Hands-on on Hyperparameter Tuning

Hyperparameters can be loosely defined as those parameters that are not changed during the training. For example, number of layers in a FFNN, number of neurons in each layer, activation functions, learning rate[[1]](#footnote-1) and so on.

In this chapter we will look at the problem of finding the best hyperparameters to get the best results from your models. Doing this is called hyperparameter tuning. We will first describe what a black-box optimization problem is, and how those classes of problems relate to hyperparameter tuning. We will look at the three most known methods to tackle those kind of problems: **grid search**, **random search** and **Bayesian optimization**. You will understand, with examples, which one works in which conditions, and you will learn a few tricks that are very helpful as coarse to fine optimization and sampling on a logarithmic scale. At the end of the chapter, you will see how you can use those techniques to tune a deep model with a real-world dataset (the Zalando dataset). The goal of this chapter is not to give you an overview of all possible Python libraries that can help you with hyperparameter tuning, but at the end of the chapter, you should know what hyperparameter tuning is, why is important and how it works, at least in its most fundamental form.

# Black box optimization

The problem of hyperparameter tuning is just a subclass of a much more general type of problems: **black-box optimization**. A black-box function

is a function whose analytic form is unknown. A black-box function can be evaluated to obtain its value for all values of on which is defined, but no other information, like its gradient, can be obtained. Generally, we talk about **global optimization of a black-box function** (sometimes called a black-box problem) when we try to find the maximum or minimum of given certain constraints. Here are some examples of this kind of problems:

* Finding the hyperparameter for a given machine learning model that maximize the chosen optimizing metric
* Finding maximum or minimum of a function, that can only be evaluated numerically or with code that we cannot look at. In some industry contexts sometime, there is legacy code that is very complicated, and there are some functions that must be maximized based on its outcome
* Finding the best place to drill for oil. Where your function would be how much oil you can find, and your is your location
* Finding the best combination of parameters for situations that are too complex to model, for example when launching a rocket in space how to optimize amount of fuel, diameter of each stage of the rocket, precise trajectory, etc.

This is a very fascinating class of problems, that has produced smart solutions. We will look at three of them: grid search, random search and Bayesian optimization.

Why finding the best hyperparameters for neural networks is a black-box problem? Since we cannot calculate information like the gradients of our network output with respect to the hyperparameters (for example the number of layers in a FFNN), especially when using complex optimizers or custom functions, we need other approaches to be able to find the best hyperparameters that maximize the chosen optimizing metric. Note that if we could have the gradients, we could use an algorithm as gradient descent to find the maximum or minimum.

**Note** Our black-box function will be our neural network model (including things like the optimizer, cost function form, etc.) that gives as output our optimizing metric given the hyperparameters as input. In this case will be the array containing the hyperparameters.

The problem may seem quite trivial, why not try all the possibilities? Well, this may be possible in the examples we have looked at in the past chapters, but if you are working on a problem and training your model takes a week this may present a challenge. Since typically you will have several hyperparameters, trying all possibilities will not be feasible. Let make an example to understand it better. Let's suppose we are training a model of a neural network with several layers. We may decide to consider the following hyperparameters to see which combination works better:

* Learning rate: let's suppose we want to try the values for (100 values)
* Regularization parameter: 0, 0.1, 0.2, 0.3, 0.4 and 0.5 (6 values)
* Choice of optimizer: GD, RMSProp or Adam (3 values)
* Number of hidden layers: 1,2,3,5 and 10 (5 values)
* Number of neuron in the hidden layers: 100, 200 and 300 (3 values)

Consider that you will need to train your network

times if you want to test all possible combinations. If your training takes 5 minutes you will need 13.4 weeks of computing time. If the training takes hours or days, you will not have any chance. If the training takes one day for example, you will need 73.9 years to try all possibilities. Most of the hyperparameter choices will come from experience, for example you can safely always use Adam, since is the better optimizer out there (in almost all cases). But you will not be able to avoid trying to tune other parameters like number of hidden layers or learning rate. You can reduce the number of combinations you need with experience (as with the optimizer), or with some smart algorithm, as we will see later in this chapter.

## Notes on black-box functions

Black-box functions are usually classified in two main classes:

* **Cheap functions**: functions that can be evaluated thousands of times
* **Costly functions**: functions that can only be evaluated few times, usually less than 100 times

If the black-box function is cheap then the choice of the optimization method is not critical. For example, we can evaluate the gradient with respect to the numerically, or simply search the maximum evaluating the functions on a high number of points. If the function is costly, we need much smarter approaches. One of these is Bayesian optimization, which we will discuss later in this chapter to give you an idea on how those methods work and how complex they are.

**Note** Neural networks are almost always, especially in the deep learning world, costly functions.

For costly functions we need to find methods that solve our problem with the smallest number of evaluation possible.

# The problem of hyperparameter tuning

Before looking at how we can find the best hyperparameters we need to quickly go back to neural networks and discuss what can we tune in deep models. Typically, when talking about hyperparameters, beginners think only of numerical parameters, like the learning rate or regularization parameter for example. Remember that also the following can be varied to see if you can get better results:

* Number of epochs: sometimes simply training your network longer will give you better results
* Choice of optimizer: you can try choosing a different optimizer. If you are using plain gradient descent, you may try Adam and see if you get better results
* Varying the regularization method: as we discussed previously, there are several ways of applying regularization. Varying the method may well be worth trying
* Choice of activation function: although the activation function always used in the past chapters for neurons in hidden layers was ReLU, others may work a lot better. Trying for example sigmoid or Swish may help you get better results
* Number of layers and number of neurons in each layer: try different configurations. Try layers with different number of neurons for example
* Learning rate decay methods: try (if you are not using optimizers that do that already) different learning rate decay methods
* Mini-batch size: vary the size of mini-batches. When you have few data you can use batch gradient descent, when you have a lot of data, mini-batches are more efficient
* Weight initialization methods

Let us classify the parameters we can tune in our models in the following three categories:

1. Parameters that are continuous real numbers, or in other words that they can assume any value. Example: learning rate, regularization parameter
2. Parameters that are discrete but can theoretically assume an infinite number of values. Example: number of hidden layers, number of neurons in each layer or number of epochs
3. Parameters that are discrete and can only assume a finite number of possibilities. Example: optimizer, activation function, learning rate decay method

For category 3 there is not much to do except trying all possibilities. They typically will change completely the model itself, and therefore is impossible to model their effect, therefore making a test the only possibility. This is also the category where experience may help the most. Is widely known that Adam optimizer is almost always the best choice, for example, so you may concentrate your effort somewhere else at the beginning. For categories 1 and 2 is a bit more difficult, and we will need to find some smart ideas to find the best values.

# Sample black-box problem

To try our hands at solving a black-box problem let us create a "fake" black-box problem. The problem is the following: find the maximum of the function given by the formula

pretending not to know the formula itself. The formula will let us check our results, but we will pretend is unknown. You may wonder why such a complicated formula. I wanted to have something with a few maxima and minima to give an idea how the methods are working on a non-trivial example. can be implemented in Python with the code

def f(x):

tmp1 = -np.cos(x/4.0)-np.sin(x/4.0)

-2.5\*np.cos(2.0\*x/4.0)+0.5\*np.sin(2.0\*x/4.0)

tmp2 = -np.cos(x/3.0)-np.sin(x/3.0)

-2.5\*np.cos(2.0\*x/3.0)+0.5\*np.sin(2.0\*x/3.0)

return 10.0+tmp1+0.5\*tmp2

In Figure 7-1 you can see how looks like.

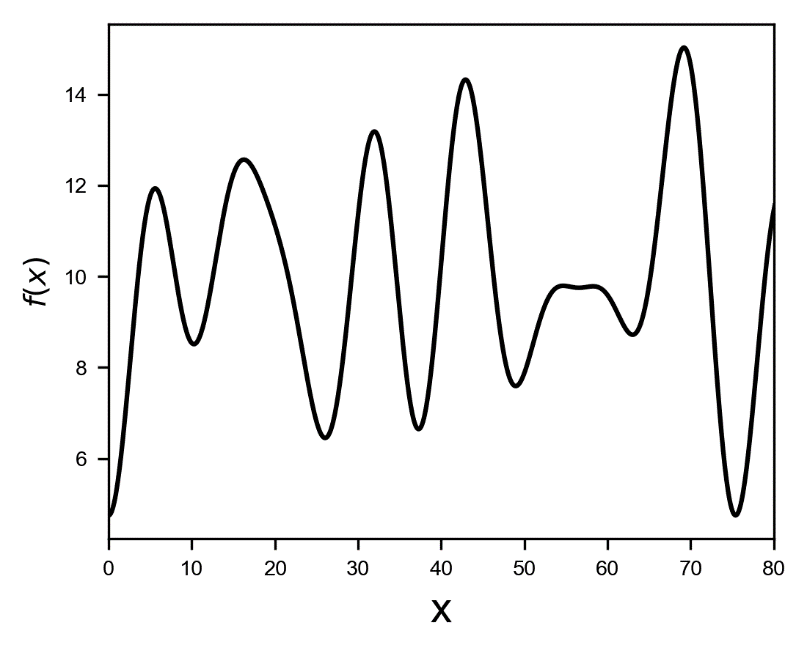


Figure 7-1. A plot of the function as described in the text.

The maximum is at an approximate value and has a value of . Our challenge is to find this maximum in the most efficient way possible, without knowing anything about except its value at any point we want. When we say "efficient" we mean of course with the smallest number of evaluations possible.

## Grid Search

The first method we look at, grid search, is also the less "intelligent". Grid search entails simply trying the function at regular intervals and seeing for which the function assumes the highest value. In this example we want to find the maximum of the function between two values and . What we would do is simply take points equally spaced between and and evaluate the function at those points. We would define a vector of points

where we defined . Then we evaluate the function at those points, obtaining a vector of values

the estimate of the maximum will then be

and supposing the maximum is found at we will also have

Now, as you may imagine the more points you use the more accurate will be your maximum estimation. The problem is that, if the evaluation of is costly, you will not be able to take as many points as you might like. You will need to find a balance between number of points and accuracy. Let us make an example with the function we have described earlier. Let us consider and and let us take points. We will have . We can create the vector easily in Python with the code:

gridsearch = np.arange(0,80,2)

the array gridsearch will look like this

array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78])

In Figure 7-2 you can see the function as continuous line, the crosses mark the points we sample in the grid search, and the black square mark the precise maximum of the function. The right plot shows a zoom around the maximum.

*Figure 7-2: the function on the range . The crosses mark the point we sample in the grid search, and the black square mark the maximum.*

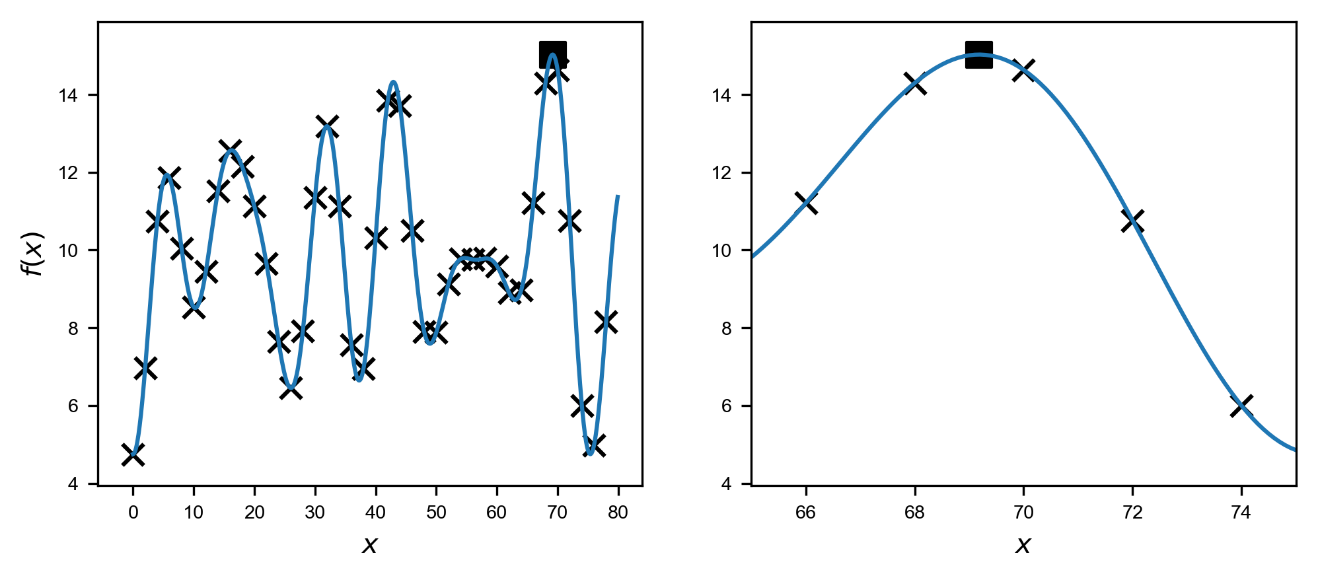


Figure 7-2. The function on the range . The crosses mark the point we sample in the grid search, and the black square mark the maximum.

You can see how the points we sample in Figure 7-2 get close to the maximum, but don’t get it exactly. Of course, sampling more points would get us closer to the maximum, but would cost us more evaluations of . We can find the maximum easily with the trivial code

x = 0

m = 0.0

for i, val in np.ndenumerate(f(gridsearch)):

if (val > m):

m = val

x = gridsearch[i]

print(x)

print(m)

that gives us

70  
14.6335957578

Close to the actual maximum but not quite right. Let us try the previous example varying the number of points we sample and let us see what results we get. We will vary the number of points sampled from 4 to 160. For each case we will find the maximum and its location as described earlier. We can do it with the code

xlistg = []

flistg = []

for step in np.arange(1,20,0.5):

gridsearch = np.arange(0,80,step)

x = 0

m = 0.0

for i, val in np.ndenumerate(maxim(gridsearch)):

if (val > m):

m = val

x = gridsearch[i]

xlistg.append(x)

flistg.append(m)

in the lists xlistg and flistg we will find the position of the maximum found and the value of the maximum for the various values of .

In Figure 7-3 we plot the distributions of the results. The black vertical line is the correct value of the maximum.

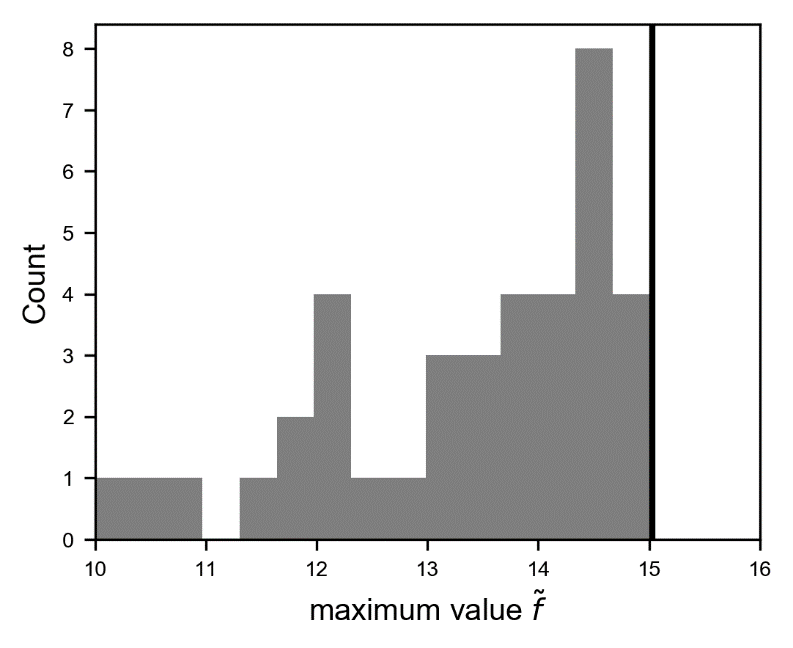


Figure 7-3. The distribution of the results for obtained by varying the number of points sampled in the grid search. The black vertical line indicates the real maximum of .

As you can see the results vary quite a lot, and can be very far from the correct value, as far as 10. This tells us that using the wrong number of points can get you very wrong results. As you can imagine the best results are the ones with the smallest step , since is more probable to get closer to the maximum. In Figure 7-4 you can see how the value of the found maximum varies with the step .

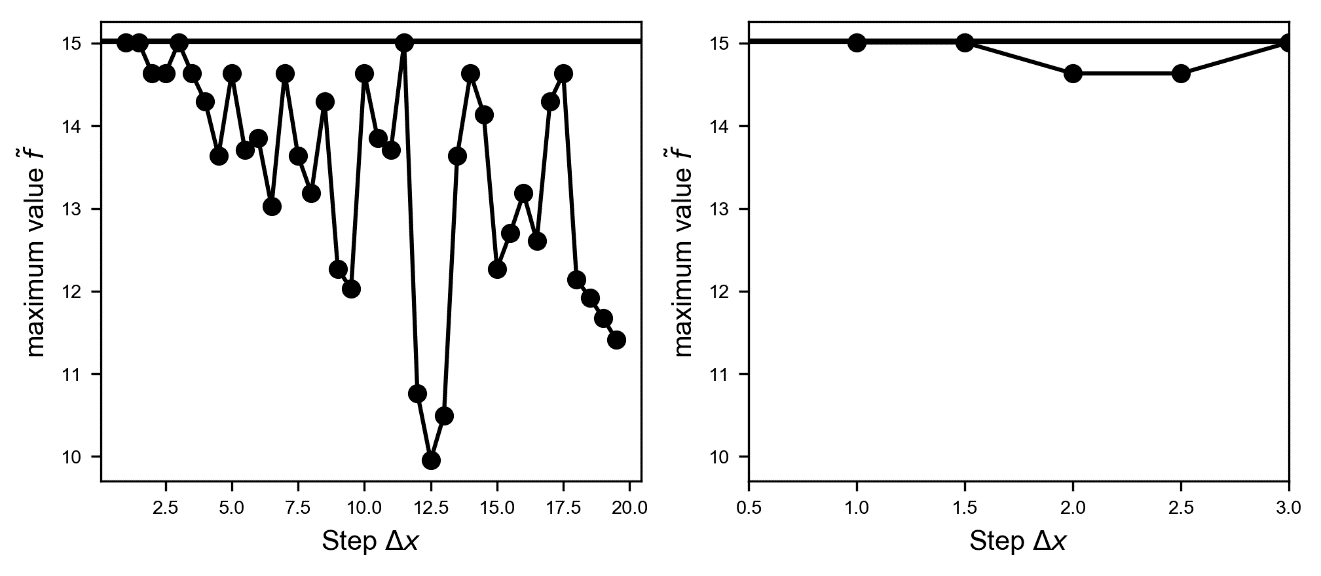


Figure 7-4. The behaviour of the found value of the maximum vs. the step .

In the zoom in the right plot of Figure 7-4, is evident how smaller values of gets you better values of . Note that a step of 1.0 means sampling 80 values of . If for example the evaluation takes 1 days, you will need to wait 80 days to get all the measurements you need.

**Tip** Grid search is a method that is efficient to be used only when the black box function is cheap. To get good results, a big number of sampling points is usually needed.

To make sure you are really getting the maximum you should decrease the step , or increase the number of sampling points, until the maximum value you find does not change appreciably anymore. In our example above, as we see from the right plot in Figure 7-4, we are sure we are close the maximum when our step gets smaller than roughly 2.0, or in other words when the number of sampled points is greater or roughly equal to 40. Remember 40 may seems quite a small number at first sight, but if evaluates the metric of your deep learning model, and the training takes for example 2 hours, you are looking at 3.3 days of computer time. Normally in the deep learning world 2 hours is not much for training a model, so make a quick calculation before starting a long grid search. Additionally, keep in mind that when doing hyper parameter tuning you are moving in a multi-dimensional space (you are not optimizing only one parameter, but many), so the number of evaluations needed get big very fast.

Let's make a quick example. Let's suppose you decide you can afford 50 evaluations of your black box function. If you decide you want to try the following hyper parameters

* Optimizer (RMSProp, Adam or plain GD) (3 values)
* Number of epochs (1000, 5000 or 10000) (3 values)

You are already looking at 9 evaluations. How many values of the learning rate can you then afford to try? Only 5! And with 5 values is not probable to get close to the optimal value. This example has the goal to let you understand how grid search is viable only for cheap black box functions. Remember that often, time is not the only problem. If you are using for example the google cloud platform to train your network, you are paying the hardware you use by the second. Maybe you have lots of time at your disposal, but costs may get over your budget very quickly.

## Random Search

A strategy that is as "dumb" as grid search, but that works amazingly a lot better, is random search. Instead of sampling points regularly in the range you sample the points randomly. We can do it with the code

import numpy as np

randomsearch = np.random.random([40])\*80.0

The array randomsearch will look like this

array([ 0.84639256, 66.45122608, 74.12903502, 36.68827838, 61.71538757, 69.29592273, 48.76918387, 69.81017465, 1.91224209, 21.72761762, 22.17756662, 9.65059426, 72.85707634, 2.43514133, 53.80488236, 5.70717498, 28.8624395 , 33.44796341, 14.51234312, 41.68112826, 42.79934087, 25.36351055, 58.96704476, 12.81619285, 15.40065752, 28.36088144, 30.27009067, 16.50286852, 73.49673641, 66.24748556, 8.55013954, 29.55887325, 18.61368765, 36.08628824, 22.1053749 , 40.14455129, 73.80825225, 30.60089111, 52.01026629, 47.64968904])

Depending on the seed you used the actual numbers you get may be different. As we have done for grid search, you can see in Figure 7-5 the plot of , where the crosses mark the sampled points, and the black square the maximum. On the right plot, you see a zoom around the maximum.

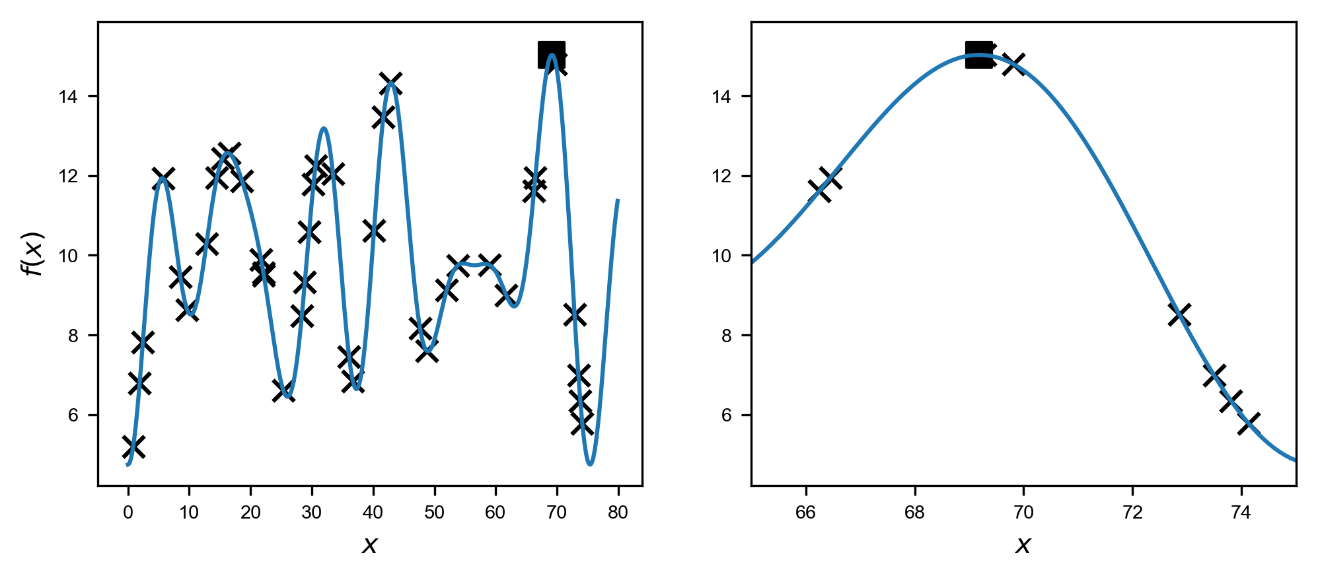


Figure 7-5. The function on the range . The crosses mark the point we sampled with random search, and the black square mark the maximum.

The risk with this method is that, if you are very unlucky, your random chosen points are nowhere close the real maximum. But that probability is quite low. Note that if you take a constant probability distribution for your random points you have the same probability of getting the points everywhere. Now it is interesting to see how this method performs. Let us consider 200 different random sets of 40 points, obtained by varying the random seed used in the code. The distributions of the maximum found  is plotted in Figure 7-6.

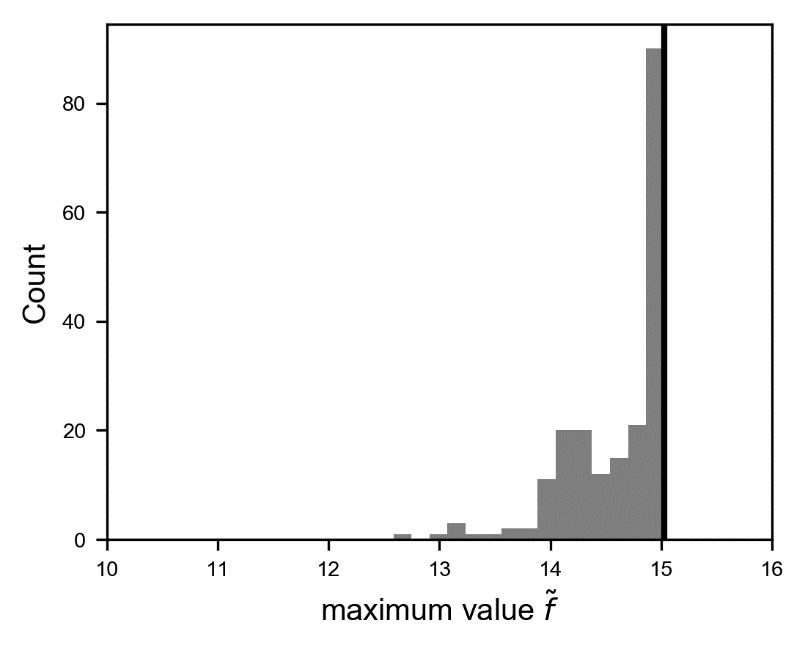


Figure 7-6. The distribution of the results for obtained by 200 different random sets of 40 points sampled in the random search. The black vertical line indicates the real maximum of .

As you can see regardless of the random sets used you get, in the most cases, very close to the real maximum. In Figure 7-7 you can see the distributions of the maximum found with random search varying the number of points sampled, from 10 to 80.

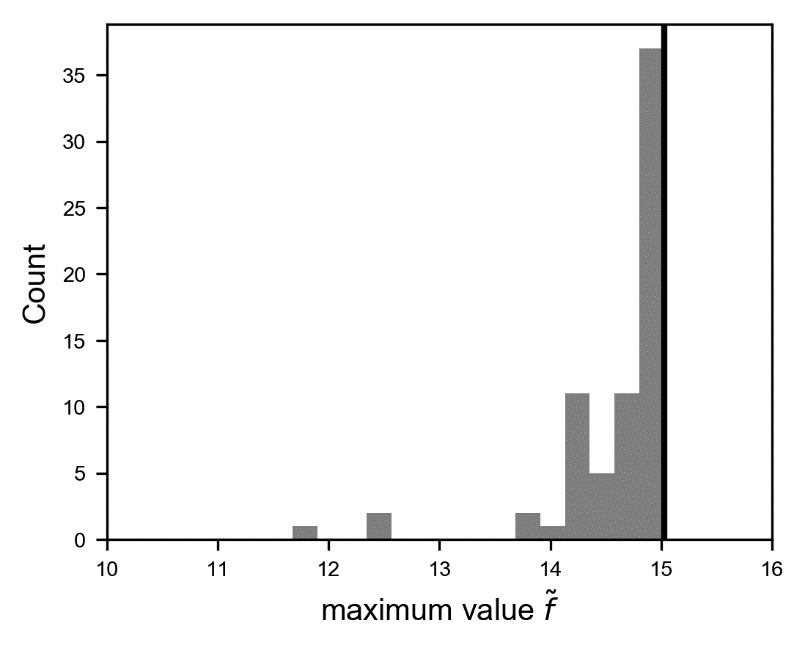


Figure 7-7. The distribution of the results for obtained by varying the number of points sampled in the random search, from 10 to 80. The black vertical line indicates the real maximum of .

If you compare with grid search, you can see random search is better at getting consistently results closer to the real maximum. In Figure 7-8 you can see a comparison between the distribution you get for your maximum when using a different number of sampling points with random and grid search. In both cases the plots were generated with 38 different sets, so that the total count is the same.

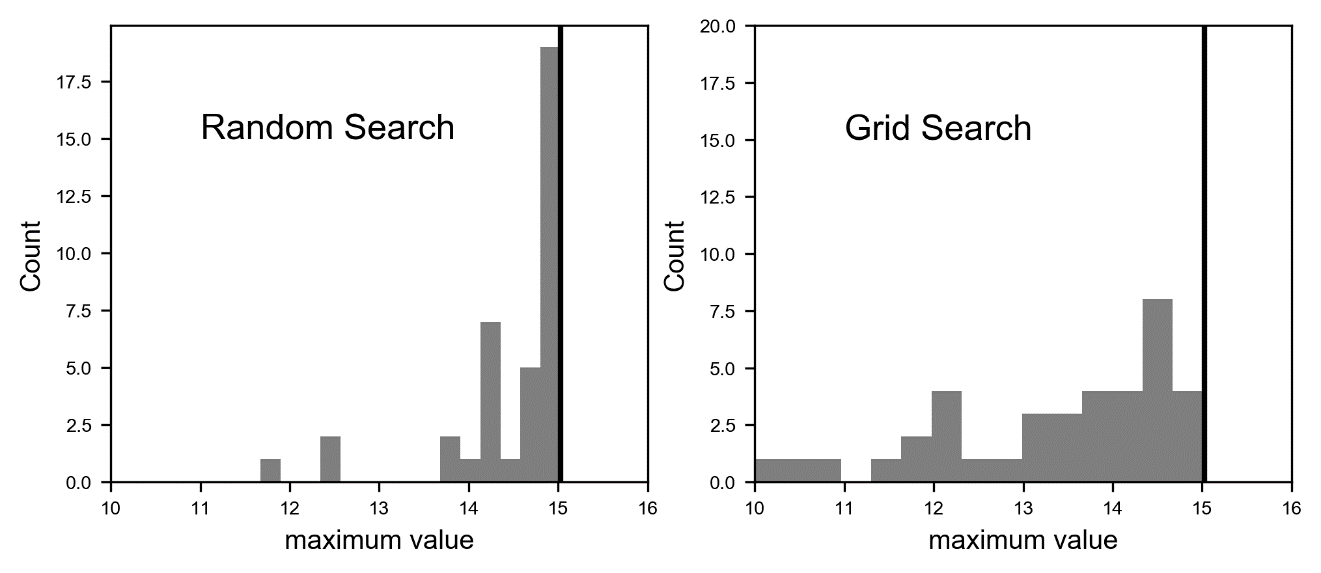


Figure 7-8. A comparison of the distribution of between grid (right) and random (left) search while varying the number of sampling points . Both plots counts sums to 38, the number of different numbers of sampling points used. The correct value of the maximum is marked by the vertical black line in both plots.

Is easy to see how on average random search is better than grid search. The values you get are consistently closer to the right maximum.

**Tip** Random search is consistently better than grid search and you should use it every time is possible. The difference between random search and grid search becomes even more marked when dealing with a multi-dimensional space for your variable . Hyper parameter tuning is practically always a multi-dimensional optimization problem.

If you are interested in a very good paper on how random search scale at high dimensional problems, you can read the paper by J. Bergstra and Y. Bengio, Random Search for Hyper-Parameter Optimization [1].

## Coarse to fine optimization

There is still an optimization trick that helps with grid or random search. It is called "coarse to fine optimization". Let's suppose we want to find the maximum of between and . We will look at the idea for random search, but it works in the same way with grid search. The following steps gives you the algorithm you need to follow for this optimization.

1. Do a random search in the region . Let's indicate the maximum found with
2. Consider now a smaller region around , for some that we will discuss later and do again a random search in this region. Our hypothesis is of course that the real maximum lies in this region. We will indicate the maximum you find here with
3. Repeat step 2 around , in the region we will indicate with with a smaller than and indicate the maximum you find in this step with
4. Now repeat step 2 again around , in the region we will indicate with with a smaller than
5. Go on in the same way as many times as you need until the maximum in the region does not change any more.

Usually just one or two iterations are used, but theoretically you could go on for a large number of iterations. The problem with this method is that you cannot be really sure that your real maximum really lies in your regions . But this optimization has a big advantage if it does. Let us consider the case where we do a standard random search. If we want to have on average a distance between the sampled points of 1% of we would need around 100 points if we decided to perform only one random search, and consequently we had to perform 100 evaluations. Now let us consider the algorithm we just described. We could start with just 10 points in region , here we will indicate the maximum we find with . Then let us take and let us take again 10 points in region ). In the interval we will have on average a distance between the points of 1% of , but we just sampled our functions only 20 times instead of 100, so a factor 5 less! For example, let us just sample our function we used above between and with 10 points with the code

np.random.seed(5)

randomsearch = np.random.random([10])\*80

x = 0

m = 0.0

for i, val in np.ndenumerate(f(randomsearch)):

if (val > m):

m = val

x = randomsearch[i]

this gives us the maximum location and value of and , so not bad but not yet as precise as the real ones 69.18 and 15.027. Now let us sample again 10 points around the maximum we have found in the regions

randomsearch = x + (np.random.random([10])-0.5)\*8

x = 0

m = 0.0

for i, val in np.ndenumerate(maxim(randomsearch)):

if (val > m):

m = val

x = randomsearch[i]

this gives us the result 69.189 and 15.027. Quite a precise result with only 20 evaluations of the function. If we do a plain random search with **500** (25 times more than what we just did) sampling points, we get and . This result shows how this trick can be really helpful. But remember the risk: if your maximum is not in your regions you will never be able to find it, since you are still dealing with random numbers. So, it is always a good idea to choose the regions relatively big to make sure that they contain your maximum. How big depends, as almost everything in the deep learning world, on your dataset and problem and may be impossible to know in advance. Testing is unfortunately required. In Figure 7-9 you can see the sampled points on the function In the plot on the left you see the first 10 points, and on the right the region with the additional 10 points. The small rectangle on the left plot marks the region .

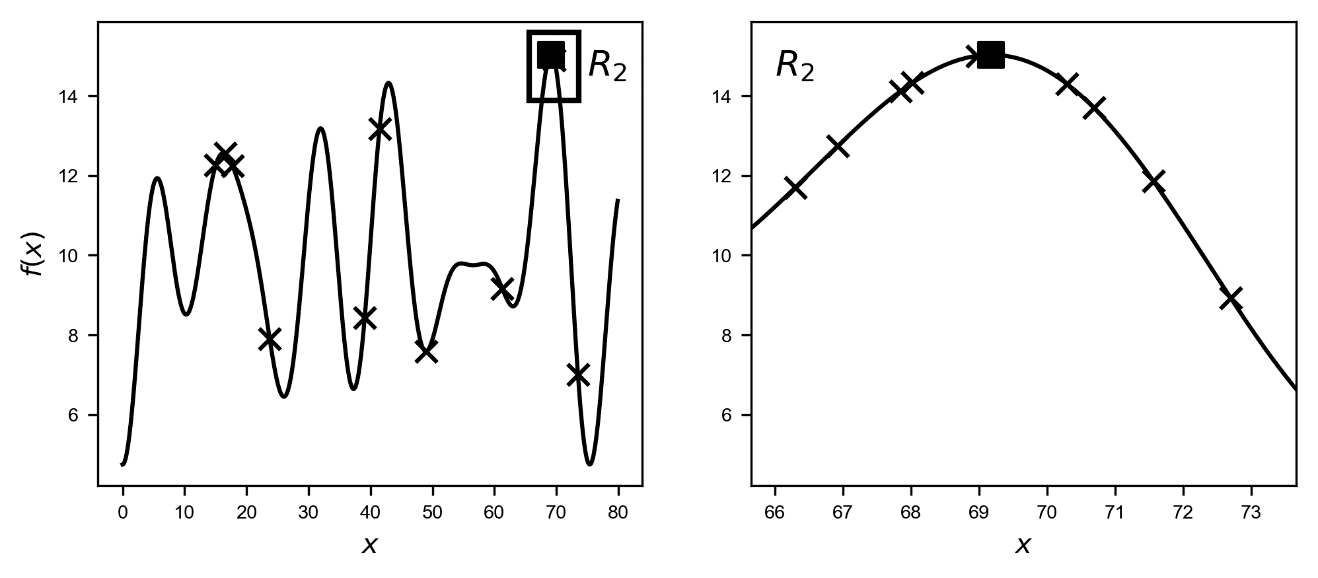


Figure 7-9. The function . The crosses mark the sampled points. On the left the 10 points sampled in the region (entire range), on the right the 10 points sampled in . The black square mark the real maximum.The plot on the right is the zoom on the region .

Now the choice of how many points you should sample at the beginning is crucial. We had luck here. Let us consider a different seed when choosing our initial 10 random points and let us see what can happen. Check Figure 7-10. Choosing the wrong initial points leads to disaster!



Figure 7-10. The function . The crosses mark the sampled points. On the left the 10 points sampled in the region (entire range), on the right the 10 points sampled in . The black square mark the real maximum. The algorithm finds a maximum very well around 16, simply not the absolute maximum. The small rectangle on the left mark the region . The plot on the right is the zoom on the region .

Note in Figure 7-10, how the algorithm finds the maximum at around 16, since in the initial sampled points the maximum value is around , as you can see on the plot on the left in Figure 7-10. No points are close to the real maximum around . The algorithm finds a maximum really well, simply not the absolute maximum. That is the danger you face when using this trick. It can even go worse than that. Consider Figure 7-11 where we just sampled one single point at the beginning. You can see on the plot on the left of Figure 7-11, how the algorithm completely misses any maximum. It simply gives as a result the highest value of the points marked by crosses on the points on the right plot: (58.4, 9.78).

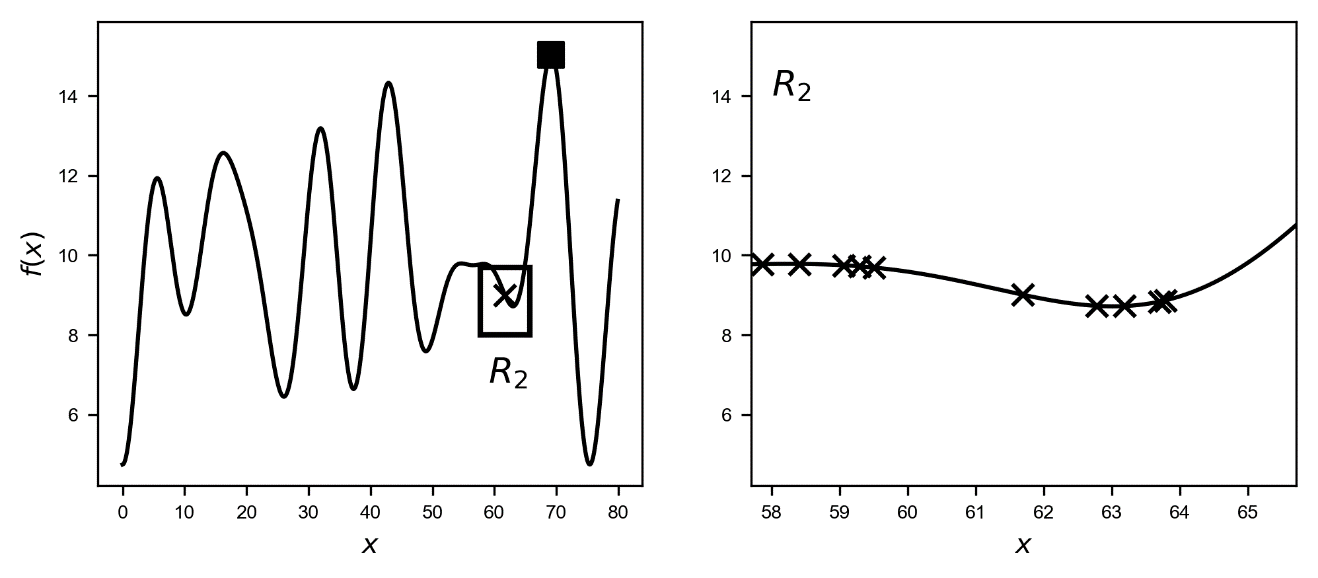


Figure 7-11. The function . The crosses mark the sampled points. On the left the 1 point sampled in the regions (entire range), on the right the 10 points sampled in . The black square marks the real maximum. The algorithm does not find any maximum since none is present in the region . The plot on the right is the zoom on the region .

If you decide to use this method, keep in mind that you will still need a good number of points at the beginning to get close to the maximum before refining your search. After you are relatively sure to have points around your maximum, you can use this technique to refine your search.

## Bayesian Optimization

### Nadaray-Watson regression

To understand Bayesian Optimization, we have first to look at some new mathematical concepts. Let us start with the Nadary-Watson regression, an idea from 1964 [2]. The basic idea is quite simple. Given an unknown function and given points we indicate with with the value of the function calculated at the different . The idea of the Nadaray-Watson regression is that we can evaluate the unknown function at an unknown point using the formula

where are weights that are calculated according to the formula

where is called a **Kernel**. Note that given how the weights are defined we have

In the literature you can find several kernels but the one we are interested in is the Gaussian one, often called the **Radial Basis Function** (RBF)

The parameter makes the gaussian shape wider or narrower. The is typically the variance of your data, but in this case, it plays no role since the weights are normalized to one. This is at the base of Bayesian optimization as we will see later.

### Gaussian process

Before talking about Gaussian processes, we first must define what a random process is. We talk of a random process when for any point we assign a random variable . A random process is Gaussian if for any finite number of points, their joint distribution is normal. That means that , and for then the vector

where with the notation we intend that the vector components follow a normal distribution, indicated with . Remember that a random variable with a Gaussian distribution is said to be normally distributed. From here comes the name Gaussian process. The probability distribution of the normal distribution is given by the function

where is the mean or expectation of the distribution and is the standard deviation. We will use the following notation from here on:

and the covariance of the random values will be indicated here by

the choice of the letter has a reason. We will assume in what follows that the covariance will have a gaussian shape, and we will use for the RBF function we defined before.

### Stationary Process

For simplicity we will consider here only stationary processes. A random process is stationary if its joint probability distribution does not change with time. That means also that mean and variance will not change when shifted in time. We will also consider a process for which its distribution depends only on the relative position of the points. This leads to the conditions

Note that to apply the method we are describing, you must convert first your data to be stationary if that is not already the case, eliminating seasonality or trends in time for example.

### Prediction with Gaussian processes

Now we have reached the interesting part: given the vector how can we estimate at an arbitrary point ? Since we are dealing with random processes, what we will estimate is the probability that the unknown function assumes a given value . Mathematically we will predict the following quantity

or in words the probability of getting the value given the vector , composed by all the points .

Assuming that is a stationary Gaussian Process with the prediction can be shown to be given by

where with we have indicated the normal distribution calculated on the points with average 0 and covariance matrix of dimensions , since we have points in the numerator. The derivation is somewhat involved and is based on several theorems, as Bayes' theorem. For more details you can refer to the (advanced) explanation by C.B. Do [3], where everything is explained in, although advanced, details. To understand what Bayesian optimization is, we can simply use the formula without derivation. will have dimensions since we have only points in the denominator.

We have

And

where we have defined

It can be shown[[2]](#footnote-2) that the ratio of two normal distributions is again a normal distribution, so that we can write

With

The derivation of the exact form for and is quite long and would go beyond the scope of the book. Basically, now we know that, on average, our unknown function will assume the value in with a variance . Let's now see how this method really works in practice in Python. Let's first define our Kernel

def K(x, l, sigm = 1):

return sigm\*\*2\*np.exp(-1.0/2.0/l\*\*2\*(x\*\*2))

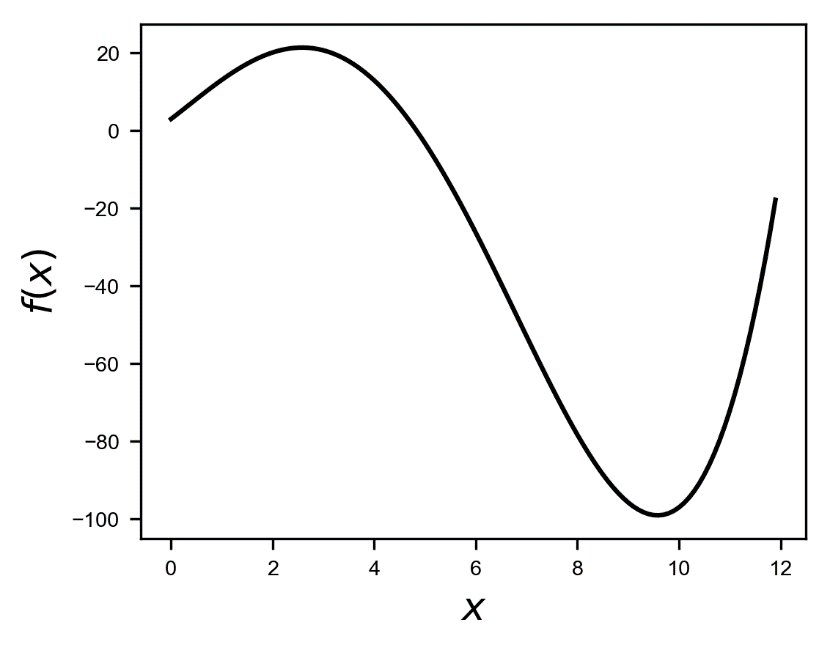
Let us simulate our unknown function with an easy one

that can be implemented as

def f(x):

return x\*\*2-x\*\*3+3+10\*x+0.07\*x\*\*4

Let us consider the function in the range . In Figure 7-12 you can see how the function looks like.



*Figure 7-12. A plot of our unknown test function as described in the text.*

Let's build first our vector with 5 points

randompoints = np.random.random([5])\*12.0

f\_ = f(randompoints)

where we have used the seed 42 for the random numbers: np.random.seed(42). In Figure 7-13 you can see the random points marked with a cross on the plot.

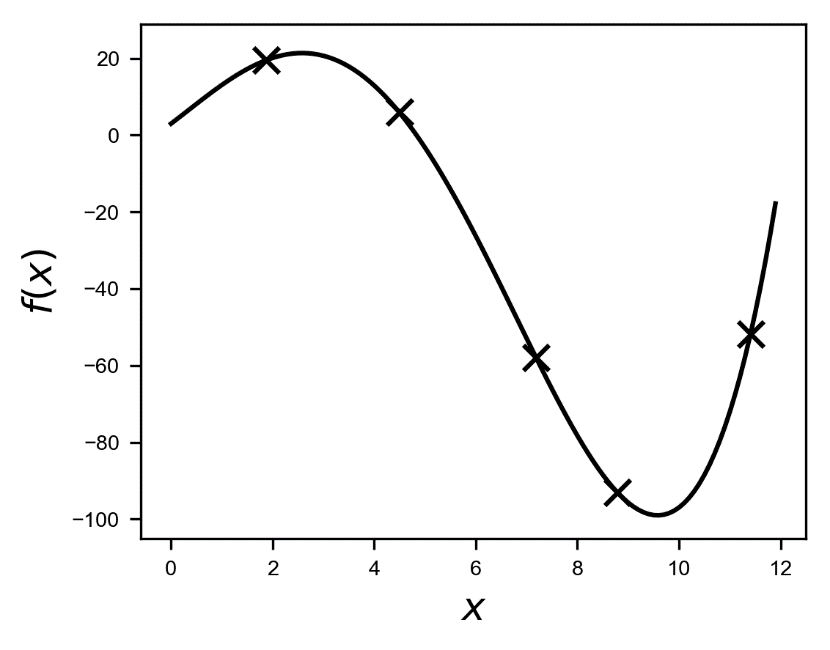


Figure 7-13. A plot of the unknown function. The crosses mark the random point chosen in the text.

We can apply the method described earlier with the following code

xsampling = np.arange(0,14,0.2)

ybayes\_ = []

sigmabayes\_ = []

for x in xsampling:

f1 = f(randompoints)

sigm\_ = np.std(f1)\*\*2

f\_ = (f1-np.average(f1))

k = K(x-randompoints, 2 , sigm\_)

C = np.zeros([randompoints.shape[0],

randompoints.shape[0]])

Ctilde = np.zeros([randompoints.shape[0]+1,

randompoints.shape[0]+1])

for i1,x1\_ in np.ndenumerate(randompoints):

for i2,x2\_ in np.ndenumerate(randompoints):

C[i1,i2] = K(x1\_-x2\_, 2.0, sigm\_)

Ctilde[0,0] = K(0, 2.0, sigm\_)

Ctilde[0,1:randompoints.shape[0]+1] = k.T

Ctilde[1:,1:] = C

Ctilde[1:randompoints.shape[0]+1,0] = k

mu = np.dot(np.dot(np.transpose(k), np.linalg.inv(C)), f\_)

sigma2 = K(0, 2.0,sigm\_)-

np.dot(np.dot(np.transpose(k),

np.linalg.inv(C)), k)

ybayes.append(mu)

sigmabayes\_.append(np.abs(sigma2))

ybayes = np.asarray(ybayes\_)+np.average(f1)

sigmabayes = np.asarray(sigmabayes\_)

now please take some time to understand it. In the list ybayes we will find the values of evaluated on the values contained in the array xsampling. Here are some hints that will help you understanding the code

* We do a loop over a range of points where we want to evaluate our function, with the code for x in xsampling:
* We build our and vectors with the code for each element of the vectors: k = K(x-randompoints, 2 , sigm\_) and f1 = f(randompoints). For the Kernel we have chosen a value for the parameter l as defined in the function of 2. We have subtracted in the vector the average to obtain to be able to apply the formulas as derived.
* We build the matrices and .
* We calculate with mu = np.dot(np.dot(np.transpose(k), np.linalg.inv(C)), f\_) and the standard deviation.
* At the end we reapply all the transformation that we have done to make our process stationary in the opposite order, simply adding the average of again to the evaluated surrogate function ybayes = np.asarray(ybayes\_)+np.average(f1)

In Figure 7-14 you can see how this method works. The dashed line is the predicted function obtained plotting as calculated in the code, when we have 5 points at disposal (. The gray area is the region between the estimated function and +/- .

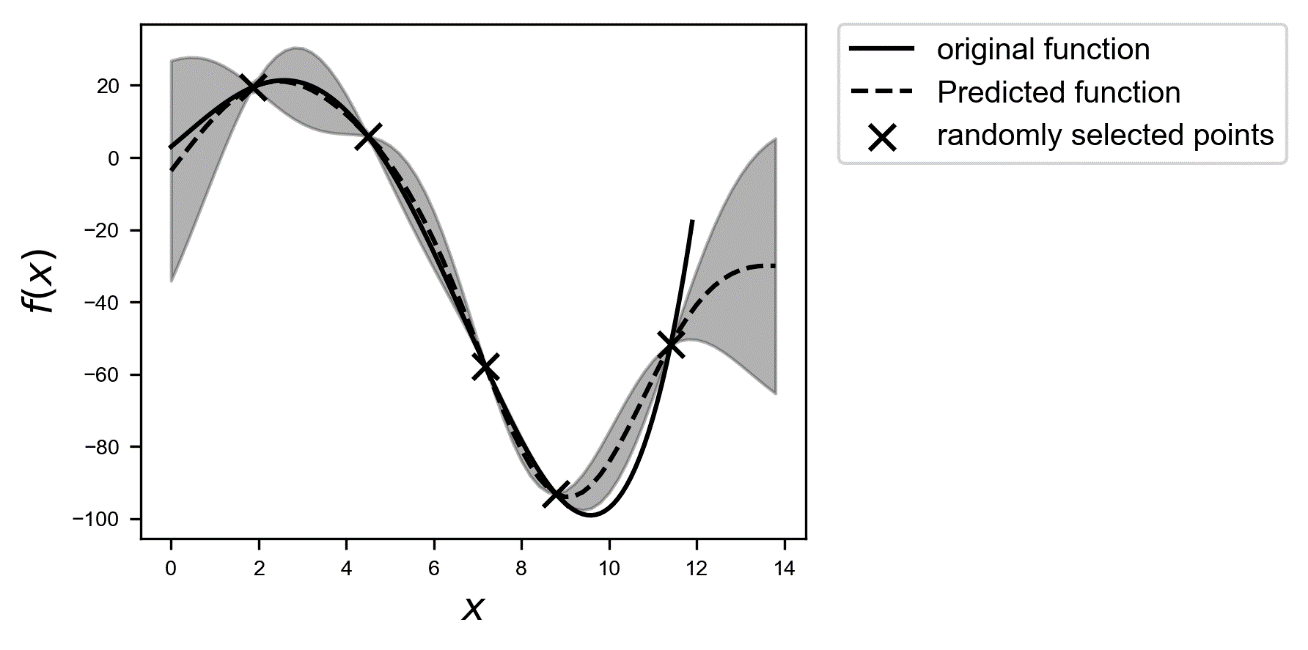


Figure 7-14. The predicted function, dashed line, calculated evaluating . The gray area is the region between the estimated function and +/- .

Given the few points we have, is actually not a bad result. Now keep in mind that you still need a few points to be able to get a reasonable approximation. In Figure 7-15 you can see the result if we have only two points at our disposal. The result is not as good. The gray area is the region around the estimated function and +/- You can see how as far as you are from the points we have, the higher the uncertainty, or the variance, of the predicted function.

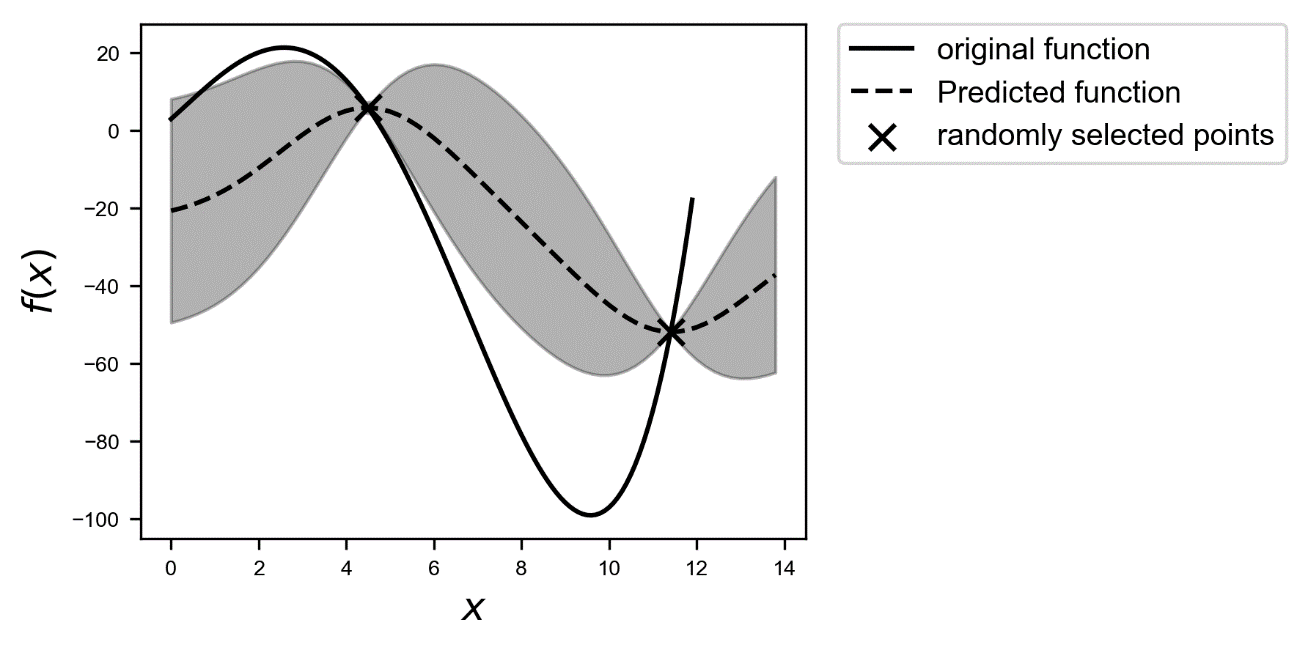


Figure 7-15. The predicted function, dashed line, when we have at our disposal only two points. The gray area is the region between the estimate function +/- .

Let us stop for a second and think about why we do all this. The idea behind it is to find a so-called **surrogate function**  that approximate our function and that have the property

This surrogate function must have another very important property: it must be cheap to evaluate. In this way we can find easily the maximum of and using the previous property we will have the maximum of , that by hypothesis is costly.

But as you have seen it may be very difficult to know if we have enough points to find the right value of the maximum. After all, by definition, we do not have any idea how our black box function looks like. So how to solve this problem?

The main idea behind the method can be described by the following algorithm:

1. We start with a small number of sample points randomly chosen (how small it should be, will depend on your problem)
2. We use this set of points to get a surrogate function as described above
3. We add an additional point to our set, with a specific method that we will discuss later, and reevaluate the surrogate function
4. If the maximum we find with the surrogate function continues to change, we will continue adding points as in step 3 until the maximum does not change anymore or we run out of time or budget and cannot perform any evaluation anymore

If the method we hinted at in step 3 is smart enough, we will be able to find our maximum relatively quickly and accurately.

### Acquisition Function

But how to choose the additional points I mentioned in Step 3 in the previous section? The idea is to use a so-called acquisition function. The algorithm works in this way

1. We choose a function (and we will see a few possibilities in a moment) called an "acquisition function"
2. Then we choose as additional point , the one at which the acquisition function has a maximum

There are several acquisition functions we can use. I will describe here only one that we will use to see how this method works, but there are several that you may want to check out like: entropy search, probability of improvement, expected improvement and upper confidence bound.

#### Upper confidence bound (UCB)

In the literature you find two variations of this acquisition function. We can write the function as

where we have indicated with the "expected" value of the surrogate function on the -range we have in our problem. The expected value is nothing else than the average of the function over the given range. is the variance of the surrogate function that we calculate with our method at the point . The new point we select is the one where is maximum. is a tradeoff parameter. This acquisition function basically selects the points where the variance is biggest. Check again Figure 7-15. The method would select the points where the variance is greater, so points as far as possible from the points we have already. In this way the approximation tends to get better and better. Another variation of the UCB acquisition function is the following

This time the acquisition function will make a tradeoff between choosing points around the surrogate function maximum and points where its variance is biggest. This second method works best to find quickly good approximation of the maximum of , while the first tends to give good approximation of over the entire range. In the next section we will see how these methods works.

#### Example

Let us start with the complex trigonometric function as described at the beginning of the chapter and let us consider the range . Our goal is to find its maximum and approximate the function. To facilitate our coding let us define two functions: one to evaluate the surrogate function and one to evaluate the new point. To evaluate the surrogate function, we can use the following function

def get\_surrogate(randompoints):

ybayes\_ = []

sigmabayes\_ = []

for x in xsampling:

f1 = f(randompoints)

sigm\_ = np.std(f\_)\*\*2

f\_ = (f1-np.average(f1))

k = K(x-randompoints, 2.0, sigm\_ )

C = np.zeros([randompoints.shape[0],

randompoints.shape[0]])

Ctilde = np.zeros([randompoints.shape[0]+1,

randompoints.shape[0]+1])

for i1,x1\_ in np.ndenumerate(randompoints):

for i2,x2\_ in np.ndenumerate(randompoints):

C[i1,i2] = K(x1\_-x2\_, 2.0, sigm\_)

Ctilde[0,0] = K(0, 2.0)

Ctilde[0,1:randompoints.shape[0]+1] = k.T

Ctilde[1:,1:] = C

Ctilde[1:randompoints.shape[0]+1,0] = k

mu = np.dot(np.dot(np.transpose(k),

np.linalg.inv(C)), f\_)

sigma2 = K(0, 2.0, sigm\_)

- np.dot(np.dot(np.transpose(k),

np.linalg.inv(C)), k)

ybayes\_.append(mu)

sigmabayes\_.append(np.abs(sigma2))

ybayes = np.asarray(ybayes\_)+np.average(f1)

sigmabayes = np.asarray(sigmabayes\_)

return ybayes, sigmabayes

This function has the same code we have already discussed in our example in the previous sections but is packed in a function that returns the surrogate function, contained in the array ybayes, and the , contained in the array sigmabayes. Additionally, we need a function that evaluate the new points using the acquisition function . We can get it easily with the function

def get\_new\_point(ybayes, sigmabayes, eta):

idxmax = np.argmax(np.average(ybayes)

+eta\*np.sqrt(sigmabayes))

newpoint = xsampling[idxmax]

return newpoint

To make things simpler we define the array that contains all the values we want at the beginning outside the functions. Let us start with just 6 randomly selected points. To check how our method is doing let us start with some definitions

xmax = 40.0

numpoints = 6

xsampling = np.arange(0,xmax,0.2)

eta = 1.0

np.random.seed(8)

randompoints1 = np.random.random([numpoints])\*xmax

In the array randompoints1 we will have our first 6 selected random points. We can easily get the surrogate function with our function with

ybayes1, sigmabayes1 = get\_surrogate(randompoints1)

In Figure 7-16 you can see the result. The dotted line is the acquisition function normalized to fit in the plot.

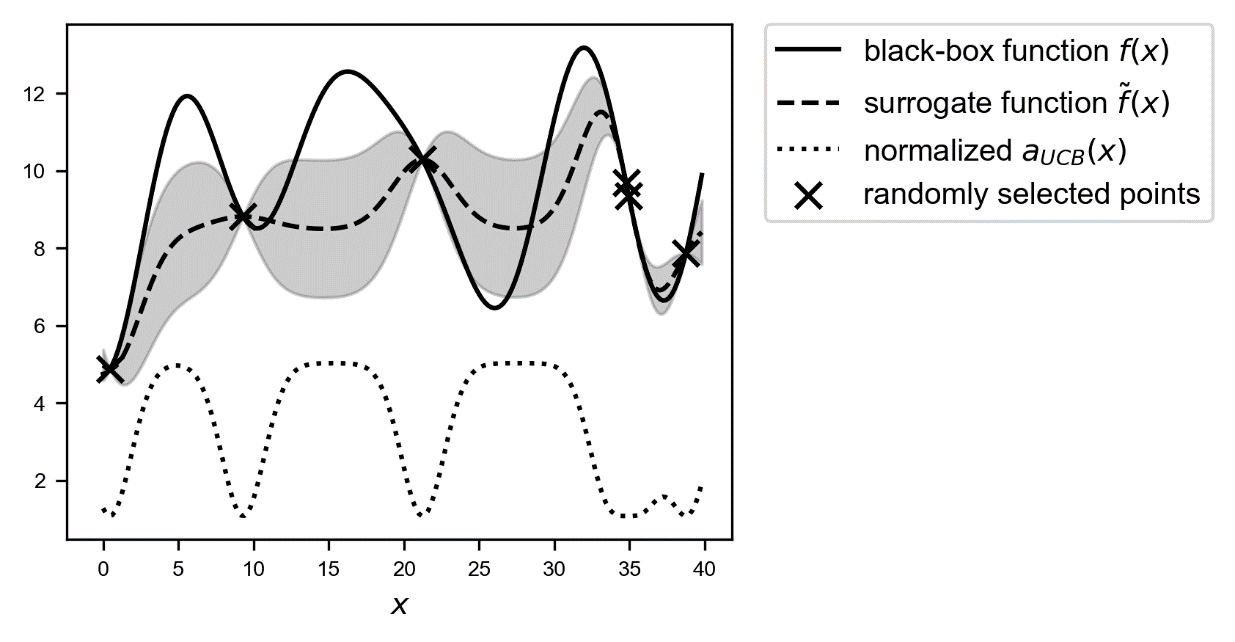


Figure 7-16. An overview of the black-box function , solid line, the randomly selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

The surrogate function is not yet very good, since we don't have enough points, and the big variance (the gray area), makes this evident. The only region that is well approximated is the region . You can see how the acquisition function is big when the surrogate function is not approximating well the black-box function, and small when it does, as for example for . So intuitively choosing as a new point the one where is maximum, is equivalent to choose the points where the function is less well approximated, or in more mathematical terms where the variance is bigger. For a comparison in Figure 7-17 you can see the same plot as in Figure 7-16, but with the acquisition function and with .

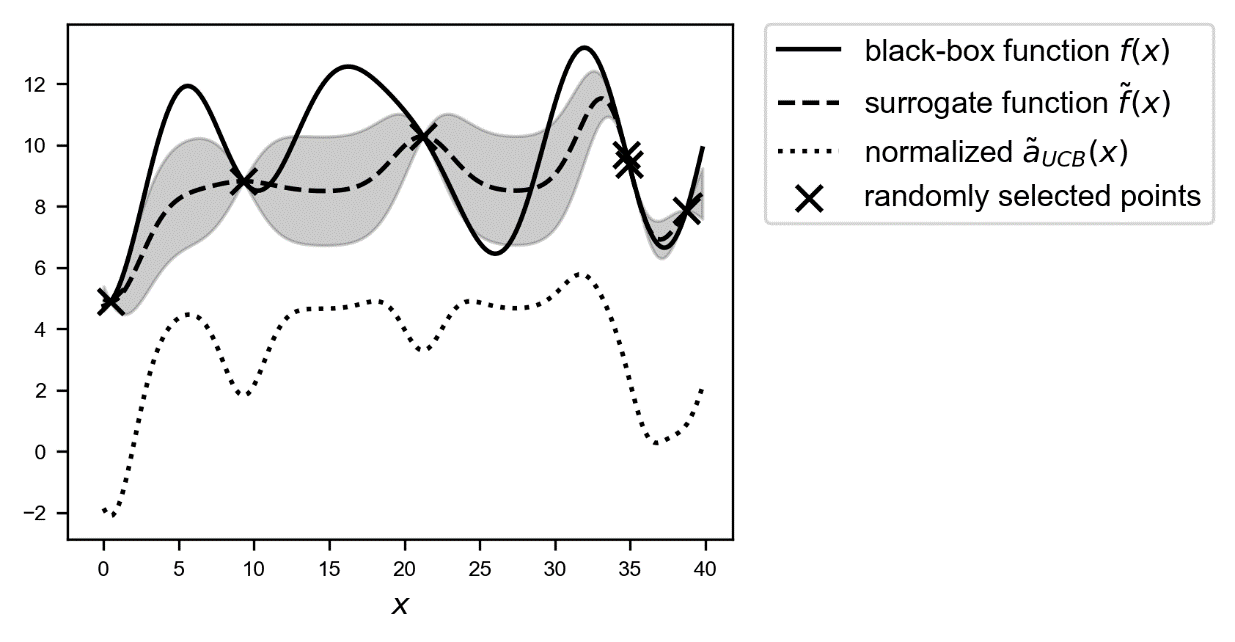


Figure 7-17. An overview of the black-box function , solid line, the randomly selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

As you can see tends to have a maximum around the maximum of the surrogate function. Keep in mind that if is big, then the maximum of the acquisition function will shift towards regions with high variance. But this acquisition function tends to find "a" maximum slightly faster. I said "a" since it depends on where the maximum of the surrogate function is, and not where the maximum of the black-box function is.

Let us see now what happens while using with . For the first additional point we need to simply run the following 3 lines of code

newpoint = get\_new\_point(ybayes1, sigmabayes1, eta)

randompoints2 = np.append(randompoints1, newpoint)

ybayes2, sigmabayes2 = get\_surrogate(randompoints2)

For sake of simplicity, we simply name each array for each step differently, instead of creating a list. But typically, you should make these iterations automatic. In Figure 7-18 you can see the result with the additional point, marked with a black circle.

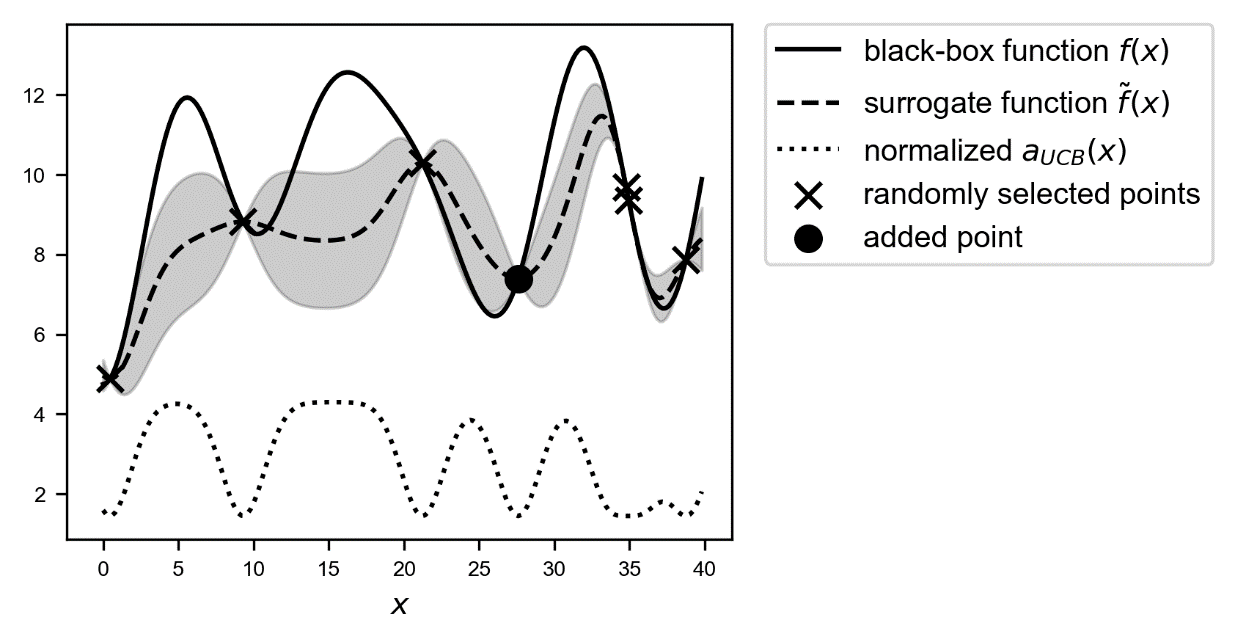


Figure 7-18. An overview of the black-box function , solid line, the randomly selected points, marked with crosses, and with the new selected point around , marked by a circle, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

The new point is around . Let's continue to add points. In Figure 7-19 you can see the results after adding 5 points.

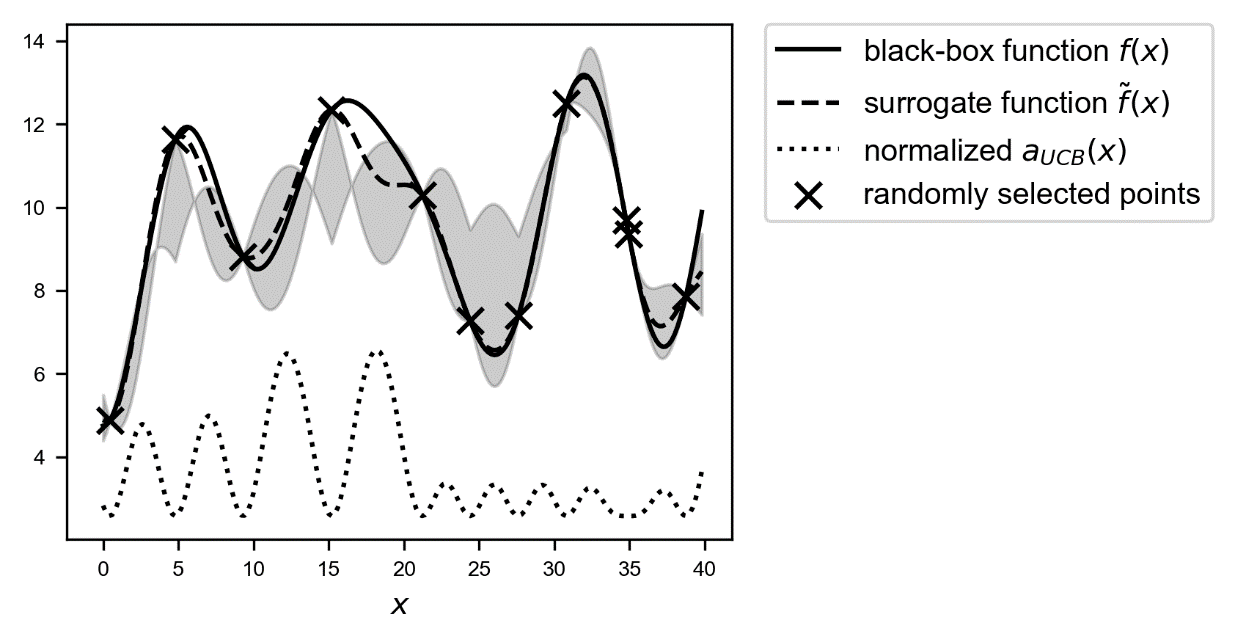


Figure 7-19. An overview of the black-box function , solid line, the randomly selected points with the new 6 selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

Check the dashed line. Now our surrogate function approximates the black-box function quite well, especially around the real maximum. Now using this surrogate function, we can find a very good approximation of our original function with just 11 evaluations in total! Keep in mind we don't have any additional information about , except the 11 evaluations.

Now let us try what happens with the acquisition function , and let us check how fast we can find the maximum. In this case let us use , to get a better balance between maximum of surrogate function and its variance. In Figure 7-20 you can check the result after just adding one single additional point, marked by a black circle. We have already quite a good approximation of the real maximum!

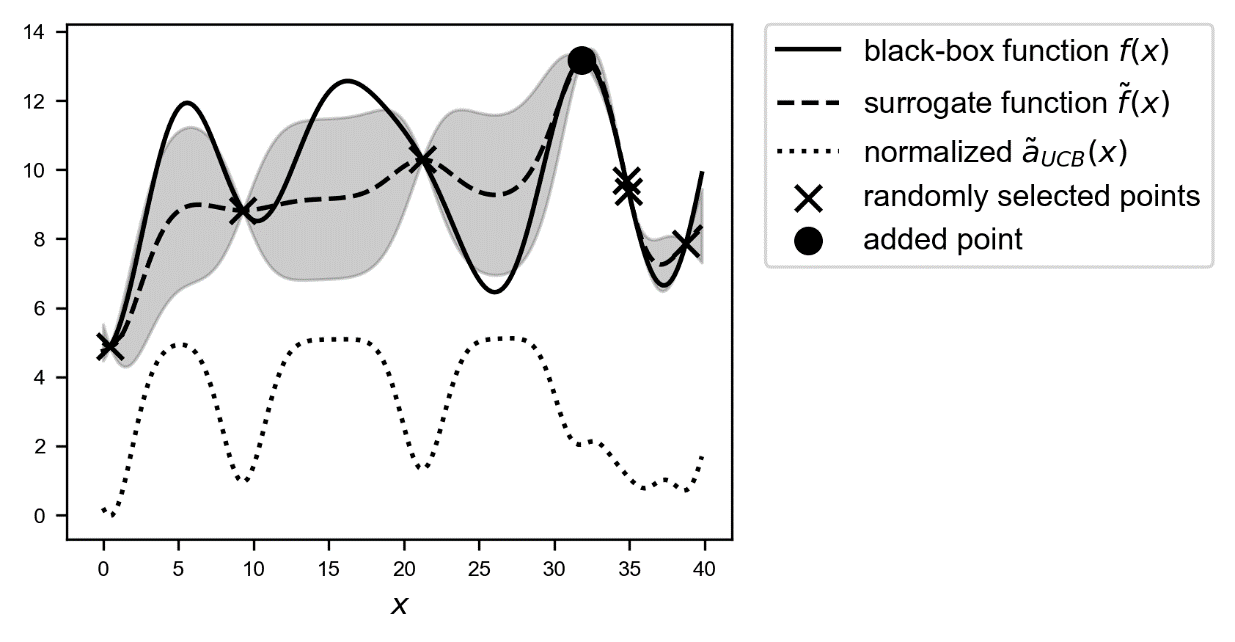


Figure 7-20. An overview of the black-box function , solid line, the randomly selected points with the additional selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

Now let us add an additional point. You can see in Figure 7-21 that the additional point is now still close to the maximum but shifted in the direction of the area with a high variance around 30.

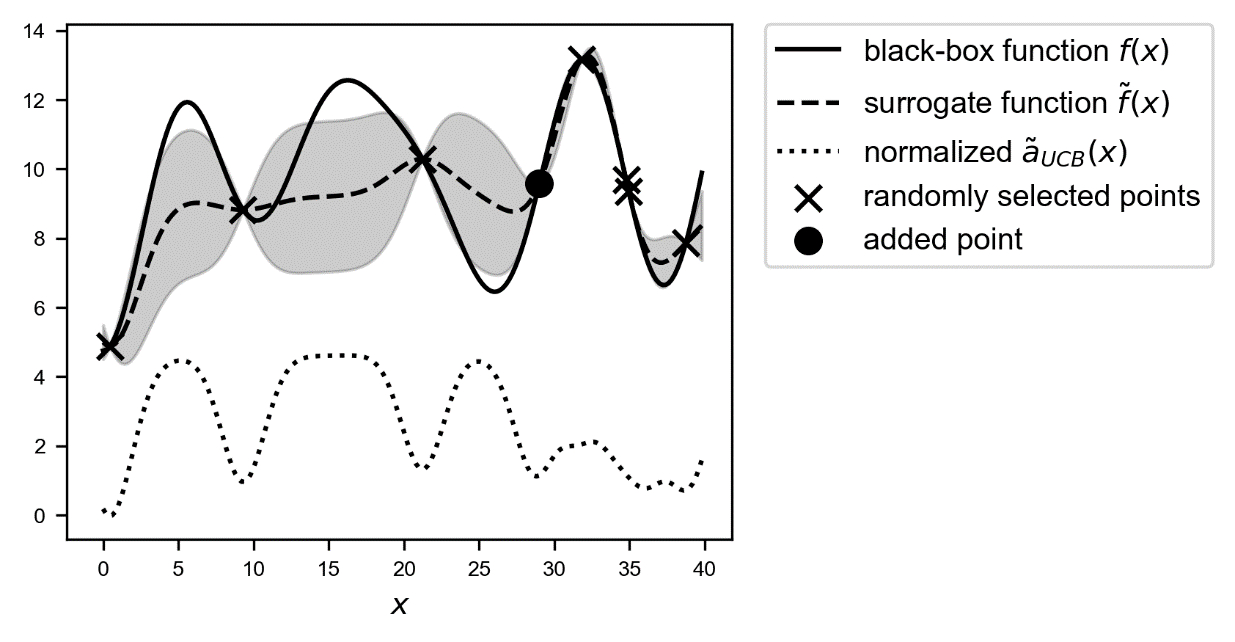


Figure 7-21. An overview of the black-box function , solid line, the randomly selected points with the additional selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

If we would choose smaller the point would be closer to the maximum, and if we would choose it bigger the point would be closer to the point with the highest variance between 25 and roughly 32. Now let us add an additional point and let us see what happens. In Figure 7-22 you can see how the method now chooses a point close to another region with high variance, between 10 and roughly 22, again marked by a black circle.

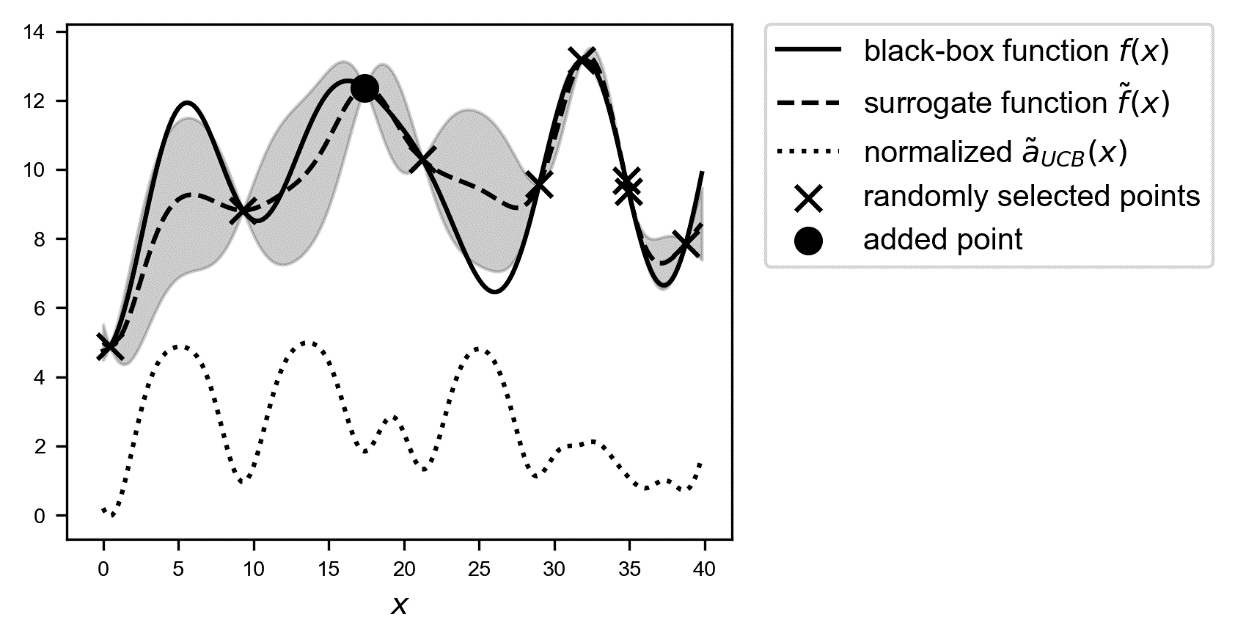


Figure 7-22. An overview of the black-box function , solid line, the randomly selected points, marked by crosses, and the additional selected point, marked by a black circle, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

And finally, the method refines the maximum area around 15 as you can see in Figure 7-23, adding a point around 14, marked by the black circle.

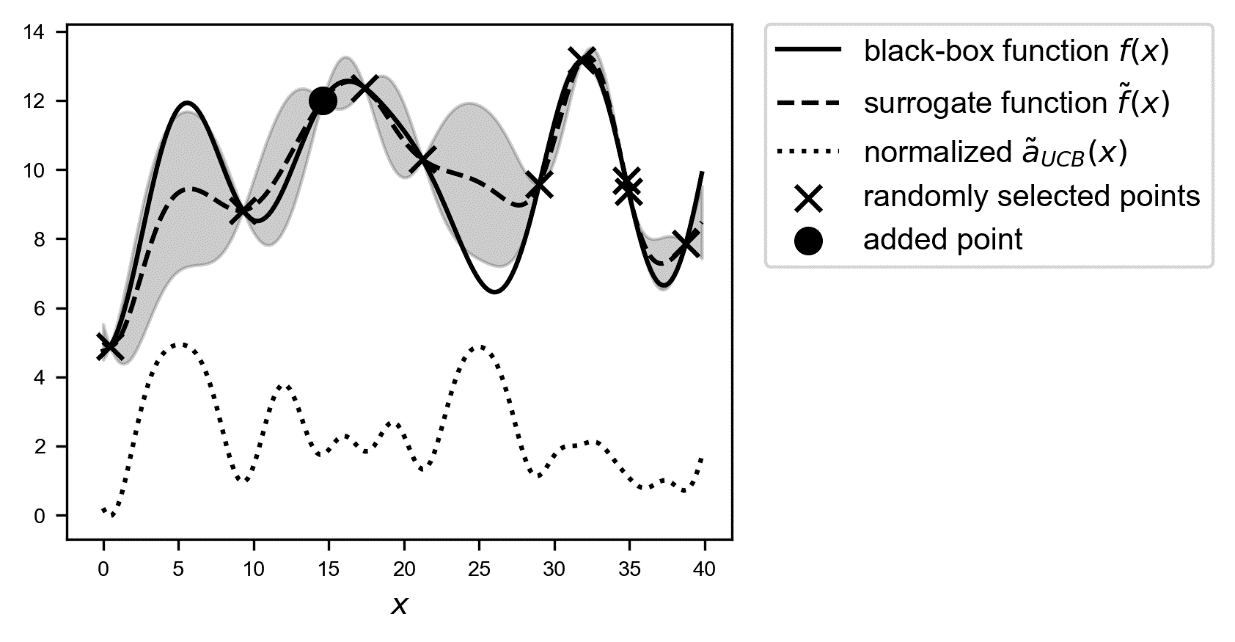


Figure 7-23. An overview of the black-box function , solid line, the randomly selected points with the additional selected points, marked by crosses, the surrogate function, dashed line, and the acquisition function , dotted line, shifted to fit in the plot. The gray area is the region contained between the lines   and .

The previous discussion and comparison of the behavior of the two types of acquisition function should have made clear how, depending on what strategy you want to apply to approximate your black-box function, you should choose the right acquisition function.

**Note** Different types of acquisition functions will give different strategies in approximating the black-box function. For example, will add points in regions with the highest variance, while will add points finding a balance, regulated by , between the maximum of the surrogate function and areas with high variance.

An analysis of all the different types of acquisition function would go far beyond the scope of this book. A good deal of research and reading of published papers is required to get enough experience and understand how different acquisition functions work and behave.

If you want to use Bayesian Optimization with your tensorflow model, you do not have to develop the method completely from scratch. You can try the library GPflowOpt from N. Knudde et al. [4].

## Sampling on a logarithmic scale

Now there is a last small subtlety that we need to discuss. Sometimes you will find yourself in a situation where you want to try a big range of possible values for a parameter, but you know from experience that probably the best value of it is in a specific range. Let us suppose you want to find the best value for the learning rate for your model, and you decide to test values from to , but you know, or at least expect, that your best value lies probably between and . Now let us suppose you are working with grid search and let us suppose you sample 1000 points. You may think you have enough points, but you will get

* 0 point between and
* 8 points between and
* 89 points between and
* 899 points between and

you get a lot more points in the less interesting ranges, and zero where you want them. In Figure 7-24 you can see the distribution of the points. Note that on the -axis I am using a logarithm scale. You can clearly see how you get much more points for bigger values of the learning rate.

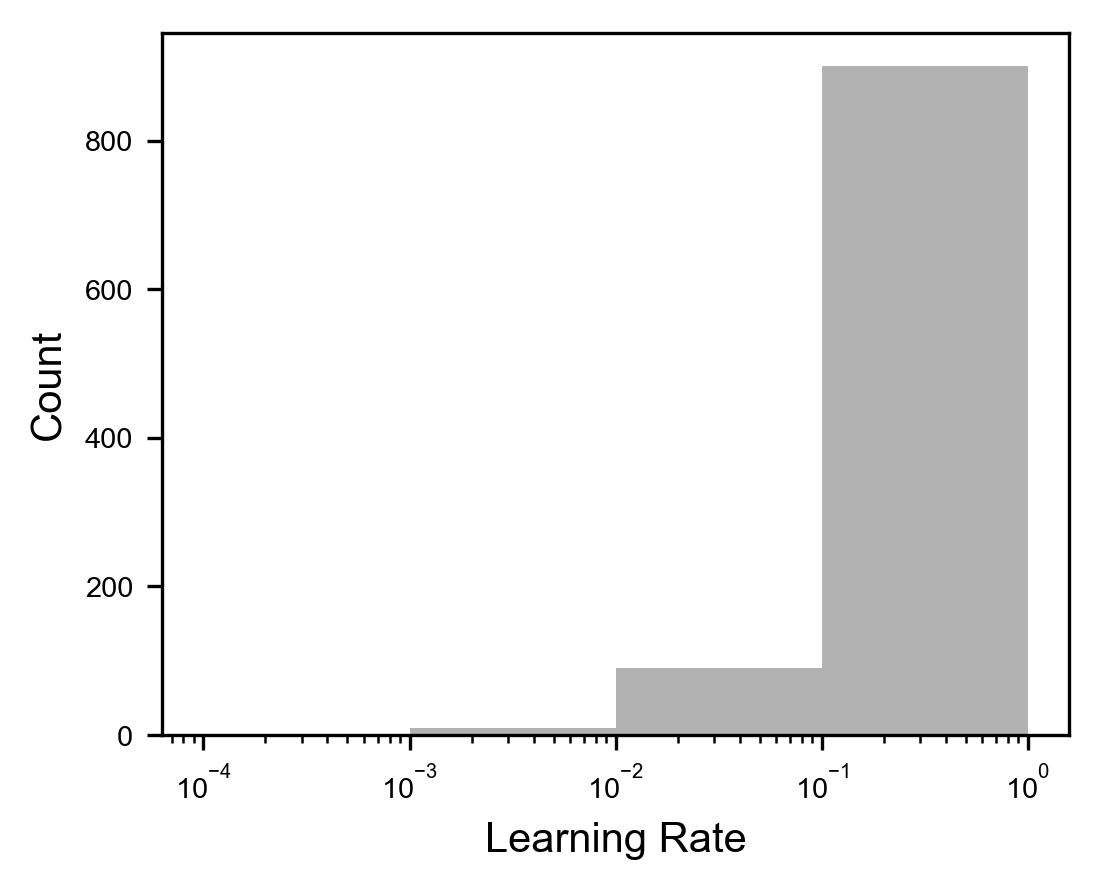


Figure 7-24. The distribution of 1000 points selected with grid search on a logarithmic x-scale.

You probably want to sample in a much finer way for smaller values of the learning rate than for bigger one. What you should do, is to sample your points on a logarithmic scale. The basic idea is that you want to sample the same number of points between and , and , and and and . To do that you can use the following Python code. First select a random number between 0 and minus the absolute value of the highest number of the power of 10 you have, in this case -4.

r = - np.arange(0,1,0.001)\*4.0

Then your array with the selected points can be created with

points2 = 10\*\*r

In Figure 7-25 you can see how now the distributions of the points contained in the array points2 is now completely flat, as we wanted.

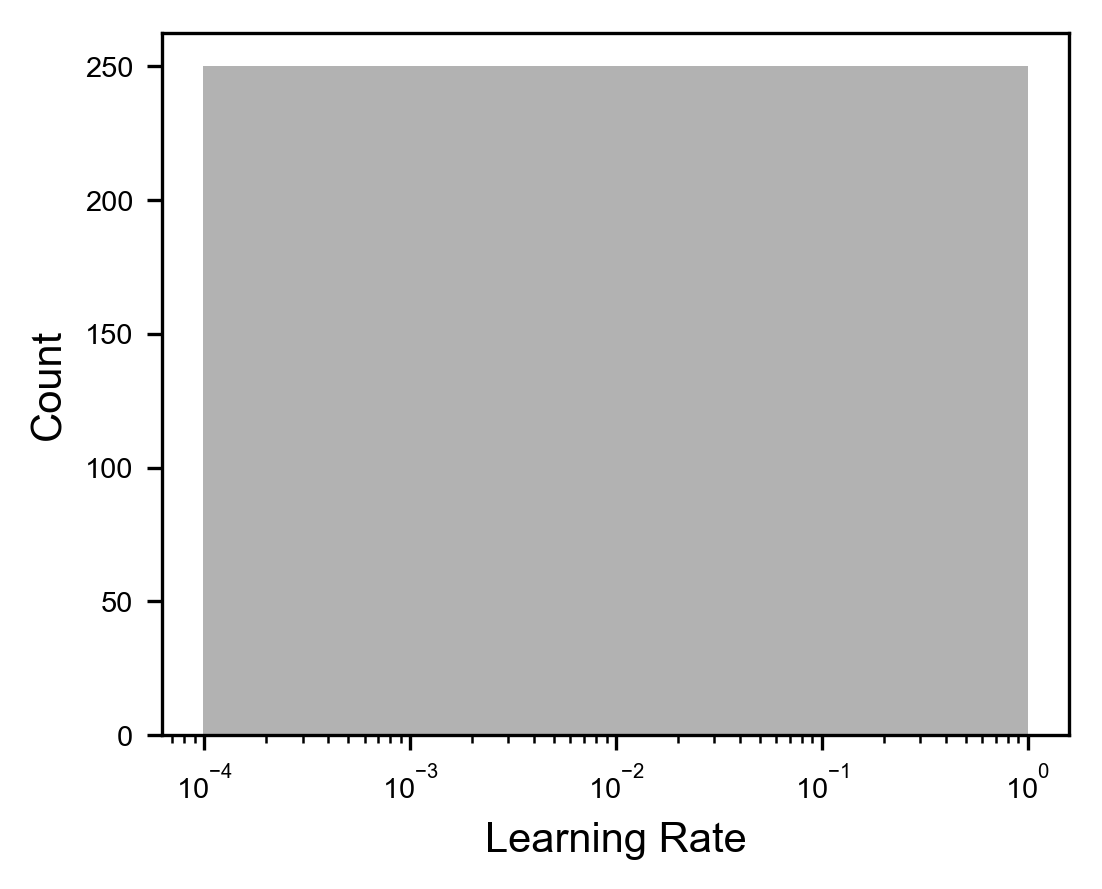


Figure 7-25. The distribution of 1000 points selected with grid search on a logarithmic x-scale with the modified selection method.

You get 250 points in each region, as you can easily check with this code for the range and . For the other ranges simply change the numbers in the code.

print(np.sum((alpha <= 1e-3) & (alpha > 1e-4)))

Now you can see how you have the same number of points between the different powers of 10. With this simple trick you can ensure that you get enough points also in region of your chosen range, where otherwise you would get almost no points. Remember that in this example, with 1000 points, with the standard method we get zero points between and . This range is the most interesting for the learning rate, so you want to have enough points in this range to optimize your model. Note that the same applies to random search. It works in the exact same way.

# Hyperparameter tuning with the Zalando dataset

To give you a concrete example of how hyperparameter tuning works, let us apply what we have learned in a simple case. Let us start with the data, as usual. Let us use the Zalando dataset we have used in Chapter 3. For a complete discussion please refer to the chapter. Let us quickly load and prepare the data, and then let us discuss tuning.

First, as usual, the necessary libraries

# 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()

Remember we have 60000 observations in the train dataset, and 10000 in the dev dataset. After executing the command above, 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 training dataset dimensions, as an example

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

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

which will return as output

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

Dimensions of the training labels: (60000,)

Remember that are the gray values of the image pixels (that have a size of pixels).

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.

The following lines reshape the matrices dimensions

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

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

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 a final problem. Labels must be provided in a different form, when performing a multiclass classification task. 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

Now we have prepared the data as we need. For a complete discussion refer to Chapter 3 where we spend lot of time discussion data preparation for this dataset. Now let us move on to the model. Let us start with something easy. As metric let us use for this example the accuracy, since the dataset is balanced. Let us consider a network with just one layer and let us see what number of neurons gives us the best accuracy. Our hyperparameter in this example will be the **number of neurons in the hidden layer**. Basically, we will need to build a new network for each value of the hyperparameter (the number of neurons in the hidden layer) and train it. We will need two functions: one to build the network and one to train it. To build the model we can define the following function

def build\_model(number\_neurons, learning\_rate):

# create model

model = keras.Sequential()

# add first hidden layer and set input dimensions

model.add(layers.Dense(number\_neurons, input\_dim = 784,

activation = 'relu'))

# add output layer

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

# compile model

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

optimizer = tf.keras.optimizers.SGD(

learning\_rate = learning\_rate),

metrics = ['categorical\_accuracy'])

return model

you should understand this function, since we have used the code several times in the book already. This function has an input parameter: number\_neurons, that will contain, as the name indicates, the number of neurons in the hidden layer. The function to train the model will look like this

def train\_model(number\_neurons, learning\_rate, number\_epochs,

mb\_size):

# build model

model = build\_model(number\_neurons, learning\_rate)

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = number\_epochs, verbose = 0,

batch\_size = mb\_size,

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

# test model

train\_loss, train\_accuracy = model.evaluate(

data\_train\_norm, labels\_train,

verbose = 0)

test\_loss, test\_accuracy = model.evaluate(

data\_test\_norm, labels\_test,

verbose = 0)

return train\_accuracy, test\_accuracy

you have already seen a function very similar to this one several times already. The main parts should be clear. You will find a few things that are new. First, we build the model in the function itself with

model = build\_model(number\_neurons, learning\_rate)

and additionally, we evaluate the accuracy on the train dataset and on the dev dataset and return the values to the caller. In this way we can run a loop for several values of the number of neurons in the hidden layer and get the accuracies. Note this time the function has an additional input parameter: number\_neurons. We need to pass this number to the function that builds the model.

Let us suppose we choose the following parameters: minibatch size = , we train for epochs, learning rate and we build our model with neurons in the hidden layer.

We then run the model

train\_accuracy, test\_accuracy = train\_model(15, 0.001, 100, 50)

print(train\_accuracy)

print(test\_accuracy)

we get for the train dataset and for the dev dataset accuracy. Can we do better? Well, we can surely do a grid search to start with

nn = [1, 5, 10, 15, 25, 30, 50, 150, 300, 1000, 3000]

for nn\_ in nn:

train\_accuracy, test\_accuracy = train\_model(nn\_, 0.001, 100, 50)

print('Number of neurons:', nn\_, 'Acc. Train:', acc\_train, 'Acc. Test', acc\_test)

Keep in mind that this will take quite some time. 3000 neurons are quite a high number, so be warned in case you want to try. We will get the following results (Table 8-1)

Table 8-1. Overview of the Accuracy on the Training and Test Datasets for a different number of neurons.

|  |  |  |
| --- | --- | --- |
| **Number of neurons** | **Accuracy on the train dataset** | **Accuracy on the test dataset** |
| 1 | 0.1000 | 0.1000 |
| 5 | 0.8242 | 0.8123 |
| 10 | 0.8476 | 0.8323 |
| 15 | 0.8549 | 0.8373 |
| 25 | 0.8604 | 0.8444 |
| 30 | 0.8577 | 0.8413 |
| 50 | 0.8661 | 0.8470 |
| 150 | 0.8677 | 0.8513 |
| 300 | 0.8694 | 0.8512 |
| 1000 | 0.8740 | 0.8555 |
| 3000 | 0.8754 | 0.8569 |

Not surprisingly more neurons deliver better accuracy, with no signs of overfitting of the train dataset, since the accuracy on the dev dataset is almost equal to the one on the train dataset. In Figure 7-26 you can see a plot of the accuracy on the test dataset vs. the number of neurons in the hidden layer. Note that the x-axis uses a logarithmic scale, to make the changes more evident.

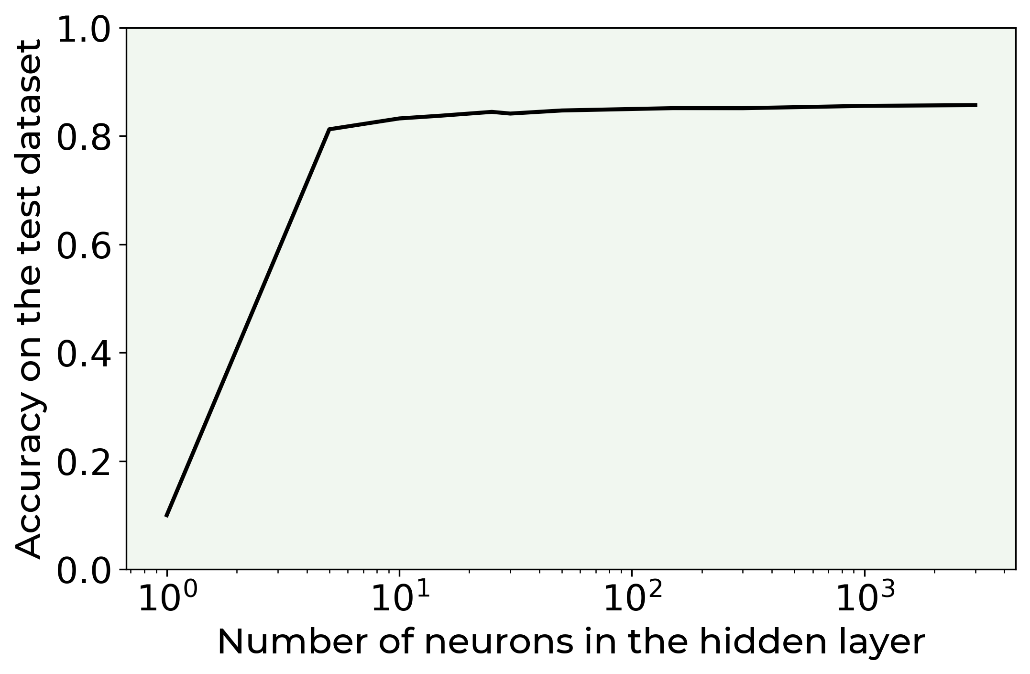


Figure 7-26. Accuracy on the test dataset vs. the number of neurons in the hidden layer.

If your goal would be to reach 80% of accuracy you could well stop here. But there are a few things to consider: first, we may be able to do better, and secondly training the network with 3000 neurons takes quite some time. On a medium performances laptop roughly 35 minutes. We should see if we can get the same result in a fraction of the time. We want a model that trains as fast as we can! Let us try a slightly different approach. Since we want to be faster let us consider a model with 4 layers. We could also tune the number of layers but let us stick to 4 for this example and let us tune the other parameters. We will try to find the optimal value for: learning rate, minibatch size, number of neurons in each layer and number of epochs. We will use random search. For each parameter we will select randomly 10 values:

* Number of neurons: between 35 and 60
* Learning rate: we will use the search on the logarithmic scale between and
* Minibatch size: between 20 and 80
* Number of epochs: between 40 and 100

We can create arrays with the possible values with this code

neurons\_ = np.random.randint(low = 35, high = 60.0, size = (10))

r = - np.random.random([10])\*3.0 - 1

learning\_ = 10\*\*r

mb\_size\_ = np.random.randint(low = 20, high = 80, size = 10)

epochs\_ = np.random.randint(low = 40, high = 100, size = (10))

Note that we will not try all possible combinations, but we will consider only ten possible combinations: the first value of each array, the second value of each array and so on. Let us consider how efficient random search can be with just 10 evaluations! We can test our model with the following loop

for i in range(len(neurons\_)):

train\_accuracy, test\_accuracy = train\_model(neurons\_[i], learning\_[i], epochs\_[i], mb\_size\_[i])

print()

print('Epochs:', epochs\_[i], 'Number of neurons:', neurons\_[i], 'Learning rate:', learning\_[i], 'Minibatch size', mb\_size\_[i],

'Training accuracy:', train\_accuracy, 'Test Accuracy', test\_accuracy)

You will find that the combinations with 44 epochs, 36 neurons in each layer, a learning rate of 0.07537 and a minibatch size of 54 gives you an accuracy on the dev dataset of 0.86. Not bad, considering that this run took 1.5 minutes, so 23 times faster than the model with 1 layer and 3000 neurons and 2% better. Our naïve initial test gave us an accuracy of 0.84, so with hyperparameter tuning we got 2% better than our initial guess. In the deep learning this is considered a good result. What we did should give you an idea of how powerful hyperparameter tuning can be if done properly. Keep in mind you should spend quite some time doing it, and especially thinking about how to do it.

**Note** Always think how you want to do your hyperparameter tuning and use your experience or ask for help to someone with experience. It is useless to invest time and resources to try combinations of parameters that you know will not work. For example, better spend time testing learning rates that are very small than testing learning rates around one. Remember that every training round of your network will cost time, even if the results are not useful!

The point of this last section is not to get the best model possible, but to give you an idea on how the tuning process may work. You could go on trying different optimizers (for example Adam), considering wider ranges for the parameters, more parameter combinations and so on.

# A quick note on the Radial Basis Function

Before finishing this chapter, let us discuss a minor point about the Radial Basis Function

It is important that you understand what the role of the parameter is. In our examples we have choosen but we have not discussed why. The reason is the following. Choosing too small, will make the acquisition function developing very narrow peaks around the points we already have as you can see in the left plot in Figure 7-27. Big values for will have a smoothing effect on the acquisition function as you can see in the middle and right plots in Figure 7-27.

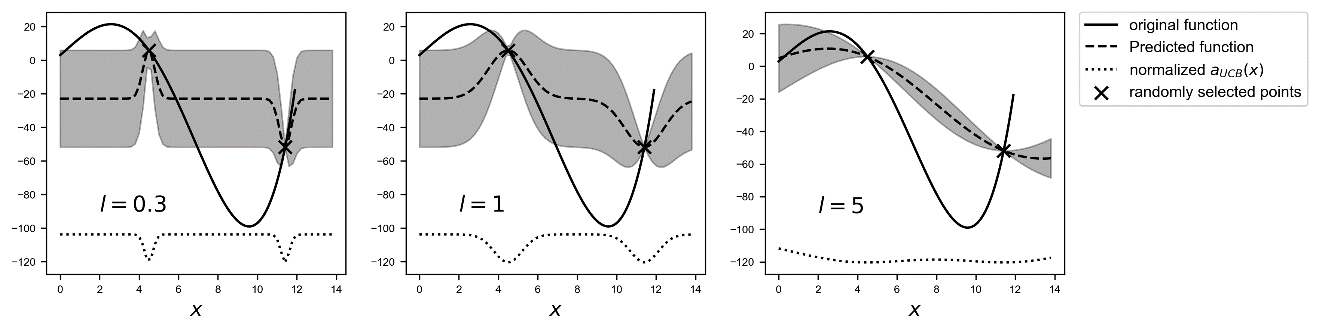


Figure 7-27. The effect of changing the parameter in the radial basis function.

Usually is good practice to avoid values for too small or too big, to be able to have a variance that varies in a smooth way between known points, as in Figure 7-27 for . Having a very small will make the variance between points almost constant and therefore make the algorithm almost always choose the middle point between points, as you can check from the acquisition function. While choosing big, will make the variance small, and therefore with some acquisition function difficult to use it. As you can see for in Figure 7-27 the acquisition function is almost constant. Typical values that are used are around 1 or 2.

# Exercises

EXERCISE 1 (EASY)

Try different optimizers (for example Adam) for the examples in this chapter and consider wider ranges for the parameters, more parameters combinations and so on. See how results change.

EXERCISE 2 (HARD)

Try to implement Bayesian Optimization from scratch as described in this chapter.

EXERCISE 3 (MEDIUM)

Try to optimize 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()

# References

[1] Bergstra, James, and Yoshua Bengio. "Random search for hyper-parameter optimization." *Journal of machine learning research* 13.2 (2012).

[2] Nadaraya, Elizbar A. "On estimating regression." Theory of Probability & Its Applications 9.1 (1964): 141-142.

[3] Do, Chuong B. "Gaussian processes." Stanford University, Stanford, CA, accessed Dec 5 (2007): 2017.

[4] Knudde, Nicolas, et al. "GPflowOpt: A Bayesian optimization library using tensorflow." *arXiv preprint arXiv:1711.03845* (2017).

1. Note that the sometime learning rate can be changed during training, for example when applying decaying rate approaches. But often when using more advanced optimizers as Adam the learning rate is assigned at the beginning and not changed anymore during training. [↑](#footnote-ref-1)
2. Remember that a normal distribution has an exponential form, and the ratio of two exponential is still an exponential. [↑](#footnote-ref-2)