Practice of Deep Learning

Day 1, Part 3/4

Matthieu De Mari

About this lecture

1. What is a **sigmoid** function? What is a **logistic** function?
2. How to perform **binary classification** using a **logistic regressor** and how is it related to linear regression?
3. What are **Neural Networks** and how do they relate to the **biology of a human brain**?
4. What is a **Neuron** in a Neural Network and how does it relate to linear/logistic regression?
5. What is the **difference** between a **shallow** and a **deep neural network**?
6. How to **implement a shallow Neural Network** manually and define a **forward propagation** method for it?

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 $s(x)$.

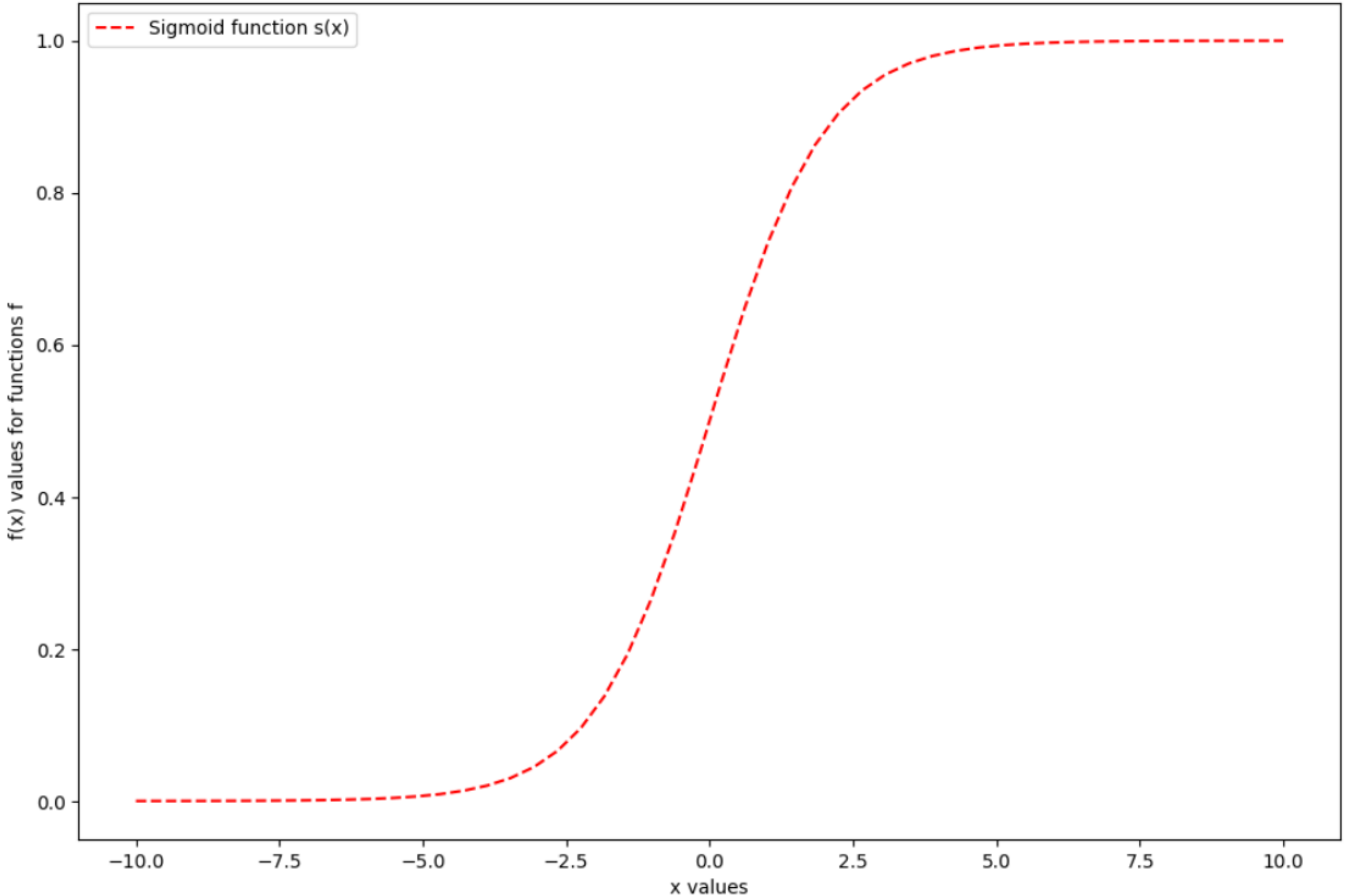
The sigmoid function has the following properties (to be proven as an extra exercise!):

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

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

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

$$s(0) = 0.5$$


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

Also 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, shown on the right.
(Prove them as an extra practice?)

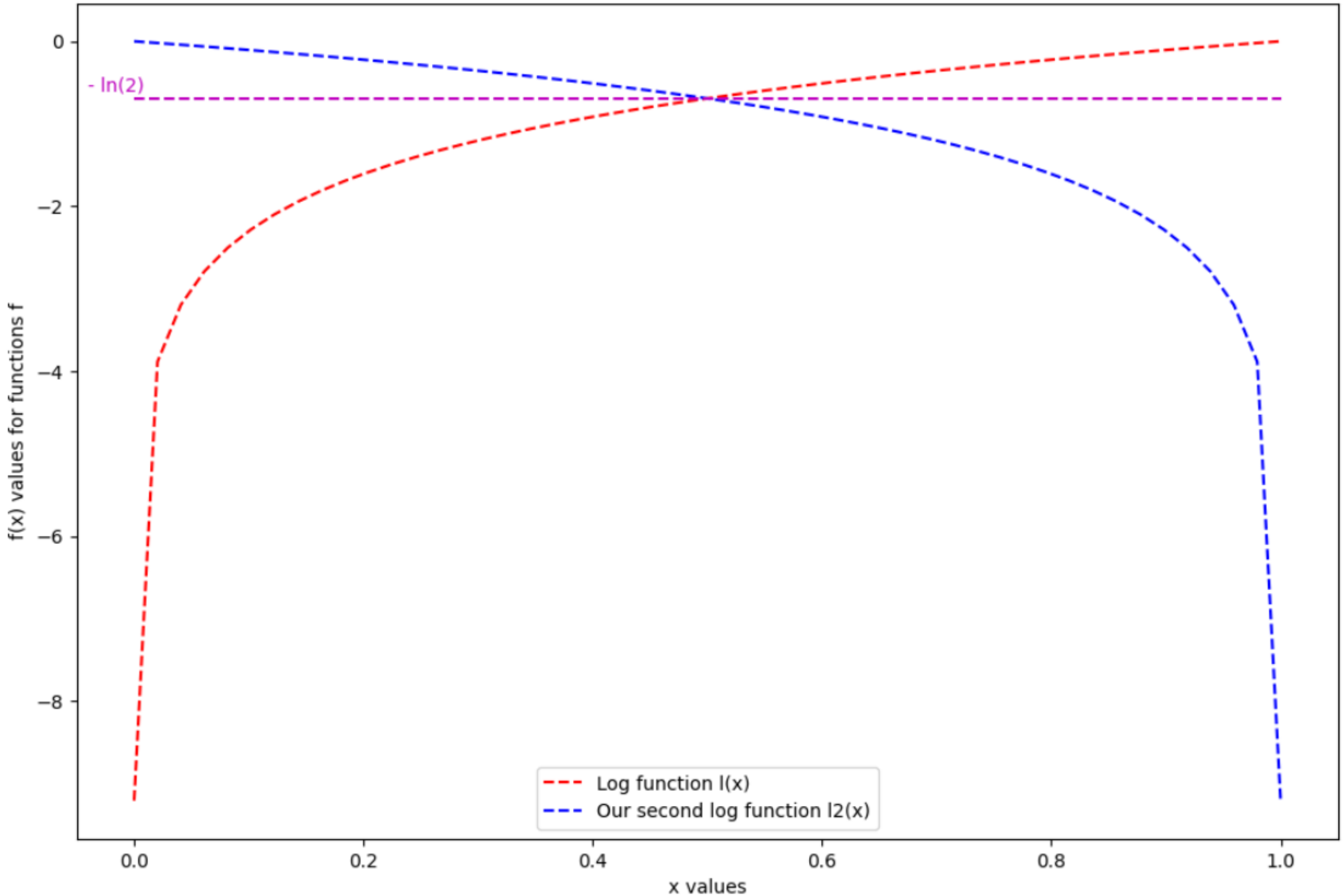
$$\lim_{x \to 0} l(x) = -\infty$$

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

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

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

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


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)

To recap: functions and implementations

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)
```

Results on mixes for our two logistic functions

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

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

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

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

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

Results on mixes for our two logistic functions

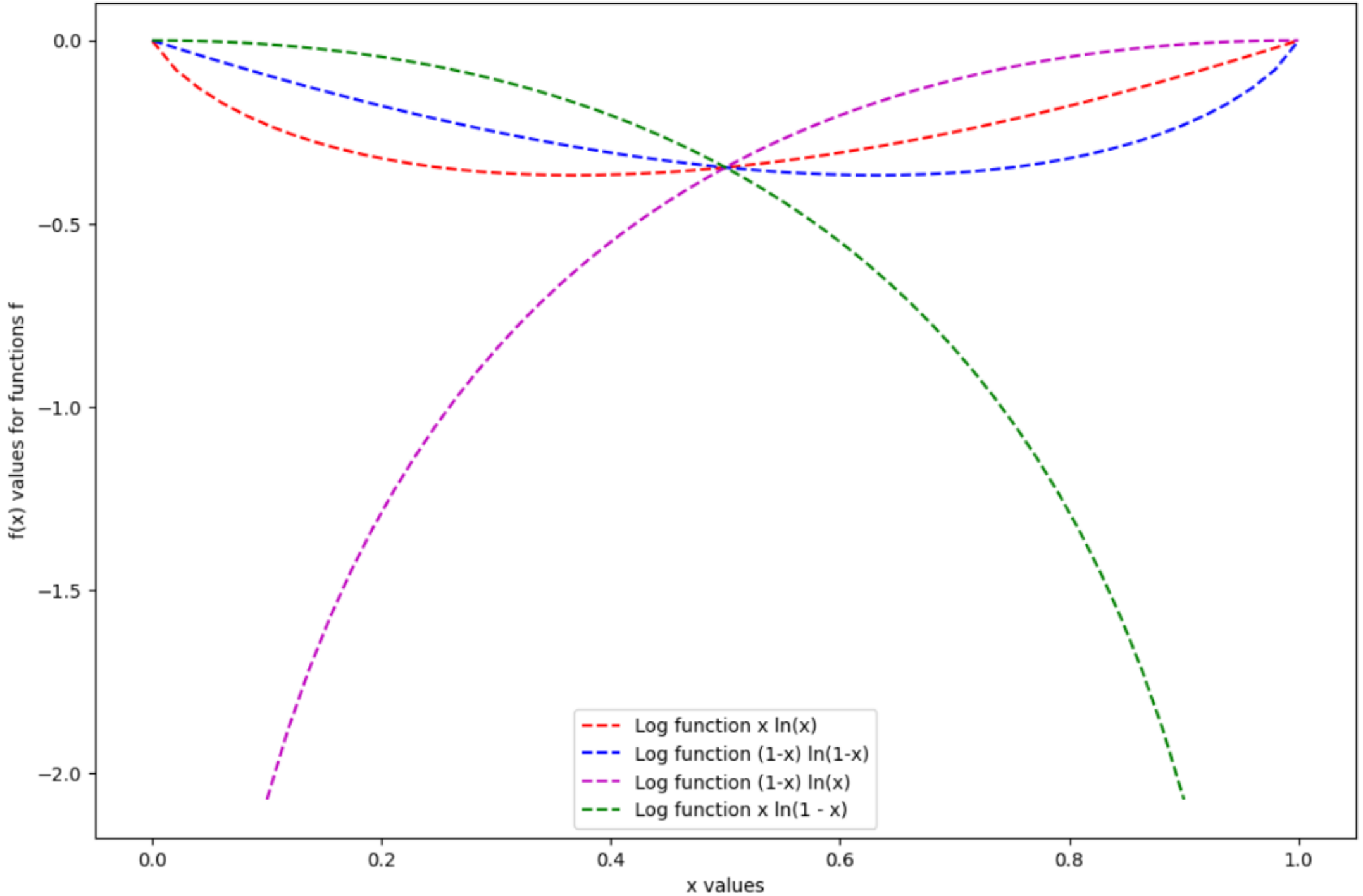
In addition, the following properties hold (some of them require using L'Hospital rule to prove them, try it out?).

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

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

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

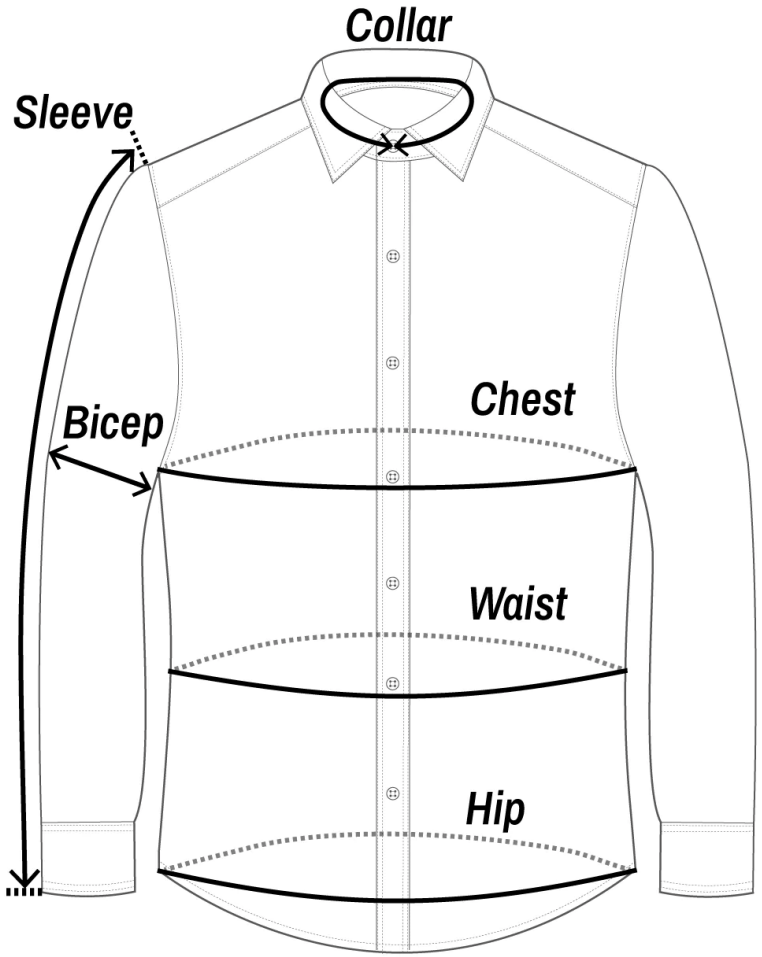
$$\lim_{x \to 1} x\, ln(1 - x) = -\infty$$


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)

New mock dataset generation

In our first notebook, we have a dataset for a **single-variable binary classification.**

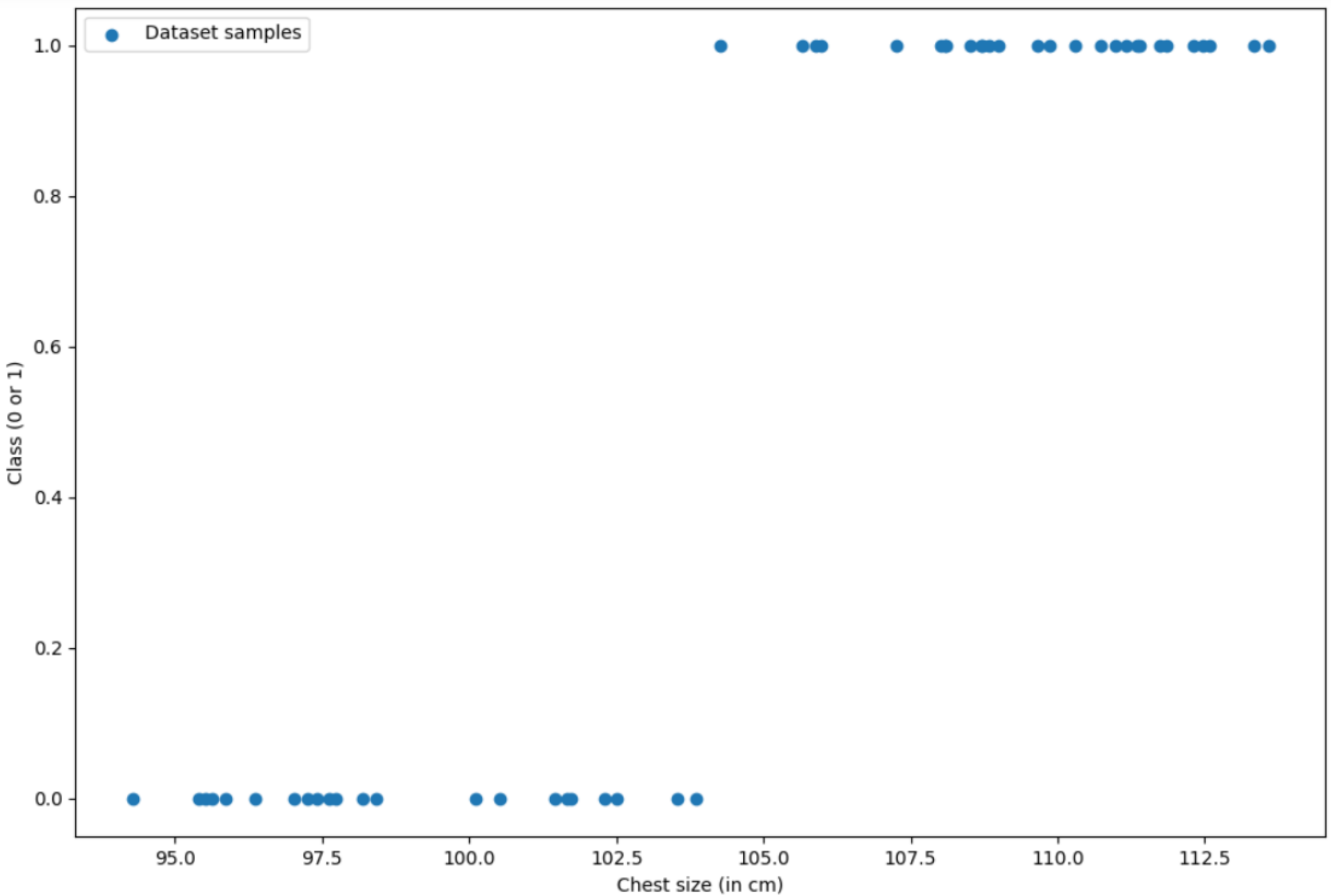
- The **inputs**, x, will consist of chest sizes for males between 94cm and 114cm, randomly generated.
- The **outputs**, y, will consists of **two classes**, corresponding to the shirt size to use, with two possible values:
  - **M** (class with value **0**, if chest size is below **104** cm) size
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

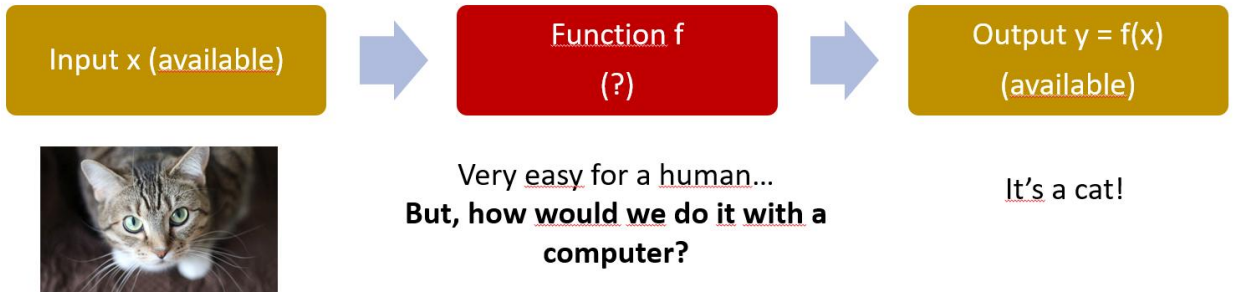
A quick word about expert systems

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 systems are "the dream" and provide a high level of performance and accuracy in solving problems or making decisions. But these are difficult to design as they require a strong, and not necessarily obvious, 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 an expert system as shown below, but most of the time the datasets will not have a clear logic and you will have to rely on ML algorithms instead, basically *"hoping the AI will 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 the probability of x being a L size shirt).

Respectively, $1 - p(x)$ is 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.

An adjusted 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.

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

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

This model has **two trainable parameters**, $\boldsymbol{a}$ and $\boldsymbol{b}$, to be decided as 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.

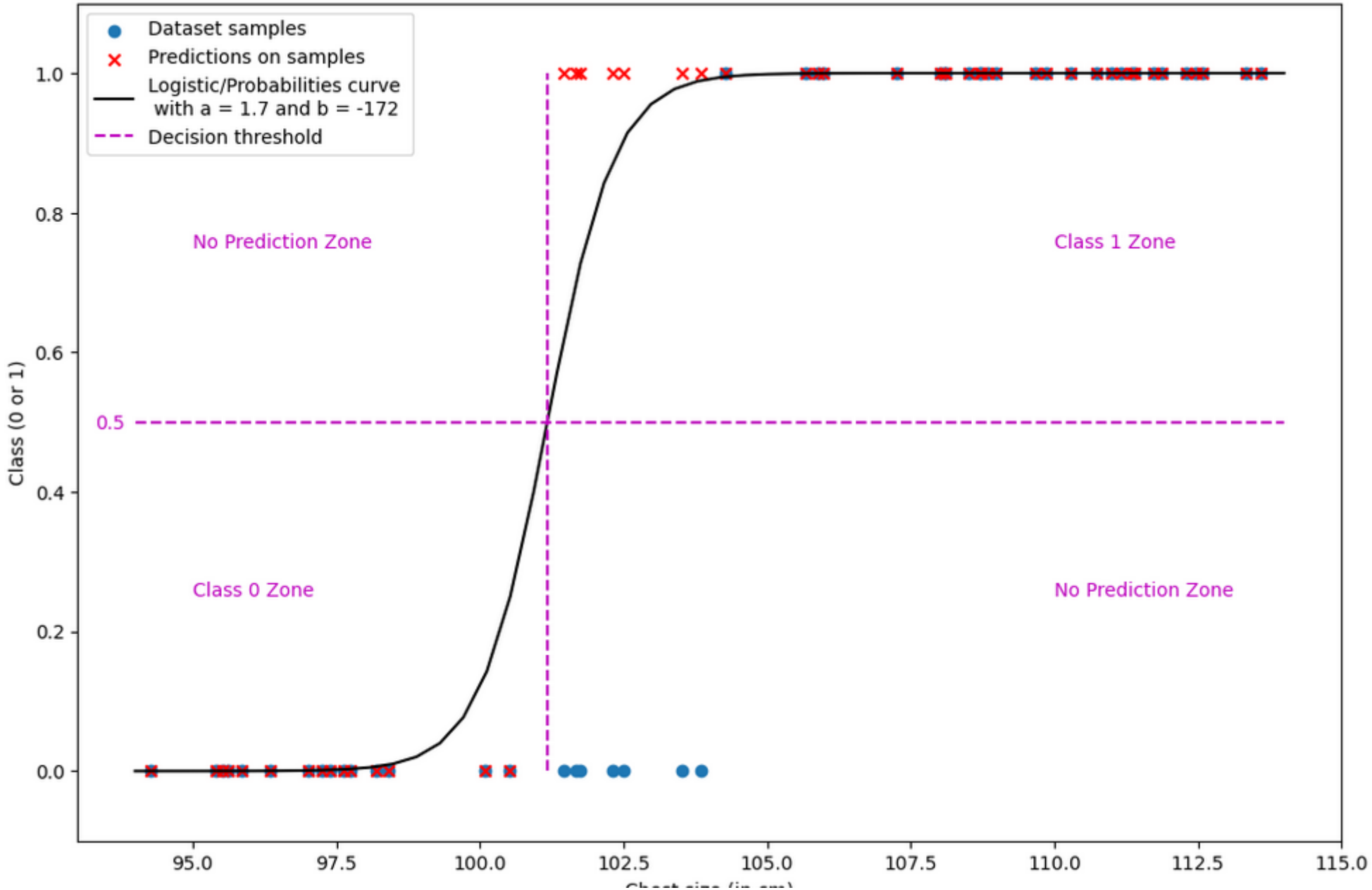
From Linear Regression to Logistic Regression

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 is not that good for logistic regression.
- We need something better!

In logistic regression, we prefer to **minimize** a loss function, called the **log-likelihood (cross-entropy)** 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 mixes of 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 large non-zero negative values instead**.

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/loss metric to measure the performance of our models.

Indeed, **maximizing this log-likelihood** loss function and bringing it to zero 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 log-likelihood loss function by -1**.

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

Implementing the log-likelihood loss

We can simply implement the log likelihood loss as shown below.

- Note: We could look for the **”normal equation”** for the logistic regression, but this simply will not work. It is simply impossible to solve the logistic regression problem analytically.
- (Not convinced? Try solving it analytically!).

```
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 the gradient descent method, or the Newton method.

- Typically, this is what the **LogisticRegression()** object in sklearn uses for training!
- (Newton method is not covered but feel free to look it up/try it out?)

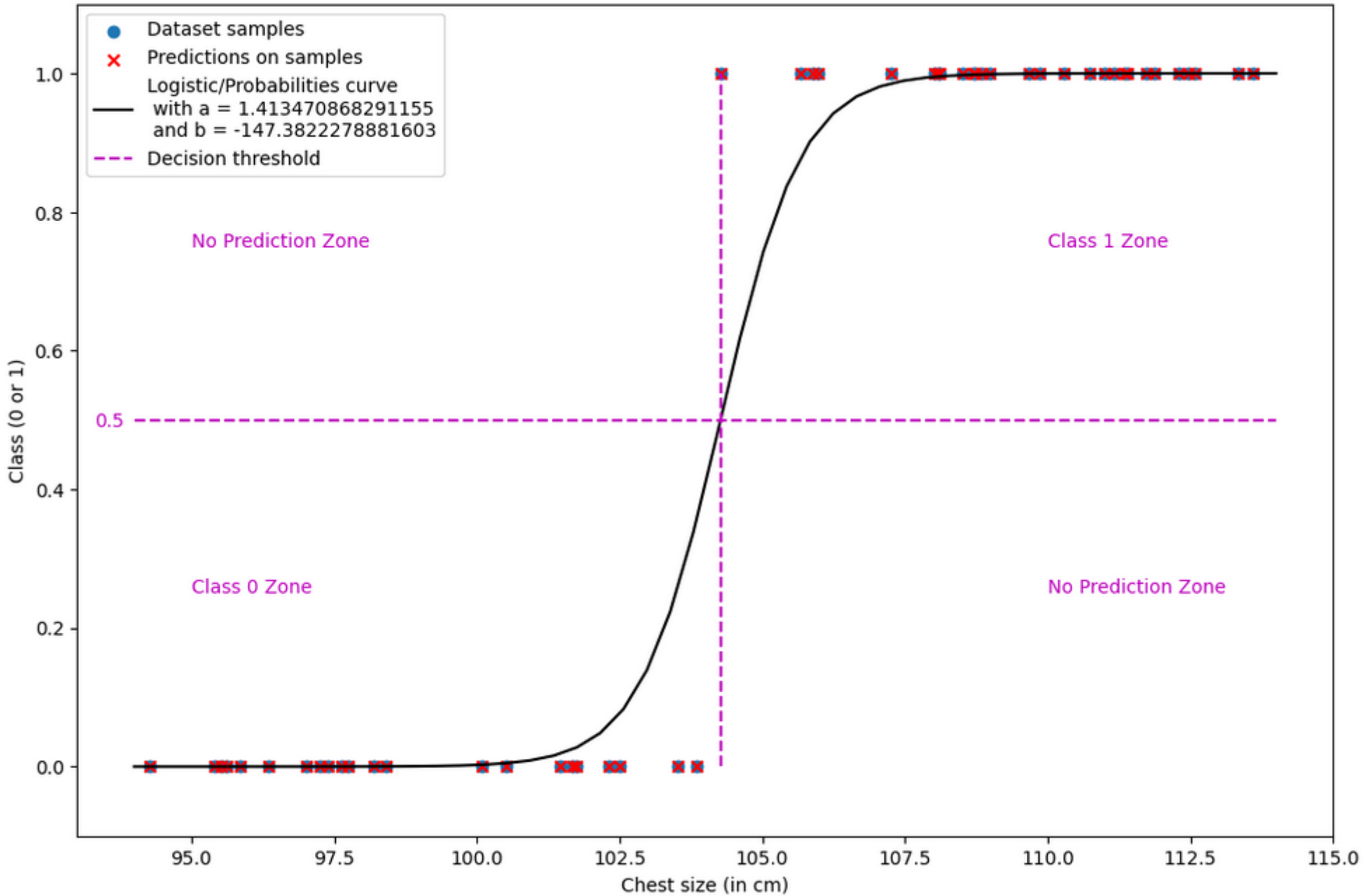
```
# Reshaping inputs as a 2D array
# (As before, this is a requirement for sklearn models)
inputs_re = inputs.reshape(-1, 1)

# 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 (test samples, preferably).

- 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 trained 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 interesting metrics, the best value is 1.0. Our model achieves a 1.0 score, suggesting 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
```

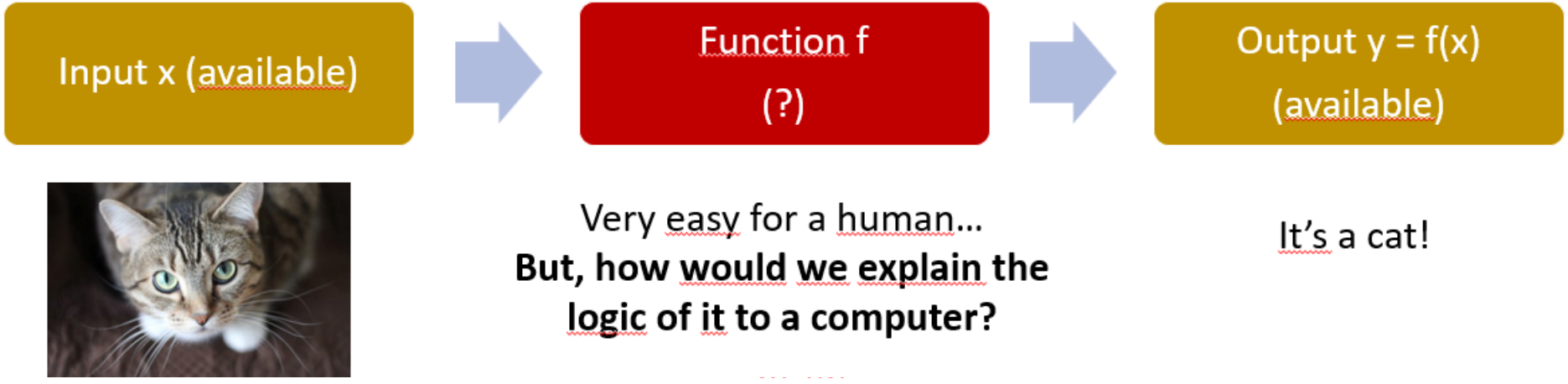
Note: the details of these metrics and what they represented is not discussed here (covered in previous ML classes).

A bit of biology

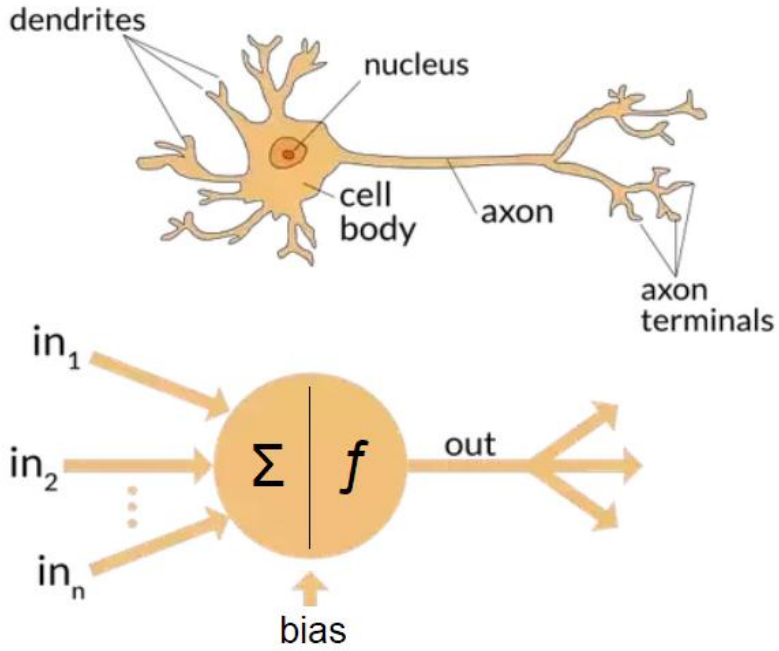
Many tasks we have seen earlier (e.g. appt prices) are “difficult” to solve, or do not have an obvious “explainable” logic.

Our brain, however, seems very capable to solve these tasks.

Million dollar question: Could we replace the linear/polynomial/logistic models with a model that mimics the brain?



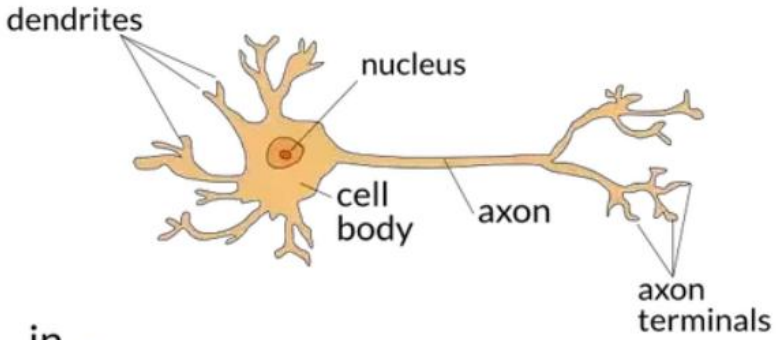
A bit of biology

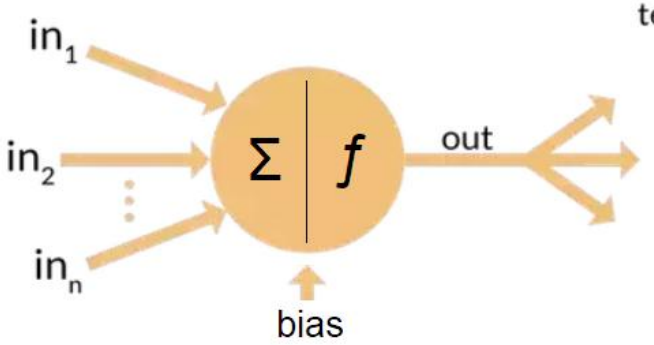


Question: In neural networks, we will implement neurons, but are they really close to the ones in the human brain?

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

A bit of biology





In general,

- A **neuron will receive stimuli** through its dendrites, in the form of electrical signals playing the role of **inputs**,
- It will **process the inputs** in a certain way in its nucleus (and other parts of the neuron),
- If the integrated signals reach a threshold, **the neuron generates an action potential**, a brief electrical impulse. This is an output coming from the axon terminal, in the form of a signal (1) or no signal (0) being produced.

In other words,

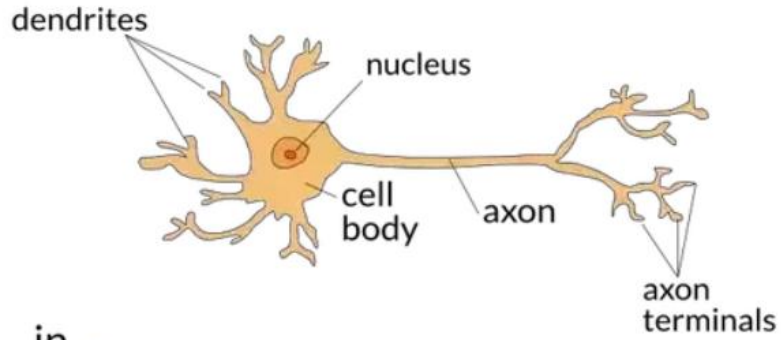
Neuron

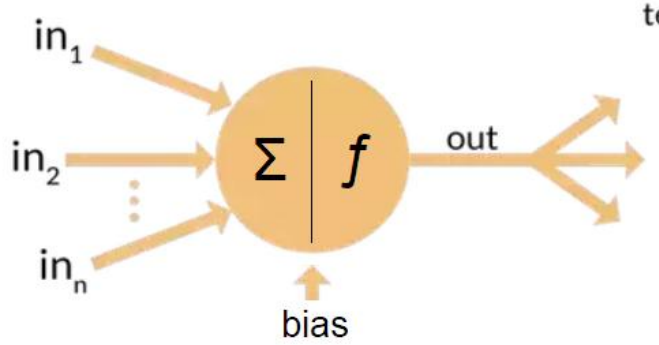
=

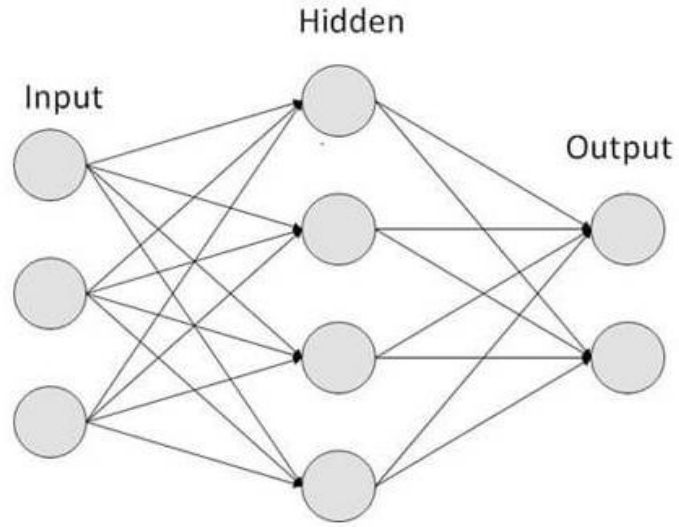
Logistic Regression?!

A bit of biology

- Eventually the output signal produced by a neuron will be passed to other neurons in the brain.
- The brain could be seen as a set of several neurons processing information, **both in parallel and in sequence**.
- This assembly, which resembles the human brain, is commonly referred to as a **neural network**.



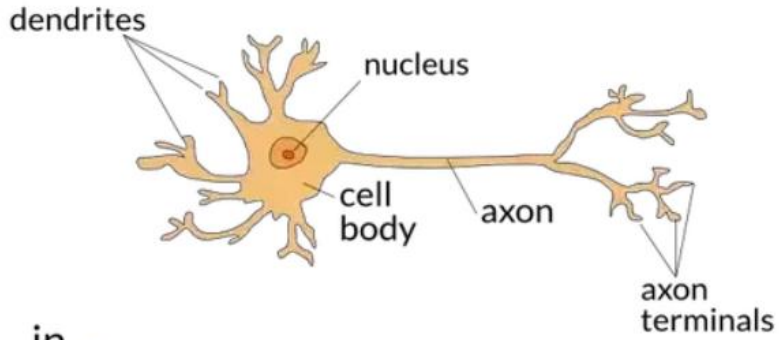


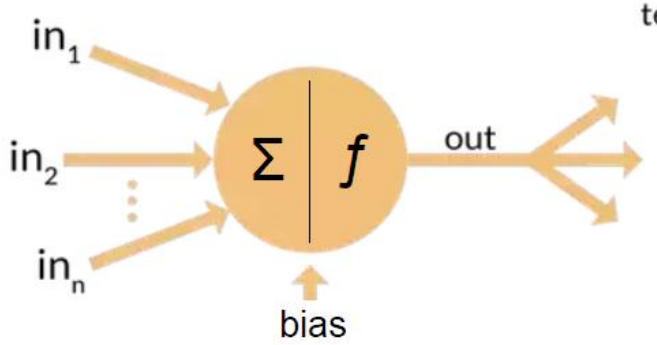


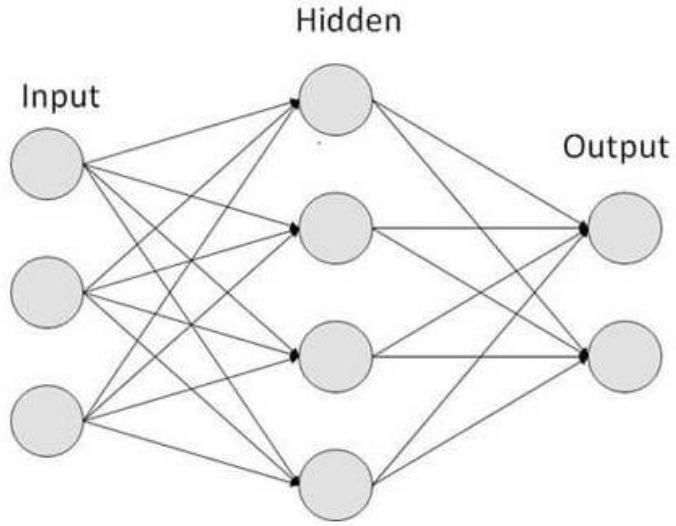
A bit of biology

Research has shown that artificial neural networks could indeed be used to solve some of the 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.
- The training process of CS neural networks is apparently very similar to the biological neural networks.
- This is actually an 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 200 sqm,
- randomly drawing **distances** between 50 and 1000m,
- generate prices by assuming that the average price is simply defined as $avgprice = 100000 + 14373 \times surface + (1000 - distance) \times 1286$

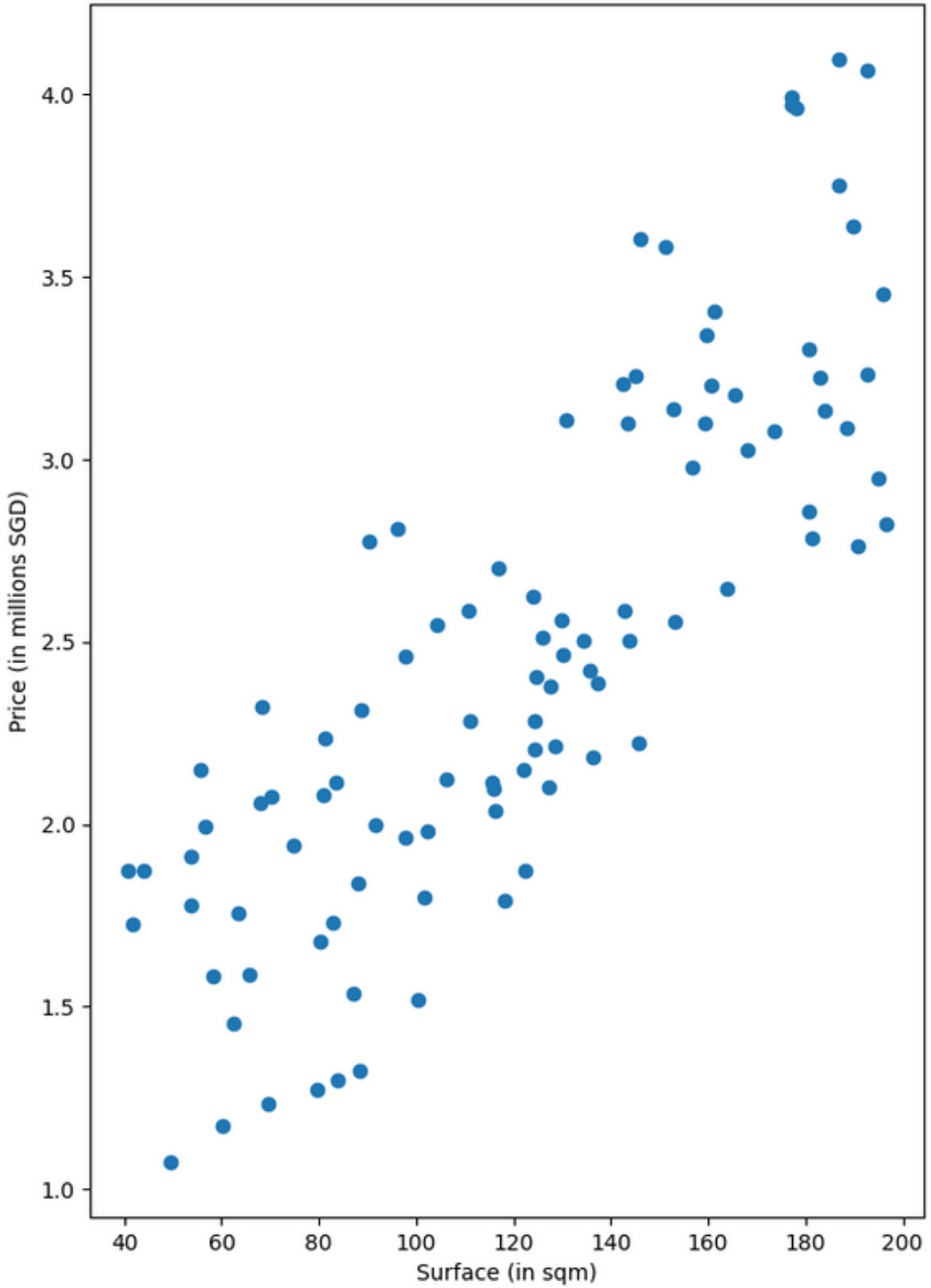
Finally, we will randomly apply a -/+ 10% variation on the average price to create some variance, and call that the output for a given sample.

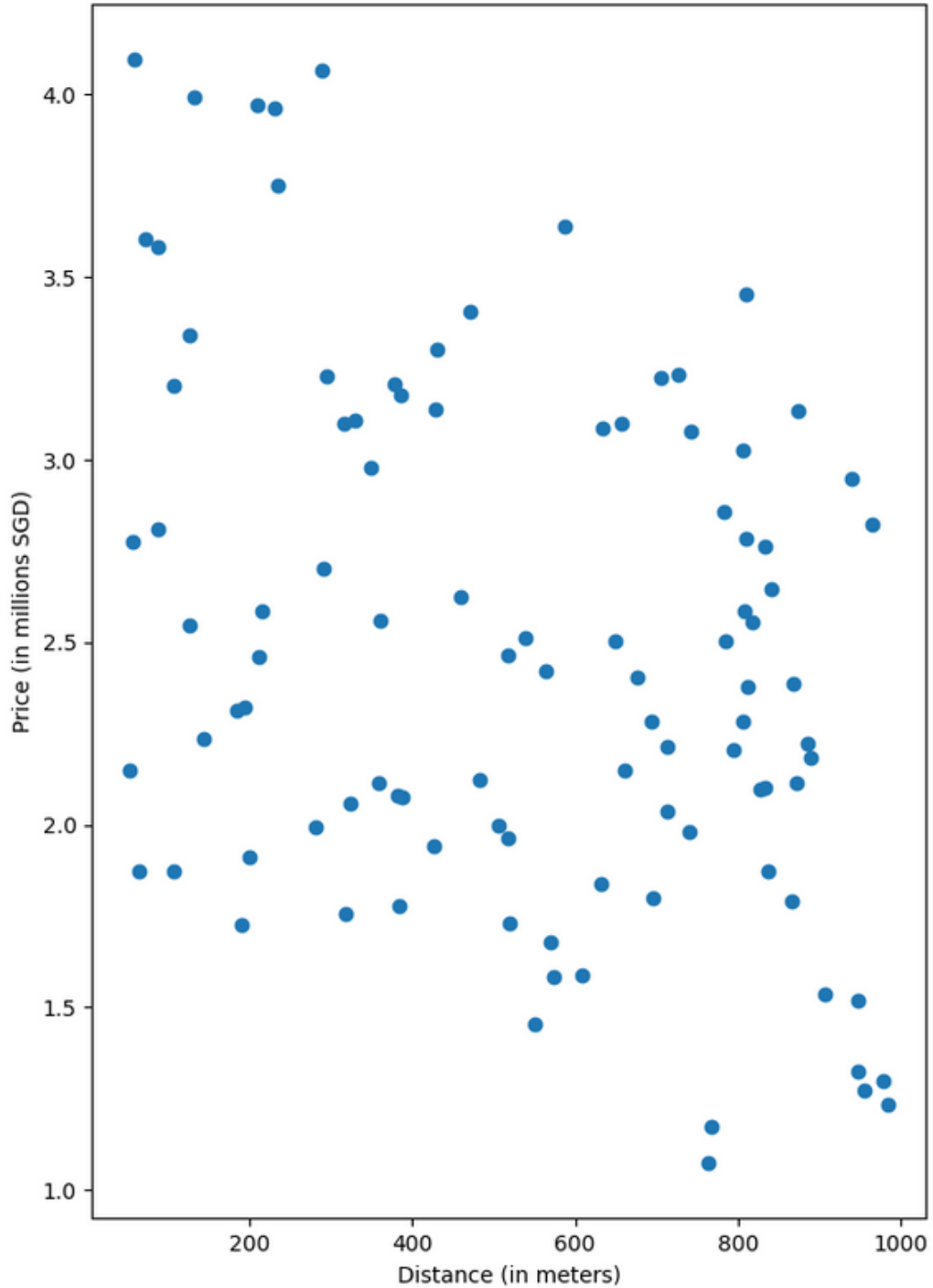
Starting with a mock dataset, again

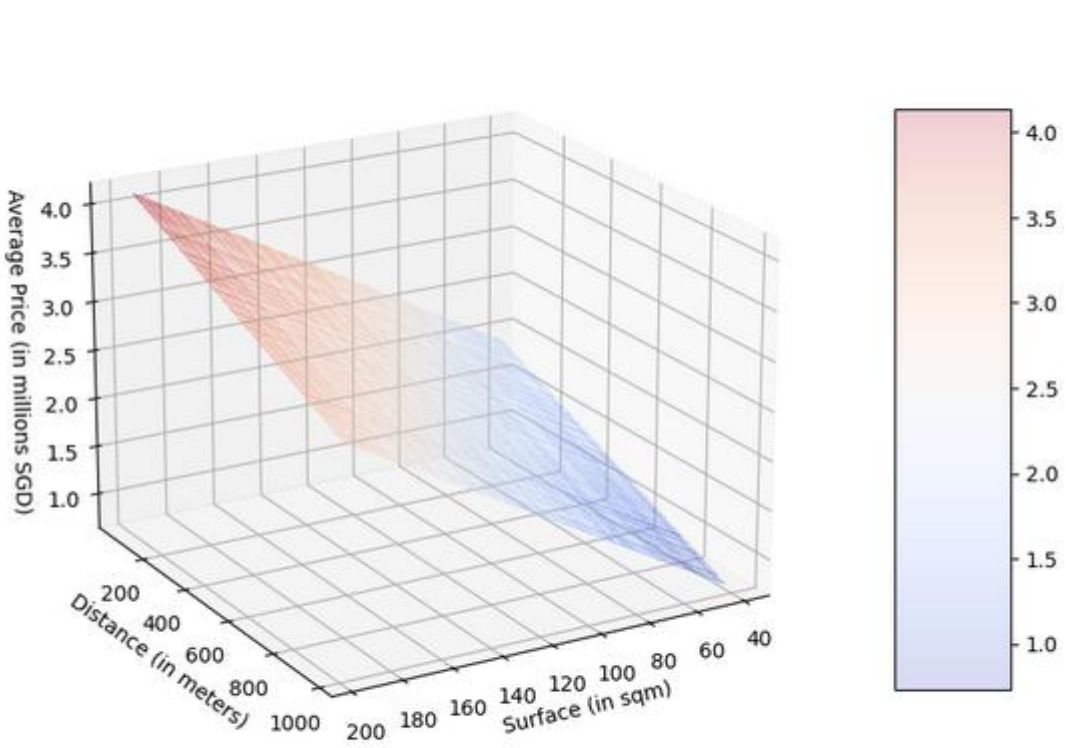
```python
# All helper functions
min_surf = 40
max_surf = 200
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
min_dist = 50
max_dist = 1000
def distance(min_dist, max_dist):
    return round(np.random.uniform(min_dist, max_dist), 2)
def price(surface, distance):
    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
```

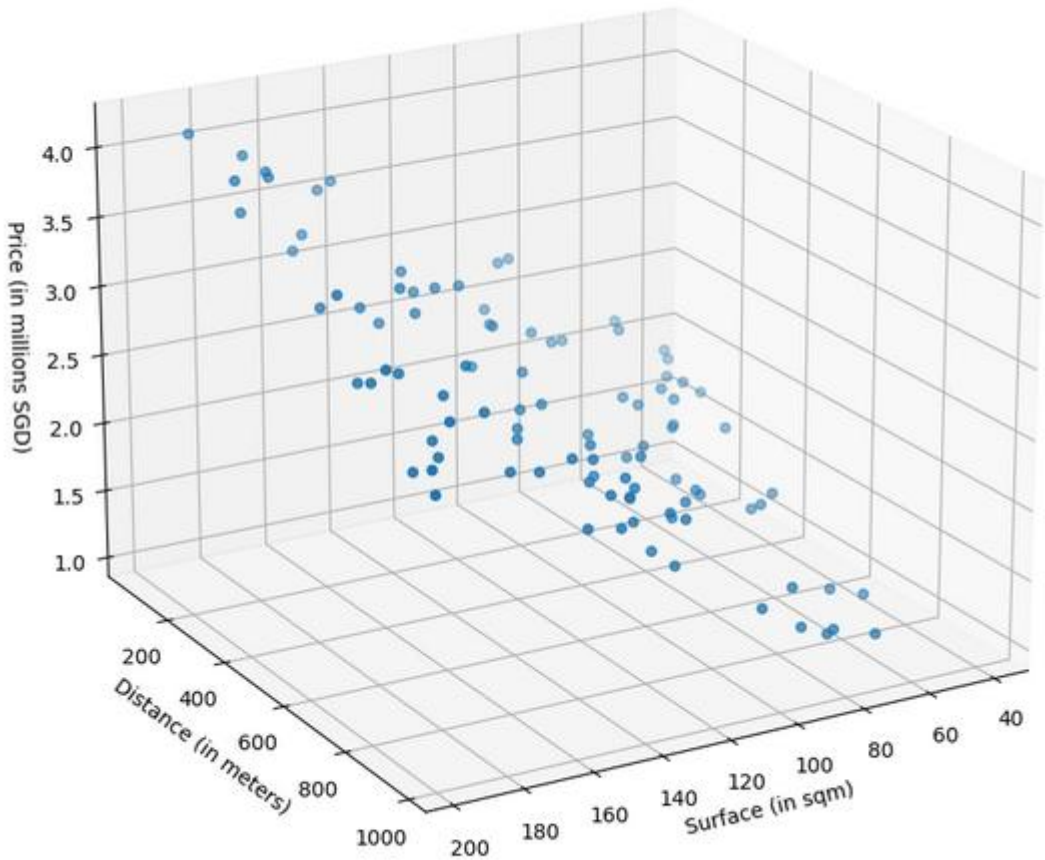
```python
# Generate dataset
np.random.seed(47)
surfaces_list, distances_list, inputs, outputs = create_dataset(n_points, \
                                                                min_surf, \
                                                                max_surf, \
                                                                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)





$$avgprice = 100000 + 14373 \times surface + (1000 - distance) \times 1286$$

is actually the equation of a 2D plane in a 3D ambient space!

Coding a Neuron and a Neural Network

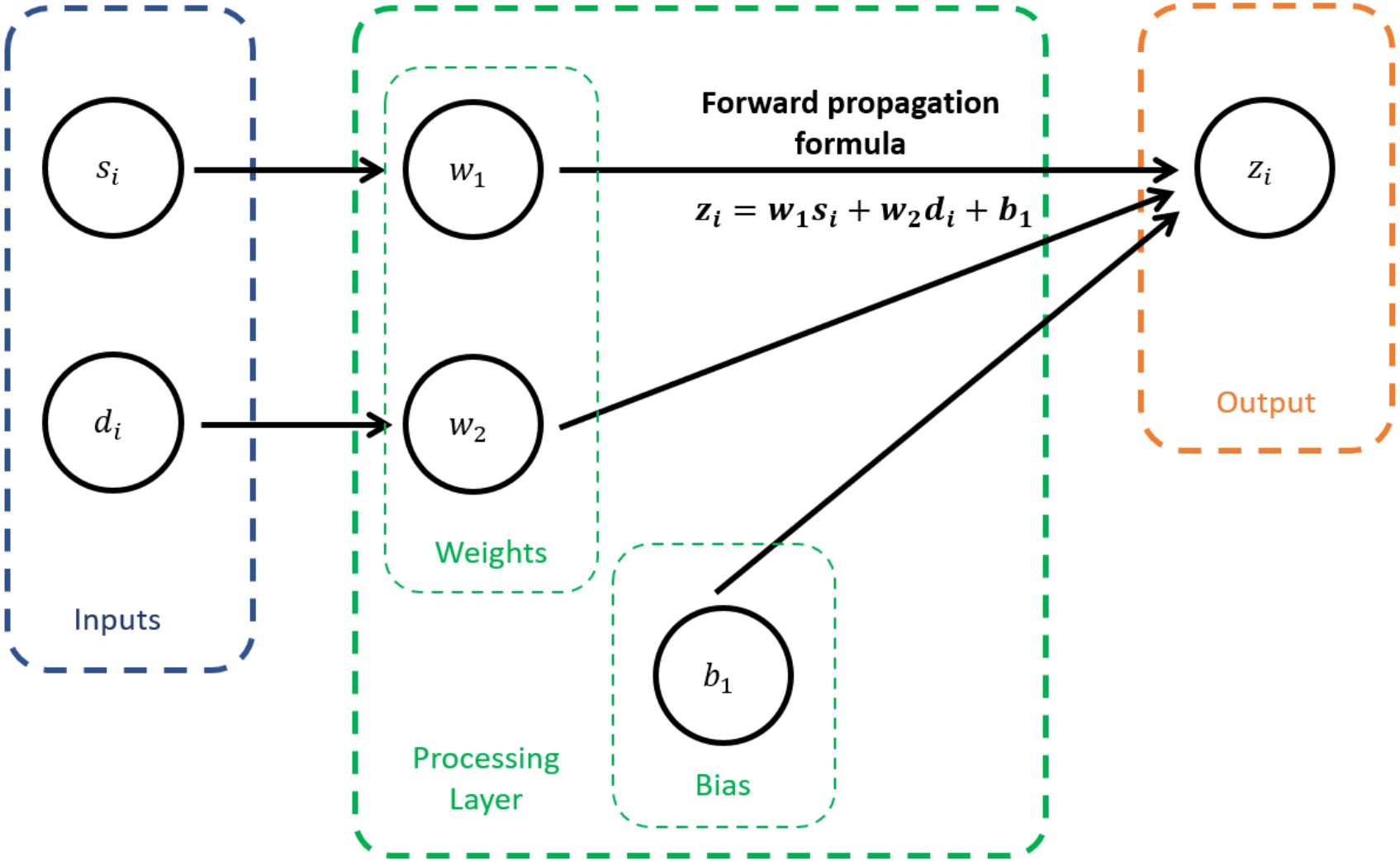
Definition (Neuron in Neural Networks):

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, in parallel and in sequence, through a network. This neural network can be thought of as pathways for information to flow.

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

Coding our first Neuron



```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 with only one neuron, 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 single neuron is equivalent to a 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 problem (task, dataset, model).

The **fourth** element we need to define is a **loss function**.

- As this is a regression task, we can reuse the **MSE loss** and add this MSE loss function as a method to our class.
- For convenience, we will compute the MSE loss in in matrix form.
- 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
```

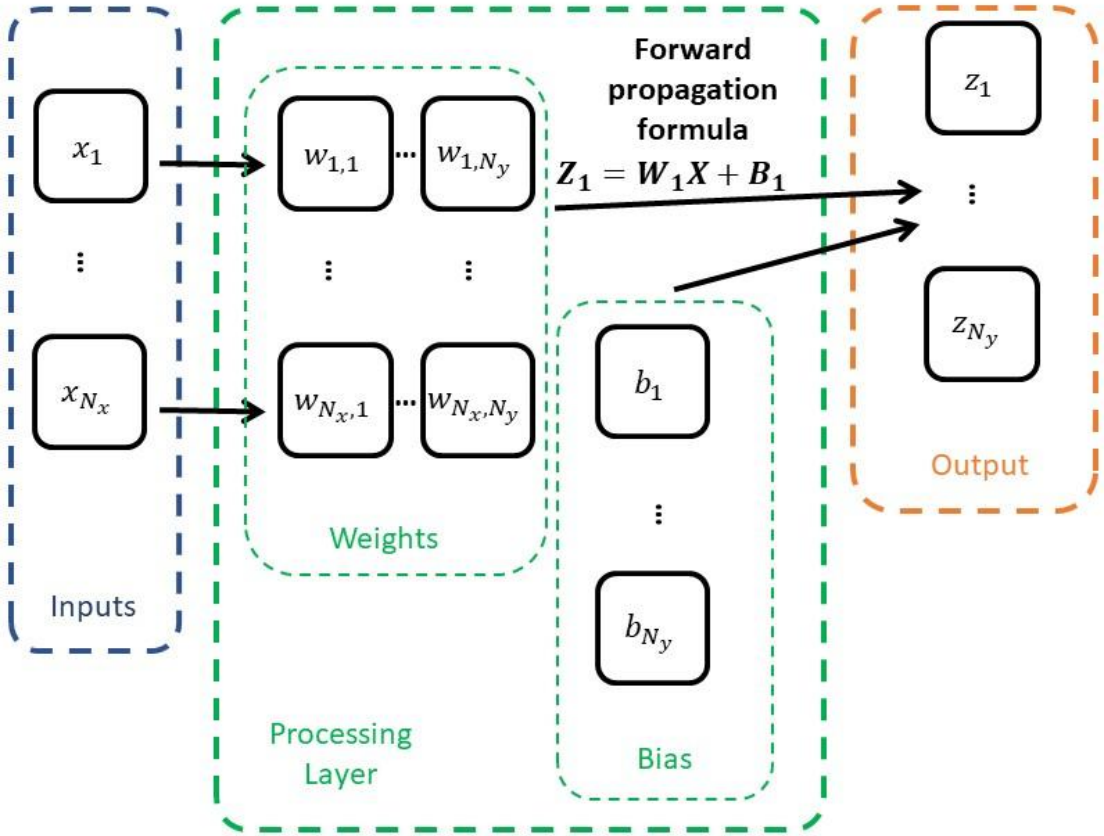
Scaling up with more parameters

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 matrices of parameters:

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

Note: This would be equivalent to having N_y neurons, each receiving N_x inputs, and each producing one output to be assembled as the Z vector.

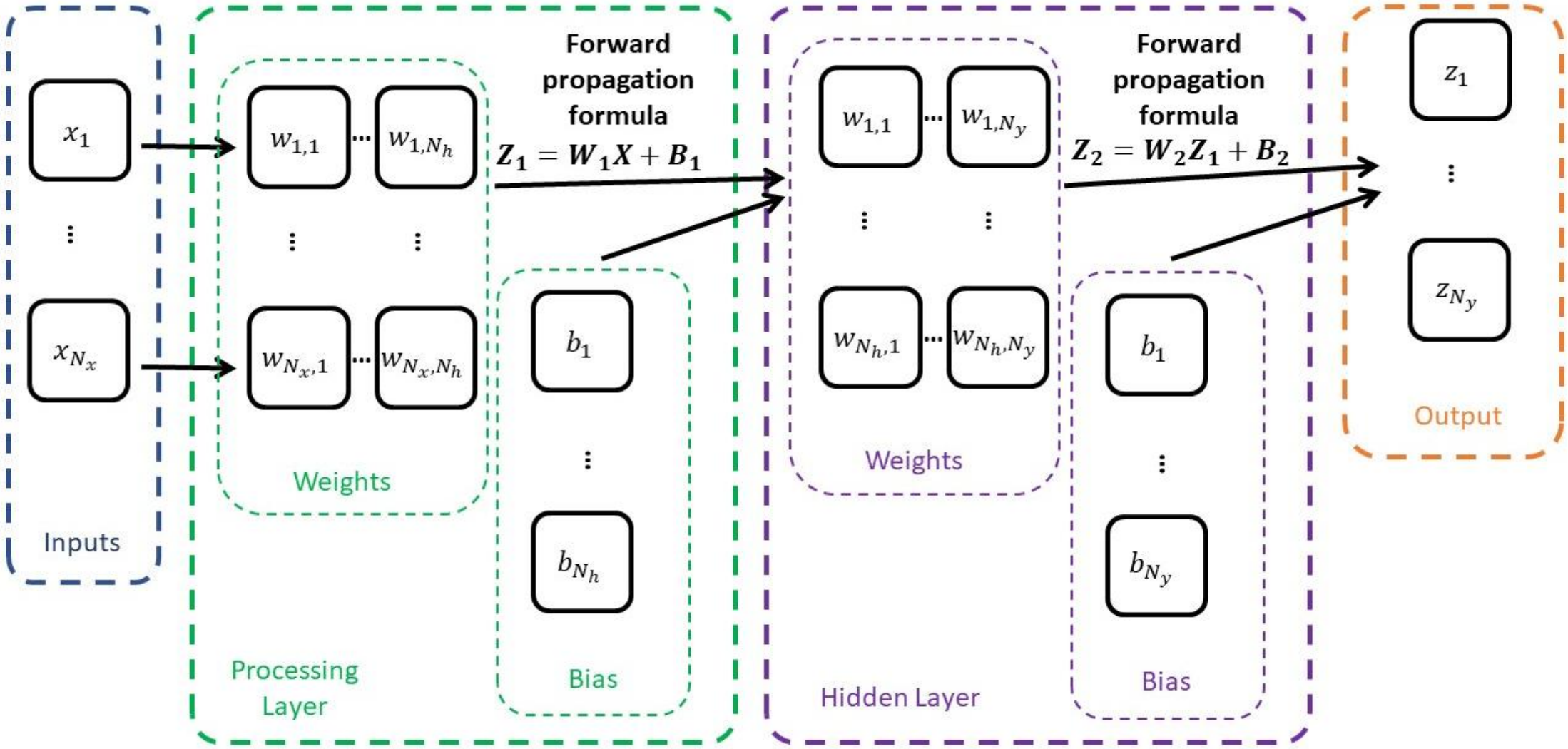


Scaling up with more layers

Definition (Shallow Neural Network):

Following the logic of our SimpleNeuralNet, we could implement a Shallow Neural Network. **It will include two processing layers of neurons in a row, instead of just one, and each layer will be implementing its own $WX + b$ operation of some sort.**

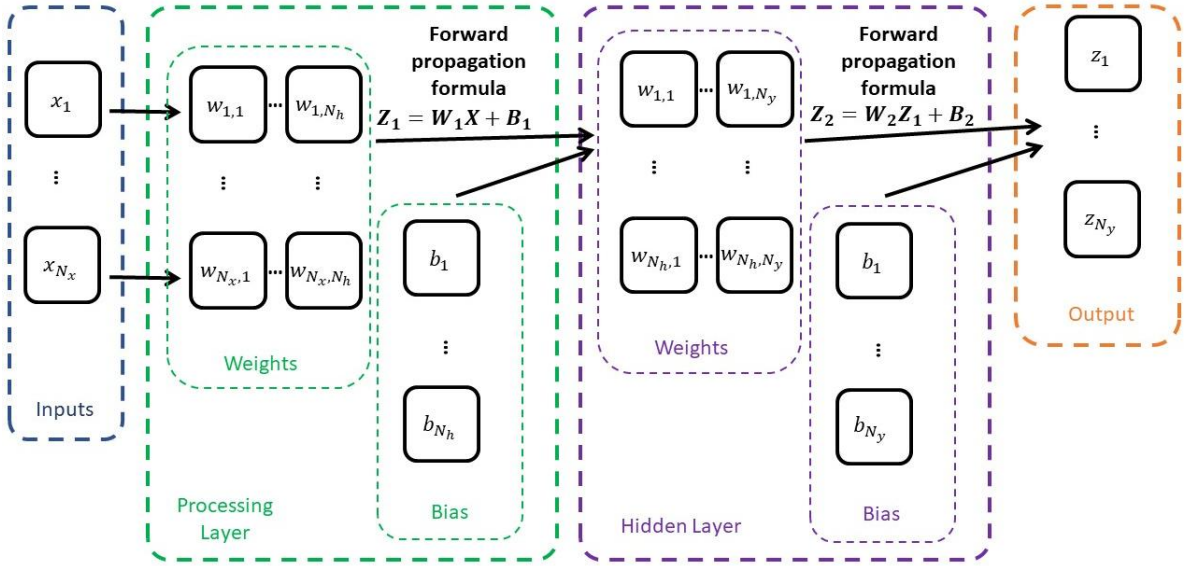
Scaling up with more layers



Scaling up with more layers

The first layer, will receive inputs with dimensionality n_x and produce outputs with dimensionality n_h.

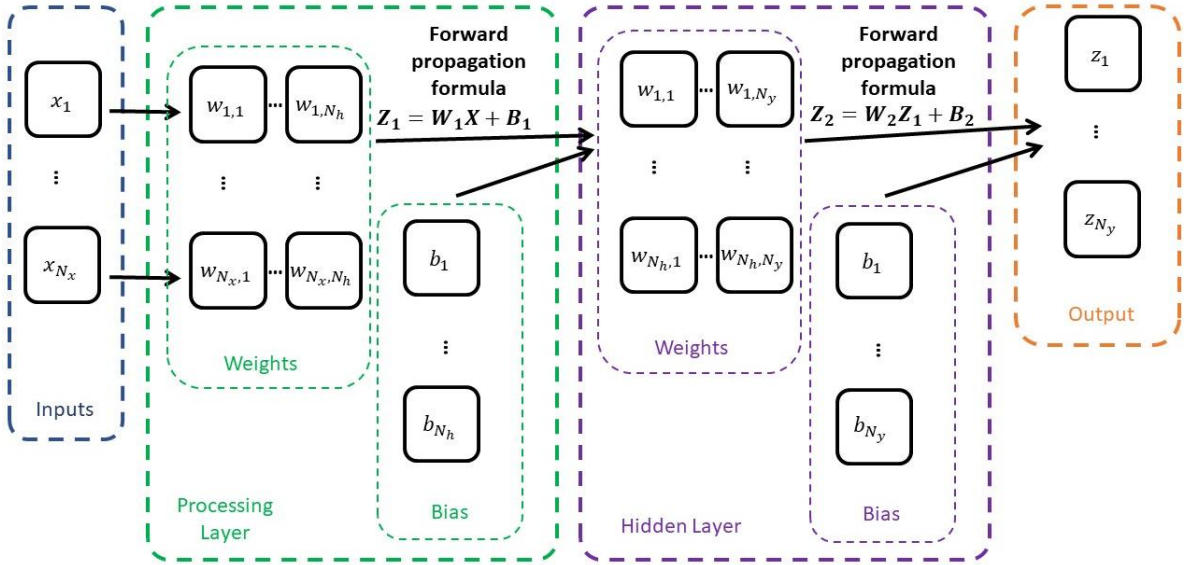
- The matrix W for this first layer will therefore be of size $n_x \times n_h$, and the matrix b will be a simple 1D vector with size n_h.



Scaling up with more layers

The second layer, also called **hidden layer**, will receive the outputs from the previous layer with dimensionality n_h, use them as inputs and will produce outputs matching the dimensionality of the outputs in our dataset, n_y.

- The matrix W for this second layer will be 2D $n_h \times n_y$, and b will be 1D n_y.



Our NN class

Small changes:

- Weights and biases are now **randomly initialized**, instead of being passed to the init method.
- We pass the **expected sizes** n_x, n_h, n_y instead.
- **And initialize parameters matrices as normal random with zero mean and variance 0.1.**

```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}
```

```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]]
```

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

```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 adjust 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!**
- **What could it be…?**

```
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
```

Let us call it a break for now

We will continue the next lecture with the training procedure for this Neural Network we just built.