$	$z = w_0 + w_1 x_1 + w_2 x_2$	x_1 OR x_2 $\sigma(z)$
0	0	$-30 + 0 + 0 = -30$	≈ 0	$30 + 0 + 0 = 30$	≈ 1	$-10 + 0 + 0 = -10$	≈ 0
0	1	$-30 + 0 + 20 = -10$	≈ 0	$30 + 0 + -20 = 10$	≈ 1	$-10 + 0 + 20 = 10$	≈ 1
1	0	$-30 + 20 + 0 = -10$	≈ 0	$30 + -20 + 0 = 10$	≈ 1	$-10 + 0 + 20 = 10$	≈ 1
1	1	$-30 + 20 + 20 = 10$	≈ 1	$30 + -20 + -20 = -10$	≈ 0	$-10 + 0 + 20 = 10$	≈ 1



weights for AND

$$w_{and} = \begin{bmatrix} -30 \\ 20 \\ 20 \end{bmatrix}$$

weights for NAND

$$w_{nand} = \begin{bmatrix} 30 \\ -20 \\ -20 \end{bmatrix}$$

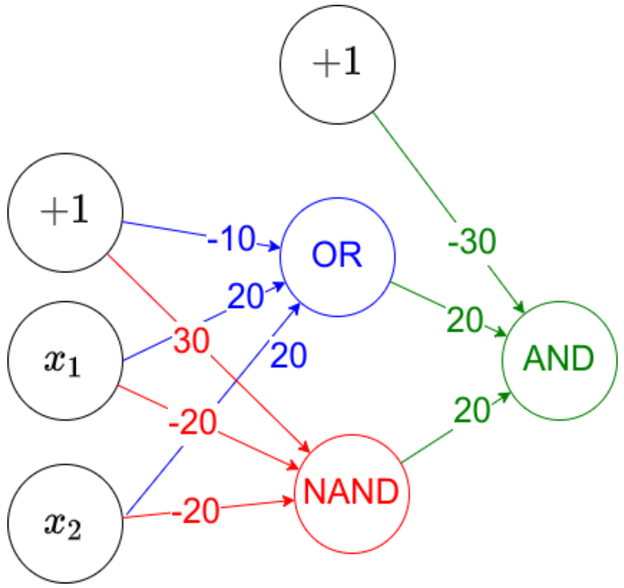
weights for OR

$$w_{or} = \begin{bmatrix} -10 \\ 20 \\ 20 \end{bmatrix}$$

❓ For Php50 load: How can we construct the XOR ***function*** using only those 3 other ***functions*** above `(AND, OR, and NAND)`?

Intuition and Examples for Universal Approx. Theorem

x_1	x_2	x_1 NAND x_2 $\sigma(z)$	x_1 OR x_2 $\sigma(z)$	$z = w_0 + w_1 x_1 + w_2 x_2$	x_1 AND x_2 $\sigma(z)$
0	0	≈ 1	≈ 0	$-30 + 20 + 0 = -30$	≈ 0
0	1	≈ 1	≈ 1	$-30 + 20 + 20 = 10$	≈ 1
1	0	≈ 1	≈ 1	$-30 + 20 + 20 = 10$	≈ 1
1	1	≈ 0	≈ 1	$-30 + 0 + 20 = -10$	≈ 0

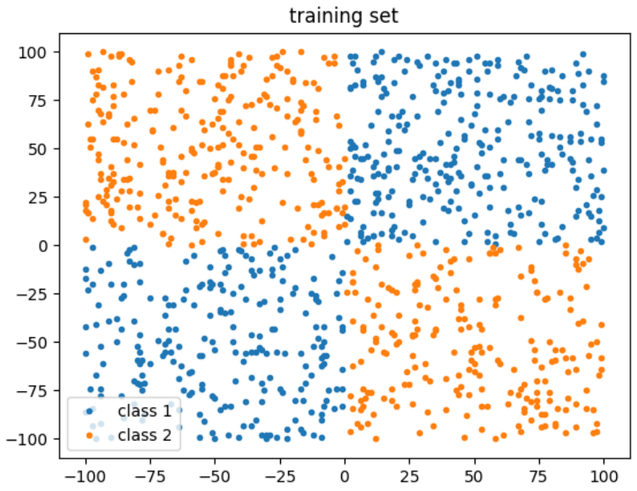


💵 💵 💵 💰 💰 💰 📱 📱 📱

x_1 XOR x_2 = (x_1 OR x_2) AND (x_1 NAND x_2)

XOR only achievable by using a hidden layer!!

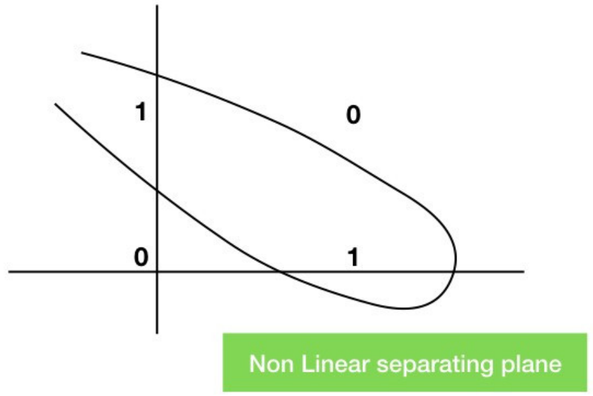
The XOR Problem Boundary



Imagine 2 sensors in the oven, and the alarm fires if 🌡️ temps are uneven which will cook only half of your 🍗🎂

- both sides cold 🔕 $y = 0$
- both sides hot 🔕 $y = 0$
- left cold, right hot 🔔 $y = 1$
- left hot, right cold 🔔 $y = 1$

Solving X-OR



- Non-linear boundaries (e.g. curves) need **hidden layers**
- The NN doesn't actually do:
  x_1 XOR x_2 = (x_1 OR x_2) AND (x_1 NAND x_2)
- it doesn't even output `0` or `1` strictly, but more like `0.00001`, `0.99999`
- It's formula is actually 😅:

$$xor_{approx}(X) = \sigma(\begin{bmatrix}1 & \sigma(Xw_{or}) & \sigma(Xw_{nand})\end{bmatrix} w_{and})$$

XOR using Σ and f …really?

```python
def sigmoid(x): return 1 / (1 + e ** (-x))

def xor(X):
    w_or = np.array([[-10, 20, 20]]).T
    w_nand = np.array([[30, -20, -20]]).T
    w_and = np.array([[-30, 20, 20]]).T

    # add bias
    X = np.insert(X, 0, np.ones(len(X)), axis=1)
    return sigmoid(
        np.hstack((
            np.ones([len(X), 1]), # add bias
            sigmoid(X @ w_or),
            sigmoid(X @ w_nand)
        )) @ w_and
    )

truth_table = np.array([
  [0, 0],
  [0, 1],
  [1, 0],
  [1, 1]
])
np.round(xor(truth_table), 3)
```

Yup! 😄

```
array([[0.],
       [1.],
       [1.],
       [0.]])
```


```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

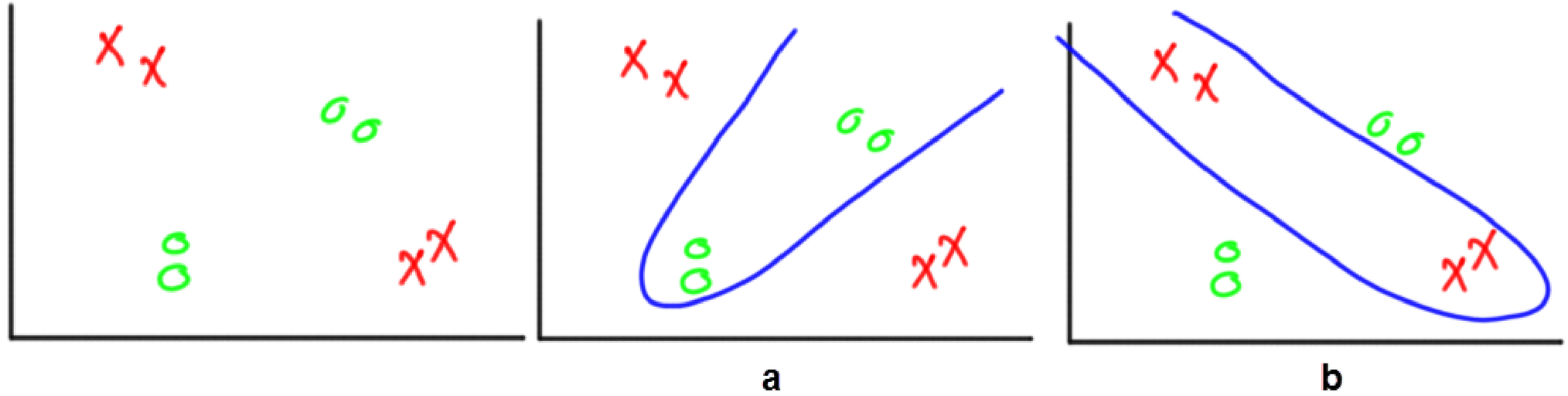
In Python, `^` means "XOR"

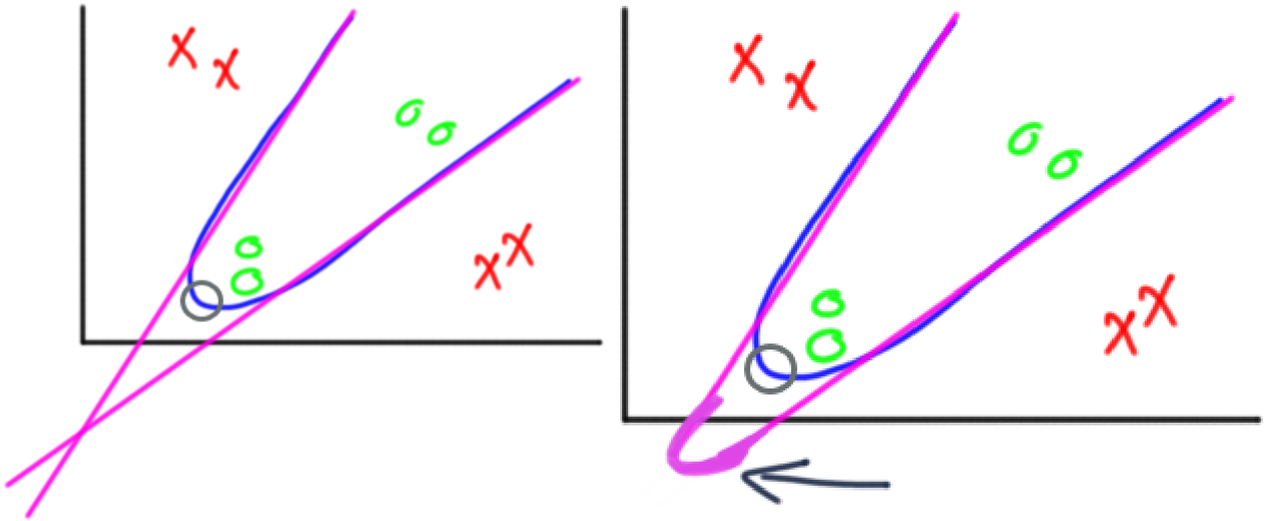
In Discrete Math `^` means "AND"

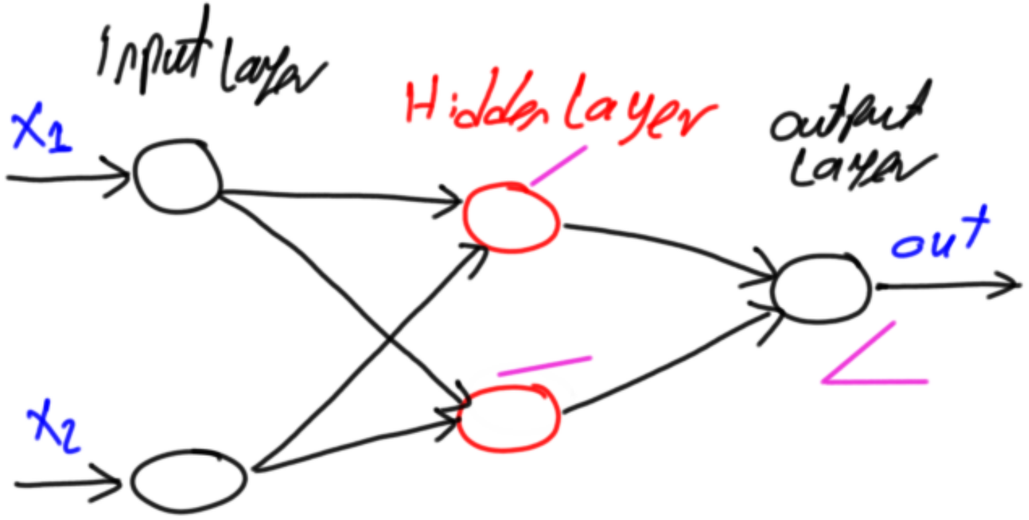
```
σ(
   [1 σ(Xo) σ(Xn)]
) a
```

Beginner Intuition on How Many Hidden Layers

Reminder: the only superpowers of our neuron are Σ summation and f activation.

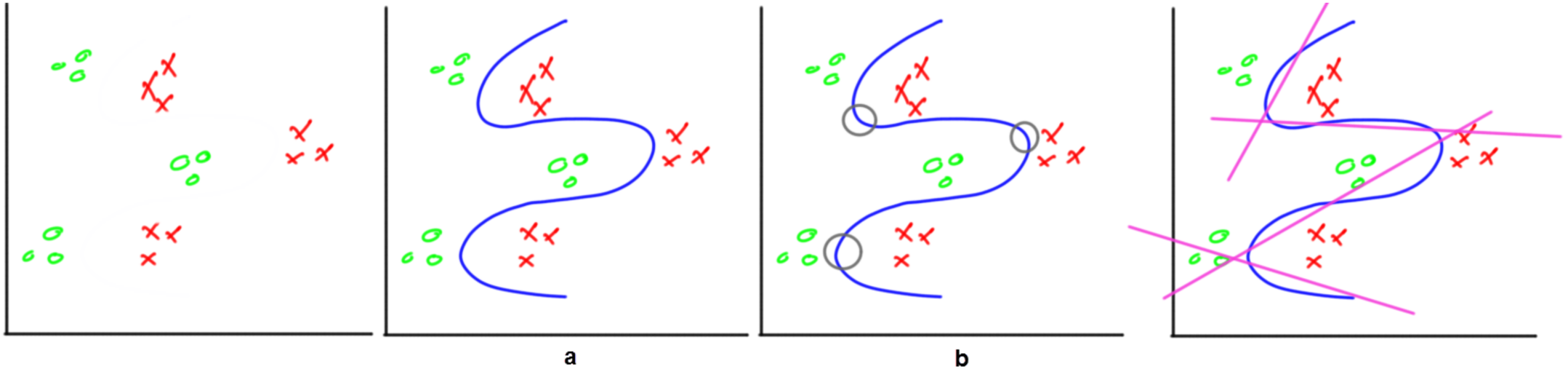


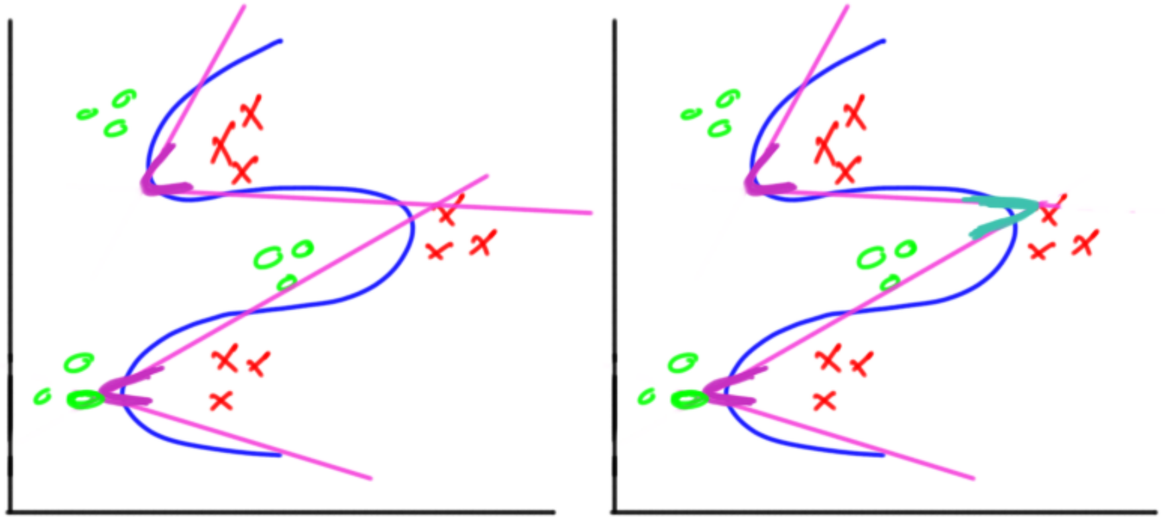


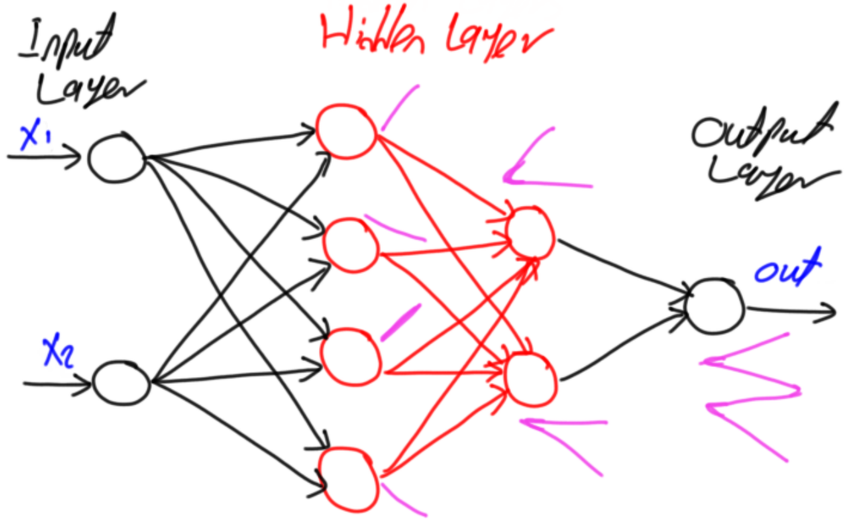


Beginner Intuition on How Many Hidden Layers

How about a polynomials? trigonometric functions? more complicated functions?





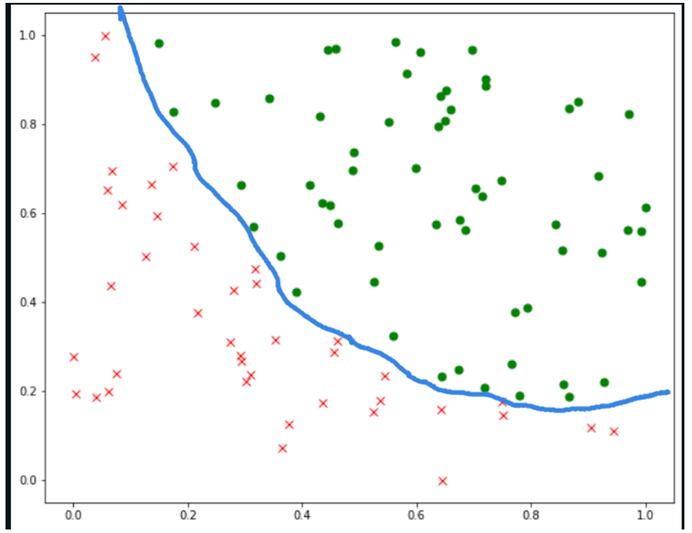


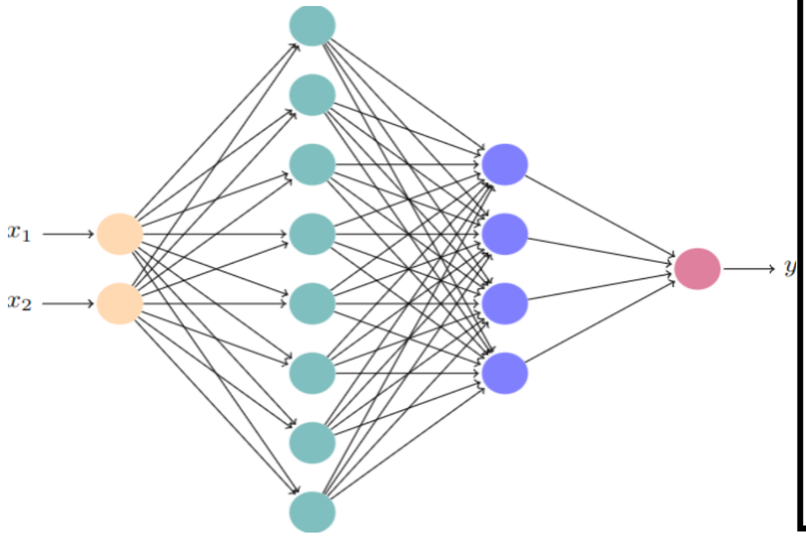
Enhance your classifier

👩‍💻 using moar neurons!

- Enhance the classifier you made yesterday:
  - Add a hidden layer with 8 units
  - and another hidden layer with 4 units
  - getting 99-100% accuracy using this NN (α = 0.2, epochs = 1200)
  - If you can't reach it, that's fine. SGD has a 👇 randomness element to it.

BTW, curves are approximated using lines, so I just estimated and drew the boundary manually





Randomly shuffle (reorder) training examples

Repeat {

for $i := 1, \ldots, m$ {

$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$

(for every $j = 0, \ldots, n$)

}

}

Ready ...

02:00 05:00 10:00 15:00

How about images? They're not just x_1 and x_2?

- experiment *(as we'll do later on)*
- read other people's ML research papers, experiment results, etc. **study the neural nets they developed**

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Yes seriously...

Especially...
If you're serious...
in pursuing

one of the hottest 🔥...
career paths right now

SD-1 contains 58,527 digit images written by 500 different writers. In contrast to SD-3, where blocks of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identities for SD-1 is available and we used this information to unscramble the writers. We then split SD-1 in two: characters written by the first 250 writers went into our new training set. The remaining 250 writers were placed in our test set. Thus we had two sets with nearly 30,000 examples each. The new training set was completed with enough examples from SD-3, starting at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set was completed with SD-3 examples starting at pattern # 35,000 to make a full set with 60,000 test patterns. Only a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3) is available on this site. The full 60,000 sample training set is available.

Many methods have been tested with this training set and test set. Here are a few examples. Details about the methods are given in an upcoming paper. Some of those experiments used a version of the database where the input images where deskewed (by computing the principal axis of the shape that is closest to the vertical, and shifting the lines so as to make it vertical). In some other experiments, the training set was augmented with artificially distorted versions of the original training samples. The distortions are random combinations of shifts, scaling, skewing, and compression.

CLASSIFIER	PREPROCESSING	TEST ERROR RATE (%)	Reference
Linear Classifiers			
linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
pairwise linear classifier	deskewing	7.6	LeCun et al. 1998
K-Nearest Neighbors			
K-nearest-neighbors, Euclidean (L2)	none	5.0	LeCun et al. 1998
K-nearest-neighbors, Euclidean (L2)	none	3.09	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	none	2.83	Kenneth Wilder, U. Chicago
K-nearest-neighbors, Euclidean (L2)	deskewing	2.4	LeCun et al. 1998
K-nearest-neighbors, Euclidean (L2)	deskewing, noise removal, blurring	1.80	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring	1.73	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring, 1 pixel shift	1.33	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring, 2 pixel shift	1.22	Kenneth Wilder, U. Chicago
K-NN with non-linear deformation (IDM)	shiftable edges	0.54	Keysers et al. IEEE PAMI 2007
K-NN with non-linear deformation (P2DHMDM)	shiftable edges	0.52	Keysers et al. IEEE PAMI 2007
K-NN, Tangent Distance	subsampling to 16x16 pixels	1.1	LeCun et al. 1998

SVMs			
SVM, Gaussian Kernel	none	1.4	
SVM deg 4 polynomial	deskewing	1.1	LeCun et al. 1998
Reduced Set SVM deg 5 polynomial	deskewing	1.0	LeCun et al. 1998
Virtual SVM deg-9 poly [distortions]	none	0.8	LeCun et al. 1998
Virtual SVM, deg-9 poly, 1-pixel jittered	none	0.68	DeCoste and Scholkopf, MLJ 2002
Virtual SVM, deg-9 poly, 1-pixel jittered	deskewing	0.68	DeCoste and Scholkopf, MLJ 2002
Virtual SVM, deg-9 poly, 2-pixel jittered	deskewing	0.56	DeCoste and Scholkopf, MLJ 2002
Neural Nets			
2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
2-layer NN, 300 HU, MSE, [distortions]	none	3.6	LeCun et al. 1998
2-layer NN, 300 HU	deskewing	1.6	LeCun et al. 1998
2-layer NN, 1000 hidden units	none	4.5	LeCun et al. 1998
2-layer NN, 1000 HU, [distortions]	none	3.8	LeCun et al. 1998
3-layer NN, 300+100 hidden units	none	3.05	LeCun et al. 1998
3-layer NN, 300+100 HU [distortions]	none	2.5	LeCun et al. 1998
3-layer NN, 500+150 hidden units	none	2.95	LeCun et al. 1998
3-layer NN, 500+150 HU [distortions]	none	2.45	LeCun et al. 1998
3-layer NN, 500+300 HU, softmax, cross entropy, weight decay	none	1.53	Hinton, unpublished, 2005
2-layer NN, 800 HU, Cross-Entropy Loss	none	1.6	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [affine distortions]	none	1.1	Simard et al., ICDAR 2003
2-layer NN, 800 HU, MSE [elastic distortions]	none	0.9	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [elastic distortions]	none	0.7	Simard et al., ICDAR 2003
NN, 784-500-500-2000-30 + nearest neighbor, RBM + NCA training [no distortions]	none	1.0	Salakhutdinov and Hinton, AI-Stats 2007
6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU) [elastic distortions]	none	0.35	Ciresan et al. Neural Computation 10, 2010 and arXiv 1003.0358, 2010

Why do some of these terms sound familar?

🤔🤔🤔🤔🤔

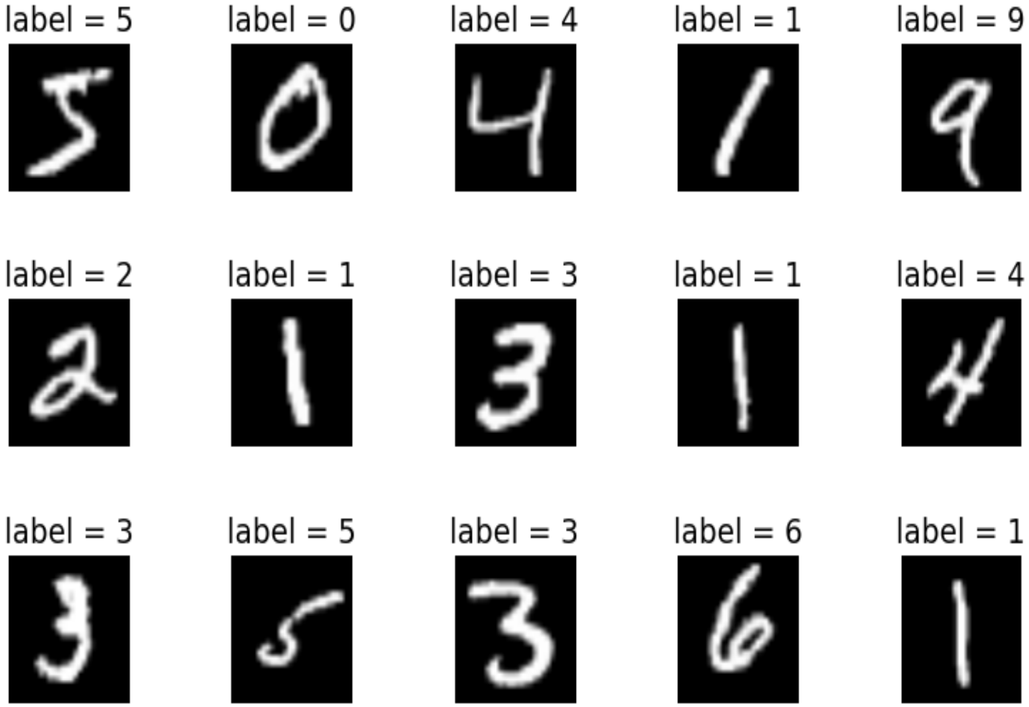
deep convex net, unsup pre-training [no distortions]	none	0.83	Deng et al. Interspeech 2010
Convolutional nets			
Convolutional net LeNet-1	subsampling to 16x16 pixels	1.7	LeCun et al. 1998
Convolutional net LeNet-4	none	1.1	LeCun et al. 1998
Convolutional net LeNet-4 with K-NN instead of last layer	none	1.1	LeCun et al. 1998
Convolutional net LeNet-4 with local learning instead of last layer	none	1.1	LeCun et al. 1998
Convolutional net LeNet-5, [no distortions]	none	0.95	LeCun et al. 1998
Convolutional net LeNet-5, [huge distortions]	none	0.85	LeCun et al. 1998
Convolutional net LeNet-5, [distortions]	none	0.8	LeCun et al. 1998
Convolutional net Boosted LeNet-4, [distortions]	none	0.7	LeCun et al. 1998
Trainable feature extractor + SVMs [no distortions]	none	0.83	Lauer et al., Pattern Recognition 40-6, 2007
Trainable feature extractor + SVMs [elastic distortions]	none	0.56	Lauer et al., Pattern Recognition 40-6, 2007
Trainable feature extractor + SVMs [affine distortions]	none	0.54	Lauer et al., Pattern Recognition 40-6, 2007
unsupervised sparse features + SVM, [no distortions]	none	0.59	Labusch et al., IEEE TNN 2008
Convolutional net, cross-entropy [affine distortions]	none	0.6	Simard et al., ICDAR 2003
Convolutional net, cross-entropy [elastic distortions]	none	0.4	Simard et al., ICDAR 2003
large conv. net, random features [no distortions]	none	0.89	Ranzato et al., CVPR 2007
large conv. net, unsup features [no distortions]	none	0.62	Ranzato et al., CVPR 2007
large conv. net, unsup pretraining [no distortions]	none	0.60	Ranzato et al., NIPS 2006
large conv. net, unsup pretraining [elastic distortions]	none	0.39	Ranzato et al., NIPS 2006
large conv. net, unsup pretraining [no distortions]	none	0.53	Jarrett et al., ICCV 2009
large/deep conv. net, 1-20-40-60-80-100-120-120-10 [elastic distortions]	none	0.35	Ciresan et al. IJCAI 2011
committee of 7 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.27 +-0.02	Ciresan et al. ICDAR 2011
committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.23	Ciresan et al. CVPR 2012

References

[LeCun et al., 1998a]

MNIST

- *Modified* National Institute of Standards and Technology
- is a large database of **handwritten digits**
- widely used for training and testing in the field of machine learning



Let’s get spoonfed 🥣 by

```
1    mnist = tf.keras.datasets.mnist
```

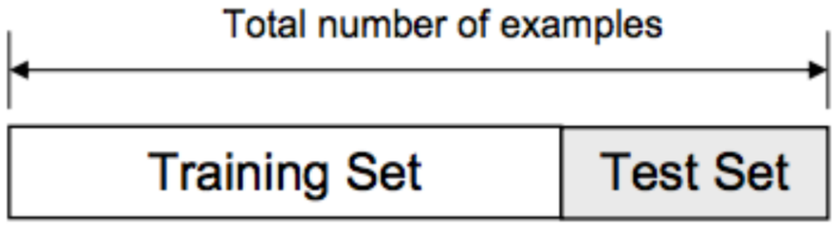
- Many **popular** datasets are *"built-in"*, or *"preloaded"* in many ML libraries
  - Boston Housing is one example: we don’t have to load house pricing data from a CSV
  - although this dataset has ethical issues, and was deprecated in Scikit-learn
- **Convenience:** gets rid of distractions in bringing your own images, loading CSVs, etc.
  - can focus on the core ML stuff
  - very helpful especially for learning and research
- Guide and List of Datasets for TensorFlow

Train-test Split

- What we did in the Admissions Classifier workshop is just measuring the ***accuracy of our training***
  - we tested our model against **data it has already seen**
  - TensorFlow also gives us this via the ***accuracy*** metric *and I let you do this the hard way??* 😈😈

```
model.compile(optimizer=sgd, loss=bce, metrics=['accuracy'])
```

- How well will our ML model predict the correct labels of **data it has never seen during training?**



There's no optimal ratio, and it depends on your project. For guidance, you can start with `80:20`, or `90:10`. MNIST dataset is already split, and uses a 6:1 ratio (60,000 for ***training***, 10,000 for ***testing***).

- Try this on your notebook. ⚠️🚧 *Tuple-ception ahead*

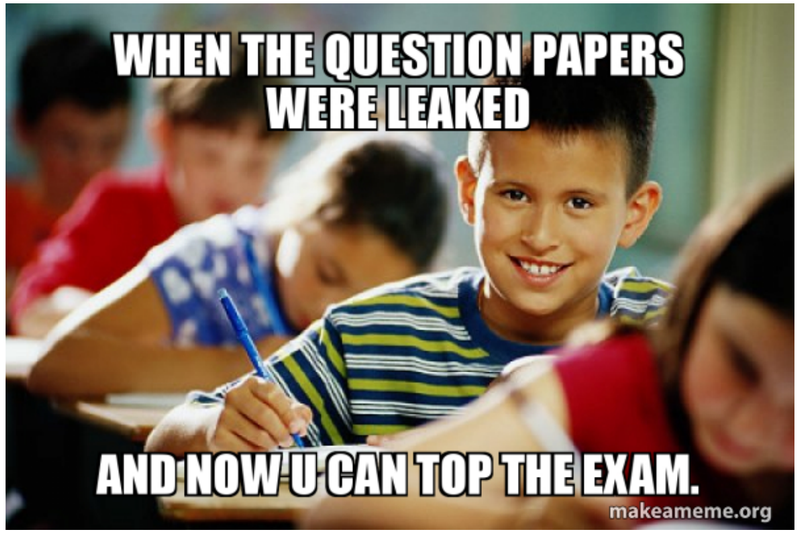
```
#     X_train   ,     y_train           X_test   ,    y_test
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
training_images.shape, test_images.shape
# Output cell should show: ((60000, 28, 28), (10000, 28, 28))
```

Congratulations on knowing an industry best-practice in ML!

Train-test Split

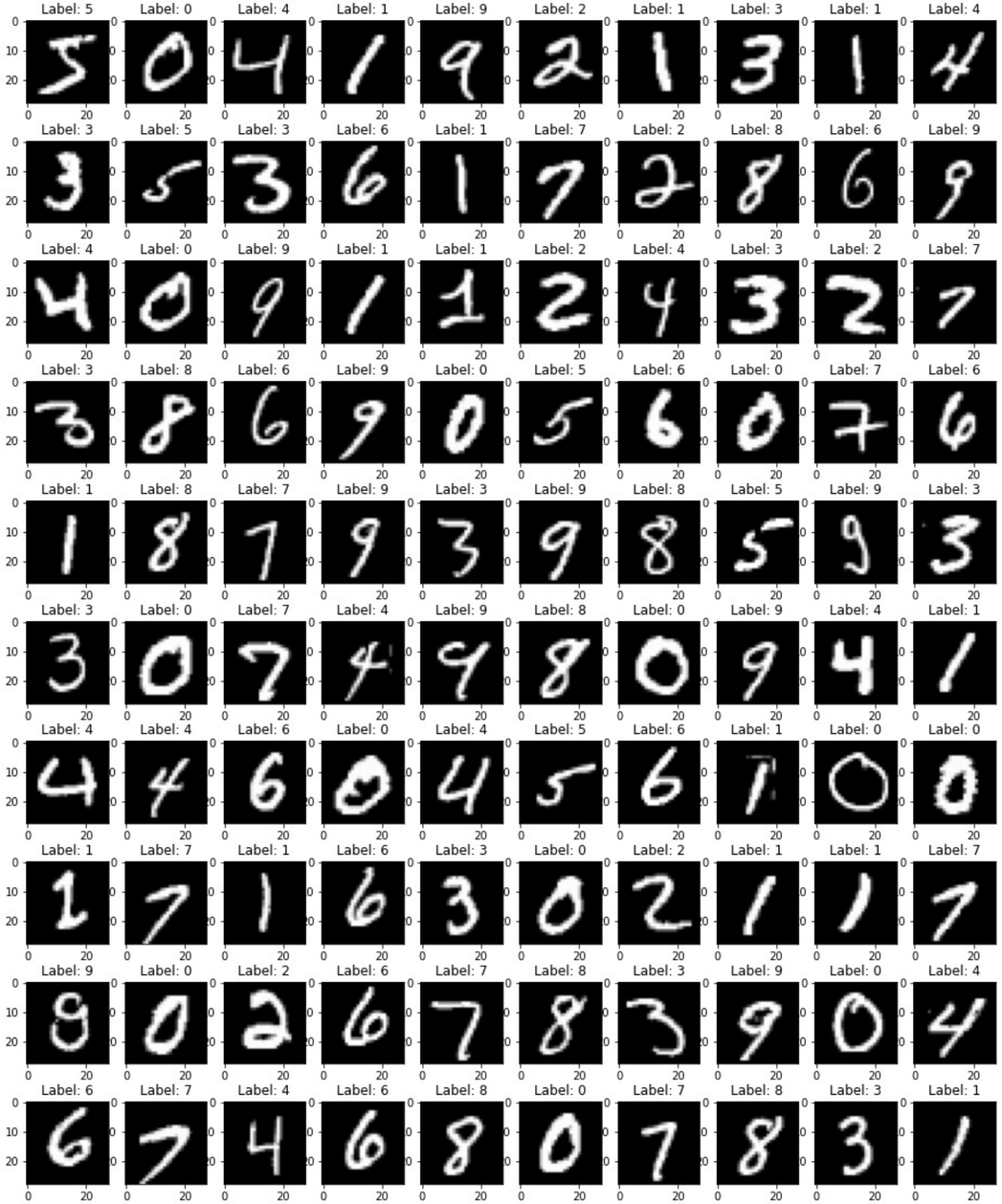
You have just "hidden" a portion of your data from the machine.

Though it was already split by TF/Keras for us, the last exercise today will let you do this yourself.



Visualizing the first 100 images from the training set

```
import matplotlib.pyplot as plt

# taller vertical size coz of Label: <digit>
plt.rcParams['figure.figsize'] = (16, 20)

for i in range(100):
    # 10×10 grid, draw at index `i` (1-based)
    plt.subplot(10, 10, i + 1)

    # without `cmap='gray'`, it will be greenish
    plt.imshow(training_images[i], cmap='gray')
    plt.title(f"Label: {training_labels[i]}")
```

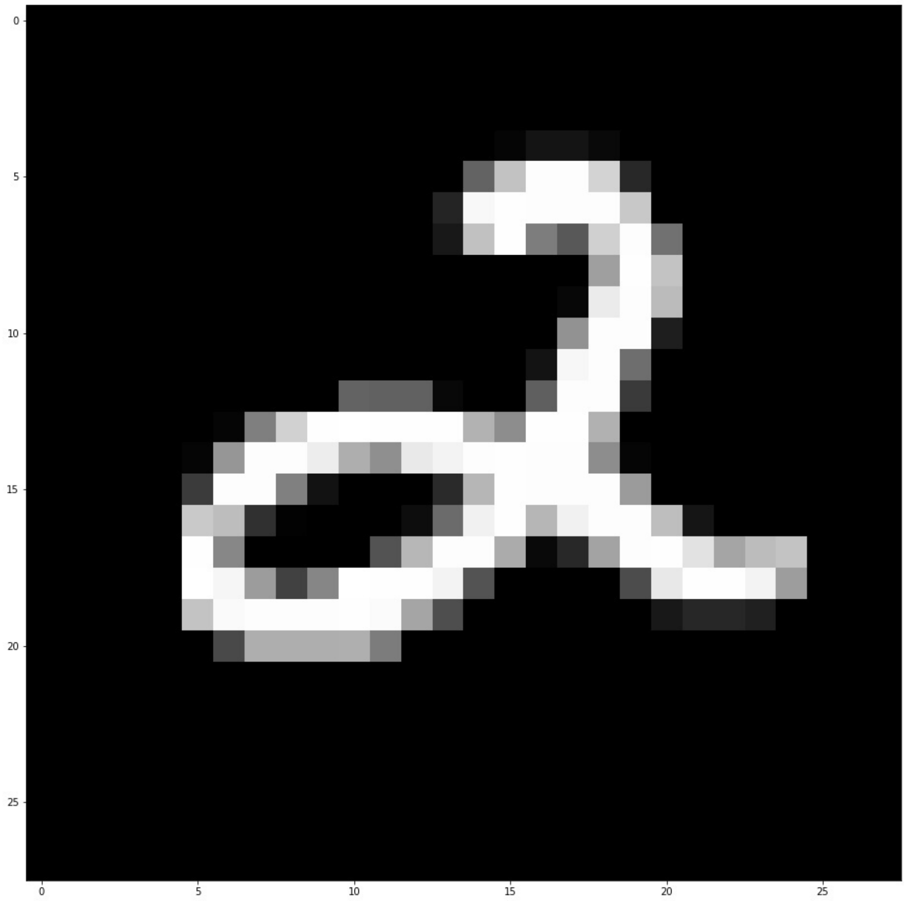


Notice the axes labels. It shows that the images are 28x28 pixels in size.

Visualize a single image

```python
# Set number of characters per row when printing
np.set_printoptions(linewidth=320)

def show_image(index):# Print the traiing label
    print(f'LABEL: {training_labels[index]}', '\n')

    # and the image's matrix representation
    print(f'MATRIX:\n {training_images[index]}')

    plt.imshow(training_images[index], cmap='gray')

# Play with this in your notebook
show_image(14_344) # indices are from 0 to 59_999
show_image(4_241)  # yeah lucky number
```

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   5  20  20   9   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  99 194 253 253 211  41   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  37 248 254 253 253 253 200   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  24 193 254 125  88 208 253 114   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 159 254 196   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   7 235 253 187   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 146 253 253  30   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  19 247 253 110   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  99  98  98   8   0   0  95 253 253  58   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   5 127 209 254 255 254 254 254 178 141 254 254 177   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   5 150 253 253 237 174 143 233 244 253 254 253 253 141   5   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0  59 253 253 128  18   0   0   0  42 182 254 253 253 253 155   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0 201 189  49   2   0   0   0  13 107 241 254 182 241 253 253 190  21   0   0   0   0   0   0]
 [  0   0   0   0   0 254 135   0   0   0   0  83 183 253 253 171   9  41 163 253 254 226 165 188 195   0   0   0]
 [  0   0   0   0   0 255 246 156  66 134 255 254 254 244  83   0   0   0   0  76 232 254 254 244 157   0   0   0]
 [  0   0   0   0   0 196 250 253 253 253 254 251 165  78   0   0   0   0   0   0  24  39  39  33   0   0   0   0]
 [  0   0   0   0   0   0  74 174 174 174 175 124   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```



- Here you see the image’s actual image, and its matrix representation.
- minus the numbers to indicate the pixels’ brightness, the matrix does look like the handwritten digit image right?

ML Model Code

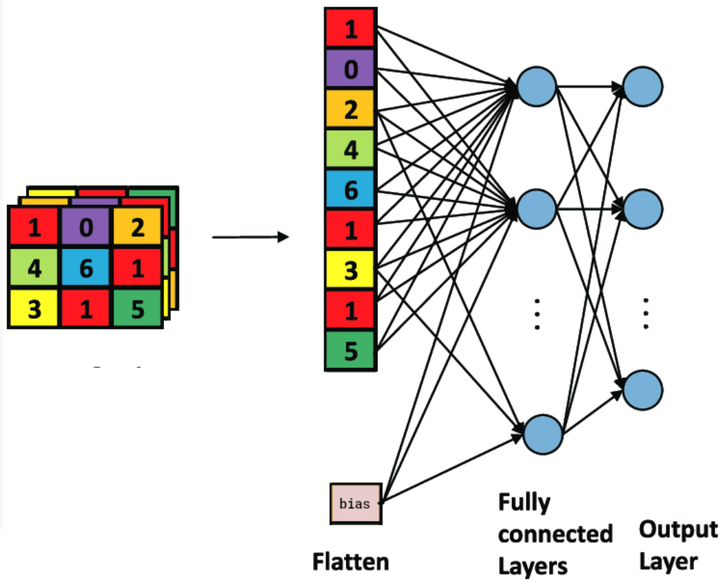
```python
# we can normalize pixel brightness w/o MinMaxScaler
training_images_norm  = training_images / 255.0
test_images_norm = test_images / 255.0

model = tf.keras.models.Sequential([
    keras.layers.Flatten(), # ❓
    keras.layers.Dense(128, activation='sigmoid'),
    keras.layers.Dense(10, activation='softmax') # ❓
])

model.compile(
    optimizer=keras.optimizers.SGD(),      # 👇🔻❓
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

# you should get around 89% accuracy w/ just 5 epochs
model.fit(
    training_images_norm, # used to call this `X_norm`
    training_labels,      # used to call this `y`
    epochs=5
)
```

Flatten Layer

(a.k.a. *let's not have a* $60000 \times 28 \times 28$ ~~*matrix*~~ *tensor*)

```python
training_images.shape # (60000, 28, 28)

keras.layers.Flatten()(training_images).shape
# (60000, 784)      # 👆 Python __call__ (callables)
```

Converts $\begin{bmatrix} 1 & 0 & 2 \\ 4 & 6 & 1 \\ 3 & 1 & 5 \end{bmatrix}$ into $\begin{bmatrix} 1 & 0 & 2 & 4 & 6 & 1 & 3 & 1 & 5 \end{bmatrix}$



but we usually ~~draw~~ visualize layer neurons vertically, so the **input layer** *kinda* receives pixel values from **one** training image *like this* 👉

Congratulations on knowing your second NN layer!

Flatten

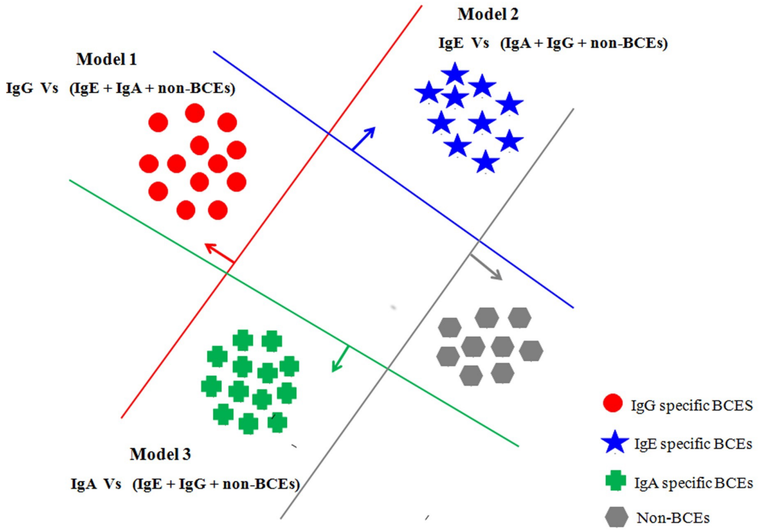
Images are 2D arrays, and the Flatten layer 😮‍💨 flattens them into 1D, so they can just be a single row in the input matrix X.

Dense was the first layer you learned, but was too early to congratulate 😂

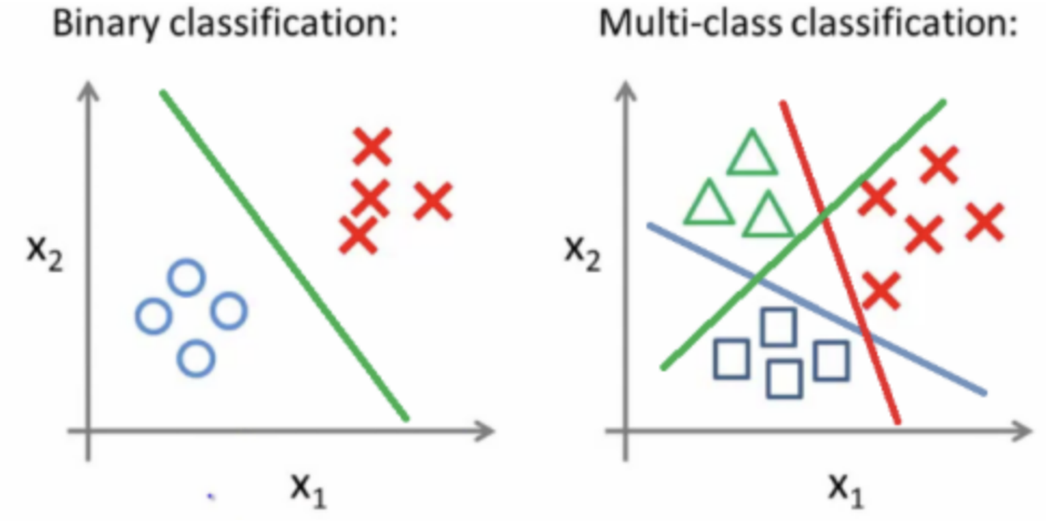
```
# just add before your first Dense() layer
keras.layers.Flatten()
```

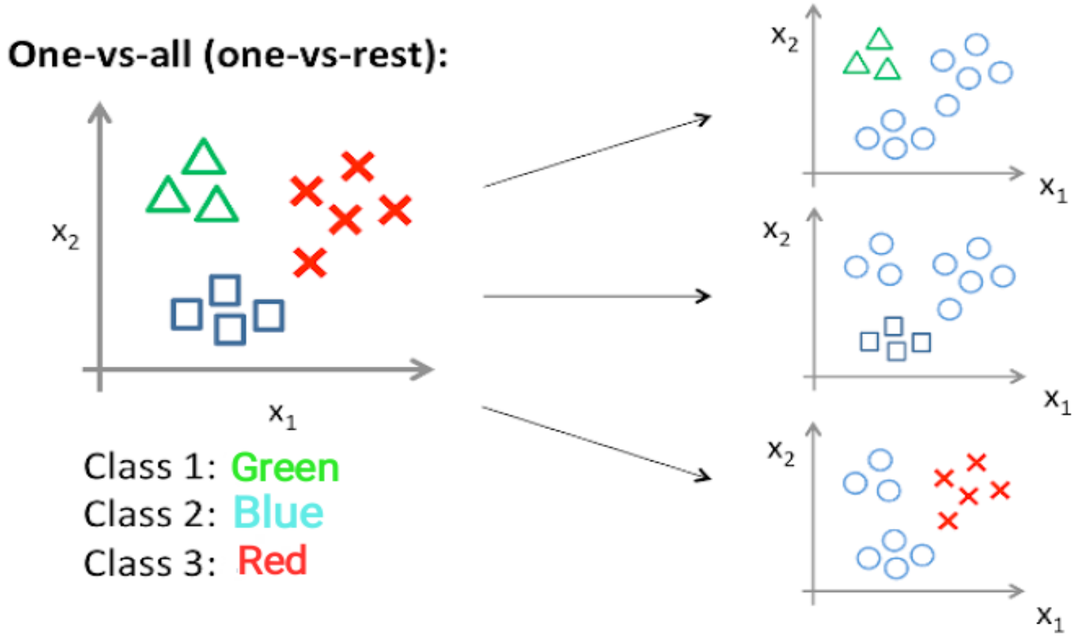
Multi-class Classification

- We only need **one** neuron in the ***output layer*** for binary classification (*one decision boundary*)
- **three** to classify △ □ ×
  - △ vs NOT △, □ vs NOT □, × vs NOT ×
- **four** neurons to classify 4 classes, etc.
- *"no such thing"* as a two-unit output layer

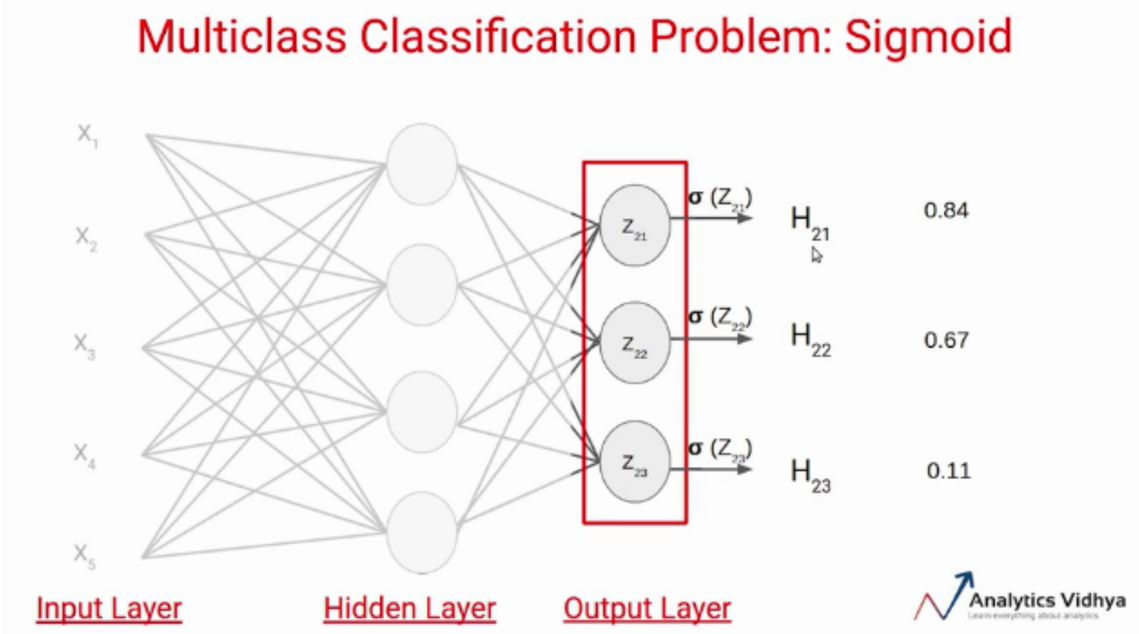


BCE here is B-cell epitopes (antibodies), not Binary Cross Entropy 😂





Softmax

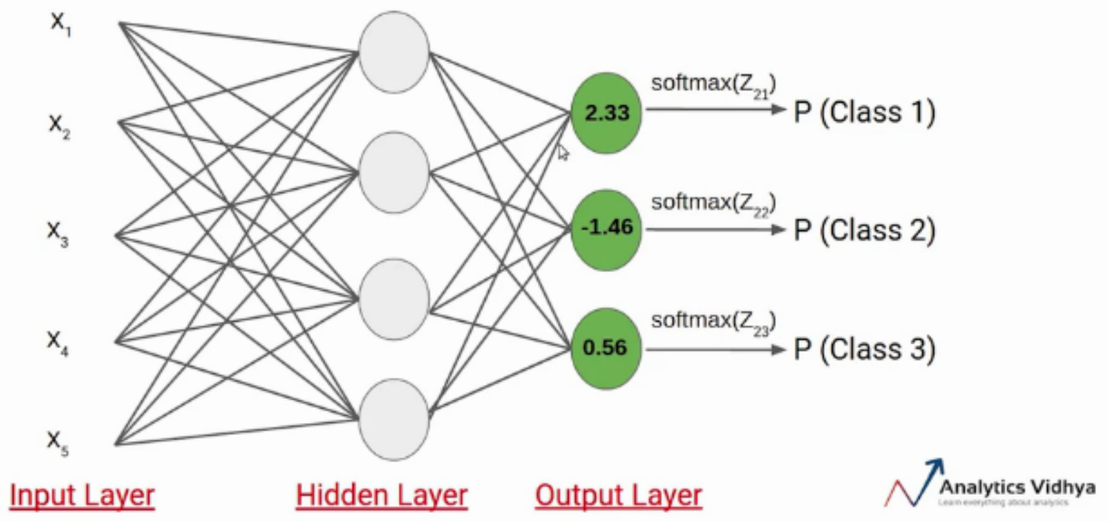


1. if we apply a thresh-hold of say 0.5
   - $H_{21} = 0.84$ and $H_{22} = 0.67$ belongs to two classes.
2. probability values don't add up to 100%.
   - 84% chance of being ~~cat~~ H_{21}
   - 78% chance of being ~~NOT cat~~ NOT H_{21}???

Sigmoid doesn't work in multi-class classification

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \text{ where } K = \#\text{ of classes}$$

Why the need for e^z?



	z	e^{z_i}	$softmax(z_i)$
	2.33	10.27794	0.83827
	-1.46	0.23224	0.01894
	0.56	1.75067	0.14279
$\sum_{j=1}^{K} e^{z_j}$ 👉		**12.26085**	**Total: 1.00000**

Congratulations on knowing your second activation function!

Remember its name: Softmax

Also remember and UNDERSTAND the how and why.

It's used only in the output layer, and only for multi-class classification.

```
keras.layers.Dense(number_of_classes, activation='softmax')
```

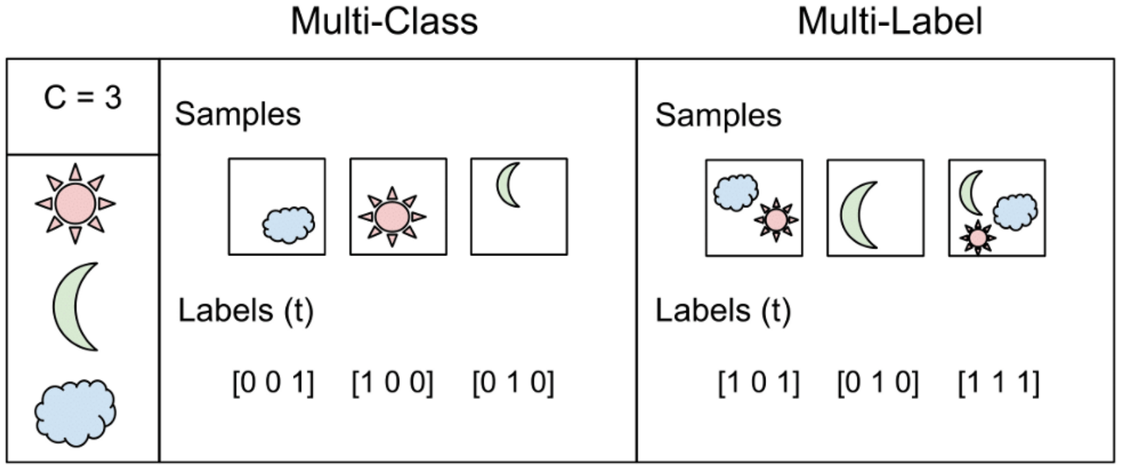
❓ For Php50 load: What will happen if you use *softmax* as the activation function for the output layer of a binary classifier? **WHY?** + *What activation function should we use instead?*

~~Binary~~ Categorical Cross Entropy (CCE)

Binary = *"2 classes"*, Categorical = *"multiple classes"*

From the Keras docs:

> Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided in a `one_hot` representation. If you want to provide labels as integers, please use `SparseCategoricalCrossentropy` loss.



Can single-label data also be one-hot encoded?

Sparse Categorical Cross Entropy (SCCE)

mnemonic: sparse = *"fewer information"*

Also from the Keras docs:

> Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided as integers. If you want to provide labels using `one-hot` representation, please use `CategoricalCrossentropy` loss.

TL:DR: look at the shape of the y labels:

- If y looks like this:

  each row is an array of 1s and 0s

  $$y = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

  use **CCE**

- if y looks like this:

  each row is JUST AN INTEGER

  $$y = \begin{bmatrix} 4 \\ 3 \\ 0 \\ 4 \end{bmatrix}$$

  use **SCCE**

/ Keras Converts SCCE …into CCE

Why do you think so?

```python
y_true = np.array([
    [1],
    [2]
])

y_pred = np.array([
    [0.04, 0.95, 0.01],
    [0.1, 0.8, 0.1],
])

scce = keras.losses.SparseCategoricalCrossentropy(
    reduction=tf.keras.losses.Reduction.NONE
)
scce(y_true, y_pred).numpy()
# array([0.05129329, 2.30258512])
```

```python
y_true_onehot = np.array([
    [0, 1, 0],
    [0, 0, 1]
])

cce = keras.losses.CategoricalCrossentropy(
    reduction=tf.keras.losses.Reduction.NONE
)
cce(y_true_onehot, y_pred).numpy()
# array([0.05129329, 2.30258512])
```

CCE vectorized cost function: $J = y \odot \log(p)$

One of the few times `A * B` is

$\odot$ means *Hadamard* (element-wise) product,
and p is a matrix of softmax probability of y_{pred}

	correct prediction	wrong prediction
y_{true}	$[0, 1, 0]$ *index 1*	$[0, 0, 1]$ *index 2*
y_{pred}	$[0.04, 0.95, 0.01]$	$[0.1, 0.8, 0.1]$
p	$[0.22, 0.56, 0.22]$	$[0.25, 0.50, 0.25]$
$-\log(p)$	$[3.22, 0.05, 4.61]$	$[2.30, 0.22, 2.30]$
$-y \odot \log(p)$	0.05 *(small penalty)*	2.3 *(big penalty)*

Congratulations on knowing your third loss function!

Sparse Categorical Cross Entropy

Another long and scary name to remember!

It's actually similar to Binary Cross Entropy, but for multi-class classification.

```
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # others ...
)
```

It even has a similar formula:

$$J = -y \odot \log(p)$$

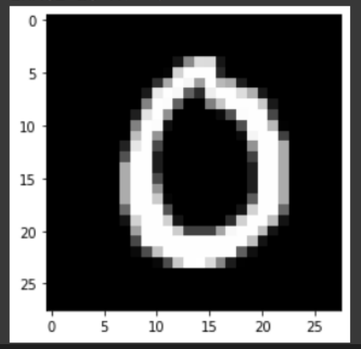
ML Model Code

```
# we can normalize pixel brightness w/o MinMaxScaler
training_images_norm  = training_images / 255.0
test_images_norm = test_images / 255.0

model = tf.keras.models.Sequential([
    keras.layers.Flatten(), # 🧘🌞💡
    keras.layers.Dense(128, activation='sigmoid'),
    keras.layers.Dense(10, activation='softmax') # 🌞
])

model.compile(
    optimizer=keras.optimizers.SGD(),      # 👇🔻🌞
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

# you should get around 89% accuracy w/ just 5 epochs
model.fit(
    training_images_norm, # used to call this `X_norm`
    training_labels,      # used to call this `y`
    epochs=5
)
```

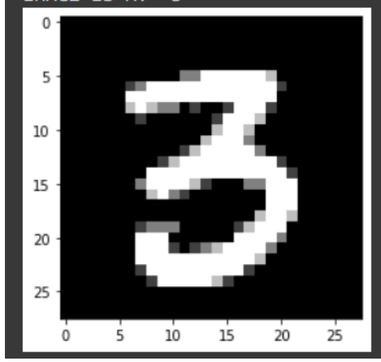
```
# Evaluate the model on unseen data, see how many
# `test_labels` it predicts correctly.  I got ~89%
model.evaluate(test_images_norm, test_labels)

np.set_printoptions(precision=4, suppress=True)
def predict(index):
    plt.rcParams['figure.figsize'] = (4, 4)
    print(f'TEST LABEL: {test_labels[index]}', '\n')
    plt.imshow(test_images[index], cmap='gray')

    single_image = np.array([
        test_images[index] # just 28 x 28 (2D array)
    ]) # need to be a (something x 28 x 28) matrix

    probabilities = model.predict(single_image)
    print('PROBABILITIES: \n', probabilities)

    # np.max → highest probability like 0.998, def.
    # not what we want, we're after THE INDEX (0-9)
    print('IMAGE IS A: ', np.argmax(probabilities))

# try in separate notebook cells
predict(4241)
predict(5555)
predict(1319)
predict(9888)
```

Did you get these results?

```
1  predict(4241)

TEST LABEL: 0

PROBABILITIES:
 [[0.9999 0.     0.     0.     0.     0.     0.0001 0.     0.     0.    ]]
IMAGE IS A:  0
```

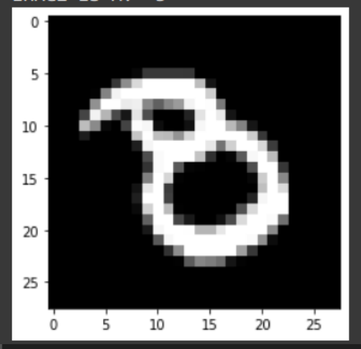


```
1  predict(5555)

TEST LABEL: 3

PROBABILITIES:
 [[0.     0.     0.     0.9998 0.     0.0002 0.     0.     0.     0.    ]]
IMAGE IS A:  3
```

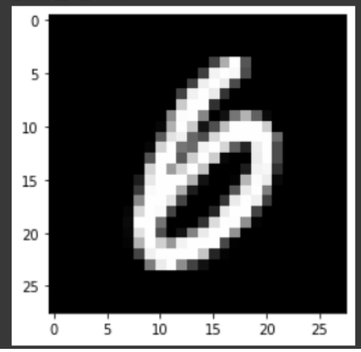


…and these mistakes?

```
1  predict(1319)

TEST LABEL: 8

PROBABILITIES:
 [[0.0679 0.0004 0.009  0.8725 0.     0.0105 0.0075 0.0007 0.0299 0.0015]]
IMAGE IS A:  3
```



```
1  predict(9888)

TEST LABEL: 6

PROBABILITIES:
 [[0.4509 0.     0.0104 0.0196 0.0002 0.0049 0.282  0.     0.2301 0.0021]]
IMAGE IS A:  0
```



Congratulations on making your second ML-powered "app"!

Handwritten Digits Recognition

using vanilla neural networks

> It’s said to be the *"Hello World"* problem in the machine learning industry.

*There's some spoonfeeding here and there (dataset, train-test split), and that it doesn't actually read **YOUR OWN** handwritten digits yet ☺... but **it's a great achievement!***

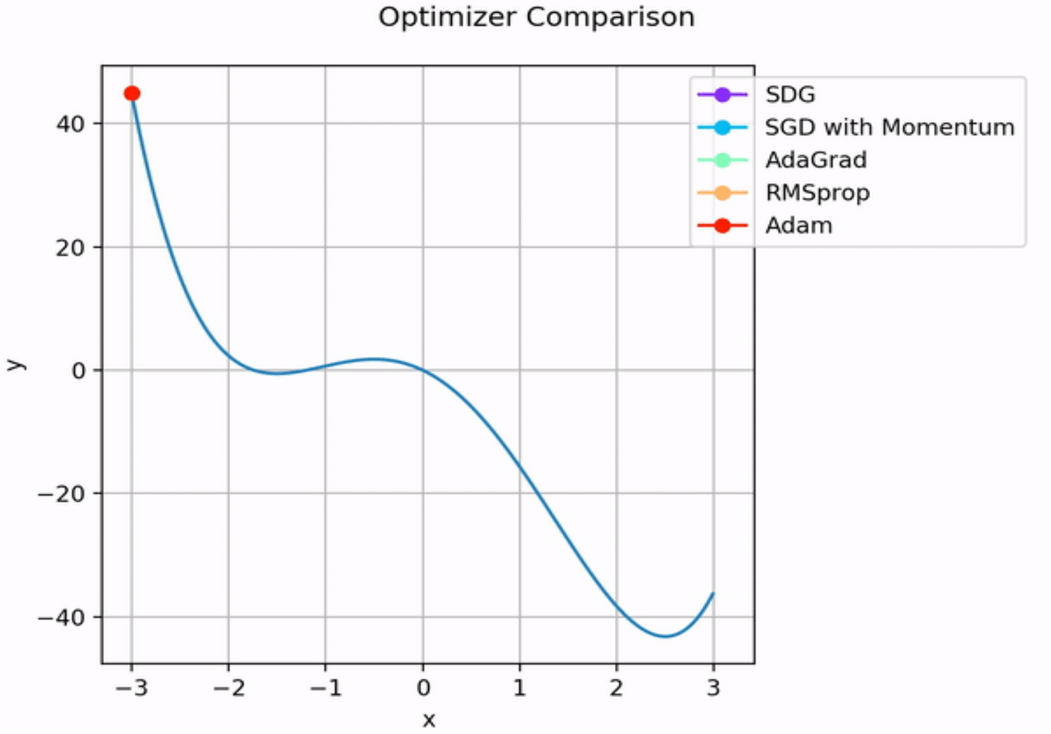
BTW: here’s a demo. Make sure your handwriting is quite centered, just like the training samples.

Converge Faster with Adam

Adaptive moment estimation

- an extension to **stochastic gradient descent**
- has recently seen broader adoption in
  - **computer vision** and
  - natural language processing
- by *Diederik Kingma* from OpenAI and *Jimmy Ba* from the University of Toronto
- 2015 ICLR (*International Conference on Learning Representations*) paper *"Adam: A Method for Stochastic Optimization"*
- SDG maintains a single learning rate (termed α)
  - learning rate does not change during training.
- Adam learning rate is maintained for each network weight and separately adapted as learning unfolds.

Change the optimizer from `SGD` to `Adam`. What happened to your accuracy after 5 epochs?

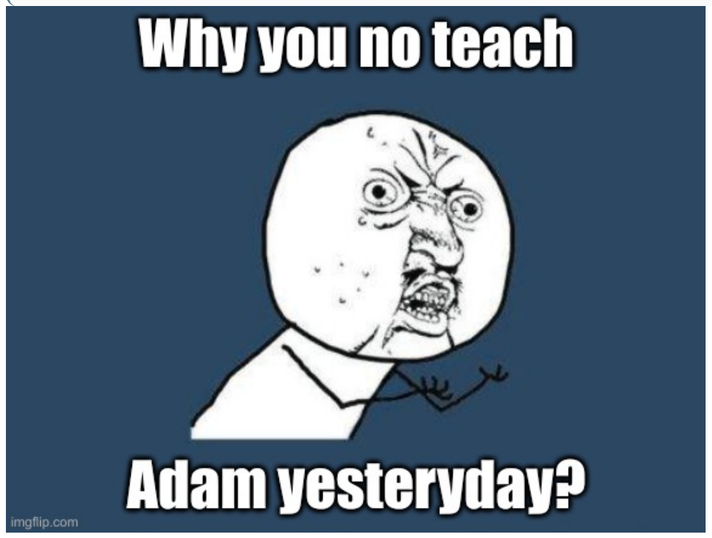


Many other algorithms have been made on top of gradient descent

- Adagrad, RMSprop, **Adam**, etc.

Gradient Descent is the backbone of every machine learning algorithm and it also acts as a base for many deep learning optimizers.

However, after a while people started noticing that despite superior training time, Adam in some areas does not converge to an optimal solution, so for some tasks (such as image classification on popular CIFAR datasets) state-of-the-art results are still only achieved by applying SGD with momentum.



❝ More than that Wilson et. al. showed in their paper ‘The marginal value of adaptive gradient methods in machine learning’ that adaptive methods (such as Adam or Adadelta) do not generalize as well as SGD with momentum when tested on a diverse set of deep learning tasks, discouraging people to use popular optimization algorithms. A lot of research has been done since to analyze the poor generalization of Adam trying to get it to close the gap with SGD. ❞

So many optimizers 🤯🤯🤯🤔🤔🤔🥶🥶🥶

The 🍦❄ chillest explanation of their gist on Youtube ⏳ 7m 22s

Optimizers

Gradient Descent:

$$\theta = \theta - \alpha \,.\nabla_\theta J(\theta)$$

Stochastic Gradient Descent

$$\theta = \theta - \alpha \,.\nabla_\theta J(\theta; sample)$$

Mini-Batch Gradient Descent

$$\theta = \theta - \alpha \,.\nabla_\theta J(\theta; N\ samples)$$

SGD + Momentum

$$v = \gamma . v + \eta . \nabla_\theta J(\theta)$$
$$\theta = \theta - \alpha\, v$$

SGD + Momentum + Acceleration

$$v = \gamma . v + \eta . \nabla_\theta J(\theta - \gamma . v)$$
$$\theta = \theta - \alpha\, v$$

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

Adadelta

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

Adam

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \times \mathrm{E}[g_{t,i}]$$

Congratulations on knowing your second optimizer!

Adam

⚠ Just remember the best optimizer is the one that the one that can traverse the loss of your problem pretty well.

Just because it's fast doesn't mean it won't have problems.

And the code...

```
model.compile(
    optimizer = tf.optimizers.Adam(),
    # others to follow...
)
```

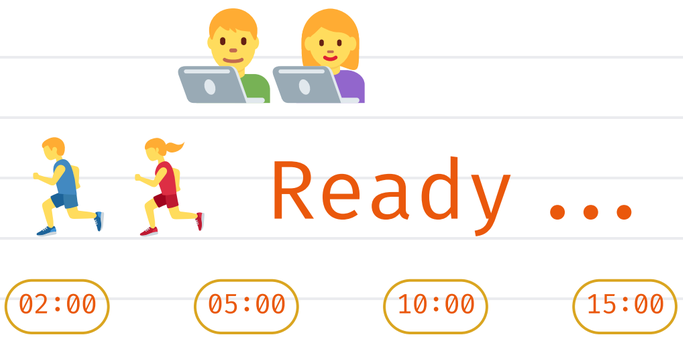
Your turn: Fashion MNIST

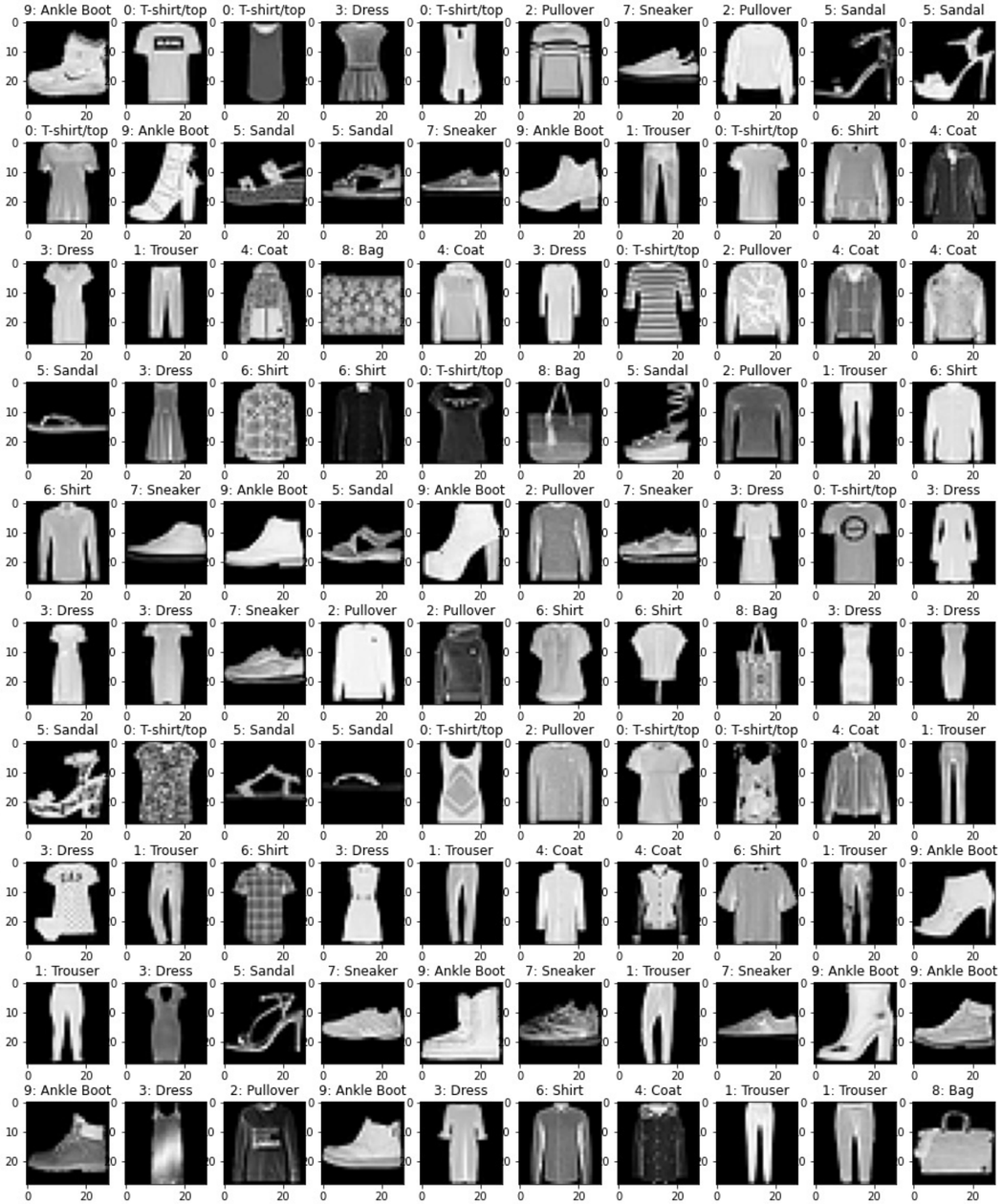
MNIST w/ a twist: grayscale images of fashion items

```
mnist = keras.datasets.fashion_mnist
```

- also 60k training images, 10k test images, also 28×28
- data is still labeled from 0-9 (matrices + strings = 💥)
- ✏️ code to convert 0-9 labels into string labels

LABEL	DESCRIPTION
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot





👩‍🔬👩‍🔬🧪 Experiments

Notebook: TBA

- Try breaking your code. What happens if you…
  1. try to remove the `Flatten()` layer?
  2. specify only `5` neurons in the output layer?

Try to put `%%time` before each code cell, so that you can note how long training takes with different combinations of hyperparameters.

```
%%time
train(hidden_layers_activation='sigmoid')
```

You should see the following added to each output cell:

```
CPU times: user 30.2 s, sys: 3.66 s, total: 33.8 s
Wall time: 28.9 s
```

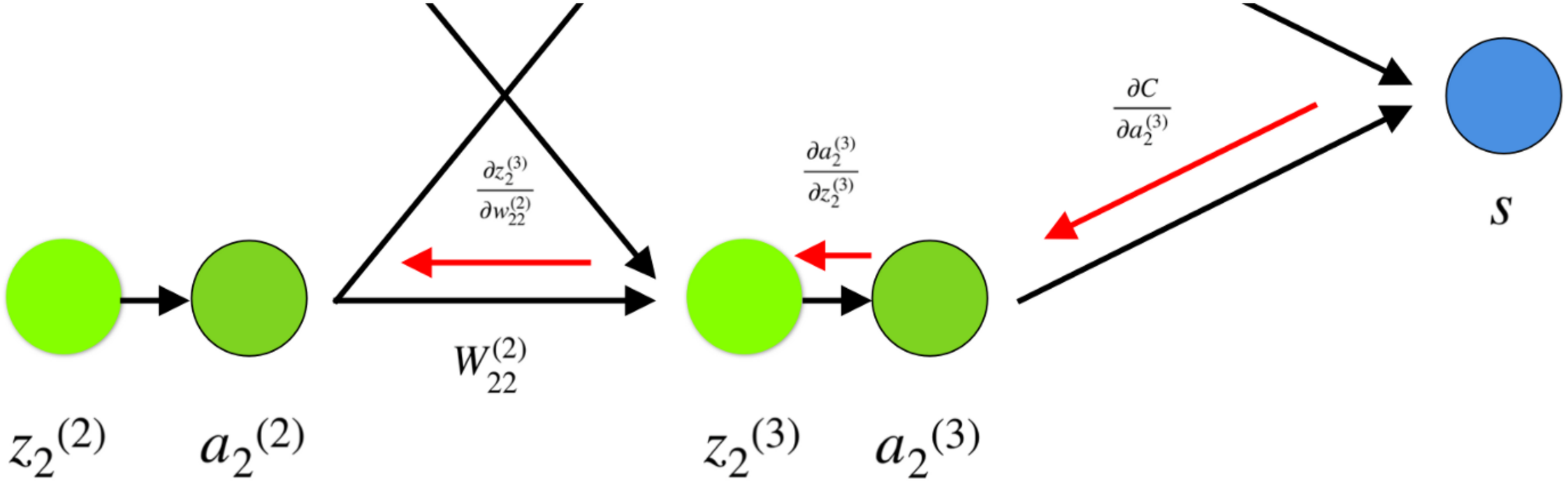
Fashion MNIST are more *"complicated"*, so lower accuracy compared to *"digits"* MNIST is understandable.

- Try reconfiguring NN hyperparameters, while making observations on training accuracy, testing accuracy, and training time. 🧐 *for thought: What's 0.5% of 60,000, and of 10,000?*
  1. Change the optimizer from `SGD()` to `Adam()` and vice versa in any of the combos below.
  2. Change the activation from `sigmoid` to `relu`.

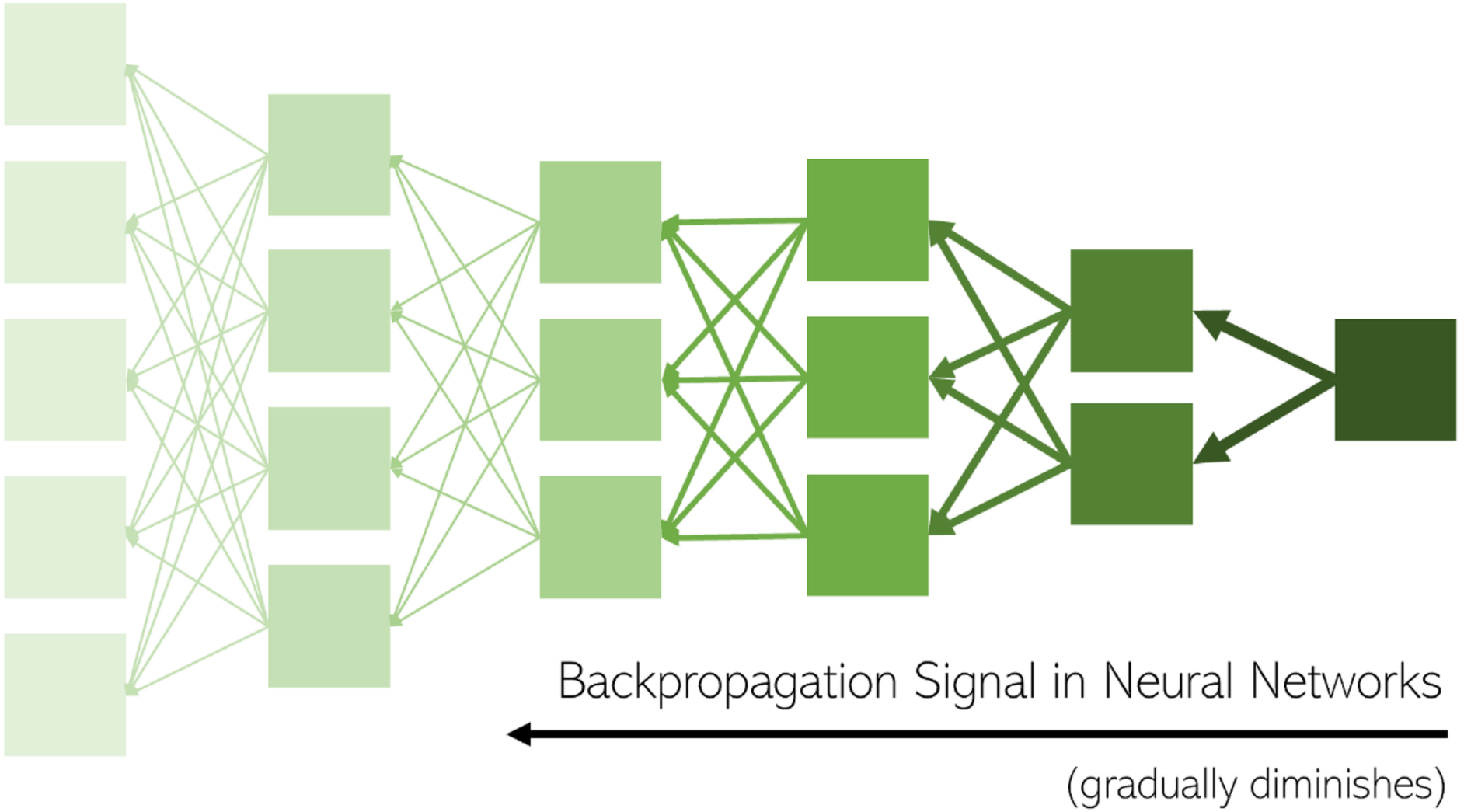
     What's that? 👆❓😮😱
  3. Try increasing the number of hidden layer neurons to 512, and also 1024.
  4. Try adding another hidden layer with 128 neurons.
  5. Try increasing the epochs to 15, and to 30.
  6. Try turning off feature normalization.

Backpropagation

- As a simple analogy:
  1. From the output layer (***the desired state***), I have this image, labeled as 2.
  2. What can I do to increase the weights of those related to 2, and decrease the ones that aren't related?
     - such that I will make less mistakes when shown a 2 in the future?
  3. Then repeatedly work backwards towards the previous layer so they too, adjust



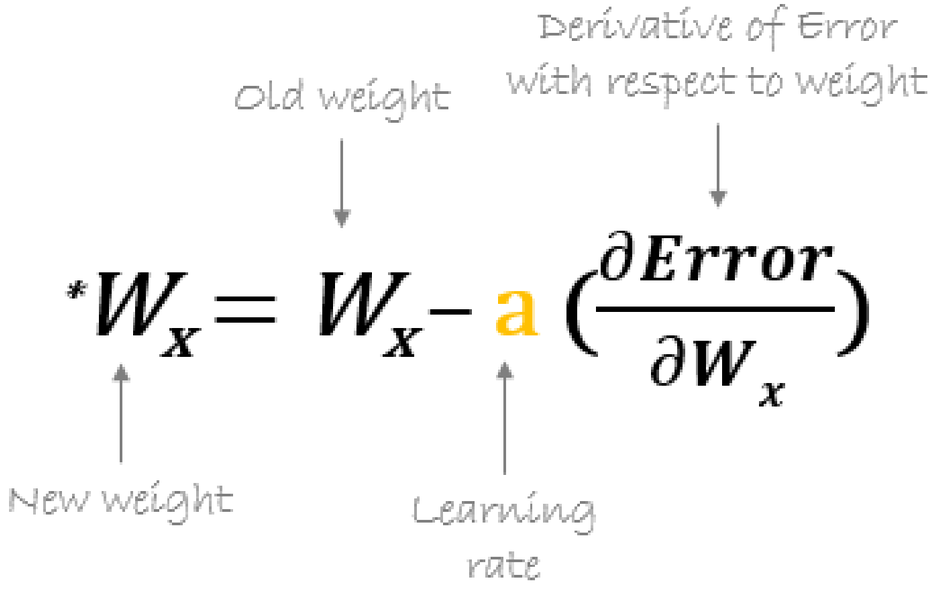
Vanishing Gradient can be a Problem in **Deep** Neural Nets

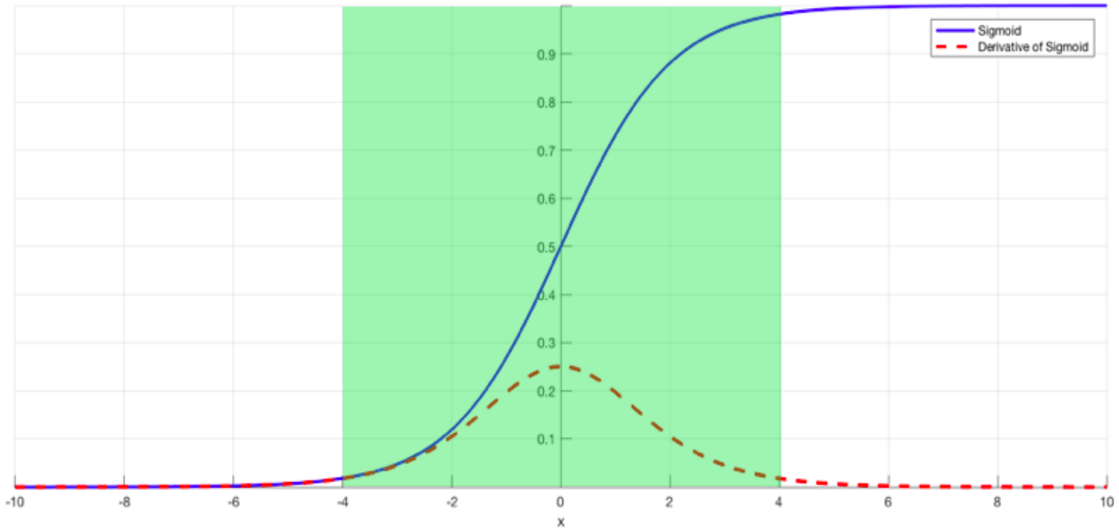


Vanishing Gradient

The best explanation on the Web 👇👇!

- What will happen if the derivative term in the equation is too small, i.e. almost zero?
- We can see that a very small derivative would update or change the value of W_x only by a minuscule amount
- hence, ${}^*W_x$ would be almost equal to the W_x.
- **No change in the weights means no learning.**
- The weights of the initial layers would continue to remain unchanged (or only change by a negligible amount)
  - no matter how many epochs you run
- The graph of sigmoid and its derivative shows that the safe zone is around -5 to $+5$, then it *"saturates"*.
- the maximum possible value for $\sigma'(z)$ is 0.25 when



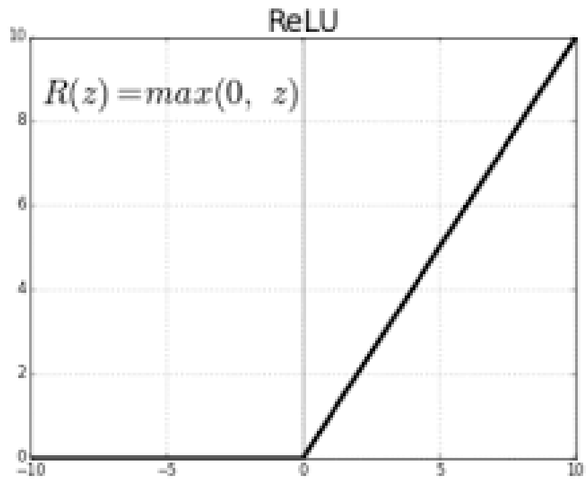


Rectified Linear Unit (ReLU)

a.k.a. "another simple formula with a scary name"

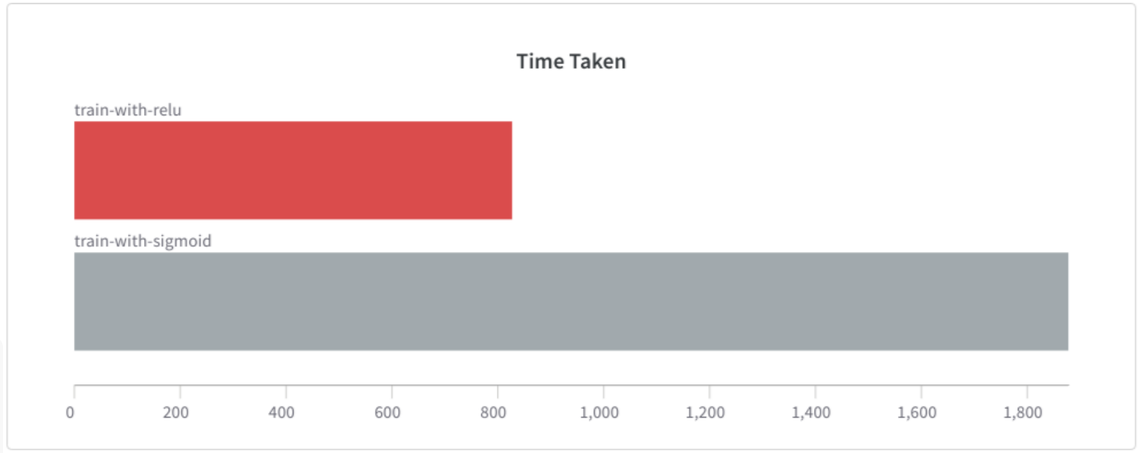
$$R(z) = \begin{cases} 0, & \text{if } z > 0 \\ z, & \text{otherwise} \end{cases} = max(0, z)$$

```
1   def relu(z): return x if z > 0 else 0 # OR
2   def relu(z): return max(0, z)
```

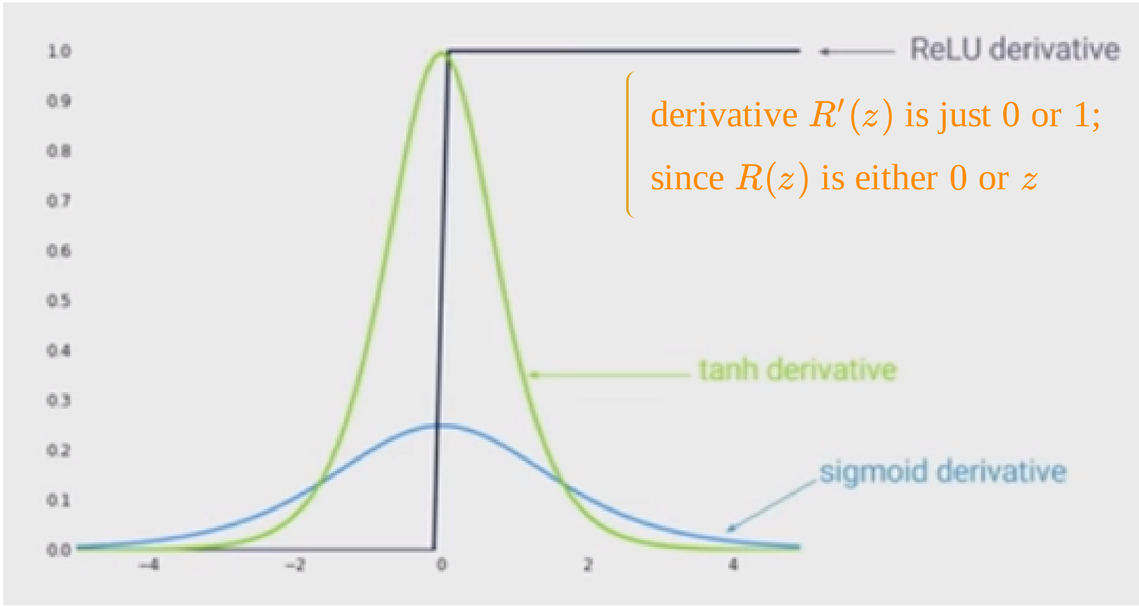


- has become the default activation function for the **hidden layer** of many neural networks. (not all)

1. ReLU is more lightweight computationally vs $\frac{1}{1+e^{-z}}$



2. Less chances of vanishing gradient



Congratulations on knowing your third activation function!

Remember its acronym: ReLU

We only use it in the hidden layer.

Don't forget about our old friend sigmoid. It's is still a valid activation in some types of NN like RNNs.

Sigmoid is also a great activation function for the output layer of a binary classifier.

```
# other layers...,
keras.layers.Dense(activation='relu'),
# other layers...
```

Convolutional Neural Nets

From Wikipedia:

> In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of Artificial Neural Network(ANN), most commonly applied to analyze visual imagery.

- CNNs are often used in image recognition systems.
- In 2012 an error rate of 0.23% on the MNIST database was reported.
- When applied to facial recognition, CNNs achieved a large decrease in error rate.
- Another paper reported a 97.6% recognition rate on "5,600 still images of more than 10 subjects".
- In 2015 a CNN demonstrated the ability to spot faces from a wide range of angles, including upside down, even when partially occluded, with competitive performance…

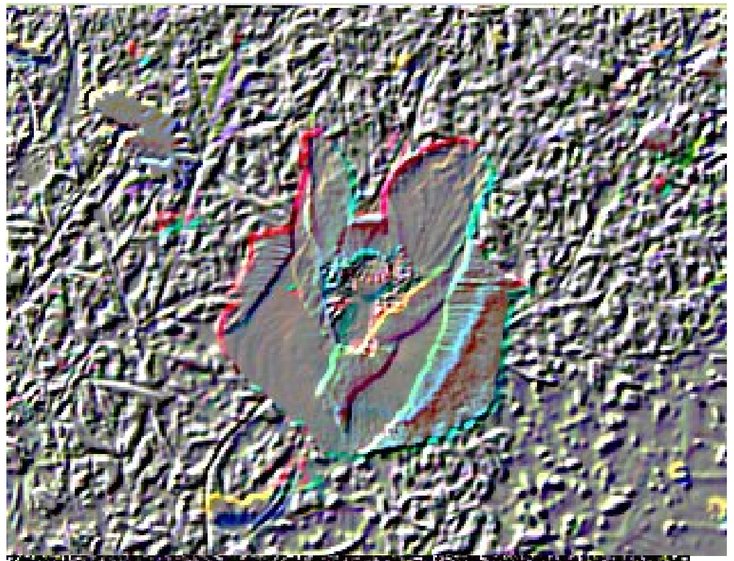
*or **CNN** for short*

- involves two special layers:
  - **convolutional layer**: the core building block of a CNN, and it is where the majority of computation occurs. It essentially applies a **filter** (or a kernal) to an image
    - GOAL: highlight important parts of the image (***high-level features***)
    - e.g. for a flight of stairs, your brain *"kinda ignores"* its color and materials.
      - but do they look like *"rectangles"* or *"bars"* stacked on top of each other?
  - **pooling layer**: also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input.
    - GOAL: reduce the size of the convolved image

Convolutional Layer Filters …are like Photoshop Filters

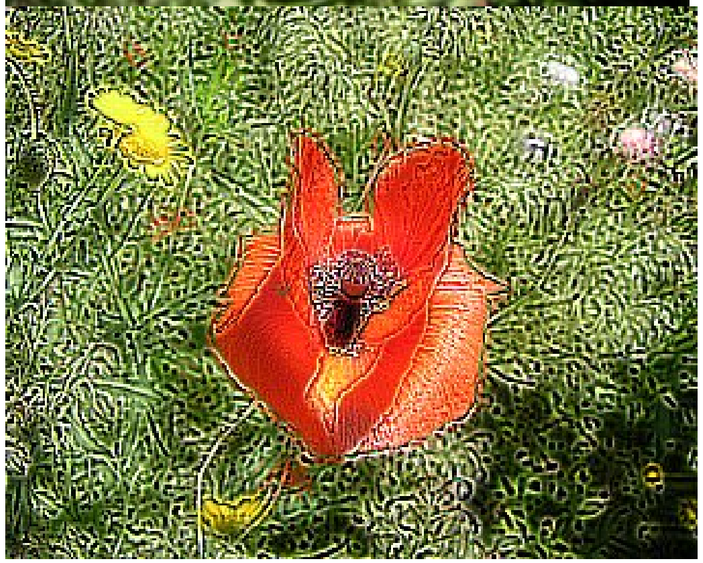
$$\begin{bmatrix} 0 & 0.2 & 0 \\ 0.2 & 0.2 & 0.2 \\ 0 & 0.2 & 0 \end{bmatrix}$$

Soft blur



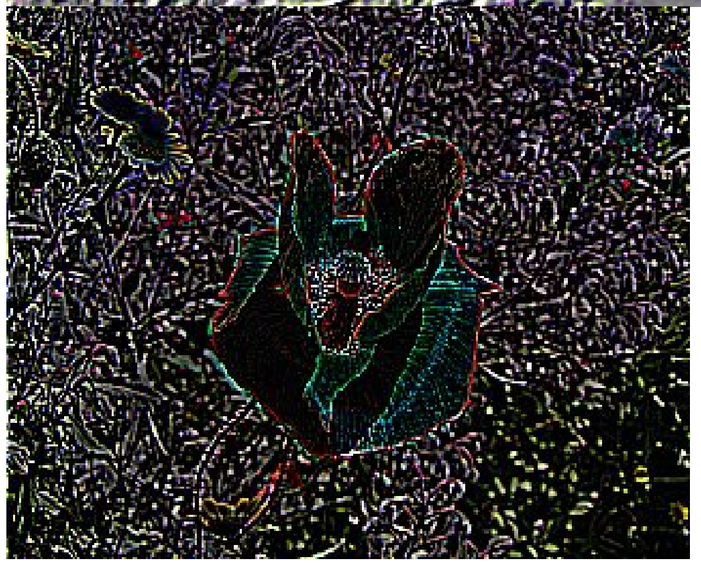
$$\begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Emboss



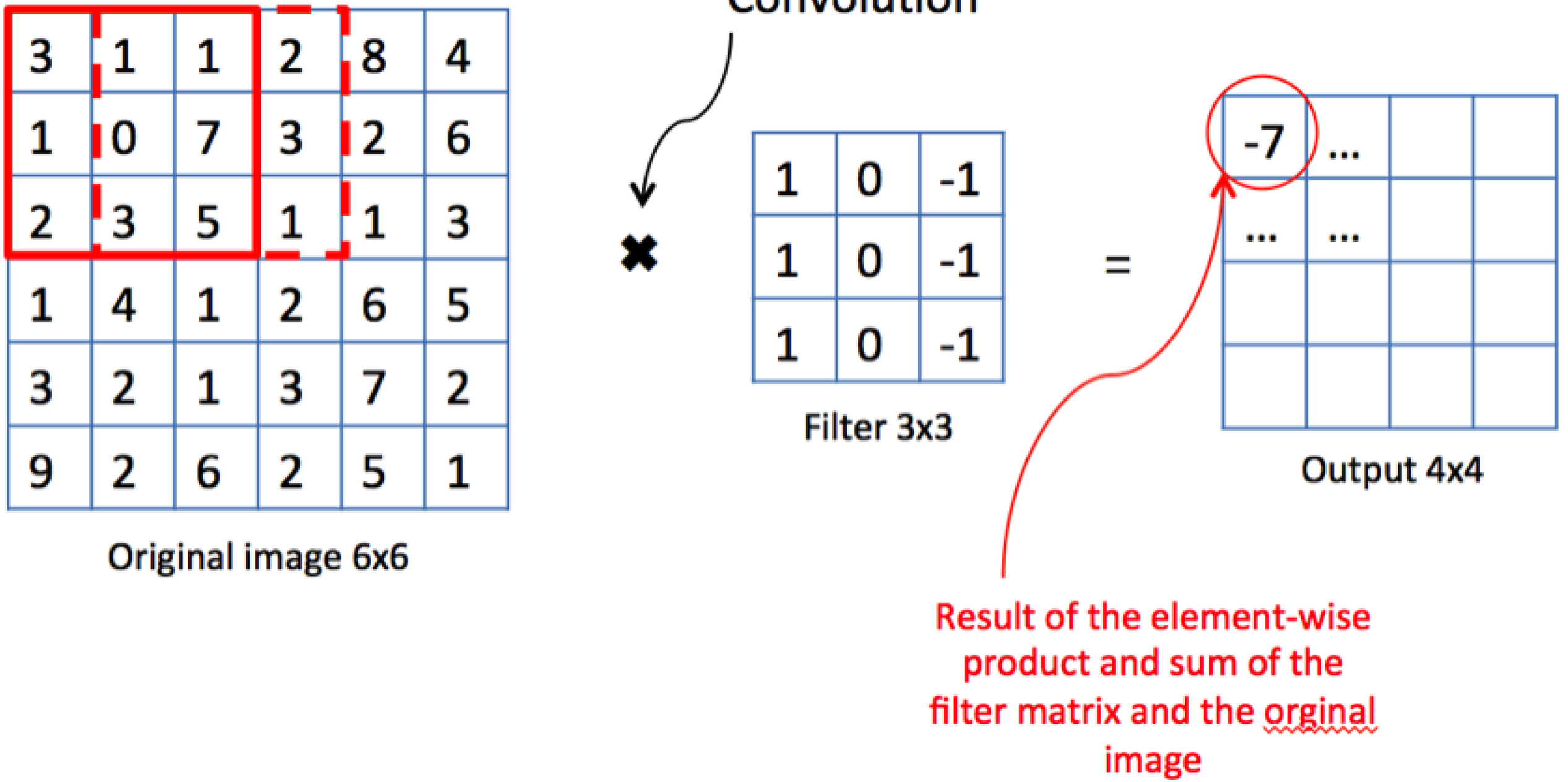
$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -7 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Excessive sharpen

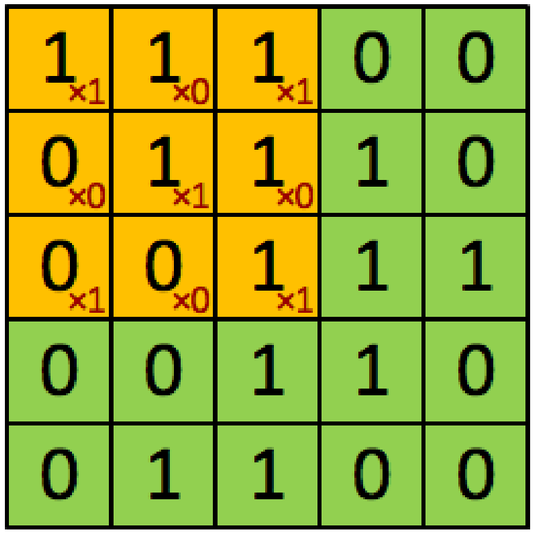


$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

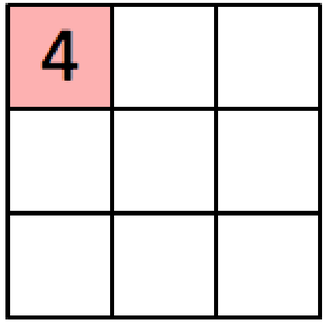
Detect Edges


"Convolution"
3 1 1 2 8 4
1 0 7 3 2 6
2 3 5 1 1 3
1 4 1 2 6 5
3 2 1 3 7 2
9 2 6 2 5 1
Original image 6x6
1 0 -1
1 0 -1
1 0 -1
Filter 3x3
=
-7 ...
... ...
Output 4x4
Result of the element-wise product and sum of the filter matrix and the orginal image

GIF showing the process



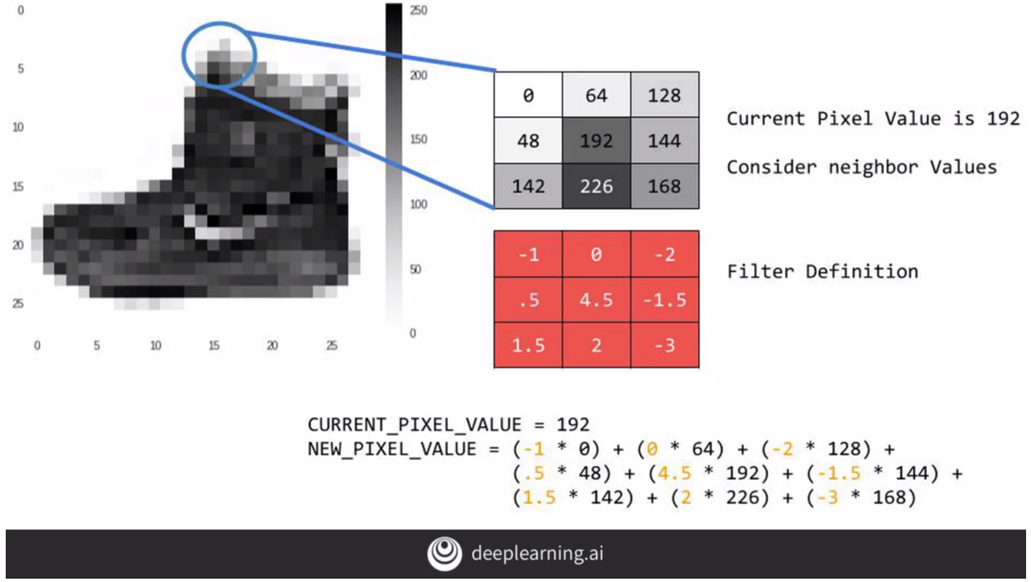
Image



Convolved Feature

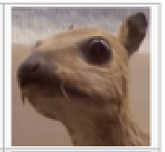
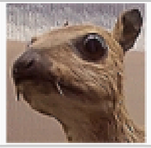
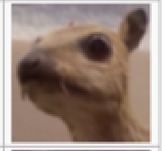
The filter matrix used here is: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$

- get the middle pixel, and get **its** new value by getting a summation of the pixel values of **that pixel** as center
  - along w/ its 8 neighboars, multiplied element-wise $\odot$
  - with each element in the filter matrix as *"multiplier"*

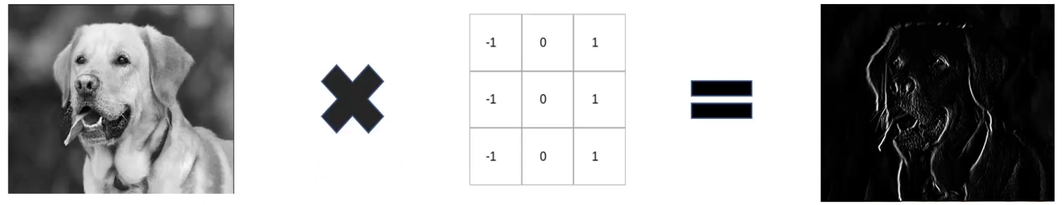


- we don't have eight neighbors around the edges
  - (e.g. `image[0][0]` only has 3 neighbors ➡️↘️⬇️)
- in effect, the convolved image loses its *"1 pixel margin"*
  - ❓❓ how much margin is lost for 5×5 filters?

Sample filters, again

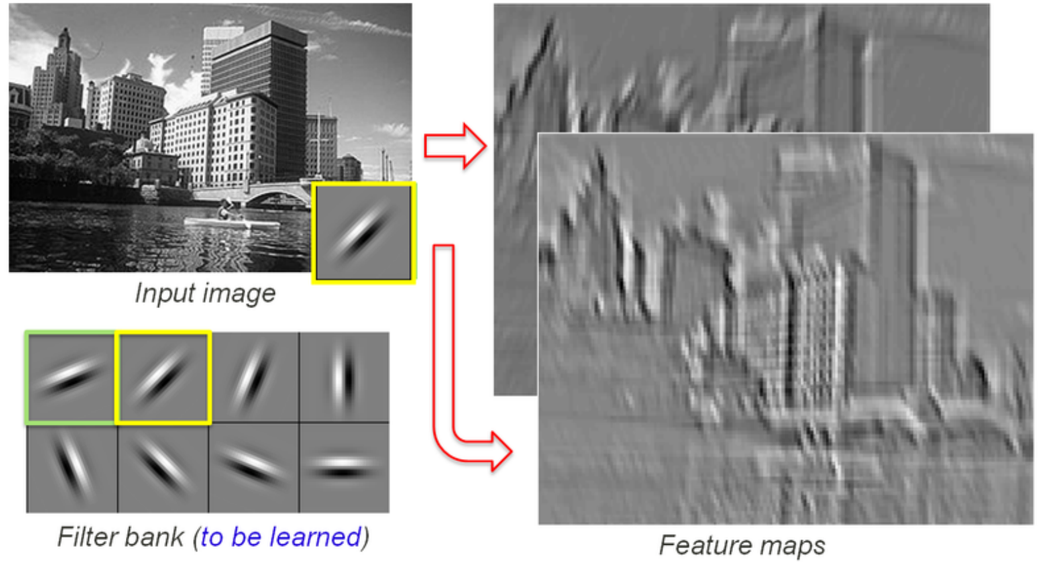
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

- Notice how the furs of the dog and even the background disappears, leaving only the 🐕's *"silhouette"*

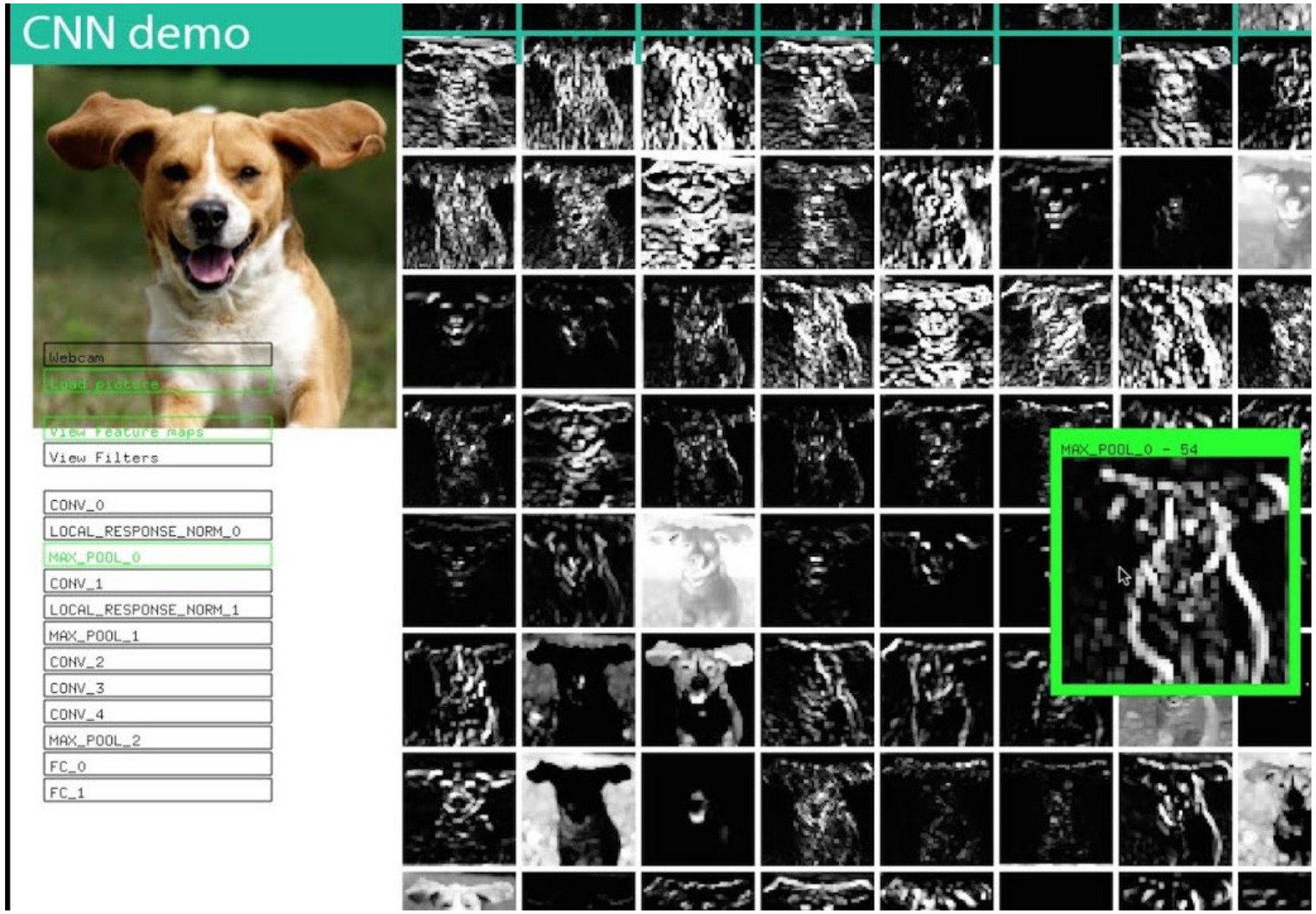


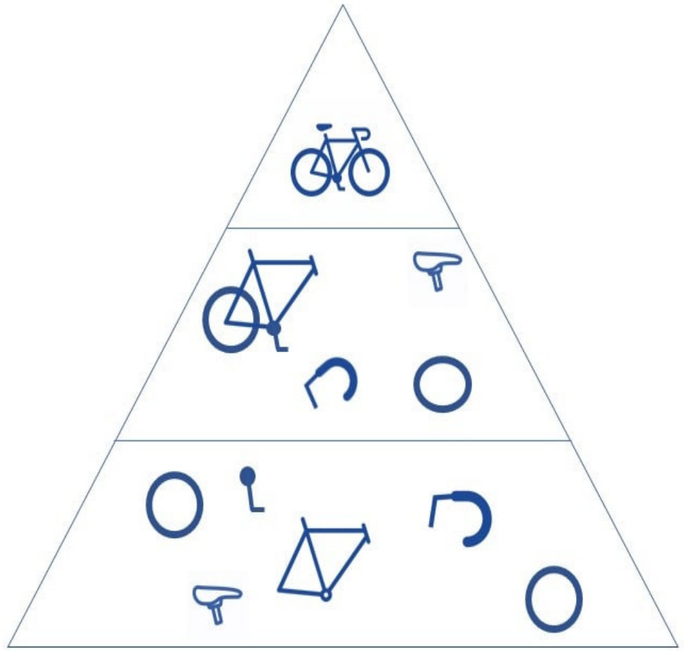
Original Image | Filter – Right Sobel | Extracted Feature – Vertical lines

- Another example:



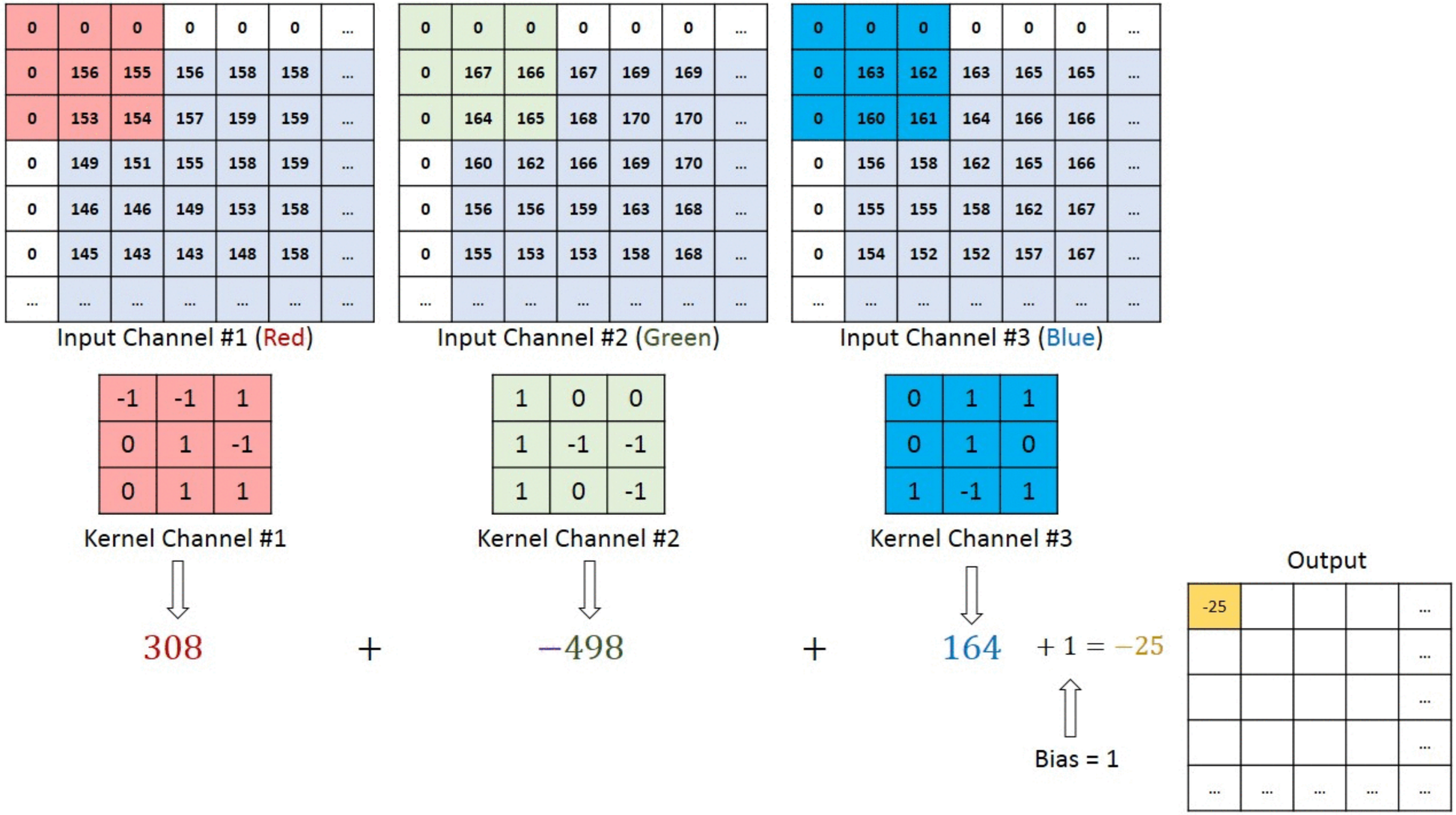
/ Keras tries out a bunch of these…





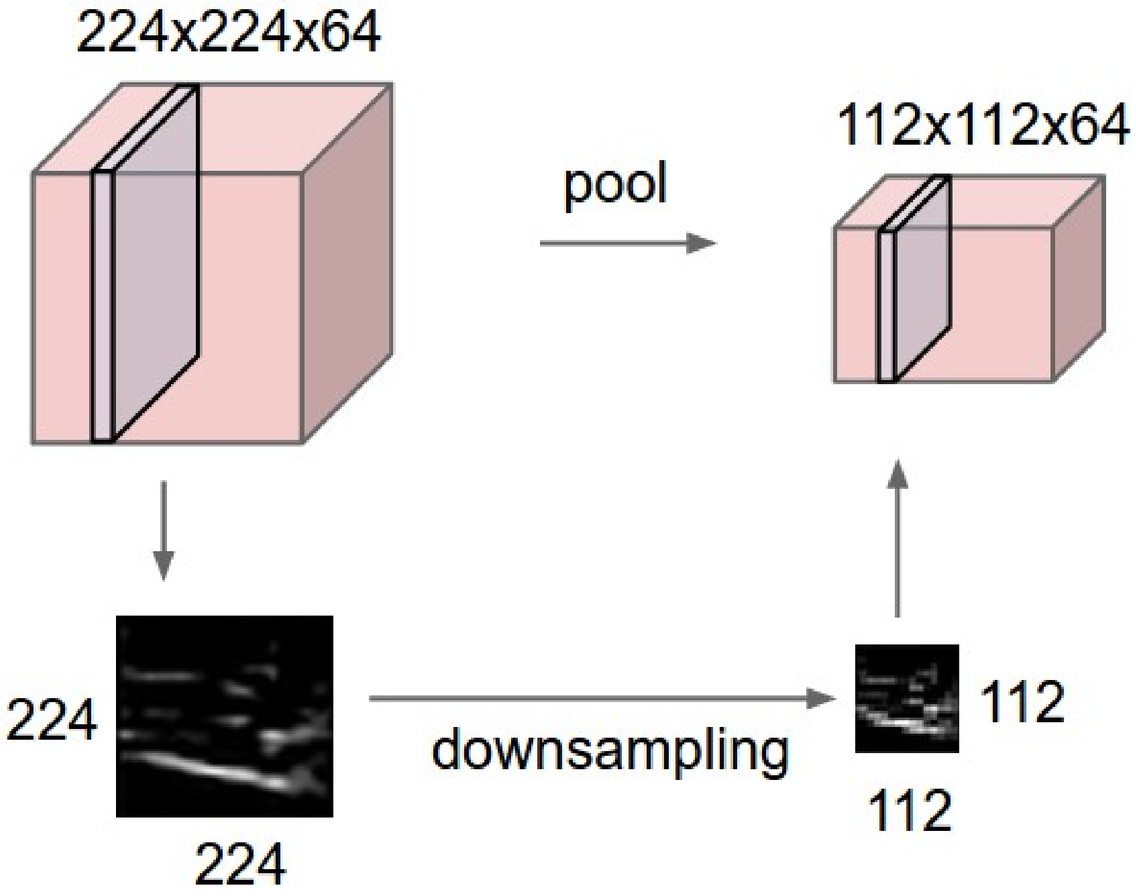
To learn what makes an image a *"dog"* or a *"bike"*

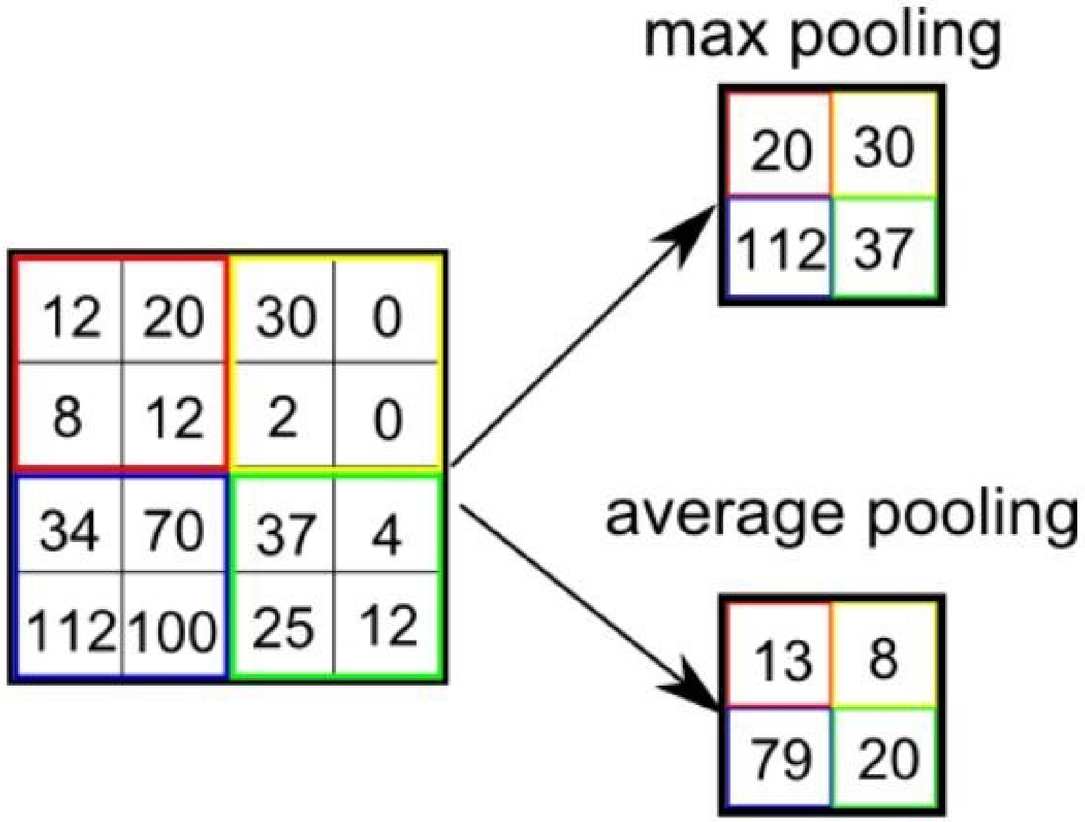
CNN GIF for colored images



Pooling Layer

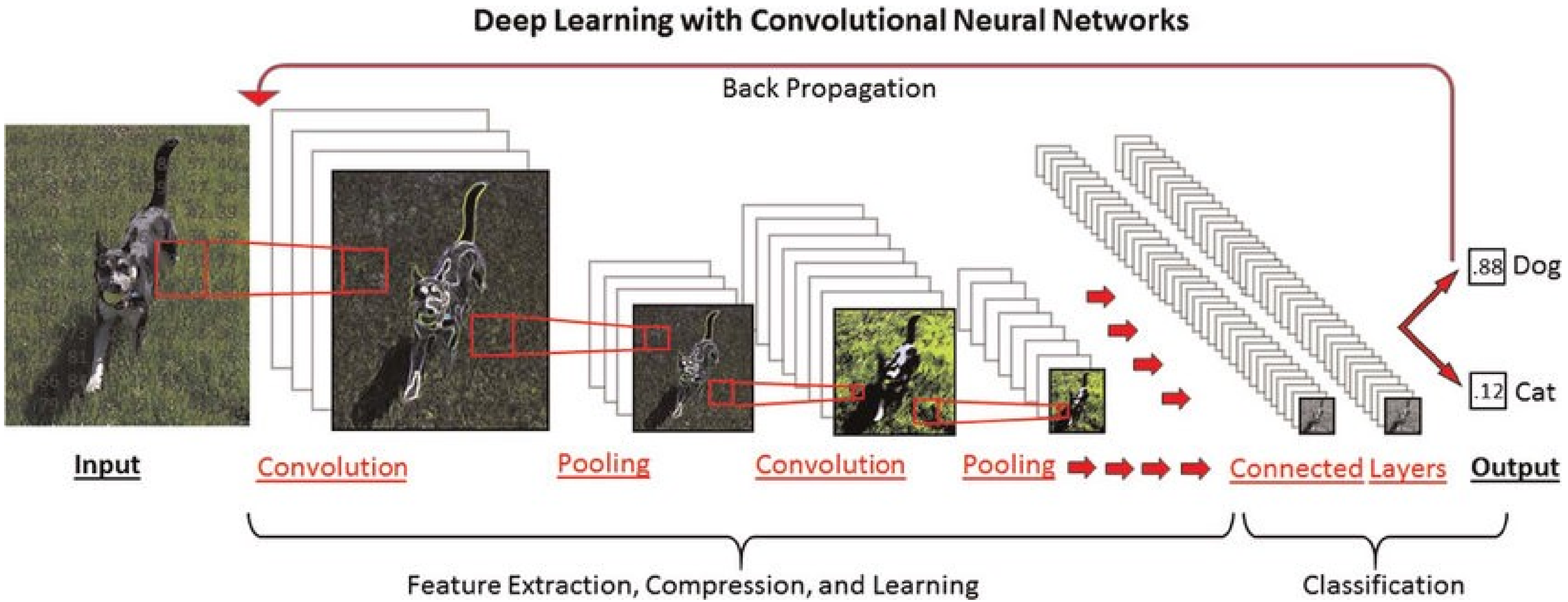
so simple a single slide explains it all 🐕





- two kinds of pooling layers:
  - max: get the brightest pixel to represent a group of 4
  - average: get the average *"brightness"* for the quartet
- a 2×2 pooling layer quarters your colvolved images
  - 25% input for our `Flatten` and `Dense` layers

Convolutional Neural Network



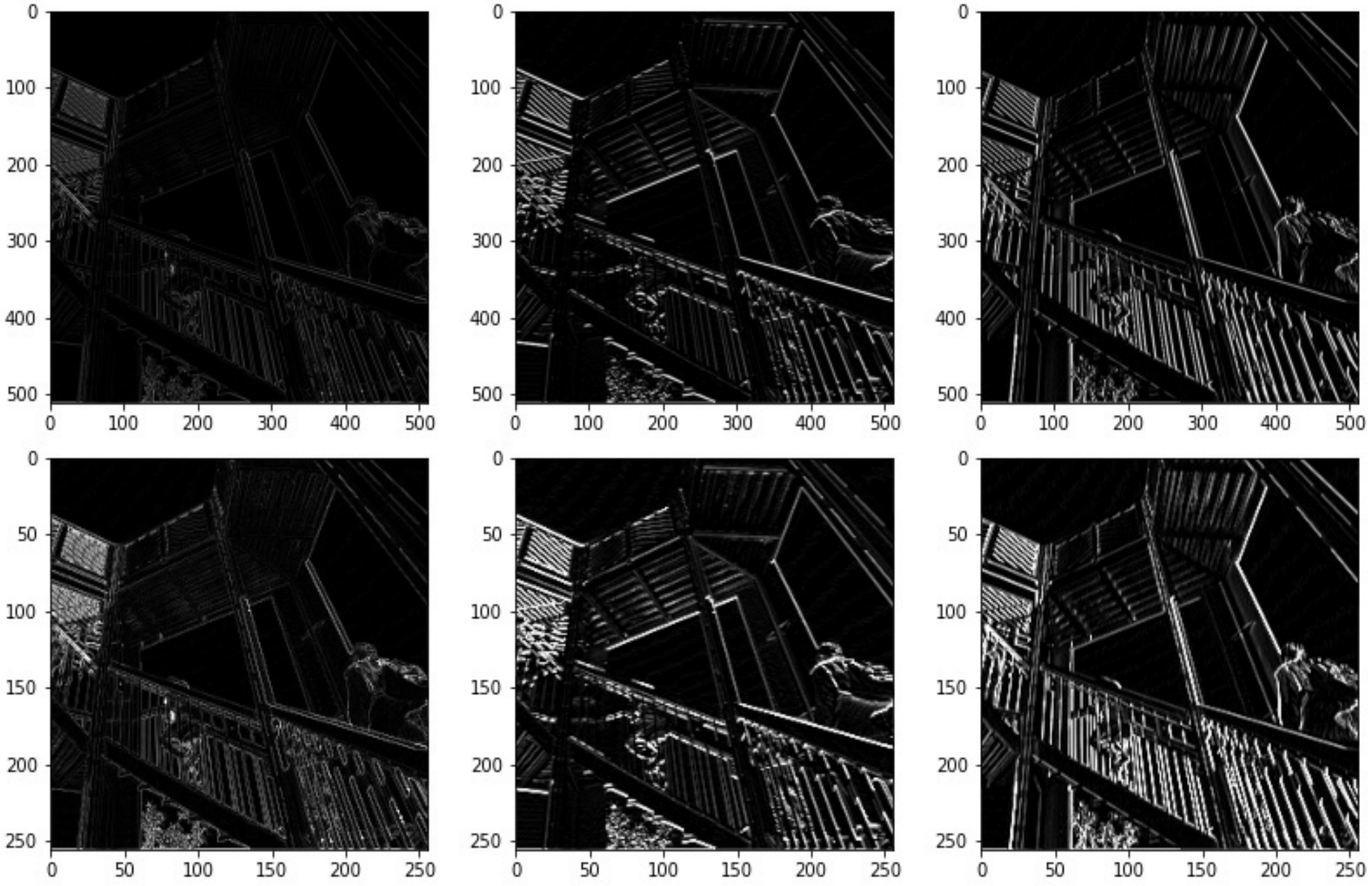
- We can just stack as many `Convolution` + `Pooling` layers depending on the accuracy needs of our ML problem.
- For TF/Keras, you also insert a `Flatten` layer before your fully-connected layers.

POOL
RELU RELU
CONV CONV
POOL
RELU RELU
CONV CONV
POOL
RELU RELU
CONV CONV
FC
car
truck
airplane
ship
horse

Experiencing convolutions and pooling

Notebook courtesy of Coursera and DeepLearning.AI

https://bit.ly/ece4241-ml-04: ***no TF or Keras code here, just normal coding with*** (although the pic was taken from `scipy`)



👨‍💻👩‍💻 You do it!

Ready ...

02:00 05:00 10:00 15:00

You already have this vanilla neural network:

```
model = keras.models.Sequential([
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

Now add the convolution and pooling layers like so:

```
model = keras.models.Sequential([
    # colored images would be `input_shape=(28, 28, 3)`
    keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D(2, 2),
    keras.layers.Conv2D(32, (3,3), activation='relu'),
    keras.layers.MaxPooling2D(2,2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
]) # what would be the accuracy on `model.fit( ... )` and `model.evaluate( ... )`??
```

Congratulations on knowing your first non-vanilla ANN!

Convolutional Neural Networks (CNN)

You also learned two new layers: Convolution and Pooling

This is one of the goto NNs for image recognition.

🎓 Time to graduate and use our own images

BTW: This is the *"easy"* part

- eventually you want to use your own images
  - unless maybe your app recognizes handbags and numbers
- very uniform
  - objects are all centered.
    - real world images have the *"target"* in different locations
  - The shirts and handbags are upright
    - real pix have weird poses and angles
  - they’re all the same size and aspect ratios
  - they’re also noisy *(contain other objects)*

Using a Training Set from Google Drive

1. Make sure your directory structure is something like this:

```
├── train
│   ├── cats
│   └── dogs
└── validation
    ├── cats
    └── dogs
```

2. Easiest way is to upload this to Google Drive, and have Colab download it. Get its File ID:

```
https://drive.google.com/file/d/1-OsSKExQU7JJP-4PuXYerBA5-rlt_qpB/view?usp=sharing
#                             ^___________THIS ONE___________^
```

3. Download it from Colab using the Colab-exclusive `gdown` command. Write this command on a Jupyter cell. `!` means a Linux command, since the Colab VM is an Ubuntu Linux machine.

```
!gdown --id 1-OsSKExQU7JJP-4PuXYerBA5-rlt_qpB
```

Using your own images

You should see something like this for the output cell.

```
Downloading...
From: https://drive.google.com/uc?id=1-OsSKExQU7JJP-4PuXYerBA5-rlt_qpB
To: /content/cats_and_dogs_filtered.tar.xz
100% 66.9M/66.9M [00:00<00:00, 288MB/s]
```

4. Unzip (or untar in my case) the training samples. Use the `zipfile` module if you used a ZIP file.

```
import tarfile

cats_dogs_tar = tarfile.open('./cats_and_dogs_filtered.tar.xz', 'r')
cats_dogs_tar.extractall('')
cats_dogs_tar.close()
```

5. You can check if the file was really untarred or unzipped using the `ls` command in Linux.

```
!ls -lh cats_and_dogs_filtered
```

Using own images

You should see the extrated folders.

```
total 8.0K
drwxr-x--- 4 1002 1003 4.0K Sep 22  2016 train
drwxr-x--- 4 1002 1003 4.0K Sep 22  2016 validation
```

6. Verify that the folders contain the correct number of files:

```
import os

train_dogs_dir = os.listdir('./cats_and_dogs_filtered/train/dogs')
train_cats_dir = os.listdir('./cats_and_dogs_filtered/train/cats')
test_dogs_dir = os.listdir('./cats_and_dogs_filtered/validation/dogs')
test_cats_dir = os.listdir('./cats_and_dogs_filtered/validation/cats')

len(train_dogs_dir), len(train_cats_dir), len(test_dogs_dir), len(test_cats_dir)
```

```
(1000, 1000, 500, 500)
```

Using your own images

7. Compile your model as usual:

```
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential([
    keras.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(150,150, 3)),
    keras.layers.MaxPooling2D(2, 2),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D(2, 2),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D(2, 2),
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])

model.summary()
```

Using your own images

8. Compiile as usual

```
model.compile(
    loss='binary_crossentropy',
    optimizer=keras.optimizers.Adam(),
    metrics=['accuracy']
)
```

9. The training is a little different. You get labeled images instead of giving `X` and `y` separately:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1/255)
test_datagen = ImageDataGenerator(rescale=1/255)

train_generator = train_datagen.flow_from_directory(
    './cats_and_dogs_filtered/train',
    target_size=(150, 150),
    batch_size=125,
    class_mode='binary'
)
```

Using your own images

```
test_generator = test_datagen.flow_from_directory(
    './cats_and_dogs_filtered/validation',
    target_size=(150, 150),
    batch_size=63,
    class_mode='binary'
)

history = model.fit(
    train_generator,
    steps_per_epoch=8,
    epochs=30,
    validation_data=test_generator,
    validation_steps=8
)
```

10. Determine what number TF assigns to your labels.

```
print(train_generator.class_indices)
```

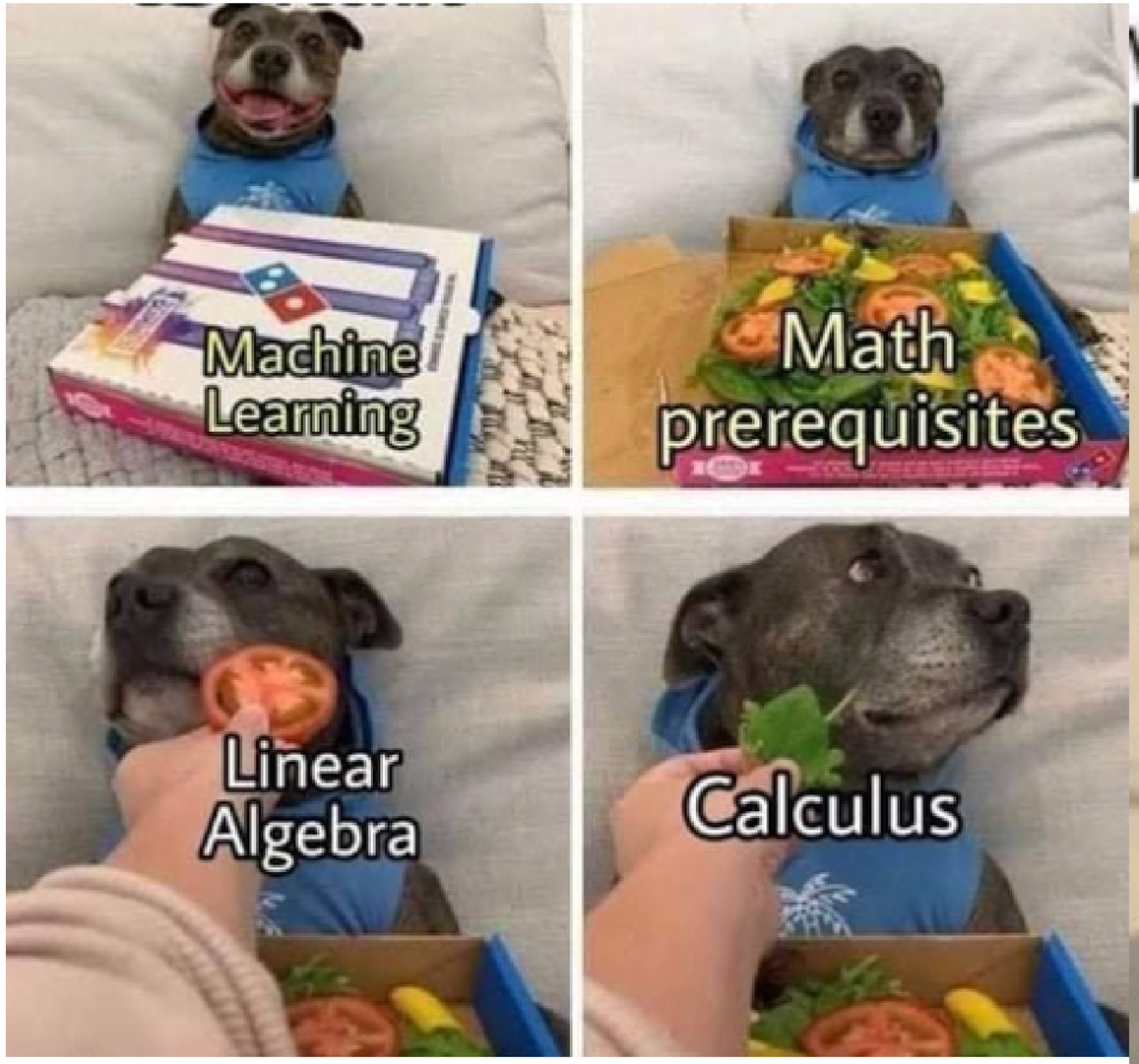
```
{'cats': 0, 'dogs': 1}
```

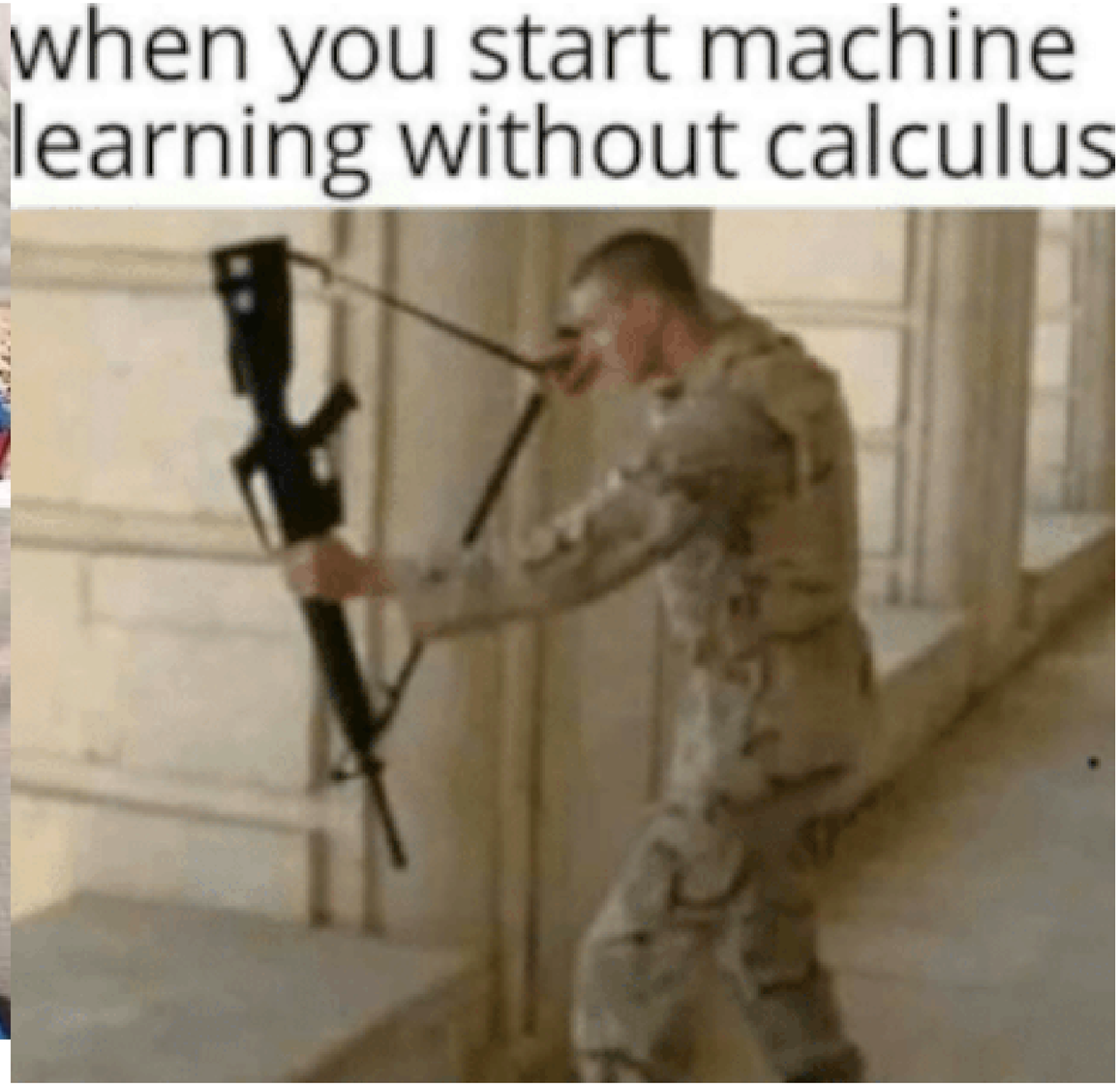
Using your own images:

11. Optional: play with it on Colab. The following code is Colab-exclusive. It won't work on any other Jupyter notebooks.

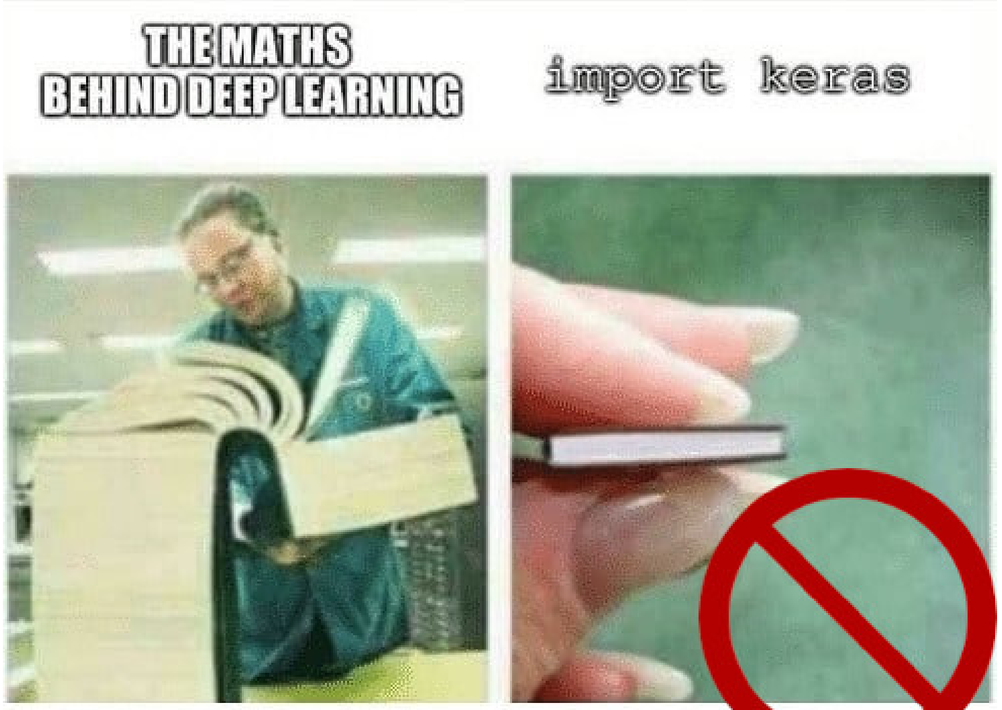
```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for filename in uploaded.keys():
    path = '/content/' + filename
    uploaded_image = image.load_img(path, target_size=(150, 150))
    X = image.img_to_array(uploaded_image)
    X_norm = X / 255
    X_norm_3d = np.expand_dims(X_norm, axis=0)

    classes = model.predict(X_norm_3d, batch_size=8)

    if classes[0] ⩾ 0.5:
        print('doggie')
    else:
        print('ugly kitty')
```

Final Advice: Eat your …veggies 🥦🥕🍅🥒





Final Advice:



Machine learning be like

Don’t be like this