A picture containing clock, drawing

Description automatically generatedHands-on with Fully Connected Networks

In Chapter 14 we did some amazing things with one neuron, but that is hardly flexible enough to tackle more complex cases. The real power of neural networks comes into light when several (thousands, even millions) neurons interact with each other to solve a specific problem. The network architecture (how neurons are connected to each other, how they behave and so on) plays a crucial role in how efficient the learning of a network is, how good its predictions are and what kind of problems it can solve. There are many kinds of architectures that have been extensively studied and that are very complex, but from a learning perspective it is important to start from the simplest kind of neural network with multiple neurons. It makes sense to start with the so-called **feed-forward neural networks**, where data enters at the input layer and passes through the network, layer by layer, until it arrives at the output layer (this gives the networks their name: feed-forward neural networks).

We will consider in this chapter networks where each neuron in a layer gets its input from all neurons from the preceding layer and feeds their output into each neuron of the next layer. As it is easy to imagine, with more complexity come more challenges, and it is more difficult to get fast learning and good accuracy, since the number of hyperparameters that are available grows due to the increase network complexity, and a simple gradient descent algorithm is not sufficient anymore when dealing with big datasets. When developing models with many neurons we will need to have at our disposal an expanded set of tools that will allow us to deal with all the challenges that those networks present. In this Chapter we will start looking at some more advanced methods and algorithms that will allow us to work efficiently with big datasets and big networks. Those complex networks become good enough to be able to do some interesting multiclass classification, one of the tasks that big networks are most often required to do (see for example handwriting recognition, face recognition, image recognition and so on), so we will use a dataset that will allow us to perform multiclass classification and study its difficulties.

We will start the Chapter with the network architecture and the needed matrix formalism. A short overview of the new hyperparameters that come with this new type of networks will be then given. How to do multiclass classification using the softmax function and what kind of output layer is needed will be then explained. Then, before starting with Python code, we will go into a brief digression to explain in more detail what exactly overfitting is with a simple example, and how to do a basic error analysis with complex networks. Then we will start using **Keras** to construct bigger networks applying them to a MNIST-similar dataset based on images of clothing items (the so-called **Fashion-MNIST dataset**, from Zalando). We will look at how to make the gradient descent algorithm (to have more details about gradient descent, see its implementation from scratch in Chapter 19) faster introducing two new variations: stochastic and mini-batch gradient descent. Then we will look at how to add many layers in an efficient way and how to initialize the weights and the biases in the best way possible to make training fast and stable: we will look at Xavier and He initialization for sigmoid and ReLU activation function, respectively. Finally, a rule of thumb on how to compare complexity of networks going beyond only the number of neurons is described, and the Chapter concludes with some tips on how to choose the right networks.

# A short recall on network’s architecture and matrix notation

The network architecture is actually quite easy to understand. It consists of an **input layer** (the inputs ), several layers (called **hidden** because they are sandwiched between the input and the output layers, so they are "invisible" from the outside so to speak) and then an **output layer**. In each layer you may have one to several neurons. Main property of such a network is that each neuron gets input from each neuron in the preceding layer and feeds its output to every neuron in the next layer. In Figure 15-1 you can see a graphical representation of such a network.

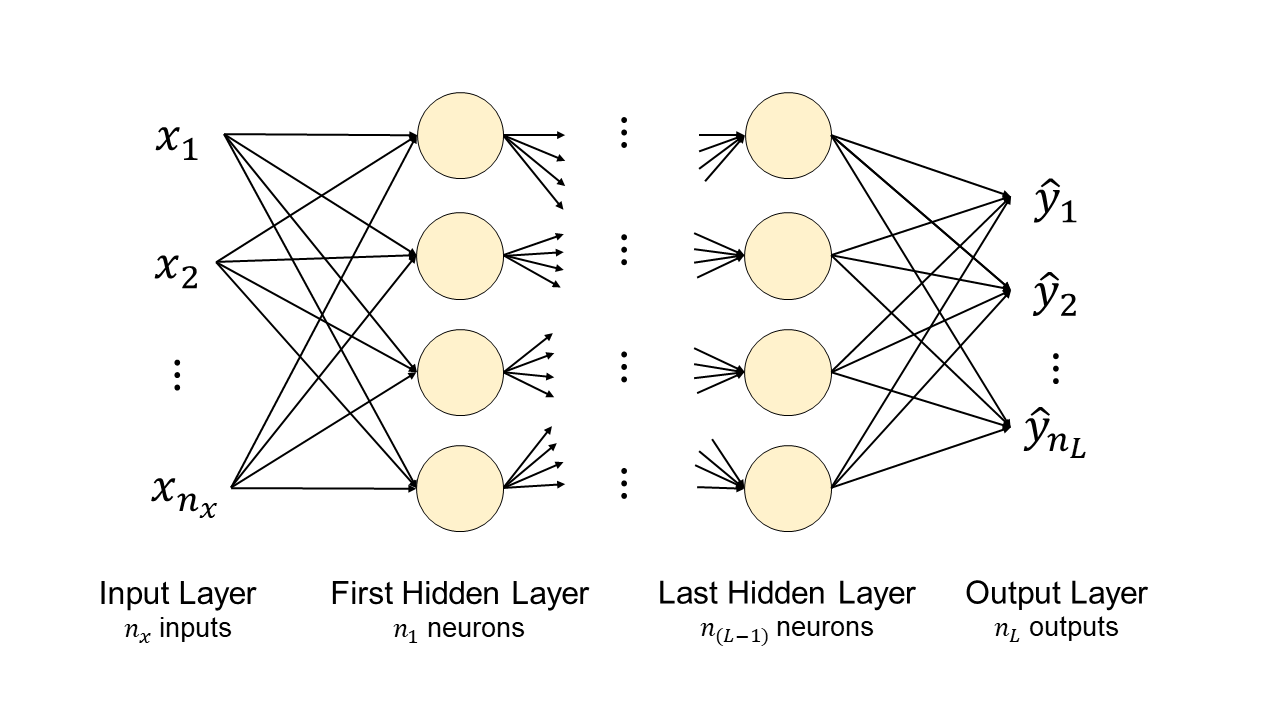


Figure 15-1. The schematic representation of a deep feed-forward neural network with many hidden layers, where each neuron gets input from each neuron in the preceding layer and feeds its output to every neuron in the next layer.

To jump from one neuron in Chapter 14 to this is quite a big step. To build the model, we will need to work with matrix formalism and therefore we need to get all the matrix dimension right. Let us first discuss some new notation:

* - number of hidden layers, excluding the input layer but including the output layer
* - number of neurons in layer

In a network as the one in Figure 15-1 we will indicate the total number of neurons with , that can be written as

Where, by convention we defined . Each connection between two neurons will have its own weight. Let us indicate the weight between neuron in layer and neuron in layer with . In Figure 15-2 we have drawn only the first two layers (input and layer 1) of our generic network of Figure 15-1 with the weights between the first neuron in the input layer and all the others in layer 1. All other neurons are grayed out for clarity.

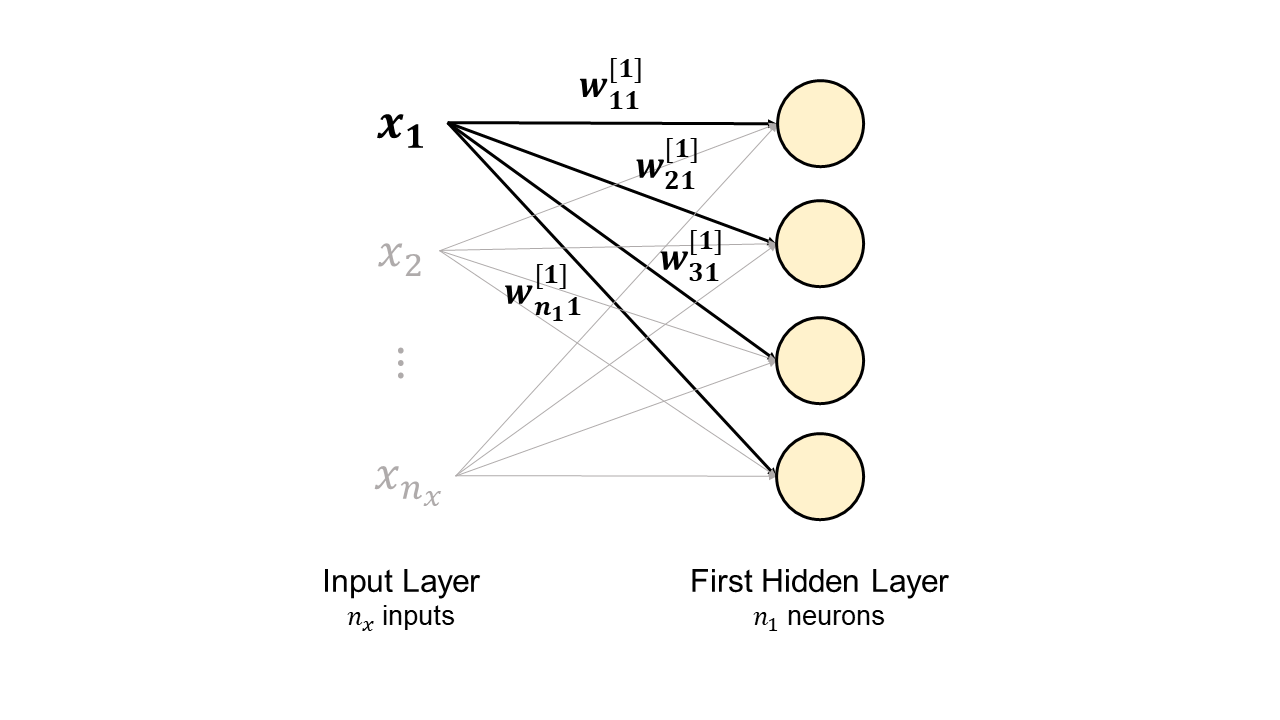


Figure 15-2. The first two layers of a generic neural network with the weights of the connections between the first neuron in the input layers and the others in the second layer. All other neurons and connections are drawn in light gray to make the diagram clearer.

The weights between the input layer and layer 1 can be written as a matrix as

that means that our matrix has dimensions . This of course can be generalized between any two layers and . Meaning that the weight matrix between two adjacent layers and , that we indicate with , will have dimensions . By convention is the number of input features (not the number of observations, that we indicate with ).

**Note** The weight matrix between two adjacent layers and , that we indicate with , will have dimensions , where, by convention, is the number of input features.

The bias (that we have indicated with in Chapter 14) will be a matrix this time. Remember that each neuron that receive inputs will have its own bias, so when considering our two layers and we will need different values of . We will indicate this matrix with and it will have dimensions .

**Note** The bias matrix for two adjacent layers and , that we indicate with , will have dimensions .

## Output of neurons

Now let us start considering the output of our neurons. To begin we will consider the neuron of the first layer (remember our input layer is by definition layer 0). Let us indicate its output with and let us assume that all neurons in layer use the same activation function that we will indicate with . Then we will have

where we have indicated, as you will remember from Chapter 14, as

As you can imagine we want to have a matrix for all the output of layer 1 so we will use the notation

Where will have dimensions , and where with we have indicated our matrix with all our observations (rows for the features and columns for observations). We assume here that all neurons in layer will use the same activation function that we will indicate with .

We can easily generalize the previous equation for a layer

Since layer will get its input from layer , we just need to substitute with . will have dimensions . Our output in matrix form will then be

where the activation function acts, as usual, element by element.

## A short summary of matrix dimensions

Let us summarize the dimensions of all the matrices we have described so far

* have dimensions (where we have by definition)
* have dimensions
* have dimensions
* have dimensions
* have dimensions

where in each case goes from 1 to .

### Example: equations for a network with 3 layers

To make all this discussion a bit more concrete, let us consider an example of a network with 3 layers (so ) with , and as depicted in Figure 15-3.

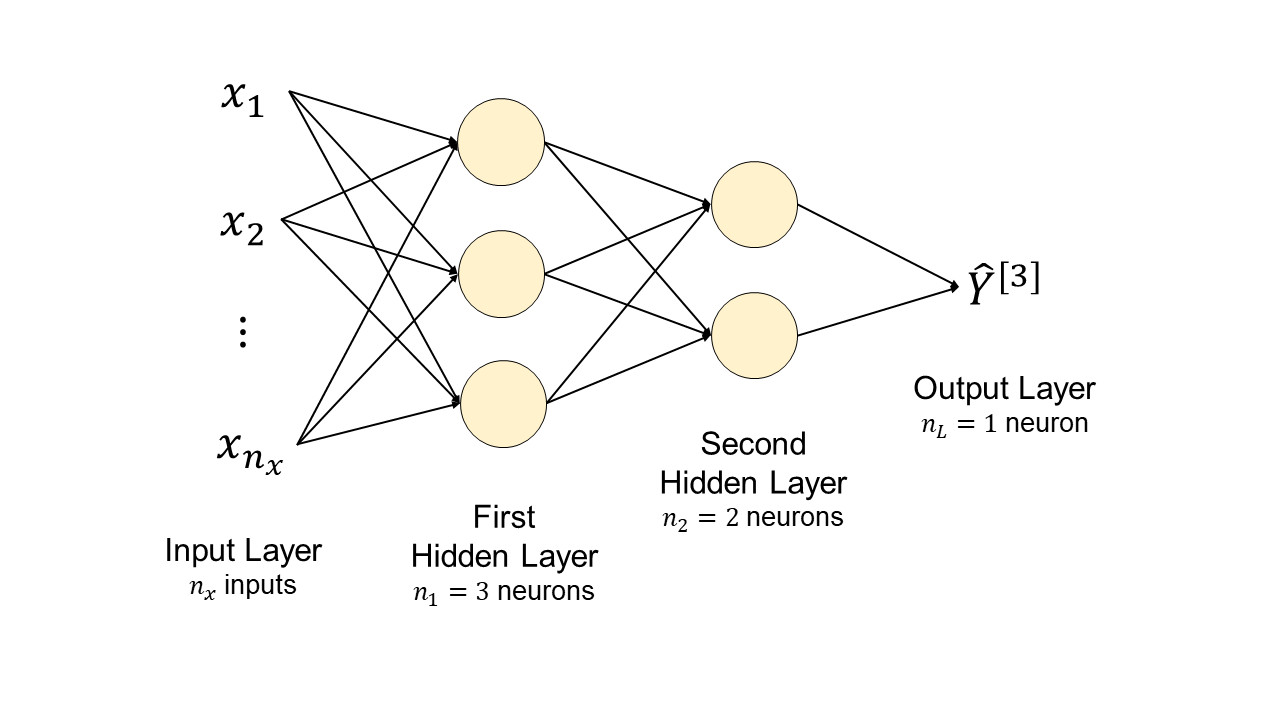


Figure 15-3. A practical example of a feed-forward neural network.

In this case we will need to calculate the following quantities

* where has dimensions , has dimensions and has dimensions
* where has dimensions , has dimensions and has dimensions
* where has dimensions , has dimensions and has dimensions

And your network output will have, as expected dimensions .

All this may seem rather abstract (and in fact it is). You will see later in the chapter how easy is to implement it in Keras simply building the right architecture, based on the steps just discussed.

## Hyperparameters in fully connected networks

In networks as the ones just discussed there are quite a few parameters that you can tune to find the best model for your problem. You will remember from Chapter 14 that parameters that you fix at the beginning and then do not change during the training phase are called hyperparameters (like the number of epochs in the neuron’s examples from Chapter 14). You will need to tune the additional following hyperparameters for feed-forward networks

* number of layers
* number of neurons in each layer for from 1 to L
* choice of activation function for each layer

Then of course you still have the following hyperparameters that you already met in Chapter 14

* number of iterations (or epochs)
* learning rate

## A short recall on softmax activation function for multiclass classification

You will still need to suffer a bit more theory before getting to some TensorFlow code. Those kinds of networks start to be complex enough to be able to perform some multiclass classification with reasonable results. To do this we must first introduce the softmax function.

Mathematically speaking the softmax function is a function that transform a dimensional vector into another dimensional vector of real values each between 0 and 1, and that sums up to 1. Given real values for we define the vector and we define the softmax vector function as

Since the denominator is always bigger than the nominator . Additionally, we have

So behaves like a probability since its sum over is 1 and its elements are all less than 1. We will look at as a probability distribution over possible outcomes. For us will simply be the probability of our input observation of being of class . Let us suppose we are trying to classify an observation into 3 classes, we may get the following output: , and . That means that our observation has a 10% probability of being of class 1, 60% probability of being of class 2 and 30% probability of being of class 3. Normally one chooses to classify the input observation into the class with the higher probability, in this example class 2 with 60% of probability.

**Note** We will look at with as a probability distribution over possible outcomes. For us will simply be the probability of our input observation of being of class .

To be able to use the softmax function for classification we will need to use a specific output layer. We will need to use ten neurons (in the case of a 10 classes multiclass classification problem, like the one we will see later in the Chapter) where each will give as its output and then one neuron that will output . This neuron will have the softmax function as activation function and will have as inputs the 10 outputs of the last layer with 10 neurons. In Keras you use the tf.keras.activations.softmax function applied to the last layer with 10 neurons. Remember that this Keras function will act element by element. We will see later in the Chapter a concrete example from start to end on how to implement it practically.

# A brief digression: overfitting

One of the most common problem that you will encounter when training deep neural networks will be overfitting. What it can happen is that your network may, due to its flexibility, learn patterns that are due to noise, errors or simply wrong data. It is very important to understand what overfitting is, so now we will go through a practical example of what can happen, to get an intuitive understanding of it. To make it easier to visualize, we will work with a simple 2-dimensional dataset that we will create for the purpose.

## A practical example of overfitting

Networks described in the previous sections are rather complex and can easily lead to overfitting of the dataset. Let us briefly explain the concept of overfitting. To understand it let us consider the following problem: find the best polynomial that approximate a given data set. Given a set of 2-dimensional points we want to find the best polynomial of degree in the form

That minimize the mean square error

where as usual indicates the number of data points we have. We do not only want to determine all the parameters but also the value of that best approximate our data. in this case measures our model complexity. For example, for we simply have (a constant), the simplest polynomial we can think of. For higher we have higher order polynomials, meaning our function is more complex, having more parameter available for training. Here is an example of our function for

Where we have 4 parameters that can be tuned during our model’s training. Let us generate some data starting from a second order polynomial ()

adding some random error (this will make overfitting visible). Let us first import our standard libraries with the addition of the curve\_fit function that will automatically minimize the standard error and find the best parameters. Do not worry too much about this function, the goal here is to show you what can happen when you use a model that is too complex.

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

Let us define a function for a second-degree polynomial

def func\_2(p, a, b, c):

return a + b\*p + c\*p\*\*2

then let us generate our dataset

x = np.arange(-5.0, 5.0, 0.05, dtype = np.float64)

y = func\_2(x, 1, 2, 3) + 18.0 \* np.random.normal(0, 1, size = len(x))

To add some random noise to the function we have used the function np.random.normal(0, 1, size = len(x)) that generates a NumPy array of random values from a normal distribution of length len(x) with average 0 and standard deviation 1.

In Figure 15-4 you can see how the data looks like for , and .

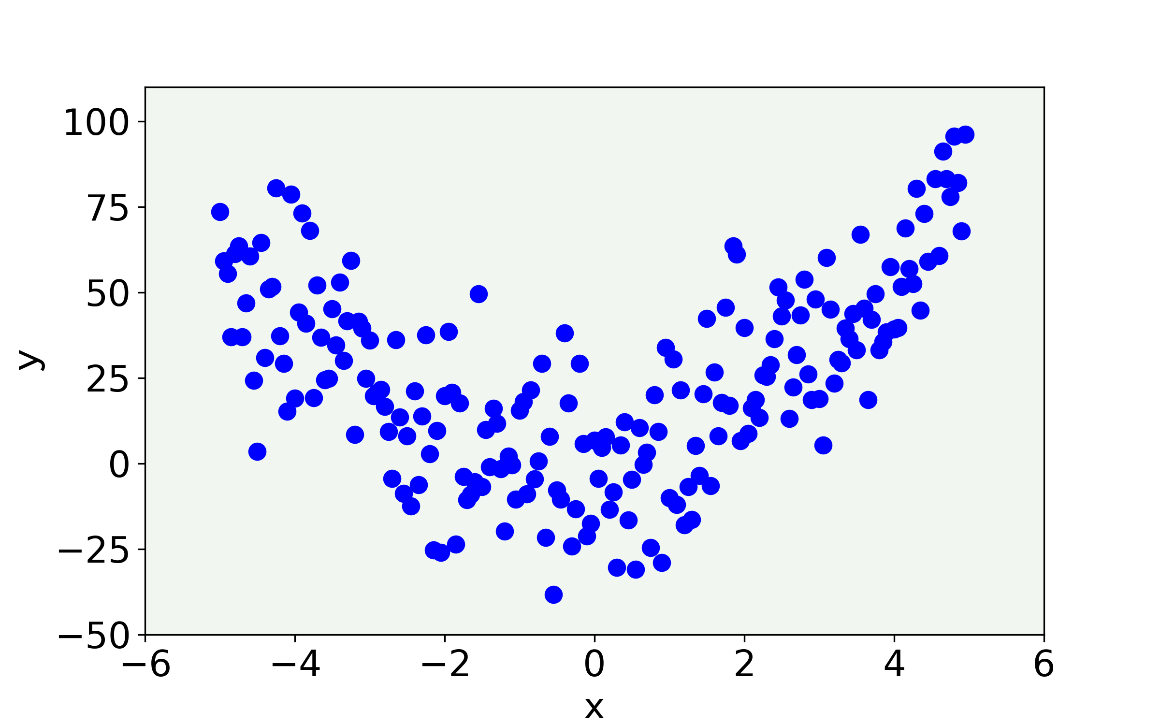


Figure 15-4. The data we have generated with , and as described in the text.

Now let us consider a model that is too simple to capture the feature of the data, meaning we will see what a model with high bias[[1]](#footnote-1) can do. Let us consider a linear model (). The code will be

def func\_1(p, a, b):

return a + b\*p

popt, pcov = curve\_fit(func\_1, x, y)

That will give the best values for and that minimize the standard error. In Figure 15-5 is clear how this model completely misses the main feature of the data, being too simple.

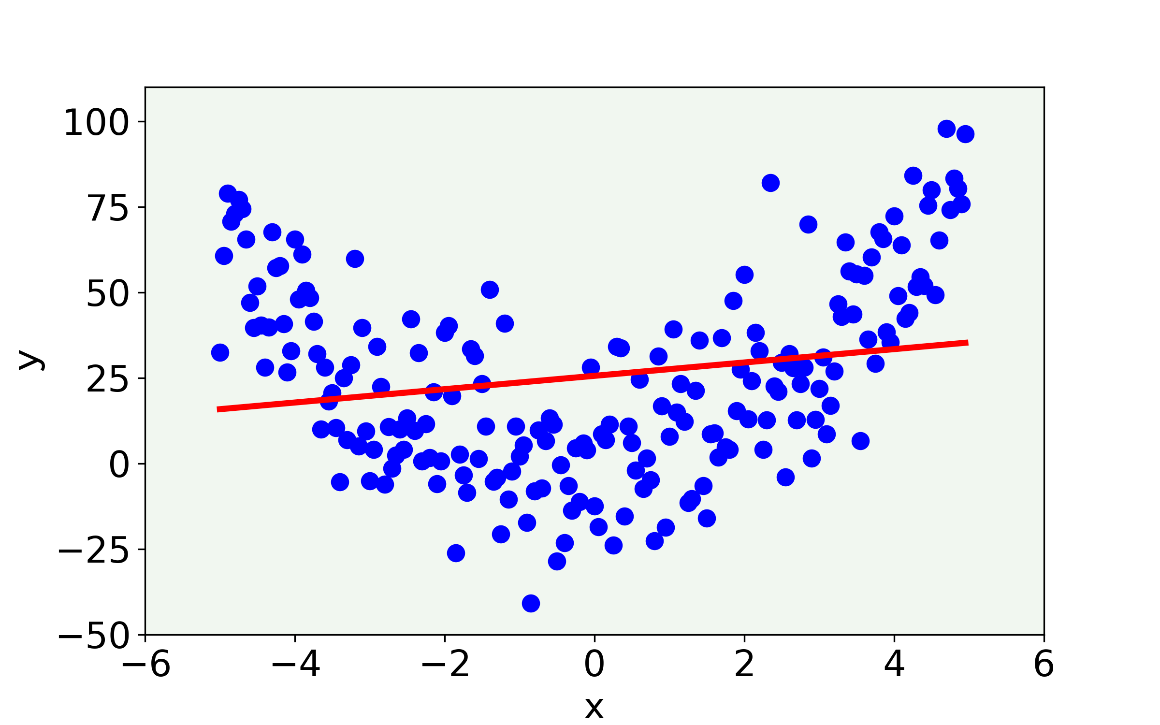


Figure 15-5. The linear model misses the main feature of the data being too simple. In this case the model has high bias.

Let us try to fit a 2-degree polynomial (). The results are shown in Figure 15-6.

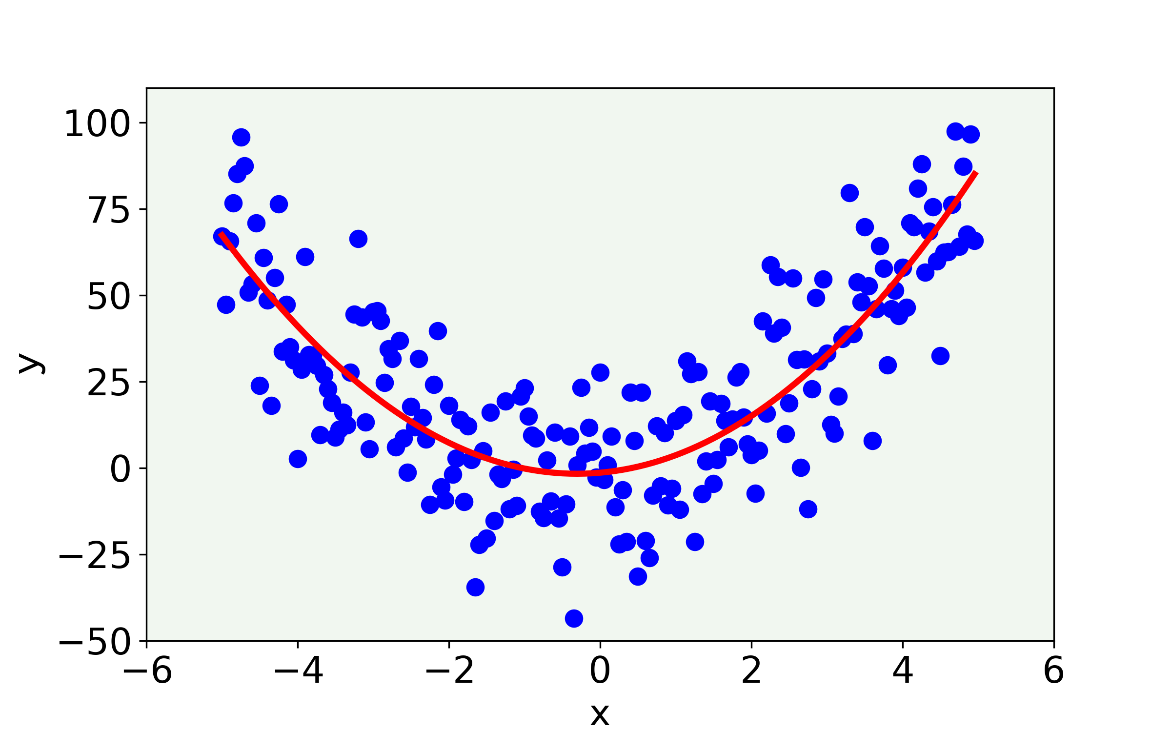


Figure 15-6. The result (red line) for a 2-degree polynomial.

That is better. This model seems to capture the main features of the data, ignoring the random noise. Now let us try a very complex model, a 21-degree polynomial (). Results are shown in Figure 15-7.

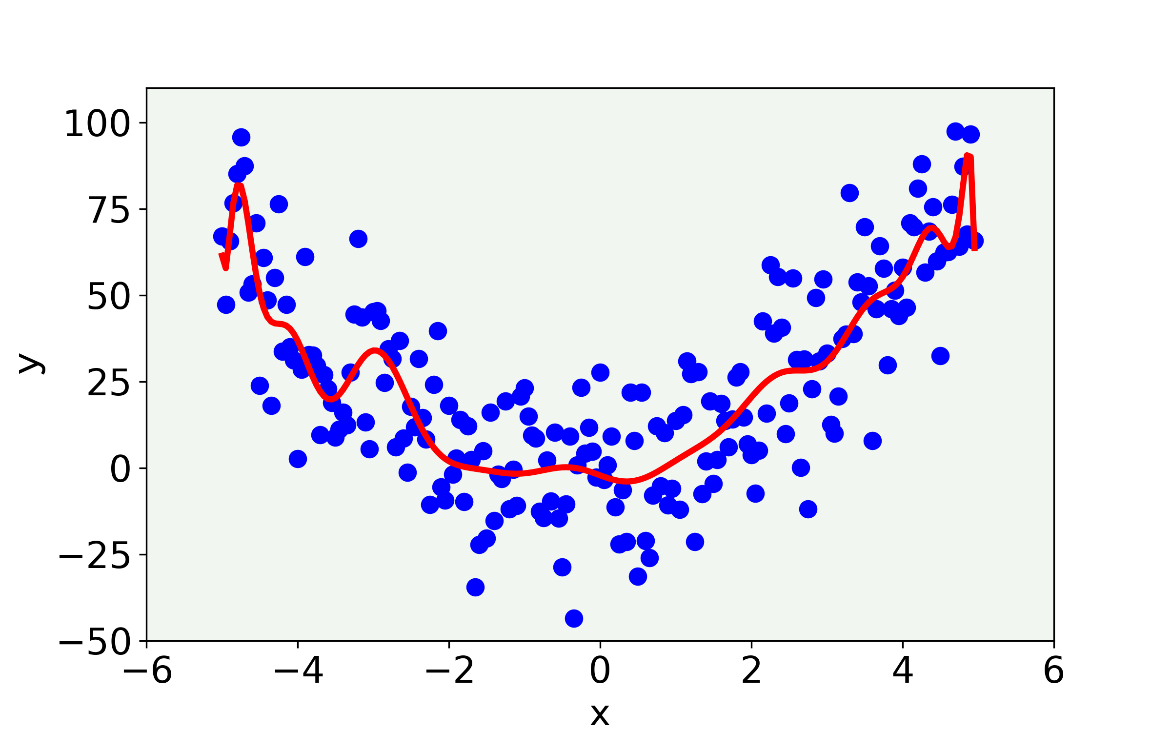


Figure 15-7. The result for a 21-degree polynomial model.

Now this model shows features that we know are wrong (since we created our data). Those features are not present, but the model is so flexible that captures the random variability that we have introduced with noise. The oscillations that have appeared using this high ordered polynomial are wrong and do not describe the data correctly.

In this case we talk about **overfitting**, meaning we start capturing with our model features that are due, for example, to random error. It is easy to understand that this generalizes quite badly. If we would apply this 21-degree polynomial model to new data it would not work well, since the random noise would be different in new data and so the oscillations (the ones represented in Figure 15-7) would make no sense on new data.

In Figure 15-8, the best 21-degree polynomial models obtained by fitting data generated with 10 different random noise values added are shown. You can clearly see how much it varies. It is not stable and is strongly dependent from the random noise present. The oscillations are always different! In this case we talk about high variance.

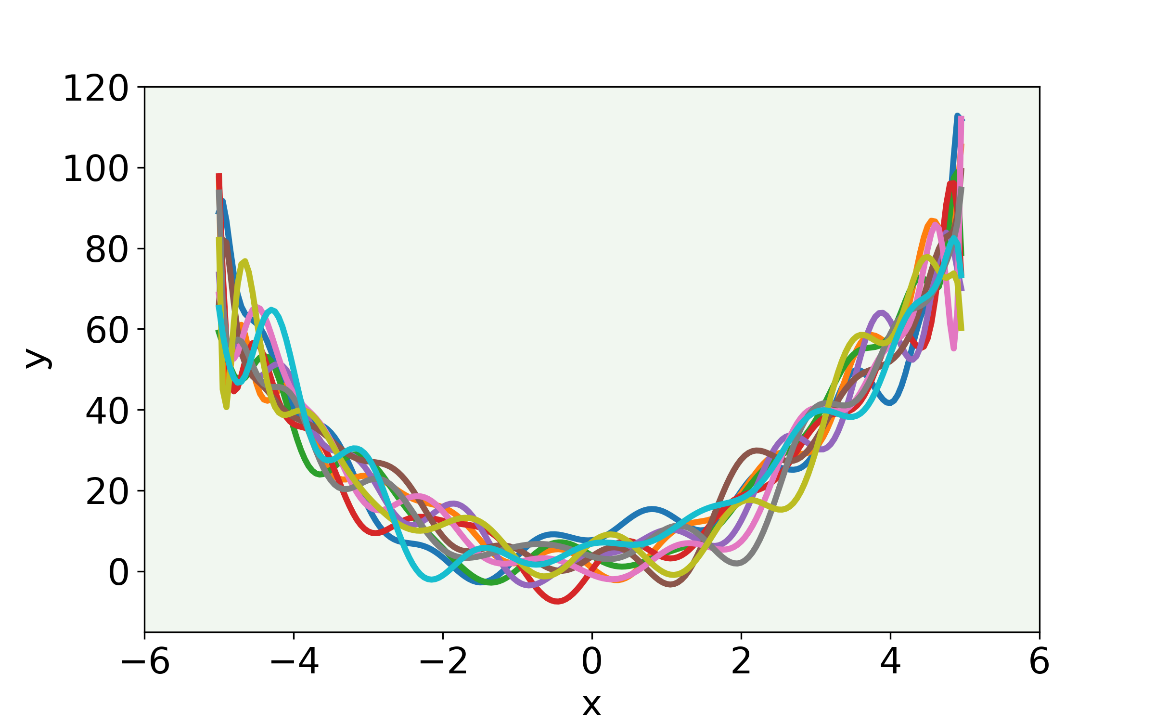


Figure 15-8. The result of our model with a 21-degree polynomial fitted to 10 different datasets generated with different random noise values added.

Now let us do the same plot with our linear model while varying our random noise as we did in Figure 15-8. You can check the results in Figure 15-9.

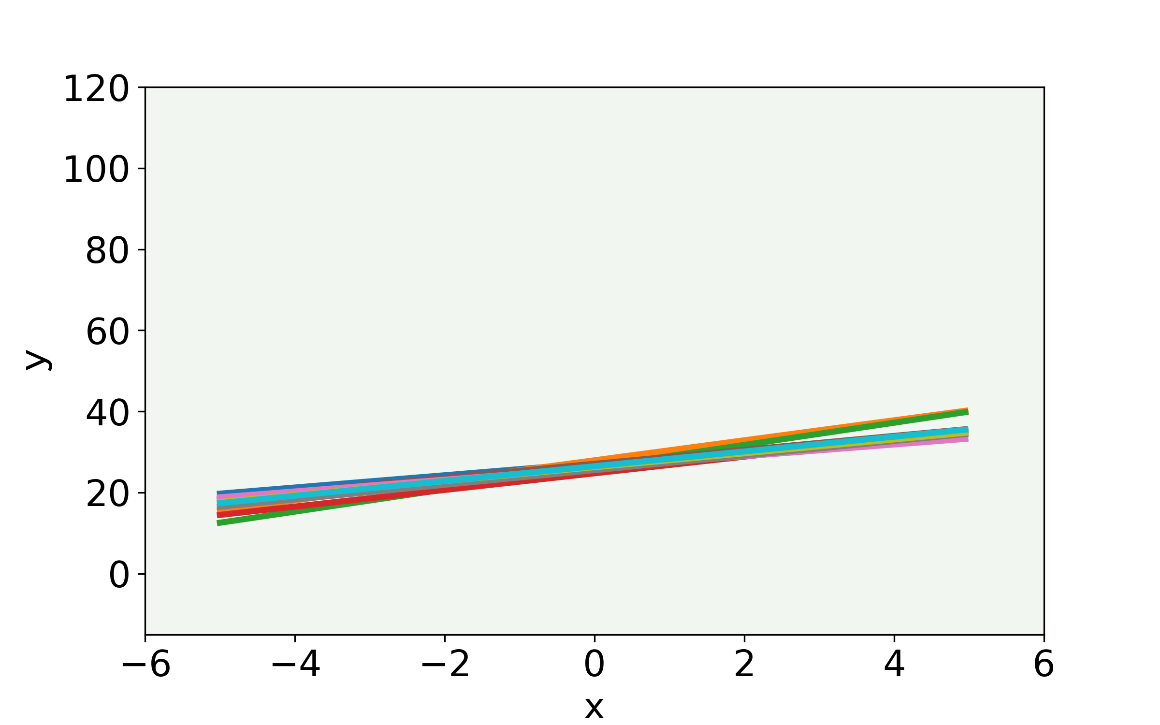


Figure 15-9. The result of our linear model applied to data where we have randomly changed the random noise. For easier comparison with Figure 15-8 we have used the same scale.

You can see that our model is much more stable. Our linear model does not capture any feature that is dependent from our noise, but it misses the main features of our data (the concave nature). We talk here of high bias.

Figure 15-10 should help you in getting an intuitive understanding of bias and variance.

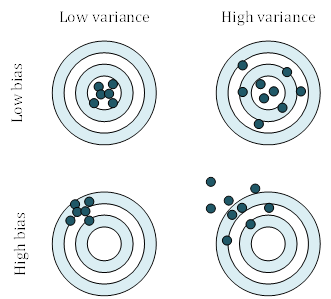


Figure 15-10. The following diagram should give you an intuitive understanding of bias and variance. Bias is a measure of how close our measurements are to the true values (the center in the figure) and variance is a measure of how spread the measurements are around the average (not necessarily the true value as you can see on the left).

In the case of neural networks, we have many hyperparameters (number of layers, number of neurons in each layer, activation function, and so on) and it is very difficult to know in which regime we are. How can we tell if our model has a high variance or a high bias for example? An entire chapter will be dedicated on this subject, but the first step to be able to perform this error analysis is to split our dataset in two different ones. Let us see what it means and why we do it.

**Note** The essence of **overfitting** is to have unknowingly extracted some of the residual variation (i.e. the noise) as if that variation represented underlying model structure [1]. The opposite is called **underfitting** when the model cannot capture the structure of the data.

The problem with overfitting and deep neural networks is that there is no way of visualizing easily the results and therefore we need a different approach to determine if our model is overfitting, underfitting or is just right. This can be achieved splitting our dataset in different parts and evaluating and comparing your metrics on all of them. Let us see the basic idea in the next section.

## Basic error analysis

To check how our model is doing and to perform a proper error analysis we need to split our dataset in two parts[[2]](#footnote-2):

* **Training dataset**: the model is trained on this dataset using the inputs and the relative labels and using an optimizer algorithm like gradient descent. Often this set is indicated as the "train set”.
* **Development (or validation) set**: the trained model will then be used on this dataset to check how it is doing. On this dataset we will test different hyperparameters. For example, we can train two different models with a different number of layers on the training dataset and test them on this dataset to check how they are doing. Often this set is indicated with the "dev set".

There will be an entire chapter dedicated to error analysis, but it is a good idea to give you an overview on why it is important to split the dataset. Let us suppose we are dealing with classification and let us suppose that our metric to judge how good is our model is 1 minus the accuracy, or in other words the percentage of the cases that are wrongly classified.

Table 15-1. Some examples on the difference between models with high bias and models with high variance.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Error** | **Case A** | **Case B** | **Case C** | **Case D** |
| Train set error | 1% | 15% | 14% | 0.3% |
| Dev set error | 11% | 16% | 32% | 1.1% |

Let us consider the three cases reported in Table 15-1:

* **Case A**: here we are overfitting (high variance), because we are doing very well on the training set, but our model generalizes very badly to our dev set (check again Figure 15-8).
* **Case B**: here we see a problem with high bias, meaning that our model is not doing very well generally, on both datasets (check again Figure 15-9)
* **Case C**: here we have a high bias (the model cannot predict very well the training set) and high variance (the model does not generalize well on the dev set).
* **Case D**: here everything seems ok. Good error on the train set and good on the dev set. That is a good candidate for our best model.

We will explain much better all those concepts later in the book where we will give recipes on how to solve problems of high bias, high variances, both, or even more complex cases.

To recap, to do a very basic error analysis you will need to split your dataset in at least two sets: train and dev. You should then calculate your metric on both sets and compare them. You want to have a model that has low error on the train set, low error on the dev set (as in Case D in the example above) and the two values should be comparable.

**Note** Your main take away from this section should be: **1)** we need a set of recipes and guidelines for understanding how our model is doing: is it overfitting, underfitting or is it just right? **2)** to answer question 1) we need to split our dataset in two parts, to be able to perform this analysis (later in the book we will also see what we can do with the dataset splitted in three or even four parts).

# How to implement a feed-forward neural network in Keras

Building a feed-forward neural network in Keras is straightforward and is simply a generalization of the one neuron model we built in Chapter 14. Let us compare the two cases. Below you have the schematic one neuron model Keras implementation

model = keras.Sequential([

layers.Dense(1, input\_shape = [...])

])

And below a feed-forward network model with one hidden layer made of 15 neurons (the first model we will use for our multiclass classification task on the Zalando dataset)

model = keras.Sequential([

layers.Dense(15, input\_shape = [...])

layers.Dense(10)

])

As you can see, we only added more neurons (15) in the hidden layer (that in the one neuron model was already the output one) and we added the output layer, made of 10 neurons, since we have 10 classes. As you can notice, you can easily create very complex models by simply adding to the stack one layer after another.

In the next paragraphs you will see a practical example on how you use this model, choosing the right activation function and the right loss function (given as additional parameters), to solve a multiclass classification task.

# Multiclass classification with feed-forward neural networks

The task we are going to solve together is a multiclass classification problem on the Zalando dataset. It consists in predicting the corresponding label among 10 possible cases (10 different types of clothing). To solve it, we will use a feed-forward network architecture and we will try different configurations (different optimizers and architectures), performing some error analysis to see which situation is better.

Let us start having a look at the data.

## The Zalando dataset for our real case example

Zalando SE is a German commerce company based in Berlin. The company maintains a cross-platform store that sells shoes, clothing, and other fashion items [2]. For a Kaggle competition (if you do not know what Kaggle is check their website [3], where you can participate in many competitions where the goal is to solve problems with data science) they prepared a MNIST-similar dataset of Zalando's clothing article images [4], where they provided 60000 training images and 10000 test images. As in MNIST, each image is 28x28 pixels in grayscale. They classified all images in 10 different classes and provided the labels for each image. The dataset has 785 columns, where the first column is the class label (an integer going from 0 to 9) and the remaining 784 contain the pixel gray value of the image (you can calculate that 28x28=784).

Each training and test example is assigned to one of the following labels (as from the documentation):

* 0 - T-shirt/top
* 1 - Trouser
* 2 - Pullover
* 3 - Dress
* 4 - Coat
* 5 - Sandal
* 6 - Shirt
* 7 - Sneaker
* 8 - Bag
* 9 - Ankle boot

In Figure 15-11 you can see an example of each class chosen randomly from the dataset.

Immagine che contiene testo

Descrizione generata automaticamente

Figure 15-11. One example from each of the 10 classes in the Zalando dataset.

The dataset has been provided under the MIT License[[3]](#footnote-3). The datafile can be downloaded from Kaggle (<https://www.kaggle.com/zalando-research/fashionmnist/data>) or directly from GitHub (<https://github.com/zalandoresearch/fashion-mnist>). If you choose the second option, you will need to prepare the data a bit (you can convert it to CSV with the script located at <https://pjreddie.com/projects/mnist-in-csv/>). If you download it from Kaggle you have all the data already in the right format. You will find two csv files zipped on the Kaggle website. When unzipped you will have: fashion-mnist\_train.csv with 60000 images (roughly 130 Mb) and fashion-mnist\_test.csv with 10000 (roughly 21 Mb).

In our following example, we will retrieve the dataset in a third way: from TensorFlow datasets catalog (<https://www.tensorflow.org/datasets/catalog/fashion_mnist>), since in this way we will not have to perform any preprocessing step and we will automatically import the data inside our notebook. Now, let us code!

We will need the following imports in our code

# general libraries

import pandas as pd

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import matplotlib.font\_manager as fm

from random import \*

import time

# tensorflow libraries

from tensorflow.keras.datasets import fashion\_mnist

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import tensorflow\_docs as tfdocs

import tensorflow\_docs.modeling

Then, to retrieve the dataset we can simply run the following command

((trainX, trainY), (testX, testY)) = fashion\_mnist.load\_data()

Incredibly easy! Now we have two numpy matrices (trainX and testX) containing all the pixels values describing each of the training and test images and two numpy arrays (trainY and testY) containing the associated labels.

Let us print the datasets dimensions

print('Dimensions of the training dataset: ', trainX.shape)

print('Dimensions of the test dataset: ', testX.shape)

print('Dimensions of the training labels: ', trainY.shape)

print('Dimensions of the test labels: ', testY.shape)

which will return as output

Dimensions of the training dataset: (60000, 28, 28)

Dimensions of the test dataset: (10000, 28, 28)

Dimensions of the training labels: (60000,)

Dimensions of the test labels: (10000,)

**Tip** Remember you should not focus on the Python implementation. Focus on the model, on the concepts behind the implementation. You can achieve the same results using pandas, numpy or even C. Try to concentrate on how to prepare the data, how to normalize it, how to check the training and so on.

As you can see, we have a training dataset made of 60000 items, stored as images of 28x28 pixels each, and a test dataset made of 10000 items, stored in the same way. Then, to each dataset is associated an array of corresponding labels.

Now we need to modify our data to obtain for each image a “flatten” version of it, meaning an array of 754 pixels, instead of a matrix of 28x28 pixels. This step is necessary, since as we saw when we recalled the feed-forward network architecture, it receives as inputs all the features as separate values. Therefore, we need to have all the pixels stored in the same array. On the contrary, Convolutional Neural Networks (CNNs) do not work with flattened versions of the images, but we will have an entire Chapter later dedicated to this topic. For now, keep this in mind.

The following lines reshape the matrices dimensions

data\_train = trainX.reshape(60000, 784)

data\_test = testX.reshape(10000, 784)

Let us summarize our data so far

* labels - has dimensions (60000,) and contains the class labels (integers from 0 to 9)
* train - has dimensions (60000x784) and contains the features, where each column contains the grayscale value of a single pixel in the image (remember 28x28=784)

Check again Figure 15-11 to get an idea of how the images look like. Finally let us normalize the input so that instead of having values from 0 to 255 (the grayscale values), it has only values between 0 and 1. This is very easy to do with the code

data\_train\_norm = np.array(data\_train/255.0)

data\_test\_norm = np.array(data\_test/255.0)

Before moving to developing our network, we need to solve another problem first. Labels must be provided in a different form, when performing a multiclass classification task. Let us discuss it in the next paragraph.

## Modifying labels for the softmax function - one hot encoding

You will remember from Chapter 14 that in classification we will use the following cost function

where contains our labels and is the result of our network. So, the two elements must have the same dimensions. In our case here we saw that our network will give as output a vector with 10 elements, while a label in our dataset is simply a scalar. So, we have that has dimensions (10,1) and that has dimensions (1,1). This will not work if we do not do something smart. We need to transform our labels in a vector that have dimensions (10,1). A vector with a value for each class, but what value should we use?

What we need to do is what is called **one-hot encoding[[4]](#footnote-4)**. Meaning we will transform our labels (integers from 0 to 9) to vectors with dimensions (1,10) with this algorithm: our one-hot encoded vector will have all zeros, except at the index of the label. For example, for a label 2 our 1x10 vector will have all zeros except at position with index 2, or in other words will be (0,0,1,0,0,0,0,0,0,0). Let make some other examples and it will become immediately clear (see Table 15-2).

Table 15-2. Examples on how one-hot encoding works. Remember labels goes from 0 to 9, as indexes.

|  |  |
| --- | --- |
| **Label** | **One-hot encoded label** |
| 0 | (1,0,0,0,0,0,0,0,0,0) |
| 2 | (0,0,1,0,0,0,0,0,0,0) |
| 5 | (0,0,0,0,0,1,0,0,0,0) |
| 7 | (0,0,0,0,0,0,0,1,0,0) |

In Figure 15-12 you can see a graphical representation of the process of one-hot encoding a label.

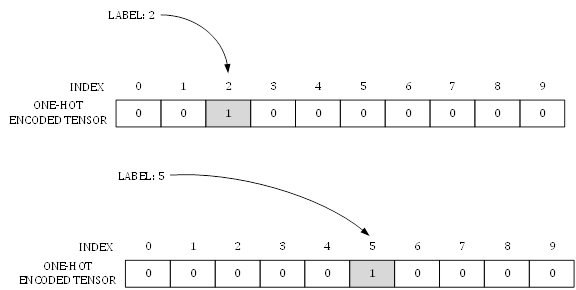


Figure 15-12. A graphical representation of the process of one-hot encoding a label. In the Figure, two labels (2 and 5) are one-hot encoded in two vectors. The grayed element of the vector is the one the becomes one, while the white marked ones remain zero.

Sklearn has several ways of doing this automatically (check for example the function OneHotEncoder()). But I think is instructive to do it manually to really see how is done. Once you understand why you need it, and in which format you need it, you can use the function you like most. The Python code to do this is very simple:

labels\_train = np.zeros((60000, 10))

labels\_train[np.arange(60000), trainY] = 1

labels\_test = np.zeros((10000, 10))

labels\_test[np.arange(10000), testY] = 1

First you create a new array with the right dimensions: (60000,10), and you fill it with zeros with the numpy function np.zeros((60000,10)). Then you set to 1 only the columns related to the label itself using pandas functionalities to slice dataframes with the line labels\_train[np.arange(60000), trainY] = 1 (the same of course is also performed in the case of the test dataset). At the end we obtain the following dimensions: (60000,10), where each row indicates a different observation.

Now in our code we can compare finally and , since both have now the dimensions (10,1) for one observation, or when considering the entire test data set of (10000,10). The same can of course be asserted for the training dataset. Each column now in will represent the probability of our observation of being of a specific class. At the very end when calculating the accuracy of our model, we will assign the class with the highest probability to each observation.

**Note** Our network will give us the 10 probabilities for the observation of being of each of the 10 classes. At the end we will assign to the observation the class that has the highest probability.

## The feed-forward network model

We will start with a network with just one hidden layer. We will have an input layer with 784 features, then a hidden layer (where we will vary the number of neurons), then an output layer of ten neurons that will feed their output into a neuron that will have as activation function the softmax function. Check first Figure 15-13 for a graphical representation of the network and then we will spend some time explaining the various parts, especially the output layers.

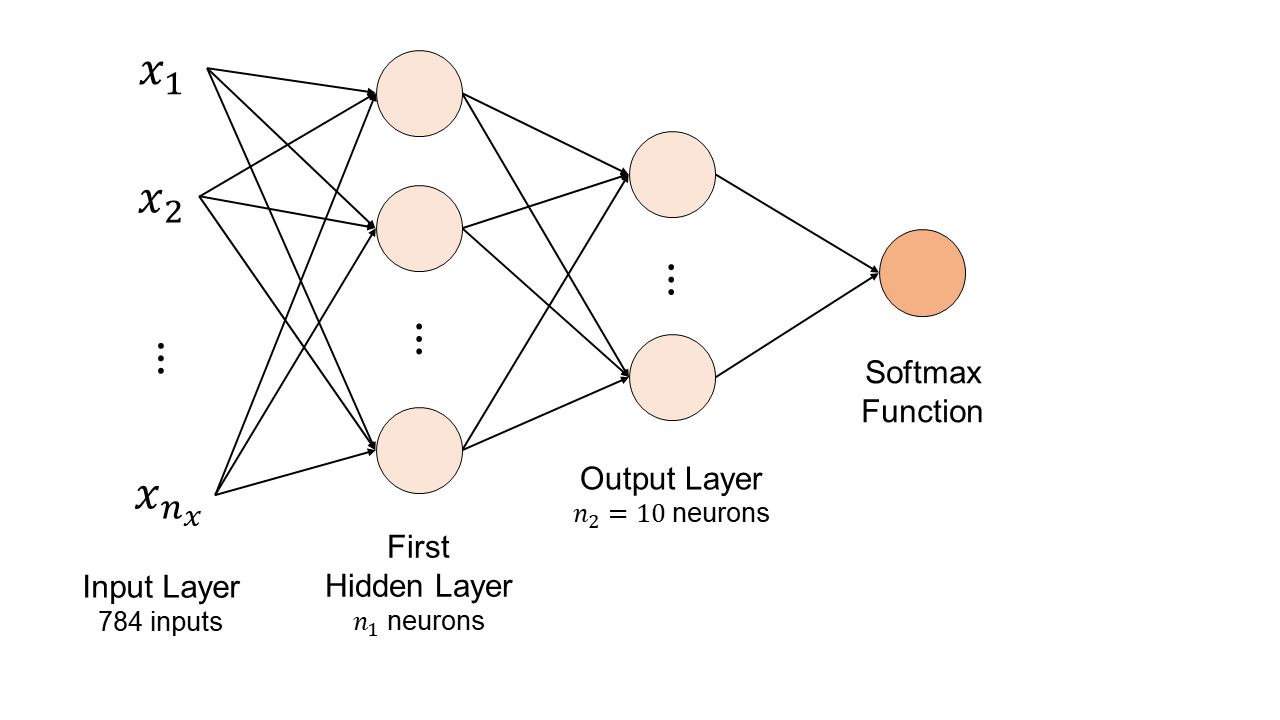


Figure 15-13. The network architecture with a single hidden layer. We will vary during our analysis the number of neurons in the hidden layer.

Let us explain why this strange output layer with ten neurons and why the need of an additional neuron for the softmax function. Remember, for each image we want to be able to tell which class it belongs. To do this, as explained when discussing the softmax function, we will need to get for each observation ten outputs: each being the probability of the image of being of each of the classes. So given an input we will need the ten values: , , …, (probability of the observation class being one of the ten possibilities given the input ) or in other words our output should be a vector of dimensions 1x10 in the form

And since the observation must be of one single class the condition

must be satisfied. This can be understood as: the observation has a 100% probability of being of one of the 10 classes. Or in other words all the probabilities must add to 1. We solve this problem in two steps:

* We create an output layer with ten neurons, in this way we will have our ten values as output
* Then we feed the ten values to a new neuron (let us call it "softmax" neuron) that will take the ten inputs and give as output ten values that are all less than one, and that adds to 1.

In Figure 15-14 you can see our "softmax" neuron in detail.

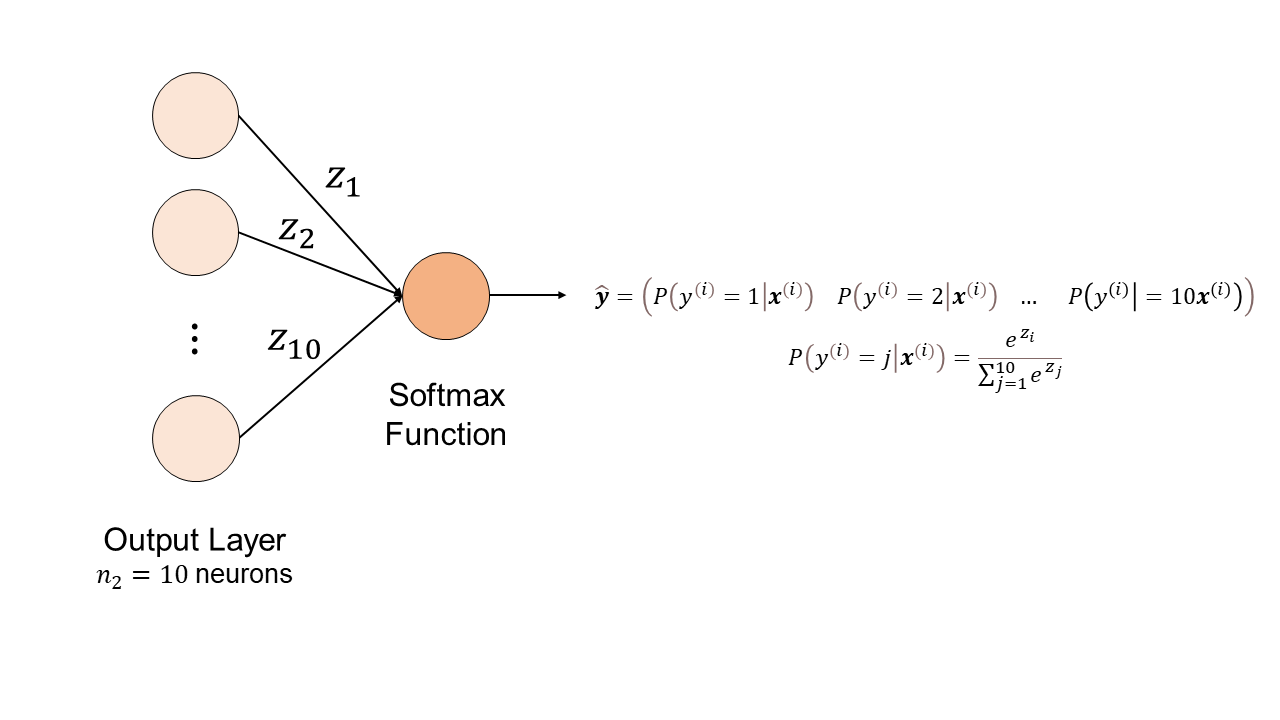


Figure 15-14. The final neuron in our network that transform the ten inputs in proabilities.

Calling the output of the neuron in the last layer (with going from 1 to 10) we will have

In Keras, this is straightforward. But it is instructive to know exactly what each line of code does. That is what the Keras function model.add(Dense(10, activation = 'softmax')) does. Take a vector as input and returns a vector with the same dimensions as the input but "normalized" as discussed above. In other words, if we feed to the function, it will return a vector with the same dimensions as , meaning 1x10, where each element added to the others gives 1.

### Keras implementation

Now is time to build our model with Keras. The following code will do the job

def build\_model(opt):

# create model

model = Sequential()

# add first hidden layer and set input dimensions

model.add(Dense(15, input\_dim = 784, activation = 'relu'))

# add output layer

model.add(Dense(10, activation = 'softmax'))

# compile model

model.compile(loss = 'categorical\_crossentropy',

optimizer = opt,

metrics = ['categorical\_accuracy'])

return model

Now we will not go through each line of code, since you should understand now how a basic Keras model is built (remember our simple one neuron model of Chapter 14). But there are a few details of the code that need to be stressed:

* Our last layer will use the softmax function: model.add(Dense(10, activation = 'softmax'))
* The two parameters 15 () and 10 ( define the number of neurons in the different layers. Remember the second (output) layer must have 10 neurons to be able to use the softmax function. But we will play with the value for . Increasing will increase the complexity of the network.
* We set the categorical cross-entropy (loss = 'categorical\_crossentropy') as loss function and the categorical accuracy (metrics = ['categorical\_accuracy']) as metrics. The reason for this choice is that we have hot-encoded the labels (see the previous paragraph) and therefore the categorical version of these function is needed.

Now let us try to perform the training as we did in Chapter 14 for our single neuron model. The code structure is always the same. Try to run the following code on your laptop

model = build\_model(tf.keras.optimizers.SGD(momentum = 0.0, learning\_rate = 0.0001))

EPOCHS = 1000

history = model.fit(

data\_train\_norm, labels\_train,

epochs = EPOCHS, verbose = 0,

callbacks = [tfdocs.modeling.EpochDots()])

We have set as optimizer the standard version of the gradient descent. You should immediately notice one thing: is **very** slow. The problem is that the model as we coded it, will create a huge matrix for all observations (that are 60000) and then will modify the weights and bias only after a complete sweep over all observations. This requires quite some resources, memory, and CPU. If that would be the only choice we have, we would have been doomed. Keep in mind that in the deep learning world 60000 examples of 784 features is not a big dataset at all. So, we need to find a way of letting our model learn faster.

The above piece of code will return the value of the loss function and the relative accuracy on the training dataset every 100 epochs

Epoch: 0, categorical\_accuracy:0.1802, loss:2.2518,

..............................................................

Epoch: 100, categorical\_accuracy:0.8062, loss:0.5901,

..............................................................

Epoch: 200, categorical\_accuracy:0.8295, loss:0.5012,

..............................................................

Epoch: 300, categorical\_accuracy:0.8404, loss:0.4652,

..............................................................

Epoch: 400, categorical\_accuracy:0.8467, loss:0.4450,

..............................................................

Epoch: 500, categorical\_accuracy:0.8510, loss:0.4316,

..............................................................

Epoch: 600, categorical\_accuracy:0.8546, loss:0.4218,

..............................................................

Epoch: 700, categorical\_accuracy:0.8567, loss:0.4140,

..............................................................

Epoch: 800, categorical\_accuracy:0.8587, loss:0.4076,

..............................................................

Epoch: 900, categorical\_accuracy:0.8605, loss:0.4019,

..............................................................

To do some basic error analysis you will also need the dev dataset, that we have already loaded and prepared in the previous paragraphs.

Do not get confused from the fact that the filename contains the word test. Sometime the dev dataset is called test dataset. When we discuss later in the book error analysis, we will use three datasets: train, dev and test. To remain consistent during the entire book we will now use the name dev not to confuse you with different names in different chapters.

To calculate accuracy on the dev dataset we will use the function model.evaluate() and we will apply the built model on the dev dataset.

test\_loss, test\_accuracy = model.evaluate(data\_test\_norm, labels\_test, verbose = 0)

print('The accuracy on the test set is equal to: ', int(test\_accuracy\*100), '%.')

which returns

The accuracy on the test set is equal to: 84 %.

To summarize, we have applied the model trained on the 60000 observations on the dev test (made of 10000 observations) and we have calculated the accuracy on both datasets.

A good exercise would be to include this calculation in your model so that your build\_model() function would automatically return the two values.

## Gradient descent variations

In Chapter 19 the very basic gradient descent algorithm (also called batch gradient descent) will be described. This is not the smartest way of finding the cost function minimum. Let us have a look at the variations that you need to know and let us compare how efficient they are using the Zalando dataset.

### Batch gradient descent

The gradient descent algorithm we will see in Chapter 19 calculates the weights and bias variations for each observation but performs the learning (weights and bias update) only after all observations have been evaluated, or in other words after a so-called epoch. (Remember a cycle through the entire dataset is called an epoch).

**Advantages**

* The fewer weights and bias updates mean a more stable gradient and usually result in a more stable convergence

**Downsides**

* Usually, this algorithm is implemented in such a way that all the dataset must be in memory, therefore is computational quite intensive
* This algorithm is typically very slow for very big datasets

A possible implementation could look like this

model\_bgd = build\_model(tf.keras.optimizers.SGD(momentum = 0.0, learning\_rate = 0.0001))

EPOCHS = 100

history\_bgd = model\_bgd.fit(

data\_train\_norm, labels\_train,

epochs = EPOCHS, verbose = 0,

callbacks = [tfdocs.modeling.EpochDots()])

Running the code for 100 epochs would give a result similar to this

Epoch: 0, categorical\_accuracy:0.1744, loss:2.2385,

..............................................................

And if we look at the last loss function and metrics values during training

hist\_bgd = pd.DataFrame(history\_bgd.history)

hist\_bgd['epoch'] = history\_bgd.epoch

hist\_bgd.tail()

loss categorical\_accuracy epoch

95 0.605132 0.795867 95

96 0.603499 0.796833 96

97 0.601903 0.796817 97

98 0.600324 0.797283 98

99 0.598758 0.797750 99

This code ran in roughly 2.5 minutes, reaching an accuracy of 80% on the training set.

### Stochastic gradient descent

The stochastic[[5]](#footnote-5) gradient descent (abbreviated SGD) calculates the gradient of the cost function and then updates weights and biases for each observation in the dataset.

**Advantages**

* The frequent updates allow an easy check on how the model learning is going (you do not need to wait until all the dataset has been considered)
* In a few problems this algorithm may be faster than batch gradient descent
* The model is intrinsically noisy and that may allow the model to avoid local minima when trying to find the absolute minimum of the cost function

**Downsides**

* On large datasets this method is quite slow, since is very computationally intensive due to the continuous updates
* The fact that the algorithm is noisy can make it hard for the algorithm to settle on a minimum for the cost function, and the convergence may be not so stable as expected

A possible implementation could look like this

model\_sgd = build\_model(tf.keras.optimizers.SGD(momentum = 0.9, learning\_rate = 0.0001))

EPOCHS = 100

history\_sgd = model\_sgd.fit(

data\_train\_norm, labels\_train,

epochs = EPOCHS, verbose = 0,

callbacks = [tfdocs.modeling.EpochDots()])

If you let the code run you will get a result that should look like this (the exact numbers will be different each time, since we initialize the weights and biases randomly, but the speed of decrease should be the same)

Epoch: 0, categorical\_accuracy:0.2938, loss:2.0294,

..............................................................

And if we look at the last loss function and metrics values during training

hist\_sgd = pd.DataFrame(history\_sgd.history)

hist\_sgd['epoch'] = history\_sgd.epoch

hist\_sgd.tail()

loss categorical\_accuracy epoch

95 0.393442 0.863067 95

96 0.393088 0.863417 96

97 0.392389 0.863383 97

98 0.391859 0.863867 98

99 0.391405 0.864433 99

This code ran in roughly 2.5 minutes, reaching an accuracy of 86% on the training set (therefore improving the batch gradient descent results). Let us try another method.

### Mini-batch gradient descent

In this variation of the gradient descent, the dataset is split into a certain number of small (from here the name mini-batch) groups of observations (called batches) and weights and biases are updated only after each batch has been fed to the model. This is by far the most common used method used in the field of deep learning.

**Advantages**

* The model update frequency is higher than with batch gradient descent but lower than SGD, therefore allowing for a more robust convergence
* This method is computationally much more efficient than batch gradient descent or SGD since less calculations and resources are needed
* This variation is by far (as we will see later) the fastest of the three

**Downsides**

* The use of this variation introduces a new hyperparameter that needs to be tuned: the batch size (number of observations in the mini-batch)

A possible implementation could look like this for a batch size of 50:

model\_mbgd = build\_model(tf.keras.optimizers.SGD(momentum = 0.9, learning\_rate = 0.0001))

EPOCHS = 100

history\_mbgd = model\_mbgd.fit(

data\_train\_norm, labels\_train,

epochs = EPOCHS, verbose = 0,

batch\_size = 50,

callbacks = [tfdocs.modeling.EpochDots()])

In this example we used 50 observations each time before updating weights and biases. Running it will give you a result that should like this (remember your numbers will be different than these due to the random initialization of weights and biases):

Epoch: 0, categorical\_accuracy:0.3677, loss:1.8962,

..............................................................

And if we look at the last loss function and metrics values during training

Loss categorical\_accuracy epoch

95 0.434905 0.848883 95

96 0.434164 0.849417 96

97 0.433344 0.849150 97

98 0.432811 0.848867 98

99 0.432150 0.849883 99

In this case we have used the same learning rate of we used with batch gradient descent and SGD and reached a cost function value of 0.43, a bigger value than 0.39 reached with SGD but much smaller than the 0.60 reached in batch gradient descent, and it needed only 1.5 minutes, so is a factor 1.7 faster than SGD. After 100 epochs we reached here an accuracy of 85%.

### Comparison of the variations

Let us summarize the findings for our three variations of gradient descent for **100 epochs** (Table 15-3).

Table 15-3. Comparison of the performances of three different variations of gradient descent.

|  |  |  |  |
| --- | --- | --- | --- |
| **Gradient descent variation** | **Running time** | **Final value of cost function** | **Accuracy** |
| Batch gradient descent | 2.5 min | 0.60 | 80% |
| Stochastic gradient descent | 2.5 min | 0.39 | 86% |
| Mini-batch gradient descent | 1 min | 0.43 | 85% |

Now you can see that SGD is the algorithm that reaches the lowest value of the cost function with the same number of epochs but, together with batch gradient descent, is slow with respect to mini-batch gradient descent.

In Figure 15-15 you can see the difference in how the cost function decreases with different mini-batch sizes. It is clear how, with respect to number of epochs, the smaller the mini-batch size the fastest (not in time) decrease.

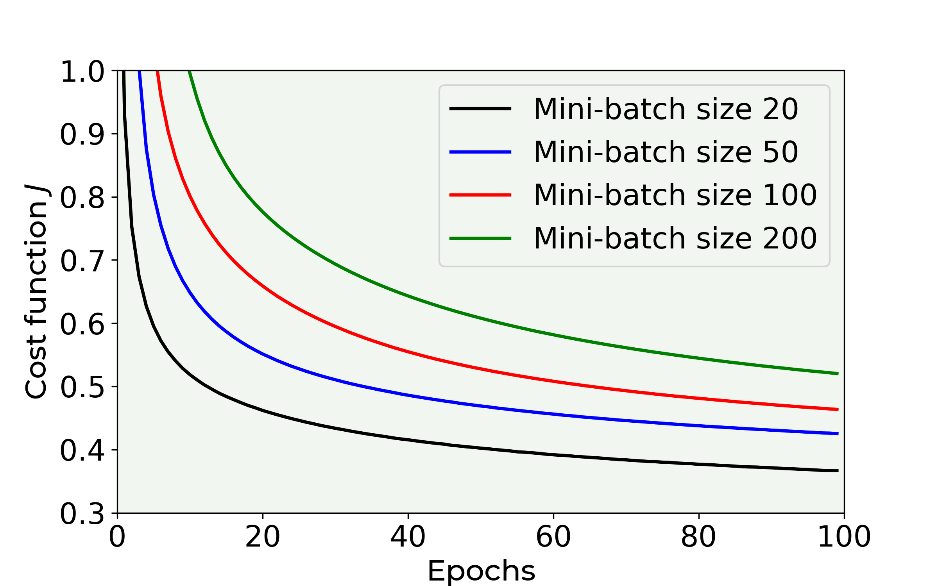


Figure 15-15. A comparison of speed of convergence of mini-batch gradient descent algorithm with different mini-batch sizes. The learning rate used for this figure was . Note that the time needed by each case is not the same and the smallest the mini-batch size, the more time the algorithm needs.

**Tip** The best compromise between running time and convergence speed (with respect to number of epochs) is achieved by the mini-batch gradient descent. The optimal size of the mini-batches is dependent on your problem, but usually small numbers like 30 or 50 are a good place to start. You will find a compromise between running time and convergence speed.

To give you an idea of how the running time depends on the value of the cost function can reach after 100 epochs check Figure 15-16. Each point is labeled with the size of the mini-batch used in that run. You can see that decreasing the mini-batch size from 300 decreases the value of after 100 epochs quickly without increasing the running time significantly, until you arrive at a value for the mini-batch size that is around 20. At that point, the time start to increase quickly and the value for after 100 epochs does not decrease anymore as quickly and flatten out.

Intuitively the best compromise is to choose a value for the mini-batch size when the curve is closer to zero (small running time and small cost function value), and that is at a mini-batch size value of 20 in this specific case. This is the reason why those are the most chosen values. After that point, the increase in running time becomes very quick and is not worth anymore to decrease the mini-batch size. Note that for other datasets the optimal value may be very different. So is worth trying different values to see which one works best. In very big datasets you may want to try bigger values, like 200, 300 or 500. In our case we have 60000 observations and a mini-batch size of 50, that gives 1200 batches. If you have much more data, for example 1e6 observations, a mini-batch size of 50 would give 20000 batches. So, keep that in mind and try different values to see which one works best.

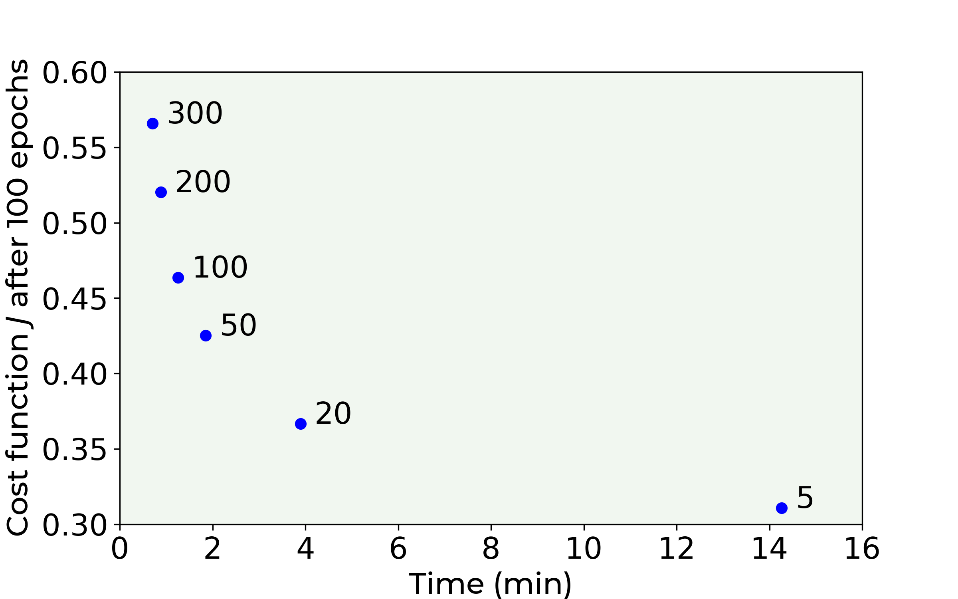


Figure 15-16. The plot shows the value of the cost function after 100 epochs for the Zalando dataset vs. the running time needed to run through 100 epochs. Note that the points are single runs, and the plot is only indicative of the dependency. Running time and cost function value have a small variance when evaluated over several runs. This variance is not shown in the plot.

As a tip, it is good programming practice to write a function that runs your evaluations. In this way you can tune your hyperparameters (like the minibatch size) without copy and pasting the same chunk of code over and over. The following function is one you can use to train our model with different mini-batch sizes (of course you can add more parameters to be optimized, like for example, the number of epochs, the learning rate, etc.):

def mini\_batch\_gradient\_descent(mb\_size):

# build model

model\_mbgd = build\_model(tf.keras.optimizers.SGD(momentum = 0.9, learning\_rate = 0.0001))

# set number of epochs

EPOCHS = 100

# train model

history\_mbgd = model\_mbgd.fit(

data\_train\_norm, labels\_train,

epochs = EPOCHS, verbose = 0,

batch\_size = mb\_size,

callbacks = [tfdocs.modeling.EpochDots()])

# save performances

hist\_mbgd = pd.DataFrame(history\_mbgd.history)

hist\_mbgd['epoch'] = history\_mbgd.epoch

return hist\_mbgd

**Tip** Writing a function with the hyperparameters as inputs is common practice. This allows you to test different models with different values for the hyperparameters and check which one is better.

## Examples of wrong predictions

Running the model with batch gradient descent, one hidden layer with 15 neurons for 1000 epochs and learning rate of 0.0001 will give us an accuracy on the training set of 86%. You can increase the accuracy using more neurons in your hidden layer. For example, using 50 neurons, using 1000 epochs, and using a learning rate of 0.0001 will allow you to reach 87% on the training set and 85% on the test set. It is interesting to check a few examples of wrongly classified images, to check if we could understand something from the errors. In Figure 15-17 you can see an example of wrongly classified images for each class.



Figure 15-17. One example of wrongly classified images for each class. Over each image the True class (labeled with "True") and the predicted (labeled with "Pred") class are reported. The model used here has 1 hidden layer with 15 neurons, has been run for 1000 epochs with a learning rate of 0.0001.

Some errors are understandable like for example the example in the upper right: a coat has been wrongly classified as a pullover. We could easily make the same mistake. The wrongly classified dress is, on the other side, easy for a human to get it right.

## Weight initialization

If you have really tried to run the code, you will have realized that the convergence of the algorithm is strongly variable, and this depends on the way you initialize your weights. In the previous sections we focused on understanding how such a network works to not get distracted by additional information, but it is time to look at this problem a bit more closely since this plays a fundamental role with many layers.

Basically, we want to avoid the gradient descent algorithm to explode and start returning nan. For example, in our first layer for the neuron we will need to calculate the ReLU activation function of the quantity (check the beginning of the Chapter for an explanation if you forgot why)

Normally in a deep network the number of weights is quite big, so you can easily imagine that if the are big the quantity can be quite big, and then the ReLU activation function can return a nan value since the argument is too big for Python to calculate it properly. So, you want the to be small enough to avoid an explosion of the output of the neurons, and big enough to avoid the outputs to die out and therefore make the convergence a very slow process.

The problem has been researched extensively [5], and there are different initialization strategies depending on the activation function you are using. Let us outline few of them in the following Table 15-4 where we assume that the weights will be initialized with a normal distribution with mean 0 and standard deviation as given in the table (note that the standard deviation will depend on the activation function you want to use)

Table 15-4. Some examples of weights initialization for deep neural networks.

|  |  |
| --- | --- |
| **Activation function** | **Standard deviation σ for a given layer** |
| Sigmoid | Usually called Xavier Initialization |
| ReLU | Usually called He Initialization |

In a layer the number of inputs will be the number of neurons of the preceding layer and the number of outputs will be the number of neurons in the layer coming next . So, we will have

And

Very often deep networks as the one we discussed before, will have several layers all with the same number of neurons, therefore you will have for most of the layers and therefore you will have for Xavier initialization

And for ReLU activation functions the He initialization will be

Let us consider the ReLU activation function (the one we have used in this chapter). Every layer, as we have discussed will have neurons. A way of initialization of the weights for our single hidden layer for example would be then

initializer = tf.keras.initializers.HeNormal()

layer = tf.keras.layers.Dense(15, kernel\_initializer = initializer)

Typically, to make evaluation and construction of networks easier the most typical initialization form used is for ReLU activation function

And

For Sigmoid activation function.

Using this initialization can speed up training considerably and is the standard way that many libraries initialize weights (for example the Caffe library).

In Keras, weights initialization is straightforward by means of the tf.keras.initializers function. Have a look to Keras documentation to see which initialization strategies are available [6].

## Adding many layers efficiently

Now always typing all this code each time is a bit tedious and error prone. Usually what one does is to define a function that creates a layer. This can be done easily with this code

def model\_nlayers(num\_neurons, num\_layers):

# build model

inputs = keras.Input(shape = 784) # input layer

# first hidden layer

dense = layers.Dense(num\_neurons,

activation = 'relu')(inputs)

# customized number of layers and neurons per layer

for i in range(num\_layers - 1):

dense = layers.Dense(num\_neurons,

activation = 'relu')(dense)

# output layer

outputs = layers.Dense(10, activation = 'softmax')(dense)

model = keras.Model(inputs = inputs,

outputs = outputs,

name = 'model')

# set optimizer and loss

opt = tf.keras.optimizers.SGD(momentum = 0.9,

learning\_rate = 0.0001)

model.compile(loss = 'categorical\_crossentropy',

optimizer = opt,

metrics = ['categorical\_accuracy'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = 200, verbose = 0,

batch\_size = 20,

callbacks = [tfdocs.modeling.EpochDots()])

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

return hist

Let us go through the code:

* First, we define the dimensions of the input layer.
* Then we define the first hidden layer (the number of neurons is given as function’s input).
* Then we add the other hidden layers one at a time. The number of layers is given as function’s input.
* Finally, we add the output layer, and we stack all the layers together inside our model.
* We then compile and train the model, returning its performances.

Notice that in the previous code we used Keras functional API (you should remember it, since we discussed it in details in Chapter 12), a functionality that provides a more flexible way to create models with respect to the tf.keras.Sequential API. With this functionality we easily created a model with customizable number of layers and neurons per layer.

So, to create our networks we can simply apply our function with different number of neurons and layers as inputs

res\_10\_1 = model\_nlayers(10, 1)

res\_10\_2 = model\_nlayers(10, 2)

res\_10\_3 = model\_nlayers(10, 3)

res\_10\_4 = model\_nlayers(10, 4)

res\_100\_4 = model\_nlayers(100, 4)

The code now is much easier to understand and you can use it to create networks as big as you wish.

With the function defined above is very easy to run several models and compare them, as we have done in Figure 15-18 where we have tested 5 different models:

* 1 layer and 10 neurons each layer
* 2 layers and 10 neurons each layer
* 3 layers and 10 neurons each layer
* 4 layers and 10 neurons each layer
* 4 layer and 100 neurons each layer

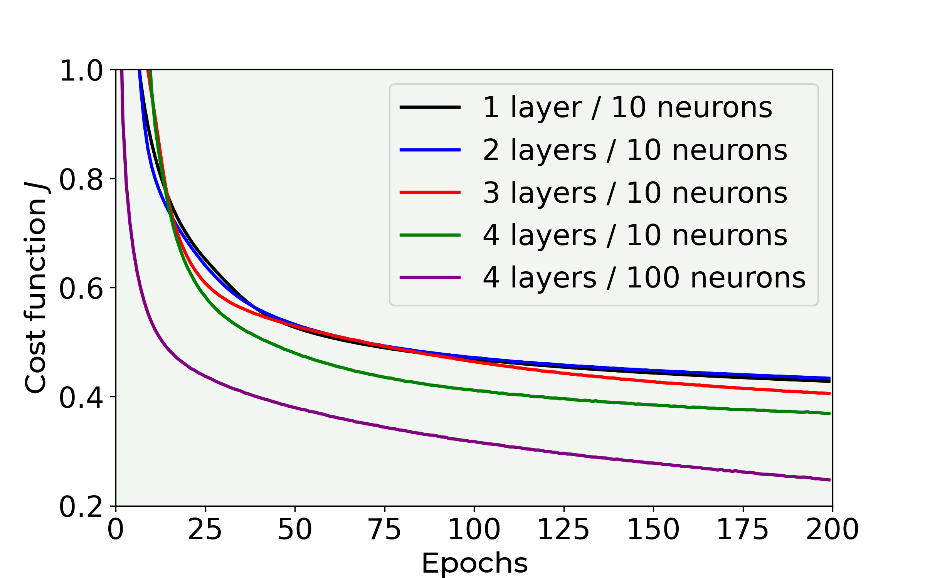


Figure 15-18. The cost function vs epochs for 5 models as described in the legend.

In case you are wondering, the model with 4 layers with each 100 neurons, that seems much better than the others, is starting to go in the overfitting regime, with a train set accuracy of 91% and of 88% on the dev set (after only 200 epochs).

### Advantages of additional hidden layers

It is instructive to play with the models. Try varying number of layers, number of neurons, how to initialize the weights and so on. If you invest some time you can reach an accuracy of over 90% in few minutes of running time, but that requires some work. If you try several models you may realize that in this case using several layers does not seem to bring benefits versus a network with just one. This is often the case.

Theoretically speaking a one-layer network can approximate every function you can imagine, but the number of neurons needed may be very large and therefore the model becomes not very useful. Now the catch is that the ability of approximating a function does not mean that the network is able to learn to do it, due for example to the sheer number of neurons involved or the time needed.

Empirically it has been shown that networks with more layers require much smaller number of neurons to reach the same results and usually generalize better to unknown data.

**Note** Theoretically speaking, you do not need to have multiple layers in your networks, but often in practice you should. It is almost always a good idea to try a network with several layers with a few neurons in each instead of a network with one layer populated by a huge number of neurons. There is no fix rule on how to decide how many neurons or layers are best. You should try starting with low numbers of layers and neurons and then increase them until your results stop getting better.

In addition, having more layers may allow your network to learn different aspects of your inputs. For example, one layer may learn to recognize vertical edges of an image, and another horizontal ones. Remember that in this chapter we have discussed networks where each layer is identical (up to the number of neurons) to all the others. We will see later in the Convolutional Neural Network Chapter how you can build networks where each layer performs very different tasks and is also structured very differently from each other, making this kind of networks much more powerful for certain tasks with respect to the models we have discussed in this chapter.

As a simple example, imagine predicting the selling prices of houses. In this case a network with several layers may learn more information on how the features relates to the price. For example, the first layer may learn basic relationships like bigger houses equal higher prices. But the second layer may learn more complex features like big house and small number of bathrooms equals to low selling prices.

## Comparing different networks

Now you should know how to build neural networks with a huge number of layers or neurons. But it is relatively easy to lose yourself in a forest of possible models without knowing which ones are worth trying. Suppose you start with a network (as we have done in the previous sections) with one hidden layer with 5 neurons, one layer with 10 neurons (for our softmax function) and our softmax neuron. Now suppose you have reached some accuracy and you would like to try different models. At first you should try increasing the number of neurons in your hidden layers to see what you can achieve. Check Figure 15-19, where we have plotted the cost function as it decreases for different number of neurons. The calculations have been performed with a mini-batch gradient descent with a batch size of 50, one hidden layer with respectively 1, 5, 15 and 30 neurons and a learning rate of 0.0001. You can see how moving from one neuron to five immediately make the convergence faster. But further increasing the number of neurons does not bring as much improvement. For example, increasing from 15 to 30 brings no improvement at all.

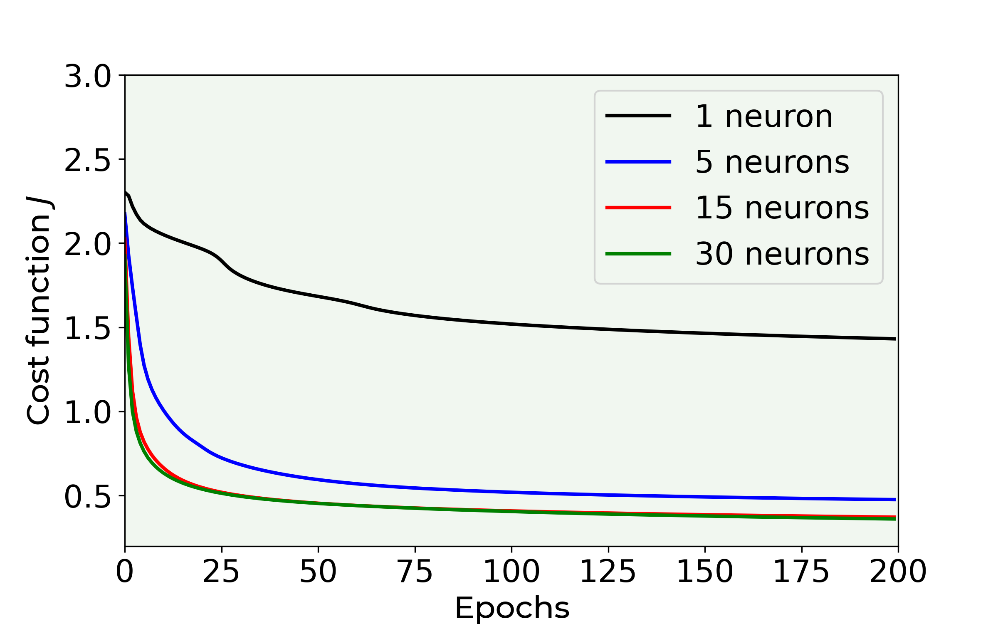


Figure 15-19. Cost function decreasing vs. epochs for a neural network with one hidden layer with respectively 1, 5, 15 and 30 neurons as indicated in the legend. The calculations have been performed with mini-batch gradient descent with a batch size of 50 and a learning rate of 0.0001.

Let us first try to find a way of comparing those networks. Only comparing the number of neurons can be very misleading as we will see shortly. Remember that your algorithm is trying to find the best combinations of weights and biases to minimize your cost function. But how many learnable parameters do we have in our model? We have the weights and the biases. You will remember from our theoretical discussion that we can associate a certain number of weights to each layer, and the number of learnable parameters in our layer that we will indicate with is given by the total number of elements in the matrix , that is (where we have by definition) plus the number of biases we have (in each layer we will have biases). The number can then be written as:

So that the total number of learnable parameters in our network (indicated here with ) can be written as

Where by definition . Now please note that the parameter of our network is strongly architecture dependent. Let us calculate it in some practical examples (see Table 15-5).

Table 15-5. Some examples of different network architectures with their corresponding parameter.

|  |  |  |
| --- | --- | --- |
| **Network architecture** | **Parameter (number of learnable parameters)** | **Number of neurons** |
| **Network A**: 784 features, 2 layers: , |  | 25 |
| **Network B**: 784 features, 16 layers: , |  | 25 |
| **Network C**: 784 features, 3 layers: , , |  | 30 |

Draw your attention to network A and B. Both have 25 neurons, but the parameter is much bigger (more than a factor of ten) than . You can easily imagine how network A will be much more flexible in learning than network B, even if the number of neurons is the same.

**Note** in practice is not a measure of how complex or how good a network is. This is not the case and it may well happen that of all the neurons only a few will play a role, therefore only calculating in this way will not tell the entire story. There is a vast amount of research on the so-called effective degrees of freedom of deep neural networks but that would go way beyond the scope of this book. But this parameter will give a good rule of thumb in deciding if the set of models you want to test are hopefully in a reasonable complexity progression.

Nonetheless, checking for the model you want to test may give you some hints on which you should neglect and which you should try. For example, let us consider the cases we have tested in Figure 15-19 and let us calculate the parameter for each network (see Table 15-6).

Table 15-6. Network architectures tested in Figure 15-19 with their corresponding parameter.

|  |  |  |
| --- | --- | --- |
| **Network architecture** | **Parameter (number of learnable parameters)** | **Number of neurons** |
| 784 features, 1 layer with 1 neuron, 1 layer with 10 neurons |  | 11 |
| 784 features, 1 layer with 5 neuron, 1 layer with 10 neurons |  | 15 |
| 784 features, 1 layer with 15 neuron, 1 layer with 10 neurons |  | 25 |
| 784 features, 1 layer with 30 neuron, 1 layer with 10 neurons |  | 40 |

From Figure 15-19 let us suppose we choose the model with 15 neurons as our candidate for our best model. Now let us suppose we want to try a model with 3 layers all with the same number of neurons that should compete (and possibly be better) than our (for the moment) candidate model with 1 layer and 15 neurons. What should we choose as a starting point for the number of neurons in the 3 layers? Let us indicate as model A the one with one layer with 15 neurons and with B a model with 3 layers with an (yet) unknown number of neurons in each layer indicated with . We can easily calculate the parameter for both networks

And

What value for will give ? We can easily solve the equation

You should be able to solve a quadratic equation, so we will only look at the solution here (hint: try to solve it). This equation is solved for a value of , but since we cannot have 14.4 neurons we will need to use the closest integer, that would be . For we will have , a value very close to 11935.

**Note** The fact that the two networks have the same number of learnable parameters does not mean that they can reach the same accuracy and does not even mean that if one learns very fast the second will learn at all!

But our model with 3 layers with each 14 neurons could be a good starting point for our further testing.

Let us discuss another point that is important when dealing with a complex dataset. Let us consider our first layer. Let us suppose we consider the Zalando dataset and we create a network with two layers: the first with one neuron and the second with many. All the complex feature that your dataset has, may well go lost in your single first neuron, since it will combine all features in one single value and pass the same exact value to all other neurons of the second layer.

### Tips for choosing the right network

We have discussed a lot of cases, we have seen a lot of formulas, but how can we decide how to design our network?

Unfortunately, there is not a fix set of rules. But you may consider the following tips:

* When considering a set of models (or network architectures) you want to test, a good rule of thumb is to start with the less complex one and move to more complex ones. A rule of thumb to estimate the relative complexity (to make sure that you are moving in the right direction) is the use of the parameter .
* In case you cannot reach a good accuracy check if any of your layers has a particular low number of neurons. This layer may kill the effective capacity of learning from a complex dataset of your network. Check for example the case with one neuron in Figure 15-19. The model cannot reach low values for the cost function because the network is too simple to learn from a complex dataset as the Zalando one.
* Remember that low or high number of neurons is always relative to the number of features you have. If you have only two features in your dataset one neuron may well be enough, but if you have few hundreds (like in the Zalando dataset where than you should not expect one neuron to be enough.
* Which architecture you need is also dependent from what you want to do. Is always worth checking online literature to see what others have already discovered for specific problems. For example, is well known that for image recognition convolutional networks are very good, so they would be a good choice.

**Final Tip** When moving from a model with layers to one with layers, is always a good idea to start with the new model using a slightly lower number of neurons in each layer and then increasing it step by step. Remember that more layers have a chance of learning complex features more effectively, so if you are lucky less neurons may be enough. It is something worth trying. Always keep track of your optimizing metric (remember from Chapter 14?) for all your models. When you are not getting much improvement anymore, it may be worth trying completely different architectures (maybe convolutional neuronal networks, etc.).

# Exercises

EXERCISE 1

Try to build a multiclass classification model like the one we saw together in this Chapter, but with a different dataset, the MNIST database of handwritten digits (<http://yann.lecun.com/exdb/mnist/>). To download the dataset from TensorFlow use the following lines of code:

from tensorflow import keras

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

Difficulty: easy.

EXERCISE 2

Try to apply He weights initialization in the multiclass classification problem we saw in the Chapter and see if you can speed up the learning phase. Difficulty: medium.

EXERCISE 3

Try to optimize the feed-forward neural network built in this Chapter to reach the best possible accuracy (without overfitting the training dataset!). Tune the number of epochs, the learning rate, the optimizer, the number of neurons, layers and mini-batches. Hint: write a function like the one we used to test different numbers of layers and neurons and give it as inputs all the tunable parameters. Difficulty: hard.

EXERCISE 4

Consider the regression problem we solved with a model made by one single neuron in Chapter 14 (predicting radon activity in U.S. houses). Try to build a feed-forward neural network to solve the same regression task. See if you can get better prediction performances. Hint: you will need to change the loss function and the metrics to evaluate your results, and one-hot encoding will not be necessary anymore. Difficulty: hard.

# References

[1] Burnham, K. P.; Anderson, D. R. (2002), Model Selection and Multimodel Inference (2nd ed.), Springer-Verlag

[2] <https://en.wikipedia.org/wiki/Zalando>, last accessed 16.02.2021

[3] [www.kaggle.com](http://www.kaggle.com), last accessed 16.02.2021

[4] Xiao, Han, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms." arXiv preprint arXiv:1708.07747 (2017)

[5] "Understanding the Difficulty of Training Deep Feedforward neural networks", X. Glorot, Y. Bengio (2010), <https://goo.gl/bHB5BM>

[6] <https://www.tensorflow.org/api_docs/python/tf/keras/initializers>, last accessed 23.02.2021

1. Bias is a measure of the error originating from models that are too simple to capture the real features of our data. [↑](#footnote-ref-1)
2. To do a proper error analysis we will need at least three parts, sometimes four. But to get a basic understanding of the process two parts are enough. [↑](#footnote-ref-2)
3. <https://en.wikipedia.org/wiki/MIT_License> [↑](#footnote-ref-3)
4. As a side note, this technique is often used to feed categorical variables to machine learning algorithms. [↑](#footnote-ref-4)
5. Stochastic means that the updates have a random probability distributions and cannot be predicted exactly. [↑](#footnote-ref-5)