Regularization

# Complex networks and overfitting

In the previous chapters you have learned how to build and train complex networks. One of the most common problems you will encounter when using complex networks is overfitting. Check again Chapter 3 for an overview of what it is. In this chapter we will face an extreme case of overfitting and we will discuss few strategies to avoid it. A perfect dataset to study this problem is the Boston housing price dataset we have already discussed in Chapter 2. Let's review how to get the data (for a more detailed discussion please refer to Chapter 2). Let's start with the packages we need

import matplotlib.pyplot as plt

%matplotlib inline

import tensorflow as tf

import numpy as np

from sklearn.datasets import load\_boston

import sklearn.linear\_model as sk

and then import the dataset

boston = load\_boston()

features = np.array(boston.data)

target = np.array(boston.target)

The dataset has 13 features (contained in the features numpy array) and the house price contained in the target numpy array. To normalize the features, we will use (as in Chapter 2) the function

def normalize(dataset):

mu = np.mean(dataset, axis = 0)

sigma = np.std(dataset, axis = 0)

return (dataset-mu)/sigma

to conclude our dataset preparation let's normalize it and then create a training and a dev datasets

features\_norm = normalize(features)

np.random.seed(42)

rnd = np.random.rand(len(features\_norm)) < 0.8

train\_x = np.transpose(features\_norm[rnd])

train\_y = np.transpose(target[rnd])

dev\_x = np.transpose(features\_norm[~rnd])

dev\_y = np.transpose(target[~rnd])

The np.random.seed(42) is there so that you will always get the same training and dev dataset (in this way your results will be reproducible). Now let's reshape the arrays we need

train\_y = train\_y.reshape(1,len(train\_y))

dev\_y = dev\_y.reshape(1,len(dev\_y))

then let's build a complex neural network with 4 layers and 20 neurons for each layer. To build it let define this function to build each layer

def create\_layer (X, n, activation):

ndim = int(X.shape[0])

stddev = 2.0 / np.sqrt(ndim)

initialization = tf.truncated\_normal((n, ndim), stddev = stddev)

W = tf.Variable(initialization)

b = tf.Variable(tf.zeros([n,1]))

Z = tf.matmul(W,X)+b

return activation(Z), W, b

Note how this time we return the weights tensor W and the bias b. We will need them when implementing regularization. You have already seen this function at the end of Chapter 3, so you should understand what it does. We use here He initialization, since we will use ReLu activation functions. The network can be created with the code

tf.reset\_default\_graph()

n\_dim = 13

n1 = 20

n2 = 20

n3 = 20

n4 = 20

n\_outputs = 1

tf.set\_random\_seed(5)

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

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

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

hidden1, W1, b1 = create\_layer (X, n1, activation = tf.nn.relu)

hidden2, W2, b2 = create\_layer (hidden1, n2, activation = tf.nn.relu)

hidden3, W3, b3 = create\_layer (hidden2, n3, activation = tf.nn.relu)

hidden4, W4, b4 = create\_layer (hidden3, n4, activation = tf.nn.relu)

y\_, W5, b5 = create\_layer (hidden4, n\_outputs, activation = tf.identity)

cost = tf.reduce\_mean(tf.square(y\_-Y))

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

In our output layer we have one neuron with the identity activation function for regression. Additionally, we use the Adam optimizer as suggested in the previous Chapter. Now let's run the model with this code

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

cost\_train\_history = []

cost\_dev\_history = []

for epoch in range(10000+1):

sess.run(optimizer, feed\_dict = {X: train\_x, Y: train\_y, learning\_rate: 0.001})

cost\_train\_ = sess.run(cost, feed\_dict={ X:train\_x, Y: train\_y, learning\_rate: 0.001})

cost\_dev\_ = sess.run(cost, feed\_dict={ X:dev\_x, Y: dev\_y, learning\_rate: 0.001})

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

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

if (epoch % 1000 == 0):

print("Reached epoch",epoch,"cost J(train) =", cost\_train\_)

print("Reached epoch",epoch,"cost J(test) =", cost\_test\_)

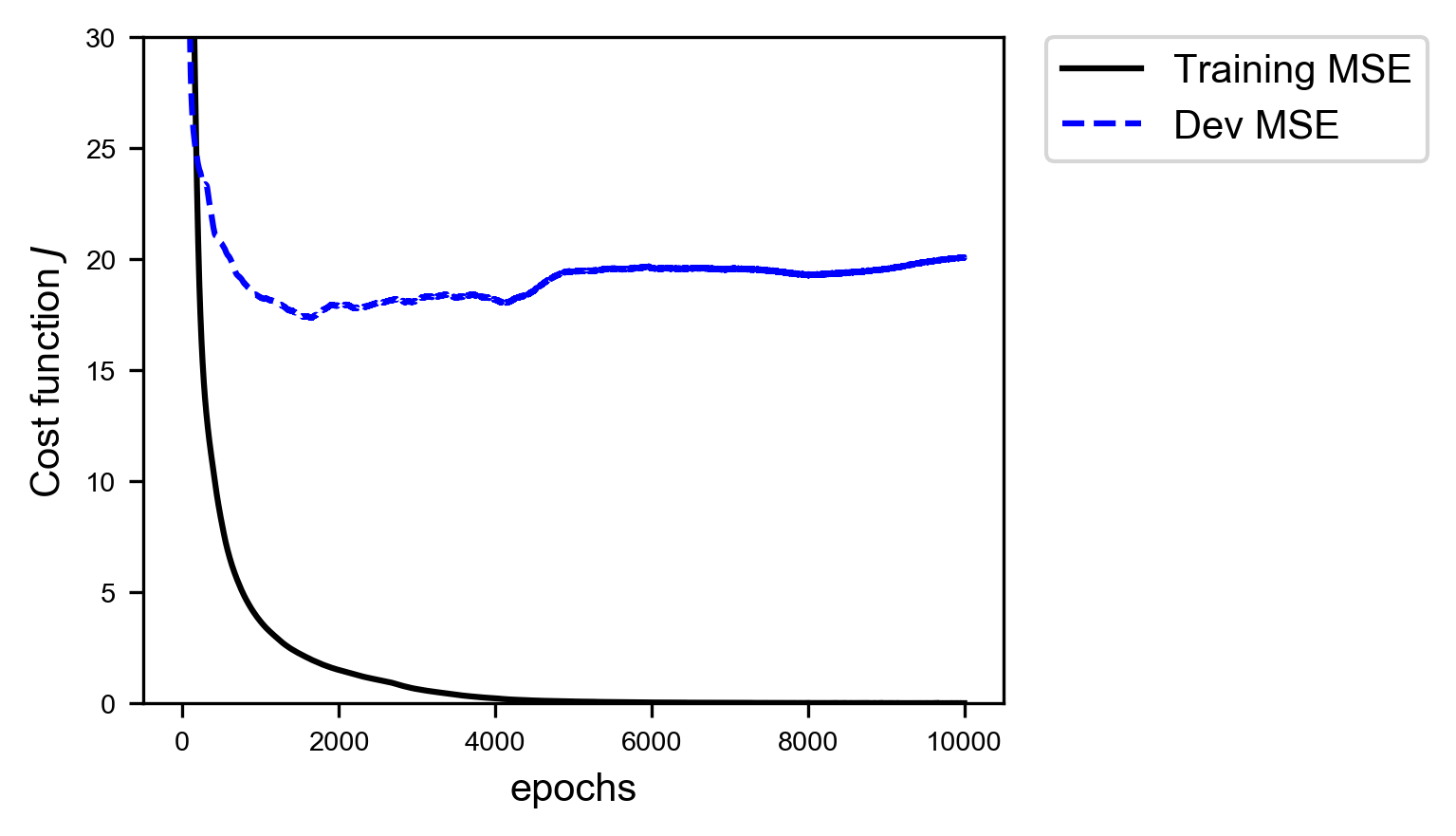
As you may notice there are a few differences with what we did before. To make things simpler, I just avoided writing a function and simply hard coded all the values in the code, since in this case we don't need to tune much the parameters. I am not using mini batches here, since we have only few hundred observations and I am calculating the MSE for both training and dev datasets with the lines

cost\_train\_ = sess.run(cost, feed\_dict={ X:train\_x, Y: train\_y, learning\_rate: 0.001})

cost\_dev\_ = sess.run(cost, feed\_dict={ X:dev\_x, Y: dev\_y, learning\_rate: 0.001})

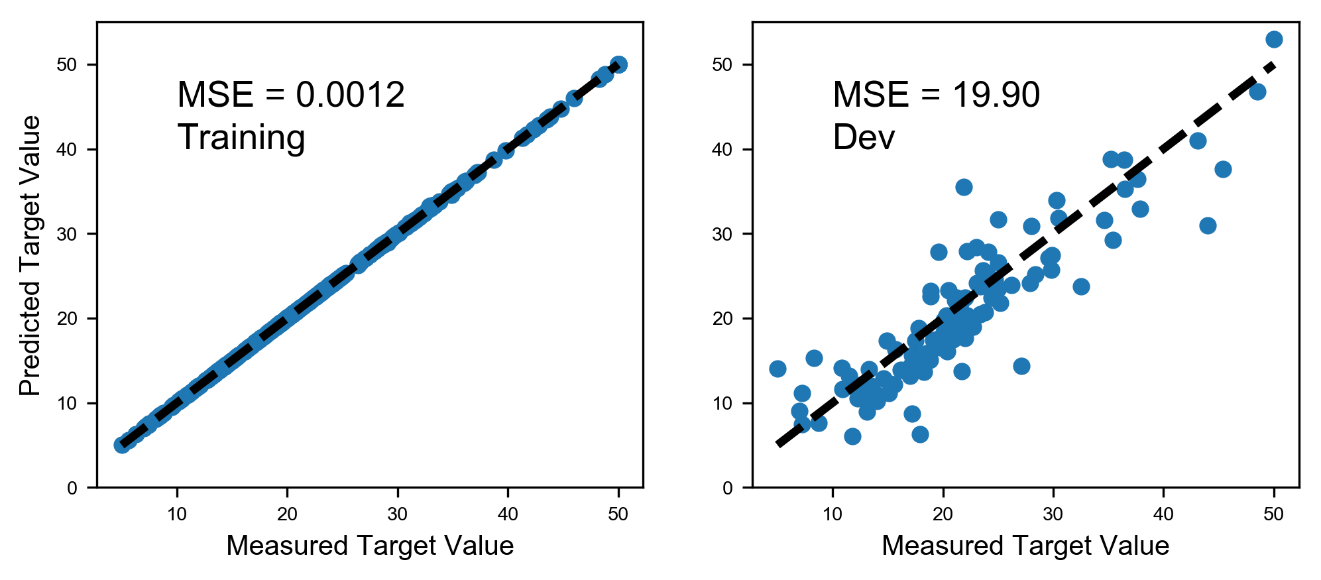
In this way we can check what is happening on both datasets at the same time. Now if you let the code run and plot the two MSEs, one for the training that we will indicate with and one for the dev dataset indicated with , we get Figure 5-1.

Figure 5-1: the MSE for the training (continous line) and the dev dataset (dahsed line) for the neural network with 4 layers, each having 20 neurons.



You will notice how the training error goes down to zero while the dev error remains constant at around a value of roughly 20 after dropping rapidly at the beginning. If you remember our basic error analysis introduction, you should know that this means that we are in a regime of extreme overfitting (when ). The error on the training dataset is practically zero, while the one on the dev dataset is not. The model cannot generalize at all when applied to new data. In Figure 5-2 you can see the predicted value plotted versus the real value. You will notice how in the left plot, for the training data, the prediction is almost perfect, while on the plot on the right, for the dev dataset, is not that good. You will remember that a perfect model would give you predicted values exactly equal to the measured ones. So while plotting one versus the other they would all lie on the 45 degrees line in the plot, as it is happening in Figure 5-2 on the left plot.

Figure 5-2: predicted value versus the real value for the target variable (the house price). You will notice how in the left plot, for the training data, the prediction is almost perfect, while on the plot on the right, for the dev dataset, the predictions are more spread.



What can we do in this case to avoid the problem of overfitting? One solution would be of course reducing the complexity of the network. Reducing the number of layers and/or the number of neurons in each layer. But, as you can imagine, this strategy is very time consuming. You must try several network architectures to see how the training error and the dev error behave. In this case this is still a viable solution, but if you are working on a problem where the training phase takes several days this can be quite difficult and extremely time consuming. Several strategies have been developed to deal with this problem, the most common is called regularization and the focus of this chapter.

# What is regularization

Before going into the different methods, I would like to quickly discuss what the deep learning community understand with the term "regularization". The term has deeply (pun intended) evolved over time. For example, in the traditional sense from the 90s the term is reserved only to a penalty term in the loss function (Bishop, C.M, (1995) Neural Networks for Pattern Recognition, Oxford University Press). Lately the term has gained a much more broader meaning. For example Goodfellow (Goodfellow, I.J. et al., Deep Learning, MIT Press) define it as any modification we make to a learning algorithm that is intended to reduce its test error but not its training error. Kukačka (Kukačka, J. et al., Regularization for deep learning: a taxonomy, arXiv:1710.10686v1, available here: https://goo.gl/wNkjXz) is generalizing the term even more and gives the definition: Regularization is any supplementary technique that aims at making the model generalize better, i.e. produce better results on the test set. So be aware when using the term and always be precise with what you mean.

You may also have heard or read the claim that regularization has been developed to fight overfitting. This is also a way of understanding it. Remember a model that is overfitting the training dataset is not generalizing well to new data. This definition is also on-line with all the others. This is just a matter of definition but is important to have heard them, so that you may better understand what is meant when reading papers or books. This is a very active research area and to give you an idea Kukačka, in his review paper referenced above, list 58 different regularization methods. Yes 58, that is not a typo. But is important to understand that with their general definition also SGD (Stochastic Gradient Descent) is considered a regularization method, something that not everyone agrees on. So be warned when reading research material and check what people understand with the term regularization.

We will look in this chapter at the three most common and known methods: , and Dropout, plus we will briefly talk about early stopping, although this method does not, technically speaking, fight overfitting. and achieve a so-called weight decay by adding a so-called regularization term to the cost function, while dropout will simply remove, in a random fashion, nodes from the network during the training phase. To understand the three methods properly we need to study them in detail. Let's start with probably the most instructive one: regularization.

At the end of the chapter we will look at few other ideas on how to fight overfitting and gets the model generalize better. Instead of changing or modifying the model or the learning algorithm we will consider strategies with the idea of modifying the training data to make learning more effective.

## About network complexity

I would like to spend a few moments discussing briefly the term we used very often: network complexity. You have read here, and can find almost everywhere, that with regularization we want to reduce network complexity. But what are we referring to really? Actually, is very difficult to give a definition of network complexity, so much that nobody actually does it. You find several research papers on the problem of model complexity (note I did not say network complexity), with roots in information theory. You will see in this chapter how for example the number of weights that is different than zero will change dramatically with the number of epochs, with the optimization algorithm and so on, therefore making this vaguely intuitive concept of complexity also dependent on how long you train your model. To make the story short the term network complexity should be used only on an intuitive level, since theoretically is a very complex concept to define. A complete discussion on the subject would be completely out of scope for this book.

# norm

Before we start studying what and regulariazion are, we need to introduce the norm notation. We define the norm of a vector with components as

where the sum is performed over all components of the vector .

Let's now start with the most instructive norm: the .

# regularization

## Theory of regularization

When doing plain regression, you will remember from Chapter 2, our cost function is simply the MSE (Mean Squared Error)

where is our measured target variable, is the predicted value, is the vector of all the weights of our network including the bias, and is the number of observations. Now let's define a new cost function

This additional term

Is called a regularization term and is nothing else than the -norm squared of multiplied by a constant factor . is a called the regularization parameter.

The new regularization parameter is a new hyper parameter that you need to tune to find the optimal value.

Now let's try to get an intuitive understanding of what is the effect of this term on the GD (Gradient Descent) algorithm. Let's consider the update equation for the weight

Since

This gives us

This is the equation that we need to use for the weights update. The difference with the one we already know from plain GD, is that now the weight is multiplied with a constant and therefore this has the effect of effectively shifting the weight values during the update toward zero, and therefore making the network less complex (intuitively), fighting in this way overfitting. Let's try to see what is really happening to the weights by applying the method to the Boston housing dataset.

## Tensorflow implementation

The implementation in tensorflow is quite easy. Remember, we need to calculate the additional term , then add it to the cost function. The model construction remains almost the same. We can do it with the code

tf.reset\_default\_graph()

n\_dim = 13

n1 = 20

n2 = 20

n3 = 20

n4 = 20

n\_outputs = 1

tf.set\_random\_seed(5)

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

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

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

hidden1, W1, b1 = create\_layer (X, n1, activation = tf.nn.relu)

hidden2, W2, b2 = create\_layer (hidden1, n2, activation = tf.nn.relu)

hidden3, W3, b3 = create\_layer (hidden2, n3, activation = tf.nn.relu)

hidden4, W4, b4 = create\_layer (hidden3, n4, activation = tf.nn.relu)

y\_, W5, b5 = create\_layer (hidden4, n\_outputs, activation = tf.identity)

lambd = tf.placeholder(tf.float32, shape=())

reg = tf.nn.l2\_loss(W1) + tf.nn.l2\_loss(W2) + tf.nn.l2\_loss(W3) + \

tf.nn.l2\_loss(W4) + tf.nn.l2\_loss(W5)

cost\_mse = tf.reduce\_mean(tf.square(y\_-Y))

cost = tf.reduce\_mean(cost\_mse + lambd\*reg)

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

for our new regularization parameter we create a placeholder

lambd = tf.placeholder(tf.float32, shape=())

Remember that in Python lambda is a reserved word, so we cannot use it, this is the reason why we use lambd. Then we calculate our regularization term

reg = tf.nn.l2\_loss(W1) + tf.nn.l2\_loss(W2) + tf.nn.l2\_loss(W3) + \

tf.nn.l2\_loss(W4) + tf.nn.l2\_loss(W5)

with the useful tensorflow function tf.nn.l2\_loss() and then we add it to the MSE function cost\_mse

cost\_mse = tf.reduce\_mean(tf.square(y\_-Y))

cost = tf.reduce\_mean(cost\_mse + lambd\*reg)

Now our cost tensor will contain the MSE plus the regularization term. Then we simply need to train the network and see what happens. To train the network we use this function

def model(training\_epochs, features, target, logging\_step = 100, learning\_r = 0.001, lambd\_val = 0.1):

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

cost\_history = []

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

sess.run(optimizer, feed\_dict = {X: features, Y: target, learning\_rate: learning\_r, lambd: lambd\_val})

cost\_ = sess.run(cost\_mse, feed\_dict={ X:features, Y: target, learning\_rate: learning\_r, lambd: lambd\_val})

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

if (epoch % logging\_step == 0):

pred\_y\_test = sess.run(y\_, feed\_dict = {X: test\_x, Y: test\_y})

print("Reached epoch",epoch,"cost J =", cost\_)

print("Training MSE = ", cost\_)

print("Dev MSE = ", sess.run(cost\_mse, feed\_dict = {X: test\_x, Y: test\_y}))

return sess, cost\_history

This time I print the MSE coming from training () and from dev () datasets to check what is going on. As mentioned applying this method make many weights go to zero, effectively reducing the complexity of the network, and therefore fighting overfitting. Let's run the model for , without regularization, and for . We can run our model with the code

sess, cost\_history = model(learning\_r = 0.01,

training\_epochs = 5000,

features = train\_x,

target = train\_y,

logging\_step = 5000,

lambd\_val = 0.0)

and that gives us

Reached epoch 0 cost J = 238.378   
Training MSE = 238.378   
Dev MSE = 205.561   
Reached epoch 5000 cost J = 0.00527479   
Training MSE = 0.00527479   
Dev MSE = 28.401

and as expected, we are in an extreme overfitting regime () after 5000 epochs. Now let's try with

sess, cost\_history = model(learning\_r = 0.01,

training\_epochs = 5000,

features = train\_x,

target = train\_y,

logging\_step = 5000,

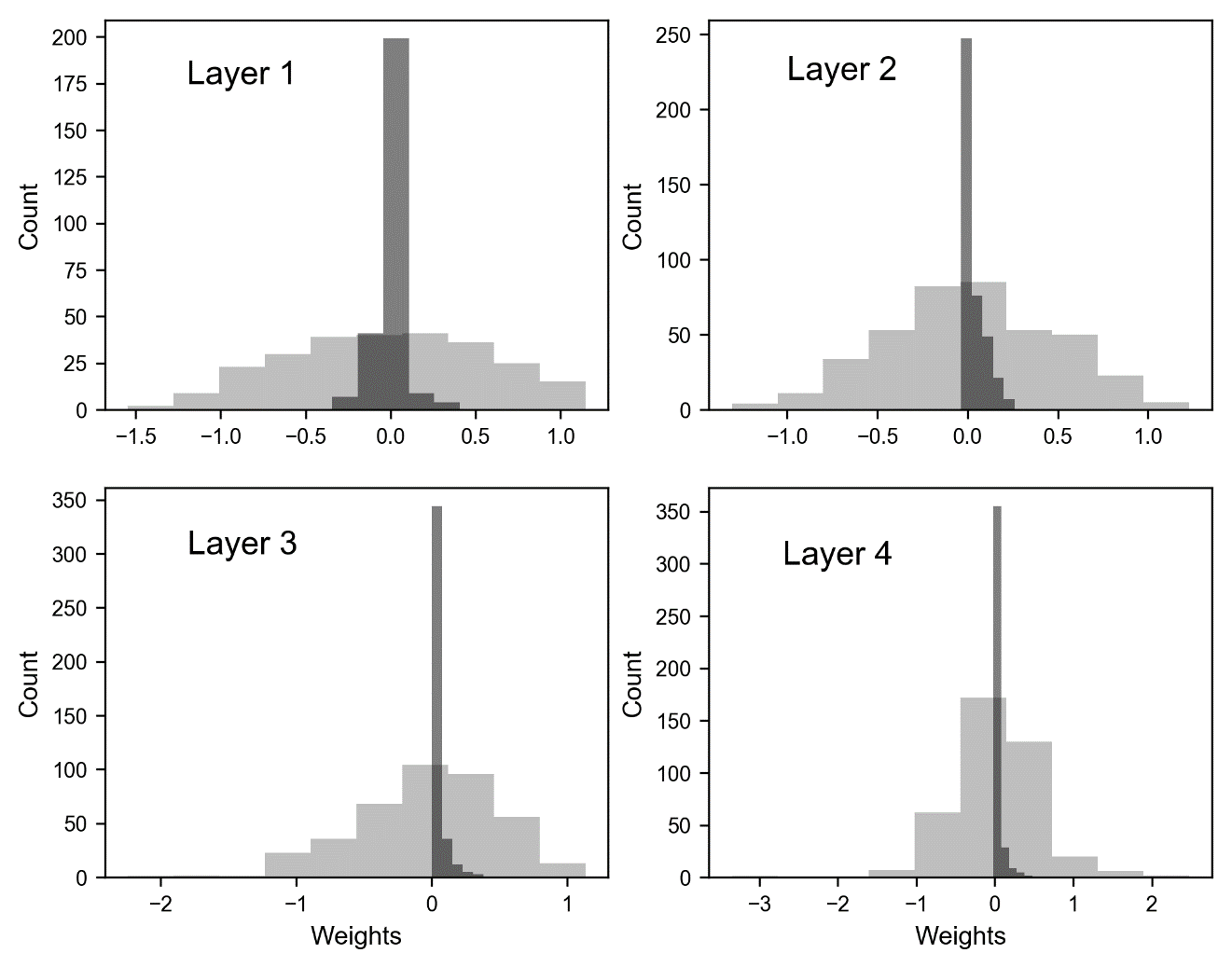
lambd\_val = 10.0)

and that gives the results

Reached epoch 0 cost J = 248.026   
Training MSE = 248.026   
Dev MSE = 214.921   
Reached epoch 5000 cost J = 23.795   
Training MSE = 23.795   
Dev MSE = 21.6406

Now we are no more in an overfitting regime, since the two MSE values are of the same order of magnitude. The best way of checking what is going on is to study the weights distribution for each layer. In Figure 5-3 the weights distribution for the first 4 layers are plotted. The light gray histogram is for the weights without regularization, and the darker (and much more concentrated around zero) is for the weights with regularization.

Figure 5-3: weights distribution for each layer. The light gray histogram is for the weights without regularization, and the darker (and much more concentrated around zero) is for the weights with regularization. I neglected layer 5, since is the output layer.



You can clearly see how the weights, when we apply regularization, are much more concentrated around zero, meaning they are much smaller than without regularization. This make the weight decay effect of regularization very evident. I would like to briefly take the chance and make another brief digression on network complexity. I said that this method reduces the network complexity. I told you previously in Chapter 3 that you can consider the number of learnable parameters an indication of the complexity of a network, but I also warned you that this can be very misleading. I would like to show you now why. You will remember from Chapter 3, that the total number of learnable parameters we have in a network like the one we are using here is given by the formula

Where is the number of neurons in layer and is the total number of layers, including the output layer. In our case we have an input layer with 13 features, then 4 layers with each 20 neurons and then an output layer with 1 neuron. Therefore is given by

is quite a big number. But already without regularization is interesting to note that we have roughly 48% of the weights that after 10000 epochs are less than , so effectively zero. This is the reason I warned you about talking about complexity in terms of number of learnable parameters. Additionally using regularization will change completely the scenario. Complexity is a difficult concept to define: it depends on many things, between others from architecture, optimization algorithm, cost function and number of epochs trained.

Defining the complexity of a network only in terms of number of weight is not completely correct. The total number of weights gives an idea, but it can be quite misleading since many may be zero after the training, effectively disappearing from the network, and making it less complex. Is more correct to talk about "Model Complexity" instead of network complexity, since many more aspects are involved than simply how many neurons or layers the network have.

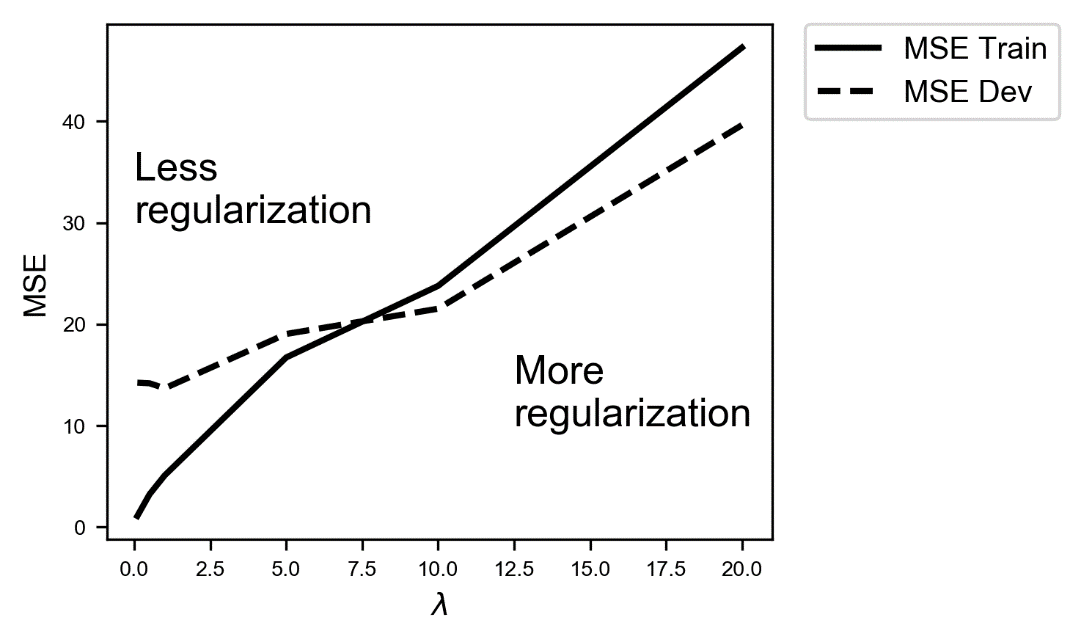
Incredibly enough only half of the weights play a role in the predictions at the end. This is the reason I told you in Chapter 3 that defining the network complexity only with the parameter is misleading. Given your problem, your loss function and optimizer you may well end up with a network that when trained is much simpler than what it was at construction phase. So be very careful when using the term complexity in the deep learning world. Be aware of the subtleties involved.

To give you an idea of how effective regularization is in reducing the weights, check the following table where we compare the percentage of weights less than 1e-3 with and without regularization after 1000 epochs in each layer.

|  |  |  |
| --- | --- | --- |
| Layer | % of weights less than 1e-3 for | % of weights less than 1e-3 for |
| 1 | 0.0 | 20.0 |
| 2 | 0.25 | 41.5 |
| 3 | 0.75 | 60.5 |
| 4 | 0.25 | 66.0 |
| 5 | 0.0 | 35.0 |

But how should we choose ? To get an idea (repeat with me: in the deep learning world there is no universal rule) is useful to see what is happening when varying the parameter to your optimizing meric (in this case the MSE). In Figure 5-4 you can see the behavior of (continuous line) and for (dashed) dataset for our network varying after 1000 epochs.

Figure 5-4: behavior of the MSE for the training (continuous line) dataset and for the dev (dashed) dataset for our network varying .



As you can see with small values of (effectively without regularization) we are in an overfitting regime (), slowly the increases, while the remains roughly constant. Until the model overfit the training data, then the two values cross and the overfitting finishes, and after that they grow together, at which point the model cannot capture the fine data structures anymore. After the crossing of the lines the model is getting too simple to capture the features of the problem, and therefore the errors grow together, and also the error on the training dataset gets bigger, since the model don't even fit well the training data. In this specific case a good value to choose for would be around , around the value when the two lines cross, since there you are anymore in an overfitting region since . Remember the main goal of having the regularization term is to get a model that generalize in the best way possible when applied to new data. You can look at it in an even different way: a value of gives you the minimum of outside the overfitting region (for , therefore would be a good choice. Note that you may observe for your problems a very different behavior for your optimizing metric, so you will have to decide on a case by case basis what is the best value for that works for you.

A good way for estimating the optimal value of the regularization parameter is to plot your optimizing metric (in this example the MSE) for the training and the dev dataset and see how they behave for various values of and then choose the value that will give the minimum of your optimizing metric on the dev dataset and at the same time gives you a model that is not overfitting your training data anymore.

I would like now to show you what are the effects of regularization in an even more visual way. Let's consider now a dataset generated with the following code

nobs = 30

np.random.seed(42)

xx1 = np.array([np.random.normal(0.3,0.15) for i in range (0,nobs)])

yy1 = np.array([np.random.normal(0.3,0.15) for i in range (0,nobs)])

xx2 = np.array([np.random.normal(0.1,0.1) for i in range (0,nobs)])

yy2 = np.array([np.random.normal(0.3,0.1) for i in range (0,nobs)])

c1\_ = np.c\_[xx1.ravel(), yy1.ravel()]

c2\_ = np.c\_[xx2.ravel(), yy2.ravel()]

c = np.concatenate([c1\_,c2\_])

yy1\_ = np.full(nobs, 0, dtype=int)

yy2\_ = np.full(nobs, 1, dtype=int)

yyL = np.concatenate((yy1\_, yy2\_), axis = 0)

train\_x = c.T

train\_y = yyL.reshape(1,60)

Our dataset has two features and . We generate two group of points xx1,yy1 and xx2,yy2 from a normal distribution. To the first we assign the label 0 (contained in the array yy1\_) and to the second the label 1 (in the array yy2\_). Now let's use a network like what we described before (with 4 layers, each having 20 neurons) to do some binary classification on this dataset. We can take the same code given before, modifying the output layer and the cost function. You will remember that for binary classification we need one neuron in the output layer with the sigmoid activation function

y\_, W5, b5 = create\_layer (hidden4, n\_outputs, activation = tf.sigmoid)

and the following cost function

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

cost = tf.reduce\_mean(cost\_class + lambd\*reg)

all the rest remains the same of what we described earlier. Let's plot the decision boundary[[1]](#footnote-1) for this problem. That means we will run our network on our dataset with the code

sess, cost\_history = model(learning\_r = 0.005,

training\_epochs = 100,

features = train\_x,

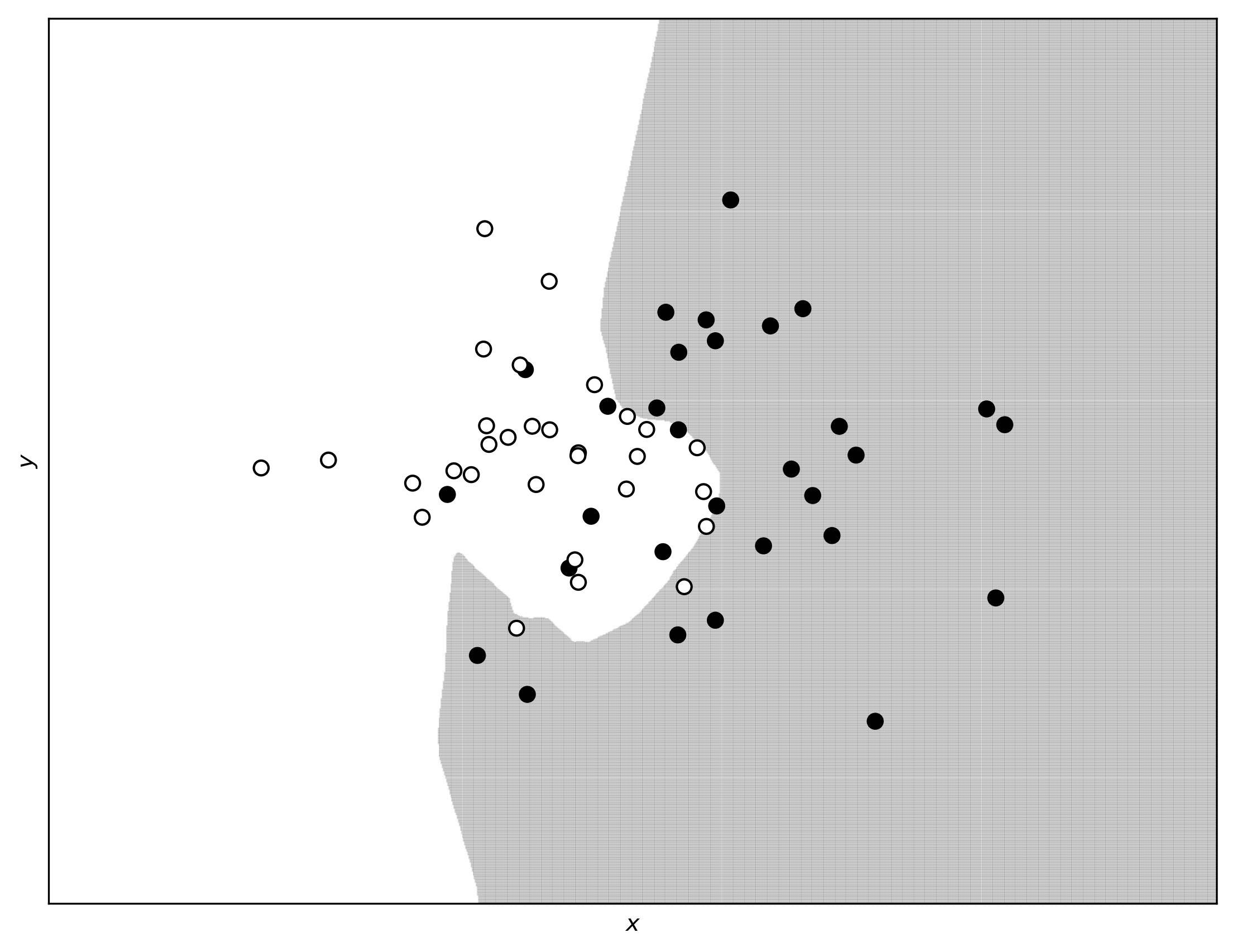
target = train\_y,

logging\_step = 10,

lambd\_val = 0.0)

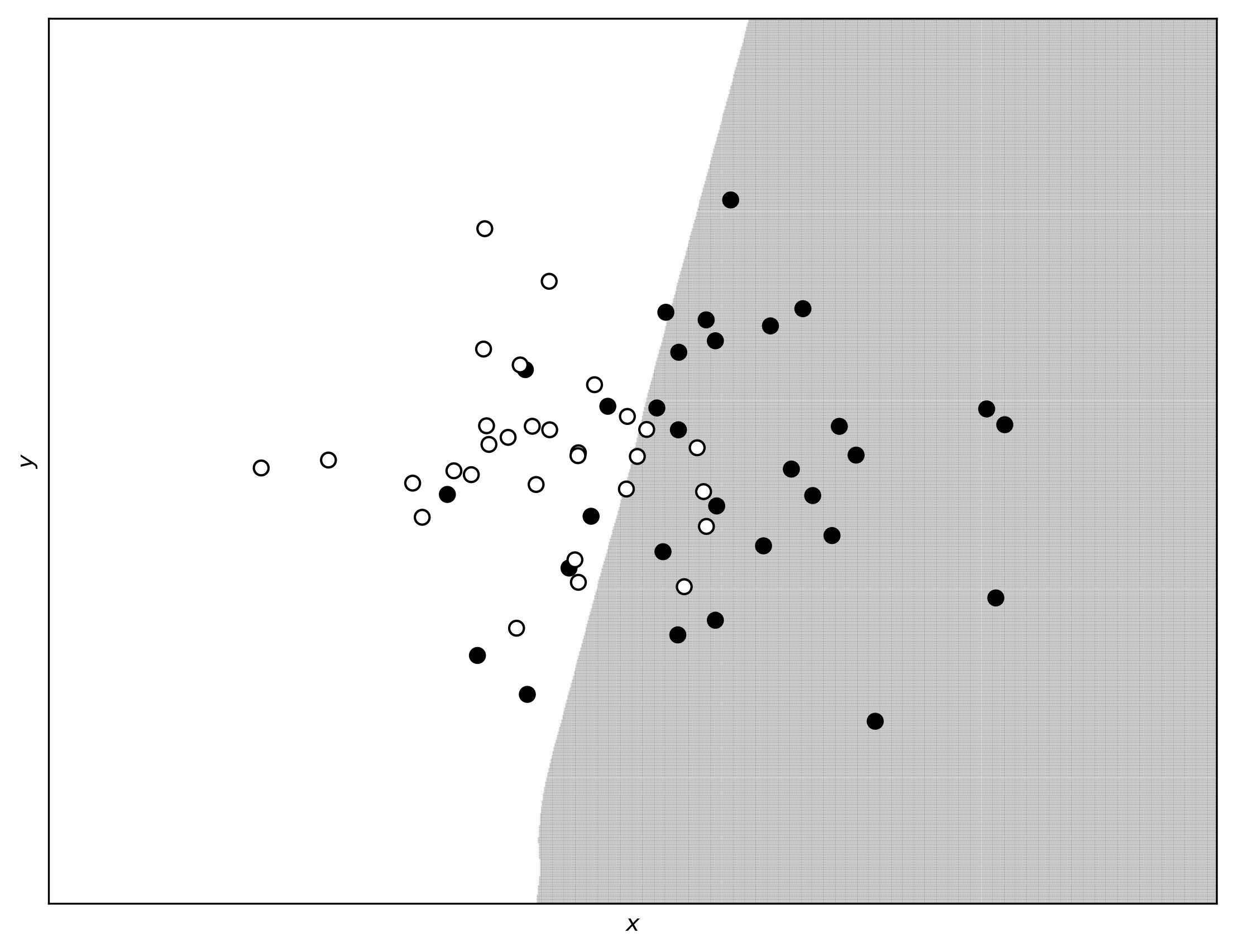
In Figure 5-5, you can see our datasets where the white points are of one class and the black of the second. The gray area is the zone that the network classifies being of one class and the white to the other. You can see that the network is able to capture the complex structure of our data in a flexible way.

Figure 5-5: decision boundary without regularization. White points are of one class and the black of the second. The gray area is the zone that the network classifies being of one class and the white to the other. You can see that the network is able to capture the complex structure of our data.



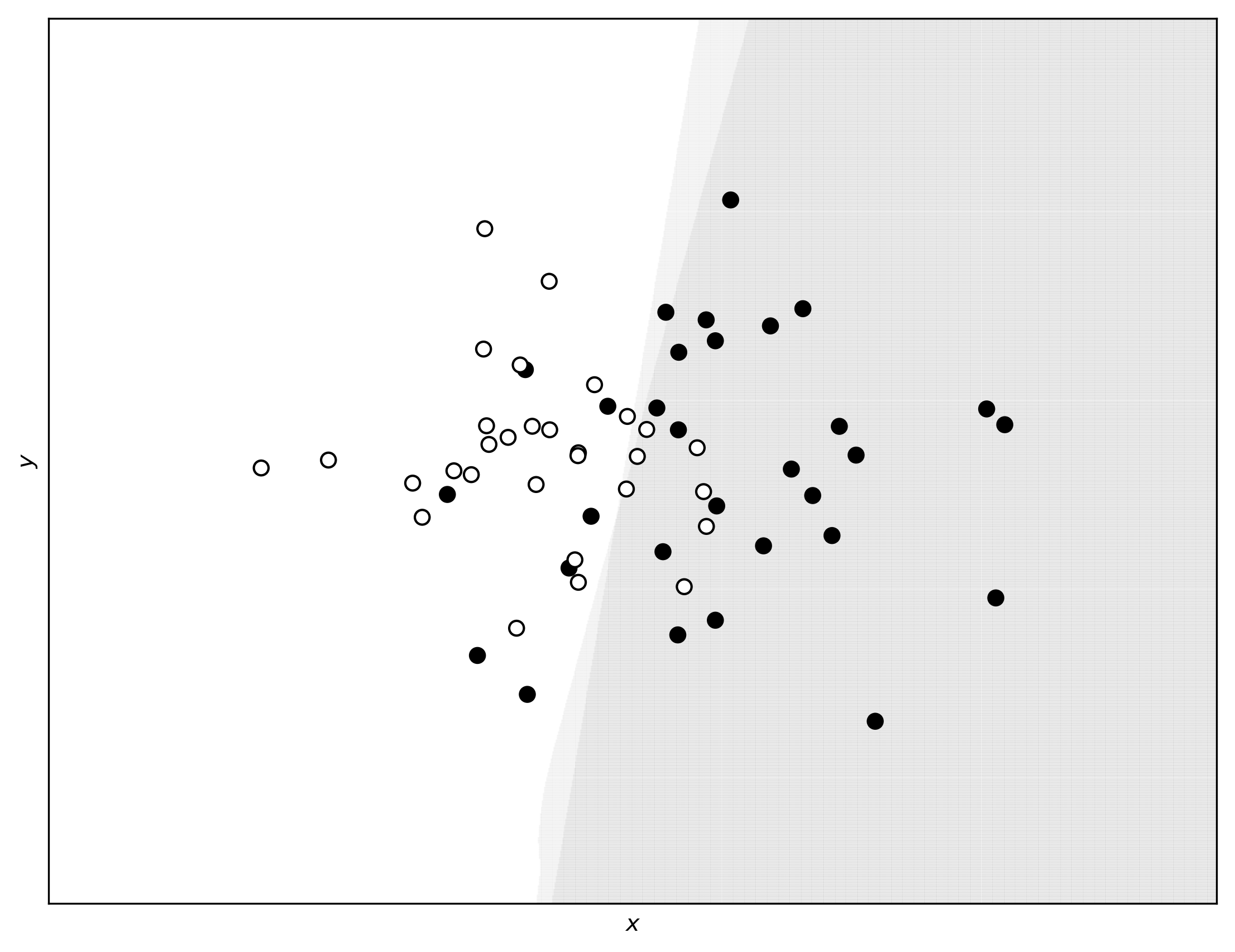
Now let's apply regularization to the network, exactly as we did before, and let see how the decision boundary is modified. We will use here a regularization parameter .

Figure 5-6: the decision boundary as predicted by the network with regularization and with a regularization parameter .



You can clearly see how in Figure 5-6 the decision boundary is almost linear and is not able to capture the complex structure of our data anymore. Exactly what we expected: the regularization term makes the model simpler, and therefore less able to capture the fine structures. Is interesting to compare the decision boundary of our network with the result of logistic regression with just one neuron. I will not put the code here for space reasons, but if you compare the two decision boundaries in Figure 5-7 (the one coming from the network with one neuron is linear) you can see that they are almost the same. A regularization term of gives effectively the same results as a network with just one neuron.

Figure 5-7: the decision boundaries for the complex network with and for one with just one neuron. The two boundaries almost overlap completely.



# regularization

## Theory of regularization and tensorflow implementation

regularization also works adding an additional term to the cost function

The effect it has on the learning is effectively similar to the one we described with regularization. Tensorflow does not have, as for , a function ready to be used. We need to code it manually, using the following code

reg = tf.reduce\_sum(tf.abs(W1))+tf.reduce\_sum(tf.abs(W2))+tf.reduce\_sum(tf.abs(W3))+\

tf.reduce\_sum(tf.abs(W4))+tf.reduce\_sum(tf.abs(W5))

the rest of the code we discussed remains the same. We can again compare the weights distribution between the model without regularization term () and with regularization () in Figure 5-8. We have used the Boston dataset for the calculation. We have trained the model with the call

sess, cost\_history = model(learning\_r = 0.01,

training\_epochs = 1000,

features = train\_x,

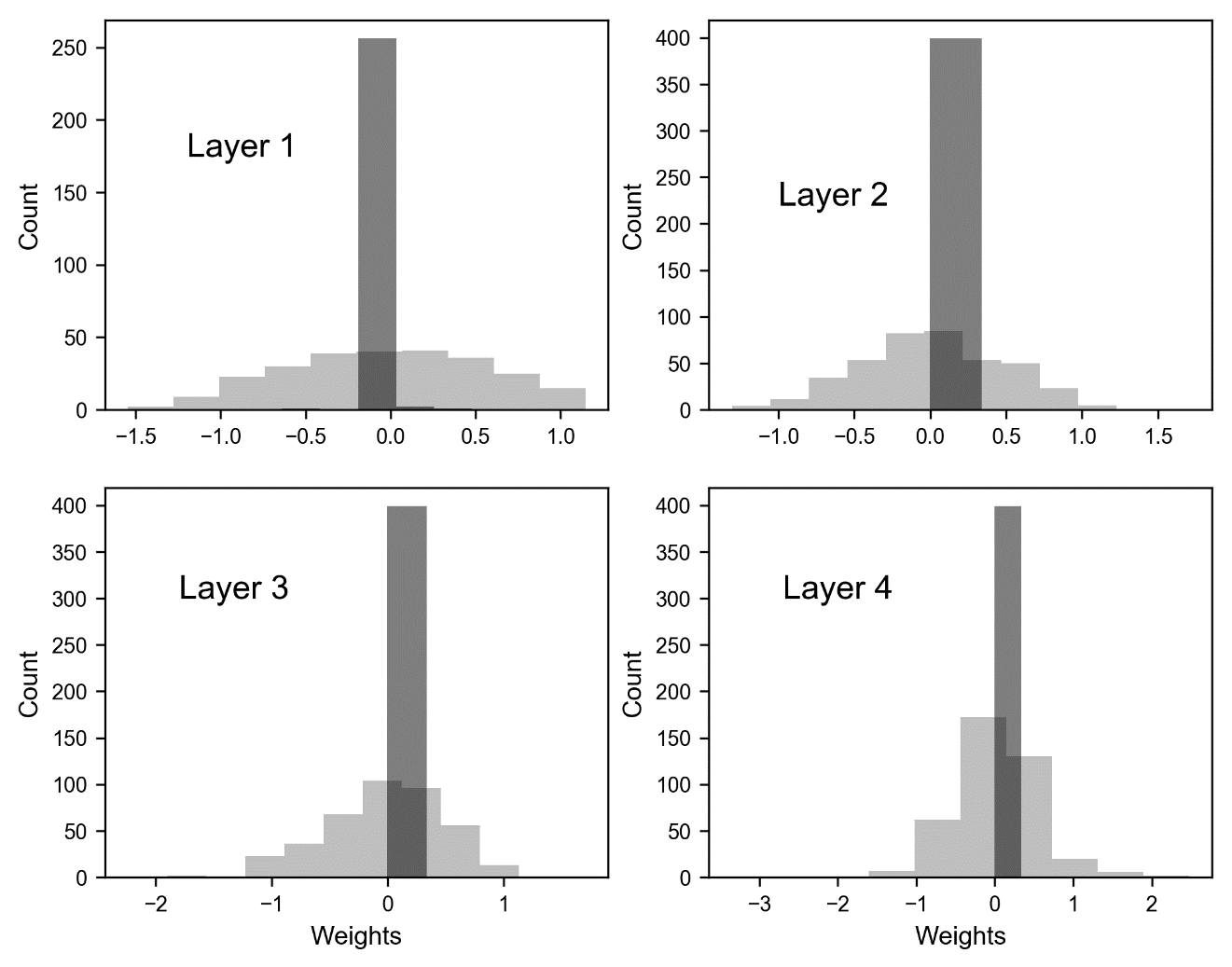
target = train\_y,

logging\_step = 1000,

lambd\_val = 3.0)

once with and once with .

Figure 5-8: weights distributioni comparison distribution between the model without regularization term (, light gray) and with regularization (, dark gray).



As you can see, also regularization has the same effect as . It reduces the effective complexity of the network, reducing many weights to zero.

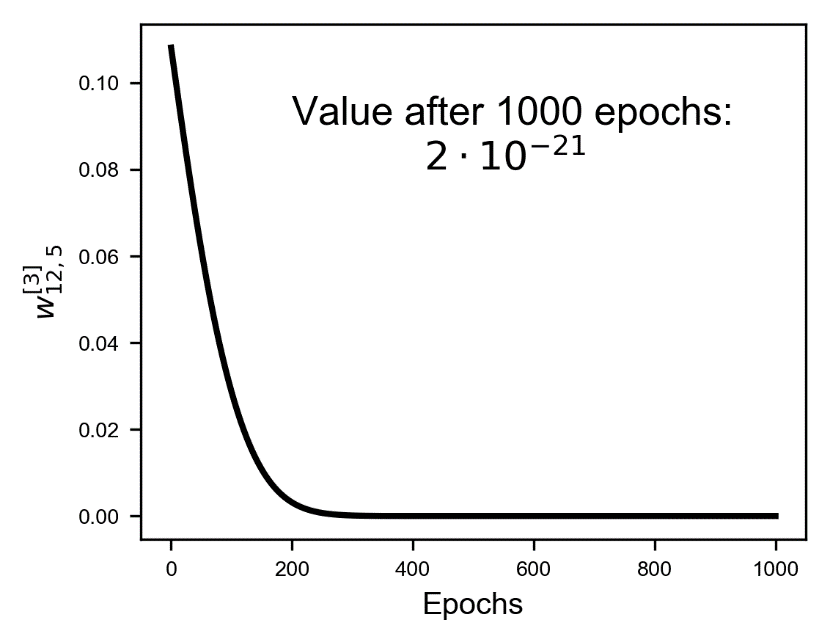
To give you an idea of how effective regularization is in reducing the weights, check the following table where we compare the percentage of weights less than 1e-3 with and without regularization after 1000 epochs.

|  |  |  |
| --- | --- | --- |
| Layer | % of weights less than 1e-3 for | % of weights less than 1e-3 for |
| 1 | 0.0 | 52.7 |
| 2 | 0.25 | 53.8 |
| 3 | 0.75 | 46.3 |
| 4 | 0.25 | 45.3 |
| 5 | 0.0 | 60.0 |

## Are weights really going to zero?

It is very instructive to see how weights are going to zero. For illustrative purpose in Figure 5-9 you can see weight (from layer 3) plotted versus the number of epochs for our artificial dataset with two features, regularization, , , after 1000 epochs. You can see how it quickly decreases to zero. The value after 1000 epochs is , so for all purposes zero.

Figure 5-9: weight plotted versus the epochs for our artificial dataset with two features, regularization, , , trained for 1000 epochs.



In case you are wondering, the weight goes to zero almost exponentially. A way of understanding why this is the case is the following. Let's consider the weight update equation for one weight

Let's now suppose we find ourselves close to the minimum, in a region where the derivative of the cost function is almost zero, so that we can neglect it. In other words, let's suppose

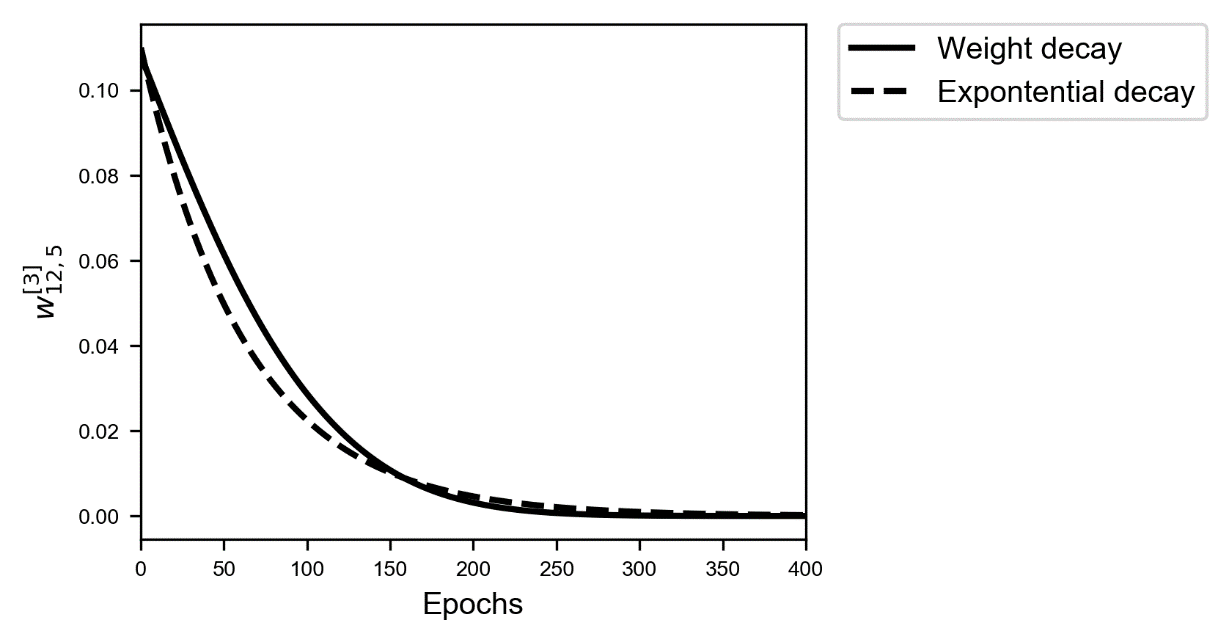
We can rewrite the weight update equation as

now the equation can be read as: the rate of variation of the weight with respect to the iteration number is proportional to the weight itself. For those of you with knowledge of differential equations you may realize that we can draw a parallel to the following equation

This can be read as: the rate of variation of with respect to time is proportional to the function itself. For those of you who know how to solve this equation you may know that a generic solution is

You can now see why the weight decay will have a decay similar to an exponential function by drawing a parallel between the two equations. In Figure 5-10 you can see the weight decay we already discussed together with a pure exponential decay. The two curves are not identical, as expected, since especially at the beginning the gradient of the cost function is surely not zero. But the similarity is remarkable and gives us an idea of how fast the weights can go to zero (read: really fast).

Figure 5-10: weight plotted versus the epochs for our artificial dataset with two features, regularization, , , trained for 1000 epochs (continous line) together with a pure expontential decay (dashed line) for illustrative purposes.



Note that when using regularization, you end up having tensors with a lot of zero elements, called sparse tensors. You can then profit from special routines that are extremely efficient with sparse tensors. Something to keep in mind when you start moving toward more complex models, but a subject too advanced for this book that would require too much space here.

# Dropout

The basic idea of dropout is different: during the training phase you remove nodes from layer randomly with a probability . In each iteration you remove different nodes effectively training at each iteration a different network (when using mini-batch, you train a different network for each batch for example). Usually the probability (called in Python often keep\_prob) is set the same for all the network (but technically speaking can be layer specific). Intuitively let's consider the output tensor Z of a layer . We can define in Python a vector like

d = np.random.rand(Z.shape[0], Z.shape[1]) < keep\_prob

and then we simply multiply the layer output Z by d

Z = np.multiply(Z, d)

Effectively removing all elements that have a probability less then keep\_prob. Very importantly when doing predictions on a dev dataset no dropout should be used!

During training dropout removes nodes randomly each iteration. But when doing predictions on a dev dataset the entire network without dropout needs to be used. In other words, you need to set keep\_prob=1.

Dropout can be layer specific. For example, for layers with many neurons keep\_prob can be small, for layers with a few neurons one can set keep\_prob = 1.0, effectively keeping all neurons in such layers.

The implementation in tensorflow is easy. First you define a placeholder that will contain the value of the keep\_prob parameter

keep\_prob = tf.placeholder(tf.float32, shape=())

and then for each layer you add a regularization operation in this way

hidden1, W1, b1 = create\_layer (X, n1, activation = tf.nn.relu)

hidden1\_drop = tf.nn.dropout(hidden1, keep\_prob)

Then when creating the next layer, instead of using hidden1, you use hidden1\_drop. The entire construction code looks like this

tf.reset\_default\_graph()

n\_dim = 13

n1 = 20

n2 = 20

n3 = 20

n4 = 20

n\_outputs = 1

tf.set\_random\_seed(5)

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

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

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

keep\_prob = tf.placeholder(tf.float32, shape=())

hidden1, W1, b1 = create\_layer (X, n1, activation = tf.nn.relu)

hidden1\_drop = tf.nn.dropout(hidden1, keep\_prob)

hidden2, W2, b2 = create\_layer (hidden1\_drop, n2, activation = tf.nn.relu)

hidden2\_drop = tf.nn.dropout(hidden2, keep\_prob)

hidden3, W3, b3 = create\_layer (hidden2, n3, activation = tf.nn.relu)

hidden3\_drop = tf.nn.dropout(hidden3, keep\_prob)

hidden4, W4, b4 = create\_layer (hidden3, n4, activation = tf.nn.relu)

hidden4\_drop = tf.nn.dropout(hidden4, keep\_prob)

y\_, W5, b5 = create\_layer (hidden4\_drop, n\_outputs, activation = tf.identity)

cost = tf.reduce\_mean(tf.square(y\_-Y))

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

Now let's analyse what happens to the cost function when using dropout. Let's run our model applied to the Boston data set for two values of the keep\_prob variable: 1.0 (without dropout) and 0.5. In Figure 5-11 you can see that when applying dropout, the cost function is very irregular. It oscillates wildly. The two models have been evaluated with the calls

sess, cost\_history05 = model(learning\_r = 0.01,

training\_epochs = 5000,

features = train\_x,

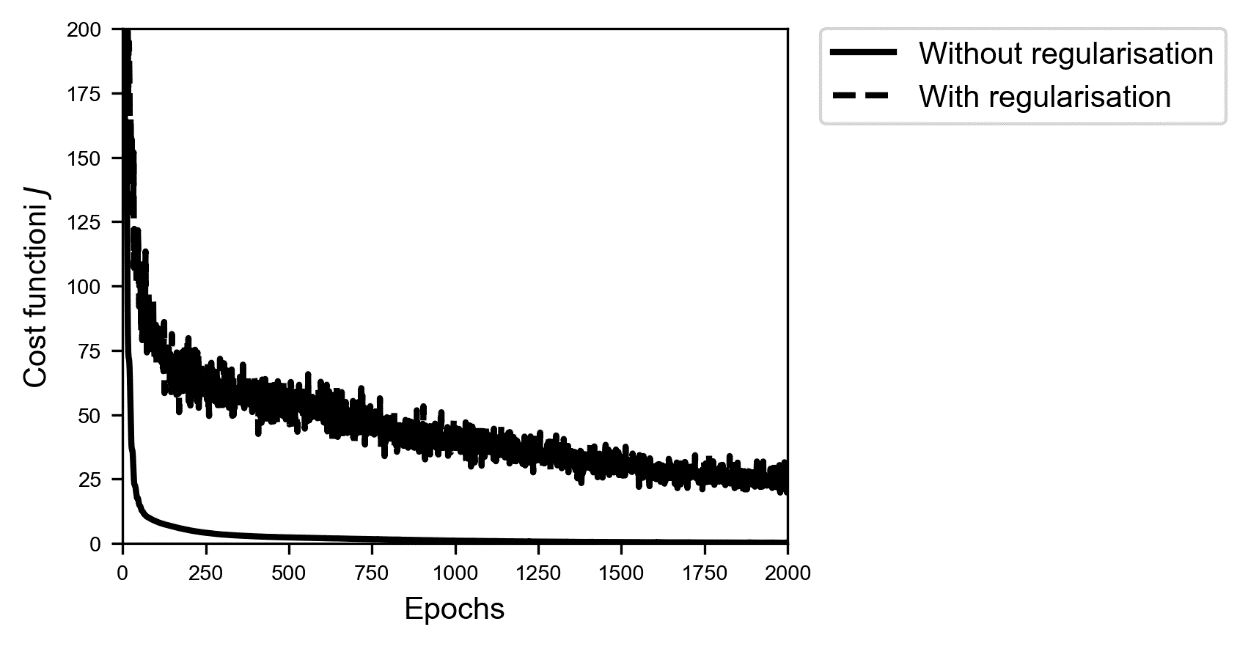
target = train\_y,

logging\_step = 1000,

keep\_prob\_val = 1.0)

for keep\_prob\_val = 1.0 and for 0.5.

Figure 5-11: the cost function for the training dataset for our model for two values of the keep\_prob variable: 1.0 (no dropout) and 0.5. The other parameters are: . The models have been trained for 5000 epochs. No minibatch has been used. The oscillating line is the one evaluated with regularization.

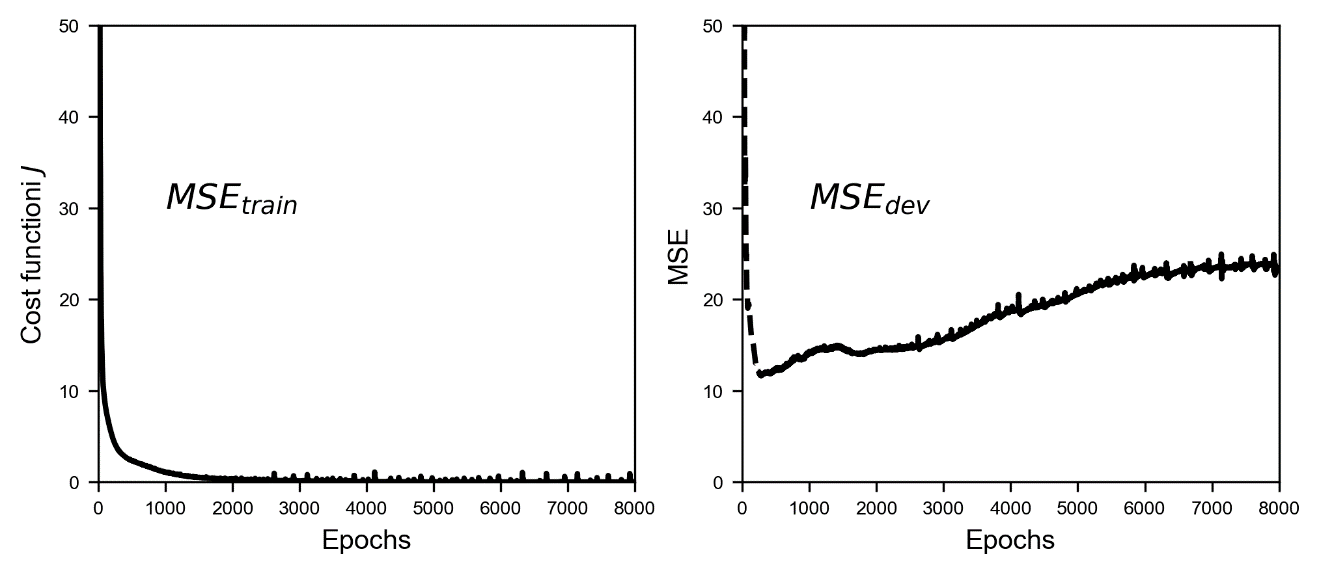


In Figure 5-12 you can see the evolution of the MSE for the training and the dev dataset in case of dropout (keep\_prob=0.4). In Figure 5-13 you can see the same plot but without dropout. The difference is quite striking. Very interesting is the fact that without dropout grows with epochs, while using dropout it is rather stable.

Figure 5-12: MSE for the training and the dev dataset in case of dropout (keep\_prob=0.4)



Figure 5-13: MSE for the training and the dev dataset in case of no dropout (keep\_prob=1.0)



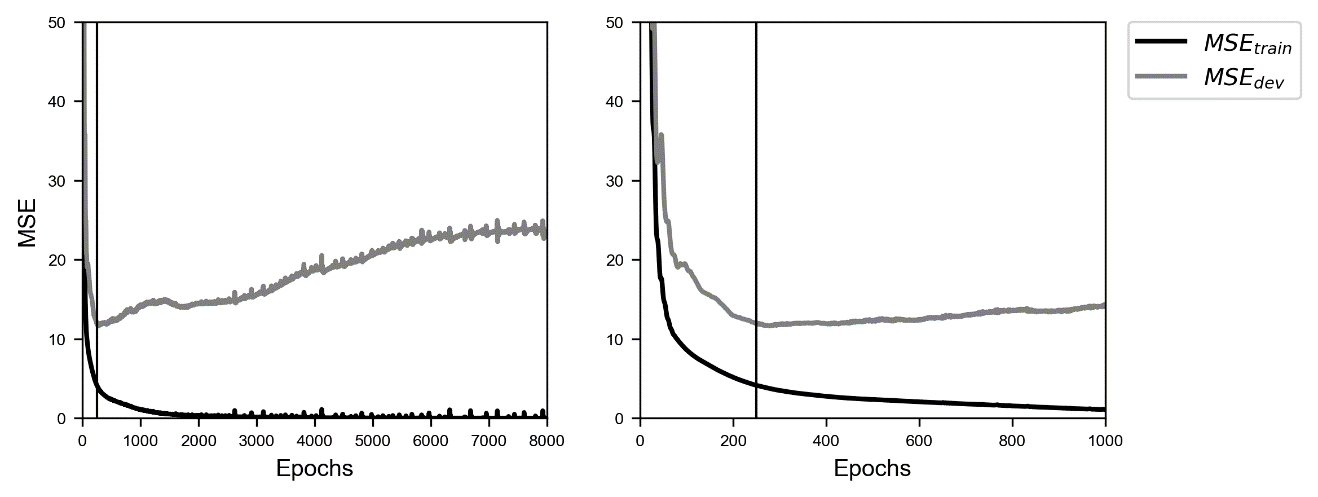
In Figure 5-13 he grows after dropping at the beginning. The model is in clear extreme overfitting regime (), and it generalize worst and worst when applied to the new data. In Figure 5-12 you can see how the and are of the same order of magnitude and the does not continue to grow, so we have a model that is a lot better at generalizing than the one whose results are shown in Figure 5-13.

When applying dropout your metric (in this case the MSE) will oscillate, so don't be surprised when trying to find the best hyperparameters if you see your optimizing metric oscillating.

# Early stopping

Now there is another technique that is sometime used to fight overfitting. Strictly speaking this method does nothing to avoid overfitting, it simply stops the learning before the overfitting problem becomes too bad. Consider the example of last section. In Figure 5-14 you can see the and plotted on the same plot.

Figure 5-14: MSE for the training and the dev dataset in case of no dropout (keep\_prob=1.0). Early stopping consists in stopping the learning phase at the iteration when the is minimum (indicated with a vertical line in the plot). Right you can see a zoom of the left plot for the first 1000 epochs.



Early stopping simply consists in stopping the training at the point when the has its minimum (see Figure 5-14, the minimum is indicated by a vertical line in the Figure). Note that this is not an ideal way of solving the overfitting problem. Your model will still most probably generalize very badly to new data. I usually prefer to use other techniques. Additionally, this is also time consuming and a manual process that is very error prone. You can get a good overview of the different application contexts checking the Wikipedia page for early stopping: https://goo.gl/xnKo2s.

# Additional methods

All the methods we discussed so far consists, in some form or another, in making the model less complex: you keep the data as it is and modify your model. But we can try to do the opposite: leave the model as it is and work on the data. Here are two common strategies that work for fighting overfitting (but not really easily applicable)

* Get more data. This is the simplest way of fighting overfitting. Unfortunately, very often in real life this is not possible. Keep in mind that this is a complicated matter that we will discuss at length in the next chapter. If you are classifying cats pictures taken with a smartphone you may think of getting more data from the web. Although this may seem a perfectly good idea, you may discover that the images have different quality, that possibly not all the images are really cats (what about cat toys?), you may only find images of white young cats, and so on. Basically, your additional observations may probably come from a very different distribution than your original data and that will be a problem as we will see. So, when getting additional data consider the problem well before proceeding.
* Augment your data. For example, if you are working with images you can generate additional images by rotating, stretching, shifting, etc. your images. That is a very common technique that may really help.

The problem of making the model generalize better on new data is one of machine learning biggest goal. It is a complicated problem that requires experience and tests. Lots of tests. Much research is going on that tries to solve those kinds of problems when working on very complex problems. We will discuss additional techniques in the next Chapter.

1. In a [statistical-classification](https://en.wikipedia.org/wiki/Statistical_classification) problem with two classes, a decision boundary or decision surface is a [surface](https://en.wikipedia.org/wiki/Hypersurface) that partitions the underlying [space](https://en.wikipedia.org/wiki/Vector_space) into two sets, one for each class (Source wikipedia: https://goo.gl/E5nELL). [↑](#footnote-ref-1)