Metric Analysis

# Introduction

Let's consider the problem we analyzed in Chapter 3 where we performed classification on the Zalando dataset. While doing all our work we made a strong assumption without explicitly saying it: we assumed that all the observations where correctly labelled! We cannot say that with certainty. To perform the classification some manual intervention was needed, and therefore there must be a certain number of images that are wrongly classified since humans are not perfect. This is an important revelation. Consider the following scenario: in Chapter 3 we reached with our model roughly 90% of accuracy. One could try to get better and better accuracy, but when is sensible to stop trying? If your labels are wrong in 10% of the cases, our model, as sophisticated as it may be, will never be able to generalize to new data with a very high accuracy, since it will have learned wrong classes for many images. We spent quite some time in checking and preparing the training data, normalizing it for example, but we never spent any time checking the labels themselves. We also assumed that all classes have similar characteristics (will discuss later in this chapter what does that exactly mean, for the moment an intuitive understanding of the idea will suffice). What if the quality of the images for specific classes is worse than for others? What if the number of pixels whose gray value different than zero is dramatically different for different classes? We also did not check if some images are completely blank. What happens in that case? As you can imagine we cannot check all images manually trying to detect such issues. Suppose we have millions of images, a manual analysis is surely not possible.

We need a new weapon in our arsenal to be able to spot such cases and to be able to tell how a model is doing. This new weapon is the focus of this chapter and is what I call "metric analysis". Very often people in the field refer to this array of methods as "error analysis". Personally, I find that this name is very confusing, especially for beginners. Error may refer to too many things: Python code bugs, error in the methods, in the algorithms, errors in the choice of optimizers and so on. We will see in this chapter how to obtain fundamental informations on how your model is doing and on how good your data is. We will do this by evaluating your optimizing metric on a set of different datasets that you can derive from your data.

We have already seen a basic example previously. You will remember that we discussed, in the case of regression, how in the case of we are in a regime of overfitting. Our metric is the MSE and evaluating it on two datasets, training and dev, and comparing the two values can give you information if the model is overfitting or not. We will expand this methdology in this chapter to be able to extract much more information from your data and model.

# Human level performance and Bayes error

In most of the datasets that we use for supervised learning someone must have labelled the observations. Take for example a dataset where we have images that are classified. If we ask people to classify all images (imagine being possible, regardless of number of images) the accuracy obtained will never be 100%. Some images maybe too blurry to be classified correctly, and people make mistakes. If for example 5% of the images are not classifiable correctly, due for example to how blurry they are, we must expect that the maximum accuracy people can reach will always be less than 95%.

Let's consider a problem of classification. First, let's define what we mean with the word "error". We will indicate in this Chapter with the word "error" the following quantity, indicated with :

For example, if with a model we reach an accuracy of 95%, we will have or expressed as percent .

A useful concept to understand is the "human level performance", that can be defined as

**Human level performance (definition 1)**: this is the lowest value for the error that can be reached by a person performing the classification task. We will indicate it with .

Let's make a concrete example. Let's suppose we have a set of 100 images. Now let's suppose we ask three people to classify the 100 images. Let's imagine that they obtain 85%, 83% and 84% accuracy. In this case, human level performance accuracy will be . Note that someone else maybe much better at this task, and therefore is always important to consider that the value of we get is always an estimate and should only serve as a guideline.

Now let's complicate things a bit. Let's suppose we are working on a problem in which doctors classify MRI scans in two classes: with signs of cancer and without. Now let's suppose we calculate from the results of untrained students obtaining 15%, from doctors with few years of experience obtaining 8%, from experienced doctors obtaining 2% and from experienced **groups** of doctors obtaining 0.5%. What is then ? You should always choose the lowest value you can get, for reasons we will discuss later.

We can now expand the definition of with a second definition:

**Human level performance (definition 2)**: this is the lowest value for the error that can be reached by people or **groups** of people performing the classification task.

You don't need to decide which definition is right. Just use the one that gives you the lowest value of .

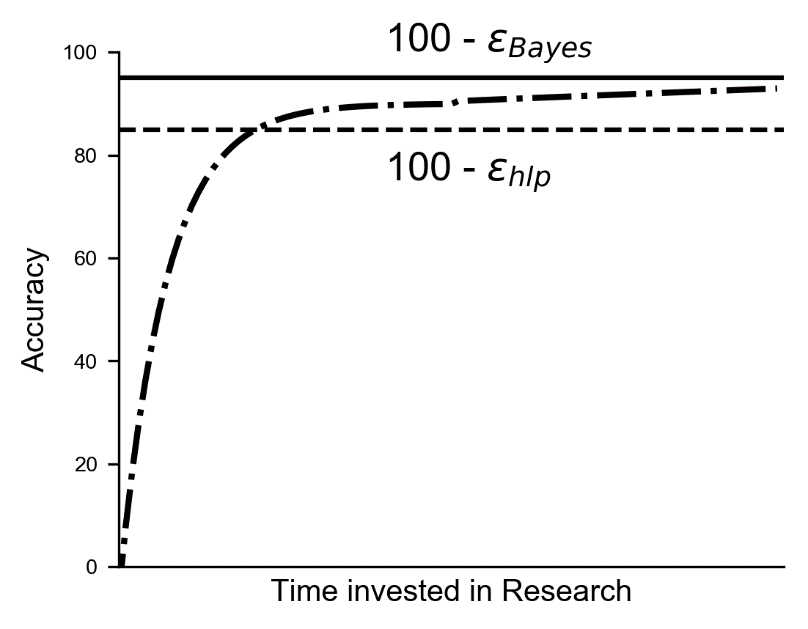
Now let's talk a bit about the why we must choose the lowest value we can get for . Let's suppose that of the 100 images, 9 are too blurry to be correctly classified. That means that the lowest error any classifier will be able to reach is 9%. The lowest error that that can be reached by any classifier is called the Bayes error. We will indicate it with . In this example . Usually is very close to , at least in tasks where humans excel like image recognition. It is commonly said that that human level performance error is a proxy for the Bayes error. Normally is impossible or very hard to know and therefore practictioners use assuming the two are close, since the latter is easier (relatively) to estimate.

Now keep in mind that it makes sense to compare the two values and assume that is a proxy for only if persons (or groups of persons) perform classification in the same way as the classifier. For example, is ok if both uses the same images to do classification. But, in our cancer example, if the doctors use additional scans and analysis to make sure of cancer than the comparison is not fair anymore since human level performance will not be a proxy for Bayes error anymore. Doctors, having more data at their disposal, will be clearly better than the model that have as input only the images at disposal.

and are close to each other only in cases where the classification is done in the same way from humans and from the model. So always check if that is the case before assuming that human level performance is a proxy for the Bayes error.

Something else that you will notice when working on models, is that with relatively low effort you can reach a quite low error, and often (almost) reach . After passing human level performance (and in several cases that is possible) progress tends to be very very slow, as it is illustratively indicated in Figure 6-1.

Figure 6-1: Typical values of accuracy that can be reached vs. amount of time invested. At the beginning is very easy with Machine learning to get quite a good accuracy, and ofter reach . This is intuitively indicated with the line in the plot. After that point the progress tends to be very slow.



As long as the error of your algorithm is bigger than you can use the following techniques to get better results:

* Get better labels from humans or groups, for example from groups of doctors in case of medical data as in our example
* Get more labelled data from humans or groups
* Do a good metric analysis to determine what is the best strategy for getting better results. You will learn how in this chapter

As soon as your algorithm gets better than human level performance you cannot rely on those techniques anymore. So is important to get an idea of those numbers to be able to decide what to do to get better results. In our example of MRI scans we could get better labels relying on other sources that are not related to humans, for example checking the diagnosis few years after the MRI time point, when usually is clear if the patient had developed cancer or not. Or for example in the case of image classification you may decide to take yourself few thousands of images of specific classes. This is not usually possible, but I wanted to make the concept clear: you can get get labels with other means other than asking humans to perform the same kind of task that your algorithm is performing.

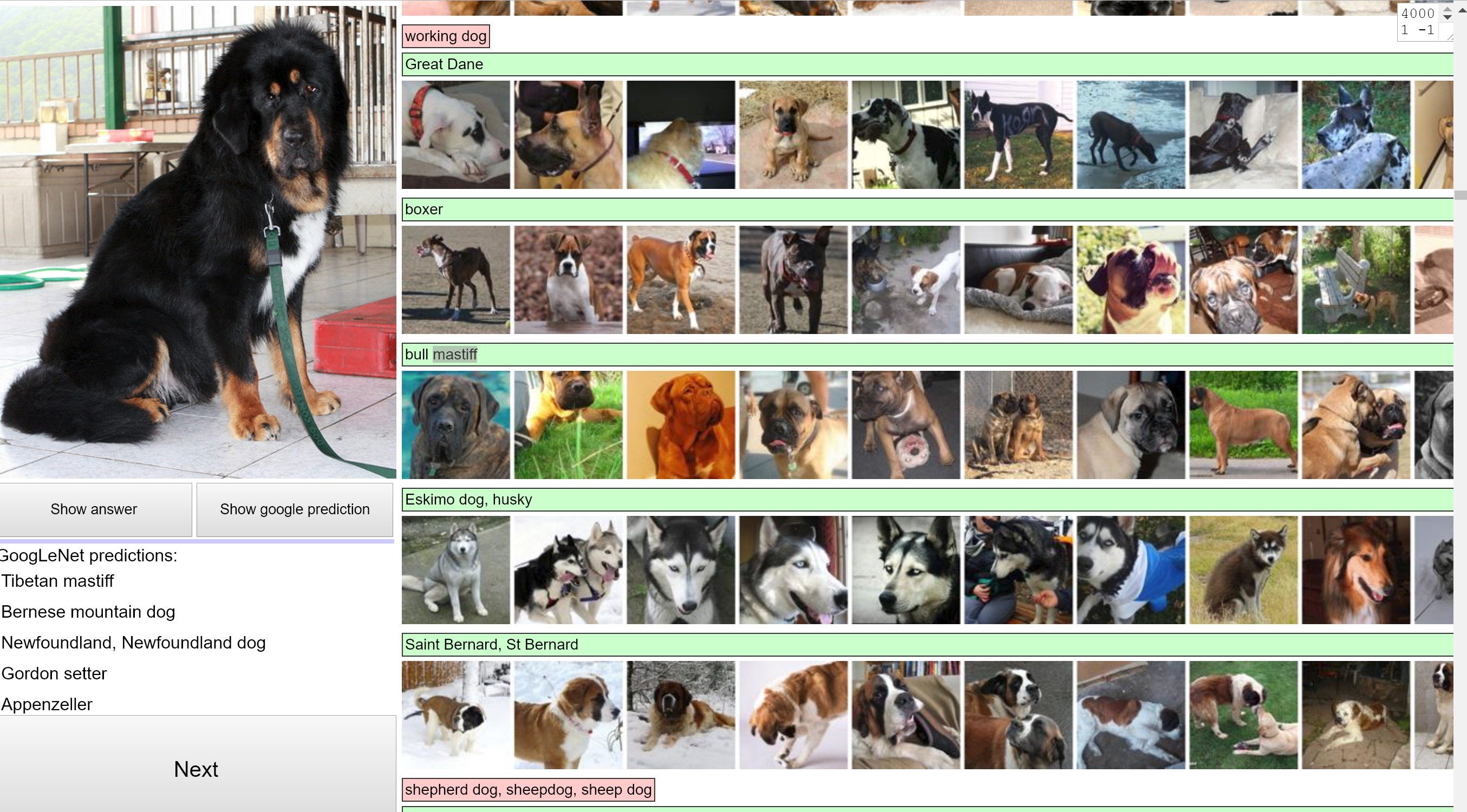
Human level performance is a good proxy for Bayes error for tasks where humans excelt at, like image recognition. It can be very far from Bayes error for tasks where humans are very bad at.

# A short story about human level performance

I want to tell you a story, about the work that Andrej Karpathy has done, while trying to estimate human level performance in a specific case. You can read the entire story in his blog post (a long post, but one that I suggest you read) here <https://goo.gl/iqCbC0>. Now let me tell you and summarize what he did, since it is extremely instructive about what really human level performance is. Karpathy was involved in the ILSVRC contest: Imagenet Large Scale Visual Recognition Challenge in 2014 (https://goo.gl/PCHWMJ). The task was made up of 1.2 million images (training set) classified in 1000 categories including objects like animals, abstract objets like a spiral, scenes and many more. Results were evaluated on a dev dataset. GoogleLeNet (model developed by Google) reached an astounding 6.7% error. Karpahty asked himself how do humans compare?

The question is a lot more complicated that it may seems at first sight. Since images were all classified by humans should not be ? Well actually not. In fact, the images were first obtained with a web search, then the images were filtered and labelled asking people binary questions: is this a hook or not (for example)? The images were collected, as Karpathy mention in his blog post, in a binary way. People were not asked to assign to each image a class choosing from the 1000 available as the algorithms were doing. You may think that this is a technicality, but the difference in how the labelling occurs, make the correct evaluation of a quite a complicated matter. So Karpathy set to work and developed a web interface that consisted on an image on the left, and the 1000 classes with examples on the right. You can see an example of the interface in Figure 6-2. You can try the interface (and I suggest you do so) here https://goo.gl/Rh8S6g to realize how complicated is such a task. People trying kept missing classes and making mistakes. The best error that was reached was around 15%. So, he did what every scientist at some point in his career have to do: he bore himself out of his mind and did a careful annotation himself, sometime needing 20 minutes for a single image. As he formulated in his blog post he did it only **#forscience**. He was able to reach a stunning . 1.7% better than the best algorithm at the time. He listed sources of errors to which GoogLeNet is more susceptible to than humans, like problems with multiple objects in an image, and sources of errors to which humans are more susceptible than GoogLeNet, like problems with classes with a huge granularity (dogs are classified in 120 different sub-classes).

Figure 6-2: the web interface developed by Karpathy. Not everyone finds funny to look at 120 breeds of dogs to try to classify the dog on the left (by the way is a tibetan mastiff).

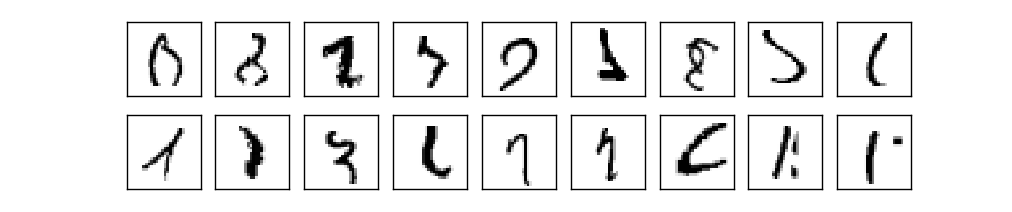


If you have a few hours to spare I suggest you try. You will get a whole new appreciation of the difficulties of evaluating human level performance. Defining and evaluating human level performance is a very tricky task. It is important to understand that is dependent on how humans approach the classification task, and is dependent on the time invested, on the patience of the persons and on many factors that are difficult to quantify. The main reason for it being so important, apart from the philosophical aspect of knowing when a machine becomes better than humans, is that it is often taken as proxy for the Bayes error that gives us a lower limit of our possibilities.

# Human level performance on MNIST

Before moving on the next subject, I would like to give you another example of human level performance on a dataset we have analyzed together: the MNIST dataset. Human level performance has been widely analyzed and has been found that (you can read a good review on the subject by Cireşan D., Multi-column Deep Neural Networks for Image Classification, Technical Report No. IDSIA-04-12, Dalle Molle Institute for Artificial Intelligence, <https://goo.gl/pEHZVB>). Now you may wonder why a human cannot reach a 100% accuracy on simple digits but check Figure 6-3 and see if you can say which digits are in the image. I surely cannot. You may understand why is not possible, and why a person cannot reach 100% accuracy. Other reasons may be related to which culture people are coming from. In some countries the digit seven is written in a very similar way to ones for example, and in some cases, mistakes can be made. In other countries the digit seven have a small dash along the vertical bar making it easy to distinguish it from a one.

Figure 6-3: A set of digits from the MNIST dataset almost impossible to recognize. Such examples are one of the reasons why cannot be zero.



# Bias

Now let's start with metric analysis: a set of procedures that will give you information on how your model is doing, how good or bad your data is, by looking at your optimizing metric evaluated on different datasets.

Metric analysis consists of a set of procedures that will give you information on how your model is doing, how good or bad your data is, by looking at your optimizing metric evaluated on different datasets.

To start, we need first to define a third error: the one evaluated on the training dataset, indicated with .

The first question we want to answer is if our model is not as flexible or complex as needed to reach human level performance. Or said in other words we want to know if our model has a high bias, with respect to human level performance.

To answer the previous question, we can do the following:

* Calculate the error from your model from your training dataset and then calculate . If the number is not small (bigger than a few percent) then we are in the presence of bias (sometime called avoidable bias), or in other words our model is too simple to capture the real subtelties of our data.

Let's define the following quantity

The bigger is the more bias has our model. In this case you want to do better on the training set, since you know you can do better on your training data (we will look at the problem of overfitting in a moment). Following techniques work in reducing Bias

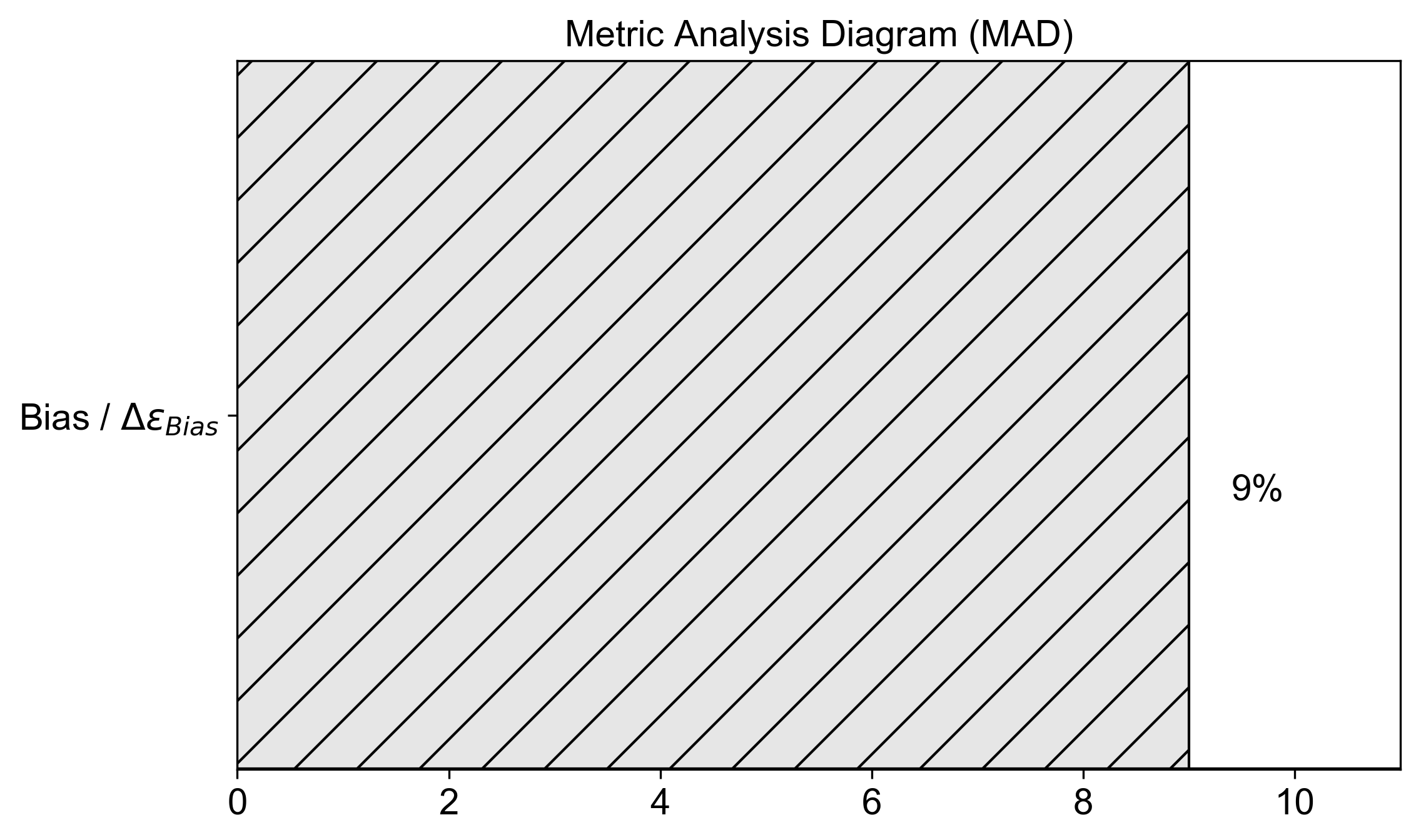
* Bigger networks (more layers or neurons)
* More complex architectures (Convolutional Neural Networks for example)
* Train your model longer (for more epochs)
* Use better optimizers (like Adam)
* Do a better hyperparameter search (we will look at it next chapter)

Now there is something else you need to understand. Knowing and reducing the bias to reach it are two very different things. Suppose you know the for your problem, it does not mean that you need to reach it. It may well be that you are using the wrong architecture, but you may not have the required skills to be able to develop a network sophisticated enough. It may even be that the effort required to reach that error level would be prohibitive (in term of hardware or infrastructure). Always keep in mind what your problem requirement are. Always try to understand what is good enough. For an application that recognize cancer you may want to invest as much as possible to reach the highest accuracy possible: you don't want to send someone home and discover only after monhts the presence of cancer. On the other side if you build a system to recognize cats from web images, you may find a higher error than completely acceptable.

# Metric Analysis diagram

In this chapter we look at different problems you will encounter when developing your models and how to spot them. We have looked at the first one: Bias, sometimes also called avoidable bias. We have seen how this can be spotted calculating . At the end of this chapter you will have a few of those quantities that you can calculate to spot problems. To make understanding them easier I use what I like to call the Metric Analysis Diagram (MAD). Is simply a bar diagram, where each bar represents a problem. Let's start building one with (for the moment) the only quantity we have discussed: bias. You can see it in Figure 6-4. At the moment it is a pretty dumb diagram, but you will see how useful is to keep things under control when we have several problems at the same time.

Figure 6-4: the Metric Analysis Diagram (MAD) with just one of the quantity we will encounter in this chapter:



# Training set Overfitting

Another problem we have discussed at length in the previous chapters is overfitting of training data. You will remember that while doing regression we saw an extreme case of overfitting in the previous chapter where . The same applies in classification problems. Let's indicate with the error our model has on our training dataset and with the one on the dev dataset. We can then say we are overfitting the training set if . Let's define a new quantity

With this quantity we can say we are overfitting the training dataset if is bigger than a few percent.

Let's summarize what we have defined and discussed so far. We have three errors:

* : the error of our classifier on the training dataset
* : human level performance (as discussed in the previous sections)
* : the error of our classifier on the dev dataset

With those three quantities we have defined

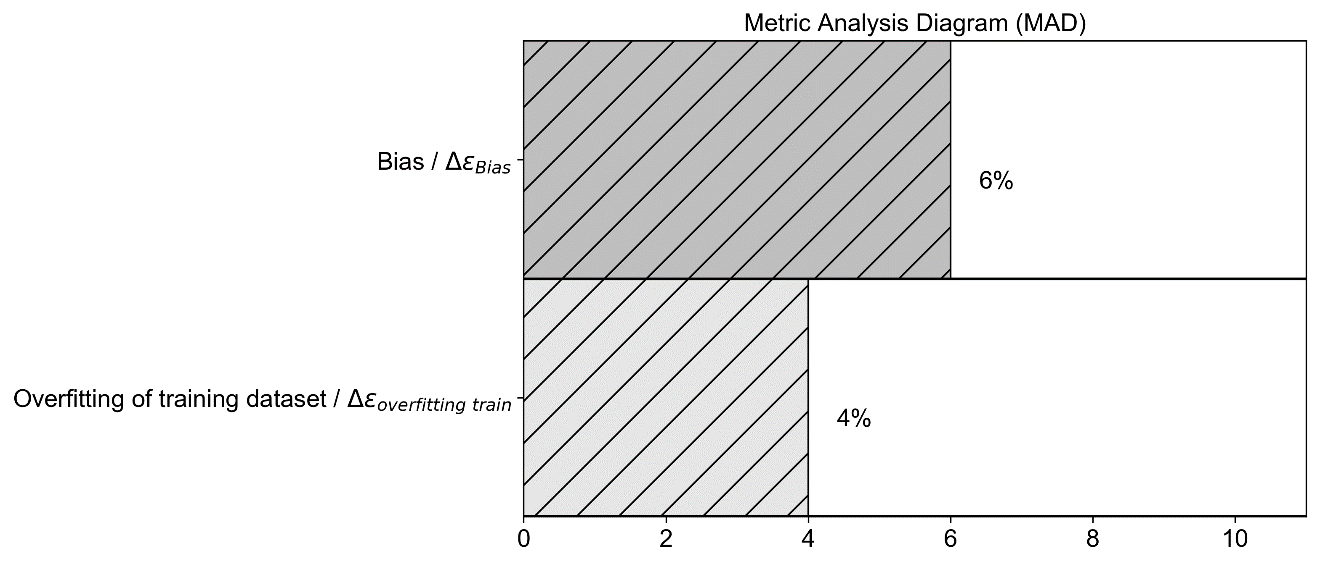
* : measuring how much "bias" we have between the training dataset and human level performance
* : measuring the amount of overfitting of the training dataset

In addition, up to now we have used two datasets

* Trainining dataset: the dataset we use to train our model (you should know it by now)
* Dev dataset: a second dataset we use to check the overfitting on the training dataset

Now let's suppose our model has bias and is slightly overfitting the training dataset, meaning we have and . Our MAD becomes now what is depicted in Figure 6-5.

Figure 6-5: the MAD diagram for our two problems: bias and overfitting of training dataset.



Now as you can see in Figure 6-5 you can have a quick overview of the relative gravity of the problems we have, and you may decide which one you want to adress first.

Usually when you are overfitting the training dataset, is commonly known as a variance problem. When this happen, you can try the following techniques to try to minimize this problem:

* Get more data for your training set
* Use regularization (check again Chapter 5 for a complete discussion of the subject)
* Try data augmentation (for example if you are working with images you can try rotating them, shifting them, etc.)
* Try "simpler" network architectures

As usual there is no fix rules and you must test which techniques work best on your problem.

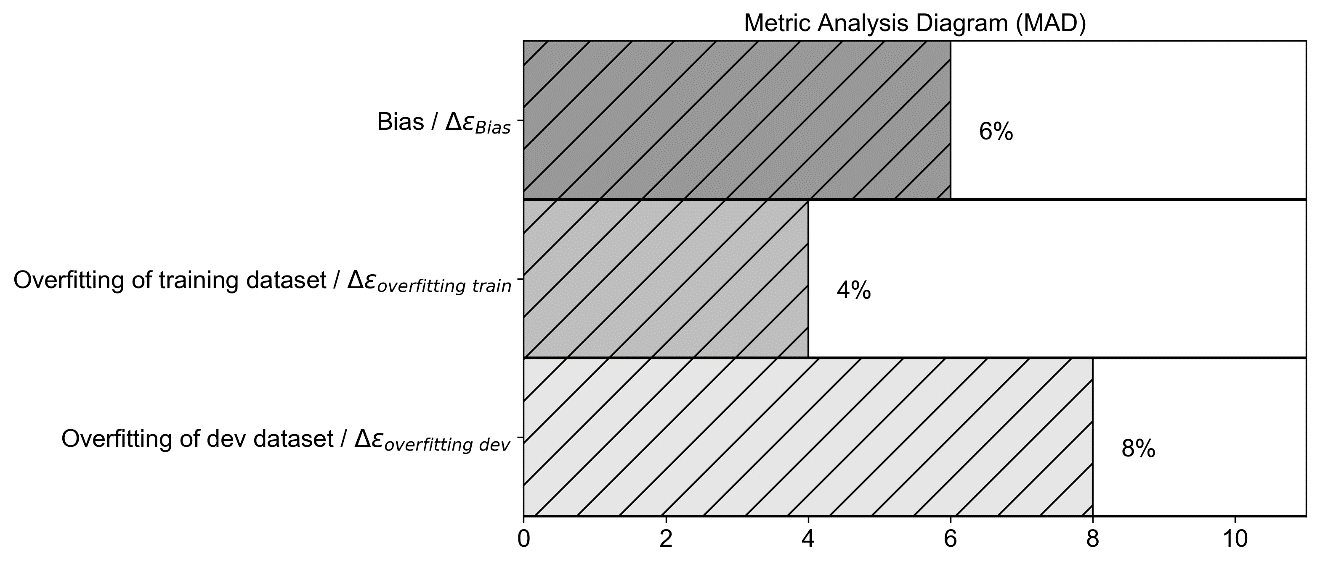
# Test set

Now I would like to quickly mention another problem you may find. We will look at it in details in Chapter 7, since is related to hyperparameter search. Let's recall how you choose the best model in a machine learning project (this is not specific to deep learning by the way). Let's suppose we are working on a classification problem. First you decide which optimizing metric you want, let's suppose you decide to use accuracy. Then you build an initial system, feed it with training data and see how it is doing on the dev dataset to check if you are overfitting your training data. You will remember that in previous chapters we have talked often about hyperparameters: parameters that are not influenced by the learning process. Examples of hyperparameters are the learning rate, regularization parameter, etc. We have seen many of them in the previous chapters. Let's say you are working with a specific neural network architecture, you need to search the best values for the hyperparameters to see how good your model can get. To do that you train several models with different values of the hyperparameters and check their performance on the dev dataset. What can happen is that your models work well on the dev dataset but don't generalize at all, since you select the best values only using the dev dataset. You incur in the risk of overfitting the dev dataset with choosing specific values of your hyperparameters. To check if this is the case, you create a third dataset, called the test dataset cutting a portion of observations from your starting dataset, that you use to check the performance of your models.

We must define a new quantity

Where is the error evaluated on the test set. We can add it to our MAD diagram.

Figure 6-6: The MAD diagram for the three problems we may encounter: bias, overfitting of training data, overfitting of dev data.



Note that if you are not doing any hyperparameter search you will not need a test dataset. Is only useful when you are doing extensive searches, otherwise in most cases is useless and take away observations that you may use for training. What we discussed so far is under the assumptions that your dev and test sets observations have the same characteristics. For example, if you are working on an image recognition problem and you decide to use images from a smartphone with high resolution for training and the dev dataset, and images from the web in low resolution for your test dataset, you may see a big but that will probably be due the differences in the images and not on an overfitting problem. We will discuss later in the chapter what can happen when different sets come from different distributions (another way of saying that the observations have different characteristics), what exactly it means and what we can do about it.

# How to split your dataset

Now I would like to discuss briefly how to split your data in a general and in the deep learning context.

But what exactly does "split" mean? Well as we have discussed in the previous section, you will need a set of observation to make the model learn, that you call your training set, then you will need a set of observations that will make your dev set and then a final set that is called the test set. Normally you will see typical splits like 60% of observations for the training set, 20% of observations for the dev and 20% of observations for test set. Usually the kind of split done is indicated in this form: 60/20/20, where the first number (60) referes to the percentage of the entire dataset that makes the training set, the second (20) to the percentage of the entire dataset that makes the dev set and the last (20) to the percentage that makse the test set. You may find in books, blogs or articles sentences like "we will split our dataset 80/10/10" for example. This is what it means.

Now usually in the deep learning field you will deal with big datasets. For example, if we have , we could use a split like 98/1/1. Keep in mind that 1% of is , so still a big number! Remember that the dev/test set must be big enough to give high confidence of the performance of the model, but not unecessarily big. Additionally, you want to save as many observations as possible for your training set.

When deciding on how to split your dataset, if you have a big number of observations (like for example or even more) you can split your dataset 98/1/1 or 90/5/5. So as soon as your dev and test dataset reach a reasonable size (that dependes on your problem) then you can stop. When deciding on how to split your dataset keep in mind how big your dev/test sets must be.

Now remember that, as you may know, size is not everything. Your dev and test datasets should be representative of your training dataset and problem. Let's make an example. Let's consider the imagenet challenge we described earlier. There you want to classify images in 1000 different classes. To know how your model is performing in your dev and test dataset you will need enough images for each class in each set. If you decide to take only 1000 observations for the dev or test dataset you are not going to get any reasonable result, since in case all classes are represented in the dev set you will only have one observation for each class. You should decide to build your dev and test dataset choosing for example 100 images for each class at least, building two datasets (dev and test) each containing observations in total (remember we have 1000 classes). In this case it would not be sensible to go below this number. This is not only relevant in the deep learning context, but in machine learning in general. One should always try to build a dev/test dataset reflecting the same distribution of observations you have in your training set. To understand what I mean take the MNIST dataset for example. Let's load the dataset (as we have done before) with the code

import numpy as np

from sklearn.datasets import fetch\_mldata

mnist = fetch\_mldata('MNIST original')

X,y = mnist["data"], mnist["target"]

total = 0

then we can check how often (in %) each digit appears in the dataset

for i in range(10):

print ("digit", i, "makes", np.around(np.count\_nonzero(y == i)/70000.0\*100.0, decimals=1), "% of the 70000 observations")

This gives us the result

digit 0 makes 9.9 % of the 70000 observations   
digit 1 makes 11.3 % of the 70000 observations   
digit 2 makes 10.0 % of the 70000 observations   
digit 3 makes 10.2 % of the 70000 observations   
digit 4 makes 9.7 % of the 70000 observations   
digit 5 makes 9.0 % of the 70000 observations   
digit 6 makes 9.8 % of the 70000 observations   
digit 7 makes 10.4 % of the 70000 observations   
digit 8 makes 9.8 % of the 70000 observations   
digit 9 makes 9.9 % of the 70000 observations

So not every digit appears the same number of times in the dataset. When building our dev and test dataset we should check that our distributions reflects this one, otherwise when applying our model to the dev or test dataset we could get a result that does not makes much sense, since the model have learned from a different class distribution. You may remember that in Chapter 5 we have created a dev dataset with a code like this one

np.random.seed(42)

rnd = np.random.rand(len(y)) < 0.8

train\_y = y[rnd]

dev\_y = y[~rnd]

In this case for the sake of clarity I just splitted the labels to see how the algorithm is working. In real life you would need to split also the features of course. Since our original distribution is almost uniform, you should expect a result that is very similar to the original one. Let's check it with the code

for i in range(10):

print ("digit", i, "makes", np.around(np.count\_nonzero(train\_y == i)/56056.0\*100.0, decimals=1), "% of the 56056 observations")

this gives us the result

digit 0 makes 9.9 % of the 56056 observations   
digit 1 makes 11.3 % of the 56056 observations   
digit 2 makes 9.9 % of the 56056 observations   
digit 3 makes 10.1 % of the 56056 observations   
digit 4 makes 9.8 % of the 56056 observations   
digit 5 makes 9.0 % of the 56056 observations   
digit 6 makes 9.8 % of the 56056 observations   
digit 7 makes 10.4 % of the 56056 observations   
digit 8 makes 9.8 % of the 56056 observations   
digit 9 makes 9.9 % of the 56056 observations

you can compare these results with the one from the entire dataset. You will notice that they are very close, not the same (compare for example digit 2), but close enough. In this case I would simply proceed without worries. But let's make a slightly different example. Let's suppose that instead of choosing randomly the observations to create your training and dev dataset you decide to take the first 80% of the observations and assign it to the training set, and the last 20% and assign it to the dev set, since you assume that your observations are randomly distributed in your original numpy arrays. Let's try and see what happens. First let's build our train and dev dataset using the first 56000 (0.8\*70000) observations for the training set and the rest for the dev set

srt = np.zeros\_like(y, dtype=bool)

np.random.seed(42)

srt[0:56000] = True

train\_y = y[srt]

dev\_y = y[~srt]

we can again check how many digits we have with the code

total = 0

for i in range(10):

print ("class", i, "makes", np.around(np.count\_nonzero(train\_y == i)/56000.0\*100.0, decimals=1), "% of the 56000 observations")

and that gives us the result

class 0 makes 8.5 % of the 56000 observations   
class 1 makes 9.6 % of the 56000 observations   
class 2 makes 8.5 % of the 56000 observations   
class 3 makes 8.8 % of the 56000 observations   
class 4 makes 8.3 % of the 56000 observations   
class 5 makes 7.7 % of the 56000 observations   
class 6 makes 8.5 % of the 56000 observations   
class 7 makes 9.0 % of the 56000 observations   
class 8 makes 8.4 % of the 56000 observations   
class 9 makes 2.8 % of the 56000 observations

Do you notice anything different? The biggest difference is that now class 9 is only appearing in 2.8% of the cases. Before was appearing in 9.9% of the cases. Apparently, our hypothesis that the classes are distributed according to a random uniform distribution was not right. This can be quite dangerous when checking how the model is doing or because your model may end up learning from a so called **unbalanced class distribution**.

Usually we talk about an unbalanced class distribution in a dataset for a classification problem when one or more classes appear a different number of times than others. Usually this becomes a problem in the learning process when the difference is significant. A few percent of difference are often not an issue.

If you have a dataset with three classes, for example, where you have 1000 observations in each class, then the dataset has a perfectly balanced class distribution, but if you have in class 1 only 100 observations, in class 2 10000 observations and in class 3 5000 then we talk about an unbalanced class distribution. You should not think that this is a rare occurrence. Suppose you need to build a model that recognize fraudulent credit card transactions. Is safe to assume that those transactions are a very small percent of the entire amount of transactions that you will have at your disposal!

When splitting your dataset, you must pay great attention not only to the number of observations you have in each dataset, but also on which observations go in each dataset. Note that this problem is not specific to deep learning but is important generally in machine learning.

To go into details on how to deal with unbalanced datasets would go beyond the scope of this book but is important to understand what kind of consquences it may have. I will show you in the next section what can happen if you feed an unbalanced dataset to a neural network so that you gain a concrete understanding of what can happen. At the end of next section, I will give you a few hints on what to do in such a case.

# Unbalanced class distribution: what can happen

Since we are talking about how to split our dataset to perform metric analysis is important to grasp the concept of unbalanced class distribution and how to deal with it. In deep learning you will find yourself very often splitting datasets and you should be aware of the problems you may encounter if you do it in the wrong way. Let me give you a concrete example of how bad things can go if you do it wrongly.

We will use the MNIST dataset and we will do basic logistic regression (as we did in Chapter 2) with a single neuron. Let's look very quickly again at how to load and prepare the data. We will do it in a similar way as in Chapter 2, apart from some modifications that I will point out to you. First, we load the data

import numpy as np

from sklearn.datasets import fetch\_mldata

from sklearn.metrics import confusion\_matrix

import tensorflow as tf

mnist = fetch\_mldata('MNIST original')

Xinput,yinput = mnist["data"], mnist["target"]

Now here comes the important part, we create a new label in this way: we assign to all observations for the digit zero the label 0, and to all other digits (1,2,3,4,5,6,7,8 and 9) the label 1 with the code

y\_ = np.zeros\_like(yinput)

y\_[np.any([yinput == 0], axis = 0)] = 0

y\_[np.any([yinput > 0], axis = 0)] = 1

now the array y\_ will contain the new labels. Note that now the dataset is heavily unbalanced. Label 0 appears roughly 10% of the cases, while label 1 appears 90% of the cases. Let's split the data randomly in a train and a dev dataset

np.random.seed(42)

rnd = np.random.rand(len(y\_)) < 0.8

X\_train = Xinput[rnd,:]

y\_train = y\_[rnd]

X\_dev = Xinput[~rnd,:]

y\_dev = y\_[~rnd]

We then normalize the training data

X\_train\_normalised = X\_train/255.0

We then transpose and prepare the tensors

X\_train\_tr = X\_train\_normalised.transpose()

y\_train\_tr = y\_train.reshape(1,y\_train.shape[0])

Then we assign proper names to the variables

Xtrain = X\_train\_tr

ytrain = y\_train\_tr

then we build our network with one single neuron, exactly as we did in Chapter 2

tf.reset\_default\_graph()

X = tf.placeholder(tf.float32, [n\_dim, None])

Y = tf.placeholder(tf.float32, [1, None])

learning\_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n\_dim]))

b = tf.Variable(tf.zeros(1))

init = tf.global\_variables\_initializer()y\_ = tf.sigmoid(tf.matmul(W,X)+b)

cost = - tf.reduce\_mean(Y \* tf.log(y\_)+(1-Y) \* tf.log(1-y\_))

training\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)

If you don't understand the code, check again Chapter 2 for more details. I expect that you now understand this simple model easily, as we have seen it several times. Then we define the function to run the model (you have seen it several times so far in the previous chapters)

def run\_logistic\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 10 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

Let's run the model with the code

sess, cost\_history = run\_logistic\_model(learning\_r = 0.01,

training\_epochs = 100,

train\_obs = Xtrain,

train\_labels = ytrain,

debug = True)

and check the accuracy with the code (all is explained at length in Chapter 2 if you don't remember)

correct\_prediction=tf.equal(tf.greater(y\_, 0.5), tf.equal(Y,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

print(sess.run(accuracy, feed\_dict={X:Xtrain, Y: ytrain, learning\_rate: 0.05}))

we get an incredible 91.2% of accuracy. Not bad right? But are we sure that the result is that good? Now let's check the confusion matrix[[1]](#footnote-1) for our labels with the code

ypred = sess.run(tf.greater(y\_, 0.5), feed\_dict={X:Xtrain, Y: ytrain, learning\_rate: 0.05}).flatten().astype(int)

confusion\_matrix(ytrain.flatten(), ypred)

When you run the code, you get the following result

array([[ 659, 4888],   
 [ 6, 50503]], dtype=int64)

Slightly more nicely formatted and with some explanatory information the matrix looks like this

|  |  |  |
| --- | --- | --- |
|  | **Predicted Class 0** | **Predicated class 1** |
| **Real class 0** | 659 | 4888 |
| **Real class 1** | 6 | 50503 |

How should we read the table? In the column "Predicted Class 0" you will see the number of observations that our model predict of being of class 0 for each real class. 659 is the number of observations our model predicts of being of class 0 and are really in class 0. 6 is the number of observations that our model predicts in class 0 but are really in class 1.

Should be easy to see now that our model predicts effectively almost all observations to be in class 1 (a total of 4888+50503 = 55391). The number of correct classified observations is 659 (for class 0) and 50503 (for class 1), for a total of 51162 observations. Since we have a total of 56056 observations in our training set, we get an accuracy of 51162/56056 = 0.912, as our tensorflow code above told us. But not because our model is good, simply because it has classified effectively all observations in class 1. We don't need a neural network in this case to reach this accuracy. What happens is that our model sees observations belonging to class 0 so rarely that they almost don't influence the learning, that is dominated by the observations in class 1.

What at the beginning seemed a nice result, turns out is a really bad one. This is an example of how bad things can go if you don't pay attention to the distributions of your classes. This of course applies not only when splitting your dataset, but in general when you approach a classification problem, regardless of the classifier you want to train (it does not apply only to neural networks).

When splitting your dataset in complex problems, you need to pay great attention not only to the number of observations you have in your datasets, but also on what observations you choose and on the distribution of the classes.

To conclude this section let me give you a few hints on how to deal with unbalanced dataset.

* Change your metric: in our example above, you may want to use something else instead of accuarcy, since it can be misleading. You could try using the confusion matrix for example, or other metrics like Precision, recall or F1 (check them if you don't know them, they are very important to know). Another important way of checking how your model is doing, and one that I suggest you strongly to learn is the ROC Curve, this will help you tremendously.
* Work with an undersampled dataset. If you have for example 1000 observations in class 1 and 100 in class 2, you may create a new dataset with 100 random observations in class 1 and the 100 you have in class 2. The problem with this method is that you usually will have a lot less data to feed to your model to train it.
* Work with an oversamples dataset. You may try to do the opposite. You may take the 100 observations in class 2 we mentioned above and just replicate them 10 times to end up with 1000 observations in class 2 (sometime called sampling with replacement).
* You can try to get more data in the class with less observations, although this is not always possible. In case of fraudolent credit card transactions you cannot go around and generate new data, unless you want to go to jail…

# Datasets with different distributions

Now I would like to discuss another terminology issue, that will lead us understanding a common problem in the deep learning world. Very often you will hear sentences like: "the sets come from different distributions". This sentence is not always easy to understand. Take for example two datasets formed by images taken with a professional DSLR, and a second one made by images taken with a crappy smartphone. In the deep learning world, we say that those two sets come from different distributions. But what is really the meaning of the sentence? The two datasets differ for various reasons: resolution of images, blurriness due to different quality of lens, amount of colors, how much is in focus, and possibly more. All those differences are what is usually meant with "distributions". Let's make another example. We could consider two datasets: one made of images of white cats and one made with images of black cats. Also, in this case we talk about different distributions. This becomes a problem when you train a model on one set and want to apply it to the other. For example, if you train a model on a set of images of white cats, you probably are not going to do very good on the dataset of black cats, since your model has never seen black cats during training.

When talking about datasets coming from different distributions, it is usually meant that the observations have different characteristics in the two datasets: black and white cats, high and low-resolution images, speech recording in italian and in german, and so on.

Since data is so precious, people often try to create the different datasets (train, dev, etc.) from different sources. For example, you may decide to train your model on a set made of images taken from the web and check how good it is on a set made of images you taken with your smartphone. This may seem a nice idea to be able to use as many data as possible, but it may give you many headaches. Let's see what happens in a real case so that you may get a feeling of the consequences of doing something like this.

Let's consider the subset of the MNIST dataset that we have used in Chapter 2, made of the two digits: 1 and 2. We will build a dev dataset coming from a different distribution, shifting a subset of the images 10 pixels to the right. We will train our model on the images as they are in the original dataset and apply the model to images shifted 10 pixels to the right and see what happens. Let's first load the data (you can check for more details Chapter 2).

import numpy as np

from sklearn.datasets import fetch\_mldata

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

from random import \*

mnist = fetch\_mldata('MNIST original')

Xinput,yinput = mnist["data"], mnist["target"]

We will do the data preparation exactly as in Chapter 2. First let's select only digit 1 and 2

X\_ = Xinput[np.any([y == 1,y == 2], axis = 0)]

y\_ = yinput[np.any([y == 1,y == 2], axis = 0)]

we have 14867 observations in our dataset. Then let's create a train and a dev dataset with our random selection (as we have done before) as in this case we have roughly the same number of ones and twos.

np.random.seed(42)

rnd\_train = np.random.rand(len(y\_)) < 0.8

X\_train = X\_[rnd\_train,:]

y\_train = y\_[rnd\_train]

X\_dev = X\_[~rnd\_train,:]

y\_dev = y\_[~rnd\_train]

Then we normalize the features

X\_train\_normalised = X\_train/255.0

X\_dev\_normalised = X\_dev/255.0

And then we transform the matrices to have them with the right dimensions.

X\_train\_tr = X\_train\_normalised.transpose()

y\_train\_tr = y\_train.reshape(1,y\_train.shape[0])

n\_dim = X\_train\_tr.shape[0]

dim\_train = X\_train\_tr.shape[1]

X\_dev\_tr = X\_dev\_normalised.transpose()

y\_dev\_tr = y\_dev.reshape(1,y\_dev.shape[0])

and finally, we shift the lables to have 0 and 1 (if you don't remember why, you can quickly check Chapter 2)

y\_train\_shifted = y\_train\_tr - 1

y\_dev\_shifted = y\_dev\_tr - 1

and finally let's give the arrays reasonable names

Xtrain = X\_train\_tr

ytrain = y\_train\_shifted

Xdev = X\_dev\_tr

ydev = y\_dev\_shifted

We can check the sizes of the arrays with the code

print(Xtrain.shape)

print(Xdev.shape)

that gives us

(784, 11893)   
(784, 2974)

We have 11893 observations in our training set and 2974 in the dev set. Now let's duplicate the dev dataset and let's shift each image to the right of 10 pixels. We can do it quickly with the code

Xtraindev = np.zeros\_like(Xdev)

for i in range(Xdev.shape[1]):

tmp = Xdev[:,i].reshape(28,28)

tmp\_shifted = np.zeros\_like(tmp)

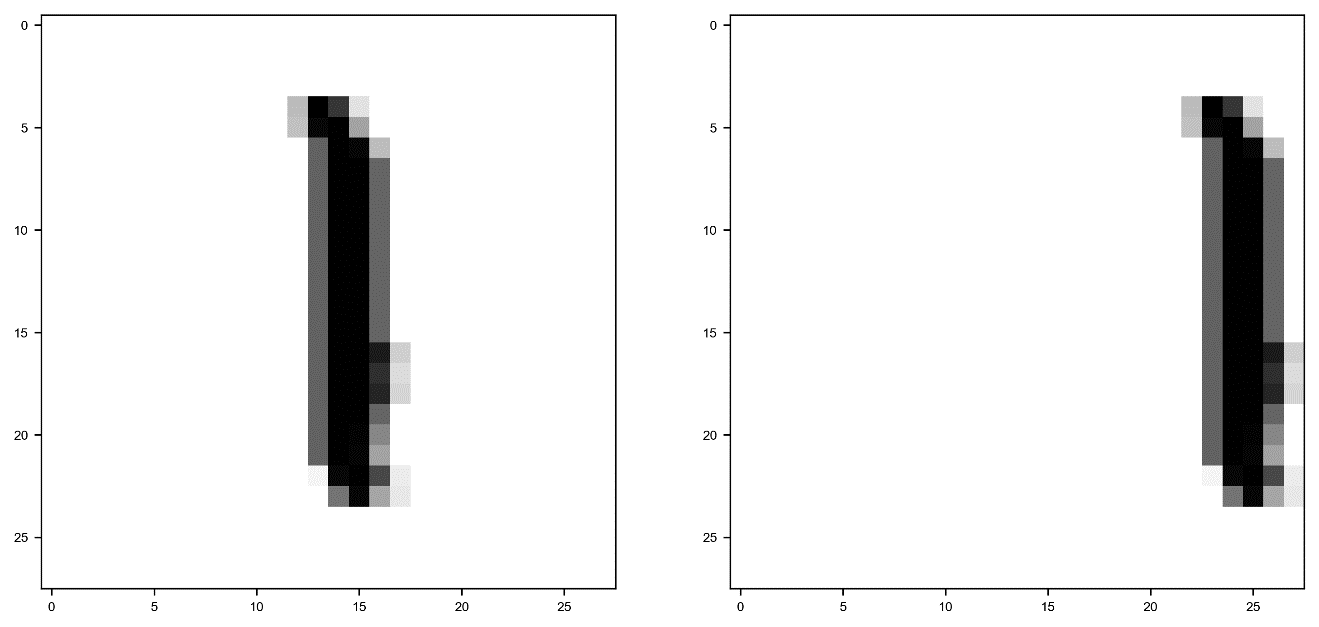
tmp\_shifted[:,10:28] = tmp[:,0:18]

Xtraindev[:,i] = tmp\_shifted.reshape(784)

ytraindev = ydev

To make the shift easy, I first reshaped the images in a 28x28 matrix, then simply shifted the columns with tmp\_shifted[:,10:28] = tmp[:,0:18] and then I simply reshape the images in a one dimensional array of 784 elements. The labels remain the same. In Figure 6-7 you can see a random image from the dev dataset on the left and its shifted version on the right.

Figure 6-7: one random image from the dataset (on the left) and its shifted version (on the right).



Now let's build a network with a single neuron and let's see what happens. We build the model as you have already seen in Chapter 2

tf.reset\_default\_graph()

X = tf.placeholder(tf.float32, [n\_dim, None])

Y = tf.placeholder(tf.float32, [1, None])

learning\_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n\_dim]))

b = tf.Variable(tf.zeros(1))

init = tf.global\_variables\_initializer()

y\_ = tf.sigmoid(tf.matmul(W,X)+b)

cost = - tf.reduce\_mean(Y \* tf.log(y\_)+(1-Y) \* tf.log(1-y\_))

training\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)

We will use the same function you have already seen to train the model

def run\_logistic\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 10 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

and we will train the model with the code

sess, cost\_history = run\_logistic\_model(learning\_r = 0.01,

training\_epochs = 100,

train\_obs = Xtrain,

train\_labels = ytrain,

debug = True)

this gives us the output

Reached epoch 0 cost J = 0.678501  
Reached epoch 10 cost J = 0.562412  
Reached epoch 20 cost J = 0.482372  
Reached epoch 30 cost J = 0.424058  
Reached epoch 40 cost J = 0.380005  
Reached epoch 50 cost J = 0.345703  
Reached epoch 60 cost J = 0.318287  
Reached epoch 70 cost J = 0.295878  
Reached epoch 80 cost J = 0.277208  
Reached epoch 90 cost J = 0.261400  
Reached epoch 100 cost J = 0.247827

Then let's calculate the accuracy for the three datasets: Xtrain, Xdev and Xtraindev wih the code

correct\_prediction=tf.equal(tf.greater(y\_, 0.5), tf.equal(Y,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

print(sess.run(accuracy, feed\_dict={X:Xtrain, Y: ytrain, learning\_rate: 0.05}))

simply using the right feed\_dict for the three datasets. We get the following results after 100 epochs

* For the training dataset we get 96.8%
* For the dev dataset we get 96.7%
* For the train-dev (you will see later why is called in this way), the one with the shifted images, we get 46.7%. A very bad result.

What has happened is that the model has learned from a dataset where all images are centered in the box and therefore could not generalize well to images shifted and no more centered.

When training a model on a dataset, you will get good results usually on observations that are like the ones in the training set. But how can you find out if you have such a problem? There is a relatively easy way of doing that, expanding our MAD diagram. Let's see how to do it.

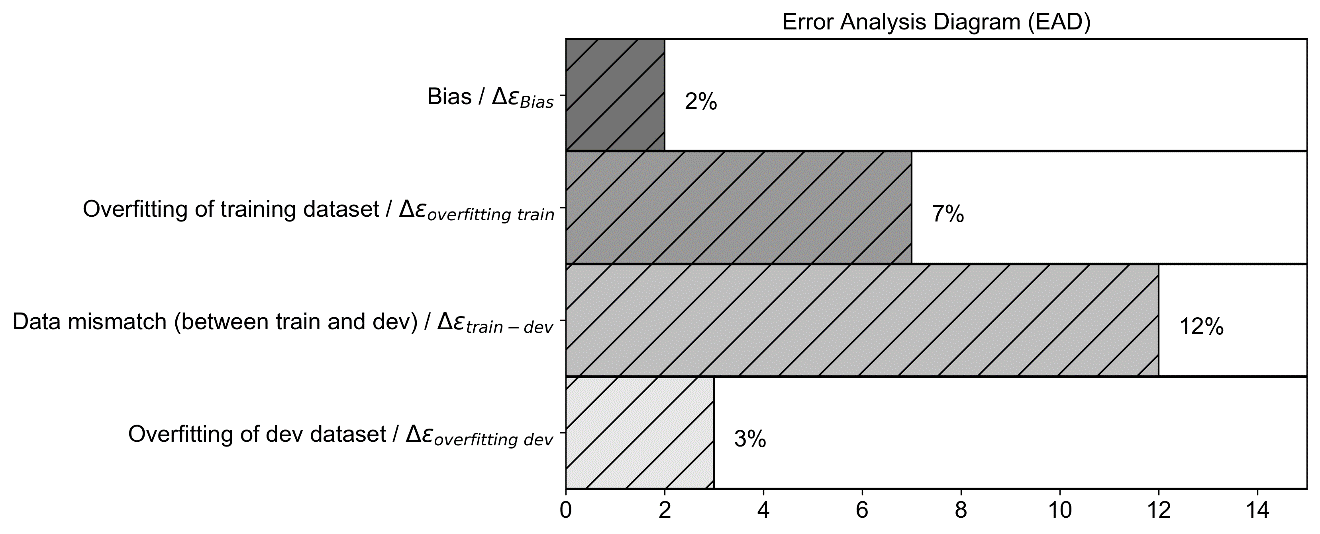
Let's suppose you have a training dataset and a dev dataset where the observations have different characteristics (coming from different distributions). What you do is you create a small subset from the training set, called the train-dev dataset, ending up with three datasets: a training and a train-dev coming from the same distribution (the observations have the same characteristics) and a dev set, where the observations are somehow different, as we have discussed previously. What you do now is you train your model on your trianing set, and then evaluate your error on the three datasets: , and . If your train and dev sets are coming from the same distributions, so does the train-dev set, in this case you should expect . If we define

we should expect . If the train (and train-dev) and the dev set are coming from different distributions (the observations have different characteristics) we should expect to be big. If we consider the MNIST example we have done before we have in fact or 43.7%, that is a huge difference. Let's recap what you should do to find our if your training and your dev (or test) dataset have observations with different characteristcs (coming from different observations)

1. You split your training set in two: one that you will use for training, and that we will call the train set, and a smaller one that you will call "train-dev" set.
2. You train your model on the train set
3. You evaluate your error on the three sets: train, dev and train-dev
4. You calculate the quantity . If is big this will give strong evidence that the original training and dev sets are coming from different distributions.

In Figure 6-8 you can see an example of the MAD diagram with added the problem we just discussed. Don't look at the numbers, they are just there for illustrative purposes (read: I just put them there).

Figure 6-8: an example of the MAD diagram with the data mismatch problem addedDon't look at the numbers, they are just there for illustrative purposes (read: I just put them there).



The MAD diagram in Figure 6-8 can tell us the following things (I highlight here only a few ideas, for a more complete list of things you can try check the previous sections):

* The bias (between training and human level performance) is quite small, so we are not that far from the best we can achieve (let's assume here that human level performance is a proxy for the Bayes error). Here you could try bigger networks, better optimizers and so on
* We are overfitting the datasets, so we could try regularization or to get more data
* We have a strong problem with data mismatch (sets coming from different distributions) between train and dev. I will suggest you at the end of the sections what you could do to solve this problem.
* We are also slightly overfitting the dev dataset, during our hyperparameter search.

Note that you don’t need to create the bar plot as I have done here. Technically you just need the four numbers to draw the same conclusions.

Once you have your MAD diagram (or simply the numbers), interpreting it will give you hints on what you should try to get better results, for example higher accuracy.

You can try the following techniques to address data mismatch between sets:

* You can carry manual error analysis to understand the difference between the sets, and then decide what to do (in the last section of the chapter I will give you an example). This is time consuming and usually quite difficult, because once you know what the difference is, it may be very difficult to find a solution;
* You could try to make the training set more similar to your dev/test sets; for example, if you are working with images and the test/dev sets have a lower resolution you may decide to lower the resolution of the images in the training set;

As usual, no fix rules. Just be aware of the problem and think about the following: your model will learn the charactersitics from your training data, so when applied to completely different data it won't (usually) do good. Always get training data that reflect the data you want your model to work on, not viceversa.

# k-fold cross validation

Now I would like to finish this chapter with another technique that is very powerful and should be known by any machine learning practictioner (not only in the deep learning world): k-fold cross validation. The technique is a way of finding a solution to the following two problems:

* What to do when your dataset is too small to split it in a train and dev/test set?
* How to get information on the variance of your metric?

Let's describe the idea with pseudo-code

1. Partition your complete dataset in k equally big subsets: , , . The subsets are also called folds. Normally the subsets are not overlapping, that means that each observation appears in one and only one fold.
2. For i going from 1 to :   
   - Train your model on all the folds except   
   - Evaluate your metric on the fold . The fold will be the dev set in iteration
3. Evaluate the average and variance of your metric on the results

A typical value for is 10, but that depends on the size of your dataset and on the characteristic of your problem.

Remember that the discussion we did on how to split a dataset applies also here.

When you are creating your folds, you must take care that your folds reflect the structure of your original dataset. If your original dataset has 10 classes for example, you must make sure that each of your folds has all the 10 classes with the same proportions.

Although this may seem a very attractive technique to deal generally with datasets with less than optimal size, it may be quite complex to implement. But, as you will see shortly, checking your metric on the different folds will give you important information on a possible overfitting of your training dataset.

Let's try it on a real dataset and let's see how to implement it. Note that you can implement k-fold cross validation easily in sklearn, but I will develop it from scratch, to show you what is happening in the background. Everyone (well, almost) can copy code from the web to implement k-fold cross validation in sklearn, but not many can explain how it works and understand it, and therefore be able to choose the right sklearn method or parameters. As a dataset we will use the same we used in Chapter 2: the reduced MNIST dataset containing only digits 1 and 2. We will do a simple logistic regression with one neuron to make the code easy to understand and to let you concentrate on the cross-validation part and not on other implementation details that are not relevant here. The goal of this section is to let you understand how k-fold cross validation is working and why is useful, not on how to implement it with the smallest number of lines of code possible.

Let's import the necessary libraries as usual

import numpy as np

from sklearn.datasets import fetch\_mldata

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

from random import \*

then let's import the MNIST dataset

mnist = fetch\_mldata('MNIST original')

Xinput\_,yinput\_ = mnist["data"], mnist["target"]

Remember that the dataset has 70000 observations and is made of gray scale images each 28x28 pixel in size. You can again check Chapter 2 for a detailed discussion. Then let's select only digit 1 and 2 and rescale the labels to make sure that digit 1 has label 0 and digit 2 has label 1. You will remember from Chapter 2 that the cost function we will use for logistic regression expect the two labels to be 0 and 1.

Xinput = Xinput\_[np.any([yinput\_ == 1,yinput\_ == 2], axis = 0)]

yinput = yinput\_[np.any([yinput\_ == 1,yinput\_ == 2], axis = 0)]

yinput = yinput - 1

We can check the number of observations with the code

Xinput.shape[0]

we have 14867 observations (images). Now we need a small trick. To keep the code simple, we want in each fold to have the same number of observations. Technically speaking this is not needed, and you will often end up with the last fold having a number of observations that is smaller than the others. In this case if we want 10 folds, we cannot have in each fold the same number of observations, since 14867 is not a multiple of 10. To make things easier let's simply remove the last 7 images from the dataset (from an estetically point of view this is horrible, but will make our code much easier to understand and to write)

Xinput = Xinput[:-7,:]

yinput = yinput[:-7]

Now let's create 10 arrays, each containing a list of indexes that we will use to select images

foldnumber = 10

idx = np.arange(0,Xinput.shape[0])

np.random.shuffle(idx)

al = np.array\_split(idx,foldnumber)

In each fold we will have, as expected, 1486 images. Now let's create the arrays containing the images

Xinputfold = []

yinputfold = []

for i in range(foldnumber):

tmp = Xinput[al[i],:]

Xinputfold.append(tmp)

ytmp = yinput[al[i]]

yinputfold.append(ytmp)

Xinputfold = np.asarray(Xinputfold)

yinputfold = np.asarray(yinputfold)

if you think this code is convoluted, you are right. There are faster ways of doing it with sklearn but is very instructive to see how to do it manually step by step, and I am convinced that the code above, where each step is well isolated, make understanding it easier. We first create empty lists: Xinputfold and yinputfold. Each element of the list will be a fold, so an array of images or of labels. So if we want to get all images in fold 2, we will simply use Xinputfold[1] (remember in Python indexes start from zero). Those list, converted with the last two lines in numpy arrays, will have 3 dimensions as you can easily see with the statements

print(Xinputfold.shape)

print(yinputfold.shape)

that gives us

(10, 1486, 784)   
(10, 1486)

In Xinputfold he first dimension indicates the fold number, the second the observation, and the third the gray values of the pixels. In yinputfold the first dimension indicates the fold number, and the second the label. For example, to get image with index 1234 from fold 0, you would need to use the code

Xinputfold[0][1234,:]

Remember you should check that we still have a balanced dataset in each fold, or in other words that we have as many ones as twos. Let's check for fold 0 (you can do the same check for the others)

for i in range(0,2,1):

print ("label", i, "makes", np.around(np.count\_nonzero(yinputfold[0] == i)/1486.0\*100.0, decimals=1), "% of the 1486 observations")

that gives us

label 0 makes 51.2 % of the 1486 observations   
label 1 makes 48.8 % of the 1486 observations

that, for our purposes, is balanced enough. Now we need to normalize the features (as we did in Chapter 2)

Xinputfold\_normalized = np.zeros\_like(Xinputfold, dtype = float)

for i in range (foldnumber):

Xinputfold\_normalized[i] = Xinputfold[i]/255.0

You could normalize the data in one shot, but I like to make it evident that we are dealing with folds to make it clear for the reader. Now let's reshape the arrays as we need them

X\_train = []

y\_train = []

for i in range(foldnumber):

tmp = Xinputfold\_normalized[i].transpose()

ytmp = yinputfold[i].reshape(1,yinputfold[i].shape[0])

X\_train.append(tmp)

y\_train.append(ytmp)

X\_train = np.asarray(X\_train)

y\_train = np.asarray(y\_train)

The code is written in the easiest way possible for instructive purposes, not in the most optimized way. now we can check the dimensions of the final arrays with

print(X\_train.shape)

print(y\_train.shape)

that gives us

(10, 784, 1486)   
(10, 1, 1486)

Exactly what we need. Now we are ready to build our network. We will use a one-neuron network for logistic regression, with the sigmoid activation function

import tensorflow as tf

tf.reset\_default\_graph()

X = tf.placeholder(tf.float32, [n\_dim, None])

Y = tf.placeholder(tf.float32, [1, None])

learning\_rate = tf.placeholder(tf.float32, shape=())

#W = tf.Variable(tf.zeros([1, n\_dim]))

W = tf.Variable(tf.random\_normal([1, n\_dim], stddev= 2.0 / np.sqrt(2.0\*n\_dim)))

b = tf.Variable(tf.zeros(1))

y\_ = tf.sigmoid(tf.matmul(W,X)+b)

cost = - tf.reduce\_mean(Y \* tf.log(y\_)+(1-Y) \* tf.log(1-y\_))

training\_step = tf.train.AdamOptimizer(learning\_rate = learning\_rate, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8).minimize(cost)

init = tf.global\_variables\_initializer()

Here we have used the Adam optimizer, but the gradient descent would work as good. This is a very easy case. We will use our well-known function to train the model

def run\_logistic\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 200 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

at this point we will need to iterate through the folds. Remember our pseudo code at the beginnig? Select one fold as dev set and train the model on all other folds concatenated. Proceed in this way for all the folds. The code could look like this (is a bit long, so take a few minutes to understand it). In the code I added comments indicating which step we are talking about, since you will find a numbered list of the steps below for explanation

train\_acc = []

dev\_acc = []

for i in range (foldnumber): # Step 1

# Prepare the folds - Step 2

lis = []

ylis = []

for k in np.delete(np.arange(foldnumber), i):

lis.append(X\_train[k])

ylis.append(y\_train[k])

X\_train\_ = np.concatenate(lis, axis = 1)

y\_train\_ = np.concatenate(ylis, axis = 1)

X\_train\_ = np.asarray(X\_train\_)

y\_train\_ = np.asarray(y\_train\_)

X\_dev\_ = X\_train[i]

y\_dev\_ = y\_train[i]

# Step 3

print('Dev fold is', i)

sess, cost\_history = run\_logistic\_model(learning\_r = 5e-4,

training\_epochs = 600,

train\_obs = X\_train\_,

train\_labels = y\_train\_,

debug = True)

# Step 4

correct\_prediction=tf.equal(tf.greater(y\_, 0.5), tf.equal(Y,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

print('Train accuracy:',sess.run(accuracy, feed\_dict={X:X\_train\_, Y: y\_train\_, learning\_rate: 5e-4}))

train\_acc = np.append( train\_acc, sess.run(accuracy, feed\_dict={X:X\_train\_, Y: y\_train\_, learning\_rate: 5e-4}))

correct\_prediction=tf.equal(tf.greater(y\_, 0.5), tf.equal(Y,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

print('Dev accuracy:',sess.run(accuracy, feed\_dict={X:X\_dev\_, Y: y\_dev\_, learning\_rate: 5e-4}))

dev\_acc = np.append( dev\_acc, sess.run(accuracy, feed\_dict={X:X\_dev\_, Y: y\_dev\_, learning\_rate: 5e-4}))

sess.close()

the code follows the steps:

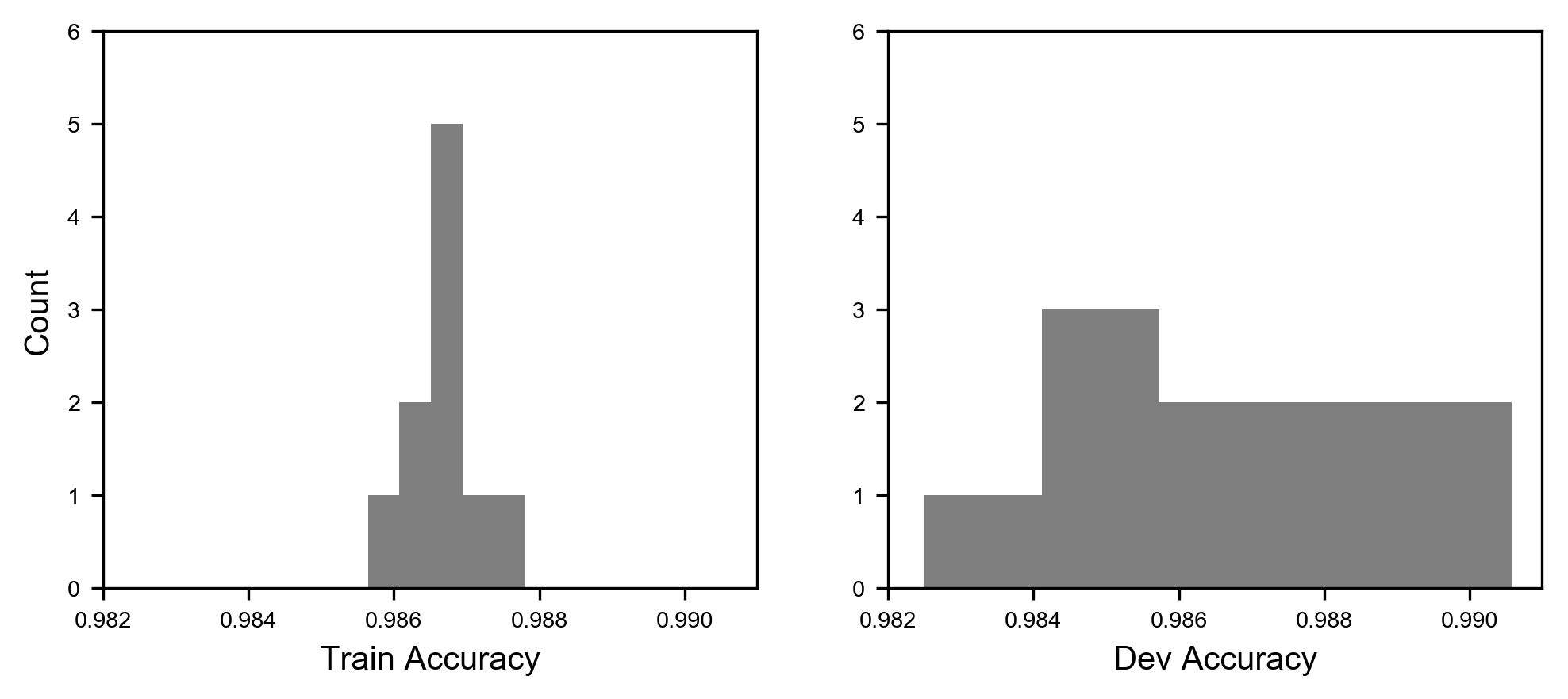
1. Do a loop over all the folds (in this case from 1 to 10) iterating with the variable i from 0 to 9
2. For each i use the fold i as the dev set and concatenate all other folds and use the result as train set.
3. For each i Train the model
4. For each i Evaluate the accuracy on the two datasets (train and dev) and save the values in the two lists: train\_acc and dev\_acc.

If you run this code, you will get an output that will look like this for each fold (you will get 10 times the following output, once for each fold)

Dev fold is 0   
Reached epoch 0 cost J = 0.766134   
Reached epoch 200 cost J = 0.169536   
Reached epoch 400 cost J = 0.100431   
Reached epoch 600 cost J = 0.074989   
Train accuracy: 0.987289   
Dev accuracy: 0.984522

You will notice that you will get for each fold slightly different accuracy values. It is very instructive to study how the accuracy values are distributed. Since we have 10 folds, we have 10 values to study. In Figure 6-9 you can see the distribution of the values for the train set (left plot) and for the dev set (right plot).

Figure 6-9: distribution of the accuracy values for the train set (left plot) and for the dev set (right plot). Note that the two plots use the same scale on both axis.



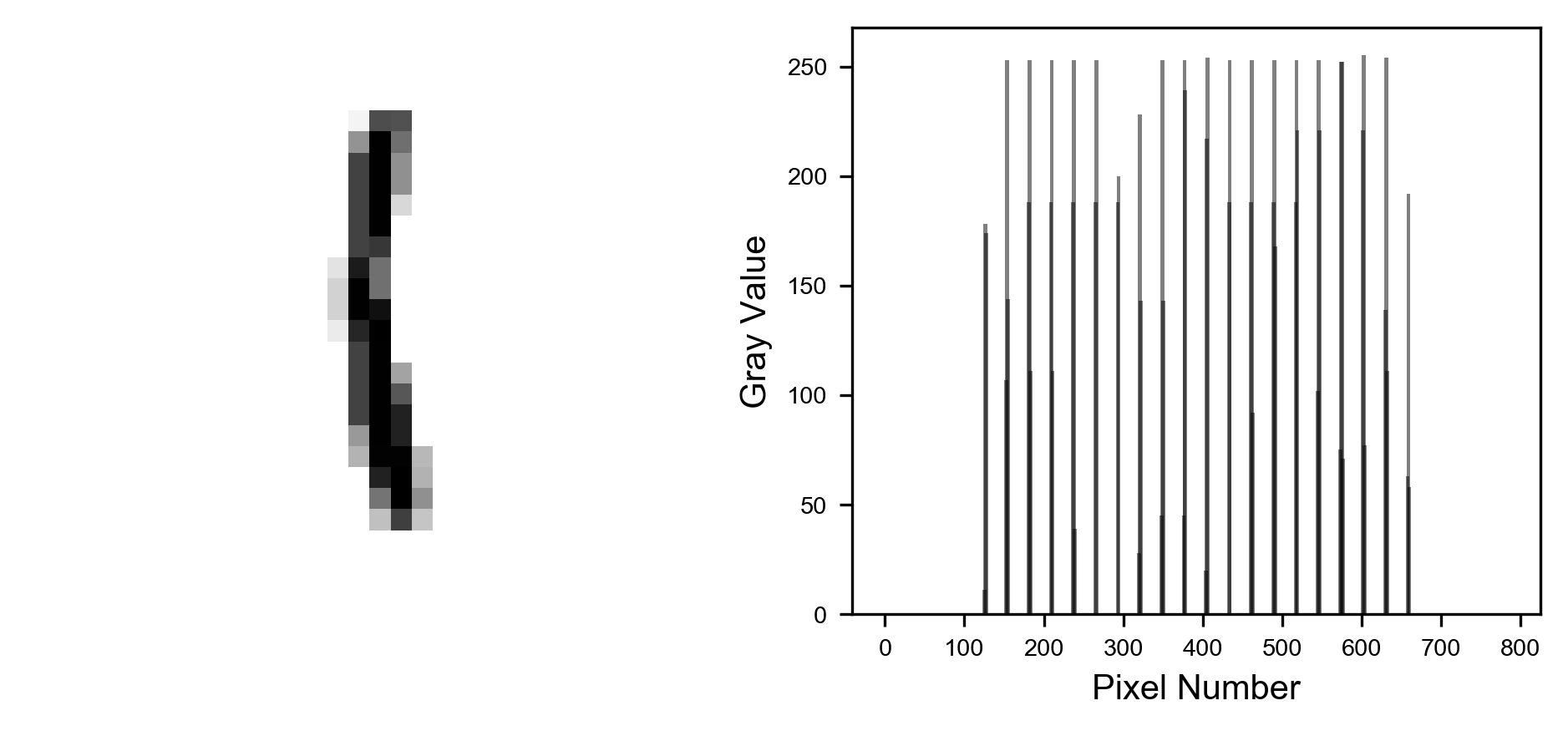
The image is quite instructive. You can see that the accuracy values for the training set are quite concentrated around the average, while the ones evaluated on the dev set are much more spread! This shows how the model on new data behave less well than on the data it has trained on. The standard deviations for the training data is and for the dev set is , 4.5 times bigger than the value on the train set. In this way you also get an estimate on the variance of your metric when applied on new data, and on how it generalizes. If you are interested in learning how to do this quickly with sklearn you can check the official documentation for the KFold method here <https://goo.gl/Gq1Ce4>. When you are dealing with datasets with many classes (remember our discussion on how to split your sets?), you must pay attention and do what is called stratified sampling[[2]](#footnote-2). Sklearn provies a method to do that too: stratifiedKFold, that can be studied here <https://goo.gl/ZBKrdt>.

You can now easily find averages and standard deviations. For the training set we have an average accuracy of 98.7% and a standard deviation of 0.054%, while for the dev set we have an average of 98.6% with a standard deviation of 0.24%. So now you can even give an estimate of the variance of your metric. Pretty cool!

# Manual Metric Analysis: an example

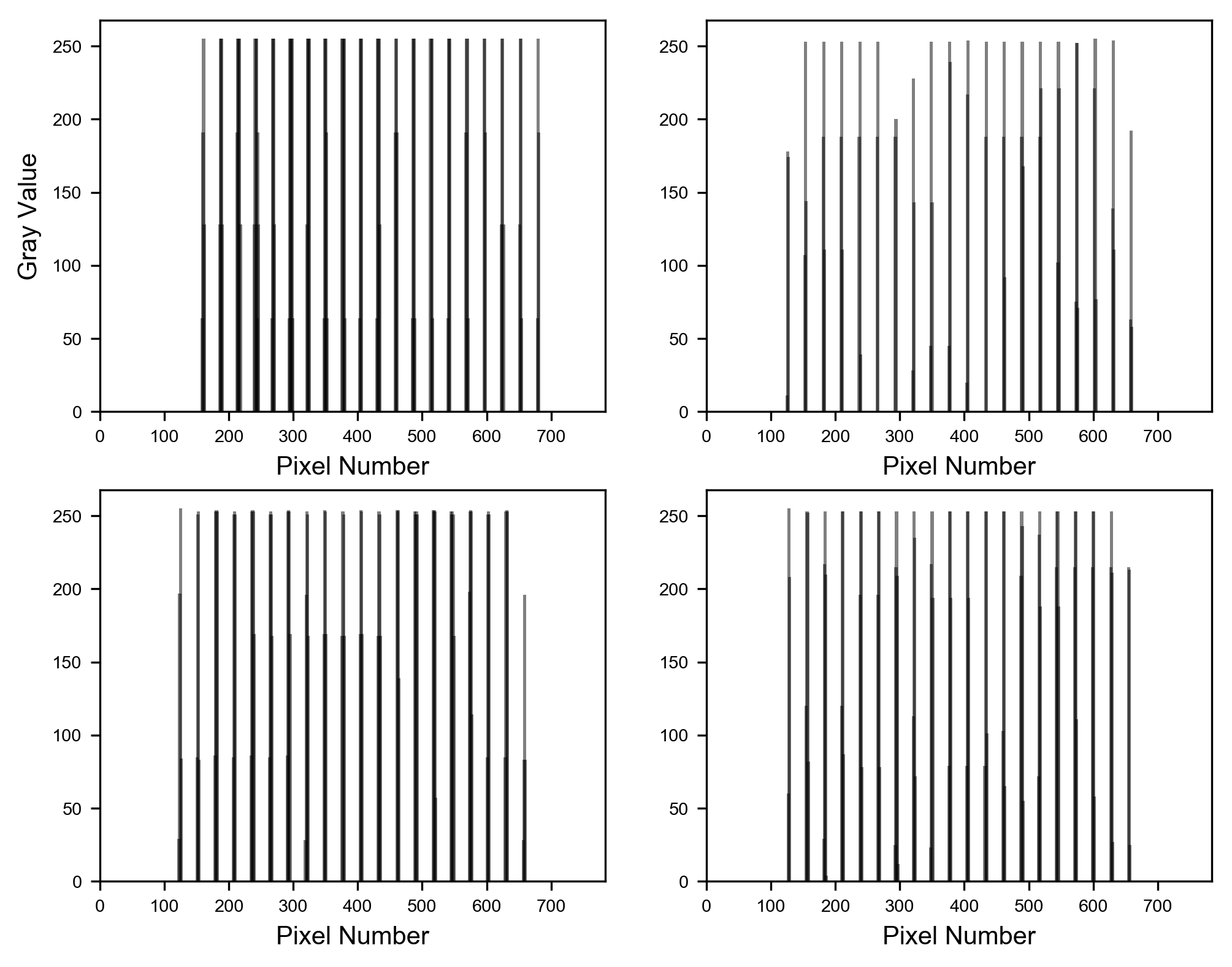
I mentioned earlier that sometime is useful to do a manual analysis of your data, to check if the results (or the errors) you are getting are plausible. I would like to give you a basic example here to give you a concrete idea of what is meant and how complicated it can be. Let's consider the following question: our very simple model (remember we are using only one neuron) can get 98% of accuracy! Is the problem of recognizing digits that easy? Let's try to see if that is the case. First of all, note that our training set does not even have the 2-dimensional information of the images. If you remember each image is converted in a one-dimensional array of values: the gray values of each pixel, starting on the top-left and going row by row from top to bottom. Are the ones and the twos so easy to recognize? Let's check how the real input for our model look like. Let's start analyzing the digit one. Let's take an example from fold 0. In Figure 6-10 you can see the image on the left and a bar plot of the gray values of the 784 pixels as they are seen from our model. Remember that as observations we have a one-dimensional array of the 784 gray values of the pixels of the image.

Figure 6-10: an example from fold 0 for the digit 1. The image on the left and a bar plot of the gray values of the 784 pixels as they are seen from our model on the right. Remember that as inputs we have a one-dimensional array of the 784 gray values of the pixels of the image.



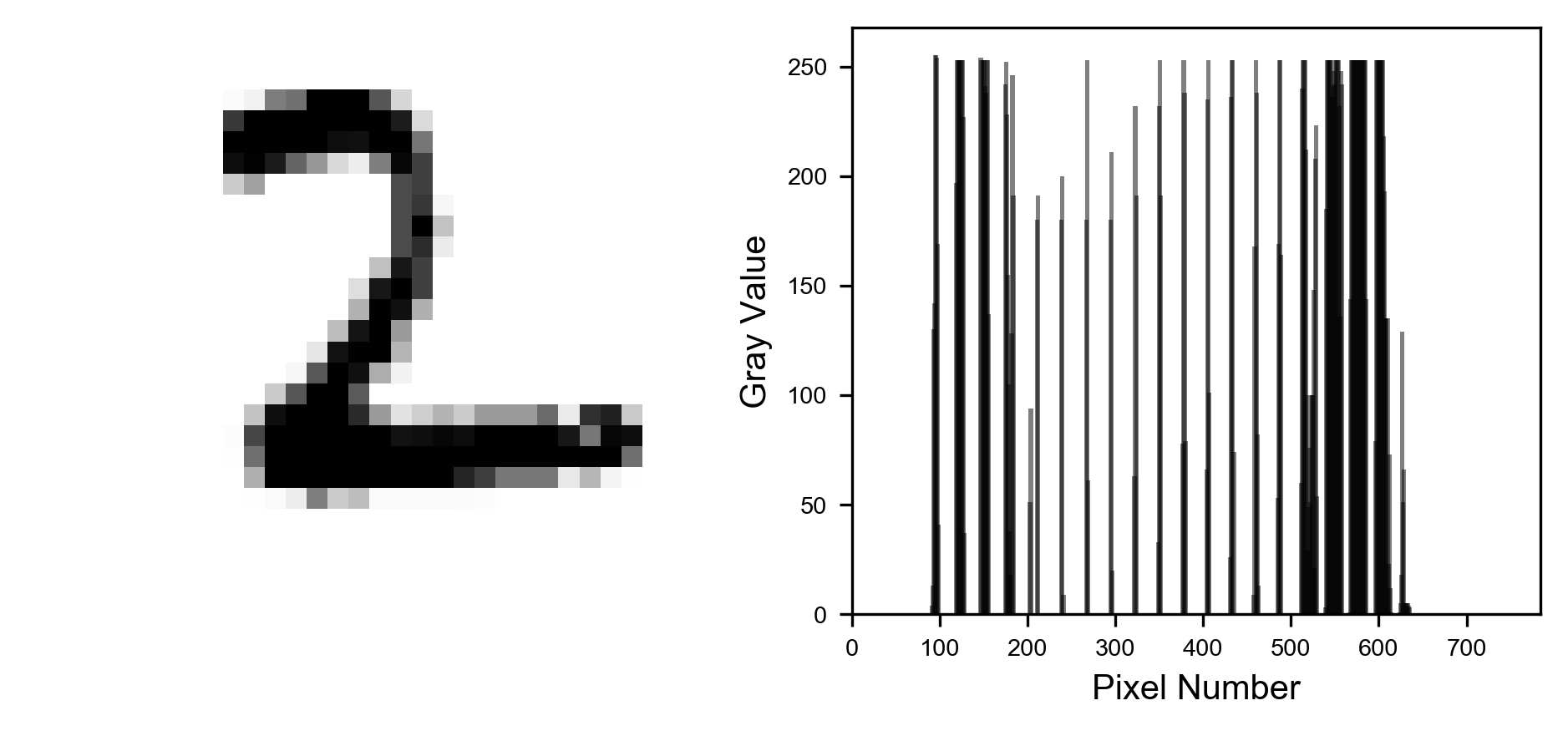
Remember that we reshape our 28x28 image in a one-dimensional array, so when reshaping the digit one in Figure 6-10 we will find black points roughly each 28 pixels, since the one is almost a vertical column of black points. In Figure 6-11 you can see other ones, and you will notice how, when reshaped as one dimensional, they look all the same: several bars roughly equally spaced. Now that you know what to look for, you can easily say that all the images in Figure 6-11 are all of digit one.

Figure 6-11: Four examples of the digit 1 reshaped as one-dimensional arrays. All look the same: a number of bars roughly equally spaced.



Now let's look at the digit two. In Figure 6-12 you can see an example, similar to what we had in Figure 6-10.

Figure 6-12: an example from fold 0 for the digit 2. The image on the left and a bar plot of the gray values of the 784 pixels as they are seen from our model. Remember that as observations we have a one-dimensional array of the 784 gray values of the pixels of the image.



Now things look different. We have two regions where the bars are much more dense in the plot on the right in Figure 6-12. Between pixels 100 and 200 and especially after pixel 500. Why? Well the two areas correspond to the two horizontal parts of the image. In Figure 6-13 I highlighted how different parts look like when reshaped as one-dimensional array.

Figure 6-13: How different parts of the images look like when reshaped as one-dimensional array. Horizontal parts are labelled as (A) and (B), while the more vertical part is labelled as (C).

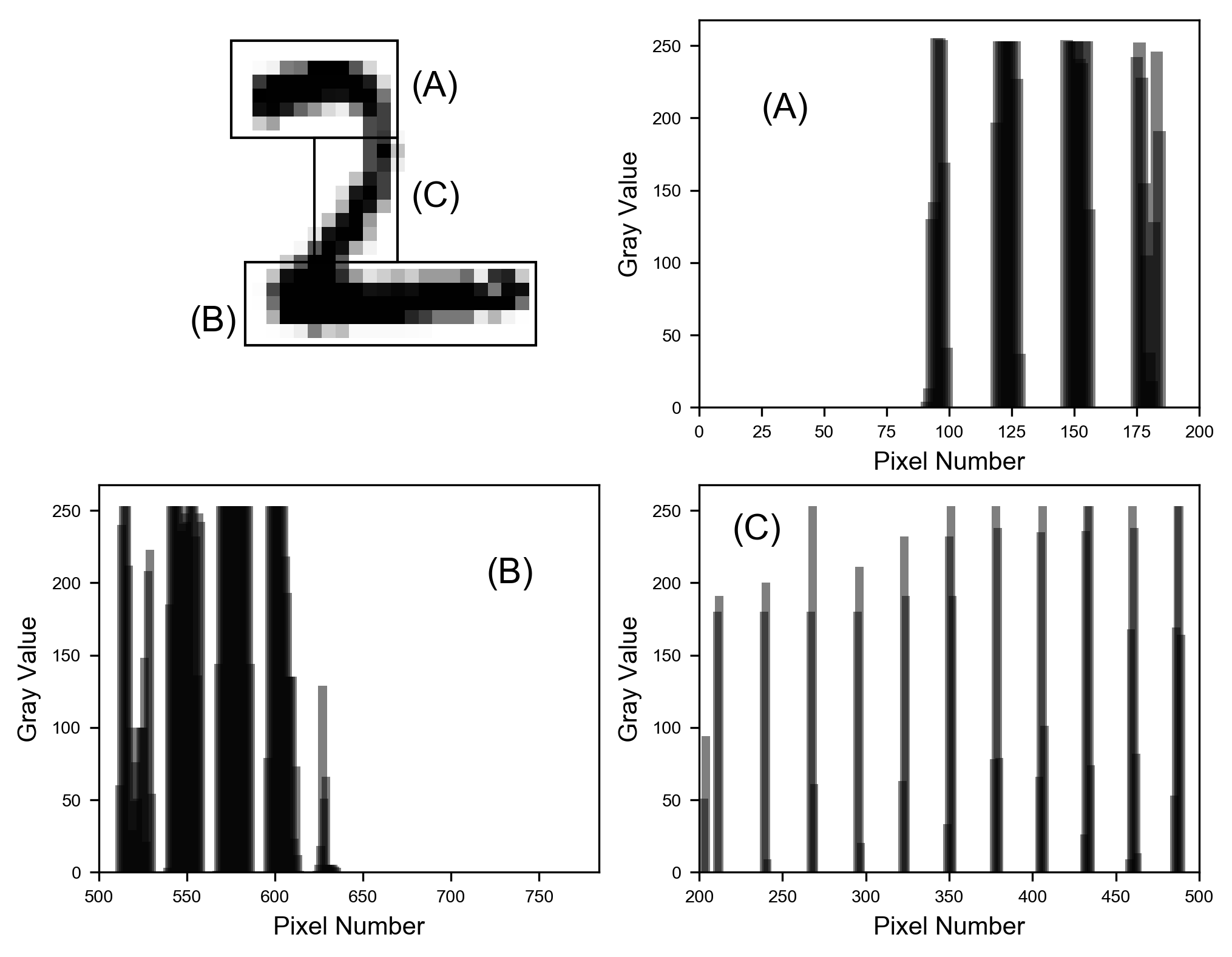
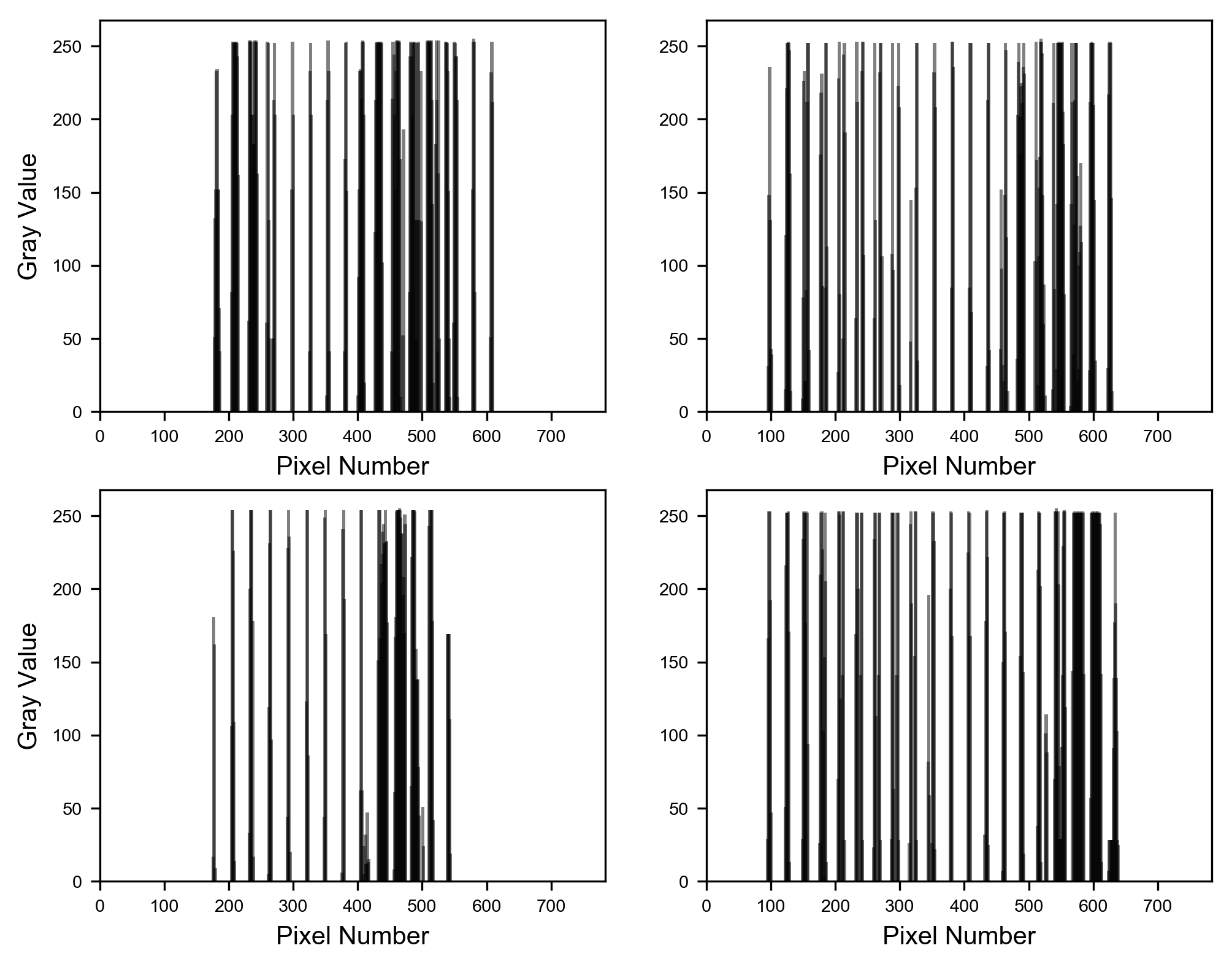
Horizontal parts (A) and (B) are clearly different than part (C) when reshaped as a one-dimensional array. The vertical part (C) looks like the digit one, many equally spaced bars, as you can check in the lower right bar plot labelled as (C). While the more horizontal parts appear as many bars clustered together in groups, as can be seen in the upper right and lower left bar plots labelled as (A) and (B). So, when reshaped, if you find those cluster of bars you are looking at a two, if you see only equally spaced small groups of bars, as in the plot (C) in Figure 6-13, you are looking at a one. You don't even need to see the two-dimensional image if you know what to look for. Note that this pattern is very constant. In Figure 6-14 you can see four examples of the digit 2, and you can clearly see the wider clusters of bars.

Figure 6-14: Four examples of the digit 2 reshaped as one-dimensional arrays. The wider clusters of bars can be clearly seen.

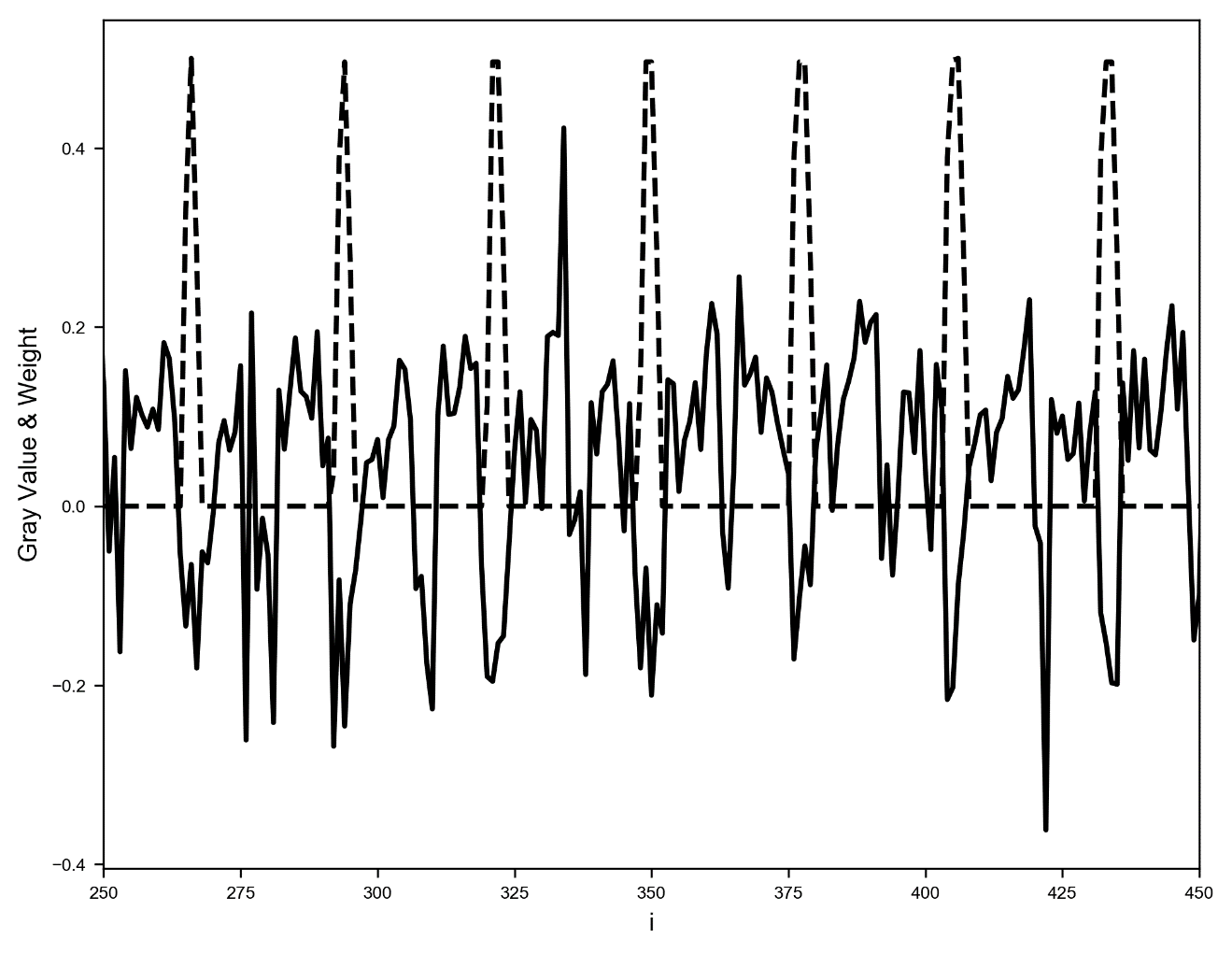


As you can imagine this is an easy pattern to spot for an algorithm, and so is to be expected that our model work so good. Even a human can spot the images, even when reshaped, without any effort. Such a detailed analysis would not be necessary in a real-life project probably but is really instructive to see what you can learn from your data. Understanding the characteristics of your data may help you in designing your model or understanding why is not working. Advanced architectures, like convolutional networks, will be able to learn exactly those two-dimensional features in a very efficient way.

Let's also check how the network learned to recognize digits. You will remember that the output of our neuron is

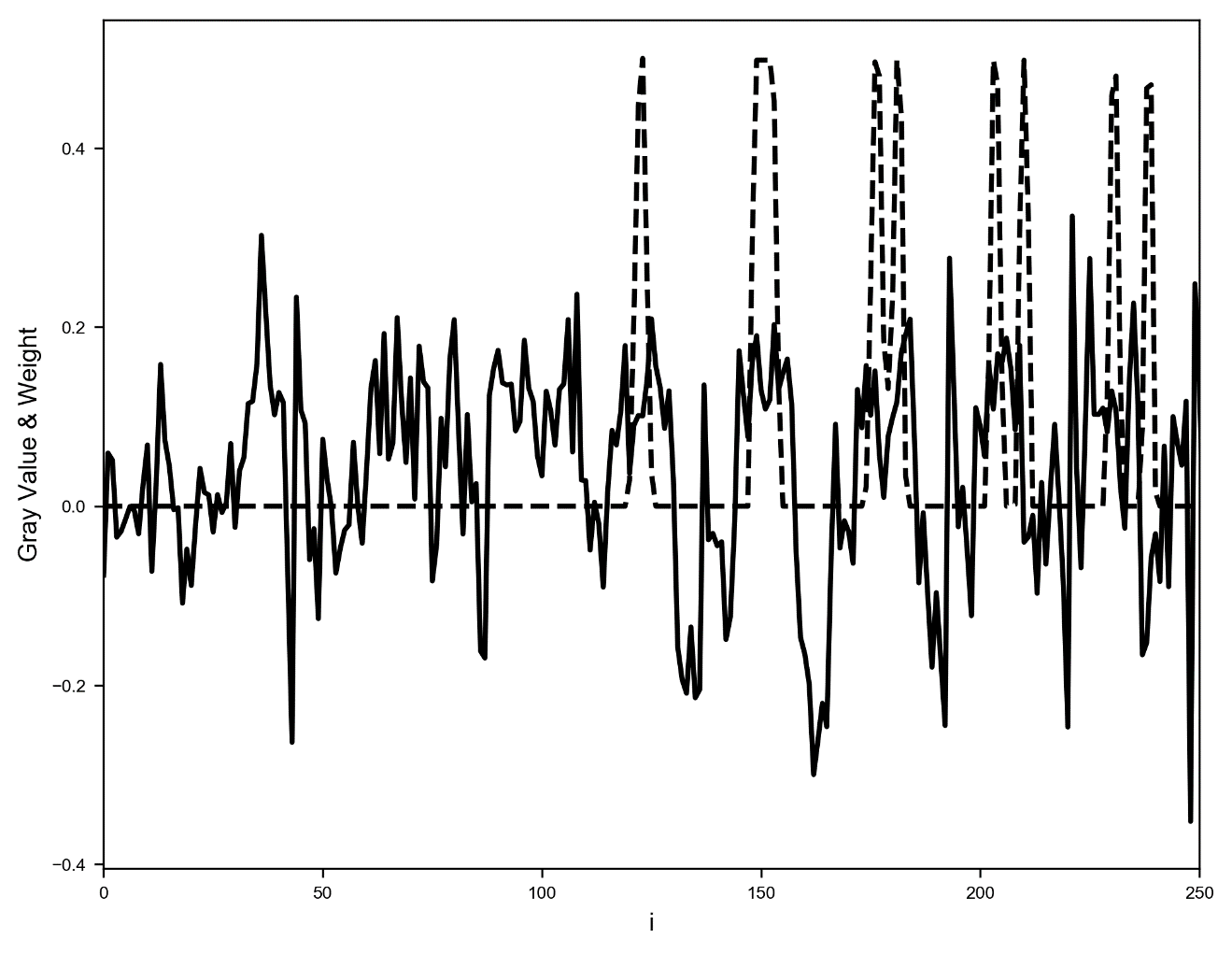
Where is the sigmoid function, for are the gray values of the pixel of the image, for are the weights and is the bias. Remember that when we classify the image in class 1 (so digit 2), and if we classify the image in class 0 (so digit 1). Now if you remember our discussion in Chapter 2 of the sigmoid function you will remember that when and for That means that our network should learn the weights in such a way that for all the ones we have and for all the twos Let's see if that is really the case. In Figure 6-15 you can see a plot for a digit 1 where you can find the weights (as a solid line) of our trained network after 600 epochs (and after reaching an accuracy of 98%) and the gray value of the pixel rescaled to have a maximum of 0.5 (as a dashed line). Look how each time is big, then is negative. And when the are almost zero. Clearly the result will be negative, and therefore and the network will identify the image as a one. In the image I zoomed in to make this behaviour more evident.

Figure 6-15: plot for a digit 1 where you can find the weights (as a solid line) of our trained network after 600 epochs (and after reaching an accuracy of 98%) and the gray value of the pixel rescaled to have a maximum of 0.5 (as a dashed line)



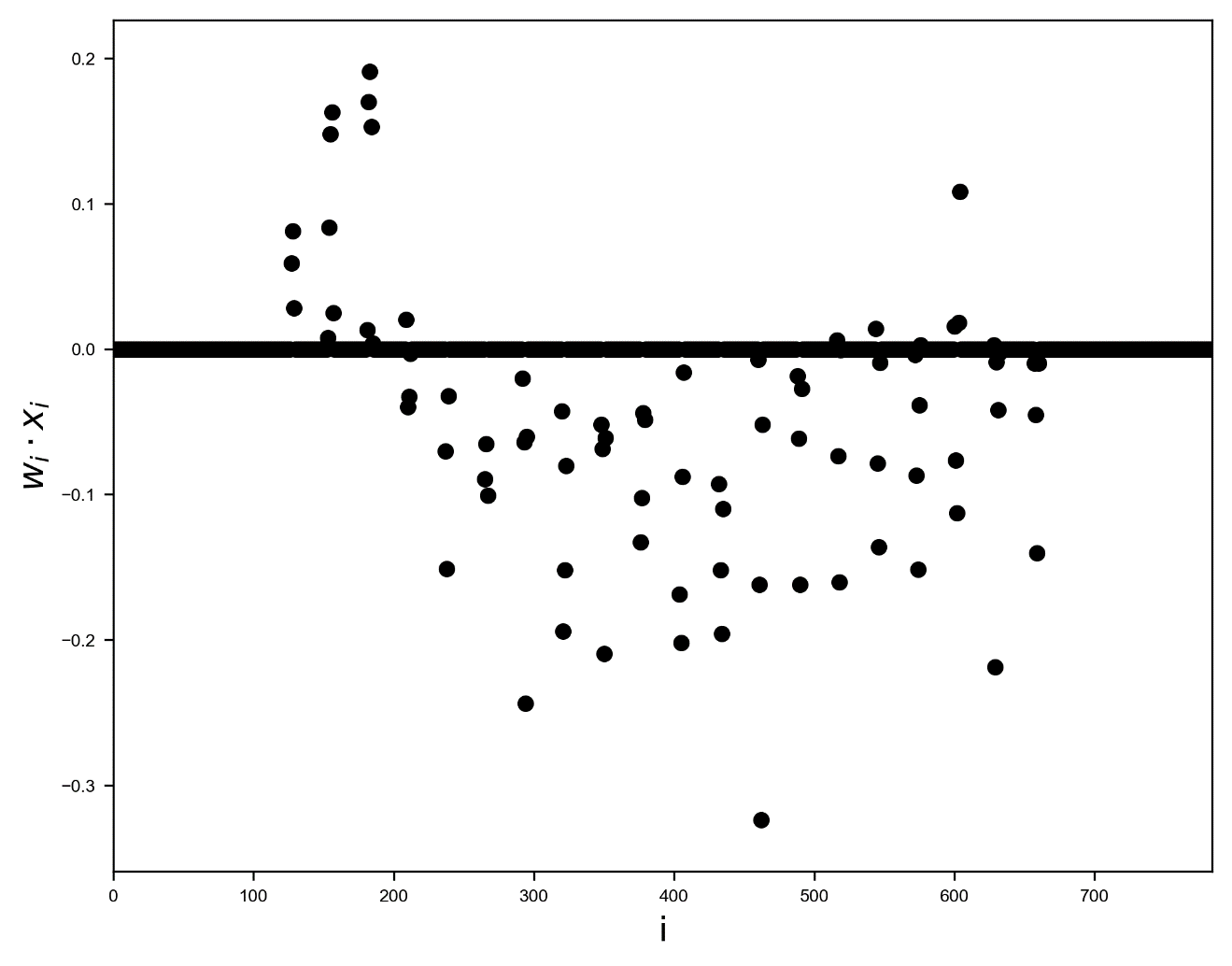
In Figure 6-16 you can see the same plot for a digit 2. You will remember from our previous discussion that for a two we can see many bars clustered together in groups up to pixel 250 (roughly). Let's check how are the weight in that region. Now you will see that where the pixel gray values are big, the weights are positive, giving then a positive value of and therefore and so the image would be classified as a two.

Figure 6-16: plot for a digit 2 where you can find the weights (as a solid line) of our trained network after 600 epochs (and after reaching an accuracy of 98%) and the gray value of the pixel rescaled to have a maximum of 0.5 (as a dashed line)



As an additional check I plotted for all values of for a digit 1 in Figure 6-17. You can see how almost all points lie below zero. Note also that in this case.

Figure 6-17: for for a digit 1. You can see how almost all values lie below zero. The thick line at zero is made of all the points such that .



As you can see in very easy cases it is possible to understand how a network learn and therefore it would be much easier to debug strange behaviours. But don't expect this to be possible when dealing with much more complex cases. The analyis we have done would not be so easy for example if you tried to do the same with digits 3 and 8 instead of 1 and 2.

1. The confusion matrix is, in Machine Learning classification, a matrix where each column of the matrix represents the number of instances in a predicted class while each row represents the number of instances in an actual class. [↑](#footnote-ref-1)
2. From wikipedia (https://goo.gl/Wd8fuD): In [statistical surveys](https://en.wikipedia.org/wiki/Statistical_survey), when [subpopulations](https://en.wikipedia.org/wiki/Subpopulation) within an overall population vary, it is advantageous to sample each subpopulation (stratum) independently. Stratification is the process of dividing members of the population into homogeneous subgroups before sampling. [↑](#footnote-ref-2)