# CHAPTER 1

# INTRODUCTION

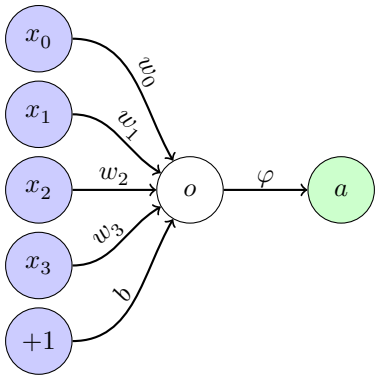
In this introduction chapter, we will present a first neural network called the Perceptron. This model is a neural network made of a single neuron, and we will use it here as a way to introduce key concepts that we will detail later in the course.

## 1.1 A first model: the Perceptron

In the neural network terminology, a neuron is a parametrized function that takes a vector $\mathbf{x}$ as input and outputs a single value $a$ as follows:

$$a = \varphi(\underbrace{\mathbf{w}\mathbf{x} + b}_{o}),$$

where the parameters of the neuron are its weights stored in $\mathbf{w}$ and a bias term $b$, and $\varphi$ is an activation function that is chosen *a priori* (we will come back to it in more details later in the course):




A model made of a single neuron is called a Perceptron.

## 1.2 Optimization

The models presented in this book are aimed at solving prediction problems, in which the goal is to find "good enough" parameter values for the model at stake given some observed data.

The problem of finding such parameter values is coined optimization and the deep learning field makes extensive use of a specific family of optimization strategies called **gradient descent**.

### 1.2.1 Gradient Descent

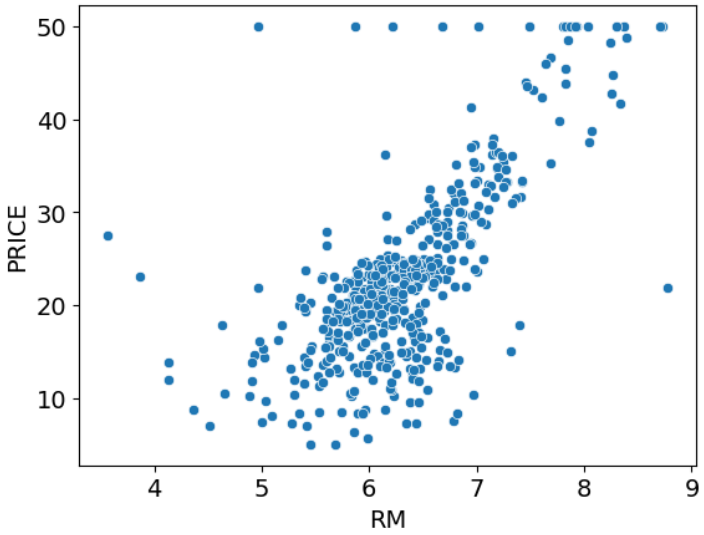
To make one's mind about gradient descent, let us assume we are given the following dataset about house prices:

```
import pandas as pd

boston = pd.read_csv("../data/boston.csv")[["RM", "PRICE"]]
boston
```

```
        RM  PRICE
0    6.575   24.0
1    6.421   21.6
2    7.185   34.7
3    6.998   33.4
4    7.147   36.2
..     ...    ...
501  6.593   22.4
502  6.120   20.6
503  6.976   23.9
504  6.794   22.0
505  6.030   11.9

[506 rows x 2 columns]
```

In our case, we will try (for a start) to predict the target value of this dataset, which is the median value of owner-occupied homes in $1000 `"PRICE"`, as a function of the average number of rooms per dwelling `"RM"` :

```
sns.scatterplot(data=boston, x="RM", y="PRICE");
```




**A short note on this model**

In the Perceptron terminology, this model:

- has no activation function (*i.e.* $\varphi$ is the identity function)
- has no bias (*i.e.* $b$ is forced to be $0$, it is not learnt)

Let us assume we have a naive approach in which our prediction model is linear without intercept, that is, for a given input $x_i$ the predicted output is computed as:

$$\hat{y}_i = wx_i$$

where $w$ is the only parameter of our model.

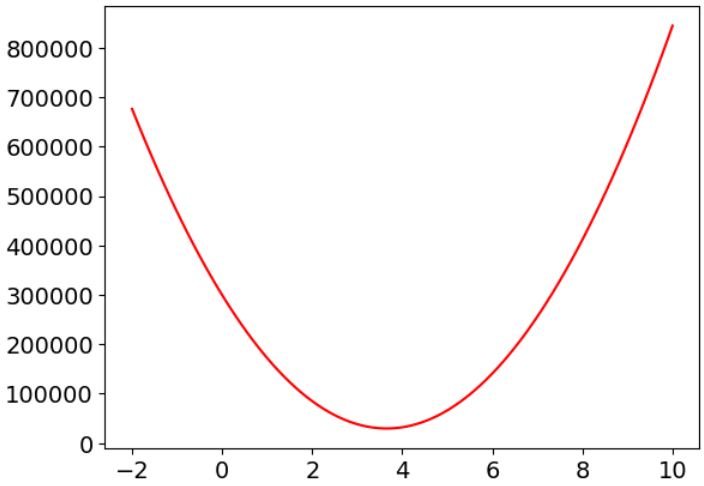
Let us further assume that the quantity we aim at minimizing (our objective, also called loss) is:

$$\mathcal{L}(w) = \sum_i (\hat{y}_i - y_i)^2$$

where $y_i$ is the ground truth value associated with the $i$-th sample in our dataset.

Let us have a look at this quantity as a function of $w$:

```
import numpy as np

def loss(w, x, y):
    w = np.array(w)
    return np.sum(
        (w[:, None] * x.to_numpy()[None, :] - y.to_numpy()[None, :]) ** 2,
        axis=1
    )

w = np.linspace(-2, 10, num=100)

x = boston["RM"]
y = boston["PRICE"]
plt.plot(w, loss(w, x, y), "r-");
```




Here, it seems that a value of $w$ around 4 should be a good pick, but this method (generating lots of values for the parameter and computing the loss for each value) cannot scale to models that have lots of parameters, so we will try something else.

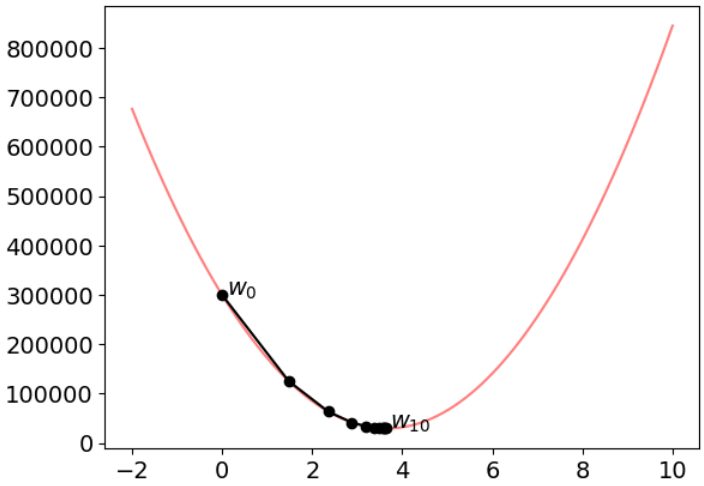
Let us suppose we have access, each time we pick a candidate value for $w$, to both the loss $\mathcal{L}$ and information about how $\mathcal{L}$ varies, locally. We could, in this case, compute a new candidate value for $w$ by moving from the previous candidate value in the direction of steepest descent. This is the basic idea behind the gradient descent algorithm that, from an initial candidate $w_0$, iteratively computes new candidates as:

$$w_{t+1} = w_t - \rho \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$$

where $\rho$ is a hyper-parameter (called the learning rate) that controls the size of the steps to be done, and $\left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$ is the gradient of $\mathcal{L}$ with respect to $w$, evaluated at $w = w_t$. As you can see, the direction of steepest descent is the opposite of the direction pointed by the gradient (and this holds when dealing with vector parameters too).

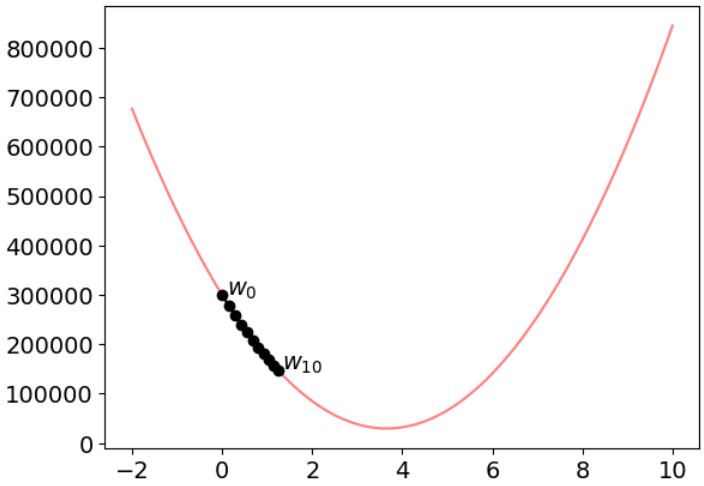
This process is repeated until convergence, as illustrated in the following visualization:

```
rho = 1e-5

def grad_loss(w_t, x, y):
    return np.sum(
        2 * (w_t * x - y) * x
    )


ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]+.1, y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]+.1, y=loss([w[10]], x, y), s="$w_{10}$");
```




What would we get if we used a smaller learning rate?

```
rho = 1e-6

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]+.1, y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]+.1, y=loss([w[10]], x, y), s="$w_{10}$");
```



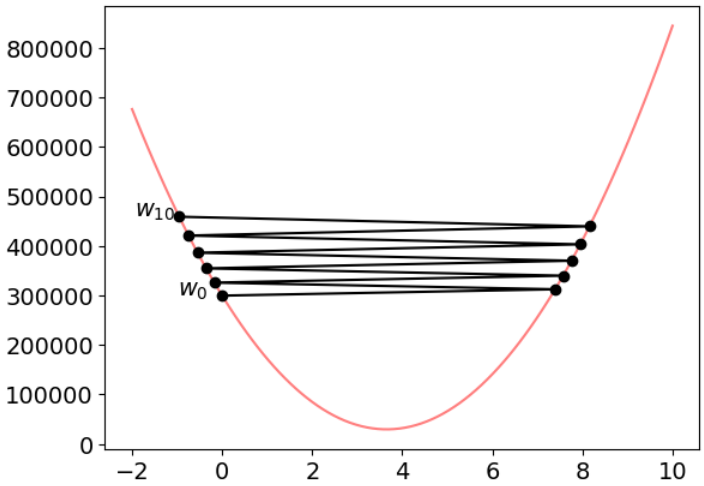

It would definitely take more time to converge. But, take care, a larger learning rate is not always a good idea:

```
rho = 5e-5

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);
```

(continues on next page)

(continued from previous page)

```
w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]-1., y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]-1., y=loss([w[10]], x, y), s="$w_{10}$");
```




See how we are slowly diverging because our steps are too large?

## 1.3 Wrap-up

In this section, we have introduced:

- a very simple model, called the Perceptron: this will be a building block for the more advanced models we will detail later in the course, such as:
  - the *Multi-Layer Perceptron*
  - *Convolutional architectures*
  - *Recurrent architectures*
- the fact that a task comes with a loss function to be minimized (here, we have used the *mean squared error (MSE)* for our regression task), which will be discussed in *a dedicated chapter*;
- the concept of gradient descent to optimize the chosen loss over a model's single parameter, and this will be extended in *our chapter on optimization*.

# CHAPTER 2

# MULTI LAYER PERCEPTRONS

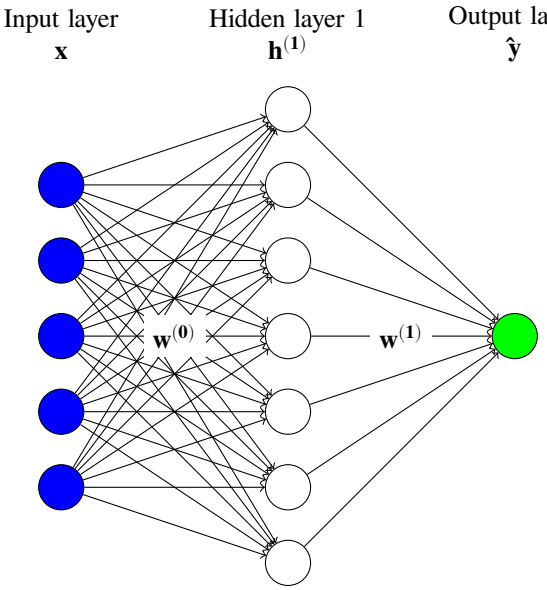
In the previous chapter, we have seen a very simple model called the Perceptron. In this model, the predicted output $\hat{y}$ is computed as a linear combination of the input features plus a bias:

$$\hat{y} = \sum_{j=1}^{d} x_j w_j + b$$

In other words, we were optimizing among the family of linear models, which is a quite restricted family.

## 2.1 Stacking layers for better expressivity

In order to cover a wider range of models, one can stack neurons organized in layers to form a more complex model, such as the model below, which is called a one-hidden-layer model, since an extra layer of neurons is introduced between the inputs and the output:



The question one might ask now is whether this added hidden layer effectively allows to cover a wider family of models. This is what the Universal Approximation Theorem below is all about.

---

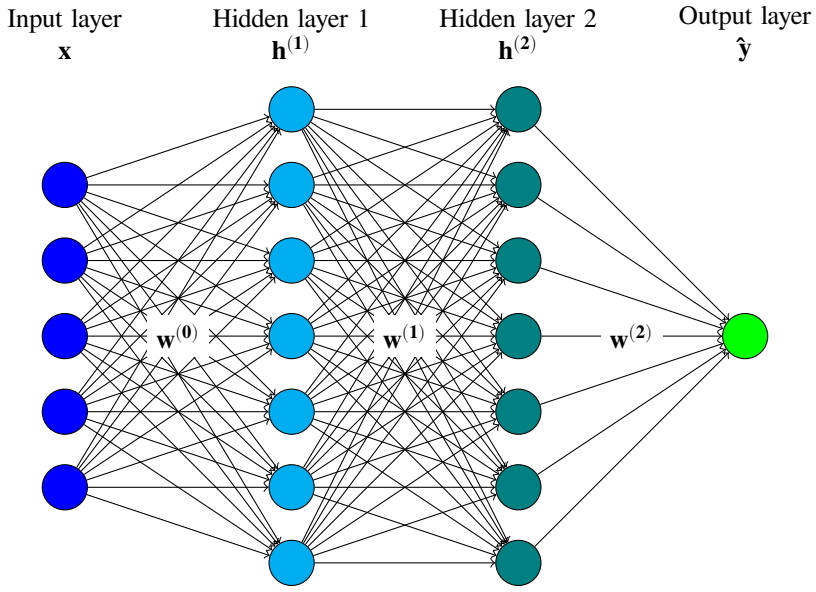
**Universal Approximation Theorem**

The Universal Approximation Theorem states that any continuous function defined on a compact set can be approximated as closely as one wants by a one-hidden-layer neural network with sigmoid activation.

---

In other words, by using a hidden layer to map inputs to outputs, one can now approximate any continuous function, which is a very interesting property. Note however that the number of hidden neurons that is necessary to achieve a given approximation quality is not discussed here. Moreover, it is not sufficient that such a good approximation exists, another important question is whether the optimization algorithms we will use will eventually converge to this solution or not, which is not guaranteed, as discussed in more details in *the dedicated chapter*.

In practice, we observe empirically that in order to achieve a given approximation quality, it is more efficient (in terms of the number of parameters required) to stack several hidden layers rather than rely on a single one:




The above graphical representation corresponds to the following model:

$$\hat{y} = \varphi_{\text{out}}\left(\sum_i w_i^{(2)} h_i^{(2)} + b^{(2)}\right) \tag{2.1}$$

$$\forall i, h_i^{(2)} = \varphi\left(\sum_j w_{ij}^{(1)} h_j^{(1)} + b_i^{(1)}\right) \tag{2.2}$$

$$\forall i, h_i^{(1)} = \varphi\left(\sum_j w_{ij}^{(0)} x_j + b_i^{(0)}\right) \tag{2.3}$$

To be even more precise, the bias terms $b_i^{(l)}$ are not represented in the graphical representation above.

Such models with one or more hidden layers are called **Multi Layer Perceptrons** (MLP).

## 2.2 Deciding on an MLP architecture

When designing a Multi-Layer Perceptron model to be used for a specific problem, some quantities are fixed by the problem at hand and other are left as hyper-parameters.

Let us take the example of the well-known Iris classification dataset:

```
import pandas as pd

iris = pd.read_csv("../data/iris.csv", index_col=0)
iris
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 ...               ...                ...               ...
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

     target
0         0
1         0
2         0
3         0
4         0
..      ...
145       2
146       2
147       2
148       2
149       2

[150 rows x 5 columns]
```

The goal here is to learn how to infer the `target` attribute (3 different possible classes) from the information in the 4 other attributes.

The structure of this dataset dictates:

- the number of neurons in the input layer, which is equal to the number of descriptive attributes in our dataset (here, 4), and
- the number of neurons in the output layer, which is here equal to 3, since the model is expected to output one probability per target class.

In more generality, for the output layer, one might face several situations:

- when regression is at stake, the number of neurons in the output layer is equal to the number of features to be predicted by the model,
- when it comes to classification

- in the case of binary classification, the model will have a single output neuron which will indicate the probability of the positive class
- in the case of multi-class classification, the model will have as many output neurons as the number of classes in the problem.

Once these number of input / output neurons are fixed, the number of hidden neurons as well as the number of neurons per hidden layer are left as hyper-parameters of the model.

## 2.3 Activation functions

Another important hyper-parameter of neural networks is the choice of the activation function $\varphi$.

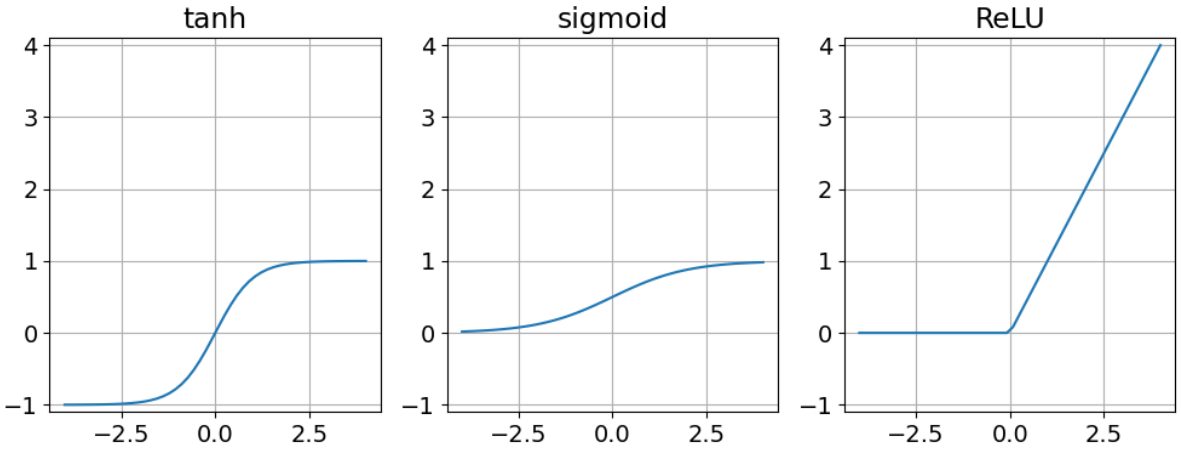
Here, it is important to notice that if we used the identity function as our activation function, then whatever the depth of our MLP, we would fall back to covering only the family of linear models. In practice, we will then use activation functions that have some linear regime but don't behave like a linear function on the whole range of input values.

Historically, the following activation functions have been proposed:

$$\tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\text{ReLU}(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases}$$




In practice the ReLU function (and some of its variants) is the most widely used nowadays, for reasons that will be discussed in more details in *our chapter dedicated to optimization*.

### 2.3.1 The special case of the output layer

You might have noticed that in the MLP formulation provided in Equation (1), the output layer has its own activation function, denoted $\varphi_{\text{out}}$. This is because the choice of activation functions for the output layer of a neural network is a bit specific to the problem at hand.

Indeed, you might have seen that the activation functions discussed in the previous section do not share the same range of output values. It is hence of prime importance to pick an adequate activation function for the output layer such that our model outputs values that are consistent to the quantities it is supposed to predict.

If, for example, our model was supposed to be used in the Boston Housing dataset we discussed *in the previous chapter*. In this case, the goal is to predict housing prices, which are expected to be nonnegative quantities. It would then be a good idea to use ReLU (which can output any positive value) as the activation function for the output layer in this case.

As stated earlier, in the case of binary classification, the model will have a single output neuron and this neuron will output the probability associated to the positive class. This quantity is expected to lie in the $[0, 1]$ interval, and the sigmoid activation function is then the default choice in this setting.

Finally, when multi-class classification is at stake, we have one neuron per output class and each neuron is expected to output the probability for a given class. In this context, the output values should be between 0 and 1, and they should sum to 1. For this purpose, we use the softmax activation function defined as:

$$\forall i, \text{softmax}(o_i) = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

where, for all $i$, $o_i$'s are the values of the output neurons before applying the activation function.

## 2.4 Declaring an MLP in `keras`

In order to define a MLP model in `keras`, one just has to stack layers. As an example, if one wants to code a model made of:

- an input layer with 10 neurons,
- a hidden layer made of 20 neurons with ReLU activation,
- an output layer made of 3 neurons with softmax activation,

the code will look like:

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential

model = Sequential([
    InputLayer(input_shape=(10, )),
    Dense(units=20, activation="relu"),
    Dense(units=3, activation="softmax")
])

model.summary()
```

```
Using TensorFlow backend
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

(continues on next page)

(continued from previous page)

```
=================================================================
 dense (Dense)               (None, 20)                220

 dense_1 (Dense)             (None, 3)                 63

=================================================================
Total params: 283 (1.11 KB)
Trainable params: 283 (1.11 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Note that `model.summary()` provides an interesting overview of a defined model and its parameters.

---

**Exercise #1**

Relying on what we have seen in this chapter, can you explain the number of parameters returned by `model.summary()` above?

---

**Solution**

Our input layer is made of 10 neurons, and our first layer is fully connected, hence each of these neurons is connected to a neuron in the hidden layer through a parameter, which already makes $10 \times 20 = 200$ parameters. Moreover, each of the hidden layer neurons has its own bias parameter, which is 20 more parameters. We then have 220 parameters, as output by `model.summary()` for the layer `"dense (Dense)"`.

Similarly, for the connection of the hidden layer neurons to those in the output layer, the total number of parameters is $20 \times 3 = 60$ for the weights plus 3 extra parameters for the biases.

Overall, we have $220 + 63 = 283$ parameters in this model.

---

---

**Exercise #2**

Declare, in `keras`, an MLP with one hidden layer made of 100 neurons and ReLU activation for the Iris dataset presented above.

---

**Solution**

```
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=100, activation="relu"),
    Dense(units=3, activation="softmax")
])
```

---

---

**Exercise #3**

Same question for the full Boston Housing dataset shown below (the goal here is to predict the `PRICE` feature based on the other ones).

---

**Solution**

```
model = Sequential([
    InputLayer(input_shape=(6, )),
    Dense(units=100, activation="relu"),
    Dense(units=1, activation="relu")
])
```

```
        RM     CRIM  INDUS    NOX   AGE    TAX  PRICE
0    6.575  0.00632   2.31  0.538  65.2  296.0   24.0
1    6.421  0.02731   7.07  0.469  78.9  242.0   21.6
2    7.185  0.02729   7.07  0.469  61.1  242.0   34.7
3    6.998  0.03237   2.18  0.458  45.8  222.0   33.4
4    7.147  0.06905   2.18  0.458  54.2  222.0   36.2
..     ...      ...    ...    ...   ...    ...    ...
501  6.593  0.06263  11.93  0.573  69.1  273.0   22.4
502  6.120  0.04527  11.93  0.573  76.7  273.0   20.6
503  6.976  0.06076  11.93  0.573  91.0  273.0   23.9
504  6.794  0.10959  11.93  0.573  89.3  273.0   22.0
505  6.030  0.04741  11.93  0.573  80.8  273.0   11.9

[506 rows x 7 columns]
```

# CHAPTER 3

# LOSSES

We have now presented a first family of models, which is the MLP family. In order to train these models (*i.e.* tune their parameters to fit the data), we need to define a loss function to be optimized. Indeed, once this loss function is picked, optimization will consist in tuning the model parameters so as to minimize the loss.

In this section, we will present two standard losses, that are the mean squared error (that is mainly used for regression) and logistic loss (which is used in classification settings).

In the following, we assume that we are given a dataset $\mathcal{D}$ made of $n$ annotated samples $(x_i, y_i)$, and we denote the model's output:

$$\forall i, \widehat{y}_i = m_\theta(x_i)$$

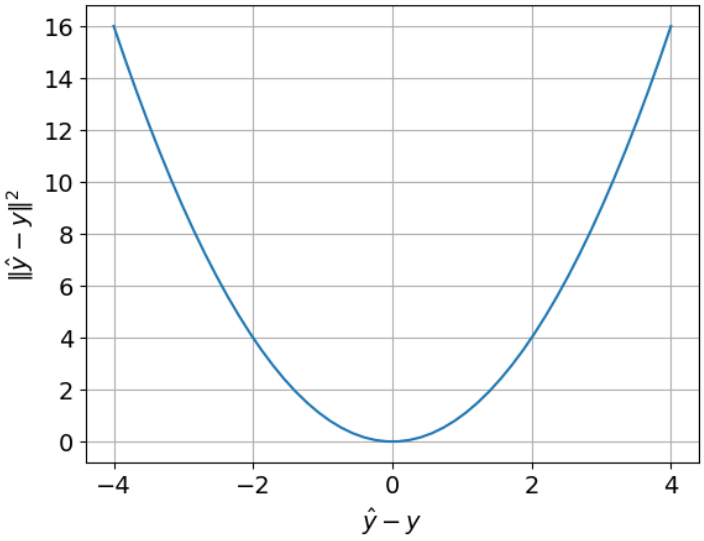
where $m_\theta$ is our model and $\theta$ is the set of all its parameters (weights and biases).

## 3.1 Mean Squared Error

The Mean Squared Error (MSE) is the most commonly used loss function in regression settings. It is defined as:

$$\begin{aligned}\mathcal{L}(\mathcal{D}; m_\theta) &= \frac{1}{n}\sum_i \|\widehat{y}_i - y_i\|^2 \\ &= \frac{1}{n}\sum_i \|m_\theta(x_i) - y_i\|^2\end{aligned}$$

Its quadratic formulation tends to strongly penalize large errors:




## 3.2 Logistic loss

The logistic loss is the most widely used loss to train neural networks in classification settings. It is defined as:

$$\mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_i -\log p(\hat{y}_i = y_i; m_\theta)$$

where $p(\hat{y}_i = y_i; m_\theta)$ is the probability predicted by model $m_\theta$ for the correct class $y_i$.

Its formulation tends to favor cases where the model outputs a probability of 1 for the correct class, as expected: