50.039 Theory and Practice of Deep Learning

W1-S3 Introduction and Machine Learning Reminders

Matthieu De Mari

About this week (Week 1)

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**?
6. What is **polynomial regression** and how to implement it?
7. What is **regularization** and how to implement it in **Ridge regression**?

About this week (Week 1)

8. What is **overfitting** and why is it bad?
9. What is **underfitting** and why is it bad?
10. What is **generalization** and how to evaluate it?
11. What is a **train-test split** and why is it related to **generalization**?
12. What is a **sigmoid** function? What is a **logistic** function?
13. How to perform **binary classification** using a **logistic regressor** and how is it related to linear regression?

About this week (Week 1)

14. What are **Neural Networks** and how do they relate to the **biology of a human brain**?
15. What is a **Neuron** in a Neural Network and how does it relate to linear/logistic regression?
16. What is the **difference** between a **shallow** and a **deep neural network**?
17. How to **implement a shallow Neural Network** manually and define a **forward propagation** method for it?
18. How to **train a shallow Neural Network** using **backpropagation**? How to define **backward propagation** and **trainer** functions?

Introducing the sigmoid function

Definition (the sigmoid function):

The **sigmoid function** is an important function used in Machine Learning. It is defined as

$$s(x) = \frac{1}{1 + \exp(-x)}$$

Or, equivalently

$$s(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Note: Sometimes, the notation $\sigma(x)$ is used instead of $s(x)$.

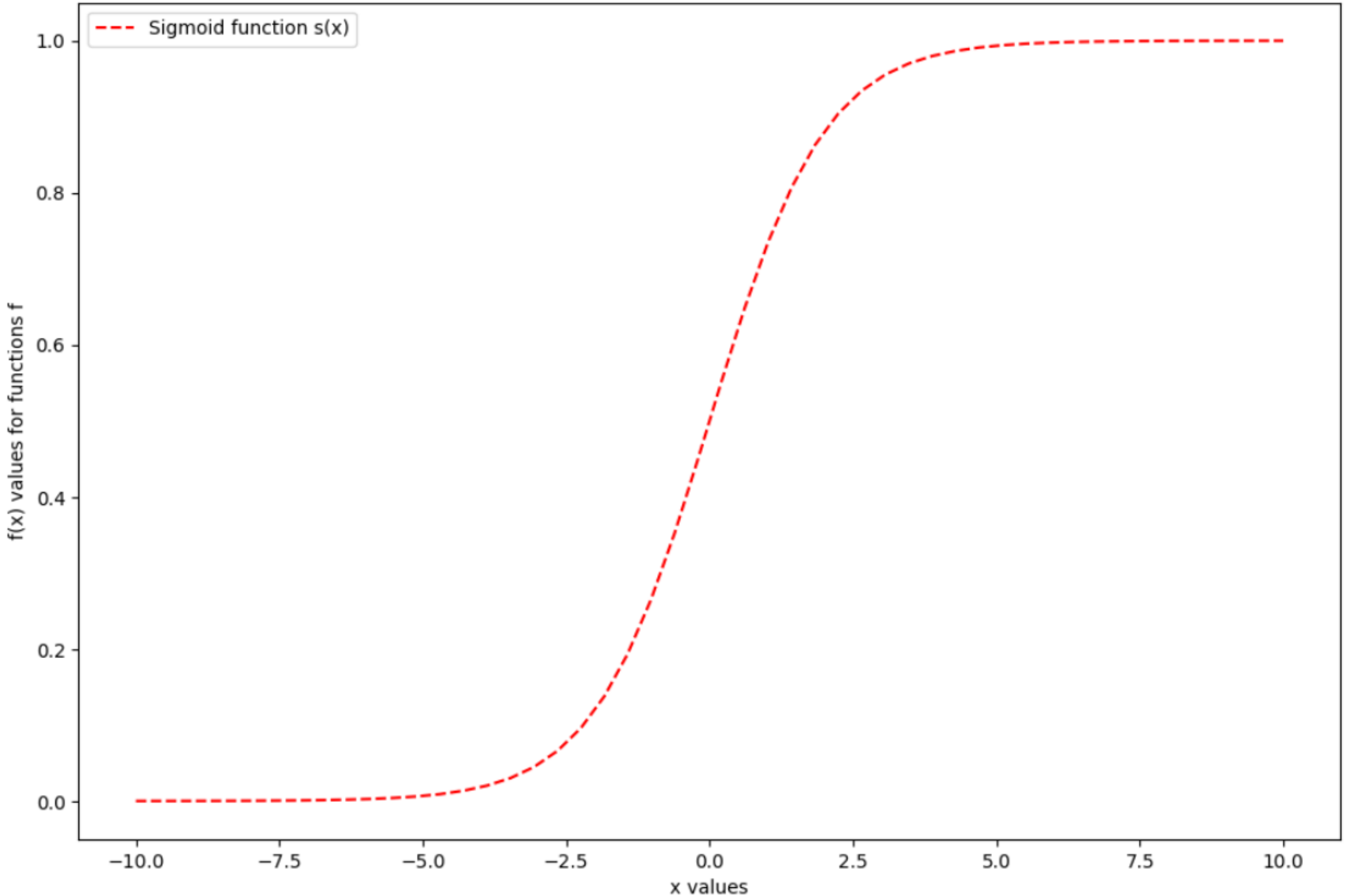
The sigmoid function has the following properties:

$$\forall\, x, \qquad 0 < s(x) < 1$$

$$\lim_{x \to -\infty} s(x) = 0$$

$$\lim_{x \to \infty} s(x) = 1$$

$$s(0) = 0.5$$


Sigmoid function s(x)
1.0
0.8
0.6
0.4
0.2
0.0
f(x) values for functions f
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x values

Introducing two logistic functions

Definition (the logistic functions):

Similarly, let us introduce two **logistic functions**, below:

$$l(x) = \ln(x)$$

$$l_2(x) = \ln(1 - x)$$

These two functions also have interesting properties.

$$\lim_{x \to 0} l(x) = -\infty$$

$$\lim_{x \to 1} l(x) = 0$$

$$\lim_{x \to 0} l_2(x) = 0$$

$$\lim_{x \to 1} l_2(x) = -\infty$$

$$l(0.5) = l_2(0.5) = -\ln(2)$$

To recap

Definition (the sigmoid function):

$$s(x) = \frac{1}{1 + \exp(-x)}$$

Or, equivalently

$$s(x) = \frac{\exp(x)}{1 + \exp(x)}$$

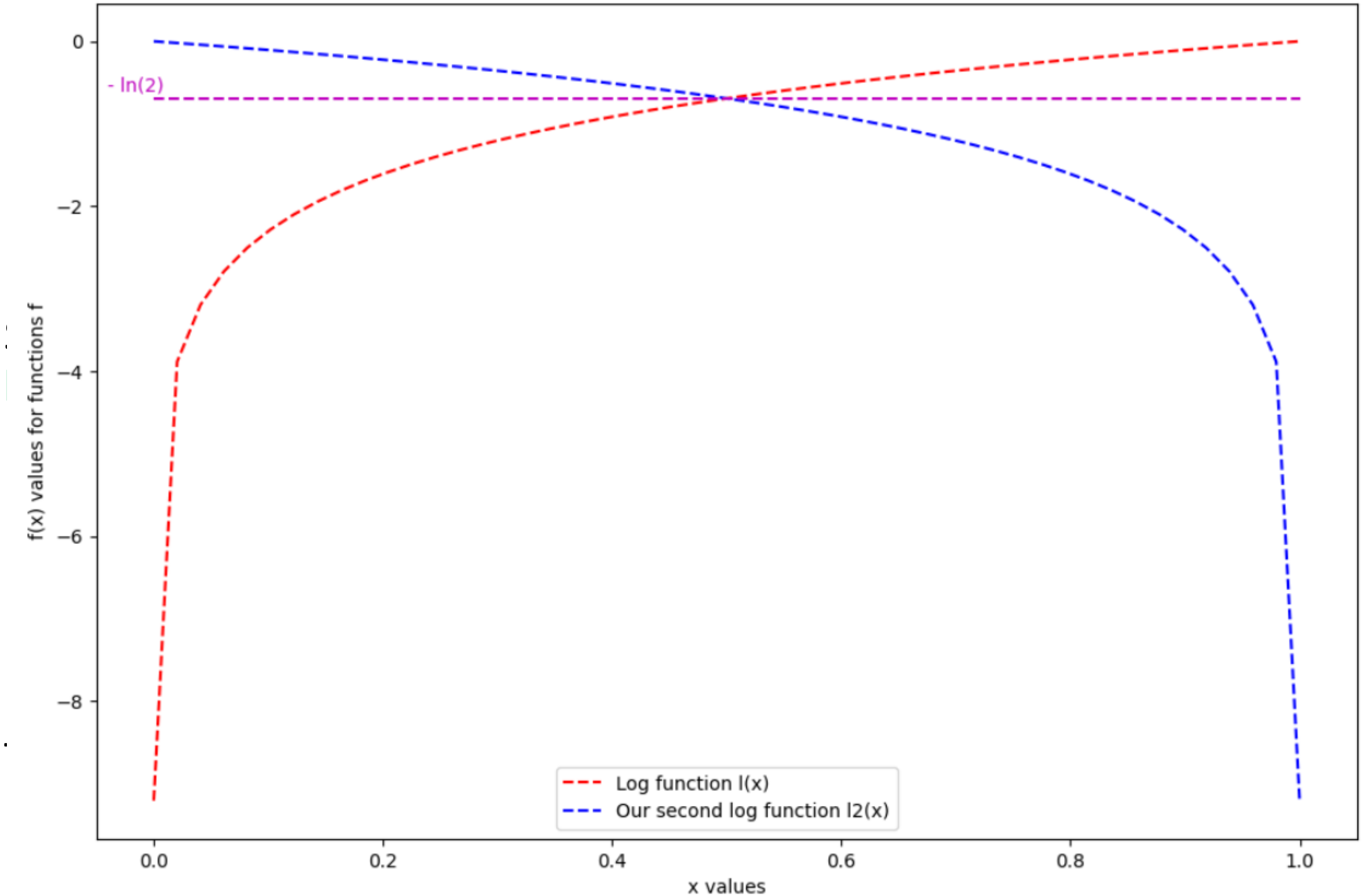
Definition (the logistic functions):

$$l(x) = \ln(x)$$

And,

$$l_2(x) = \ln(1 - x)$$

```
1  def s(x):
2      return 1/(1 + np.exp(-x))
3  def l(x):
4      return np.log(x)
5  def l2(x):
6      return np.log(1 - x)
```


0
- ln(2)
−2
−4
−6
−8
f(x) values for functions f
0.0
0.2
0.4
0.6
0.8
1.0
x values
Log function l(x)
Our second log function l2(x)

Introducing two logistic functions

In addition, the following properties hold (some of them require using L'Hospital rule to prove them, try it out?).

$$\lim_{x \to 0} x \, ln(x) = 0$$

$$\lim_{x \to 1} x \, ln(x) = 0$$

$$\lim_{x \to 0} (1 - x) \, ln(1 - x) = 0$$

$$\lim_{x \to 1} (1 - x) \, ln(1 - x) = 0$$

Introducing two logistic functions

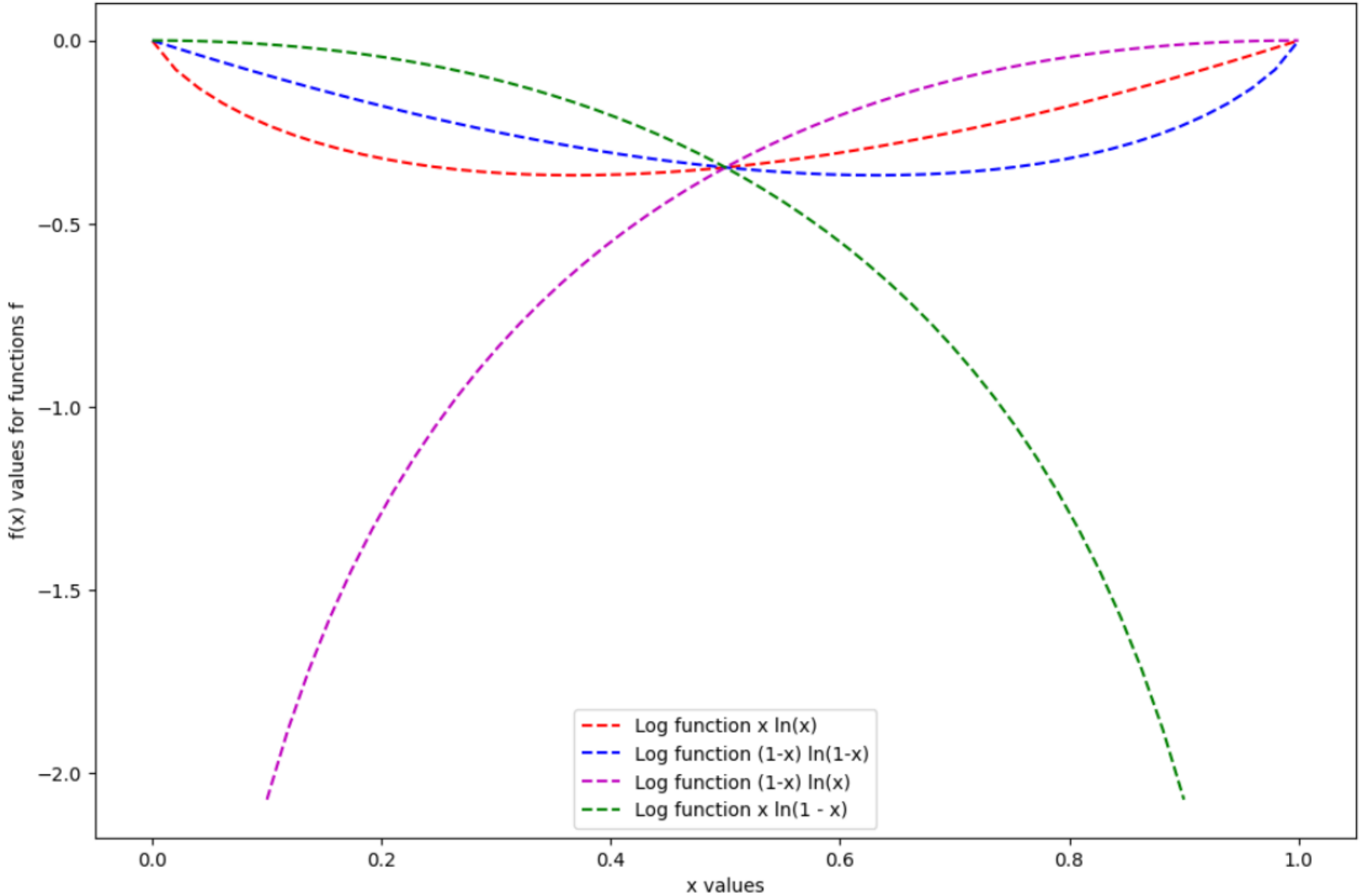
In addition, the following properties hold (some of them require using L'Hospital rule to prove them, try it out?).

$$\lim_{x \to 0} (1 - x)\, ln(x) = -\infty$$

$$\lim_{x \to 1} (1 - x)\, ln(x) = 0$$

$$\lim_{x \to 0} x\, ln(1 - x) = 0$$

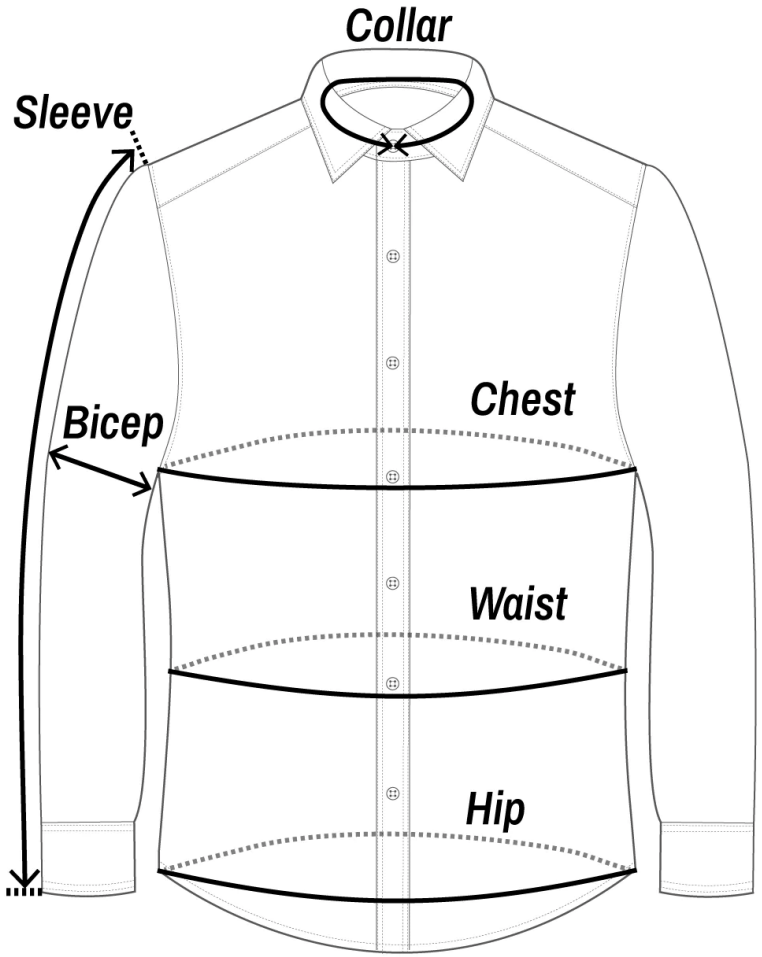
$$\lim_{x \to 1} x\, ln(1 - x) = -\infty$$


0.0
−0.5
−1.0
−1.5
−2.0
f(x) values for functions f
0.0
0.2
0.4
0.6
0.8
1.0
x values
Log function x ln(x)
Log function (1-x) ln(1-x)
Log function (1-x) ln(x)
Log function x ln(1 - x)

Mock dataset generation

In notebook 6, we will prepare a dataset for a **single-variable binary classification.**

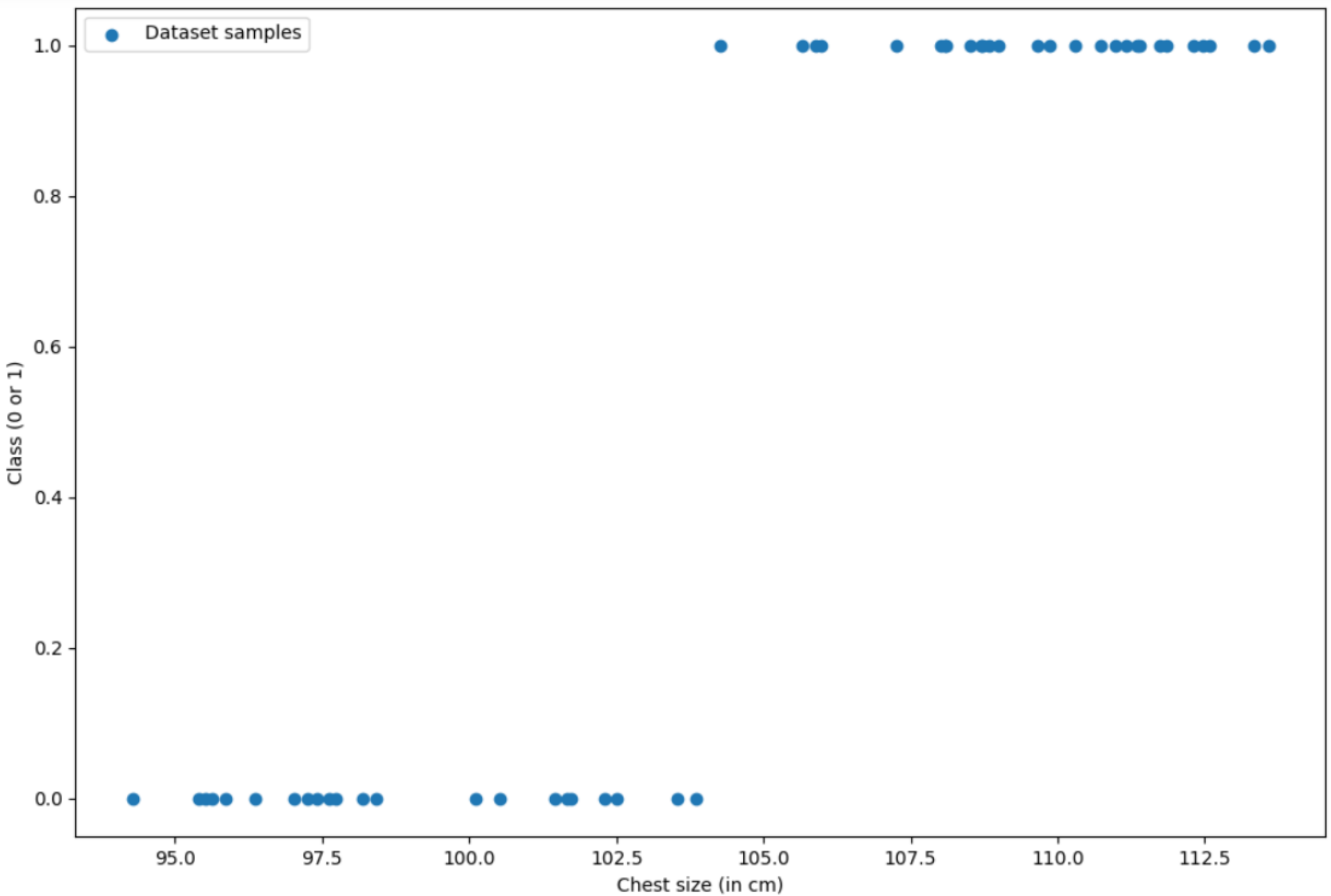
- The **inputs**, x, will consist of chest sizes for male between 94cm and 114cm, randomly generated.
- The **outputs**, y, will consists of **two classes**, corresponding to the shirt size to use, with two possible values:
  - M (class with value 0, if chest size is below 104 cm) size
  - and L size (class with value 1, if chest size is above 104 cm).



```python
# All helper functions
def chest_size(min_size, max_size):
    return round(np.random.uniform(min_size, max_size), 2)
def shirt_size(size, threshold):
    return int(size >= threshold)
def generate_datasets(n_points, min_size, max_size, threshold):
    x = np.array([chest_size(min_size, max_size) for _ in range(n_points)])
    y = np.array([shirt_size(i, threshold) for i in x])
    return x, y
```

```python
# Dataset generation (n_points points will be generated).
# We will use a seed for reproducibility.
min_size = 94
max_size = 114
threshold = 104
np.random.seed(27)
n_points = 50
inputs, outputs = generate_datasets(n_points, min_size, max_size, threshold)
print(inputs)
print(outputs)
```

```
[102.51 110.29 108.71 111.36 101.67 113.59 111.86  98.19 108.84 107.26
 111.74 111.16 108.99 111.4   97.74 100.51 101.46 109.87  97.02  97.4
  95.62 100.1  109.67  97.26  95.41 108.02  97.62 105.98 102.31 104.27
  98.41 108.51 110.99 112.58 108.72 103.53 103.86 105.89  95.52  96.35
 113.33 105.67  95.85  94.27 110.74 112.3  108.09 101.74 108.11 112.47]
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
```


Dataset samples
1.0
0.8
0.6
0.4
0.2
0.0
Class (0 or 1)
95.0
97.5
100.0
102.5
105.0
107.5
110.0
112.5
Chest size (in cm)

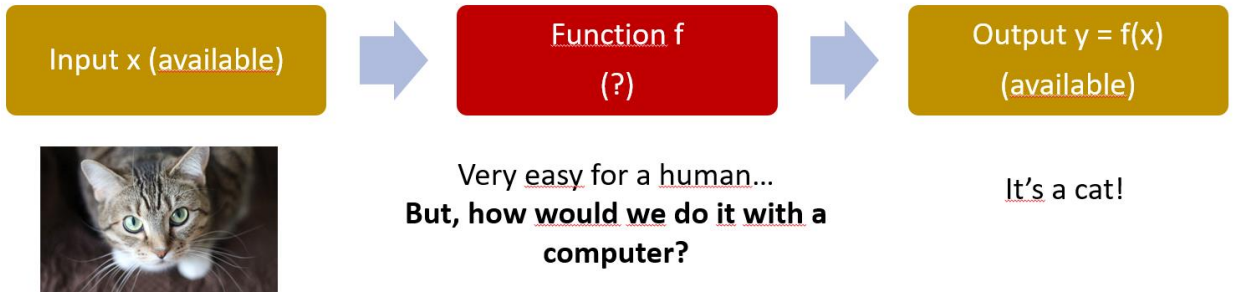
A quick word about expert systems

Definition (expert systems):

An **expert system** is a computer program that mimics the decision-making abilities of a human expert in a specific domain or field.

It uses knowledge represented in a symbolic form, such as rules or ontologies, to make inferences and decisions based on the data it receives.

Expert system provide a high level of performance and accuracy in solving problems or making decisions but these are difficult to design as they require human expertise.



A quick word about expert systems

Definition (expert systems):

An **expert system** is a computer program that mimics the decision-making abilities of a human expert in a specific domain or field.

It uses knowledge represented in a symbolic form, such as rules or ontologies, to make inferences and decisions based on the data it receives.

Important note: in our case, we could define as expert system like so, but most of the time the datasets will not have a clear logic and you will have to rely on ML algorithms instead, basically "praying for the AI to figure out a logic that works".

```
# A simple expert system
def expert_system(inputs):
    return [int(size > 104) for size in inputs]
```

```
# Try it and compare results to see we have 100% accuracy here!
pred = expert_system(inputs)
print((pred == outputs).all())
```

```
True
```

From Linear Regression to Logistic Regression

Definition(Logistic Regression):

The **logistic regression** model assumes that the classes y_i for every sample x_i are connected via the formula below.

$$\forall i, y_i = \begin{cases} 1 & if\ p(x_i) > 0.5 \\ 0 & otherwise \end{cases}$$

This value $p(x)$ is often referred to as the **probability** of sample with value x being of class 1 (that is L size).

Respectively, $1 - p(x)$ is therefore the **probability** of sample with value x being of class 0 (that is M size).

The **probability** function is then simply defined as

$$p(x) = s(ax + b)$$

The function p always has values between 0 and 1, thanks to the sigmoid function s we defined earlier.

From Linear Regression to Logistic Regression

Definition(Logistic Regression):

The **logistic regression** model assumes that the classes y_i for every sample x_i are connected via the formula below.

$$\forall i, y_i = \begin{cases} 1 & if\ s(a\,x\ +\ b) > 0.5 \\ 0 & otherwise \end{cases}$$

This value $s(ax + b)$ is often referred to as the **probability** of sample with value x being of class 1 (that is L size).

This model has **two trainable parameters**, $\boldsymbol{a}$ and $\boldsymbol{b}$, to be decided like in the linear regression.

In fact, we could see the logistic regression as the combination of linear regression and sigmoid together.

In addition, note that the function $f(x) = a\,x + b$ is also often referred to as the **logit** function with value x.

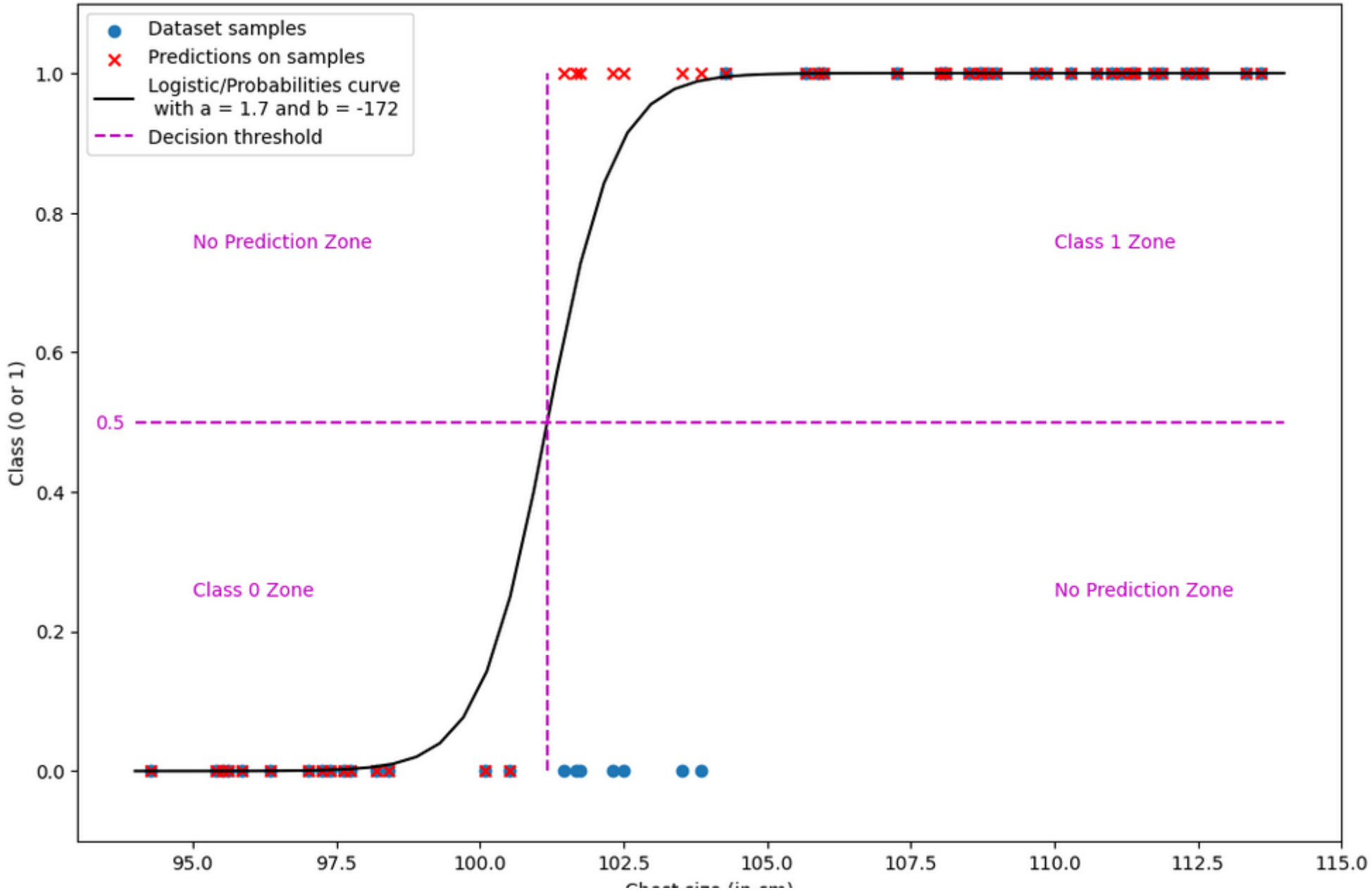
From Linear Regression to Logistic Regression

We can implement the logistic regression below and try it out with some values a and b!

```
def logistic_regression(x, a, b):
    return s(a*x + b)
def predict_samples(inputs, a, b):
    return [int(logistic_regression(x, a, b) >= 0.5) for x in inputs]
def logistic_regression_matplotlib(a, b, min_size, max_size):
    x_plt = np.linspace(min_size, max_size, 50)
    y_plt = np.array([s(a*x + b) for x in x_plt])
    return x_plt, y_plt
```

```
# Trying to predict with our logistic regression
# model, for two given values of a and b.
a = 1.7
b = -172
pred_y = predict_samples(inputs, a, b)
print(pred_y)
```

```
[1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
1, 1, 0, 0, 1, 1, 1, 1, 1, 1]
```



Training a logistic regressor

- As before, in linear regression, we will have to **train the model**, i.e. choose values for a and b, that fit the dataset.
- While **MSE** seemed to work as a **loss function** to train the **linear regression** model, it will simply not do here.
- We need something else.

- In logistic regression, we **minimize** a function, called the **log-likelihood** function, instead.
- This function has to do with the two logistic functions we have introduced earlier.

The log-likelihood function and loss

Definition (the log-likelihood function):

The **log-likelihood function** is defined as

$$L = \frac{1}{N}\sum_{i}^{N}\left[y_i \ln\left(s(ax_i + b)\right) + (1 - y_i)\ln\left(1 - s(ax_i + b)\right)\right]$$

This is a good function to use for our problem for the two reasons:

- When $p(x_i)$ is close to the ground truth value y_i, both terms of the loss function, will have values close to 0. This follows from the properties we defined earlier for the logistic functions.
- On the opposite, the log-likelihood loss function will have a different behaviour when $p(x_i)$ and the ground truth value y_i are different, producing larger non-zero negative values.

The log-likelihood function and loss

Definition (using the log-likelihood function as a loss):

This log-likelihood loss function is therefore a good choice of a performance metric to measure the performance of our models.

Indeed, **maximizing this log-likelihood** loss function would be equivalent to finding the best choice of parameters a and b, and therefore the best fit.

In practice however, we **prefer to minimize loss functions**. This requires a simple adjustment, **multiplying the loss function by -1**.

$$a^*, b^* = \arg\min_{a,b} \left[\frac{-1}{N} \sum_{i}^{N} \left[x_i \ln\big(s(ax_i + b)\big) + (1 - x_i) \ln\big(1 - s(ax_i + b)\big) \right] \right]$$

Implementing the log-likelihood loss

- We can simply implement the log likelihood loss as shown below.
- We could look for the normal equation for logistic regression, but this simply will not work: indeed, it is simply impossible to solve the logistic regression problem analytically.
- (Try solving it analytically if you are not convinced?).

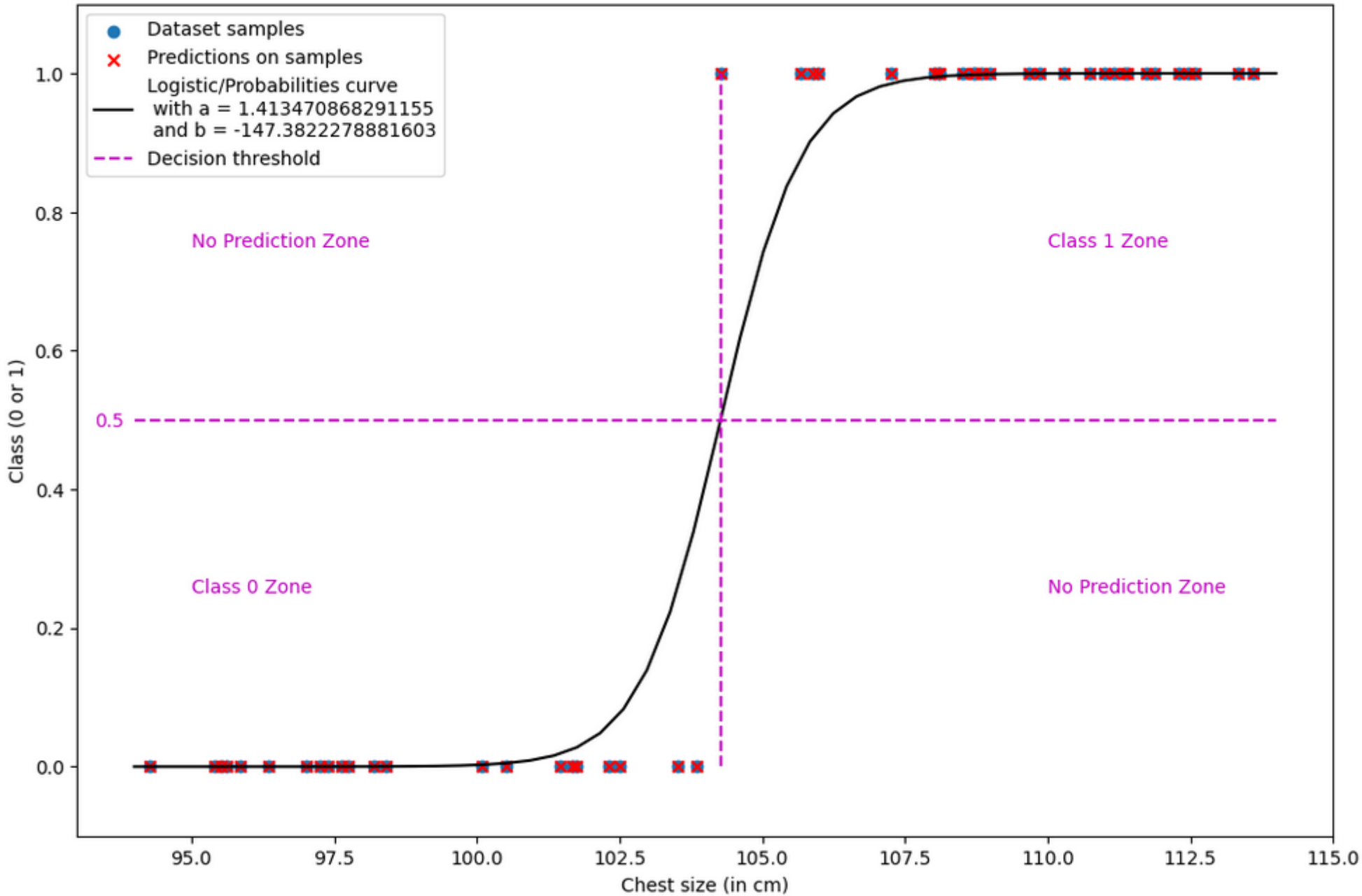
```
1  def log_likelihood_loss(a, b, x, y):
2      # Using Numpy broadcasting
3      return -1*np.mean(y*np.log(logistic_regression(x, a, b)) \
4                        + (1 - y)*np.log(1 - logistic_regression(x, a, b)))
```

Training a logistic regressor

- **Instead, it is often preferable to rely on heuristics, such as the gradient descent method, or the Newton method.**
- Typically, this is what the **LogisticRegression()** object in sklearn uses for training!
- (Not covered but feel free to try it out?)

```
# Reshaping inputs as a 2D array
# (As before, this is a requirement for sklearn models)
inputs_re = inputs.reshape(-1, 1)

# Create a logistic regression model, use the Newton method
# to find the best parameters a and b
logreg_model = LogisticRegression(solver = 'newton-cg')
logreg_model.fit(inputs_re, outputs)

# Test your model and inputs
pred_outputs = logreg_model.predict(inputs_re)
print(pred_outputs)
print(outputs)

# Retrieving coefficients a and b
a_sk = logreg_model.coef_[0, 0]
b_sk = logreg_model.intercept_[0]
print(a_sk, b_sk)
```

```
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
1.413470868291155 -147.3822278881603
```



Evaluating the logistic regression model

- When attempting to evaluate a classification model, it is often a good idea to first display a confusion matrix for the predicted values of the dataset samples.
- We can simply calculate it by using the confusion_matrix() function from sklearn. The non-diagonal elements have zero values, which is the sign that our model classifies perfectly.
- Learn more, here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html.

```
1 # Using a confusion matrix.
2 print(confusion_matrix(outputs, pred_outputs))
```

```
[[22  0]
 [ 0 28]]
```

```
1 # Using a classification report.
2 print(classification_report(outputs, pred_outputs))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        28

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```

Evaluating the logistic regression model

- We can also ask for a classification report for the model.
- This shows the precision, recall and F1 scores for each class, as well as the overall accuracy of the model.
- For each one of these metrics, the best value is 1.0, and our model exhibits it, suggesting again that the model classifies perfectly.
- Learn more, here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn.metrics.classification_report.

```
1 # Using a confusion matrix.
2 print(confusion_matrix(outputs, pred_outputs))
```

```
[[22  0]
 [ 0 28]]
```

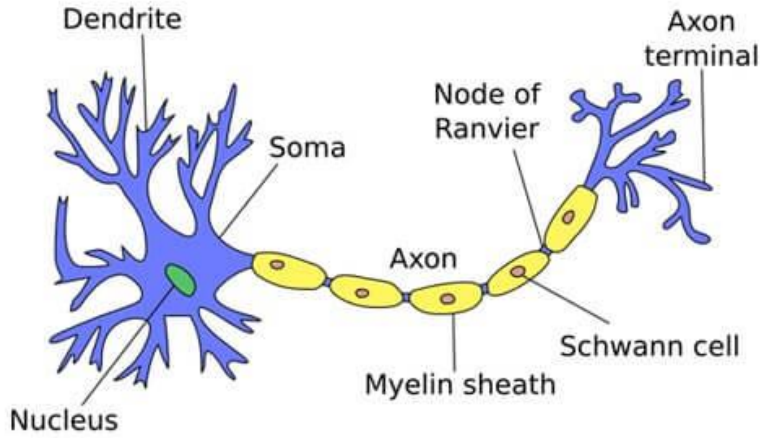
```
1 # Using a classification report.
2 print(classification_report(outputs, pred_outputs))
```

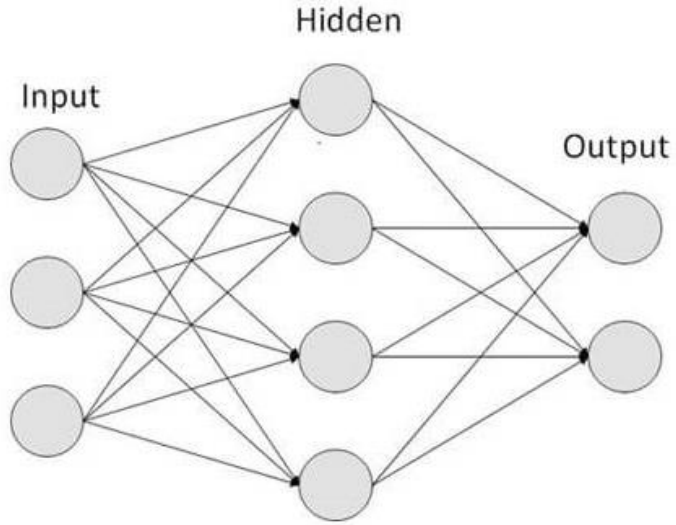
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        28

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```

A bit of biology

Question: Are neurons in Neural Networks really close to the actual human brain?

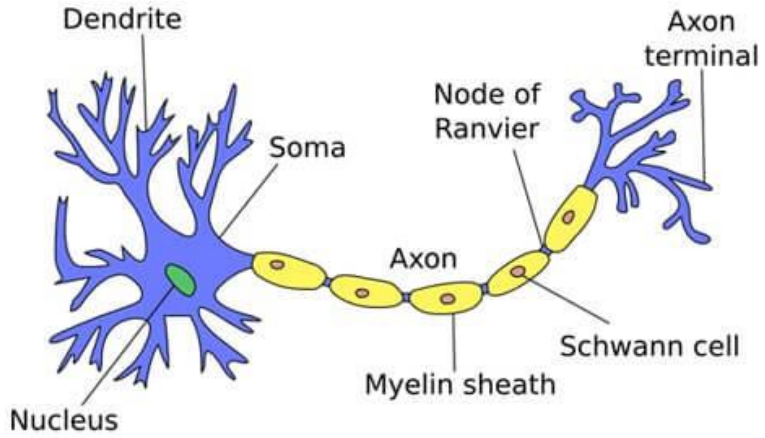
- Neural networks are inspired by the neural structure and processes of the human brain.
- Neural networks are comprised of interconnected nodes, which are called neurons.
- Neural networks are related to biology in that they are modeled after biological neural systems.

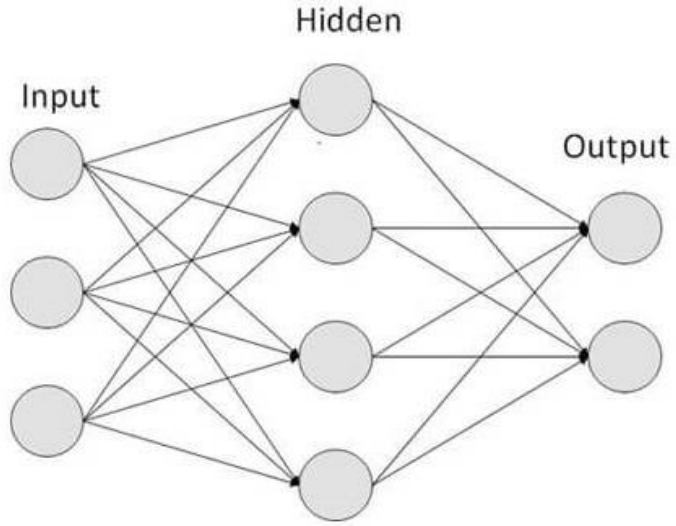




A bit of biology

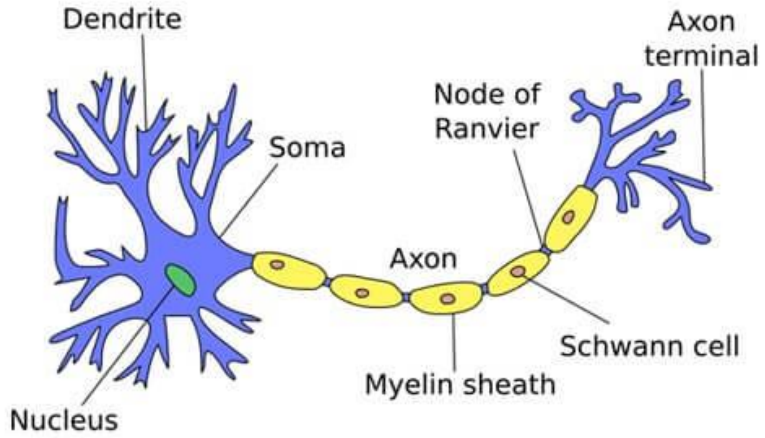
- The neurons in a neural network are similar to the neurons in the brain in that they are connected to each other and can send and receive signals.
- Furthermore, the training process of a neural network is analogous to the learning process of a biological neural network.
- The neurons are adjusted and adapted in order to learn, just as neurons in the brain are modified in order to learn new information.

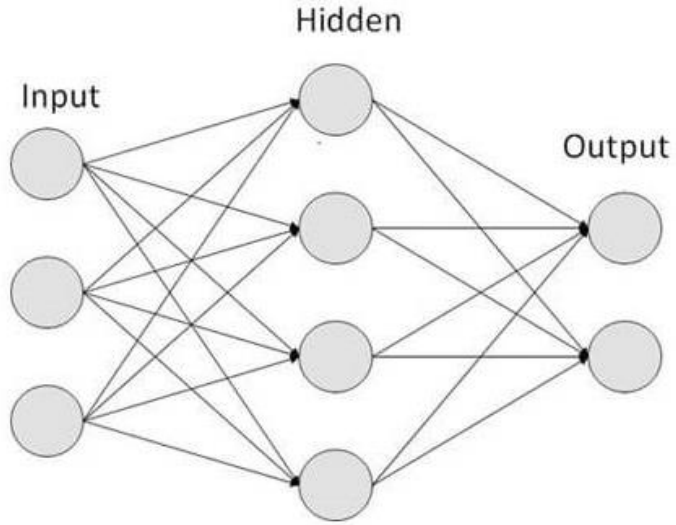




A bit of biology

- Research has also shown that artificial neural networks can be used to solve some of the same types of problems that biological neural networks can solve.
- For example, neural networks can be used to identify objects in images, just as biological neural networks can.
- Ongoing argument: see [Quanta2021] and [MITNews2022].





Starting with a mock dataset, again

As before, we will try to come up with a model capable of predicting the price of an apartment based on some of its features.

The inputs will this time consist of two parameters:

- the **surface** of the apartment, in sqm, just like before,
- and the **distance** between the apartment and the closest MRT station in meters.

We will generate a mock dataset by:

- randomly drawing **surfaces** between 40 sqm and 200sqm,
- randomly drawing **distances** between 50 and 1000m,
- generate prices by assuming that the average price is simply defined as $avgprice = 100000 + 14373 \times surface + (1000 - distance) \times 1286$

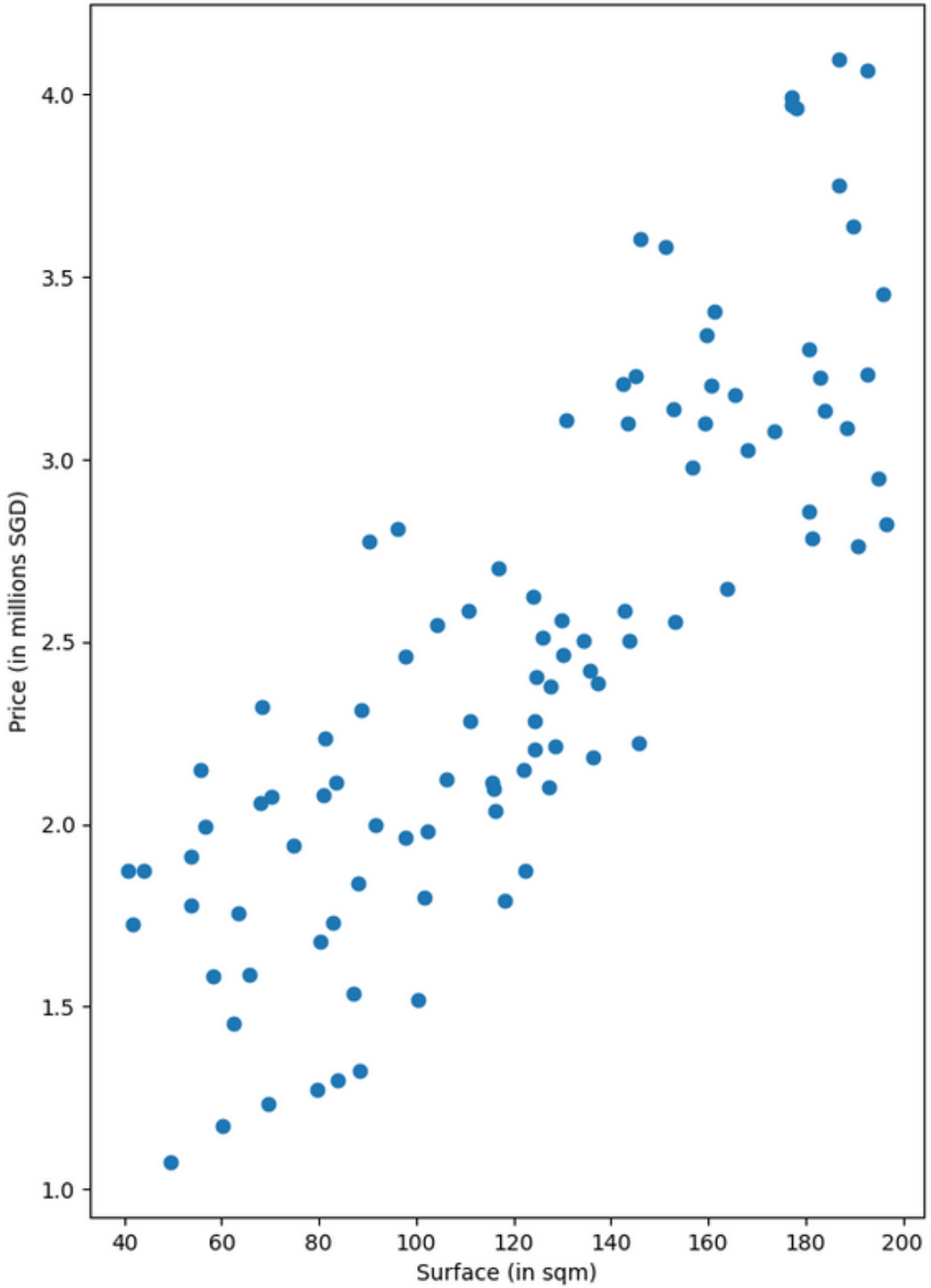
Finally, we will randomly apply a -/+ 10% variation on the average price to create some variance, and call that the output for a given sample.

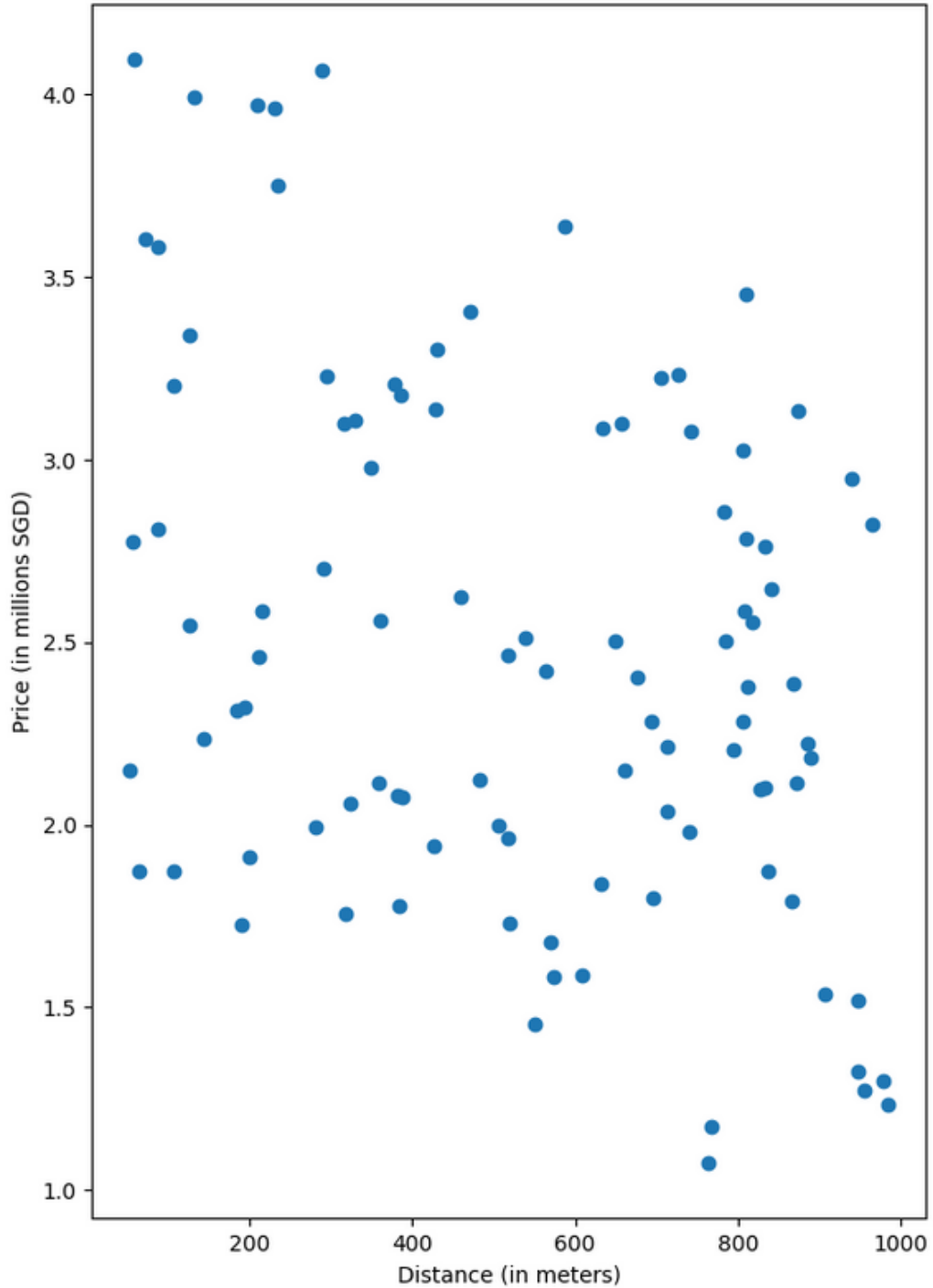
Starting with a mock dataset, again

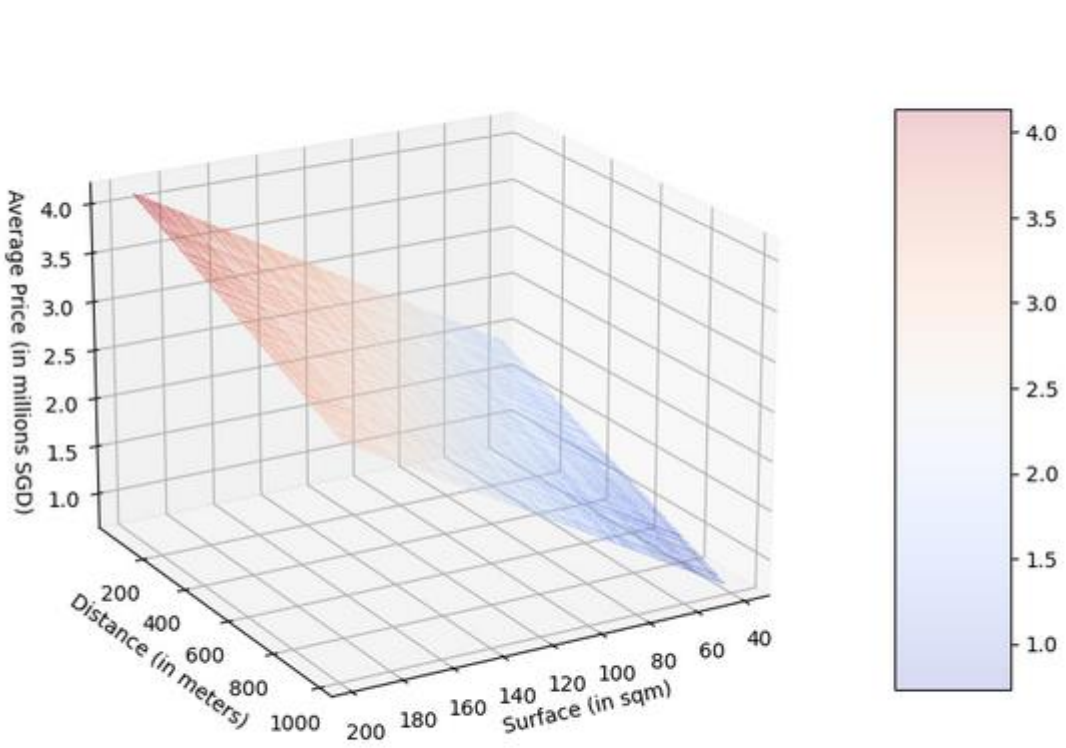
```python
# All helper functions
min_surf = 40
max_surf = 200
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
min_dist = 50
max_dist = 1000
def distance(min_dist, max_dist):
    return round(np.random.uniform(min_dist, max_dist), 2)
def price(surface, distance):
    return round((100000 + 14373*surface + (1000 - distance)*1286)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
n_points = 100
def create_dataset(n_points, min_surf, max_surf, min_dist, max_dist):
    surfaces_list = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    distances_list = np.array([distance(min_dist, max_dist) for _ in range(n_points)])
    inputs = np.array([[s, d] for s, d in zip(surfaces_list, distances_list)])
    outputs = np.array([price(s, d) for s, d in zip(surfaces_list, distances_list)]).reshape(n_points, 1)
    return surfaces_list, distances_list, inputs, outputs
```

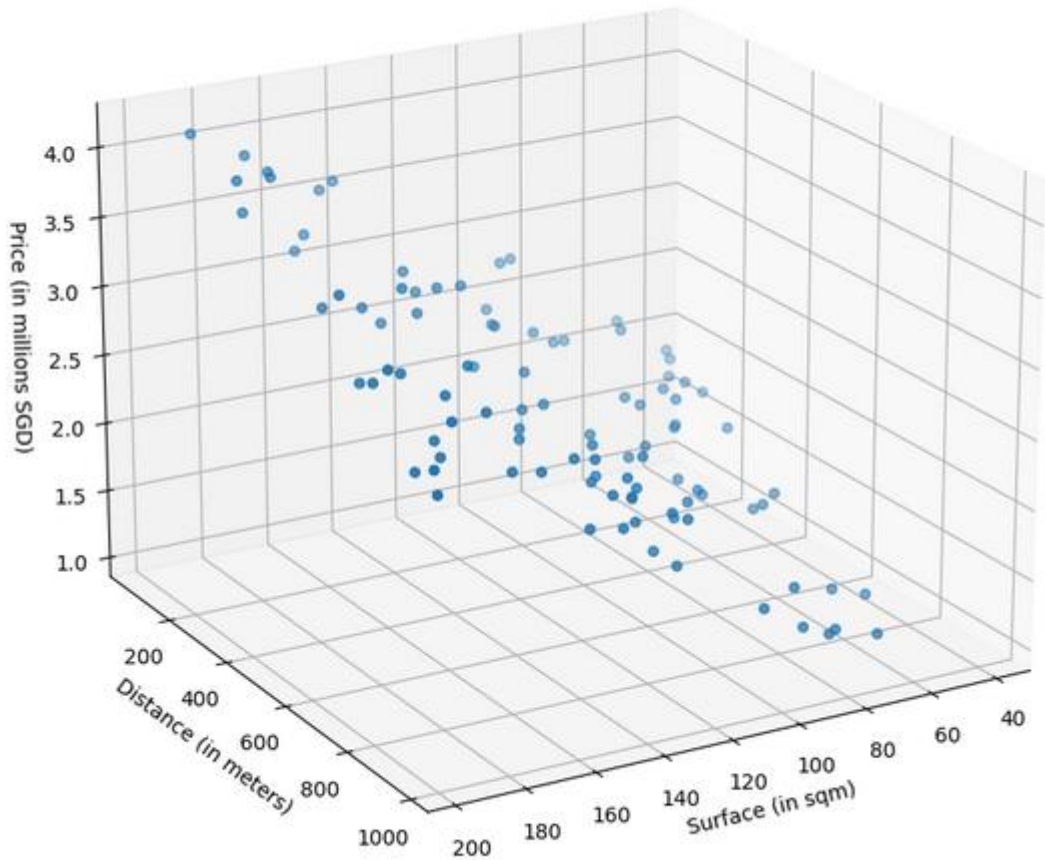
```python
# Generate dataset
np.random.seed(47)
surfaces_list, distances_list, inputs, outputs = create_dataset(n_points, \
                                                                min_surf, \
                                                                max_surf, \
                                                                min_dist, \
                                                                max_dist)
# Check a few entries of the dataset
print(surfaces_list.shape)
print(distances_list.shape)
print(inputs.shape)
print(outputs.shape)
print(inputs[0:10, :])
print(outputs[0:10])
```

```
(100,)
(100,)
(100, 2)
(100, 1)
[[ 58.16 572.97]
 [195.92 809.8 ]
 [156.6  349.04]
 [ 96.23  86.82]
 [153.22 817.92]
 [167.94 806.25]
 [143.29 315.92]
 [106.34 482.67]
 [152.96 427.77]
 [ 79.46 955.76]]
[[1.581913]
 [3.450274]
 [2.978769]
 [2.808258]
 [2.556398]
 [3.023983]
 [3.099523]
 [2.121069]
 [3.136544]
 [1.273443]]
```


Price (in millions SGD)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
40
60
80
100
120
140
160
180
200
Surface (in sqm)


Price (in millions SGD)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
200
400
600
800
1000
Distance (in meters)





Coding a Neuron and a Neural Network

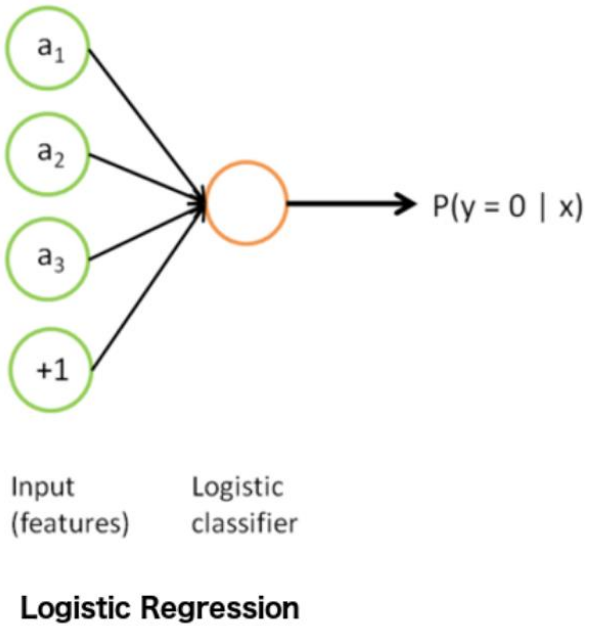
Definition (Neuron in Neural Networks):

Each artificial neuron in a neural network might for instance be represented as a linear regression (or later on, a logistic regression) model, which receives input, applies a transformation on that input, and produces an output.

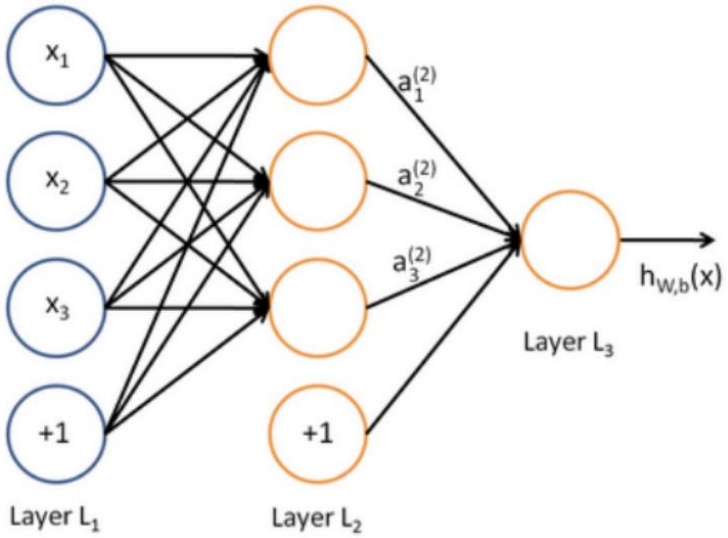
And similar to the way that biological neurons are connected and communicate with each other, artificial neurons are connected to each other through a network of edges, which can be thought of as pathways for information to flow through the network.

Later on, we will see more advanced ways to represent neurons than linear or logistic regression.

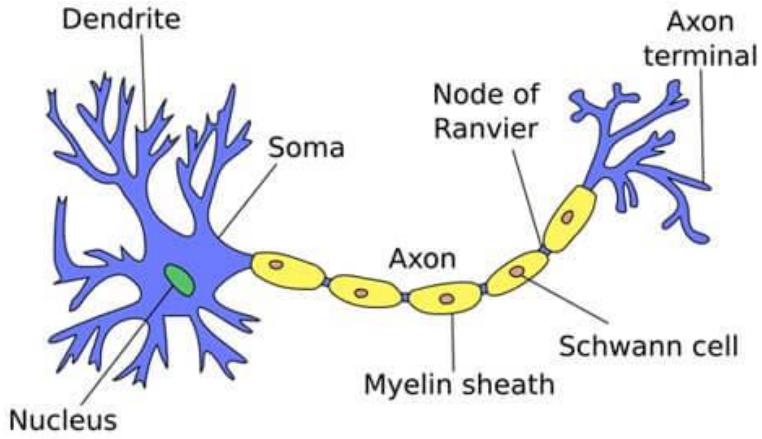
Coding a Neuron and a Neural Network



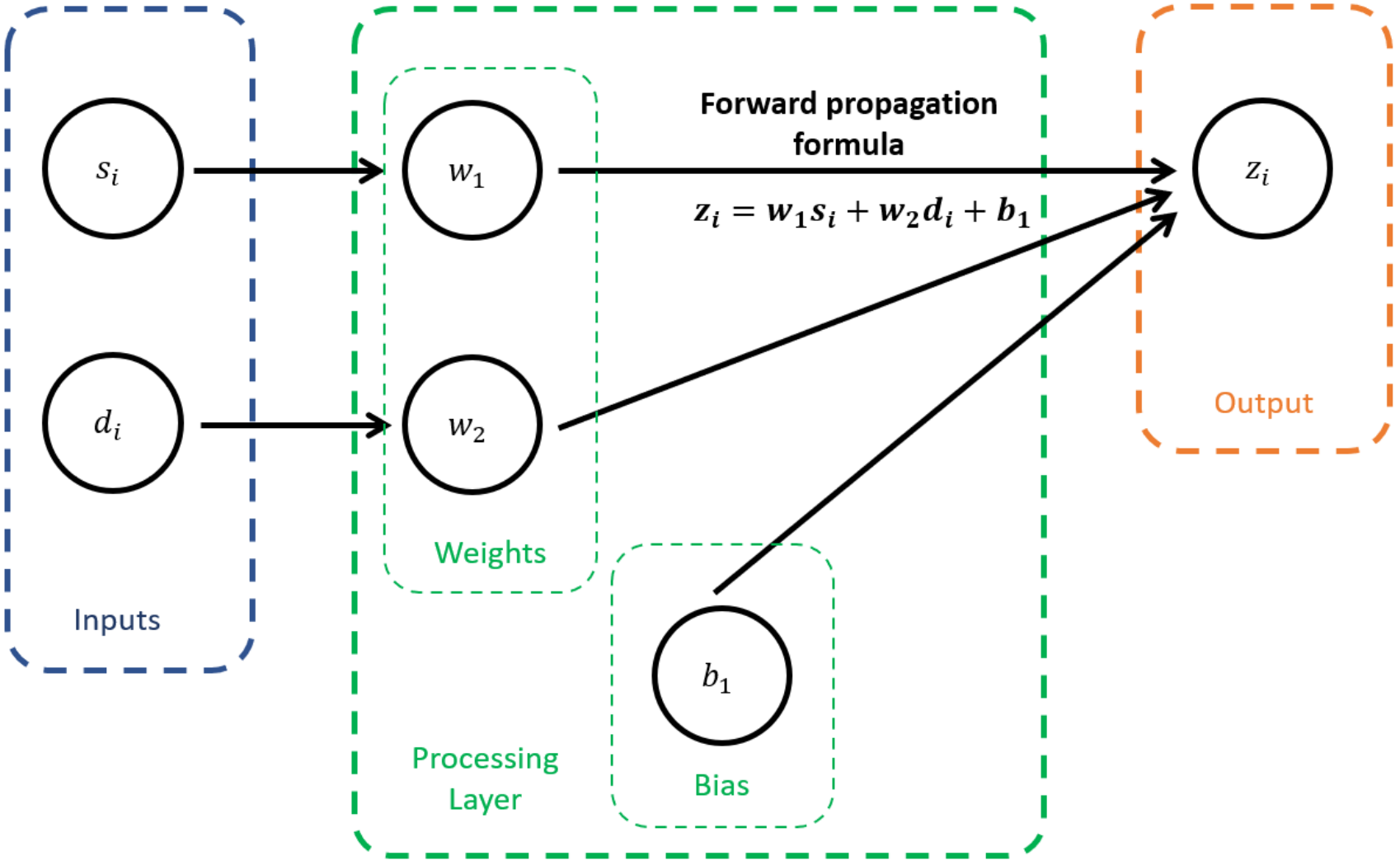
Logistic Regression



Neural Network



Coding a Neuron and a Neural Network



```python
# Let us define matrix W first, with two parameters w1 and w2.
# And the matrix b as well.
# Are these values randomly defined?
W = np.array([[14373/1000000], [-1286/1000000]]) # 14373/1000000 and -1286/1000000
b = np.ones(shape = (1, 1))*1.386 # 100000/1000000 + 1000*1286/1000000
```

```python
# We can then implement the multiplication operation
# we defined above by using the matmul operation.
Z = np.matmul(inputs, W)
pred = Z + b
# We can then check the shape of the matrices we obtained
print(inputs.shape)
print(W.shape)
print(b.shape)
print(Z.shape)
print(pred.shape)
print(outputs.shape)
```

```
(100, 2)
(2, 1)
(1, 1)
(100, 1)
(100, 1)
(100, 1)
```

Defining a minimal Neural Network

Definition (defining a minimal Neural Network):

In order to define a minimal neural network, we need two things:

- First, an **__init__ method,** listing the **trainable parameters** for the model.
- In our case that is a **weight vector** $W = (w_1, w_2)$ with 2 elements and a single scalar **bias** value b.
- Second, a **forward method**, which will be used to formulate predictions for any set of given inputs.
- The **prediction formula** is simply $y_i = w_1 s_i + w_2 d_i + b$.
- **(For now, this is equivalent to Linear Regression!)**

```python
class SimpleNeuralNet():

    def __init__(self, W, b):
        # Weights and biases matrices
        self.W = W
        self.b = b

    def forward(self, x):
        # Wx + b operation as above
        Z = np.matmul(x, self.W)
        pred = Z + self.b
        return pred
```

```python
# Create a minimal neuron neural network using our class
simple_neural_net = SimpleNeuralNet(W = np.array([[14373/1000000], [-1286/1000000]]), \
                                    b = np.ones(shape = (1, 1))*1.386)
print(simple_neural_net.__dict__)
```

```
{'W': array([[ 0.014373],
       [-0.001286]]), 'b': array([[1.386]])}
```

```python
# Predict and show first five samples
pred = simple_neural_net.forward(inputs)
print(pred[:5])
```

```
[[1.48509426]
 [3.16055536]
 [3.18794636]
 [2.65746327]
 [2.53638594]]
```

Adding a loss

- This takes care of the **first three elements** of a machine learning model (task, dataset, model).
- The **fourth** element we need is a **loss function**.
- As this is a regression class again, we can reuse the **MSE loss** and add this loss function as a method to our class.
- We will also reuse the **forward()** method we just coded to make predictions and then compared said predictions with the expected outputs from our dataset.

Adding a loss

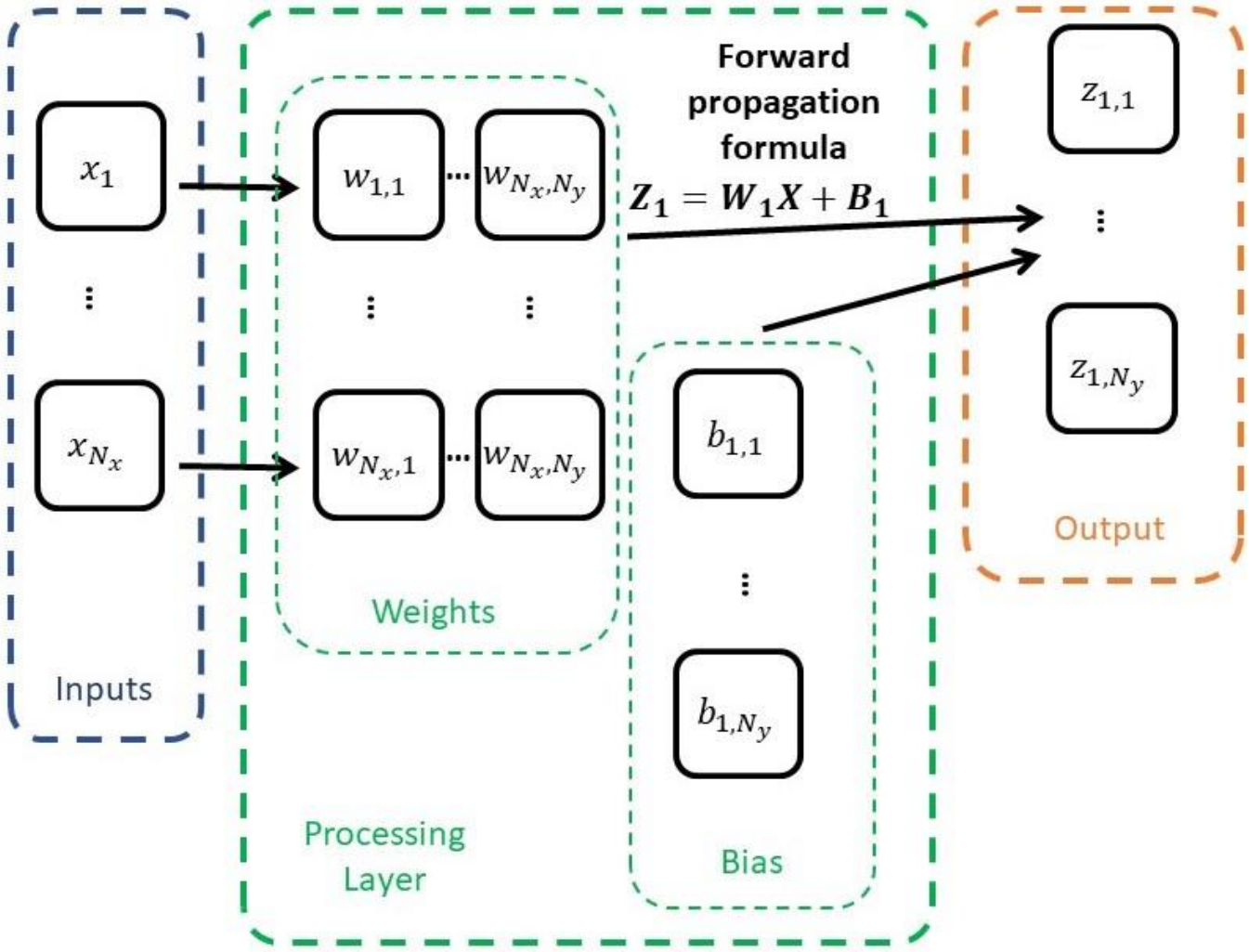
```python
class SimpleNeuralNet():

    def __init__(self, W, b):
        # Weights and biases matrices
        self.W = W
        self.b = b
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def forward(self, inputs):
        Z = np.matmul(inputs, self.W)
        pred = Z + self.b
        return pred

    def MSE_loss(self, inputs, outputs):
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        losses = (pred - outputs_re)**2
        self.loss = np.sum(losses)/outputs.shape[0]
        return self.loss
```

Scaling up with more parameters

Technically, our Neural Network could operate with any number of inputs N_x and any number of outputs N_y.

We would simply need to adjust the sizes of the parameters, making:

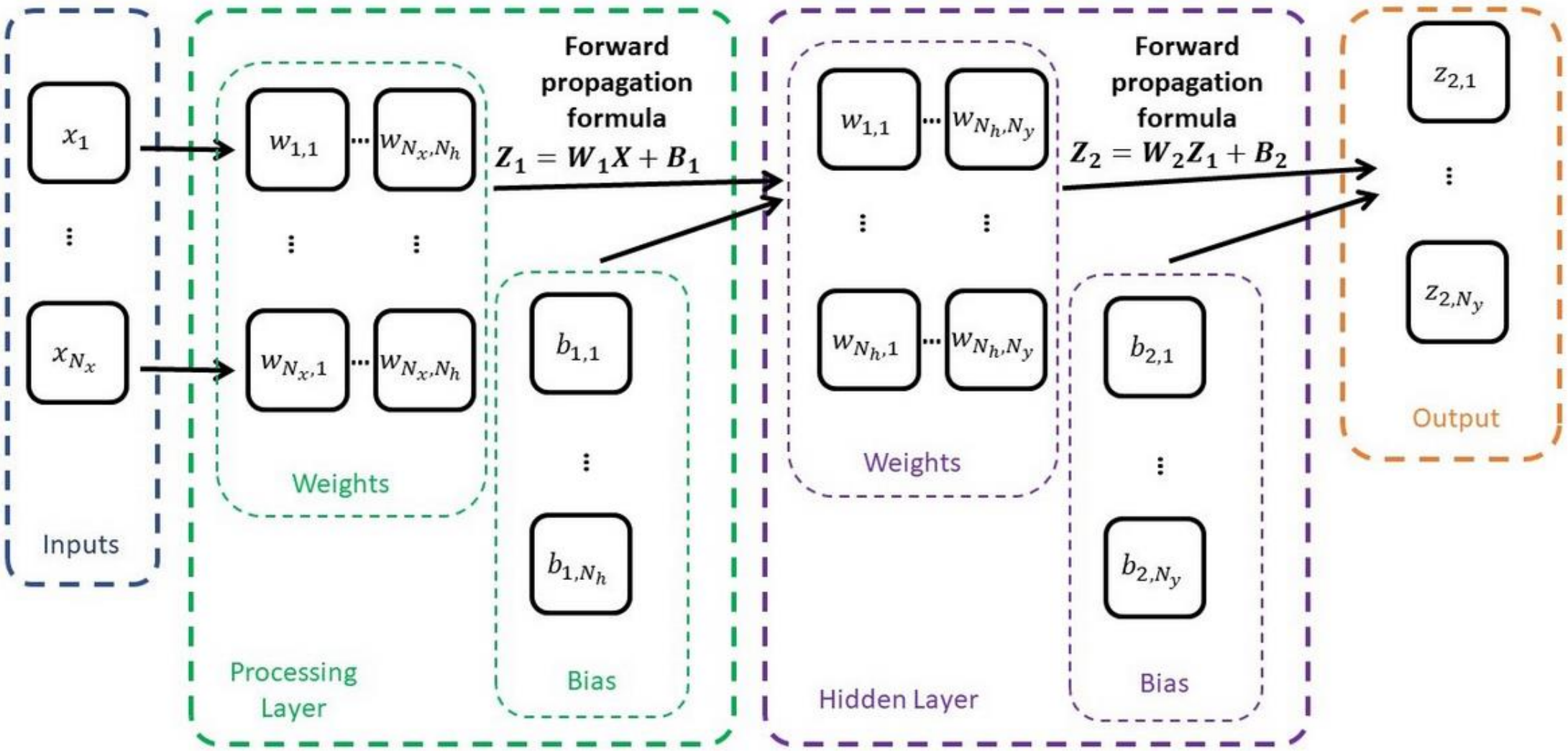
- W as a $N_x \times N_y$ matrix,
- B as a N_y vector.



Scaling up with more layers

- Following the logic of our SimpleNeuralNet, we could now implement a Shallow Neural Network, with its own class ShallowNeuralNet.
- It will simply include two processing layers instead of just one, both implementing a $WX + b$ operation of some sort.
- The first layer, will produce receive inputs with dimensionality n_x and produce outputs with dimensionality n_h. The matrix W for this layer will therefore be of size $n_x \times n_h$, and the matrix b will be a simple 1D vector with size n_h.
- The second layer, also called **hidden layer**, will produce receive inputs from the previous layer with dimensionality n_h and will produce outputs matching the dimensionality of the outputs in our dataset, that is n_y. The matrix W for this layer will therefore be of size $n_h \times n_y$, and the matrix b will be a simple 1D vector with size n_y.

Scaling up with more layers



Our NN class

Changes:

- Weights and biases are now **randomly generated**, instead of being passed to the init method.
- We pass the **expected sizes** n_x, n_h, n_y instead.
- **Initializing as normal random with zero mean and variance 1.**

```python
class ShallowNeuralNet():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Weights and biases matrices
        self.W1 = np.random.randn(n_x, n_h)*0.1
        self.b1 = np.random.randn(1, n_h)*0.1
        self.W2 = np.random.randn(n_h, n_y)*0.1
        self.b2 = np.random.randn(1, n_y)*0.1
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        # Wx + b operation for the second layer
        Z2 = np.matmul(Z1_b, self.W2)
        Z2_b = Z2 + self.b2
        return Z2_b

    def MSE_loss(self, inputs, outputs):
        # MSE loss function as before
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        losses = (pred - outputs_re)**2
        self.loss = np.sum(losses)/outputs.shape[0]
        return self.loss
```

```python
# Define neural network structure
n_x = 2
n_h = 4
n_y = 1
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
print(shallow_neural_net.__dict__)
```

```
{'n_x': 2, 'n_h': 4, 'n_y': 1, 'W1': array([[-0.10476816,  0.18570216,  0.03204007, -0.10951262],
       [-0.13867874, -0.03539496, -0.02856421,  0.20592501]]), 'b1': array([[ 0.0232776 , -0.16122469,  0.00718537,  0.0666335
1]]), 'W2': array([[ 0.03321156],
       [-0.0336505 ],
       [ 0.04977554],
       [-0.1794089 ]]), 'b2': array([[0.03460341]]), 'loss': inf}
```

1. Initialize model
2. Forward to predict
3. Compute loss to evaluate model (it is bad, because trainable parameters have received random values!)

```python
pred = shallow_neural_net.forward(inputs)
print(pred.shape)
print(outputs.shape)
print(pred[0:5])
print(outputs[0:5])
```

```
(100, 1)
(100, 1)
[[-23.24055489]
 [-31.54945952]
 [-12.75105332]
 [ -2.49026451]
 [-32.3803654 ]]
[[1.581913]
 [3.450274]
 [2.978769]
 [2.808258]
 [2.556398]]
```

```python
loss = shallow_neural_net.MSE_loss(inputs, outputs)
print(loss)
```

```
677.625448852107
```

The need for a training procedure

- At the moment, we have coded a model that can initialize trainable parameters randomly, formulate predictions and evaluate its own performance.
- It is great, but we have no way to compute the weights manually.
- We can only try a few different initialization and pray the RNG gods to give us trainable parameters with a good loss.
- **We need a training procedure!**

```
np.random.seed(963)
shallow_neural_net1 = ShallowNeuralNet(n_x, n_h, n_y)
loss1 = shallow_neural_net1.MSE_loss(inputs, outputs)
shallow_neural_net2 = ShallowNeuralNet(n_x, n_h, n_y)
loss2 = shallow_neural_net2.MSE_loss(inputs, outputs)
shallow_neural_net3 = ShallowNeuralNet(n_x, n_h, n_y)
loss3 = shallow_neural_net3.MSE_loss(inputs, outputs)
shallow_neural_net4 = ShallowNeuralNet(n_x, n_h, n_y)
loss4 = shallow_neural_net4.MSE_loss(inputs, outputs)
print(loss1, loss2, loss3, loss4)
```

```
21.318364917457647 58.190236579106184 4.770288142049728 0.2093294704434788
```

The backpropagation mechanism

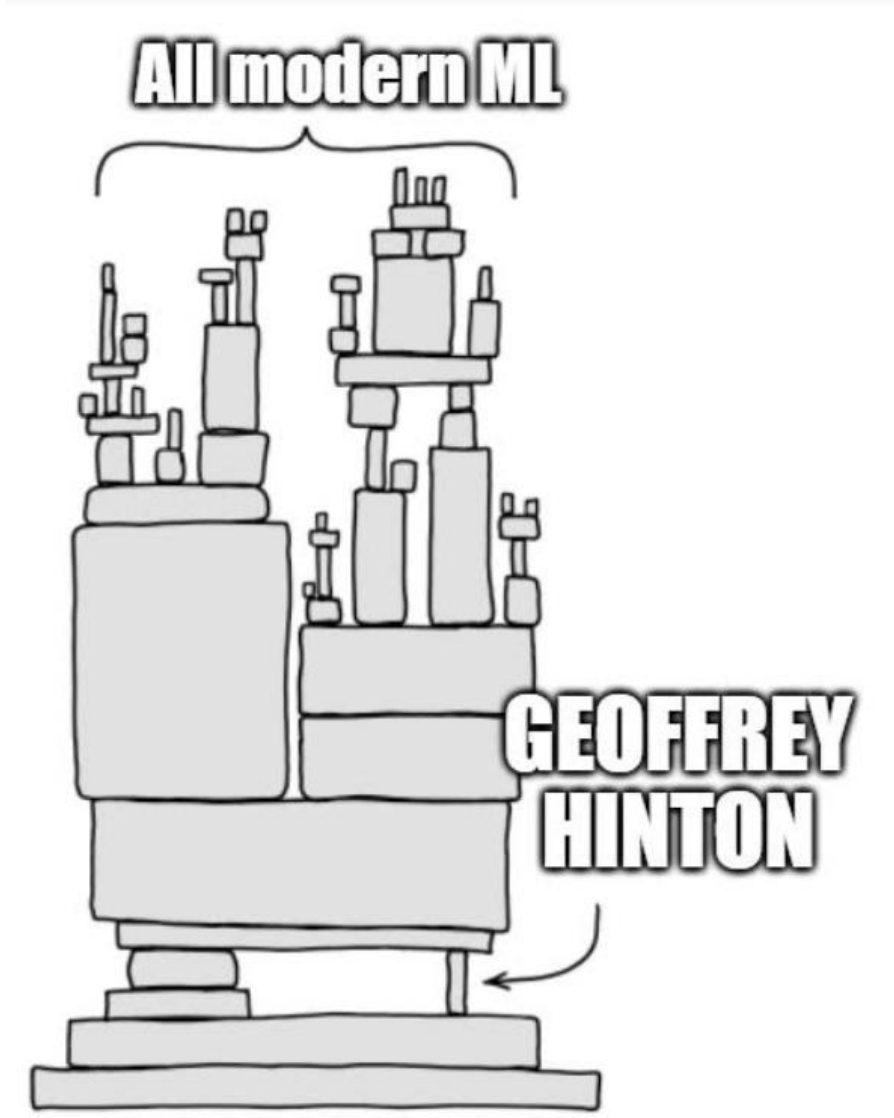
Definition (Backpropagation):

Backpropagation is an **algorithm** used to **train** neural networks.

It is a type of **gradient descent** algorithm that allows the network to learn by **adjusting the weights** of the connections between the neurons in the network.

Introduced by **Hinton** and **Rumelhart** [Rumelhart1986].

(Or was it? [Medium2020])



The backpropagation mechanism

Definition (Backpropagation):

Backpropagation is an **algorithm** used to **train** neural networks.

It is a type of **gradient descent** algorithm that allows the network to learn by **adjusting the weights** of the connections between the neurons in the network.

Introduced by **Hinton** and **Rumelhart** [Rumelhart1986].

(Or was it? [Medium2020])

Backpropagation will tell our model **how to adjust** the W and b matrices to improve its loss and prediction capabilities.

It will be implemented in the **backward()** method of our model.

Backpropagation in our model

While our Neural Network seems to be complicated in its architecture, at the end of the day, training it consists of solving this **optimization problem**.

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1, b_1, W_2, b_2} \left[\frac{1}{M} \left(Y_{pred}(X, W_1, b_1, W_2, b_2) - Y \right)^2 \right]$$

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1, b_1, W_2, b_2} \left[\frac{1}{M} (W_2(W_1 X + b_1) + b_2 - Y)^2 \right]$$

With M being the number of samples in the dataset. We will use **gradient descent**, like we did for the Linear Regression, except that this time we have 4 parameters and a loss function that is a bit more complex.

Before we continue

Before we continue, a quick note about square matrices.

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1,b_1,W_2,b_2} \left[\frac{1}{M}\left(Y_{pred}(X, W_1, b_1, W_2, b_2) - Y\right)^2\right]$$

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1,b_1,W_2,b_2} \left[\frac{1}{M}(W_2(W_1X + b_1) + b_2 - Y)^2\right]$$

The notation $(A)^2$ is somewhat abusive when A is a matrix, the correct way would be to write $A.A^T$.

But in the formula above, that would lead to a very large formula (too large for slides!)

Chain rule, to the rescue!

Let us denote the error term ϵ.

$$\epsilon = Y_{pred}(X, W_1, b_1, W_2, b_2) - Y = W_2(W_1 X + b_1) + b_2 - Y.$$

Using the chain rule, we can simply compute $\frac{\partial L}{\partial W_2}$ as

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial W_2}$$

Chain rule, to the rescue!

Since we have

$$\frac{\partial L}{\partial Y_{pred}} = \frac{2}{M} Y_{pred}(X, W_1, b_1, W_2, b_2) - Y = \frac{2\epsilon}{M}$$

And

$$\frac{\partial Y_{pred}}{\partial W_2} = W_1 X + b_1$$

The gradient descent update rule for W_2 is then

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)$$

Chain rule, to the rescue!

We have just computed the formula for the gradient descent update rule for W_2, which was deifned as

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)$$

Similarly, we can define the gradient descent update rule for b_2 as:

$$b_2 \leftarrow b_2 - \frac{2\epsilon\alpha}{M}$$

Chain rule, to the rescue!

Similarly, we have

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial W_1} = \left(\frac{2\epsilon}{M}\right)(W_2 X)$$

$$And\ \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial b_1} = \left(\frac{2\epsilon}{M}\right)(W_2)$$

And therefore, we have

$$W_1 \leftarrow W_1 - \frac{2\epsilon\alpha}{M} W_2 X$$

$$b_1 \leftarrow b_1 - \frac{2\epsilon\alpha}{M} W_2$$

Strongly encouraging to try and do the math again on paper on your own!

That’s the best way to learn and you can always refer to these slides for answers!

(Remember, you have no homework this week, so let us call that your homework!)

Backpropagation in our model

Update rules for W_2 and b_2

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)$$

$$b_2 \leftarrow b_2 - \frac{2\epsilon\alpha}{M}$$

```python
def backward(self, inputs, outputs, alpha = 1e-5):
    # Get the number of samples in dataset
    m = inputs.shape[0]

    # Forward propagate
    Z1 = np.matmul(inputs, self.W1)
    Z1_b = Z1 + self.b1
    Z2 = np.matmul(Z1_b, self.W2)
    y_pred = Z2 + self.b2

    # Compute error term
    epsilon = y_pred - outputs

    # Compute the gradient for W2 and b2
    dL_dW2 = (2/m)*np.matmul(Z1_b.T, epsilon)
    dL_db2 = (2/m)*np.sum(epsilon, axis = 0, keepdims = True)

    # Compute the loss derivative with respect to the first layer
    dL_dZ1 = np.matmul(epsilon, self.W2.T)
```

Backpropagation in our model

Update rules for W_1 and b_1

$$W_1 \leftarrow W_1 - \frac{2\epsilon\alpha}{M} W_2 X$$

$$b_1 \leftarrow b_1 - \frac{2\epsilon\alpha}{M} W_2$$

```
# Compute the loss derivative with respect to the first layer
dL_dZ1 = np.matmul(epsilon, self.W2.T)

# Compute the gradient for W1 and b1
dL_dW1 = (2/m)*np.matmul(inputs.T, dL_dZ1)
dL_db1 = (2/m)*np.sum(dL_dZ1, axis = 0, keepdims = True)

# Update the weights and biases using gradient descent
self.W1 -= alpha*dL_dW1
self.b1 -= alpha*dL_db1
self.W2 -= alpha*dL_dW2
self.b2 -= alpha*dL_db2

# Update loss
self.MSE_loss(inputs, outputs)
```

Backpropagation in our model

Effect of the backpropagation

- Every time we call the backward() method, our model will update its parameters.
- Eventually it will become better and better at the task in question.
- This can be seen by looking at the loss values decreasing after each iteration of the backward() method.

```
# Define neural network structure
n_x = 2
n_h = 4
n_y = 1
np.random.seed(967)
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
loss = shallow_neural_net.MSE_loss(inputs, outputs)
print(shallow_neural_net.loss)
```

```
13.1869714693353
```

```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
8.576174190387835
```

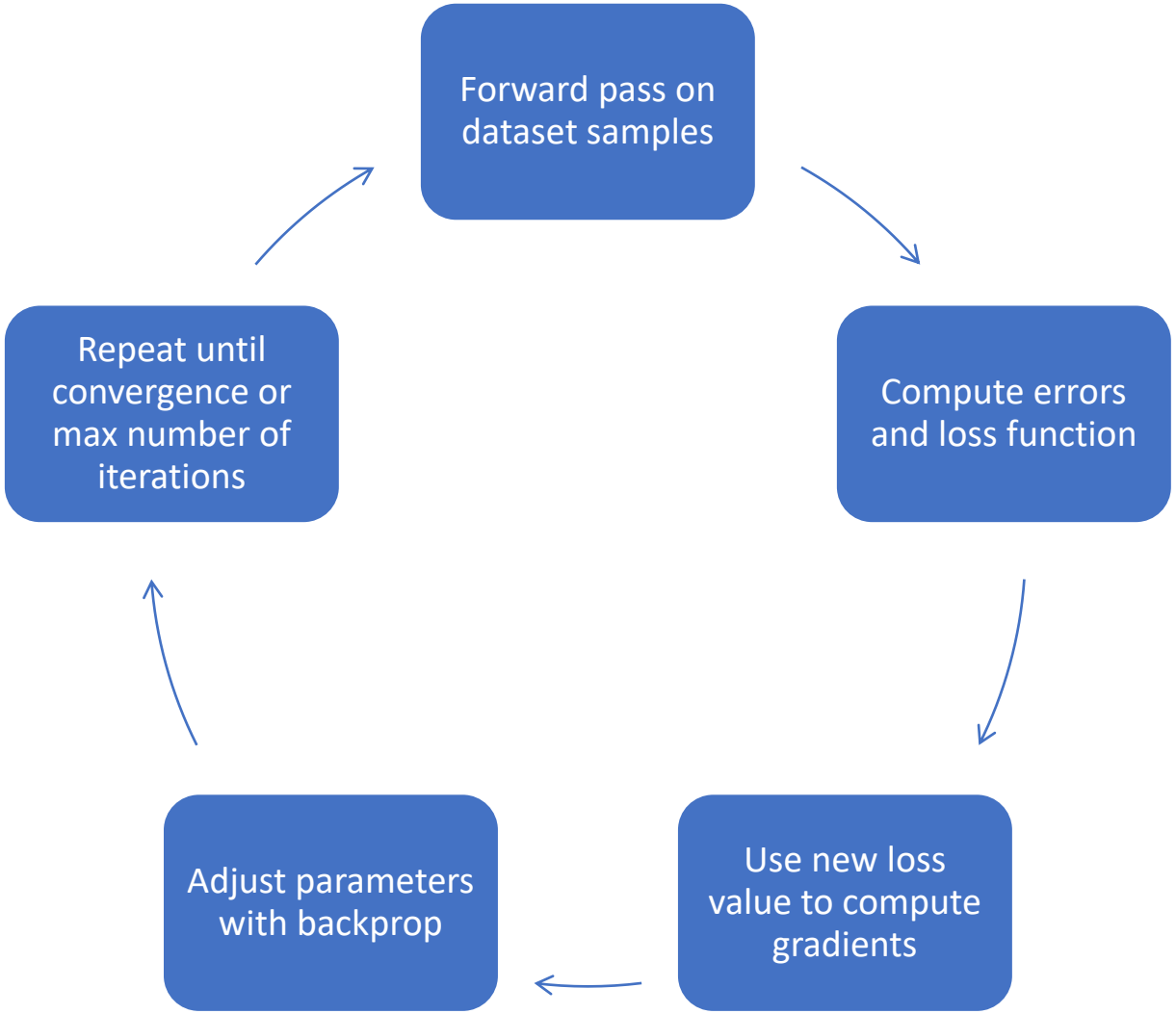
```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
5.818528741557871
```

```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
4.14790610131927
```

Training procedure, in short.



Trainer function for our model

Train function implementation

- Iterate the backward() method for a given number of iteration $N_{\max}$.
- Display new loss values and append them in a list for display later.
- Also implemented an early stopping, which will break for loop if no change in parameters during an iteration of backward().

```python
# Our trainer function
def train(shallow_neural_net, inputs, outputs, N_max = 1000, \
          alpha = 1e-5, delta = 1e-5, display = True):
    # List of losses, starts with the current loss
    losses_list = [shallow_neural_net.loss]
    # Repeat iterations
    for iteration_number in range(1, N_max + 1):
        # Backpropagate
        shallow_neural_net.backward(inputs, outputs, alpha)
        new_loss = shallow_neural_net.loss
        # Update losses list
        losses_list.append(new_loss)
        # Display
        if(display):
            print("Iteration {} - Loss = {}".format(iteration_number, new_loss))
        # Check for delta value and early stop criterion
        difference = abs(losses_list[-1] - losses_list[-2])
        if(difference < delta):
            if(display):
                print("Stopping early - loss evolution was less than delta.")
            break
    else:
        # Else on for loop will execute if break did not trigger
        if(display):
            print("Stopping - Maximal number of iterations reached.")
    return losses_list
```

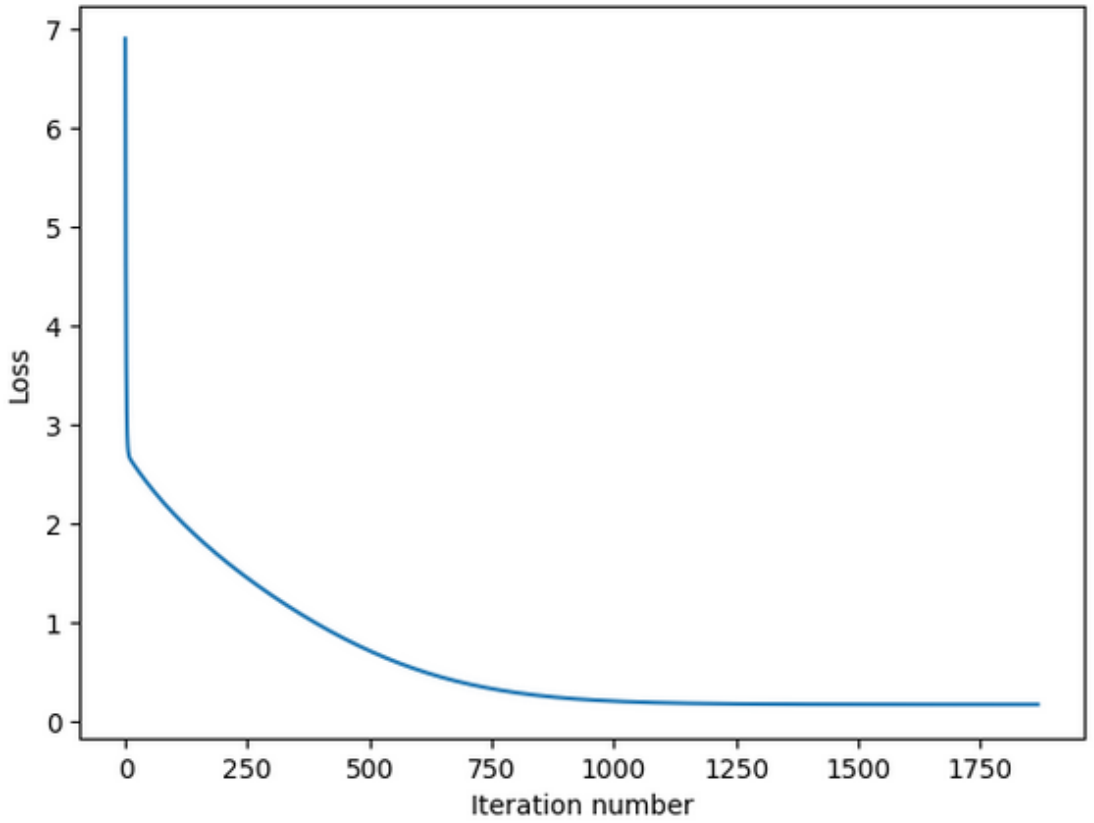
Trainer function for our model

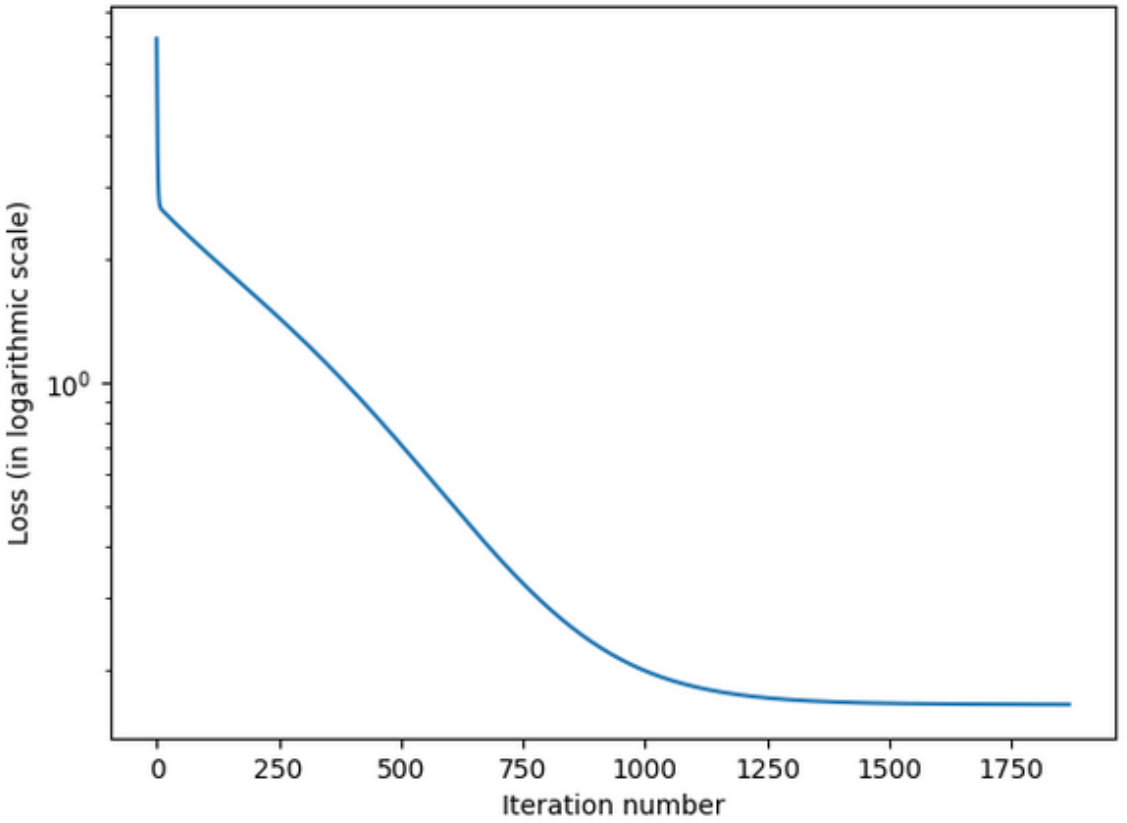
```
1 losses_list = train(shallow_neural_net, inputs, outputs, N_max = 10000, alpha = 1e-5, delta = 1e-6, display = True)
```

```
Iteration 1 - Loss = 4.639845655363769
Iteration 2 - Loss = 3.6055739235388646
Iteration 3 - Loss = 3.1219544752264055
Iteration 4 - Loss = 2.891370725402625
Iteration 5 - Loss = 2.778663798354082
Iteration 6 - Loss = 2.721333641596964
Iteration 7 - Loss = 2.6901646643755135
Iteration 8 - Loss = 2.671407441095785
Iteration 9 - Loss = 2.6585626629544303
Iteration 10 - Loss = 2.648548914291606
Iteration 11 - Loss = 2.639902127093024
Iteration 12 - Loss = 2.6319255476288577
Iteration 13 - Loss = 2.6242871305644138
Iteration 14 - Loss = 2.6168284145839156
Iteration 15 - Loss = 2.60947364178999
Iteration 16 - Loss = 2.6021864900028118
Iteration 17 - Loss = 2.5949494536811937
Iteration 18 - Loss = 2.58775401064343
Iteration 19 - Loss = 2.5805959297914653
```

Trainer function for our model

The curves below are called **performance/training curves** and show how the training went.





Finally, add the trainer function and the performance curves function as methods to our class.

```python
    def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, delta = 1e-5, display = True):
        # List of losses, starts with the current loss
        self.losses_list = [self.loss]
        # Repeat iterations
        for iteration_number in range(1, N_max + 1):
            # Backpropagate
            self.backward(inputs, outputs, alpha)
            new_loss = self.loss
            # Update losses list
            self.losses_list.append(new_loss)
            # Display
            if(display):
                print("Iteration {} - Loss = {}".format(iteration_number, new_loss))
            # Check for delta value and early stop criterion
            difference = abs(self.losses_list[-1] - self.losses_list[-2])
            if(difference < delta):
                if(display):
                    print("Stopping early - loss evolution was less than delta.")
                break
        else:
            # Else on for loop will execute if break did not trigger
            if(display):
                print("Stopping - Maximal number of iterations reached.")

    def show_losses_over_training(self):
        # Initialize matplotlib
        fig, axs = plt.subplots(1, 2, figsize = (15, 5))
        axs[0].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[0].set_xlabel("Iteration number")
        axs[0].set_ylabel("Loss")
        axs[1].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[1].set_xlabel("Iteration number")
        axs[1].set_ylabel("Loss (in logarithmic scale)")
        axs[1].set_yscale("log")
        # Display
        plt.show()
```

Conclusion (Week 1)

- Reminders of Machine Learning
- Linear
- Using a normal equation to train a linear regression
- Gradient descent as a training procedure for linear regression
- Polynomial Regression
- Regularization in Ridge/Lasso Regression
- Train-test split
- Overfitting and underfitting
- Generalization
- Sigmoid and Logistic functions
- From linear regression to logistic regression
- Using logistic regression for binary classification

Conclusion (Week 1)

- Implementing a shallow neural network in Numpy
- Forward propagation method, to formulate predictions
- The backpropagation mechanism, as the gradient descent on Neural Network
- Backward method for training and trainer functions to iterate backward iterations
- Performance/training curves

Next week?

- Initializations to break symmetries
- Exploding and vanishing gradients
- Activation functions and non-linearities in Neural Networks
- Advanced optimizers
- Validation sets, early stopping, saver and loader functions

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Géron2019] A. Géron , “How Neural Networks Work”, 2019.
- [Yamins2016] Yamins et al., “Deep neural networks are robust computational models of the human visual system”, 2016.
- [Kriegeskorte2013] Kriegeskorte and Kievit, “Neural Network Models of the Human Brain”, 2013.
- [Rumelhart1986] D.E. **Rumelhart**, G. **Hinton**, Williams, “Learning representations by back-propagating errors“, 1986 https://www.nature.com/articles/323533a0

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Geoffrey Hinton: Professor** at **University of Toronto**, **one of the three Godfathers of Deep Learning** and **2018 Turing Award** winner (highest distinction in Computer Science).
  https://www.cs.toronto.edu/~hinton/
  https://scholar.google.co.uk/citations?user=JicYPdAAAAAJ&hl=en
- **David Rumelhart: Former professor** at **University of California**, credited for inventing **backpropagation along** with **Hinton**. Passed away in 2011.
  https://www.nytimes.com/2011/03/19/health/19rumelhart.html

Twitter, the theater for AI/DL drama and announcements

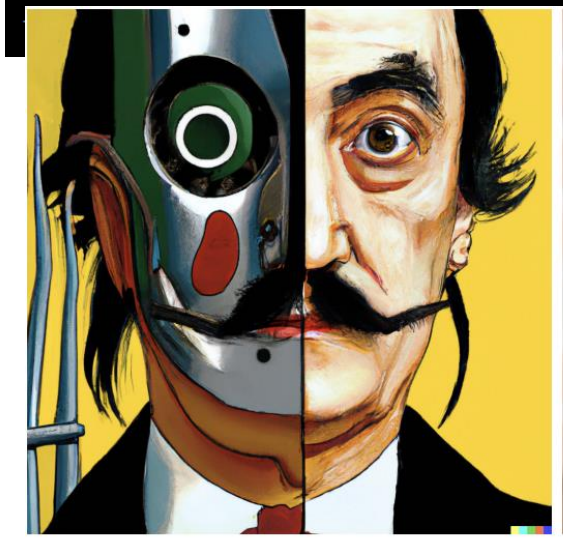
OpenAI ✓
@OpenAI

Here's a look at what DALL·E 2 can do. 👀🧵👇

Want to see more? Follow along on Instagram:
instagram.com/openaidalle/

Traduire le Tweet

Announcing Dall-E 2 and its future release



vibrant portrait painting of Salvador Dalí with a robotic half face

a shiba inu wearing a beret and black turtleneck

Gary Marcus @GaryMarcus · 8 mai
You obviously never read a single one of the papers of mine that you have been criticizing.

One thing for you to disagree, another to show a lack of awareness of what the person you disagree with is saying.

2 3 15

NYU Professor

Yann LeCun
@ylecun

En réponse à @GaryMarcus et @Tesla

You obviously never engineered nor built a working AI system.
Arguments in favor "hybrid systems" are non-falsifiable strawmen, whose only purpose to allow the author to declare victory regardless of the outcome.
As I said, this debate is vacuous.

Traduire le Tweet

12:18 AM · 9 mai 2022 · Twitter Web App

2 Retweets **9** Tweets cités **173** J'aime

NYU Professor and Chief of AI at Meta (Facebook)

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [Quanta2021] “Artificial Neural Nets Finally Yield Clues to How Brains Learn“, 2021. https://www.quantamagazine.org/artificial-neural-nets-finally-yield-clues-to-how-brains-learn-20210218/
- [MITNews2022] “Study urges caution when comparing neural networks to the brain”, 2022. https://news.mit.edu/2022/neural-networks-brain-function-1102
- [Medium2020] “Who Invented Backpropagation? Hinton Says He Didn’t, but His Work Made It Popular“, 2020. https://medium.com/syncedreview/who-invented-backpropagation-hinton-says-he-didnt-but-his-work-made-it-popular-e0854504d6d1/