Deep Learning Basics (lecture notes)

Romain Tavenard

November 28, 2023

CONTENTS

by Romain Tavenard

This document serves as lecture notes for a course that is taught at Université de Rennes 2 (France) and EDHEC Lille (France).

The course deals with the basics of neural networks for classification and regression over tabular data (including optimization algorithms for multi-layer perceptrons), convolutional neural networks for image classification (including notions of transfer learning) and sequence classification / forecasting.

The labs for this course will use `keras`, hence so will these lecture notes.

CHAPTER 1

INTRODUCTION

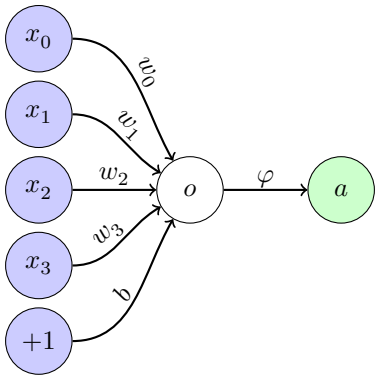
In this introduction chapter, we will present a first neural network called the Perceptron. This model is a neural network made of a single neuron, and we will use it here as a way to introduce key concepts that we will detail later in the course.

1.1 A first model: the Perceptron

In the neural network terminology, a neuron is a parametrized function that takes a vector **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 **w** and a bias term b, and φ 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

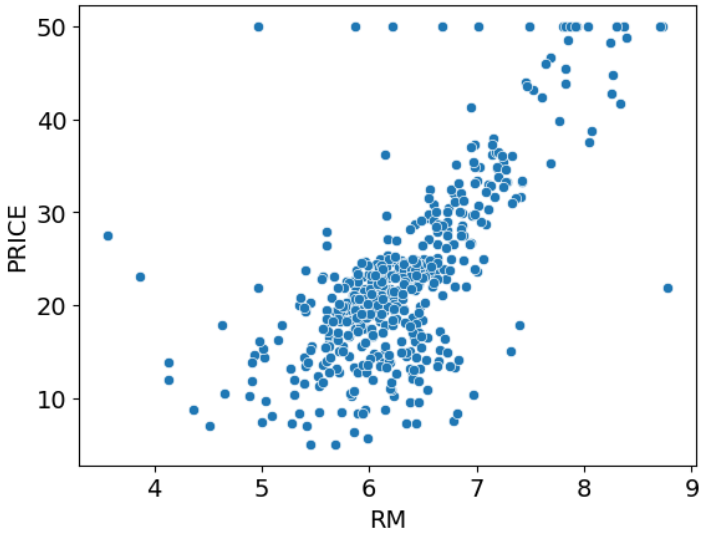
To make one's mind about gradient descent, let us assume we are given the following dataset about house prices:

```
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.* φ 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.

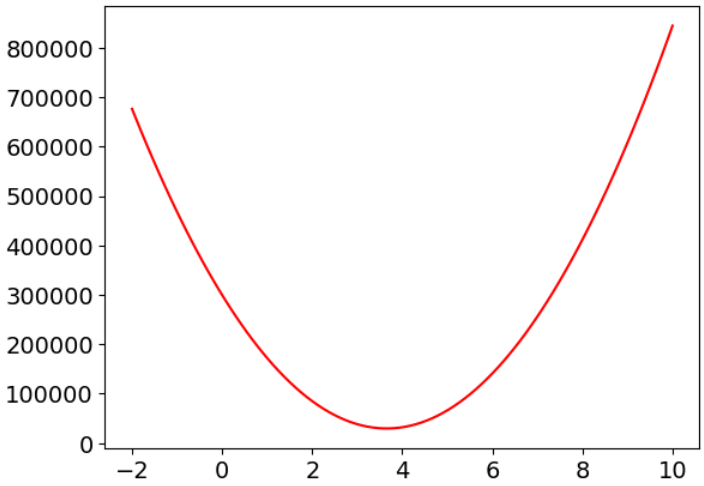
Let us further assume that the quantity we aim at minimizing (our objective, also called loss) is:

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

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.

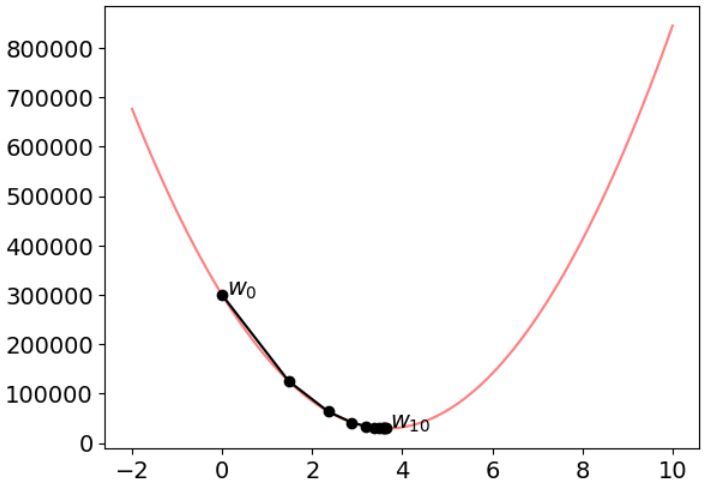
Let us suppose we have access, each time we pick a candidate value for w, to both the loss $\mathcal{L}$ and information about how $\mathcal{L}$ varies, locally. We could, in this case, compute a new candidate value for w by moving from the previous candidate value in the direction of steepest descent. 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 ρ 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).

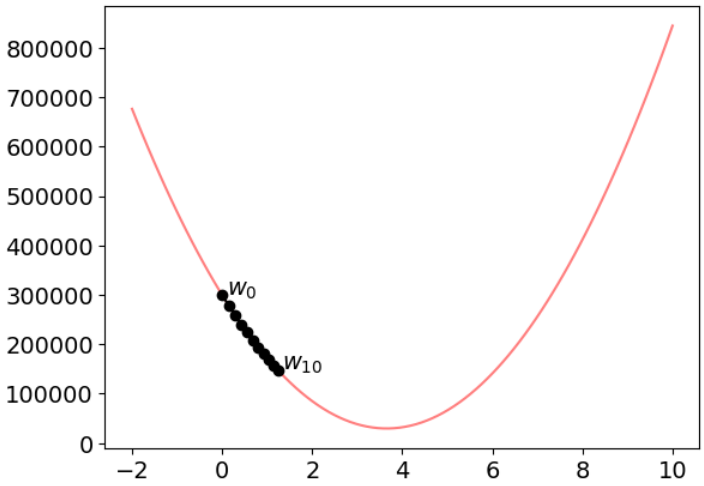
This process is repeated until convergence, as illustrated in the following visualization:

```
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="$w_{10}$");
```



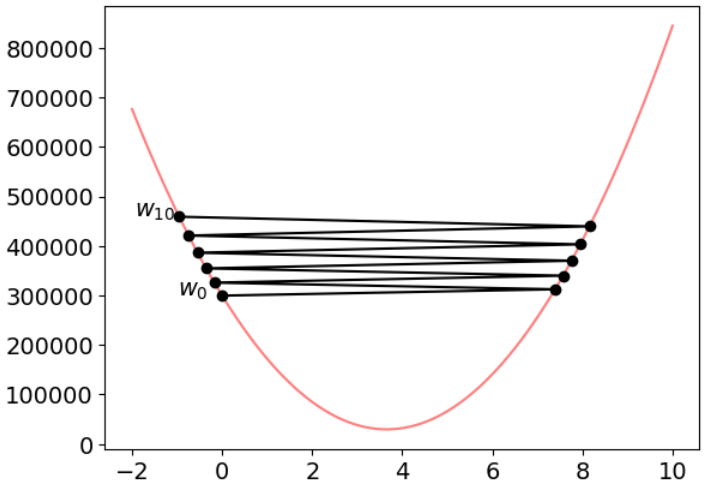
It would definitely take more time to converge. But, take care, a larger learning rate is not always a good idea:

```
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

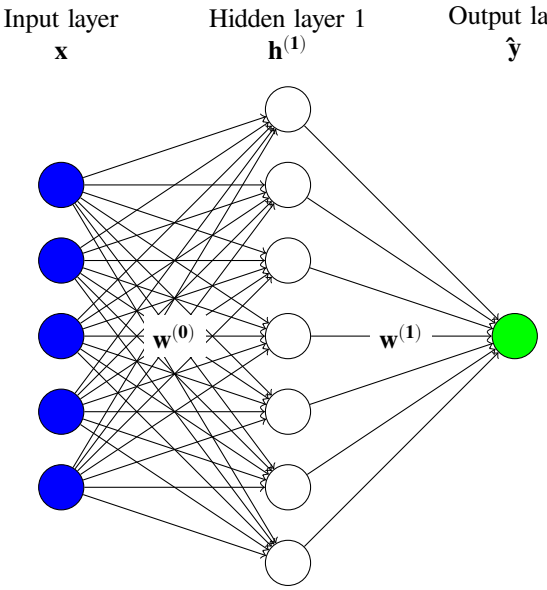
In the previous chapter, we have seen a very simple model called the Perceptron. In this model, the predicted output $\hat{y}$ is computed as a linear combination of the input features plus a bias:

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

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:



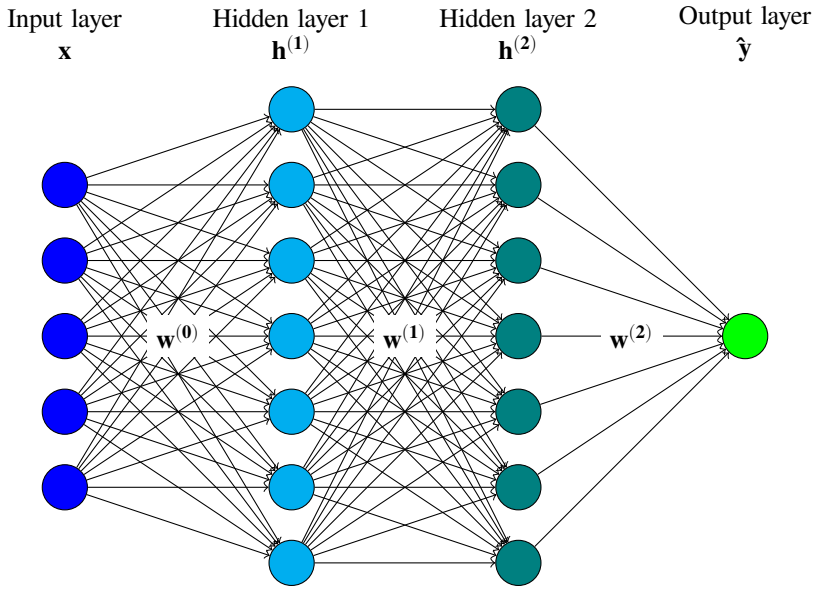
The question one might ask now is whether this added hidden layer effectively allows to cover a wider family of models. This is what the Universal Approximation Theorem below is all about.

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 φ.

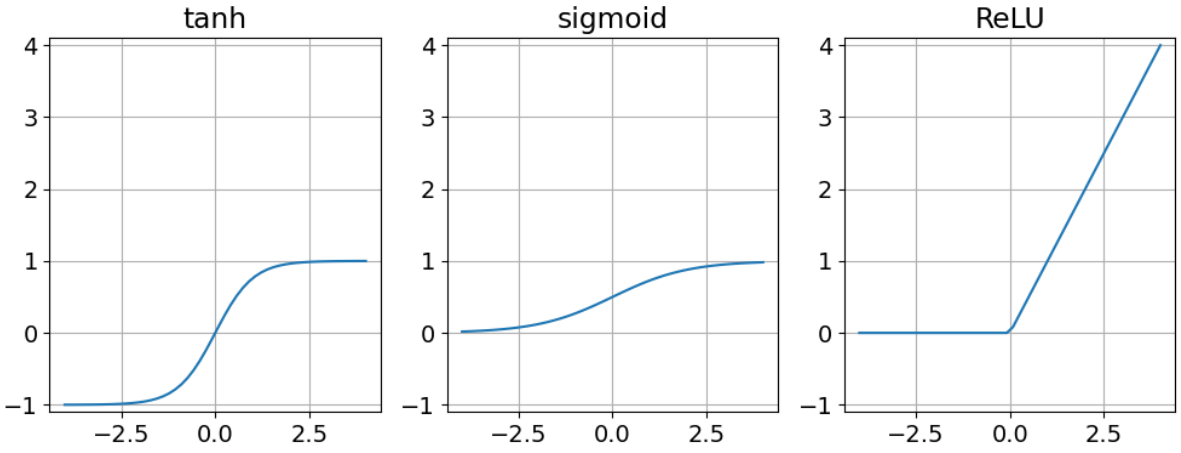
Here, it is important to notice that if we used the identity function as our activation function, then whatever the depth of our MLP, we would fall back to covering only the family of linear models. In practice, we will then use activation functions that have some linear regime but don't behave like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

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

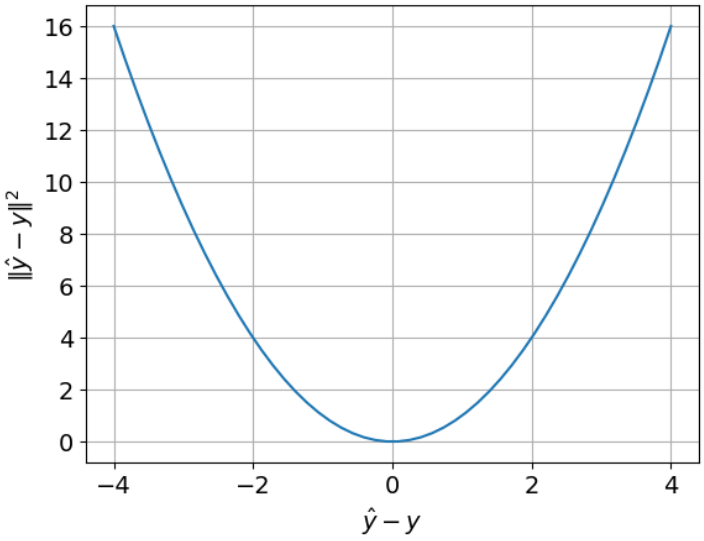
where m_θ is our model and θ is the set of all its parameters (weights and biases).

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:



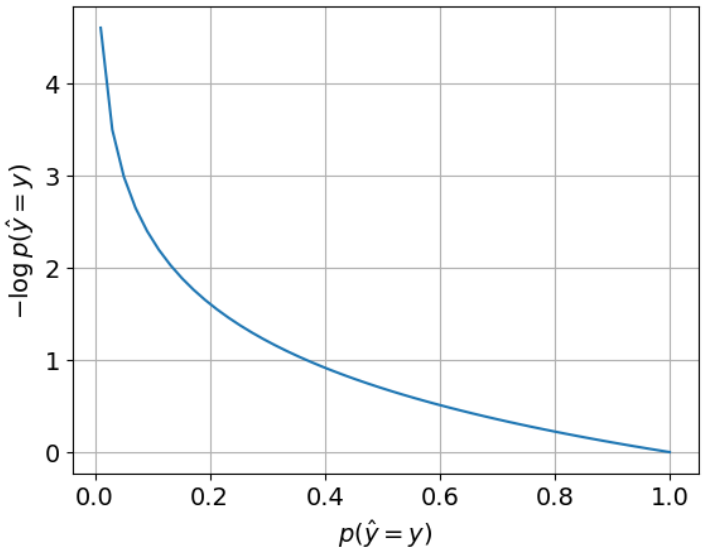
3.2 Logistic loss

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_θ 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:



CHAPTER 4

OPTIMIZATION

In this chapter, we will present variants of the **Gradient Descent** optimization strategy and show how they can be used to optimize neural network parameters.

Let us start with the basic Gradient Descent algorithm and its limitations.

Algorithm 1 (Gradient Descent)

Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $(x_i, y_i) \in \mathcal{D}$
        1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
        2. Compute gradient $\nabla_\theta \mathcal{L}_i$
    2. Compute overall gradient $\nabla_\theta \mathcal{L} = \frac{1}{n} \sum_i \nabla_\theta \mathcal{L}_i$
    3. Update parameters θ based on $\nabla_\theta \mathcal{L}$

The typical update rule for the parameters θ at iteration t is

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \rho \nabla_\theta \mathcal{L}$$

where ρ is an important hyper-parameter of the method, called the learning rate. Basically, gradient descent updates θ in the direction of steepest decrease of the loss $\mathcal{L}$.

As one can see in the previous algorithm, when performing gradient descent, model parameters are updated once per epoch, which means a full pass over the whole dataset is required before the update can occur. When dealing with large datasets, this is a strong limitation, which motivates the use of stochastic variants.

4.1 Stochastic Gradient Descent (SGD)

The idea behind the Stochastic Gradient Descent algorithm is to get cheap estimates for the quantity

$$\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{D}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

where $\mathcal{D}$ is the whole training set. To do so, one draws subsets of data, called *minibatches*, and

$$\nabla_\theta \mathcal{L}(\mathcal{B}; m_\theta) = \frac{1}{b} \sum_{(x_i, y_i) \in \mathcal{B}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

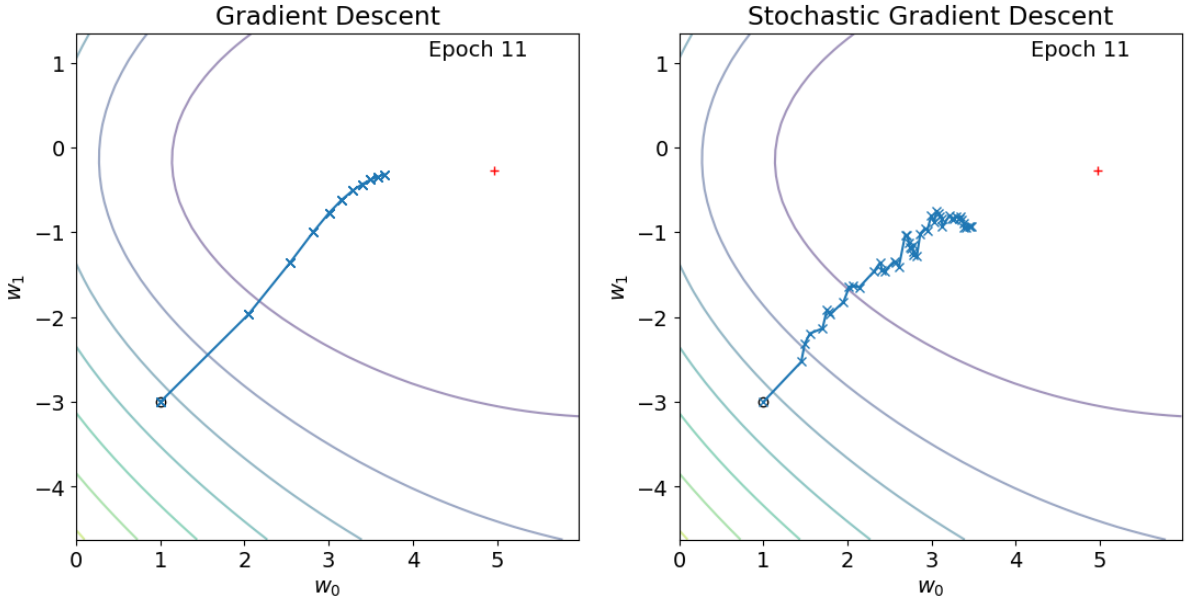
is used as an estimator for $\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta)$. This results in the following algorithm in which, interestingly, parameter updates occur after each minibatch, which is multiple times per epoch.

Algorithm 2 (Stochastic Gradient Descent)

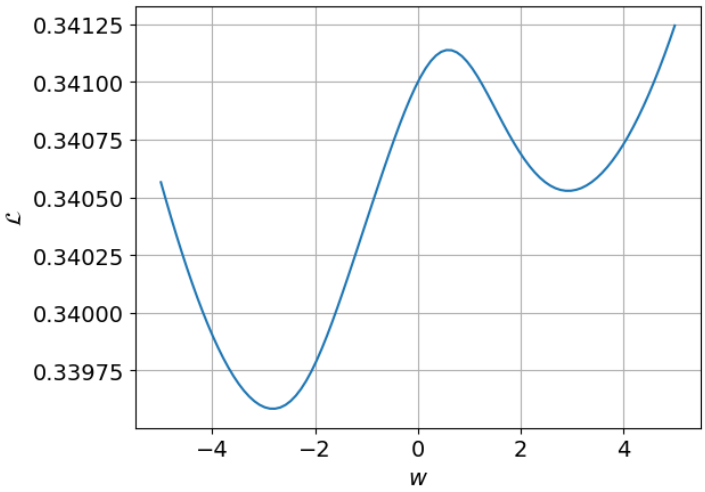
Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $t = 1..n_{\text{minibatches}}$
        1. Draw minibatch $\mathcal{B}$ as a random sample of size b from $\mathcal{D}$
        2. for $(x_i, y_i) \in \mathcal{B}$
            1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
            2. Compute gradient $\nabla_\theta \mathcal{L}_i$
        3. Compute minibatch-level gradient $\nabla_\theta \mathcal{L}_\mathcal{B} = \frac{1}{b} \sum_i \nabla_\theta \mathcal{L}_i$
        4. Update parameters θ based on $\nabla_\theta \mathcal{L}_\mathcal{B}$

As a consequence, when using SGD, parameter updates are more frequent, but they are "noisy" since they are based on an minibatch estimation of the gradient instead of relying on the true gradient, as illustrated below:



Apart from implying more frequent parameter updates, SGD has an extra benefit in terms of optimization, which is key for neural networks. Indeed, as one can see below, contrary to what we had in the Perceptron case, the MSE loss (and the same applies for the logistic loss) is no longer convex in the model parameters as soon as the model has at least one hidden layer:



Gradient Descent is known to suffer from local optima, and such loss landscapes are a serious problem for GD. On the other hand, Stochastic Gradient Descent is likely to benefit from noisy gradient estimations to escape local minima.

4.2 A note on Adam

Adam [Kingma and Ba, 2015] is a variant of the Stochastic Gradient Descent method. It differs in the definition of the steps to be performed at each parameter update.

First, it uses what is called momentum, which basically consists in relying on past gradient updates to smooth out the trajectory in parameter space during optimization. An interactive illustration of momentum can be found in [Goh, 2017].

The resulting plugin replacement for the gradient is:

$$\mathbf{m}^{(t+1)} \leftarrow \frac{1}{1-\beta_1^t}\left[\beta_1 \mathbf{m}^{(t)} + (1-\beta_1)\nabla_\theta \mathcal{L}\right]$$

When β_1 is zero, we have $\mathbf{m}^{(t+1)} = \nabla_\theta \mathcal{L}$ and for $\beta_1 \in]0,1[$, $\mathbf{m}^{(t+1)}$ balances the current gradient estimate with information about past estimates, stored in $\mathbf{m}^{(t)}$.

Another important difference between SGD and the Adam variant consists in using an adaptive learning rate. In other words, instead of using the same learning rate ρ for all model parameters, the learning rate for a given parameter θ_i is defined as:

$$\hat{\rho}^{(t+1)}(\theta_i) = \frac{\rho}{\sqrt{s^{(t+1)}(\theta_i) + \epsilon}}$$

where ϵ is a small constant and

$$s^{(t+1)}(\theta_i) = \frac{1}{1-\beta_2^t}\left[\beta_2 s^{(t)}(\theta_i) + (1-\beta_2)\left(\nabla_{\theta_i}\mathcal{L}\right)^2\right]$$

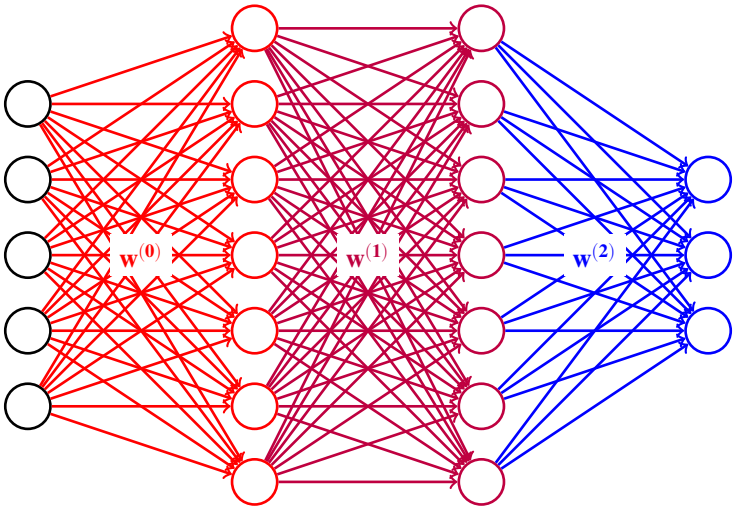
Here also, the s term uses momentum. As a result, the learning rate will be lowered for parameters which have suffered large updates in the past iterations.

Overall, the Adam update rule is:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \hat{\rho}^{(t+1)}(\theta)\mathbf{m}^{(t+1)}$$

4.3 The curse of depth

Let us consider the following neural network:



and let us recall that, at a given layer (ℓ), the layer output is computed as

$$a^{(\ell)} = \varphi(o^{(\ell)}) = \varphi(w^{(\ell-1)}a^{(\ell-1)})$$

where φ is the activation function for the given layer (we ignore the bias terms in this simplified example).

In order to perform (stochastic) gradient descent, gradients of the loss with respect to model parameters need to be computed.

By using the chain rule, these gradients can be expressed as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial w^{(2)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial w^{(1)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(0)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}
\end{aligned}$$

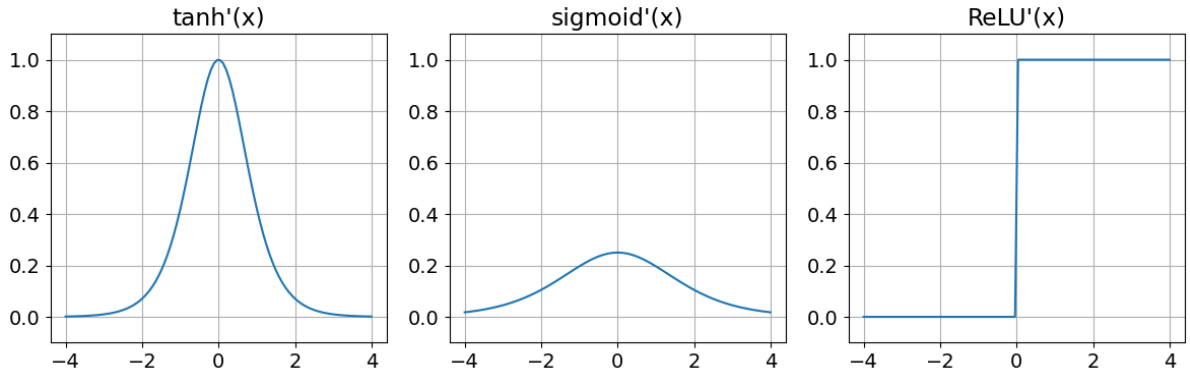
There are important insights to grasp here.

First, one should notice that weights that are further from the output of the model inherit gradient rules made of more terms. As a consequence, when some of these terms get smaller and smaller, there is a higher risk for those weights that their gradients collapse to 0, this is called the **vanishing gradient** effect, which is a very common phenomenon in deep neural networks (*i.e.* those networks made of many layers).

Second, some terms are repeated in these formulas, and in general, terms of the form $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ and $\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}}$ are present in several places. These terms can be further developed as:

$$\begin{aligned}
\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}} &= \varphi'(o^{(\ell)}) \\
\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}} &= w^{(\ell-1)}
\end{aligned}$$

Let us inspect what the derivatives of standard activation functions look like:



One can see that the derivative of ReLU has a wider range of input values for which it is non-zero (typically the whole range of positive input values) than its competitors, which makes it a very attractive candidate activation function for deep neural networks, as we have seen that the $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ term appears repeatedly in chain rule derivations.

4.4 Wrapping things up in `keras`

In `keras`, loss and optimizer information are passed at compile time:

```
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 #
=================================================================
 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)
_________________________________________________________________
```

```
model.compile(loss="categorical_crossentropy", optimizer="adam")
```

In terms of losses:

- `"mse"` is the mean squared error loss,

- `"binary_crossentropy"` is the logistic loss for binary classification,
- `"categorical_crossentropy"` is the logistic loss for multi-class classification.

The optimizers defined in this section are available as `"sgd"` and `"adam"`. In order to get control over optimizer hyper-parameters, one can alternatively use the following syntax:

```
from keras.optimizers import Adam, SGD

# Not a very good idea to tune beta_1
# and beta_2 parameters in Adam
adam_opt = Adam(learning_rate=0.001,
                beta_1=0.9, beta_2=0.9)

# In order to use SGD with a custom learning rate:
# sgd_opt = SGD(learning_rate=0.001)

model.compile(loss="categorical_crossentropy", optimizer=adam_opt)
```

4.5 Data preprocessing

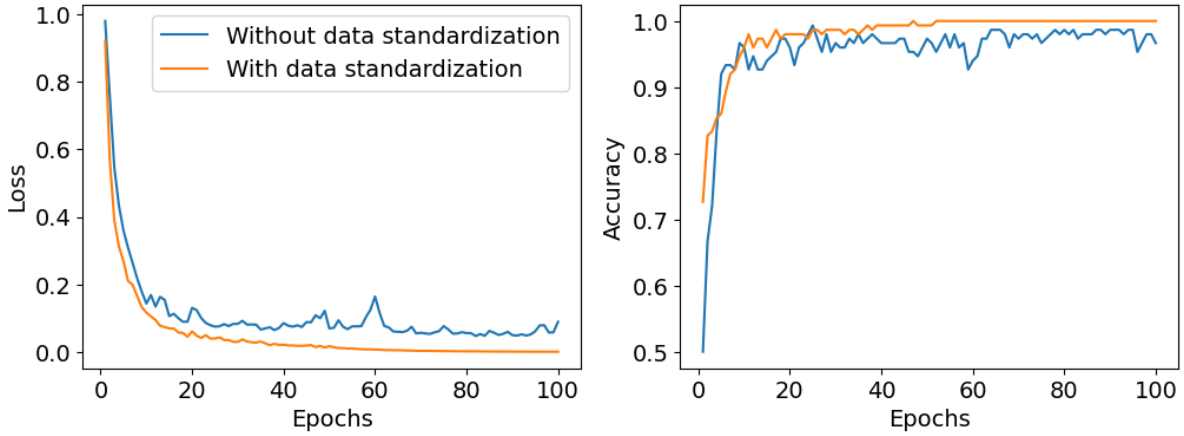
In practice, for the model fitting phase to behave well, it is important to scale the input features. In the following example, we will compare two trainings of the same model, with similar initialization and the only difference between both will be whether input data is center-reduced or left as-is.

```
import pandas as pd
from keras.utils import to_categorical

iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
```

```
from keras.layers import Dense, InputLayer
from keras.models import Sequential
from keras.utils import set_random_seed


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```

Let us now standardize our data and compare performance:

```
X -= X.mean(axis=0)
X /= X.std(axis=0)


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h_standardized = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```



CHAPTER 5

REGULARIZATION

As discussed in previous chapters, one of the strengths of the neural networks is that they can approximate any continuous functions when a sufficient number of parameters is used. When using universal approximators in machine learning settings, an important related risk is that of overfitting the training data. More formally, given a training dataset $\mathcal{D}_t$ drawn from an unknown distribution $\mathcal{D}$, model parameters are optimized so as to minimize the empirical risk:

$$\mathcal{R}_e(\theta) = \frac{1}{|\mathcal{D}_t|} \sum_{(x_i, y_i) \in \mathcal{D}_t} \mathcal{L}(x_i, y_i; m_\theta)$$

whereas the real objective is to minimize the "true" risk:

$$\mathcal{R}(\theta) = \mathbb{E}_{x,y \sim \mathcal{D}} \mathcal{L}(x, y; m_\theta)$$

and both objectives do not have the same minimizer.

To avoid this pitfall, one should use regularization techniques, such as the ones presented in the following.

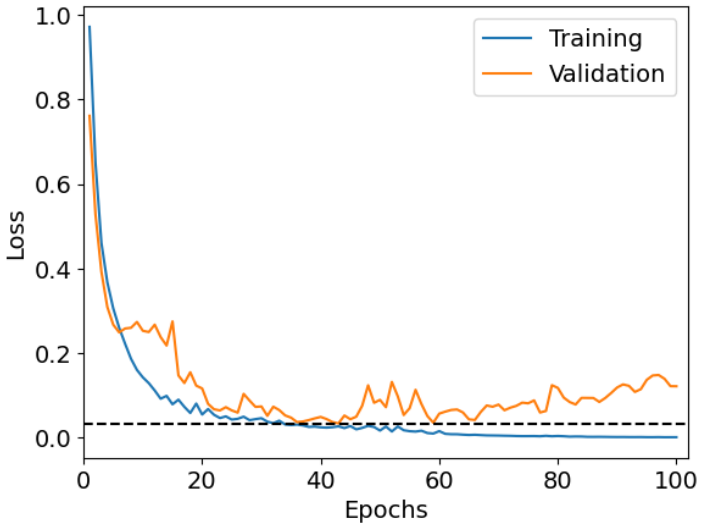
5.1 Early Stopping

As illustrated below, it can be observed that training a neural network for a too large number of epochs can lead to overfitting. Note that here, the true risk is estimated through the use of a validation set that is not seen during training.

```
Using TensorFlow backend
```

```
iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
X -= X.mean(axis=0)
X /= X.std(axis=0)
```

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


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Here, the best model (in terms of generalization capabilities) seems to be the model at epoch 43. In other words, if we had stopped the learning process after epoch 43, we would have gotten a better model than if we use the model trained during 70 epochs.

This is the whole idea behind the "early stopping" strategy, which consists in stopping the learning process as soon as the validation loss stops improving. As can be seen in the visualization above, however, the validation loss tends to oscillate, and one often waits for several epochs before assuming that the loss is unlikely to improve in the future. The number of epochs to wait is called the *patience* parameter.

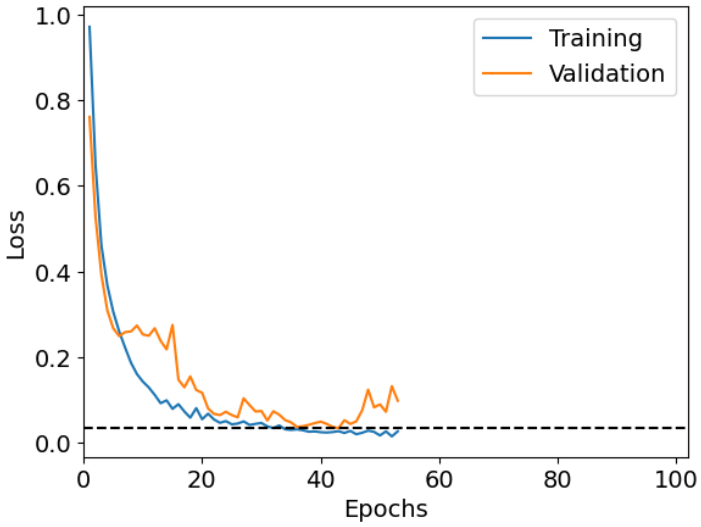
In `keras`, early stopping can be set up via a callback, as in the following example:

```
from keras.callbacks import EarlyStopping


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
```

(continues on next page)

(continued from previous page)

```
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

cb_es = EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True)

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y,
              validation_split=0.3, epochs=n_epochs, batch_size=30,
              verbose=0, callbacks=[cb_es])
```



And now, even is the model was scheduled to be trained for 70 epochs, training is stopped as soon as it reaches 10 consecutive epochs without improving on the validation loss, and the model parameters are restored as the parameters of the model at epoch 43.

5.2 Loss penalization

Another important way to enforce regularization in neural networks is through loss penalization. A typical instance of this regularization strategy is the L2 regularization. If we denote by $\mathcal{L}_r$ the L2-regularized loss, it can be expressed as:

$$\mathcal{L}_r(\mathcal{D}; m_\theta) = \mathcal{L}(\mathcal{D}; m_\theta) + \lambda \sum_\ell \|\theta^{(\ell)}\|_2^2$$

where $\theta^{(\ell)}$ is the weight matrix of layer ℓ.

This regularization tends to shrink large parameter values during the learning process, which is known to help improve generalization.

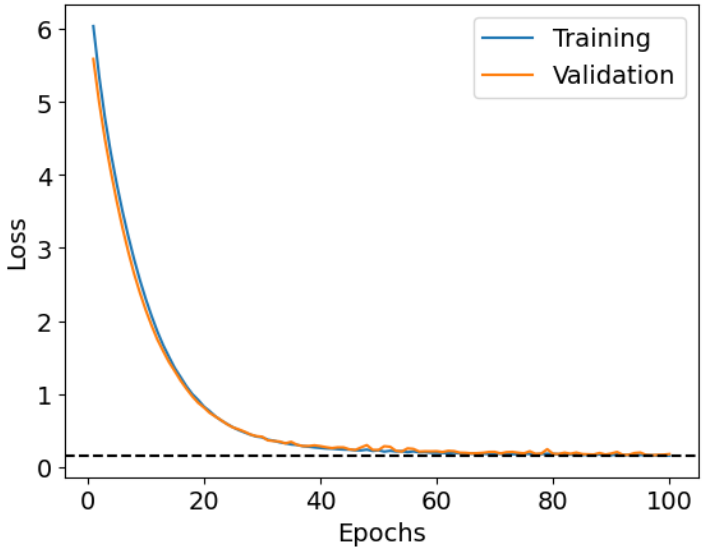
In `keras`, this is implemented as:

```
from keras.regularizers import L2

λ = 0.01
```

(continues on next page)

(continued from previous page)

```
set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



5.3 DropOut

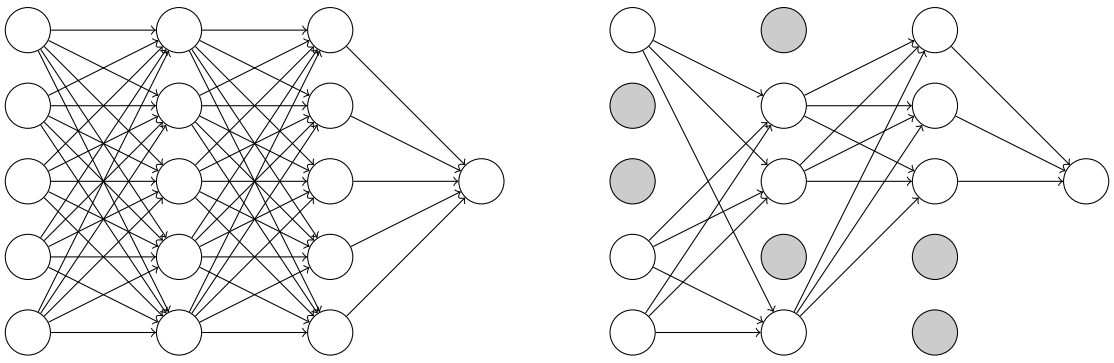


Fig. 5.1: Illustration of the DropOut mechanism. In order to train a given model (left), at each mini-batch, a given proportion of neurons is picked at random to be "switched off" and the subsequent sub-network is used for the current optimization step (*cf.* right-hand side figure, in which 40% of the neurons – coloured in gray – are switched off).

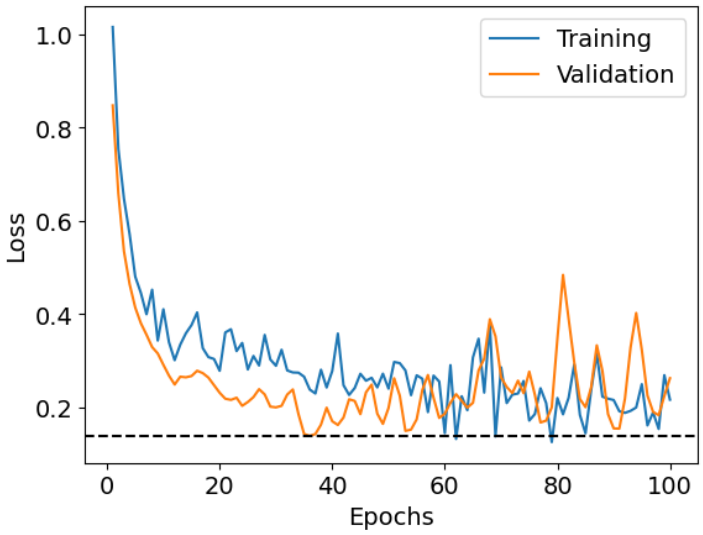
In this section, we present the DropOut strategy, which was introduced in [Srivastava *et al.*, 2014]. The idea behind DropOut is to *switch off* some of the neurons during training. The switched off neurons change at each mini-batch such

that, overall, all neurons are trained during the whole process.

The concept is very similar in spirit to a strategy that is used for training random forest, which consists in randomly selecting candidate variables for each tree split inside a forest, which is known to lead to better generalization performance for random forests. The main difference here is that one can not only switch off *input neurons* but also *hidden-layer ones* during training.

In `keras`, this is implemented as a layer, which acts by switching off neurons from the previous layer in the network:

```
from keras.layers import Dropout

set_random_seed(0)
switchoff_proba = 0.3
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Exercise #1

When observing the loss values in the figure above, can you explain why the validation loss is almost consistently lower than the training one?

Solution

In fact, the training loss is computed as the average loss over all training mini-batches during an epoch. Now, if we recall that during training, at each minibatch, 30% of the neurons are switched-off, one can see that only a subpart of the full

model is used when evaluating the training loss while the full model is retrieved when predicting on the validation set, which explains why the measured validation loss is lower than the training one.

CHAPTER 6

CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (*aka* ConvNets) are designed to take advantage of the structure in the data. In this chapter, we will discuss two flavours of ConvNets: we will start with the monodimensional case and see how ConvNets with 1D convolutions can be helpful to process time series and we will then introduce the 2D case that is especially useful to process image data.

6.1 ConvNets for time series

Convolutional neural networks for time series rely on the 1D convolution operator that, given a time series $\mathbf{x}$ and a filter $\mathbf{f}$, computes an activation map as:

$$(\mathbf{x} * \mathbf{f})(t) = \sum_{k=-L}^{L} f_k x_{t+k} \tag{6.1}$$

where the filter $\mathbf{f}$ is of length $(2L + 1)$.

The following code illustrates this notion using a Gaussian filter:

Convolutional neural networks are made of convolution blocks whose parameters are the coefficients of the filters they embed (hence filters are not fixed *a priori* as in the example above but rather learned). These convolution blocks are translation equivariant, which means that a (temporal) shift in their input results in the same temporal shift in the output:

```
/tmp/ipykernel_11148/368849627.py:32: UserWarning: This figure includes Axes that␣
↪are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
/tmp/ipykernel_11148/368849627.py:23: MatplotlibDeprecationWarning: Auto-removal␣
↪of overlapping axes is deprecated since 3.6 and will be removed two minor␣
↪releases later; explicitly call ax.remove() as needed.
  fig2 = plt.subplot(2, 1, 2)
/tmp/ipykernel_11148/368849627.py:32: UserWarning: The figure layout has changed␣
↪to tight
  plt.tight_layout()
```

```
<IPython.core.display.HTML object>
```

Convolutional models are known to perform very well in computer vision applications, using moderate amounts of parameters compared to fully connected ones (of course, counter-examples exist, and the term "moderate" is especially vague).

Most standard time series architectures that rely on convolutional blocks are straight-forward adaptations of models from the computer vision community ([Le Guennec *et al.*, 2016] relies on an old-fashioned alternance between convolution and pooling layers, while more recent works rely on residual connections and inception modules [Fawaz *et al.*, 2020]). These basic blocks (convolution, pooling, residual layers) are discussed in more details in the next Section.

These time series classification models (and more) are presented and benchmarked in [Fawaz *et al.*, 2019] that we advise the interested reader to refer to for more details.

6.2 Convolutional neural networks for images

We now turn our focus to the 2D case, in which our convolution filters will not slide on a single axis as in the time series case but rather on the two dimensions (width and height) of an image.

6.2.1 Images and convolutions

As seen below, an image is a pixel grid, and each pixel has an intensity value in each of the image channels. Color images are typically made of 3 channels (Red, Green and Blue here).

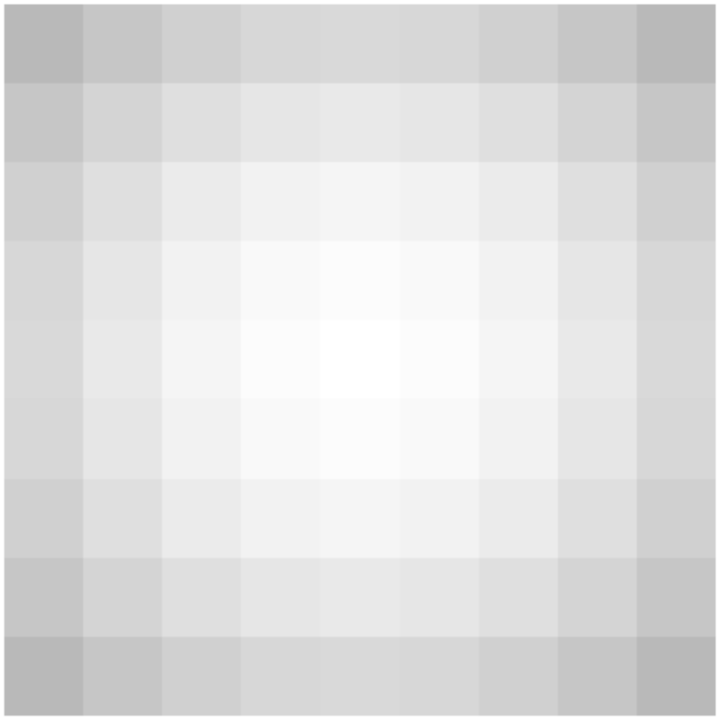
Fig. 6.1: An image and its 3 channels (Red, Green and Blue intensity, from left to right).

The output of a convolution on an image $\mathbf{x}$ is a new image, whose pixel values can be computed as:

$$(\mathbf{x} * \mathbf{f})(i,j) = \sum_{k=-K}^{K} \sum_{l=-L}^{L} \sum_{c=1}^{3} f_{k,l,c} x_{i+k,j+l,c}. \tag{6.2}$$

In other words, the output image pixels are computed as the dot product between a convolution filter (which is a tensor of shape $(2K+1, 2L+1, c)$) and the image patch centered at the given position.

Let us, for example, consider the following 9x9 convolution filter:



Then the output of the convolution of the cat image above with this filter is the following greyscale (*ie.* single channel) image:

One might notice that this image is a blurred version of the original image. This is because we used a Gaussian filter in the process. As for time series, when using convolution operations in neural networks, the contents of the filters will be learnt, rather than set *a priori*.

6.2.2 CNNs *à la* LeNet

In [LeCun *et al.*, 1998], a stack of convolution, pooling and fully connected layers is introduced for an image classification task, more specifically a digit recognition application. The resulting neural network, called LeNet, is depicted below:

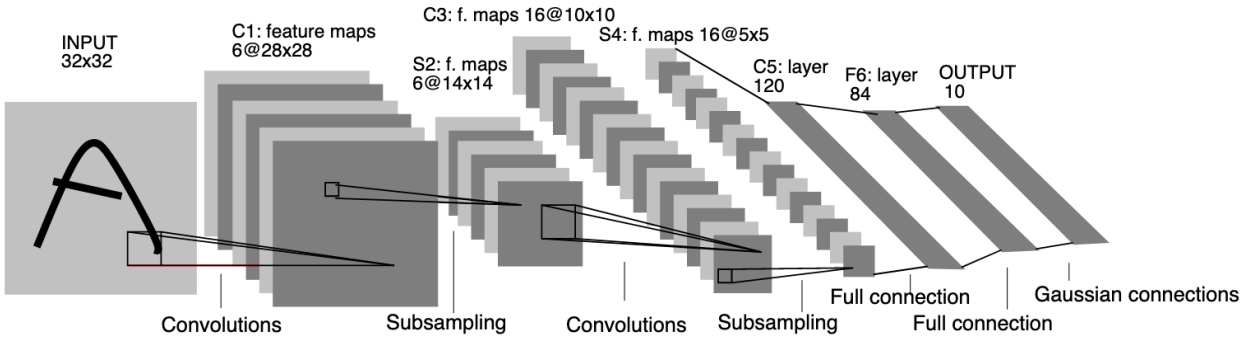


Fig. 6.2: LeNet-5 model

Convolution layers

A convolution layer is made of several convolution filters (also called *kernels*) that operate in parallel on the same input image. Each convolution filter generates an output activation map and all these maps are stacked (in the channel dimension) to form the output of the convolution layer. All filters in a layer share the same width and height. A bias term and an activation function can be used in convolution layers, as in other neural network layers. All in all, the output of a convolution filter is computed as:

$$(\mathbf{x} * \mathbf{f})(i,j,c) = \varphi\left(\sum_{k=-K}^{K}\sum_{l=-L}^{L}\sum_{c'} f^{c}_{k,l,c'} x_{i+k,j+l,c'} + b_c\right) \tag{6.3}$$

where c denotes the output channel (note that each output channel is associated with a filter f^c), b_c is its associated bias term and φ is the activation function to be used.

Tip: In `keras`, such a layer is implemented using the `Conv2D` class:

```
import keras_core as keras
from keras.layers import Conv2D

layer = Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu")
```

Padding

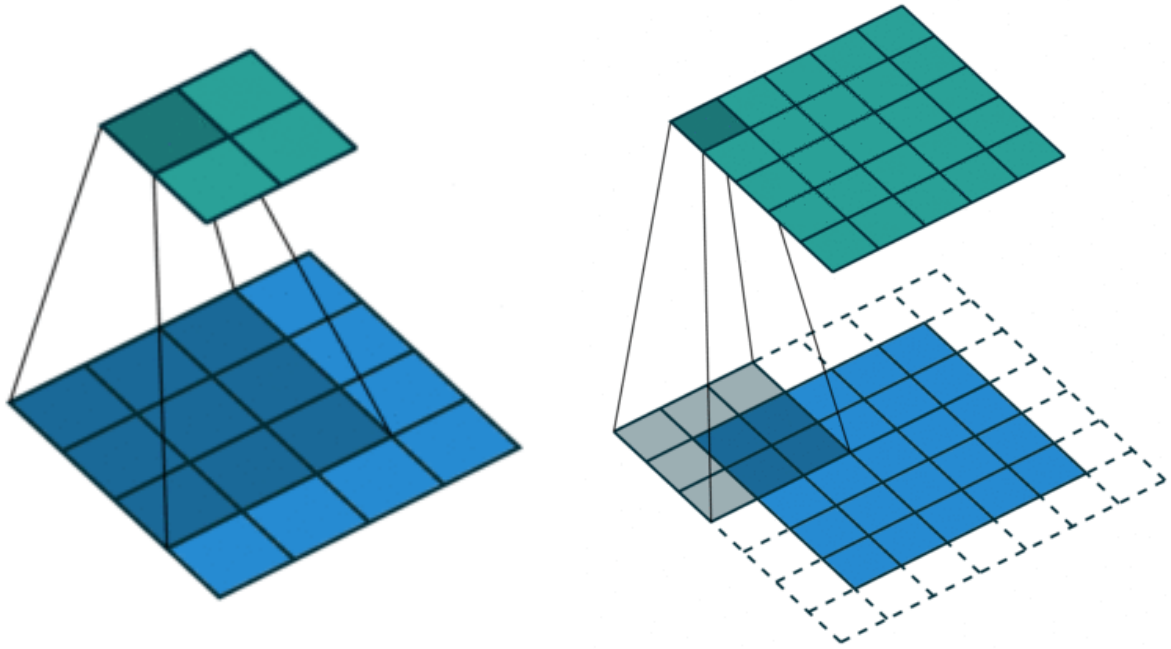


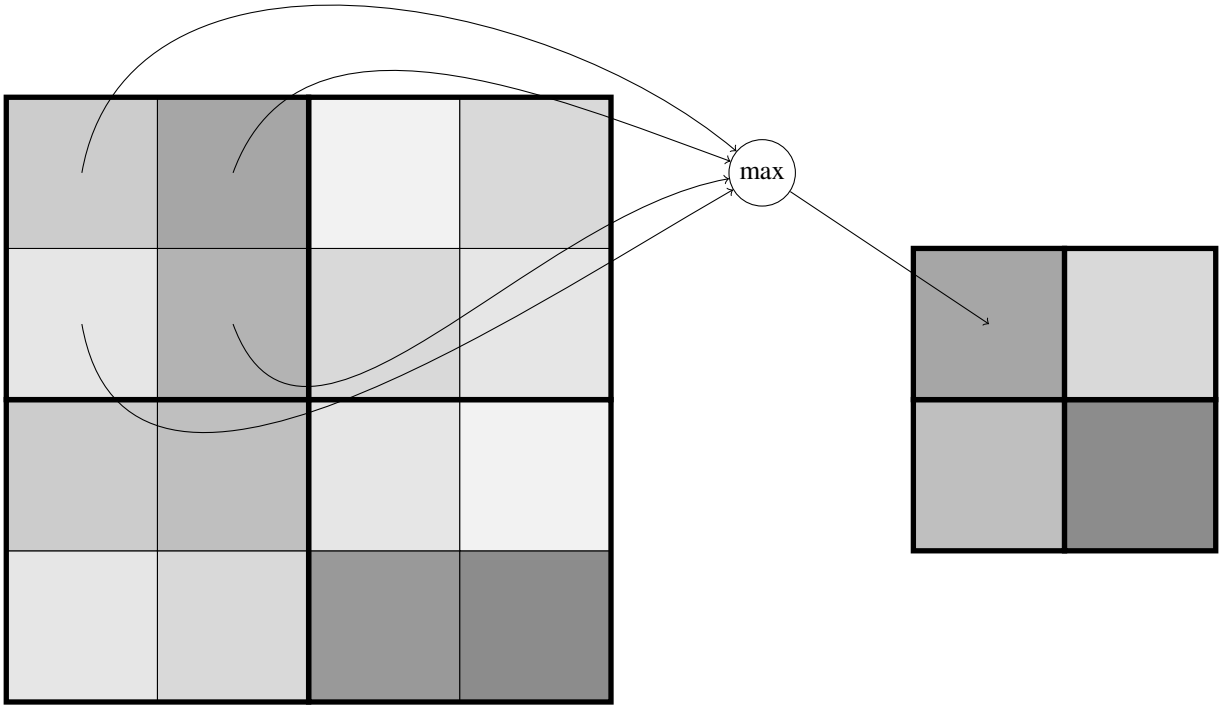
Fig. 6.3: A visual explanation of padding (source: V. Dumoulin, F. Visin - A guide to convolution arithmetic for deep learning). Left: Without padding, right: With padding.

When processing an input image, it can be useful to ensure that the output feature map has the same width and height as the input image. This can be achieved by padding the input image with surrounding zeros, as illustrated in Fig. 6.3 in which the padding area is represented in white.

Pooling layers

Pooling layers perform a subsampling operation that somehow summarizes the information contained in feature maps in lower resolution maps.

The idea is to compute, for each image patch, an output feature that computes an aggregate of the pixels in the patch. Typical aggregation operators are average (in this case the corresponding layer is called an average pooling layer) or maximum (for max pooling layers) operators. In order to reduce the resolution of the output maps, these aggregates are typically computed on sliding windows that do not overlap, as illustrated below, for a max pooling with a pool size of 2x2:



Such layers were widely used in the early years of convolutional models and are now less and less used as the available amount of computational power grows.

Tip: In `keras`, pooling layers are implemented through the `MaxPool2D` and `AvgPool2D` classes:

```
from keras.layers import MaxPool2D, AvgPool2D

max_pooling_layer = MaxPool2D(pool_size=2)
average_pooling_layer = AvgPool2D(pool_size=2)
```

Plugging fully-connected layers at the output

A stack of convolution and pooling layers outputs a structured activation map (that takes the form of 2d grid with an additional channel dimension). When image classification is targeted, the goal is to output the most probable class for the input image, which is usually performed by a classification head that consists in fully-connected layers.

In order for the classification head to be able to process an activation map, information from this map needs to be transformed into a vector. This operation is called flattening in `keras`, and the model corresponding to Fig. 6.2 can be implemented as:

```
from keras.models import Sequential
from keras.layers import InputLayer, Conv2D, MaxPool2D, Flatten, Dense

model = Sequential([
    InputLayer(input_shape=(32, 32, 1)),
    Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu"),
```

(continues on next page)

(continued from previous page)

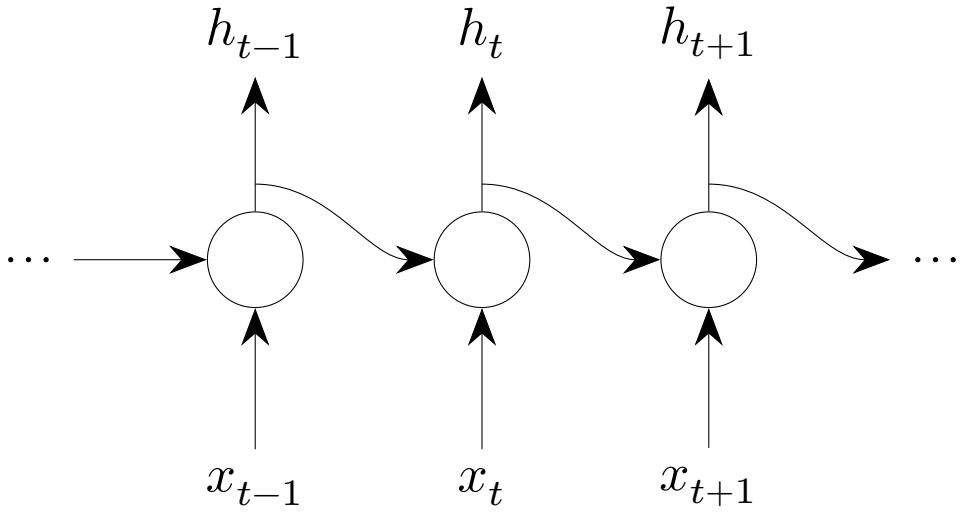
```
    MaxPool2D(pool_size=2),
    Conv2D(filters=16, kernel_size=5, padding="valid", activation="relu"),
    MaxPool2D(pool_size=2),
    Flatten(),
    Dense(120, activation="relu"),
    Dense(84, activation="relu"),
    Dense(10, activation="softmax")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

CHAPTER 7

RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) proceed by processing elements of a time series one at a time. Typically, at time t, a recurrent block will take both the current input x_t and a hidden state h_{t-1} that aims at summarizing the key information from past inputs $\{x_0, ..., x_{t-1}\}$, and will output an updated hidden state h_t:



There exist various recurrent modules that mostly differ in the way h_t is computed.

7.1 "Vanilla" RNNs

The basic formulation for a RNN block is as follows:

$$\forall t, h_t = \tanh(W_h h_{t-1} + W_x x_t + b) \tag{7.1}$$

where W_h is a weight matrix associated to the processing of the previous hidden state, W_x is another weight matrix associated to the processing of the current input and b is a bias term.

Note here that W_h, W_x and b are not indexed by t, which means that they are **shared across all timestamps**.

An important limitation of this formula is that it easily fails at capturing long-term dependencies. To better understand why, one should remind that the parameters of these networks are optimized through stochastic gradient descent algorithms.

To simplify notations, let us consider a simplified case in which h_t and x_t are both scalar values, and let us have a look at what the actual gradient of the output h_t is, with respect to W_h (which is then also a scalar):

$$\nabla_{W_h}(h_t) = \tanh'(o_t) \cdot \frac{\partial o_t}{\partial W_h} \tag{7.2}$$

where $o_t = W_h h_{t-1} + W_x x_t + b$, hence:

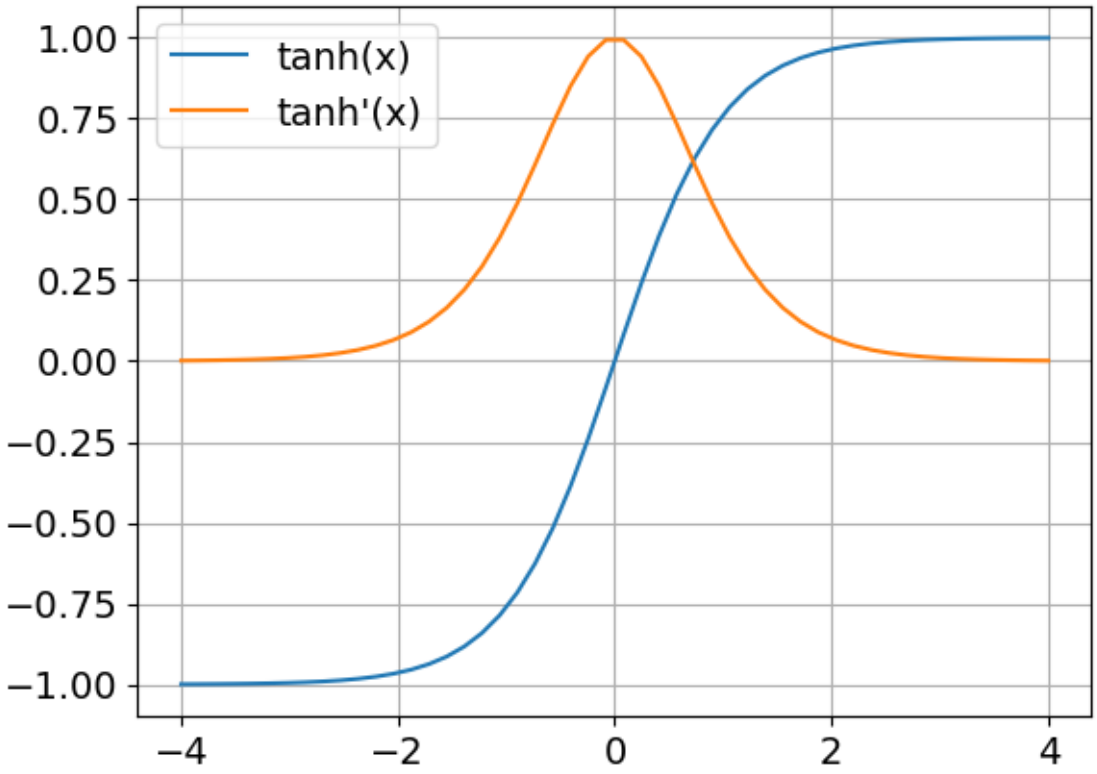
$$\frac{\partial o_t}{\partial W_h} = h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h} \,. \tag{7.3}$$

Here, the form of $\frac{\partial h_{t-1}}{\partial W_h}$ will be similar to that of $\nabla_{W_h}(h_t)$ above, and, in the end, one gets:

$$\begin{aligned}
\nabla_{W_h}(h_t) &= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h} \right] && (7.4)\\
&= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \tanh'(o_{t-1}) \cdot [h_{t-2} + W_h \cdot [...]] \right] && (7.5)\\
&= h_{t-1}\tanh'(o_t) + h_{t-2} W_h \tanh'(o_t)\tanh'(o_{t-1}) + ... && (7.6)\\
&= \sum_{t'=1}^{t-1} h_{t'} \left[W_h^{t-t'-1} \tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t) \right] && (7.7)
\end{aligned}$$

In other words, the influence of $h_{t'}$ will be mitigated by a factor $W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t)$.

Now recall what the tanh function and its derivative look like:



One can see how quickly gradients gets close to 0 for inputs larger (in absolute value) than 2, and having multiple such terms in a computation chain will likely make the corresponding terms vanish.

In other words, the gradient of the hidden state at time t will only be influenced by a few of its predecessors $\{h_{t-1}, h_{t-2}, \dots\}$ and long-term dependencies will be ignored when updating model parameters through gradient descent. This is an occurrence of a more general phenomenon known as the **vanishing gradient** effect.

7.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM, [Hochreiter and Schmidhuber, 1997]) blocks have been designed as an alternative recurrent block that aims at mitigating this vanishing gradient effect through the use of gates that explicitly encode pieces of information that should (resp. should not) be kept in computations.

Gates in neural networks

In the neural networks terminology, a gate $g \in [0,1]^d$ is a vector that is used to filter out information from an incoming feature vector $v \in \mathbb{R}^d$ such that the result of applying the gate is: $g \odot v$ where $\odot$ is the element-wise product. The gate g will hence tend to remove part of the features in v (those corresponding to very low values in g).

In these blocks, an extra state is used, referred to as the cell state C_t. This state is computed as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{7.8}$$

where f_t is the forget gate (which pushes the network to forget about useless parts of the past cell state), i_t is the input gate and $\tilde{C}_t$ is an updated version of the cell state (which, in turn, can be partly censored by the input gate).

Let us delay for now the details about how these 3 terms are computed, and rather focus on how the formula above is significantly different from the update rule of the hidden state in vanilla RNNs. Indeed, in this case, if the network learns

so (through f_t), the full information from the previous cell state C_{t-1} can be recovered, which would allow gradients to flow through time (and not vanish anymore).

Then, the link between the cell and hidden states is:

$$h_t = o_t \odot \tanh(C_t)\,. \tag{7.9}$$

In words, the hidden state is the tanh-transformed version of the cell state, further censored by an output gate o_t.

All gates used in the formulas above are defined similarly:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{7.10}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{7.11}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7.12}$$

where σ is the sigmoid activation function (which has values in $[0, 1]$) and $[h_{t-1}, x_t]$ is the concatenation of h_{t-1} and x_t features.

Finally, the updated cell state $\tilde{C}_t$ is computed as:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)\,. \tag{7.13}$$

Many variants over these LSTM blocks exist in the literature that still rely on the same basic principles.

7.3 Gated Recurrent Unit

A slightly different parametrization of a recurrent block is used in the so-called Gatted Recurrent Unit (GRU, [Cho *et al.*, 2014]).

GRUs also rely on the use of gates to (adaptively) let information flow through time. A first significant difference between GRUs and LSTMs, though, is that GRUs do not resort to the use of a cell state. Instead, the update rule for the hidden state is:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{7.14}$$

where z_t is a gate that balances (per feature) the amount of information that is kept from the previous hidden state with the amount of information that should be updated using the new candidate hidden state $\tilde{h}_t$, computed as:

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b)\,, \tag{7.15}$$

where r_t is an extra gate that can hide part of the previous hidden state.

Formulas for gates z_t and r_t are similar to those provided for f_t, i_t and o_t in the case of LSTMs.

A graphical study of the ability of these variants of recurrent networks to learn long-term dependencies is provided in [Madsen, 2019].

7.4 Conclusion

In this chapter, we have reviewed neural network architectures that are used to learn from time series datasets. Because of time constraints, we have not tackled attention-based models in this course. We have presented convolutional models that aim at extracting discriminative local shapes in the series and recurrent models that rather leverage the notion of sequence. Concerning the latter, variants that aim at facing the vanishing gradient effect have been introduced. Note that recurrent models are known to require more training data than their convolutional counterparts in order to learn meaningful representations.

BIBLIOGRAPHY

[Goh17] Gabriel Goh. Why momentum really works. *Distill*, 2017. URL: http://distill.pub/2017/momentum.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *ICLR*. 2015.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[FFW+19] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[FLF+20] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F Schmidt, Jonathan Weber, Geoffrey I Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. Inceptiontime: finding alexnet for time series classification. *Data Mining and Knowledge Discovery*, 34(6):1936–1962, 2020.

[LGMT16] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. Data Augmentation for Time Series Classification using Convolutional Neural Networks. In *ECML/PKDD Workshop on Advanced Analytics and Learning on Temporal Data*. Riva Del Garda, Italy, September 2016.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[CVMerrienboerBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder-decoder approaches. 2014. arXiv:1409.1259.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Mad19] Andreas Madsen. Visualizing memorization in rnns. *Distill*, 2019. URL: https://distill.pub/2019/memorization-in-rnns.