Practice of Deep Learning
Course Introduction

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

A quick word about instructors

Matthieu (Matt) De Mari

- Senior Lecturer at SUTD (Python, Deep Learning, Prog. Language Concepts, Capstone, etc.)
- Co-Director (Master of Science in Design and Artificial Intelligence for Enterprise)
- PhD from CentraleSupelec (France)
- Email: matthieu_demari@sutd.edu.sg
- Office @ SUTD (in SUTD Academy/AI cluster): 2.401-07.

Objectives of this course

- Introduce the technical aspects related to the **implementation of Deep Neural Networks, a.k.a. Deep Learning**.
- Teach the students about the **most popular framework** for implementing Deep Neural Networks (as of Jan 2025): **PyTorch.**
- Discuss the **mathematical foundations and intuitions** behind Deep Learning tools, layers and techniques.
- Give the students a **global overview of the foundational and advanced techniques** related to Deep Learning and Neural Networks.
- Describe **examples of practical applications** of Deep Learning, showing how some key AI tools are implemented.

Skills needed for this course

- **Must-have CS:** Python, Numpy, Matplotlib.
- **Must-have Math:** Linear Algebra, Multiple Variables Calculus, Differentiation/Derivatives, Optimization.
- **Good-to-have CS:** Machine Learning, Data Processing, Pandas, Scipy, Sklearn.
- **Good-to-have Math:** Probabilities and Statistics, Graph Theory.

Need to revise?
Feel free to check your previous courses, we are going to build upon them!

The way we teach things

An expert is not someone who

- Took a 6h-long online video course,
- Implements Neural Networks by randomly connecting basic layers (typically Linear and Conv),
- Uses a random framework they do not understand (or autoML!),
- Cannot obtain more than 90% accuracy on a simple dataset like MNIST or CIFAR-10.

An expert is someone who

- Understands how Neural Networks operate and the mathematical intuition behind why typical operations used in layers,
- Understands how the frameworks have been built and what they do behind the scenes,
- Understands that AI is a fast-evolving field and knows how to stay up to date when it comes to AI.

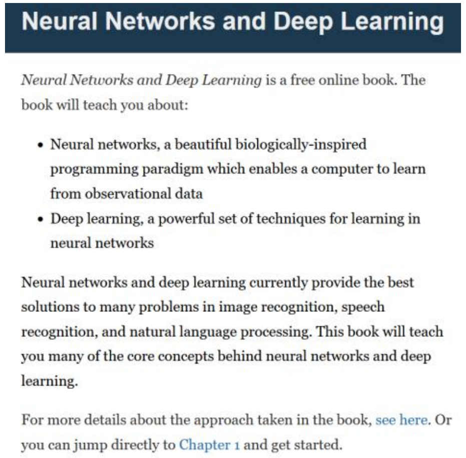
The way we teach things

- **Lectures slides**, along with **supporting notebooks**.
- **Mathematical aspects** discussed in class (to show what is happening behind the scenes of Neural Networks/Deep Learning).
- **Made a choice:** providing lots of code to demonstrate concepts but might **<u>need you to play with them autonomously</u>**!
- **Extra reading**, for curiosity (supporting papers, articles, etc.) and suggestions of pointers to stay up-to-date.
- Suggestions on how to **continue your learning**, after this course.
- (If time allows, sharing some video recordings as well).

Supporting Textbooks

Supporting textbooks, for your curiosity (not needed to understand this course, just for reference).

- Michael A. Nielsen, “Neural networks and deep learning”, 2015. (http://neuralnetworksanddeeplearning.com/)
- Ian Goodfellow, Yoshua Bengio and Aaron Courville, “Deep learning”, 2016. (https://www.deeplearningbook.org/)
- Stevens et al., “Deep Learning with PyTorch”, 2020. (https://www.manning.com/books/deep-learning-with-pytorch)



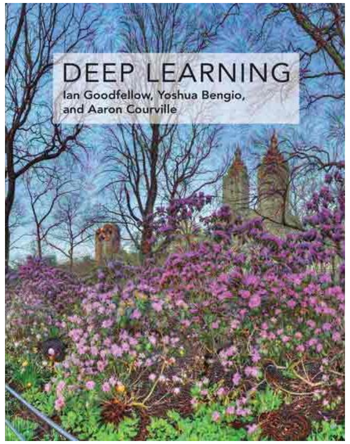
Neural Networks and Deep Learning

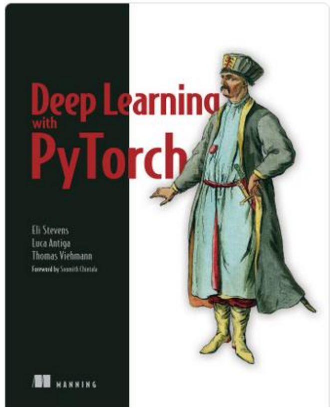
Neural Networks and Deep Learning is a free online book. The book will teach you about:

- Neural networks, a beautiful biologically-inspired programming paradigm which enables a computer to learn from observational data
- Deep learning, a powerful set of techniques for learning in neural networks

Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing. This book will teach you many of the core concepts behind neural networks and deep learning.

For more details about the approach taken in the book, see here. Or you can jump directly to Chapter 1 and get started.





Evaluation and grading

- **Quiz at the end of Day 1 and Day 2 (30%):** Simple MCQ to be taken online, after each day.
- **Lab at the end of Day 2 (60%):** A more technical implementation of the concepts seen in class on a real-life problem.
- **Participation and Attendance (10%):** Not punishing, as long as you attend classes, you should be fine!

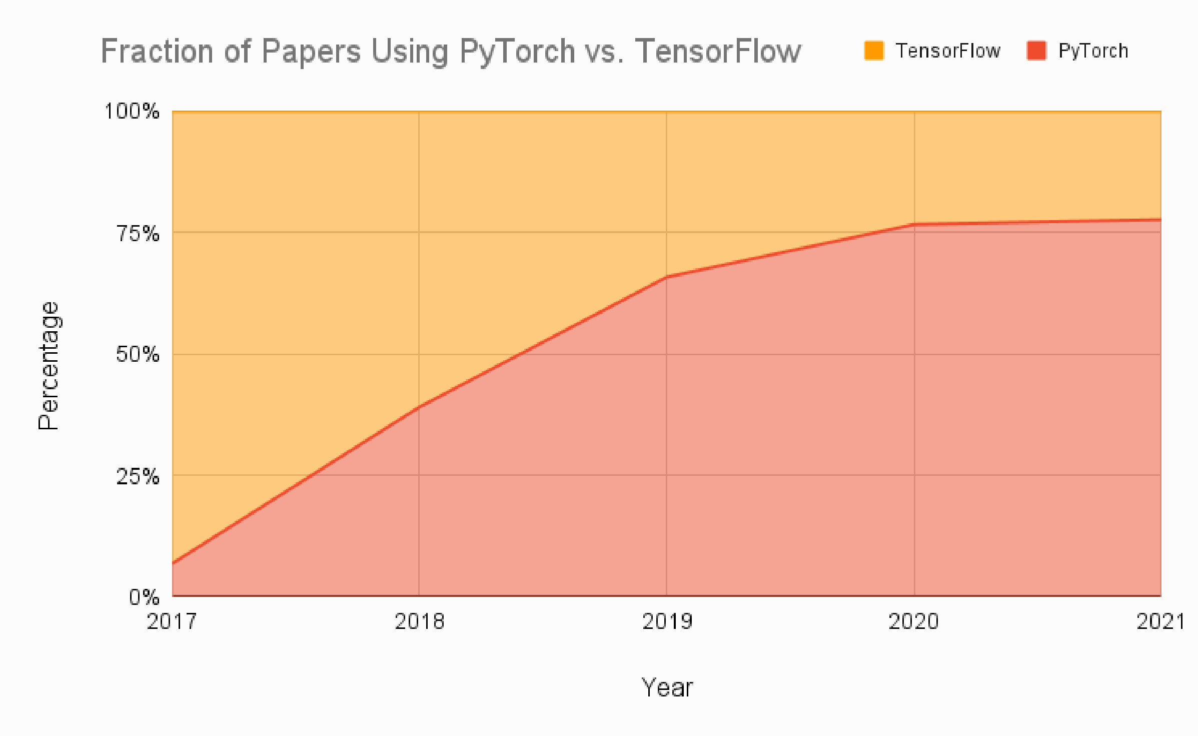
Technical pre-requisites

- Install **Python 3**, if you have not done so already (this course does use the C++/Java version of PyTorch).
- **Libraries needed (maybe more based on projects/homeworks):** numpy, matplotlib, scipy, sklearn, torch, torchvision, torchmetrics, possibly more.
- **Jupyter notebooks** for demos of code, along with the slides.
- In doubt, you can always use **Google Colab** to run the codes.

Technical pre-requisites

- Framework of choice will be **PyTorch!** (not Tensorflow, not Keras, not MXNet, etc.)
- Increasing popularity and preferred to Google's Tensorflow these days for many reasons.
- Learn more, if curious: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/
- Also, this: https://arxiv.org/abs/2508.04035

PyTorch



Installing PyTorch and CUDA

- Install PyTorch, by getting the latest version (not the one shown below!) from https://pytorch.org/get-started/locally/
- Setting up CUDA/GPU acceleration is always a good idea! (More details in bonus slides).

PyTorch Build	Stable (2.0.1)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.7	CUDA 11.8	~~ROCm 5.4.2~~	CPU
Run this Command:	pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu117			

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.
org/whl/cu117
```

Practice of Deep Learning

Day 1, Part 1/4

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

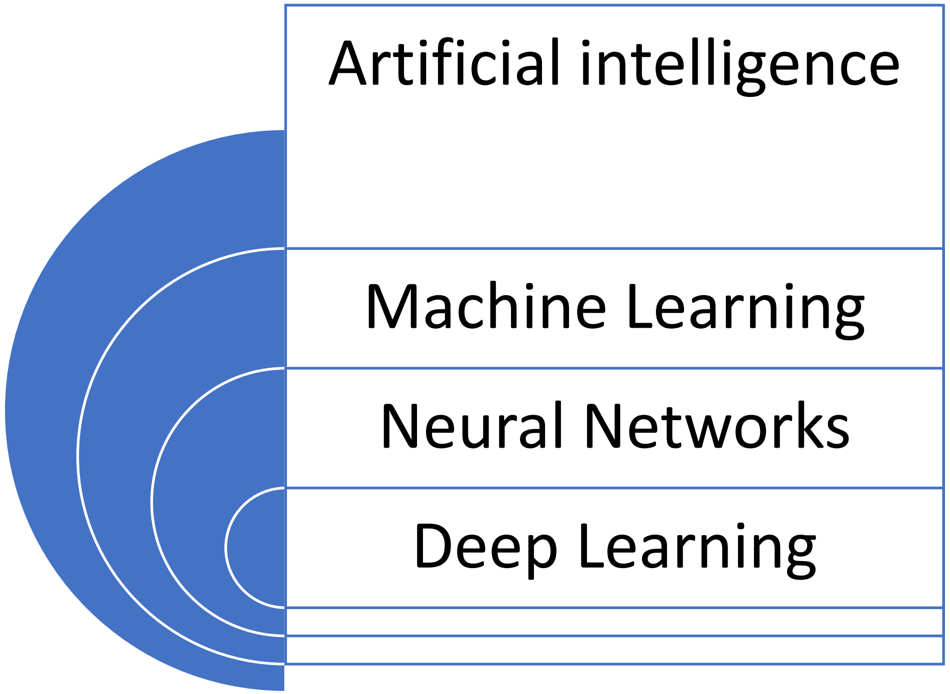
About this lecture

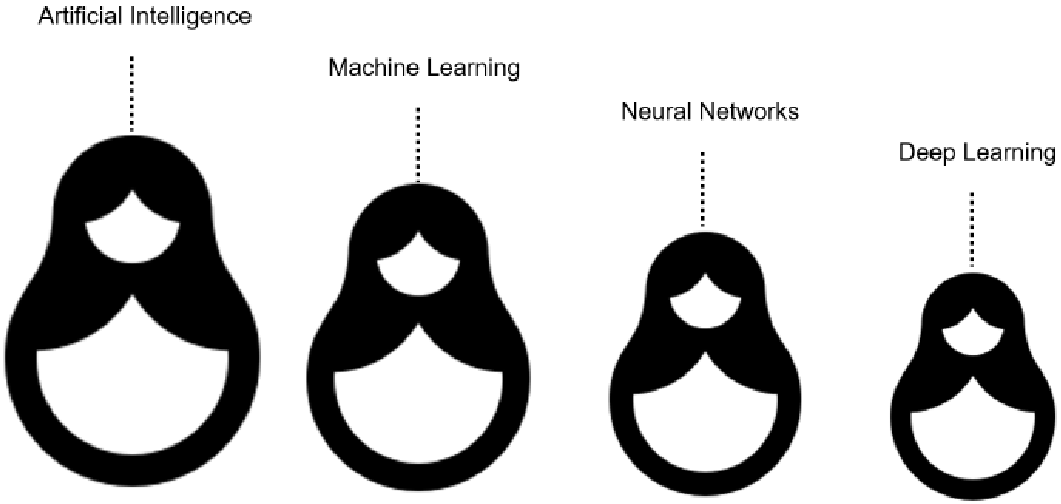
1. What are the **typical concepts of Machine Learning** to be used as a starting point for this course?
2. What are the **different families of problems** in Deep Learning?
3. What is the **typical structure of a Deep Learning problem**?
4. What is **linear regression** and how to implement it?
5. What is the **gradient descent algorithm** and how is it used to **train Machine Learning models**?

What is AI/ML/NN/DL?

Definition (Artificial Intelligence):

In Computer Science, **Artificial Intelligence (AI)** refers to the theory and development of computer systems capable to **replicate/emulate the human brain**, more specifically **perform cognitive tasks** that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages.



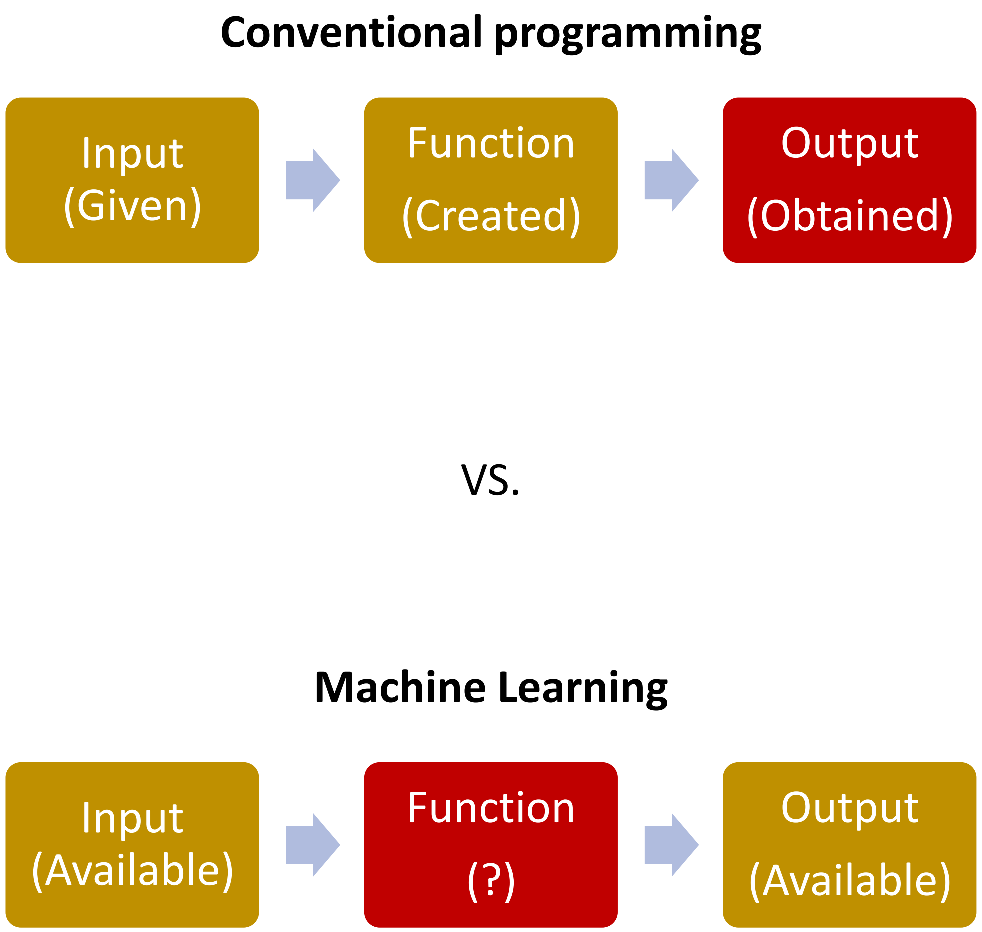


What is AI/ML/NN/DL?

Definition (Machine Learning):

In Computer Science, **Machine Learning (ML)** refers to the field of study that describes **techniques** and **algorithms** that give computers the **ability to learn without being explicitly programmed**.

Some implementations of machine learning may rely on **data** and **neural networks**.



Restricted

What is AI/ML/NN/DL?

What we normally do in programming is to design functions,

- which would do **specific operations,**
- and return **outputs,**
- for any **input** we could give it.

Input (Given)

Function (Coded Manually)

Output (Obtained)

But sometimes, we can encounter problems where

- we can easily find **inputs** and **expected** outputs,
- but the **function** to be coded is **not simple** to figure out.
- **E.g., what animal in picture?**

Input (Available)

Function (?)

Output (Available)

What is AI/ML/NN/DL?

E.g., what animal is in the picture?

Typical problem in **Computer Vision**, called **Image Recognition**.

Input x (available)

Function f

(?)

Output y = f(x)

(available)

Very easy for a human…

But, how would we explain the logic of recognizing animals in pictures to a computer?

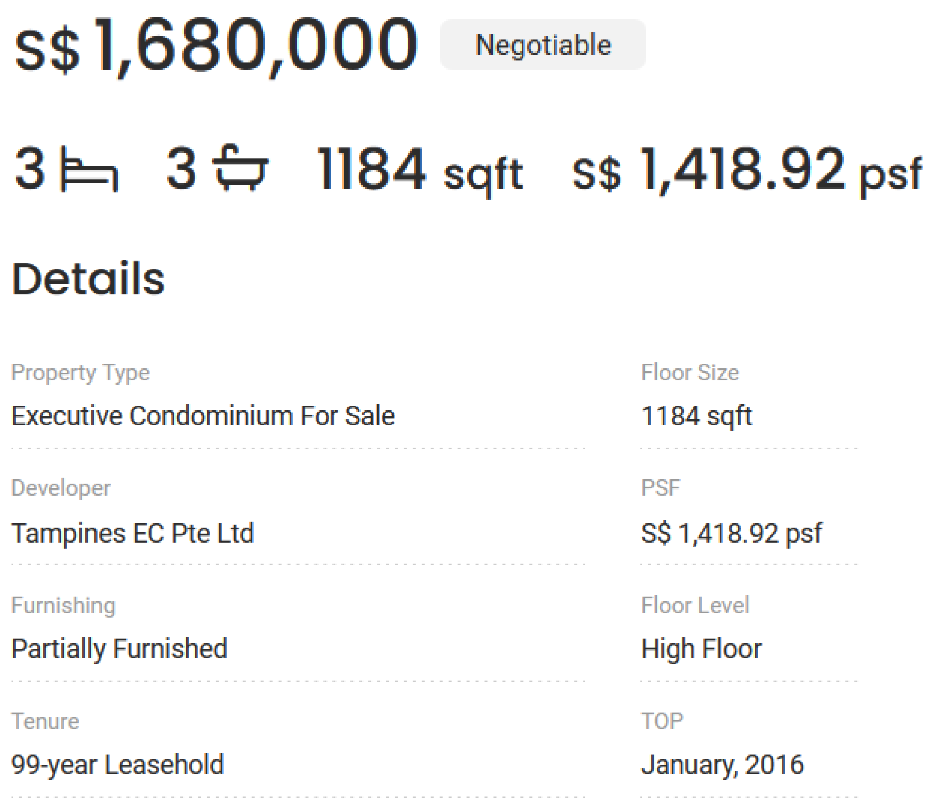
It's a cat!

What is AI/ML/NN/DL?

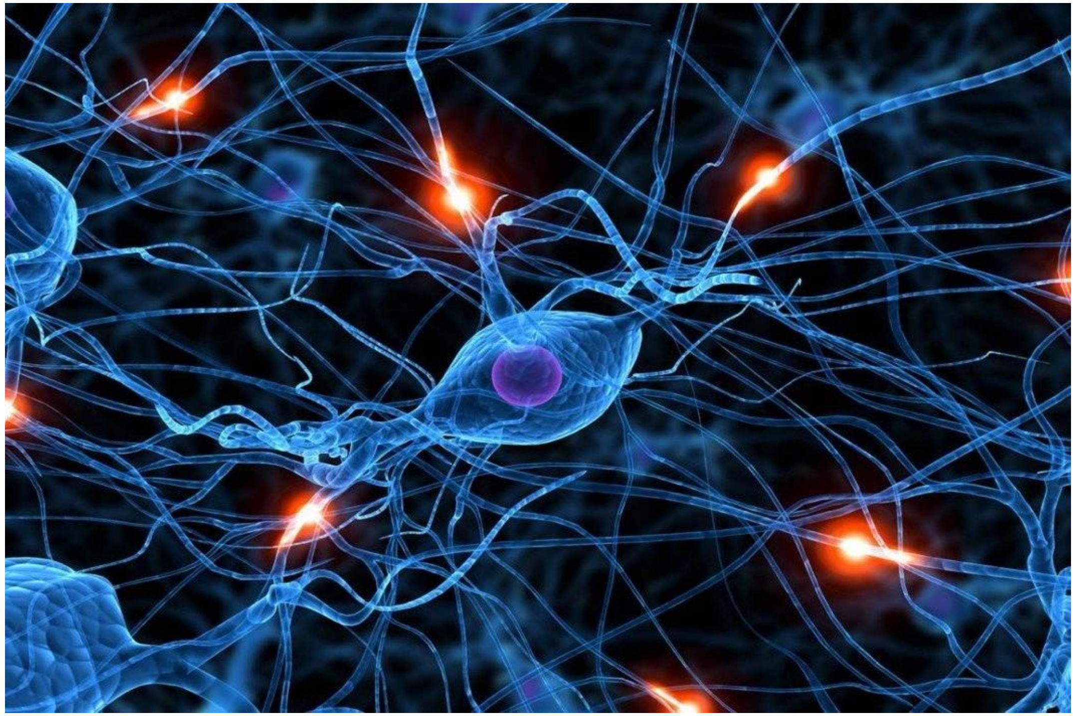
Other scenarios have to do with tasks where there is **no easy closed-form expression connecting inputs to outputs.**

- **E.g., what is a good selling price for my appartment?**
- How would you guess the selling price of an appartment based on its parameters (size, location, etc.) and previous sales?

→ **Typical Machine Learning Problems!**



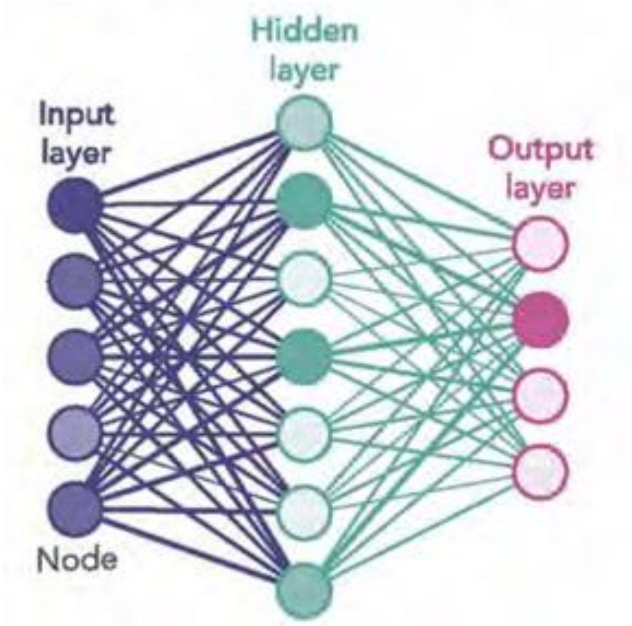
What is AI/ML/NN/DL?



Definition (Neural Networks):

Neural Networks (**NNs**) are computing systems inspired by the biological neural networks that constitute animal brains.

NNs are based on a **collection of connected units (or nodes) called artificial neurons**, which are a (loose) model for the neurons in a biological brain.



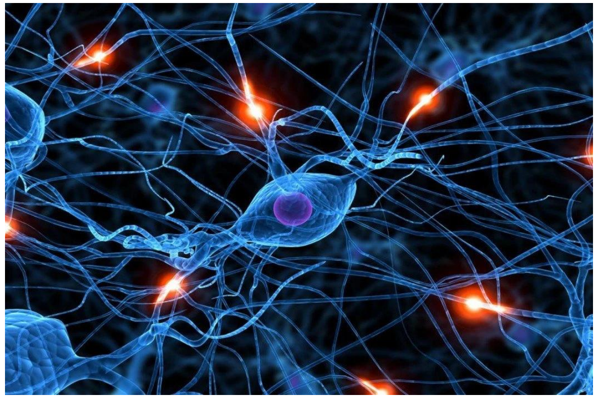
What is AI/ML/NN/DL?

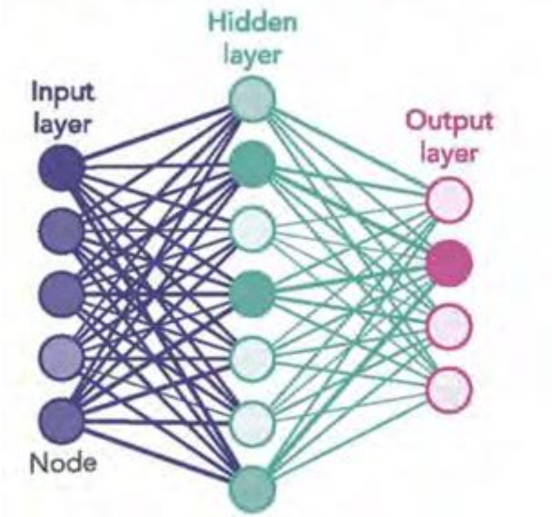
Definition (Neural Networks):

Neural Networks (**NNs**) are computing systems inspired by the biological neural networks that constitute animal brains.

NNs are based on a **collection of connected units (or nodes) called artificial neurons**, which are a (loose) model for the neurons in a biological brain.

Each connection, like the synapses in a biological brain, can transmit a signal to other neurons, therefore **processing any information given as inputs and producing a final signal as output**.



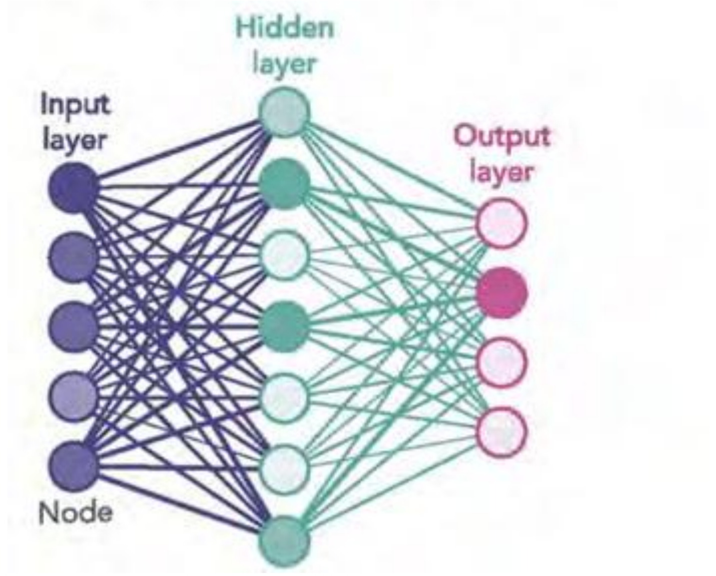


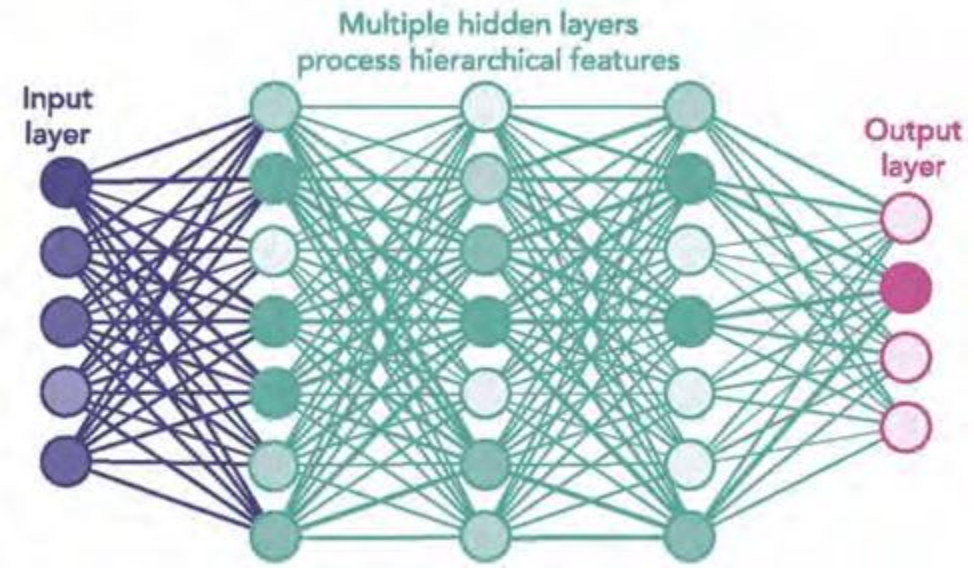
What is AI/ML/NN/DL?

Definition (Deep Learning):

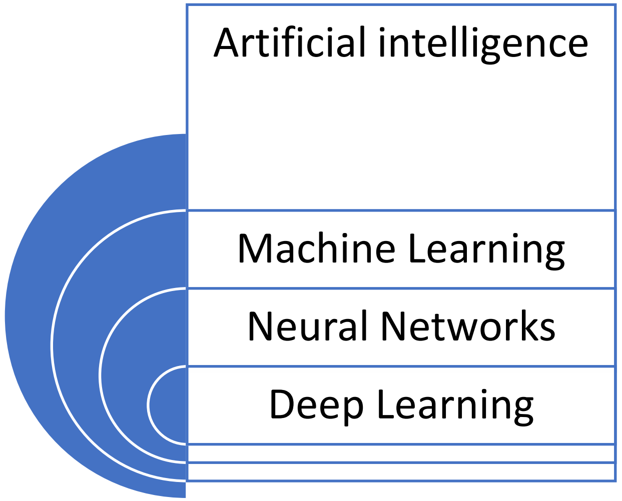
Deep Learning (DL) is a subfield of machine learning, and **deep neural networks** make up the backbone of deep learning algorithms.

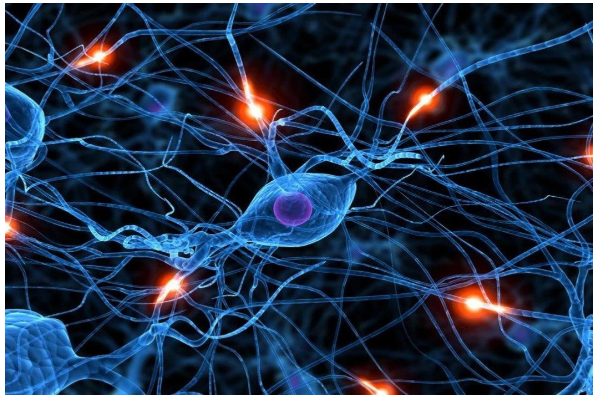
The **number of node layers,** or **depth**, of neural networks is what distinguishes a **shallow neural network** from a **deep neural network**, which must have more than three.

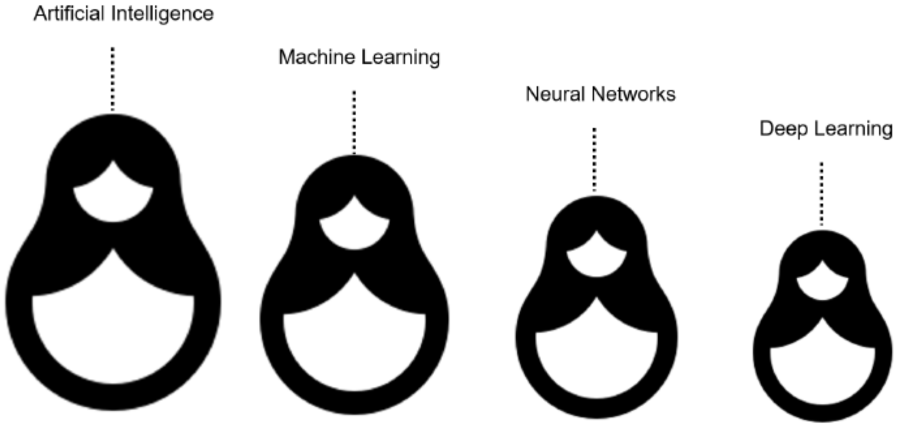


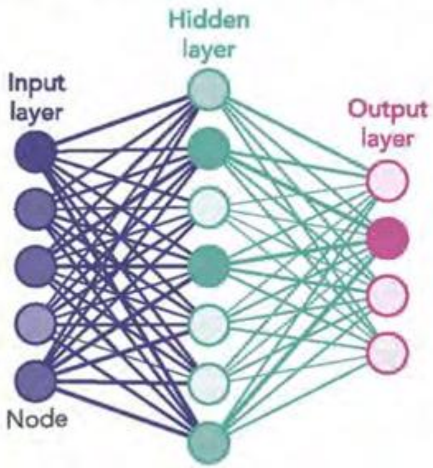


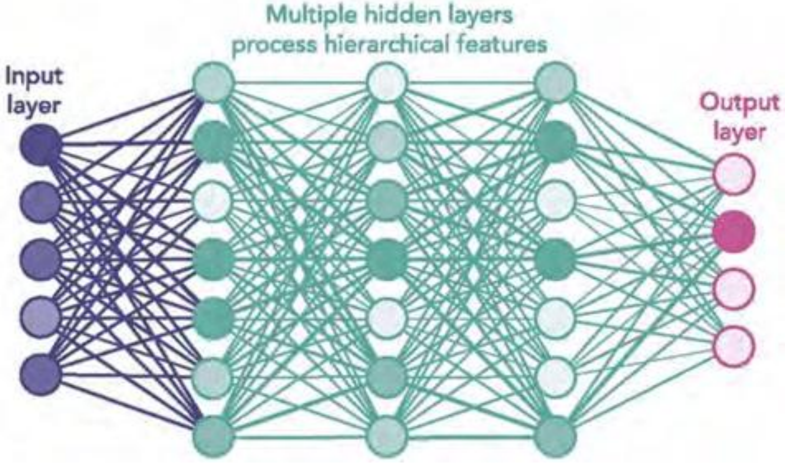
What is AI/ML/NN/DL?











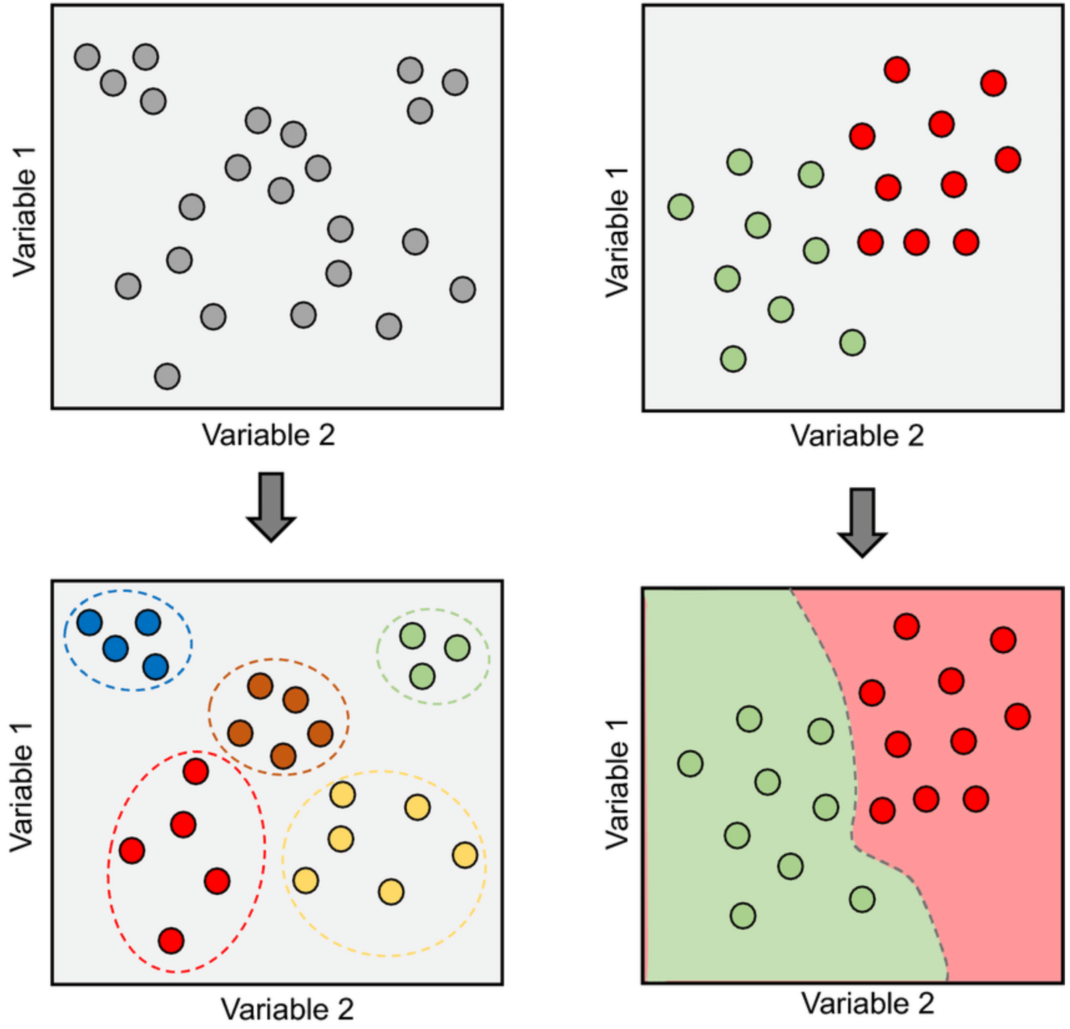
Supervised vs Unsupervised Learning

Definition (Supervised vs. Unsupervised Learning):

Supervised and **Unsupervised Learning** are the two techniques of ML.

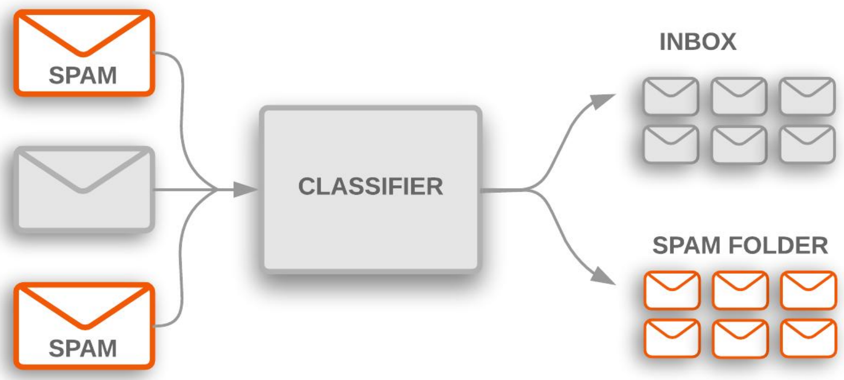
The main difference is **the need for labelled training data**:

- **Supervised machine learning** relies on **labelled input and output data** to learn and make predictions,
- while **unsupervised machine learning does not require labelled data.**



Supervised Learning Examples

- Examples of **supervised learning:** spam detection, text classification, predicting the stock market, etc.



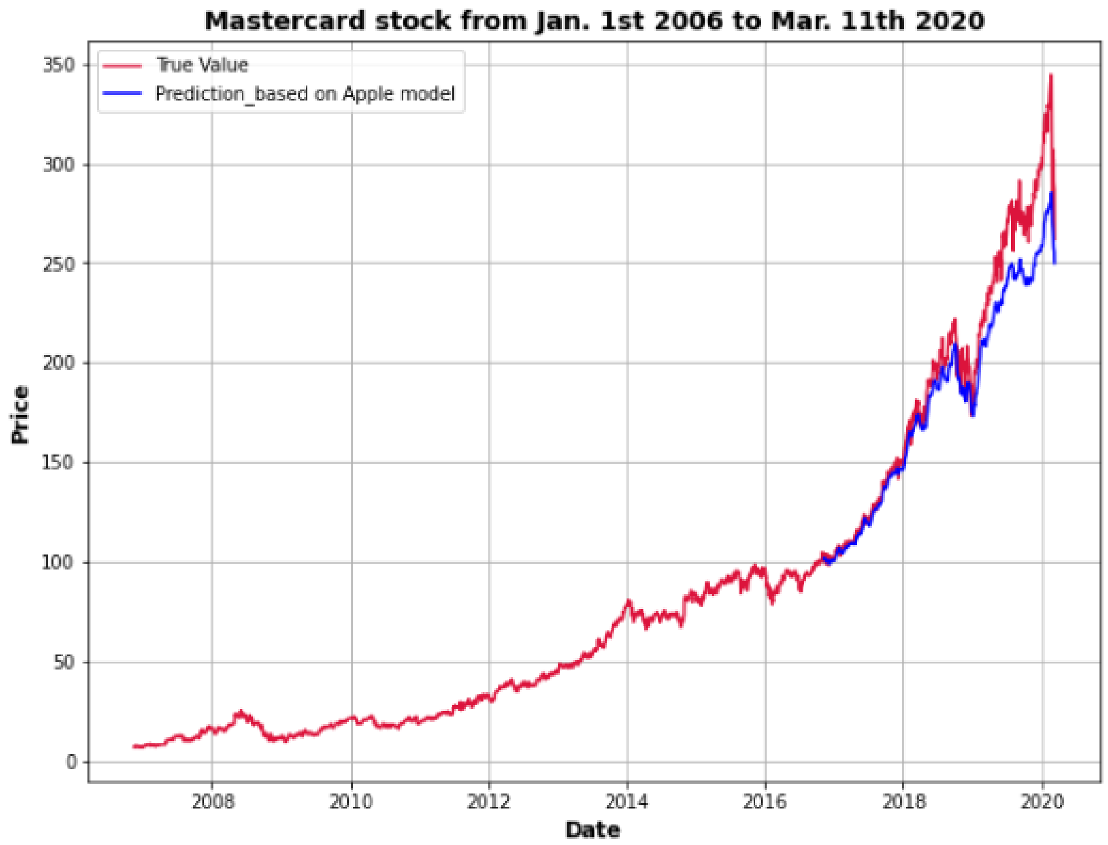


Figure 7. MAST stock price predictions using LSTM trained on AAPL

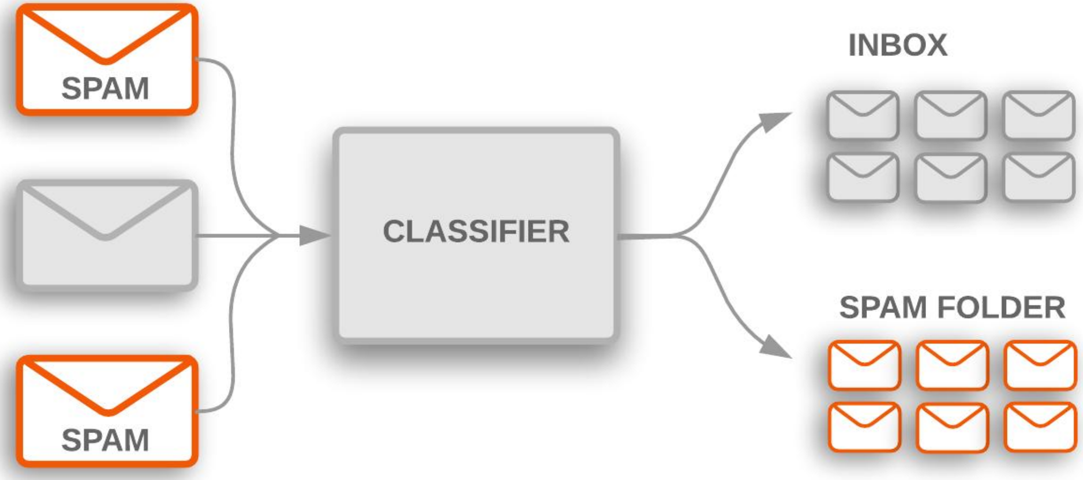
Regression vs. Classification (Supervised)

Definition (Classification):

Classification models are used to divide the samples in the dataset into different **classes**.

Classification algorithms are then used to predict/classify **discrete or finite number values**, such as Male or Female, True or False, Spam or Not Spam, etc.

In Classification, the outputs are **discrete** or **categorical values**.



It is a cat!

Regression vs. Classification

Definition (Regression):

Regression models are used to identify the relationships between the input and output variables.

Regression algorithms are used to **predict continuous values** such as price, salary, age, etc.

In regression, the outputs are often **continuous numerical values**, and not a finite list of possibilities.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type	Floor Size
Executive Condominium For Sale	1184 sqft
Developer	PSF
Tampines EC Pte Ltd	S$ 1,418.92 psf
Furnishing	Floor Level
Partially Furnished	High Floor
Tenure	TOP
99-year Leasehold	January, 2016

Supervised Learning Examples

Example: predicting the market.

Question: is it a **regression** or a **classification** task?

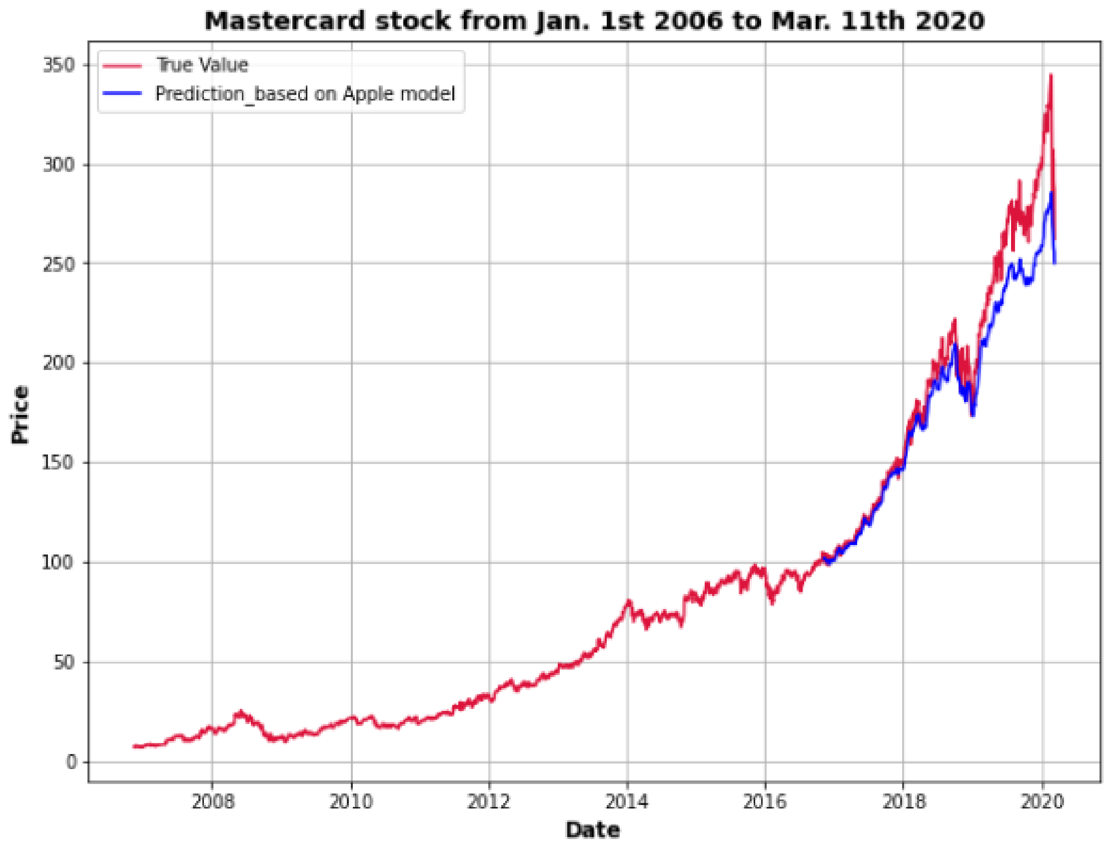


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Supervised Learning Examples

Example: predicting the market.

Question: is it a **regression** or a **classification** task?

Depends on you approach it.

If the output we are **predicting is the <u>value</u> of the stock in the future**, then it is a **regression**.

If the plan is to **predict whether we should <u>buy, sell or hold</u>**, then it is a **classification**.

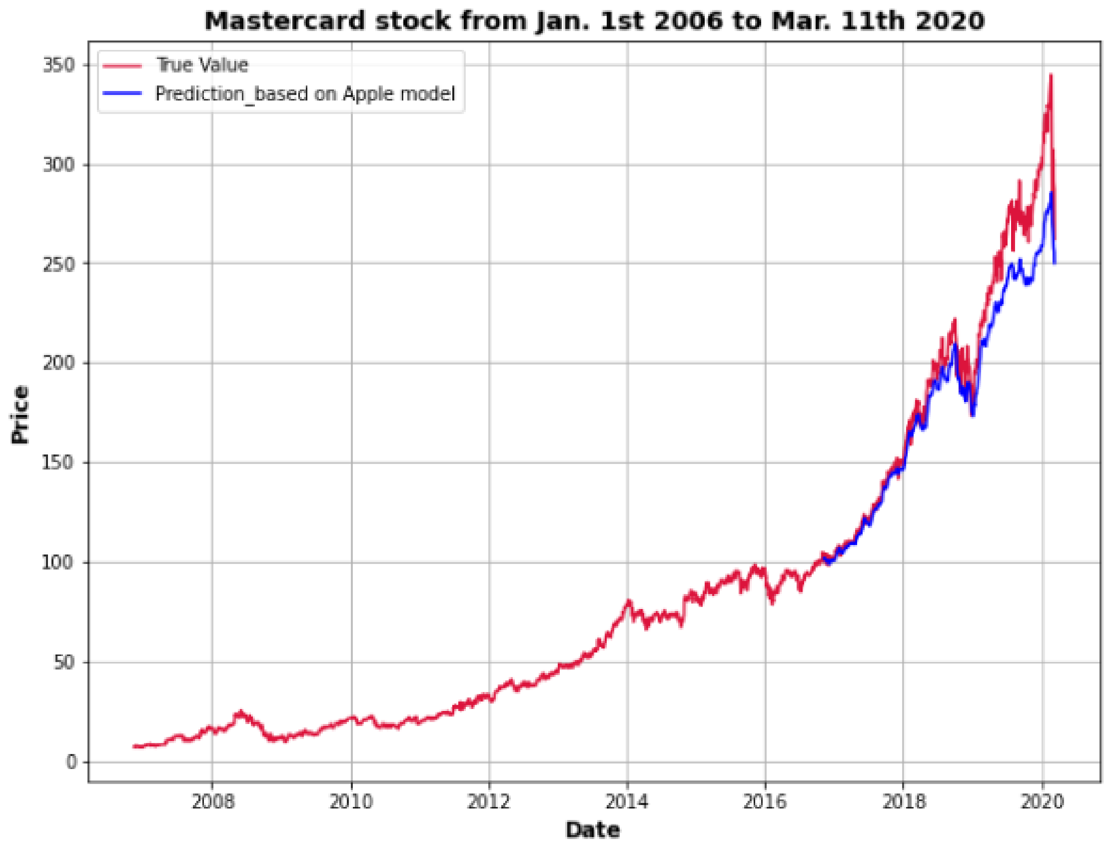


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Deep Learning and Supervised/Unsupervised

While it is possible to use Deep Learning models and Neural Networks to perform both Supervised and Unsupervised tasks…

The vast majority of DL/NN tasks fall in the category of **supervised learning**.

(And we will therefore mostly focus on that for now)

Elements of an ML problem

Definition (Elements of a ML problem):

Machine learning problems should be well-defined, i.e. the following elements must be clearly identified:

1. **Task (T)** at hand,
2. **Inputs (I)** and **Outputs (O)**,
3. **Model (M)** definition and its **trainable parameters (P)**,
4. **Loss (L)** function to measure the performance of current model on said task and dataset.

S$ 1,680,000 Negotiable

3 bed 3 bath 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Our first toy example

Our first toy example: simplified version of the apartment price prediction.

1. **Task (T)** at hand:

- **Predict price of an apartment based on the apartment features.**
- We will generate a "fake" dataset consisting of apartments with features and selling prices.
- It is a **supervised regression task**.

S$ 1,680,000 Negotiable

3 bed 3 bath 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Restricted

Our first toy example

Our first toy example: simplified version of the apartment price prediction.

2. Inputs (I) and **Outputs (O):**

- **Inputs** will be a **list of apartment features**, which here will only consist of the **surface of the apartment**, in square meters (sqm).
- **Output** will be the **selling price** in millions of SGD (mSGD).

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type	Floor Size
Executive Condominium For Sale	1184 sqft
Developer	PSF
Tampines EC Pte Ltd	S$ 1,418.92 psf
Furnishing	Floor Level
Partially Furnished	High Floor
Tenure	TOP
99-year Leasehold	January, 2016

Our first toy example (mock dataset)

Have a look at Notebook 1.

We will create a dataset, which uses surfaces as inputs, with values drawn randomly between 40 and 150 sqm.

We will assume that the price is randomly generated, using a uniform distribution, with:

- Average price being 100000 + 14373 times the surface in sqm,
- and a random "variation" of +/- 10% around the average value.

Our first toy example (mock dataset)

```python
# Two random generator functions to generate a mock dataset.
# 1. Surfaces randomly generated as uniform between min_surf and max_surf
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
```

```python
# 2. Price is 100000 + 14373 times the surface in sqm, +/- 10%
# (randomly shifted to give the dataset some diversity).
def price(surface):
    # Note: this will return the price in millions of SGD.
    return round((100000 + 14373*surface)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
```

```python
# 3. Generate dataset function
def generate_datasets(n_points, min_surf, max_surf):
    x = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    y = np.array([price(i) for i in x])
    return x, y
```

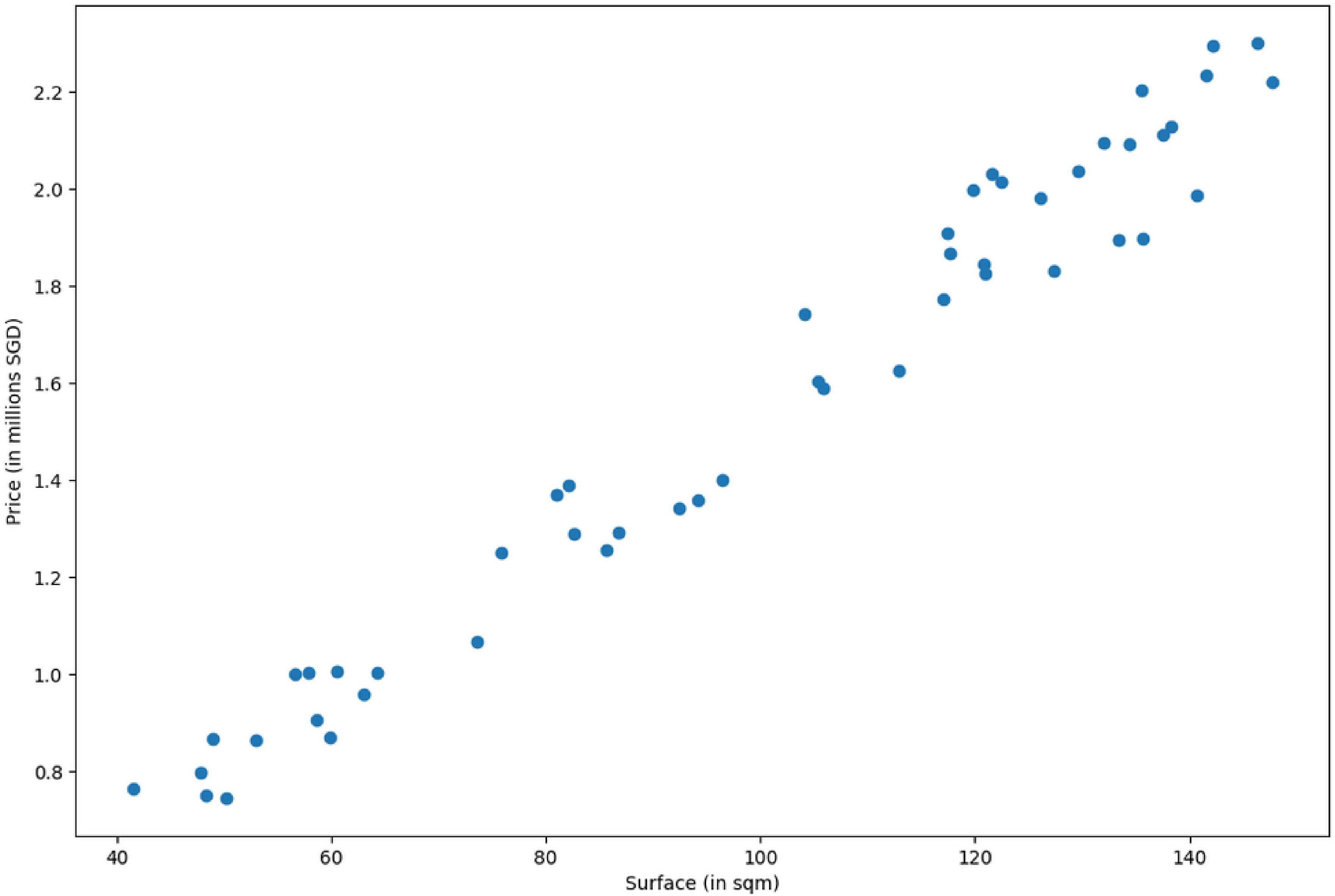
Our first toy example (mock dataset)

```python
# 4. Dataset generation (n_points points will be generated).
# Surfaces randomly generated as uniform between 40sqm and 150sqm.
# We will use a seed for reproducibility.
min_surf = 40
max_surf = 150
np.random.seed(27)
n_points = 50
inputs, outputs = generate_datasets(n_points, min_surf, max_surf)
print(inputs)
print(outputs)
```

```
[ 86.83 129.6  120.89 135.48  82.17 147.74 138.25  63.07 121.6  112.95
 137.55 134.38 122.42 135.72  60.54  75.81  81.02 127.31  56.62  58.69
  48.93  73.57 126.16  57.92  47.77 117.12  59.91 105.88  85.68  96.49
  64.27 119.81 133.44 142.18 120.95  92.42  94.22 105.4   48.36  52.92
 146.31 104.17  50.17  41.5  132.06 140.63 117.5   82.57 117.63 141.56]
[1.290893 2.034977 1.84501  2.201767 1.389632 2.218678 2.127228 0.959054
 2.029469 1.623609 2.111638 2.09194  2.012386 1.89553  1.004256 1.250228
 1.368325 1.830127 1.000719 0.906513 0.867629 1.065907 1.979544 1.001403
 0.796199 1.771816 0.867878 1.587176 1.25434  1.40047  1.002361 1.9972
 1.894479 2.293443 1.823577 1.340533 1.358613 1.602167 0.750759 0.863093
 2.30035  1.741468 0.7448   0.763732 2.093772 1.986868 1.90702  1.289541
 1.86578  2.231851]
```

Restricted

```python
# Display dataset and see that there is a rather clear linear trend.
plt.figure(figsize = (12, 8))
plt.scatter(inputs, outputs)
plt.xlabel("Surface (in sqm)")
plt.ylabel("Price (in millions SGD)")
plt.show()
```



Our first toy example

Our first toy example: simplified version of the apartment price prediction.

3. **Model (M)** definition and its **trainable parameters (P):**

- Here, our model will consist of a **linear regression model**.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type	Floor Size
Executive Condominium For Sale	1184 sqft
Developer	PSF
Tampines EC Pte Ltd	S$ 1,418.92 psf
Furnishing	Floor Level
Partially Furnished	High Floor
Tenure	TOP
99-year Leasehold	January, 2016

Our first model, the linear regression

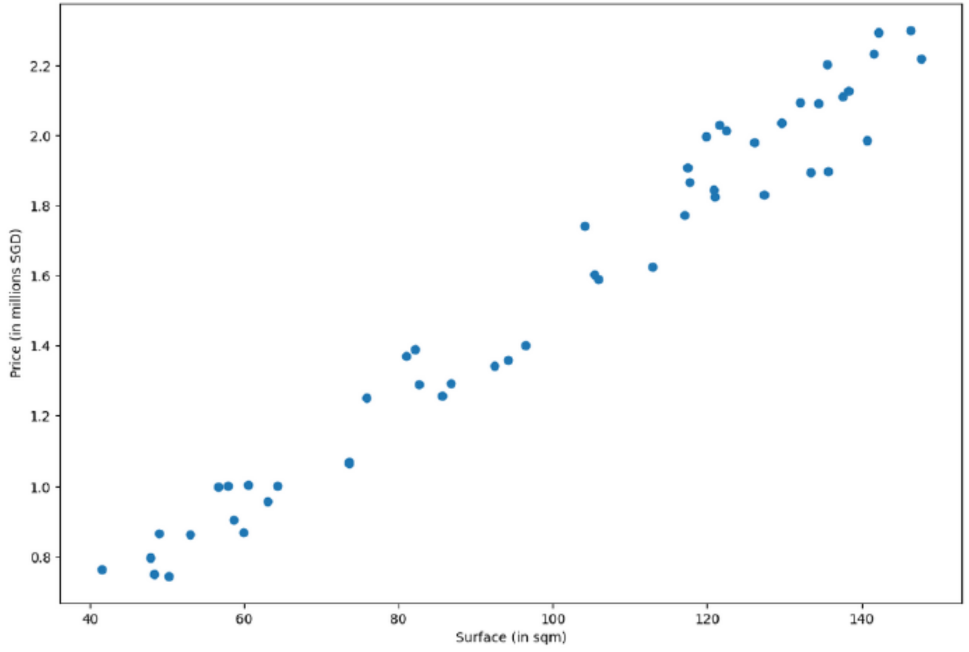
Definition (Linear Regression):

Linear Regression is a model, which **assumes that there is a linear relationship between inputs and outputs.**

It has **two trainable parameters** $(\boldsymbol{a}, \boldsymbol{b})$, to be freely chosen.

These will connect any **input** x_i to its predicted **output** y_i, with the following equation:

$$y_i \approx a\, x_i + b$$




```python
# Linear regression has two trainable parameters (a and b).
# Other parameters, like min_surf, max_surf, n_points will
# help get points for the upcoming matplotlib displays.
def linreg_matplotlib(a, b, min_surf, max_surf, n_points = 50):
    x = np.linspace(min_surf, max_surf, n_points)
    y = a*x + b
    return x, y
```

Let us try different values of (a, b)

```
# Display dataset
plt.figure(figsize = (12, 8))
plt.scatter(inputs, outputs)
plt.xlabel("Surface (in sqm)")
plt.ylabel("Price (in millions SGD)")

# Add some linreg (model 1)
a1 = 25000/1000000
b1 = 0
linreg_dataset1_inputs, linreg_dataset1_outputs = linreg_matplotlib(a1, b1, min_surf, max_surf, n_points)
plt.plot(linreg_dataset1_inputs, linreg_dataset1_outputs, 'r', label = "Model 1")

# Another linreg (model 2)
a2 = 17000/1000000
b2 = 50000/1000000
linreg_dataset2_inputs, linreg_dataset2_outputs = linreg_matplotlib(a2, b2, min_surf, max_surf, n_points)
plt.plot(linreg_dataset2_inputs, linreg_dataset2_outputs, 'g', label = "Model 2")

# A final linreg (model 3)
a3 = 14500/1000000
b3 = 95000/1000000
linreg_dataset3_inputs, linreg_dataset3_outputs = linreg_matplotlib(a3, b3, min_surf, max_surf, n_points)
plt.plot(linreg_dataset3_inputs, linreg_dataset3_outputs, 'k', label = "Model 3")

# Display
plt.legend(loc = 'best')
plt.show()
```

Model 1 (red)

Model 2 (green)

Model 3 (black)

Which model is the best fit for this task in your opinion? (red, green or black?)

Why is one better than another?

The need for a loss function

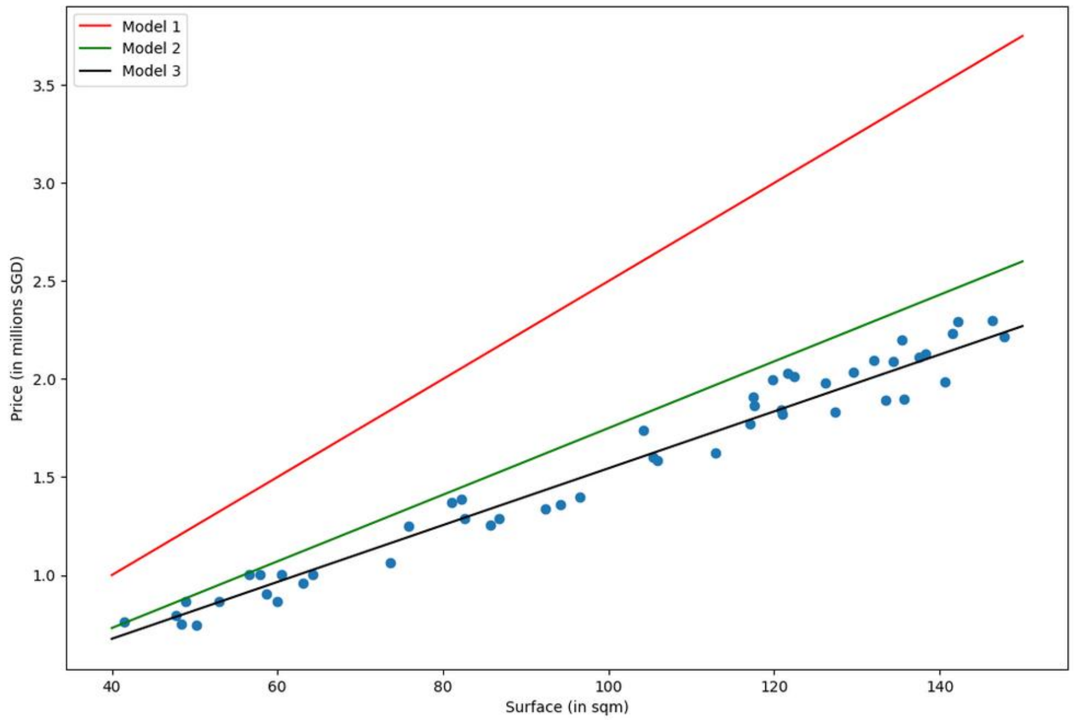
Telling which model is best at a given ML task can prove difficult, especially for close values of its trainable parameters, here (a, b).

Good news, there is one last element we have not used yet.

4. **Loss (L)** function to measure the performance of the current model and its current parameters on said task and dataset.

Which model is the best fit for this task in your opinion? (red, green or black?)

Why is one better than another?



The need for a loss function

Definition (Loss function):

A **loss function** (also known as a **cost function**) is a mathematical function that measures the difference between the predicted outputs of a model and the true outputs we should be matching.

It is typically used to:

- **Evaluate the performance of the model,**
- And more specifically **evaluate how good our choice of trainable parameters** (here, the parameters a and b) are at fitting the data.

More importantly, the **loss** function will later be used to guide the optimization process during **training** (more on this later!).

The need for a loss function

Definition (The Mean Square Error loss function):

In our linear model, a good loss function we could use consists of the **Mean Square Error (MSE) loss function**.

The MSE is computed by calculating the square difference between:

- The **prediction** $\boldsymbol{p_i(x_i, a, b) = ax_i + b}$ formed by the model for some given inputs x_i,
- The **true value** $\boldsymbol{y_i}$ that we should be matching.

We then repeat this operation for every sample i, and average those errors together, i.e.

$$L(a, b, x, y) = \frac{1}{N}\sum_{i=1}^{N}(p(x_i, a, b) - y_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(ax_i + b - y_i)^2$$

The need for a loss function

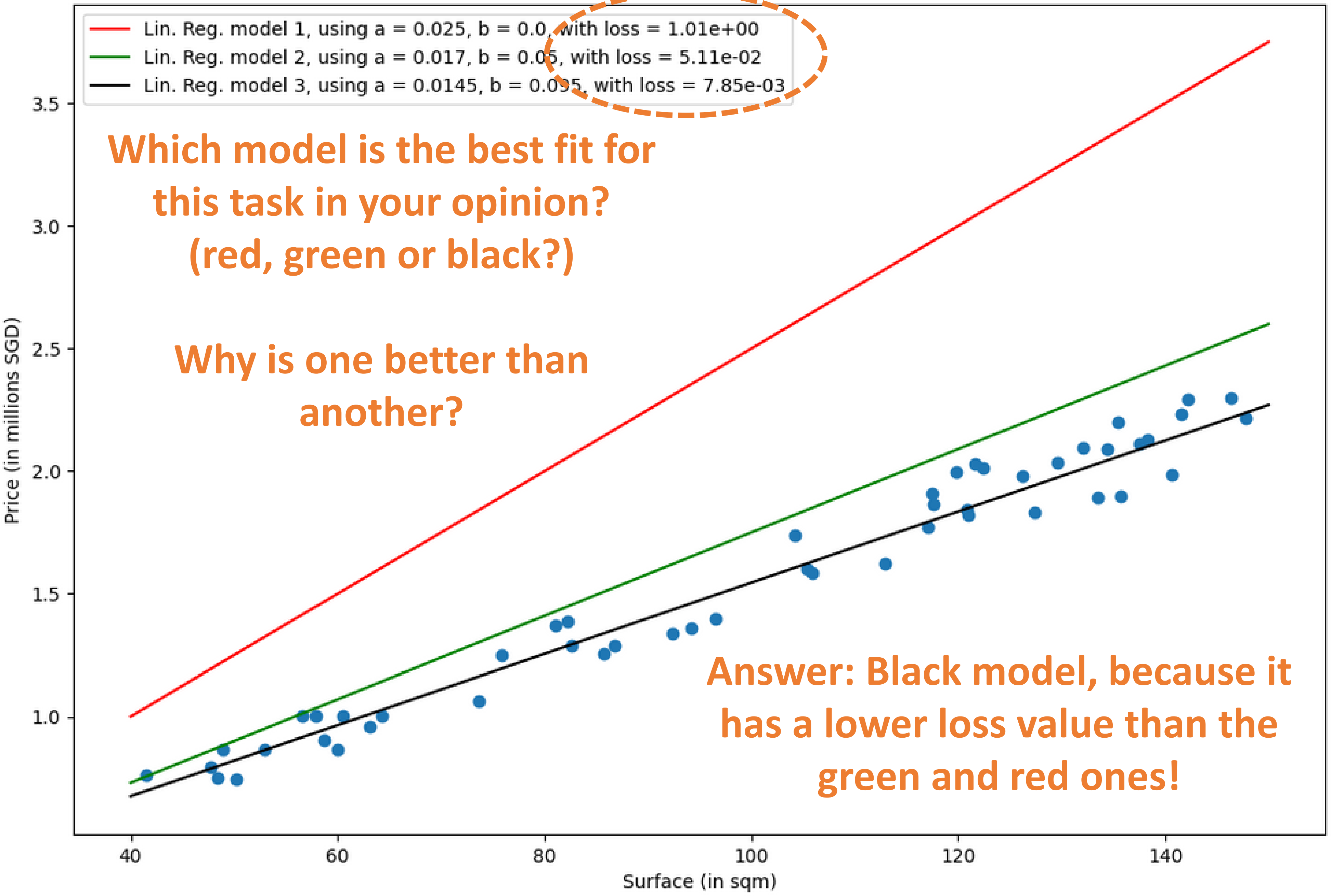
Recall the definition of the MSE loss function

$$L(a, b, x, y) = \frac{1}{N}\sum_{i=1}^{N}(ax_i + b - y_i)^2$$

In [9]:
```
# Mean square error as a loss function
# Displaying loss using exponential notation (XXXe-YYY)
def loss_mse(a, b, x, y):
    val =  np.sum((y - (a*x + b))**2)/x.shape[0]
    return '{:.2e}'.format(val)
```

In [10]:
```
# The lower the loss function, the better the linear regression values (a, b) fit the dataset.
loss1 = loss_mse(a1, b1, inputs, outputs)
loss2 = loss_mse(a2, b2, inputs, outputs)
loss3 = loss_mse(a3, b3, inputs, outputs)
print(loss1, loss2, loss3)
```

```
1.01e+00 5.11e-02 7.85e-03
```



Which model is the best fit for this task in your opinion? (red, green or black?)

Why is one better than another?

Answer: Black model, because it has a lower loss value than the green and red ones!

Training a model

Definition (Training a model):

Training a model consists of **finding the best values for the trainable parameters of the chosen model**, for the given task and dataset.

By best values for the trainable parameters, we mean the values that **minimize the chosen loss function**. It is therefore an optimization problem of some sort!

Once a minimal value is obtained on the loss function, then we can claim that the "optimal" trainable parameters have been found.

Or in other words, that the model has been **trained** to solve its machine learning task.

Training a linear regression model

In the case of linear regression, training the model consists of solving the following optimization problem.

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) \right]$$

That is, using the explicit formula of the loss function,

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[\frac{1}{N} \sum_{i=1}^{N} (ax_i + b - y_i)^2 \right]$$

Training a linear regression model

Definition (The normal equation for linear regression):

Solving this optimization problem can be done analytically, as it can be proven that the optimal choice of parameters $W^* = \begin{pmatrix} b^* \\ a^* \end{pmatrix}$ follows the **normal equation**, below.

$$W^* = (X^T X)^{-1} X^T Y$$

(Proof in bonus slides)

With

$$X = \begin{pmatrix} 1 & x_1 \\ \dots & \dots \\ 1 & x_N \end{pmatrix}$$

$$Y = \begin{pmatrix} y_1 \\ \dots \\ y_N \end{pmatrix}$$

Finally, X^T denotes the transposed matrix of X and $(X)^{-1}$ its inverse, which is a typical linear algebra operation.

Training a linear regression model

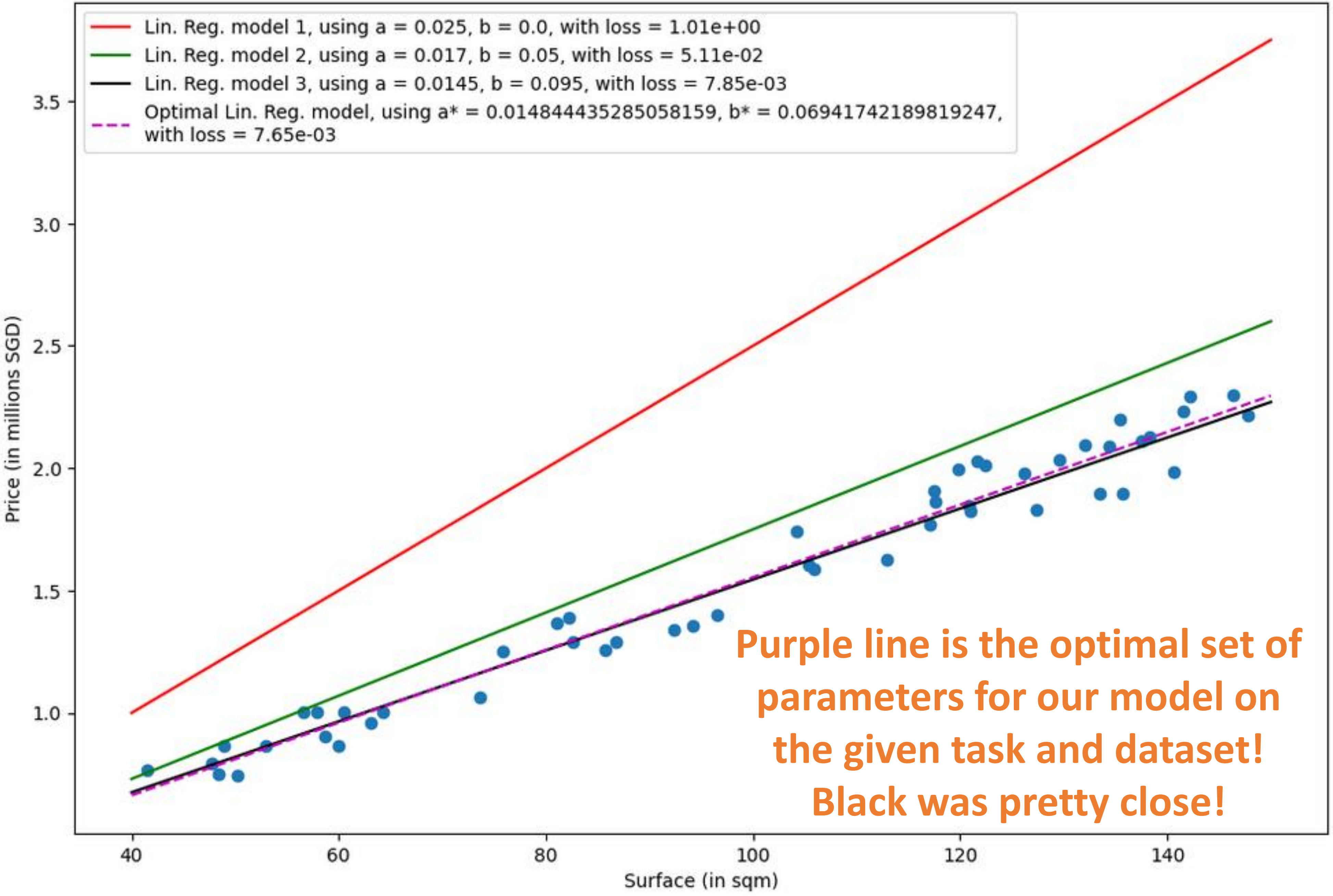
Normal Equation: $W^* = (X^T X)^{-1} X^T Y$

```
# Defining the X matrix, following the notation above, as a numpy array.
X = np.array([[1, x_i] for x_i in inputs])
print(X)
```

```
# While we are at it, let us define the transposed version of X.
XT = np.transpose(X)
print(XT)
```

```
# Defining the Y matrix, following the notation above, as a numpy array.
Y = np.array([[y_i] for y_i in outputs])
print(Y)
```

```
# Defining W_star according to our formula.
W_star = np.matmul(np.linalg.inv(np.matmul(XT,X)), np.matmul(XT,Y))
print(W_star)
# Extracting a_star and b_star values from W_star.
b_star, a_star = W_star[0, 0], W_star[1, 0]
print("Optimal a_star value: ", a_star)
print("The value we used for a in the mock dataset generation: ", 14373/1000000)
print("Optimal b_star value: ", b_star)
print("The value we used for b in the mock dataset generation: ", 100000/1000000)
```



Purple line is the optimal set of parameters for our model on the given task and dataset! Black was pretty close!

The problem with the normal equation

The normal equation $W^* = (X^T X)^{-1} X^T Y$ immediately gives the optimal set of parameters (a^*, b^*) to use for linear regression.

A few problems when using the normal equation:

- It can become **computationally expensive when the number of samples (and/or features) in the dataset are large, and therefore the dimensions of X are large**.
- In some scenarios, it might even be **impossible to use when the matrix product of feature variables $\left(X^T X\right)^{-1}$ cannot be calculated (because $X^T X$ might not always be invertible)**.

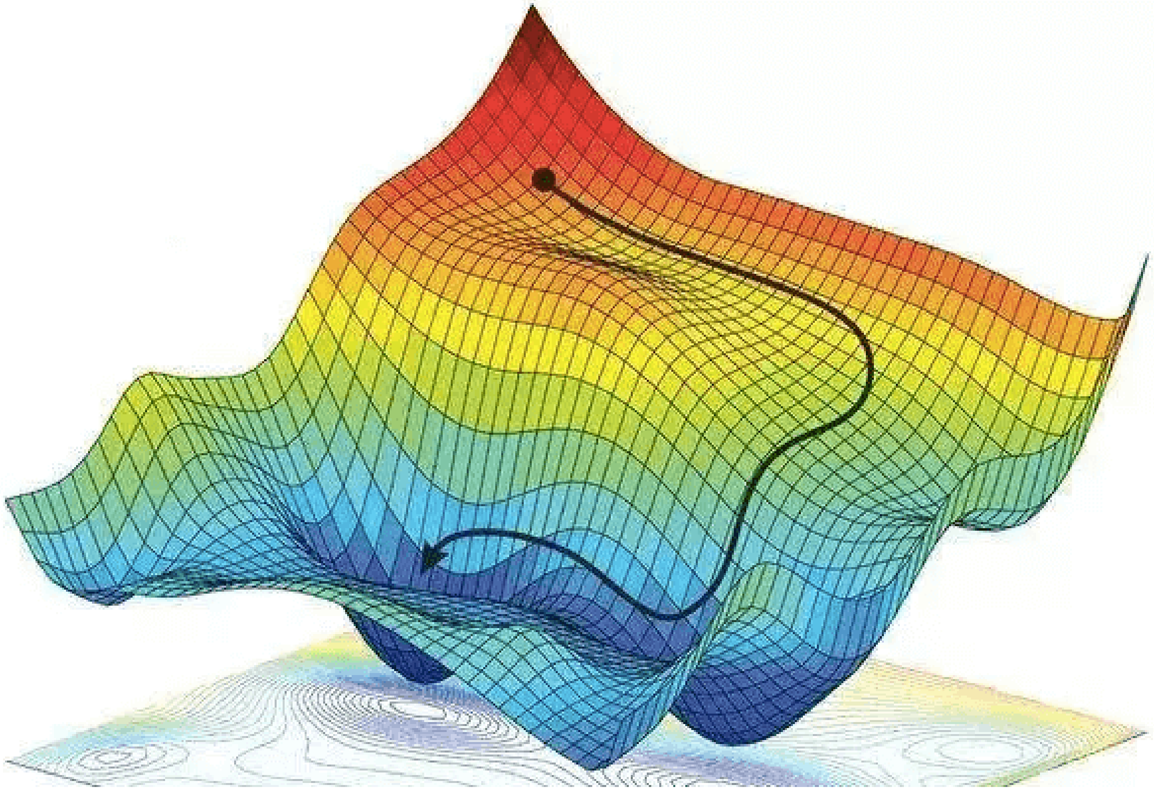
Another problem: More sophisticated models than linear regression will NOT have a normal equation, anyway.

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial non-optimal set of parameters for our model.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given loss/cost function, until it converges to a local minimum.



Forgot about GD?
Have a look at your notes from 10.013 M&A!

Restricted

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial non-optimal set of parameters for our model.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given loss/cost function, until it converges to a local minimum.

- The normal equation has many problems…
- But GD, on the other hand can be used to find the optimal solution even when the normal equation is not applicable.
- **While sub-optimal, the main advantage of gradient descent is that it can handle very large datasets and it can also be used for non-linear models.**

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) aims to solve an optimization problems such as:

$$a^* = arg\,\max_a[f(a)]$$

It does so by

1. Initializing a as some value a_0, and a counter i as 0,
2. Calculating the derivative $D = \frac{df(a_i)}{da_i}$,
3. Calculating a_{i+1} as $a_{i+1} = a_i - \alpha D$ and incrementing i by 1,
4. Repeating operations 2 and 3 until a maximum number of iterations is reached or the values of a converge (and no longer change).

GD in Linear Regression

In the case of Linear Regression, with the MSE loss defined earlier, we have

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) = \frac{1}{N} \sum_{i=1}^{N} (ax_i + b - y_i)^2 \right]$$

Technically, we have two decision variables (a and b) and we are using the loss function as our objective function to minimize, but it works the same!

GD in Linear Regression

We can calculate (by hand) the derivatives, with respect to a and b:

$$D_a = \frac{\partial L}{\partial a} = \frac{-2}{N} \sum_{i=1}^{N} x_i \left(y_i - (a\, x_i + b)\right)$$

$$D_b = \frac{\partial L}{\partial b} = \frac{-2}{N} \sum_{i}^{N} y_i - (a\, x_i + b)$$

GD in Linear Regression

The gradient descent **update rules** are therefore defined as

$$a \leftarrow a \ - \alpha D_a$$

$$a \leftarrow a + \frac{2\alpha}{N} \sum_{i=1}^{N} x_i \left(y_i \ - \ (a\, x_i \ + \ b)\right)$$

And

$$b \leftarrow b \ - \alpha D_b$$

$$b \leftarrow b + \frac{2\alpha}{N} \sum_{i}^{N} y_i \ - \ (a\, x_i \ + \ b)$$

We call parameter $\boldsymbol{\alpha}$ the **learning rate** for the gradient descent, and this parameter value will have to be decided manually, later.

GD in Linear Regression

Gradient Descent Linear Regression procedure:

- We initialize a and b with some initial values (could be zero, random values, or something else).
- For a given number of iterations, we will apply the two update rules defined earlier.
- *(Optionally, we might decide to stop iterating, if we realize that the values of a and b are no longer changing.*
  *This is called **early stopping**, and is typically implemented by tracking the changes on each iteration.*
  Typically, we could decide to break the GD loop if the changes on both parameters a and b are less than a given threshold δ.)

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Initialize a and b as you please.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradient descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Number of samples in dataset.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Two possible stopping conditions, change < delta or counter > max_count.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradient descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Update using our GD update rules from earlier.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # Compute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Compute change on this iteration (to decide on early stopping if changes become insignificant).

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Calculate new MSE value using the new parameters of a and b. We also keep track of these loss values by adding them to a list, used for display later.

```
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = -2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Return trained parameters and loss values list on each iteration.

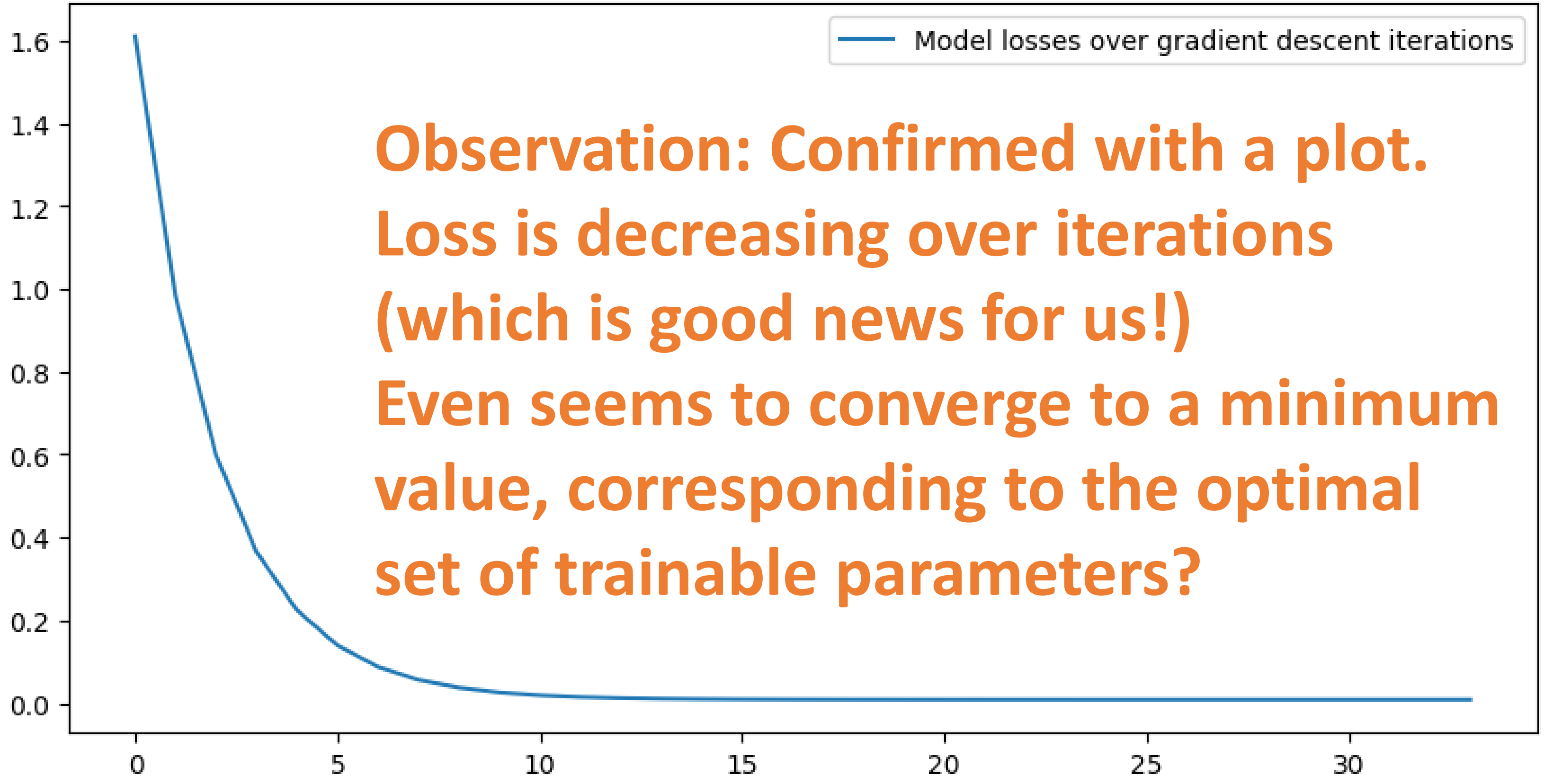
```
1 a_gd, b_gd, losses = gradient_descent_linreg(a_0 = 0, b_0 = 0, x = inputs, y = outputs, alpha = 1e-5, delta = 1e-6)
```

```
-----
Gradients:  -342.3996226384001 3.105429920000001
New values for (a, b):  0.003423996226384013 -3.105429920000001e-05
Change:  0.003423996226384013
Loss:  1.61e+00
-----
Gradients:  -266.62829357559235 2.4212214483389536
New values for (a, b):  0.006090279162139925 -5.526651368338955e-05
Change:  0.0026662829357559236
Loss:  9.82e-01
-----
Gradients:  -207.62478780001163 1.8884249597020164
New values for (a, b):  0.008166527040140042 -7.415076328040972e-05
Change:  0.0020762478780001164
Loss:  5.99e-01
-----
Gradients:  -161.6784683033283 1.4735337252735503
New values for (a, b):  0.009783311723173324 -8.888610053314522e-05
Change:  0.001616784683033283
```

Observation: Loss is decreasing over iterations (which is good news for us!)

```python
# Display dataset
plt.figure(figsize = (10, 5))
plt.plot(losses, label = "Model losses over gradient descent iterations")

# Display
plt.legend(loc = 'best')
plt.show()
```



Observation: Confirmed with a plot. Loss is decreasing over iterations (which is good news for us!)
Even seems to converge to a minimum value, corresponding to the optimal set of trainable parameters?

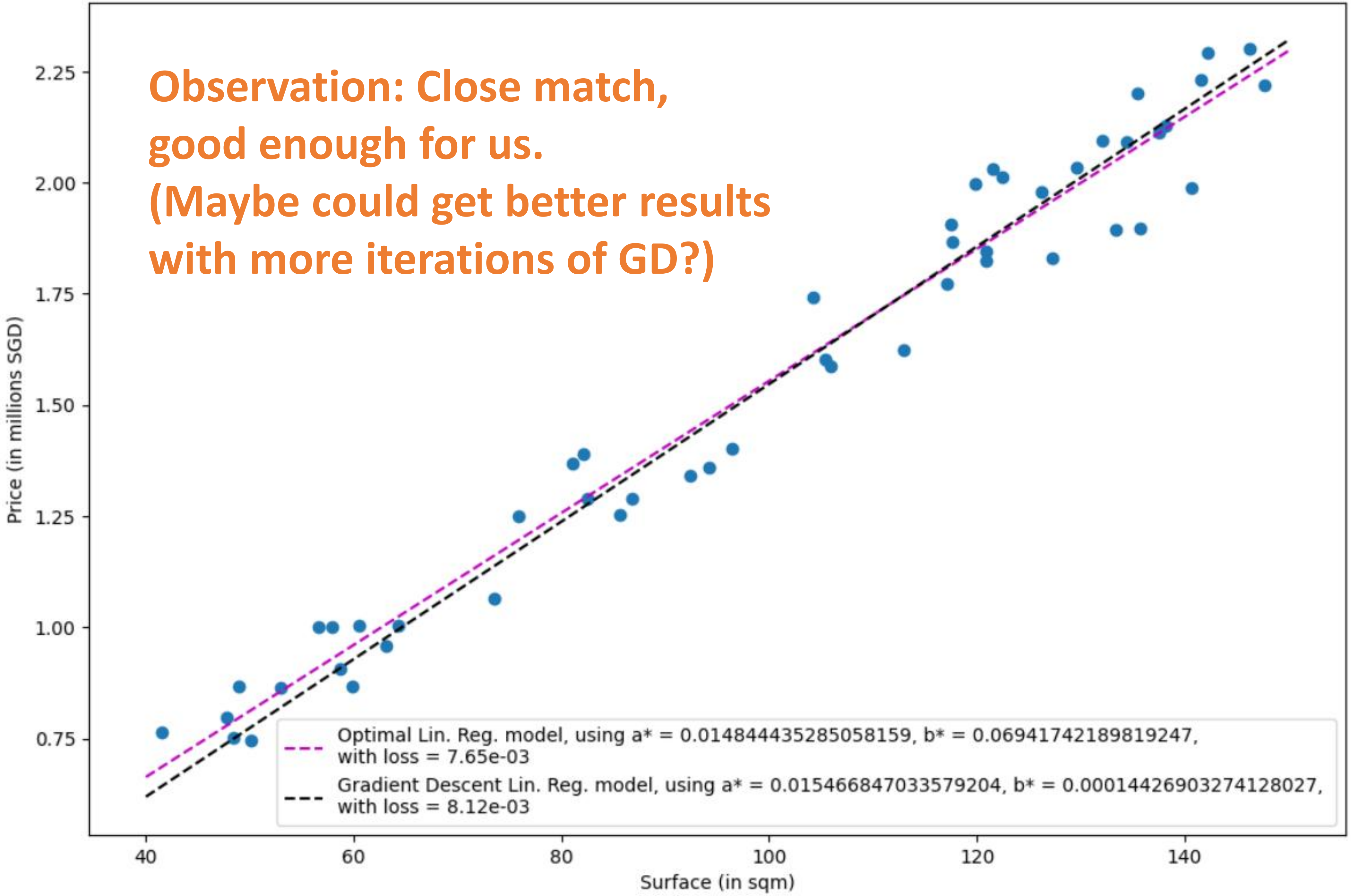
Checking for optimal parameters

We have generated the dataset ourselves, so we know the optimal values for a and b! (And we can even compute using the normal equation anyway…)

```
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by gradient descent: ", a_gd)
print("We used 14373/1000000 in the mock dataset generation, which is: ", 14373/1000000)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by gradient descent: ", b_gd)
print("The value we used in the mock dataset generation: ", 100000/1000000)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by gradient descent:  0.015466847033579204
We used 14373/1000000 in the mock dataset generation, which is:  0.014373
Optimal b_star value:  0.06941742189819247
Value for b_star, found by gradient descent:  0.00014426903274128027
The value we used in the mock dataset generation:  0.1
```

Observation: Close match, good enough for us.
(Maybe could get better results with more iterations of GD?)



Using Sklearn for Linear regression

In practice, we never implement the linear regression model ourselves (but like many other things in CS, it is a good practice to try it once!)

- It is often better/faster to rely on the **sklearn** library and use the **LinearRegression** object!

```
# Creating a sklearn Linear Regression model.
# It uses the same analytical formula from earlier, i.e. W^* = (X^T X)^{-1} X^T Y.
reg = LinearRegression().fit(sk_inputs, sk_outputs)
# The coefficients for a* and b* are found using coeff_ and intercept_ respectively.
a_sk = reg.coef_[0]
b_sk = reg.intercept_
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by sklearn: ", a_sk)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by sklearn: ", b_sk)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by sklearn:  0.014844435285058166
Optimal b_star value:  0.06941742189819247
Value for b_star, found by sklearn:  0.06941742189818956
```

Predicting using Sklearn Linear regression

After training, it is also good practice to check if the predictor makes sense, by **testing** it and asking it to predict the price of an apartment it has never seen before.

- Confirm the pertinence of predicted values manually, if possible.

```
# We can later use this Linear Regression model, to predict the price of
# a new appartment with surface 105 sqm (price in millions SGD).
new_surf = 105
pred_price = reg.predict(np.array([[new_surf]]))[0]
print(pred_price)
```

```
1.628083126829297
```

```
avg_price = 14373*new_surf + 100000
min_val = 0.9*avg_price
max_val = 1.1*avg_price
print("Min, max, avg prices: ", [min_val, max_val, avg_price])
```

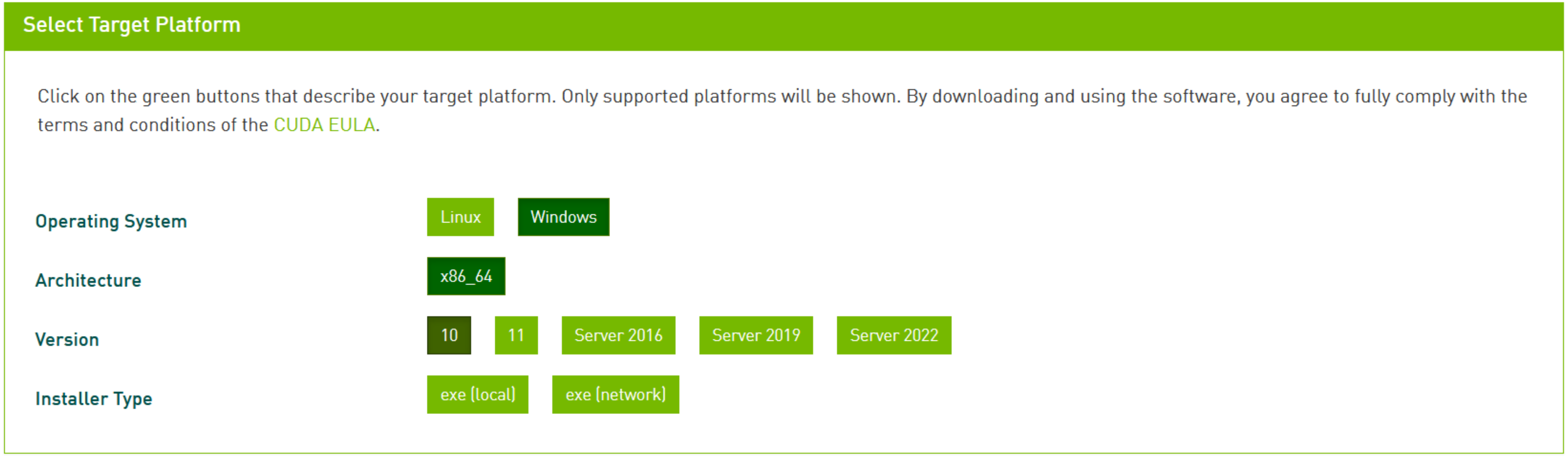
```
Min, max, avg prices:  [1448248.5, 1770081.5000000002, 1609165]
```

Let us call it a break for now

We will continue on the next lecture with more ML reminders.
(Bonus slides after this)

Installing PyTorch and CUDA

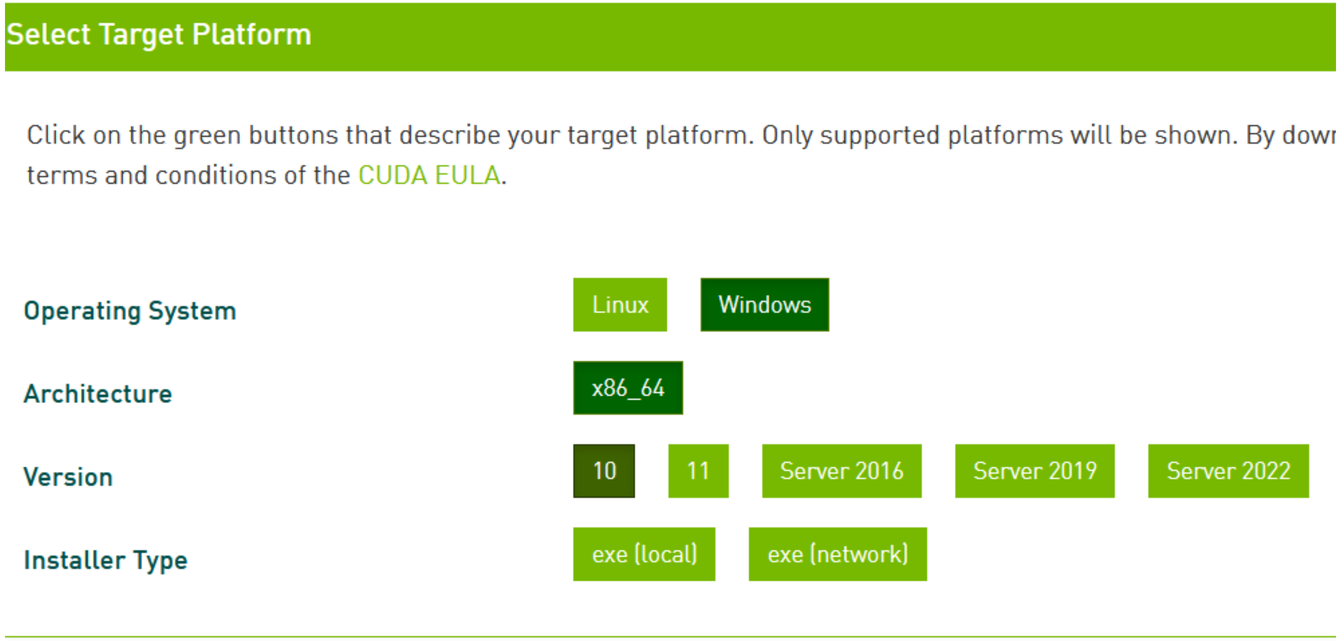
- Check if your GPU is in the list of acceptable GPUs. https://developer.nvidia.com/cuda-gpus
- If so, install CUDA (check that you CUDA version number matches the one of your PyTorch install!) https://developer.nvidia.com/cuda-downloads

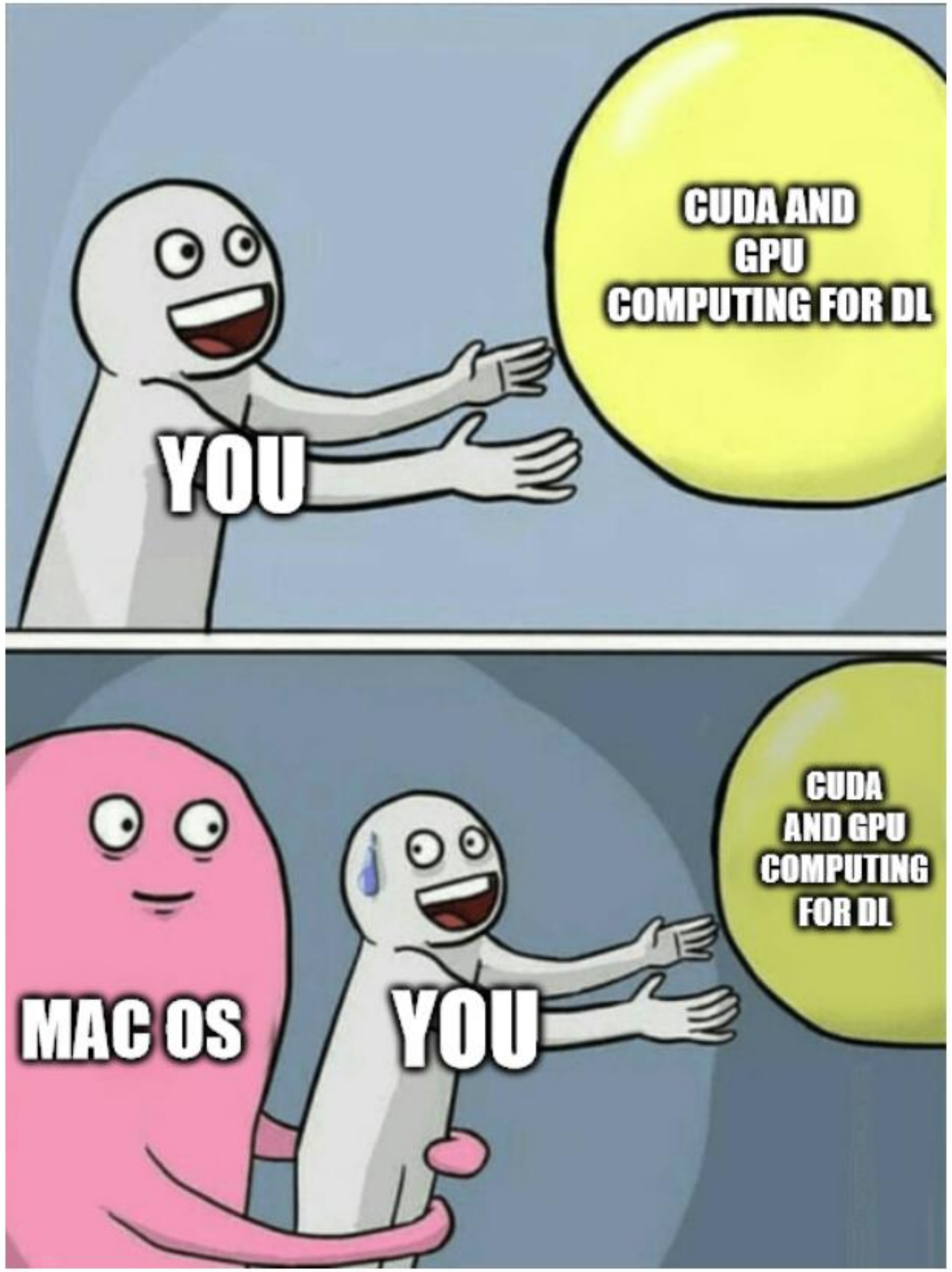


Installing PyTorch and CUDA

Question: How to use CUDA with MacOS?

Efforts are being made to make GPU computing compatible with AMD and MacOS GPUs, but not the best (afaik).



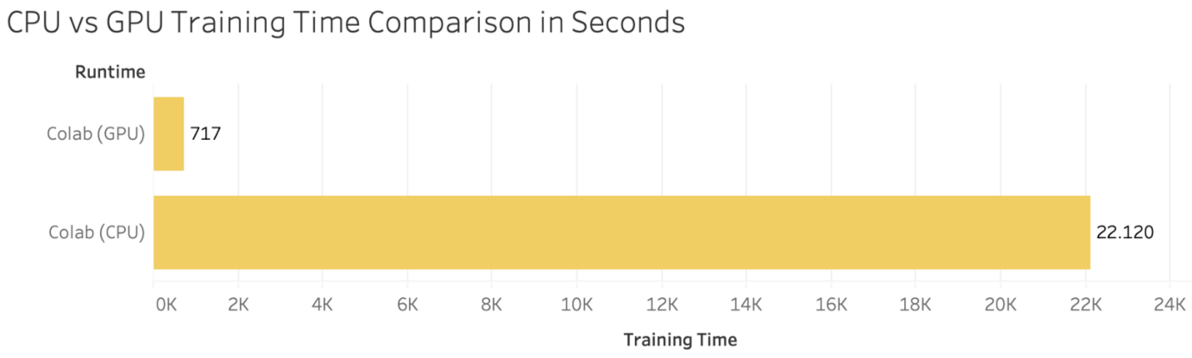


Installing PyTorch and CUDA

Your GPU not in the list of CUDA-enabled GPUs?

You bought a MacOS laptop?

- **Most notebooks can still run on CPU (but they might take significantly longer to run).**
- Also, always the option of using **Google Colab**, or create an education account on AWS.



Checking your PyTorch and CUDA install

- **Hello World for PyTorch:** to check you have PyTorch installed correctly. The code below should run and display a tensor.

```
import torch
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

- **Hello World for CUDA:** to check you have correctly installed CUDA on top of PyTorch. The code below should print *True*, as the output of *torch.cuda.is_available()*

```
import torch
torch.cuda.is_available()
```

A quick word about PyTorch 2.0

PyTorch v2.0 is the current (latest version released in March 2023).

- Should be fully retro-compatible.
- Revising and testing notebooks (they were designed last year and tested on v1.13).
- Kindly let me know if you have any issues!
- Will let you know about new v2.0 features on W13.

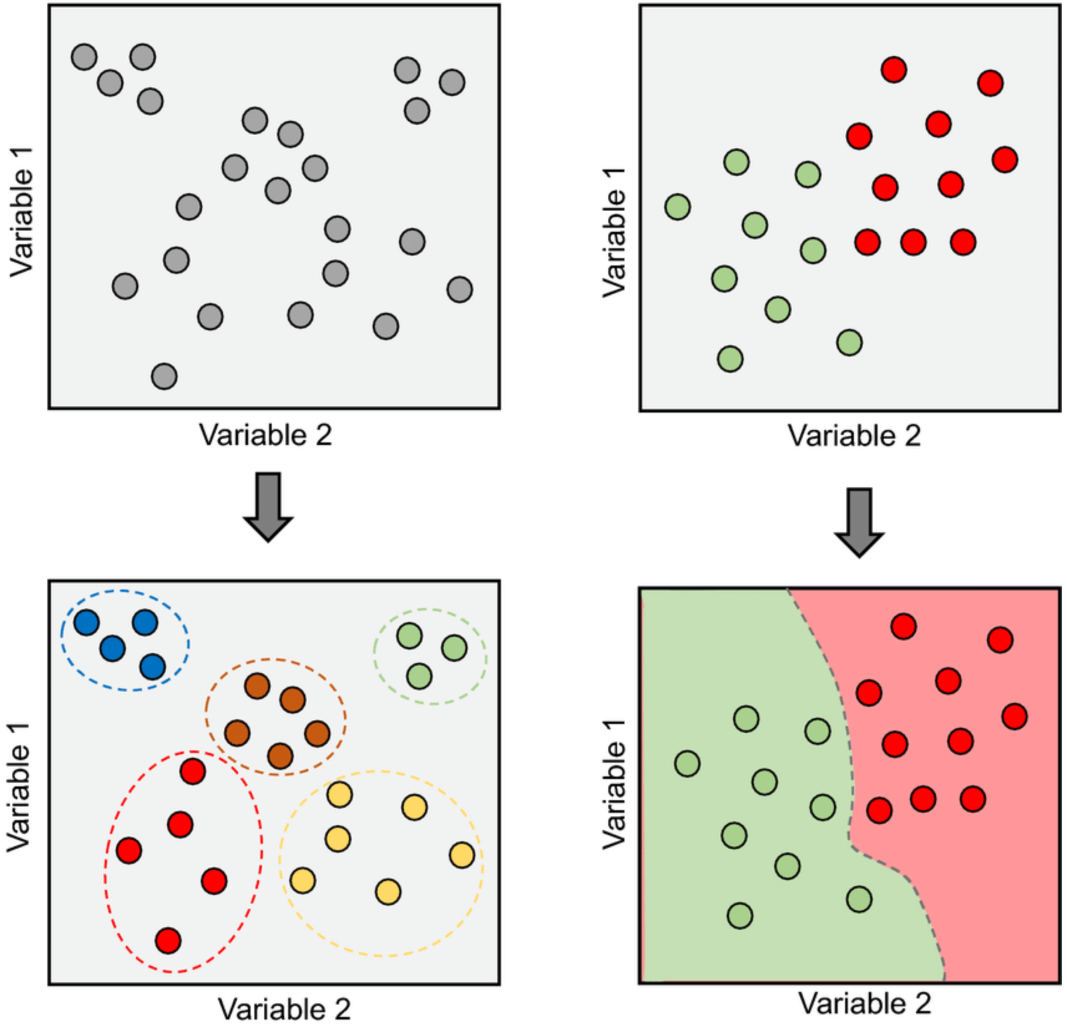
Supervised vs Unsupervised Learning

Definition (Supervised vs. Unsupervised Learning):

Supervised and **Unsupervised Learning** are the two techniques of ML.

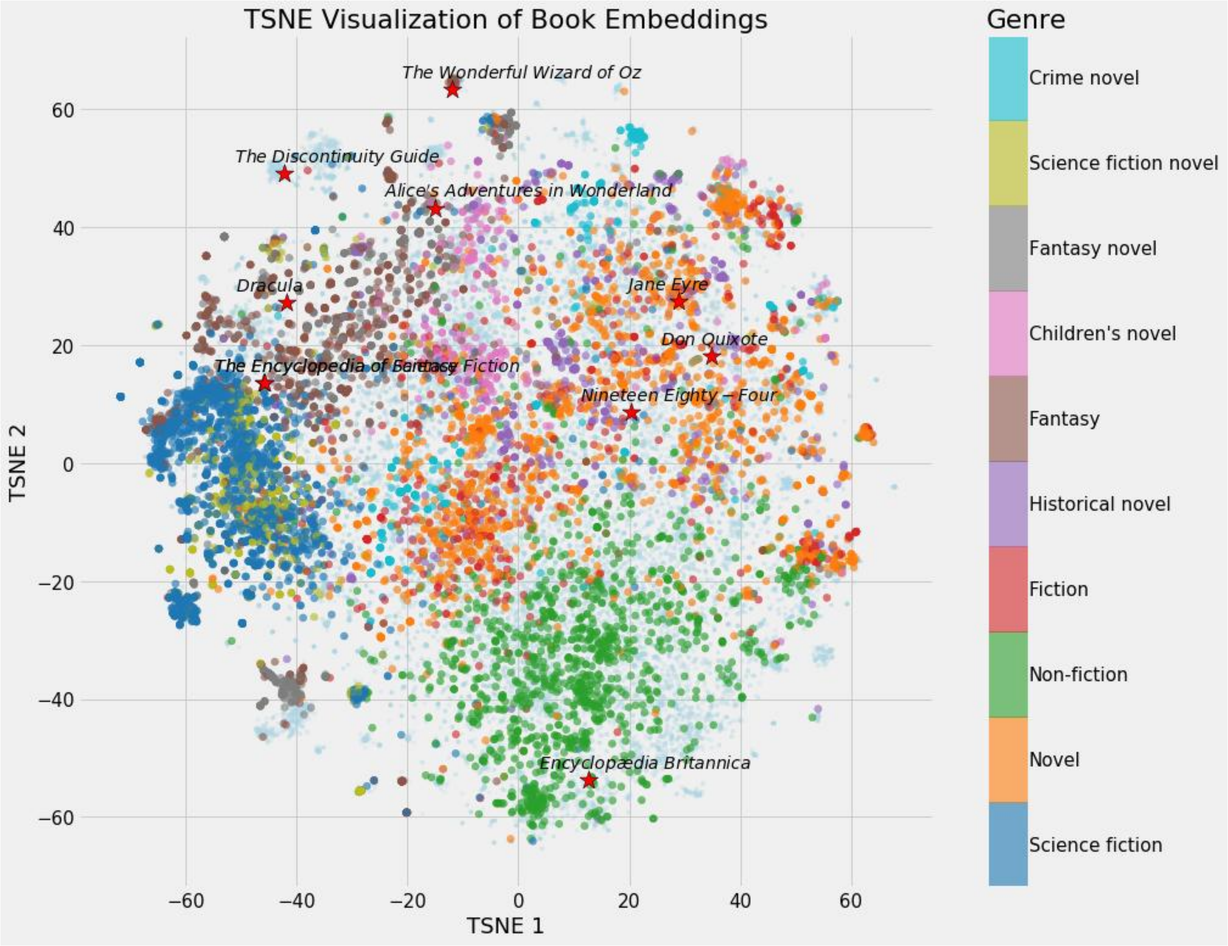
The main difference is **the need for labelled training data**:

- Supervised machine learning relies on labelled input and output data to learn and make predictions,
- while **unsupervised learning does not require labelled data.**



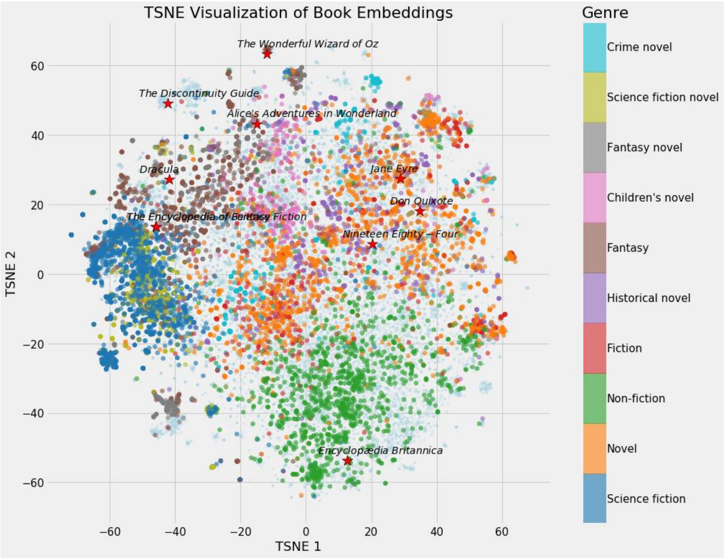
Unsupervised Learning Examples

- Examples of **unsupervised learning:** clustering customers based on their behaviours, segmenting images, and identifying topics in a document.

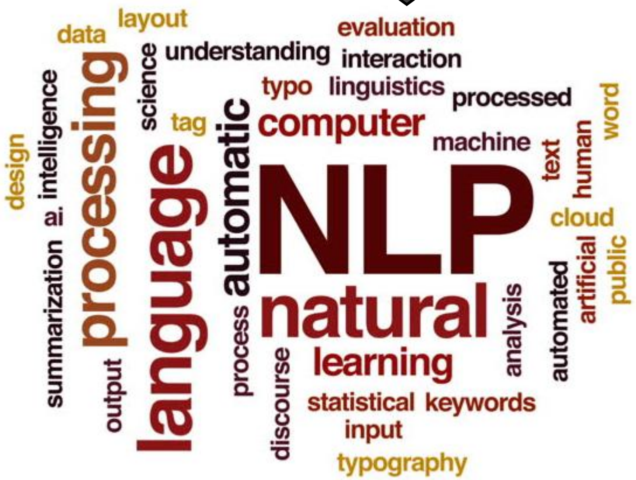


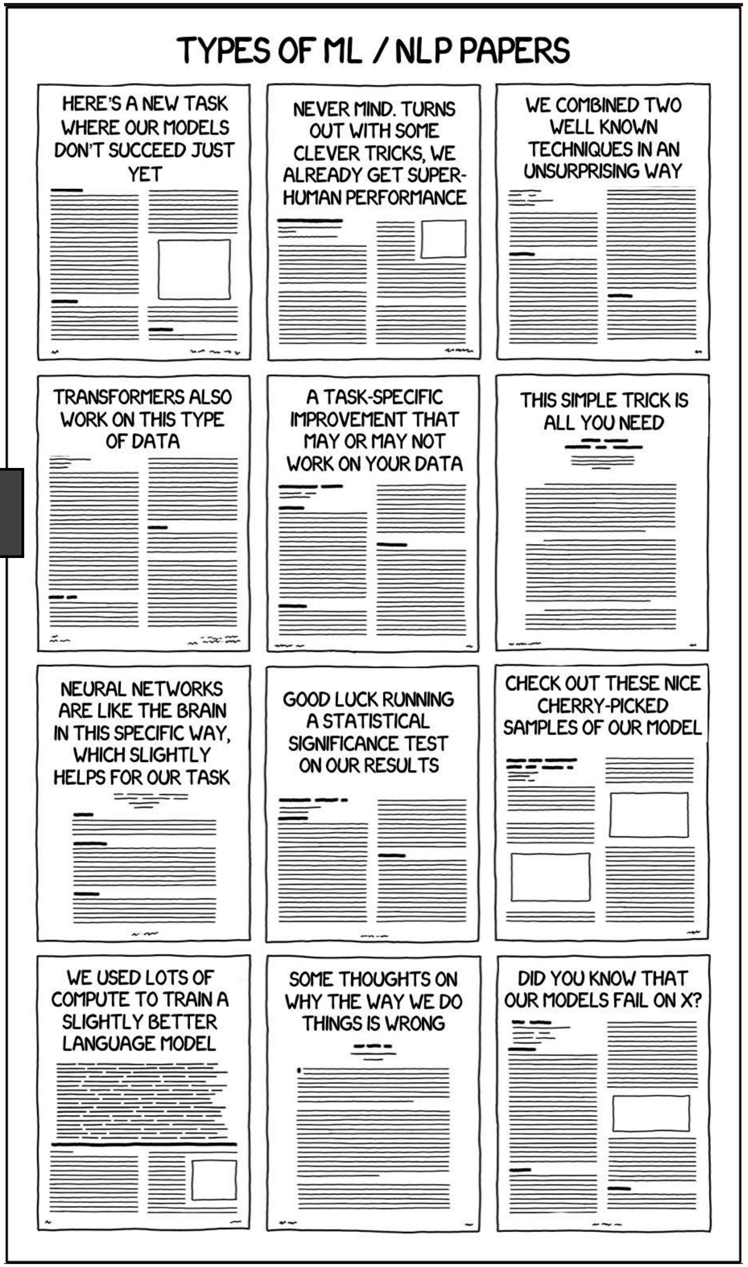
Unsupervised Learning Examples

- Examples of **unsupervised learning:** clustering customers based on their behaviours, segmenting images, and identifying topics in a document.



Identify topics that seem to appear in those papers and their prominence





Clustering vs. Association (Unsupervised)

Definition (Clustering vs. Association):

Unsupervised learning can be used for two types of problems: **Clustering** and **Association**.

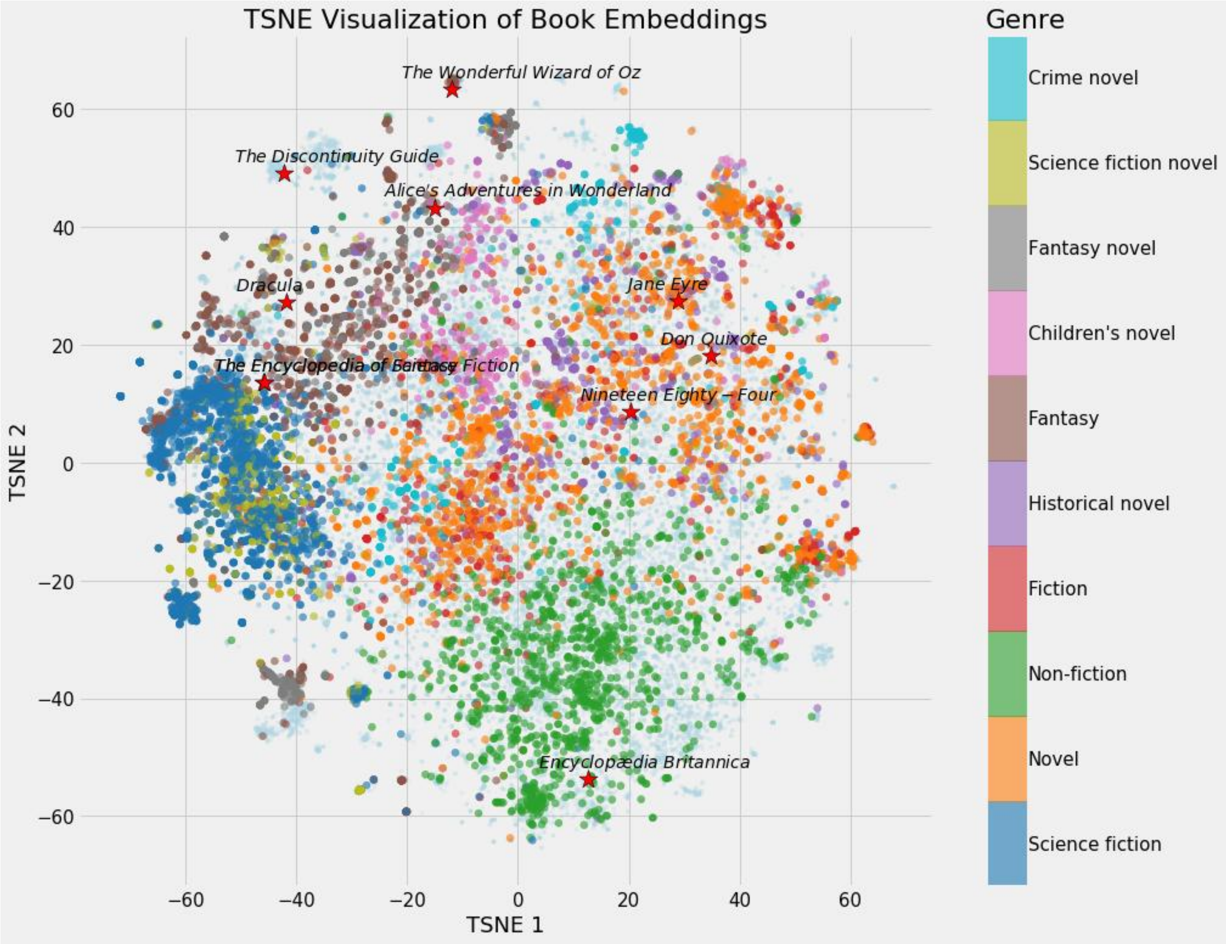
The main difference is that

- **clustering groups data based on similarity**,
- while **association looks for relationships between variables**.

Clustering vs. Association (Unsupervised)

Examples of clustering:

- Group customers into clusters based on the similarities in their purchasing behaviours, such as their spending habits or types of products they like.
- Group the catalogue of movies available on Netflix and identify what movies have in common, so you can later recommend a movie to a user based on what he/she has seen before.



Clustering vs. Association (Unsupervised)

Example of association:

- Finding the relationship between customer buying behaviors and customer satisfaction.
- Association can be used to understand how changes in customer behavior impacts customer satisfaction, such as if buying a certain product increases customer satisfaction (or the opposite).

Proof of the Normal Equation

Using the notations

$$X = \begin{pmatrix} 1 & x_1 \\ \dots & \dots \\ 1 & x_N \end{pmatrix},$$

$$Y = \begin{pmatrix} y_1 \\ \dots \\ y_N \end{pmatrix},$$

$$W^* = \begin{pmatrix} b^* \\ a^* \end{pmatrix}$$

We have, in matrix form

$$XW^* = \begin{pmatrix} a^* x_1 + b^* \\ \dots \\ a^* x_N + b^* \end{pmatrix}$$

And, then it follows that

$$XW^* - Y = \begin{pmatrix} a^* x_1 + b^* \ - y_1 \\ \dots \\ a^* x_N + b^* - y_N \end{pmatrix}$$

Restricted

Proof of the Normal Equation

Our loss function, the MSE L, is therefore defined, in matrix format as:

$$L = \frac{1}{N}(XW^* - Y)^T(XW^* - Y)$$

This is equivalent to

$$L = \frac{1}{N}\left((XW^*)^T - Y^T\right)(XW^* - Y)$$

Proof of the Normal Equation

We then get

$$L = \frac{1}{N}\left({W^*}^T X^T - Y^T\right)(XW^* - Y)$$

Expanding

$$L = \frac{1}{N}({W^*}^T X^T X W^* - Y^T X W^* - {W^*}^T X^T Y + Y^T Y)$$

Proof of the Normal Equation

Recall that we are trying to find the solution to optimization problem

$$(a^*, b^*) = \arg\min_{a,b}[L(a, b, x, y)]$$

With

$$L = \frac{1}{N}({W^*}^T X^T X W^* - Y^T X W^* - {W^*}^T X^T Y + Y^T Y)$$

Recognizing that $Y^T X W^* = {W^*}^T X^T Y$, differentiating our function L with respect to W and equating it to zero gives:

$$\frac{\partial L}{\partial W} = \frac{1}{N}(2X^T X W - 2X^T Y) = 0$$

Proof of the Normal Equation

This is equivalent to

$$(X^T X W^* - X^T Y) = 0$$

Or,

$$X^T X W^* = X^T Y$$

Assuming $\boldsymbol{X^T X}$ is invertible (and that might not always be the case based on your dataset!), this finally gives:

$$\boldsymbol{W^* = \left(X^T X\right)^{-1} X^T Y}$$