Regularization

In this chapter, we will look at a very important technique often used when training deep networks: regularization. We will look at techniques such as the:  **and**  **methods**, **dropout**, and **early stopping**. We will see how these methods help avoid the problem of overfitting and achieve much better results from your models, when applied correctly. We will look at the mathematics behind the methods and at how to implement it in Python and Keras correctly.

# 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 last chapter 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 [1].

This contains information collected by the US Census Bureau concerning housing around Boston. Each record in the database describes a Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. The attributes are defined as follows:

* CRIM: Per capita crime rate by town
* ZN: Proportion of residential land zoned for lots over 25000 square feet
* INDUS: Proportion of non-retail business acres per town
* CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
* NOX: Nitric oxides concentration (parts per 10 million)
* RM: Average number of rooms per dwelling
* AGE: Proportion of owner-occupied units built prior to 1940
* DIS: Weighted distances to five Boston employment centers
* RAD: Index of accessibility to radial highways
* TAX: Full-value property-tax rate per $10000
* PTRATIO: Pupil-teacher ratio by town
* : Proportion of blacks by town
* LSTAT: % lower status of the population
* MEDV: Median value of owner-occupied homes in $1000s

Let us review how to get the data.

# sklearn libraries

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 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 us normalize it and then create a training and a dev dataset

features\_norm = normalize(features)

np.random.seed(42)

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

train\_x = features\_norm[rnd]

train\_y = target[rnd]

dev\_x = features\_norm[~rnd]

dev\_y = target[~rnd]

print(train\_x.shape)

print(train\_y.shape)

print(dev\_x.shape)

print(dev\_y.shape)

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).

Then let us 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, train the model and validate it against the dev dataset

def create\_and\_train\_model\_nlayers(data\_train\_norm, labels\_train, data\_dev\_norm, labels\_dev, num\_neurons, num\_layers):

# build model

# input layer

inputs = keras.Input(shape = data\_train\_norm.shape[1])

# he initialization

initializer = tf.keras.initializers.HeNormal()

# first hidden layer

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer)(inputs)

# customized number of layers and neurons per layer

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

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer)(dense)

# output layer

outputs = layers.Dense(1)(dense)

model = keras.Model(inputs = inputs, outputs = outputs,

name = 'model')

# set optimizer and loss

opt = keras.optimizers.Adam(learning\_rate = 0.001)

model.compile(loss = 'mse', optimizer = opt,

metrics = ['mse'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = 10000, verbose = 0,

batch\_size = data\_train\_norm.shape[0],

validation\_data = (data\_dev\_norm, labels\_dev))

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

return hist, model

The above function builds and trains a feed-forward neural network model for linear regression and evaluates it on the training and dev sets. You have already seen feed-forward neural networks implementation in the last chapter and linear regression in so you should now understand what it does.

We use here He initialization since we will use ReLU activation functions. In our output layer we have one neuron with the identity activation function for regression (remember that, in Keras, when you do not specify any activation function, the default one will be the identity function). Additionally, we use the Adam optimizer[[1]](#footnote-1).

Now let us run the model with this code

hist, model = create\_and\_train\_model\_nlayers(train\_x, train\_y, dev\_x, dev\_y, 20, 4)

As you may notice, to make things simpler, we just avoided writing a function with many input parameters and simply hard coded all the values in the body function (as for the learning rate, for example), since in this case we do not need to tune much the parameters. Moreover, we are not using mini batches here, since we have only few hundred observations.

We are calculating the MSE for both training and dev datasets, as the typical cost function for linear regression problems. 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 4-1.

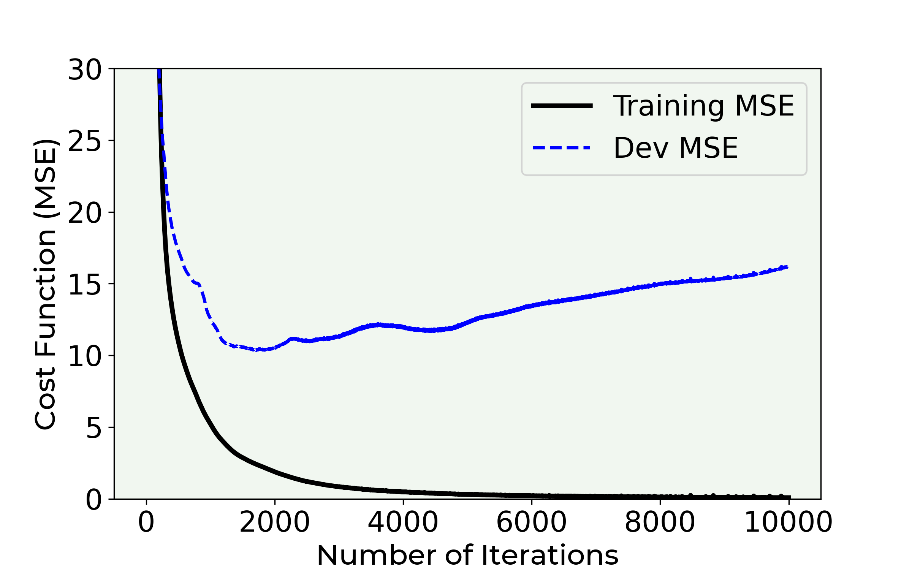


Figure 4-1. The MSE for the training (continuous line) and the dev dataset (dashed line) for the neural network with 4 layers, each having 20 neurons.

You will notice how the training error goes down to zero while the dev error reaches a value of about 10 and then it starts increasing up to a value of about 15, 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 16-2 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 4-2 on the left plot.

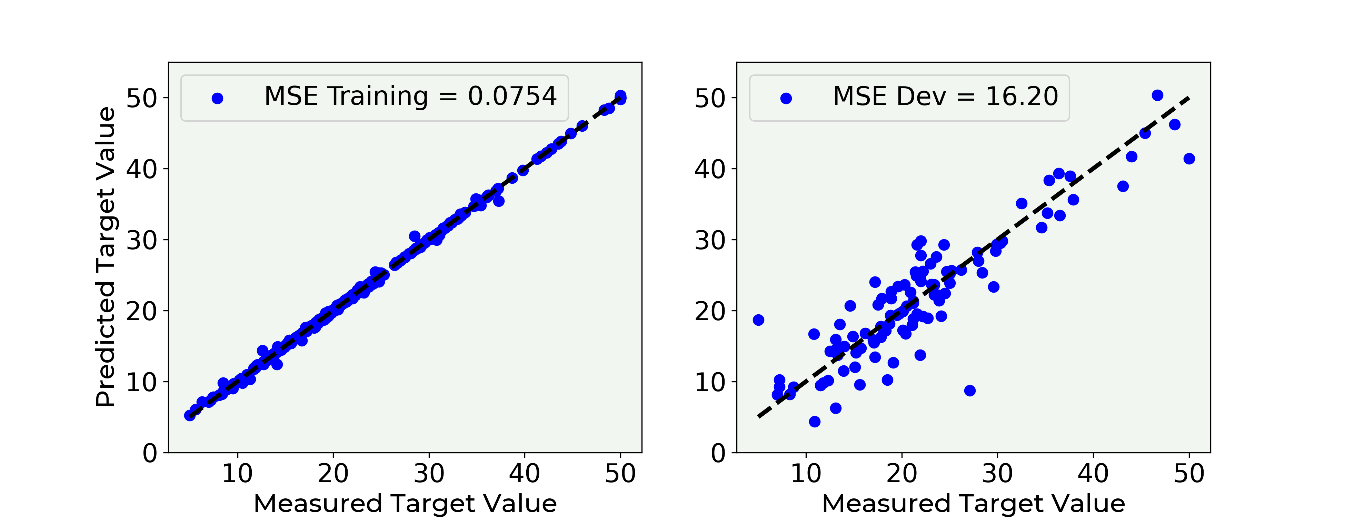


Figure 4-2. Predicted value versus the real value for the target variable (the house price). You will notice how in the left plot, for the training data, the prediction is almost perfect, while on the plot on the right, for the dev dataset, the predictions are more spread.

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 is the focus of this chapter.

# What regularization is

Before going into the different methods, we must 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 [2]. Lately the term has gained a much more broader meaning. For example, Goodfellow [3] 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 [4] 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 in 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 it 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 us 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

Let us spend a few moments briefly discussing 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? It is very difficult to give a definition of network complexity, so much that nobody does it, actually. You find several research papers on the problem of model complexity (note that this is not exactly network complexity), with roots in information theory. You will see in this chapter how for example the number of weights that is different from 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 regularization is, 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 us now start with the most instructive norm: the .

## regularization

One of the most common regularization methods,  regularization consists of adding a term to the cost function that has the goal of effectively reducing the capacity of the network to adapt to complex datasets. Let us first have a look at the mathematics behind the method.

### Theory of regularization

When doing plain regression, 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 us define a new cost function

This additional term

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

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

Now let us try to get an intuitive understanding of what is the effect of this term on the GD (Gradient Descent) algorithm. Let us 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 us try to see what is really happening to the weights by applying the method to the Boston housing dataset.

### Keras implementation

The implementation in Keras is extremely easy. The library performs all the computations for us, and we have just to decide which regularization we want to use, set the parameter and apply it to each layer. The model construction remains the same.

**Note** In Keras the regularization is applied at the layer level, and not globally on the cost function. You will notice how it is added at each layer and not when defining the cost function. But the explanation above remains valid, and it works in the same way as we discussed. The reason for this is that it can be helpful to add regularization only on certain layers, for example the largest, and not on layers with just a few neurons.

We can do it with the function

def create\_and\_train\_reg\_model\_L2(data\_train\_norm, labels\_train, data\_dev\_norm, labels\_dev, num\_neurons, num\_layers, n\_epochs, lambda\_):

# build model

# input layer

inputs = keras.Input(shape = data\_train\_norm.shape[1])

# he initialization

initializer = tf.keras.initializers.HeNormal()

# regularization

reg = tf.keras.regularizers.l2(l2 = lambda\_)

# first hidden layer

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(inputs)

# customized number of layers and neurons per layer

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

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(dense)

# output layer

outputs = layers.Dense(1)(dense)

model = keras.Model(inputs = inputs, outputs = outputs,

name = 'model')

# set optimizer and loss

opt = keras.optimizers.Adam(learning\_rate = 0.001)

model.compile(loss = 'mse', optimizer = opt,

metrics = ['mse'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = n\_epochs, verbose = 0,

batch\_size = data\_train\_norm.shape[0],

validation\_data = (data\_dev\_norm, labels\_dev))

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

# print performances

print('Cost function at epoch 0')

print('Training MSE = ', hist['loss'].values[0])

print('Dev MSE = ', hist['val\_loss'].values[0])

print('Cost function at epoch ' + str(n\_epochs))

print('Training MSE = ', hist['loss'].values[-1])

print('Dev MSE = ', hist['val\_loss'].values[-1])

return hist, model

The main differences with respect to the previous function (the one that we used to build a network without regularization) are highlighted in bold.

With the line reg = tf.keras.regularizers.l2(l2 = lambda\_) we defined the regularizer, setting the value for . Then we apply the regularizer to each layer, assigning it to the kernel\_regularizer, which applies the penalty on the layer’s kernel. The layers may expose also the keywork arguments bias\_regularizer and activity\_regularizer, which apply a penalty on the layer's bias and output, respectively, but are less often used. Here we will employ only the kernel\_regularizer argument.

Remember that in Python lambda is a reserved word, so we cannot use it, this is the reason why we use lambda\_.

Now let us train and evaluate our network to see what happens. This time we will 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 us run the model for , without regularization, and for .

We can run our model with the code

hist, model = create\_and\_train\_reg\_model\_L2(train\_x, train\_y, dev\_x, dev\_y, 20, 4, 0.0)

and that gives us

Cost function at epoch 0

Training MSE = 653.5233764648438

Dev MSE = 623.965087890625

Cost function at epoch 5000

Training MSE = 0.2870051860809326

Dev MSE = 25.645526885986328

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

hist, model = create\_and\_train\_reg\_model\_L2(train\_x, train\_y, dev\_x, dev\_y, 20, 4, 10.0)

and that gives the results

Cost function at epoch 0

Training MSE = 2141.39599609375

Dev MSE = 2100.5986328125

Cost function at epoch 5000

Training MSE = 58.91643524169922

Dev MSE = 56.80580139160156

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 4-3 the weights distribution for the 4 hidden 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.

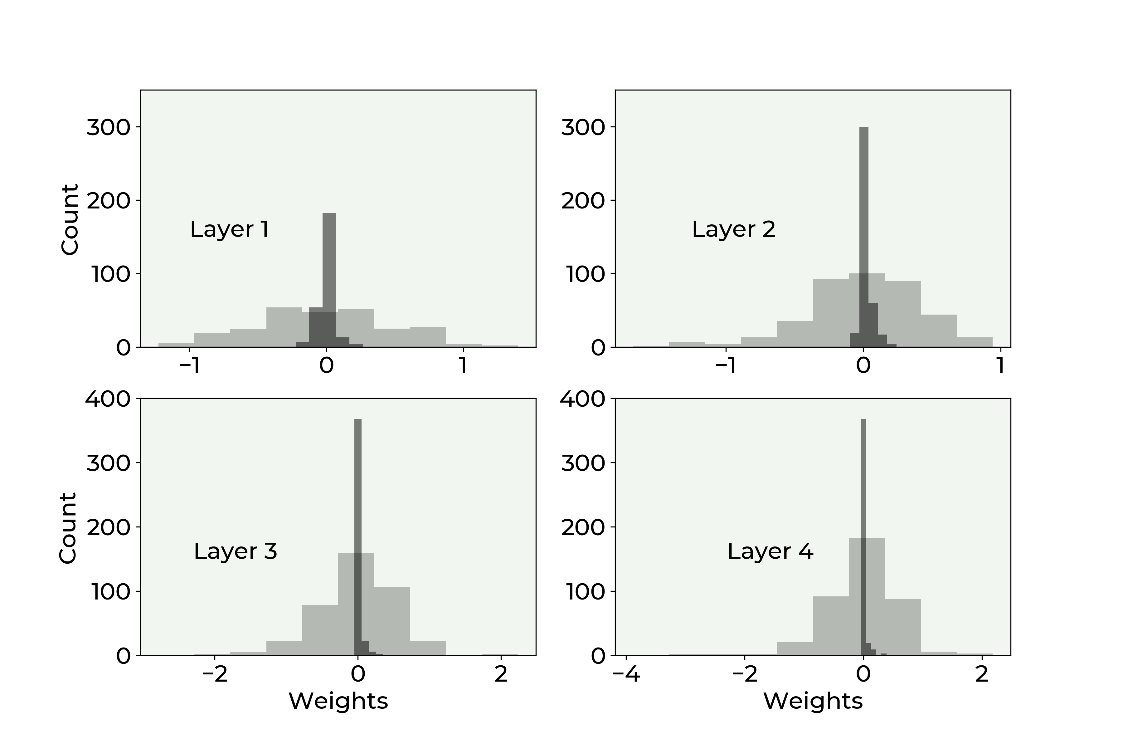


Figure 4-3. Weights distribution for each layer. The light gray histogram is for the weights without regularization, and the darker (and much more concentrated around zero) is for the weights with regularization.

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 makes the weight decay effect of regularization very evident. We have now the chance to make another brief digression on network complexity. We said that this method reduces the network complexity. We saw previously that you can consider the number of learnable parameters an indication of the complexity of a network, but you have also been warned that this can be very misleading. Now let us see why. You will remember, 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 one. 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 it is interesting to note that we have roughly 6% of the weights that after 1000 epochs are less than , so effectively close to zero. This is the reason why talking about complexity in terms of number of learnable parameters is risky. 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.

**Note** Defining the complexity of a network only in terms of number of weights 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. It 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 why 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 with and without regularization after 1000 epochs in each hidden layer.

Table 4-1. Percentage of weights less than with and without regularization after 1000 epochs.

|  |  |  |
| --- | --- | --- |
| **Layer** | **% of Weight Less Than for** | **% of Weight Less Than for** |
| 1 | 0.77 | 1.54 |
| 2 | 0.0 | 28.25 |
| 3 | 1.0 | 40.0 |
| 4 | 0.25 | 45.75 |

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

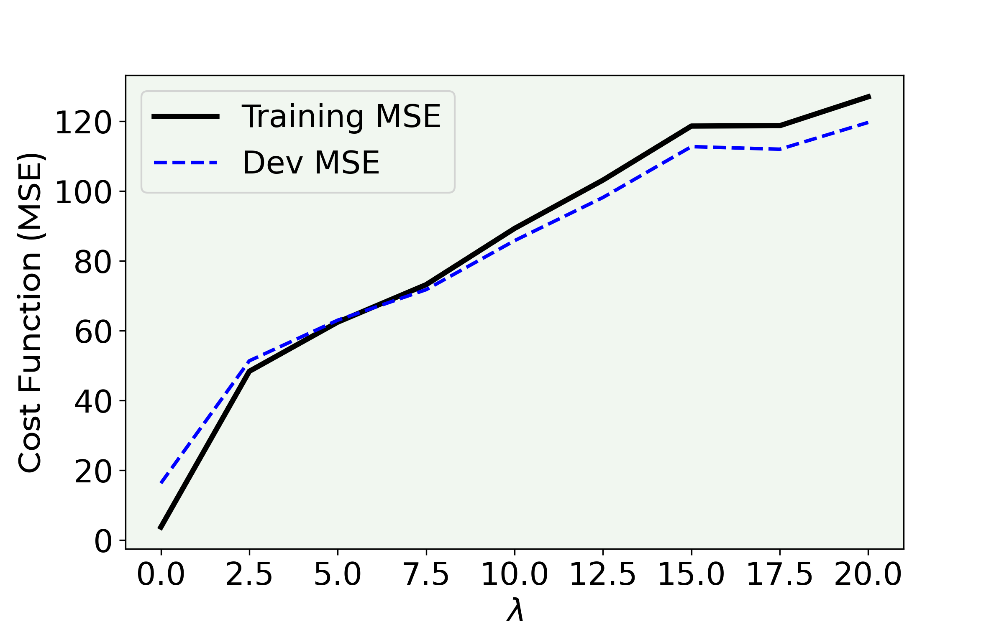


Figure 4-4. Behavior of the MSE for the training (continuous line) dataset and for the dev (dashed) dataset for our network varying .

As you can see with small values of (effectively without regularization) we are in an overfitting regime (), with the and the increasing. 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 not anymore in an overfitting region since . Remember that 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.

**Note** 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.

Let us discuss what are the effects of regularization in an even more visual way. Let us consider now a dataset generated with the following code

nobs = 30 # number of observations

np.random.seed(42) # making results reproducible

# first set of observations

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)])

# second set of observations

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)])

# concatenating observations

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

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

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

# creating the labels

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

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

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

# defining training points and labels

train\_x = c

train\_y = yyL

Our dataset has two features: and . We have generated two group of points xx1, yy1 and xx2, yy2 from a normal distribution. To the first we have assigned the label 0 (contained in the array yy1\_) and to the second the label 1 (in the array yy2\_). Now let us use a network like the one 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

def create\_and\_train\_regularized\_model(data\_train\_norm, labels\_train, num\_neurons, num\_layers, n\_epochs, lambda\_):

# build model

# input layer

inputs = keras.Input(shape = data\_train\_norm.shape[1])

# he initialization

initializer = tf.keras.initializers.HeNormal()

# regularization

reg = tf.keras.regularizers.l2(l2 = lambda\_)

# first hidden layer

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(inputs)

# customized number of layers and neurons per layer

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

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(dense)

# output layer

outputs = layers.Dense(1, activation = 'sigmoid')(dense)

model = keras.Model(inputs = inputs, outputs = outputs,

name = 'model')

# set optimizer and loss

opt = keras.optimizers.Adam(learning\_rate = 0.005)

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

optimizer = opt, metrics = ['accuracy'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = n\_epochs, verbose = 0,

batch\_size = data\_train\_norm.shape[0])

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

return hist, model

As you can see the code is almost the same, except the different cost function and the activation function of the output layer. Let us plot the decision boundary[[2]](#footnote-2) for this problem. That means we will run our network on our dataset with the code

hist, model = create\_and\_train\_regularized\_model(train\_x, train\_y, 20, 4, 100, 0.0)

In Figure 4-5, you can see our dataset, 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.

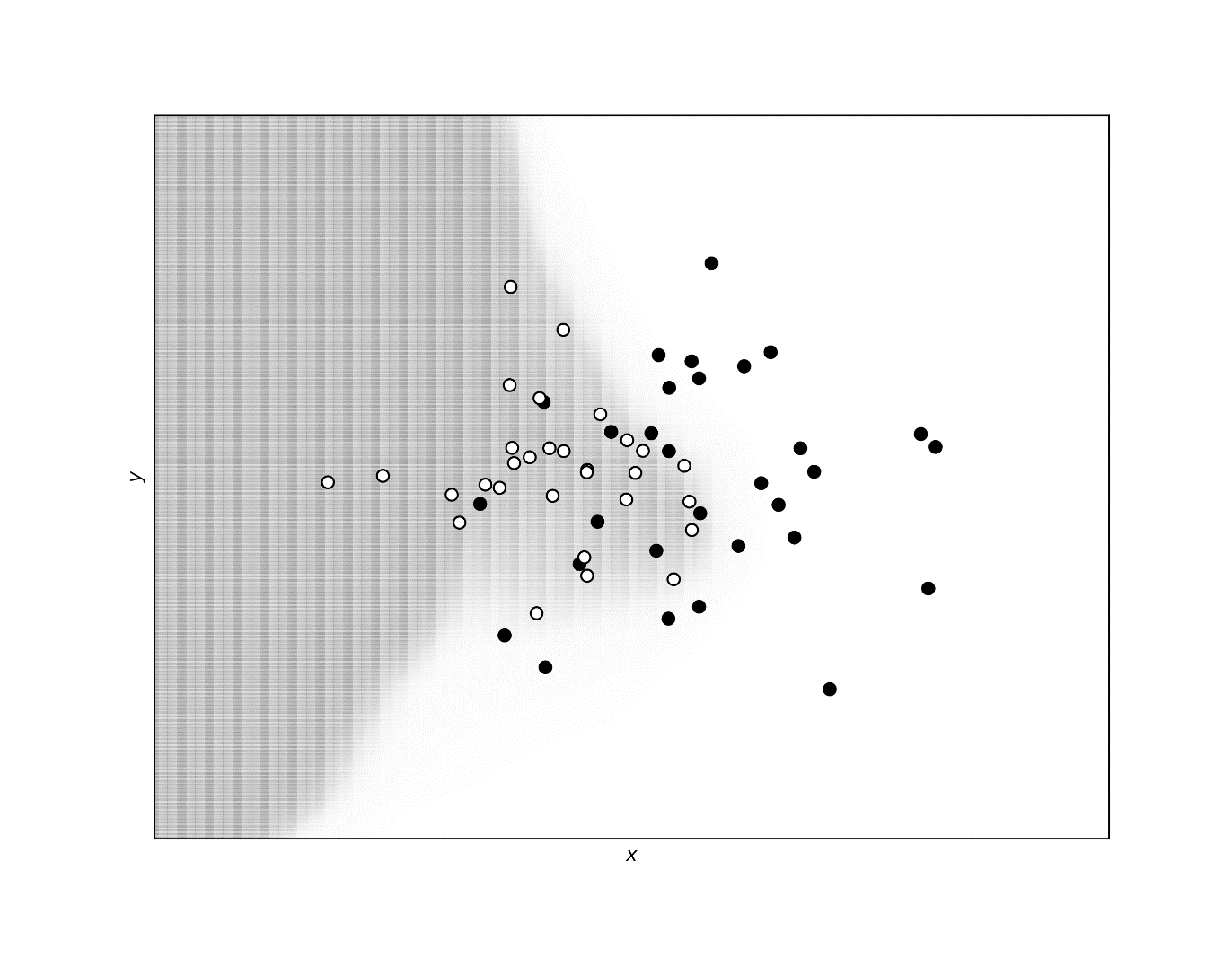


Figure 4-5. Decision boundary without regularization. White points are of one class and black of the second. The gray area is the zone that the network classifies being of one class and the white to the other. You can see that the network is able to capture the complex structure of our data.

Now let us 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 .

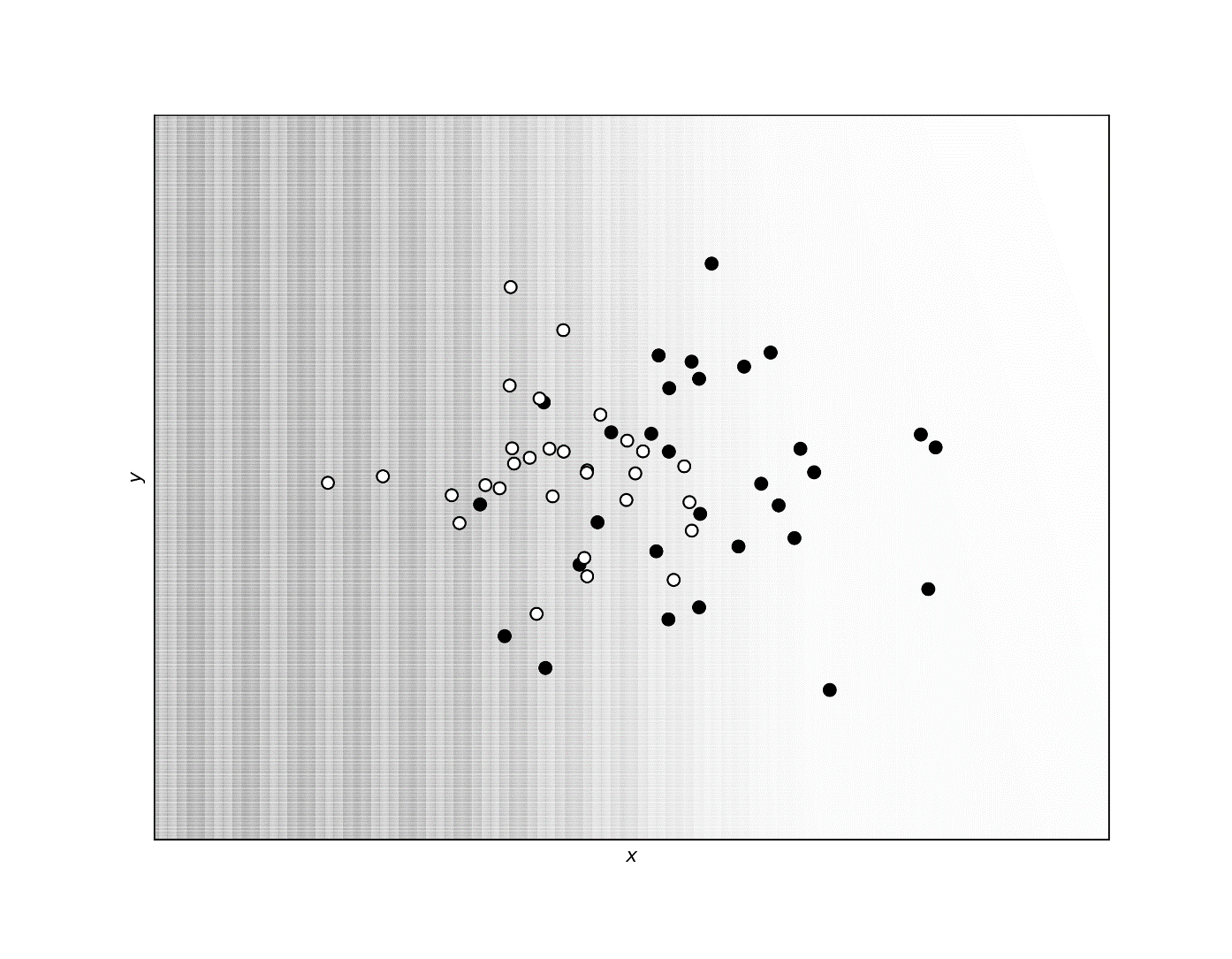


Figure 4-6. The decision boundary as predicted by the network with regularization and with a regularization parameter .

You can clearly see how in Figure 4-6 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. It is interesting to compare the decision boundary of our network with the result of logistic regression with just one neuron. We will not repeat the code here for space reasons (you can find the complete code version in the online version of the book), but if you compare the two decision boundaries in Figure 4-7 (the one coming from the network with one neuron is linear) you can see that they are almost the same. The difference is that the regularized version presents a smoother decision boundary. To conclude, a regularization term of effectively gives the same results as a network with just one neuron.



Figure 4-7. The decision boundaries for the complex network with and for one with just one neuron. The two boundaries almost overlap completely.

## regularization

Now we will look at a regularization technique that is very similar to regularization. It is based on the same principle, adding a term to the cost function. This time, the mathematical form of the added term is different, but the method works very similarly to what we saw in the previous sections. Let us again first have a look at the mathematics behind the algorithm.

### Theory of regularization and Keras 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. Keras has, as for , a function ready to be used. The code is the same as before, with the only difference in the regularized definition

def create\_and\_train\_reg\_model\_L1(data\_train\_norm, labels\_train, data\_dev\_norm, labels\_dev, num\_neurons, num\_layers, n\_epochs, lambda\_):

# build model

# input layer

inputs = keras.Input(shape = data\_train\_norm.shape[1])

# he initialization

initializer = tf.keras.initializers.HeNormal()

# regularization

reg = tf.keras.regularizers.l1(l1 = lambda\_)

# first hidden layer

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(inputs)

# customized number of layers and neurons per layer

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

dense = layers.Dense(num\_neurons, activation = 'relu',

kernel\_initializer = initializer,

kernel\_regularizer = reg)(dense)

# output layer

outputs = layers.Dense(1)(dense)

model = keras.Model(inputs = inputs,

outputs = outputs,

name = 'model')

# set optimizer and loss

opt = keras.optimizers.Adam(learning\_rate = 0.001)

model.compile(loss = 'mse',

optimizer = opt,

metrics = ['mse'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = n\_epochs, verbose = 0,

batch\_size = data\_train\_norm.shape[0],

validation\_data = (data\_dev\_norm, labels\_dev))

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

# print performances

print('Cost function at epoch 0')

print('Training MSE = ', hist['loss'].values[0])

print('Dev MSE = ', hist['val\_loss'].values[0])

print('Cost function at epoch ' + str(n\_epochs))

print('Training MSE = ', hist['loss'].values[-1])

print('Dev MSE = ', hist['val\_loss'].values[-1])

return hist, model

We can again compare the weights distribution between the model without regularization term () and with regularization () in Figure 4-8. We have used the Boston dataset for the calculation. We have trained the model with the call

hist\_reg, model\_reg = create\_and\_train\_reg\_model\_L1(train\_x, train\_y, dev\_x, dev\_y, 20, 4, 1000, 3.0)

once with and once with .

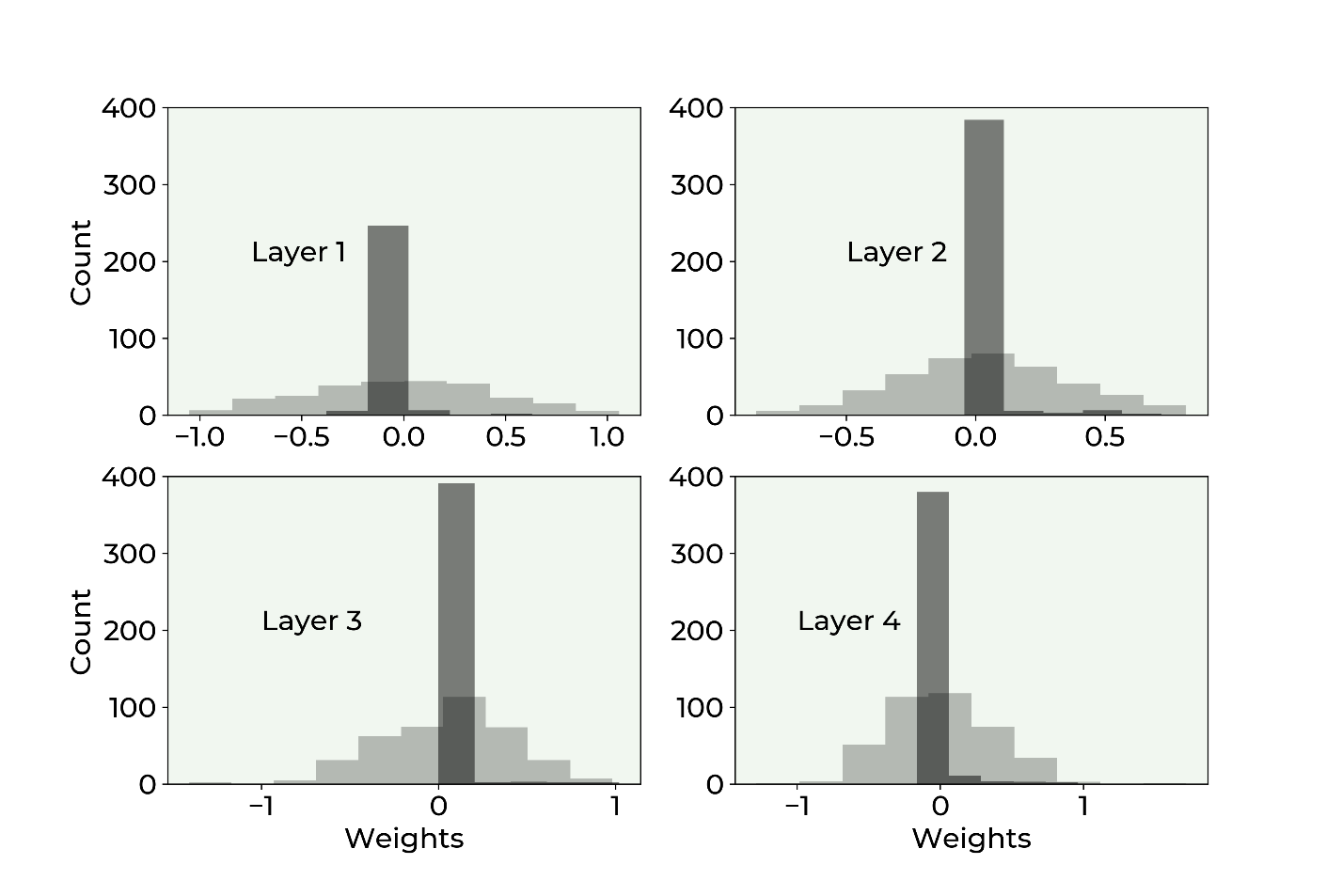


Figure 4-8. Weights distribution comparison distribution between the model without regularization term (, light gray) and with regularization (, dark gray).

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 with and without regularization after 1000 epochs.

Table 4-2. Percentage of weights less than with and without regularization after 1000 epochs.

|  |  |  |
| --- | --- | --- |
| **Layer** | **% of Weight Less Than for** | **% of Weight Less Than for** |
| 1 | 0.0 | 90.77 |
| 2 | 0.5 | 94.50 |
| 3 | 0.0 | 96.75 |
| 4 | 0.0 | 94.50 |

## Are weights really going to zero?

It is very instructive to see how weights are going to zero. For illustrative purpose in Figure 4-9 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.

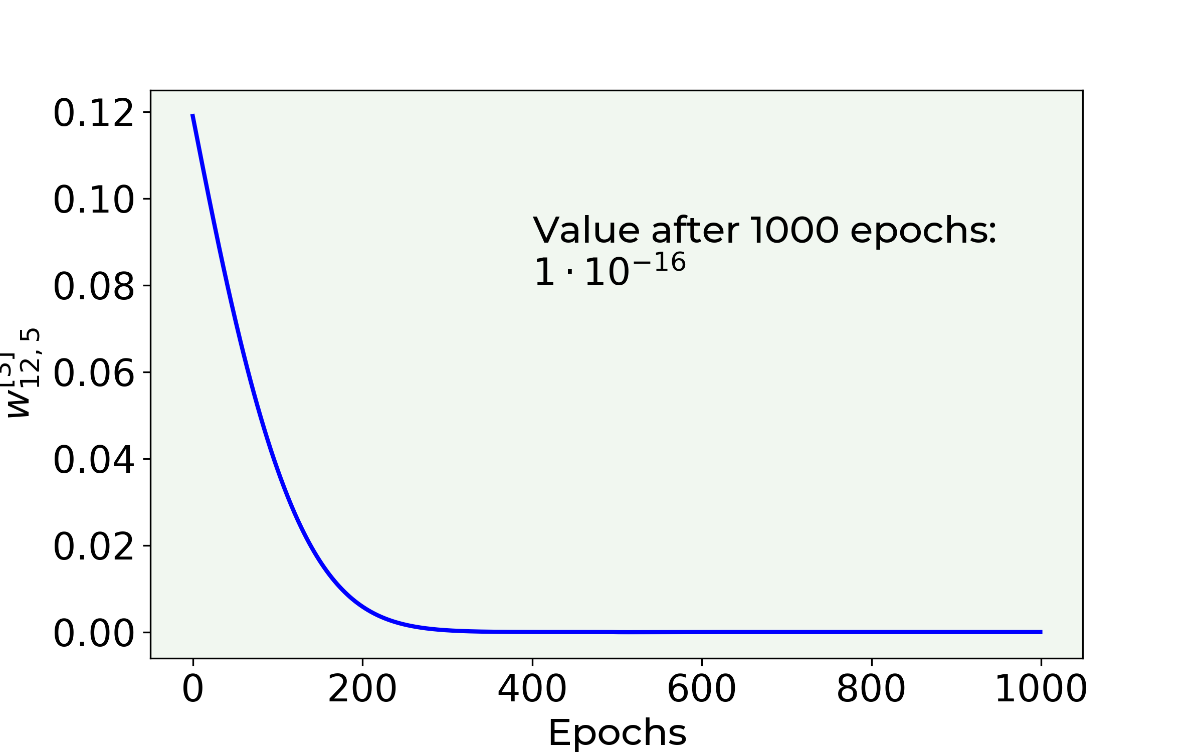


Figure 4-9. Weight plotted versus the epochs for our artificial dataset with two features, regularization, , , trained for 1000 epochs.

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 us consider the weight update equation for one weight

Let us 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 us 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 4-10 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).

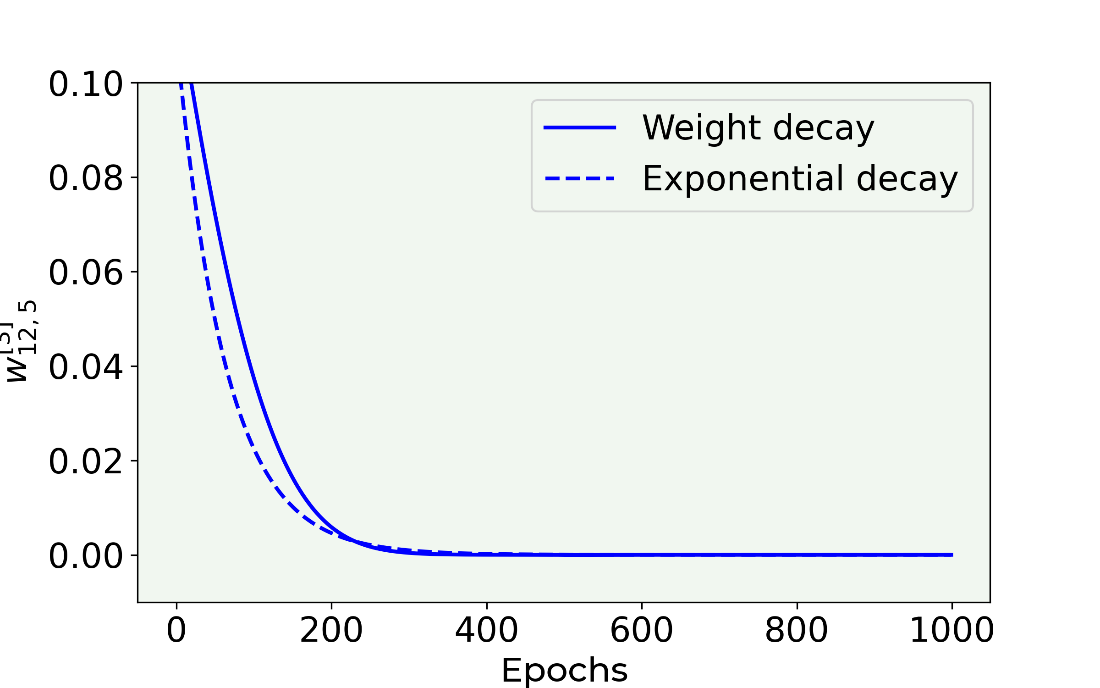


Figure 4-10. Weight plotted versus the epochs for our artificial dataset with two features, regularization, , , trained for 1000 epochs (continous line) together with a pure expontential decay (dashed line) for illustrative purposes.

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-batches, you train a different network for each batch for example).

In Keras, you simply add how many dropout layers as you want after the layer you need to drop, with the following function: keras.layers.Dropout(rate). In the above function you must put as input the layer you want to drop, and you must set the rate parameter. This parameter can assume float values in the following range: , since it represents the fraction of the input units to drop. Therefore, it is not possible to drop all the units (setting a rate equal to 1). Usually, the rate parameter is set the same for all the network (but technically speaking can be layer specific).

Very importantly, when doing predictions on a dev dataset no dropout should be used! Keras will automatically apply dropout during the training phase of the model, without dropping any additional unit during the model’s evaluation on a different set.

**Note** 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. Keras will automatically consider this case for you.

Dropout can be layer specific. For example, for layers with many neurons rate can be small, for layers with a few neurons one can set rate = 0.0, effectively keeping all neurons in such layers.

The implementation in Keras is easy.

def create\_and\_train\_reg\_model\_dropout(data\_train\_norm, labels\_train, data\_dev\_norm, labels\_dev, num\_neurons, num\_layers, n\_epochs, rate):

# build model

# input layer

inputs = keras.Input(shape = data\_train\_norm.shape[1])

# he initialization

initializer = tf.keras.initializers.HeNormal()

# first hidden layer

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer)(inputs)

# first dropout layer

dense = keras.layers.Dropout(rate)(dense)

# customized number of layers and neurons per layer

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

dense = layers.Dense(

num\_neurons, activation = 'relu',

kernel\_initializer = initializer)(dense)

# customized number of dropout layers

dense = keras.layers.Dropout(rate)(dense)

# output layer

outputs = layers.Dense(1)(dense)

model = keras.Model(inputs = inputs,

outputs = outputs,

name = 'model')

# set optimizer and loss

opt = keras.optimizers.Adam(learning\_rate = 0.001)

model.compile(loss = 'mse', optimizer = opt,

metrics = ['mse'])

# train model

history = model.fit(

data\_train\_norm, labels\_train,

epochs = n\_epochs, verbose = 0,

batch\_size = data\_train\_norm.shape[0],

validation\_data = (data\_dev\_norm, labels\_dev))

# save performances

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

# print performances

print('Cost function at epoch 0')

print('Training MSE = ', hist['loss'].values[0])

print('Dev MSE = ', hist['val\_loss'].values[0])

print('Cost function at epoch ' + str(n\_epochs))

print('Training MSE = ', hist['loss'].values[-1])

print('Dev MSE = ', hist['val\_loss'].values[-1])

return hist, model

As you can see, you must put a dropout layer (highlighted in bold) after the layer you want to modify, setting the rate parameter.

Now let us analyze what happens to the cost function when using dropout. Let us run our model applied to the Boston dataset for two values of the rate variable: 0.0 (without dropout) and 0.5. In Figure 4-11 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

hist\_reg, model\_reg = create\_and\_train\_reg\_model\_dropout(train\_x, train\_y, dev\_x, dev\_y, 20, 4, 8000, 0.50)

for rate = 0.0 and for 0.50.

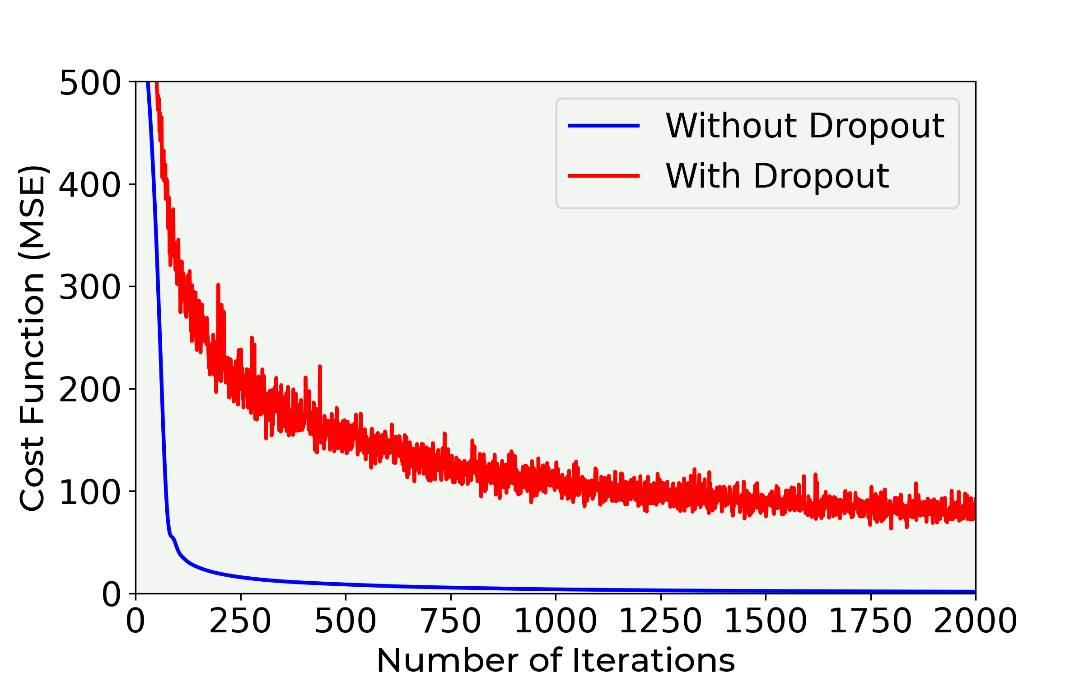


Figure 4-11. The cost function for the training dataset for our model for two values of the rate variable: 0.0 (no dropout) and 0.50. The other parameters are: . The models have been trained for 8000 epochs. No minibatch has been used. The oscillating line is the one evaluated with regularization.

In Figure 4-12 you can see the evolution of the MSE for the training and the dev dataset in case of dropout (rate = 0.5).

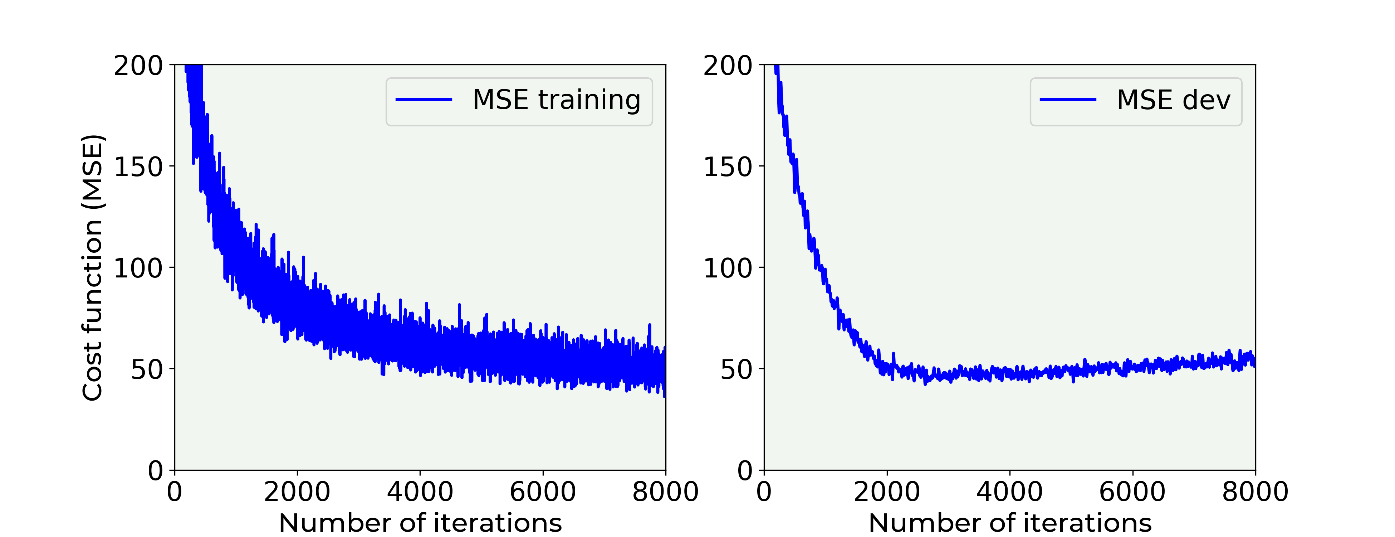


Figure 4-12. MSE for the training and the dev dataset in case of dropout (rate = 0.50)

In Figure 4-13 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.

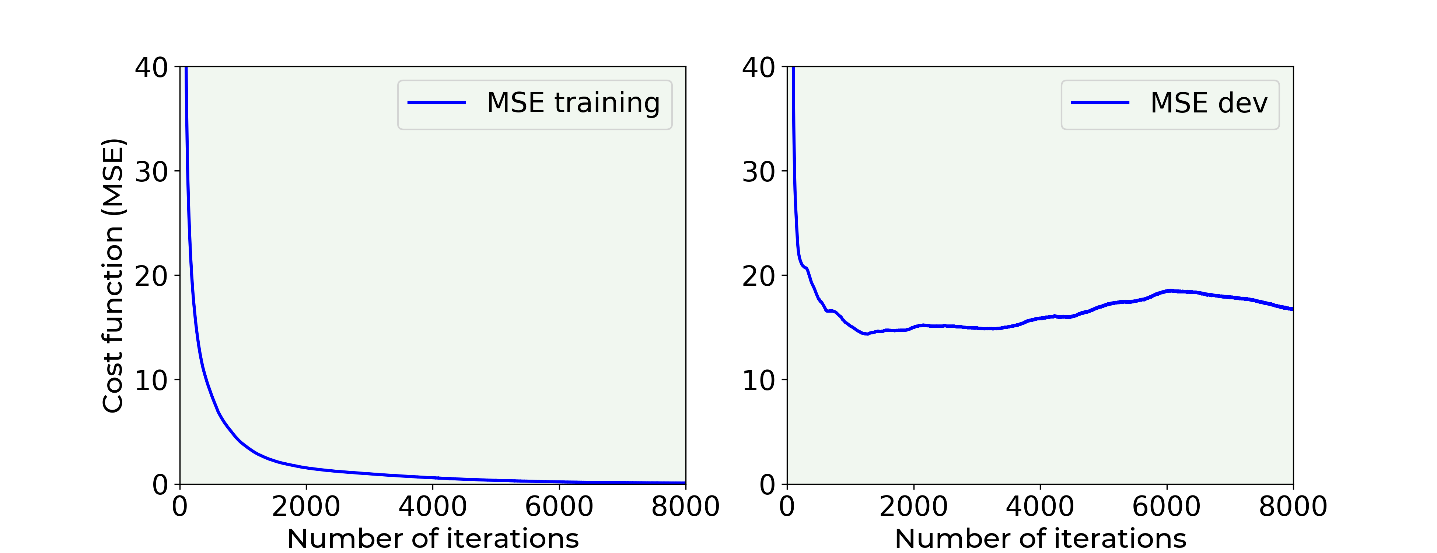


Figure 4-13. MSE for the training and the dev dataset in case of no dropout (rate = 0.0)

In Figure 4-13 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 4-12 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 4-13.

**Note** When applying dropout, your metric (in this case the MSE) will oscillate, so do not 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 4-14 you can see the and plotted on the same plot.

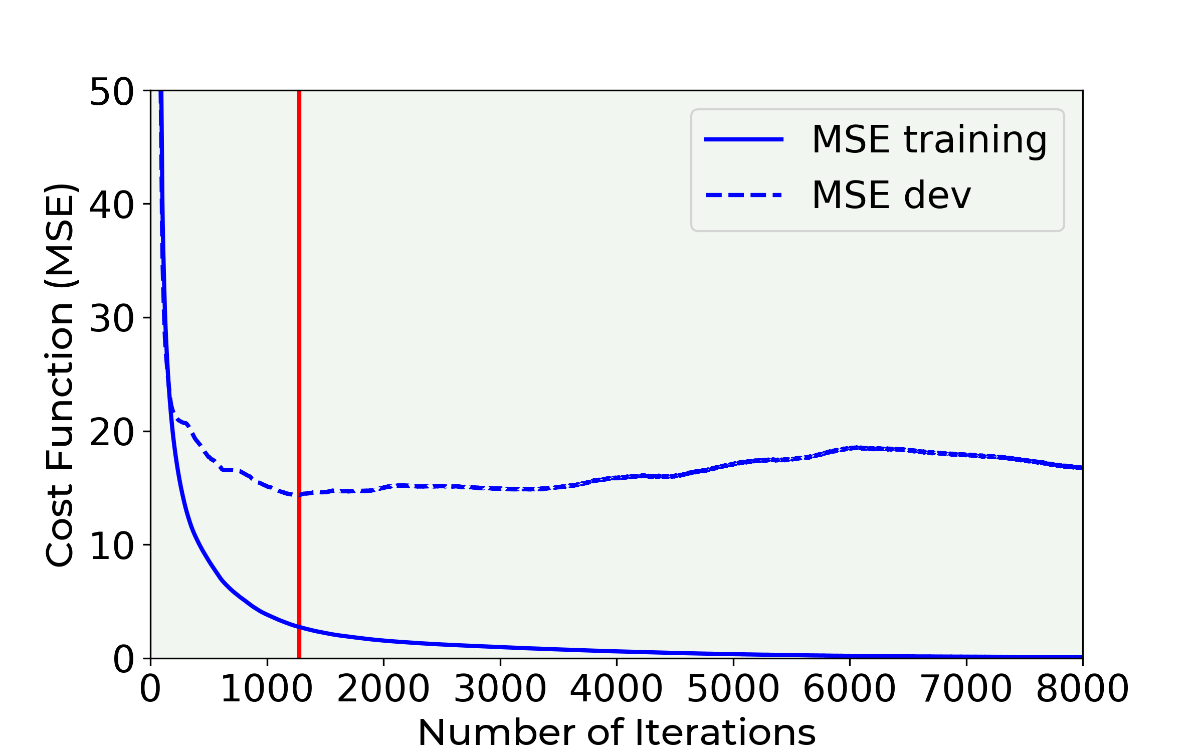


Figure 4-14. MSE for the training and the dev dataset in case of no dropout (rate = 0.0). Early stopping consists in stopping the learning phase at the iteration when the is minimum (indicated with a vertical line in the plot).

Early stopping simply consists in stopping the training at the point when the has its minimum (see Figure 4-14, 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. It is usually preferable 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 easily applicable):

* Get more data. This is the simplest way of fighting overfitting. Unfortunately, very often in real life this is not possible. 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.

# Exercises

EXERCISE 1 (LEVEL EASY)

Try to determine which architecture (number of layers and number of neurons) is not overfitting the Boston dataset. When the network starts overfitting? Which network would give a good result? Try (at least) the following combinations:

|  |  |
| --- | --- |
| **Number of layers** | **Number of neurons for each layer** |
| 1 | 3 |
| 1 | 5 |
| 2 | 3 |
| 2 | 5 |

EXERCISE 2 (LEVEL MEDIUM)

Find the minimum value for (in the case of ) for which the overfitting stops. Perform a set of tests using the function hist, model = create\_and\_train\_reg\_model\_L2(train\_x, train\_y, dev\_x, dev\_y, 20, 4, 0.0) varying the value of from 0 to 10.0 in regular increment (you can decide what values you want to test). Use at minimum the values: 0, 0.5, 1.0, 2.0, 5.0, 7.0, 10.0, 15.0. After that, make a plot of the value for the cost function on the training dataset and on the dev dataset vs. .

EXERCISE 3 (LEVEL MEDIUM)

In the regularization example applied to the Boston dataset, plot the amount of weights close to zero in hidden layer 3 vs. . Considering only layer 3, plot the quantity (np.sum(np.abs(weights3) < 1e-3)) / weights3.size \* 100.0 we have evaluated before and calculate it for several values of . Consider at least: 0, 0.5, 1.0, 2.0, 5.0, 7.0, 10.0, 15.0. Plot then the value vs. . What shape do the curve have? Does it flatten out?

EXERCISE 4 (LEVEL VERY HARD)

Implement regularization from scratch.

# References

[1] Delve (Data for Evaluating Learning in Valid Experiments), “The Boston Housing Dataset”, [www.cs.toronto.edu/~delve/data/boston/bostonDetail.html](http://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html), 1996, last accessed 22.03.2021

[2] Bishop, C.M, (1995) Neural Networks for Pattern Recognition, Oxford University Press

[3] Goodfellow, I.J. et al., Deep Learning, MIT Press

[4] Kukačka, J. et al., Regularization for deep learning: a taxonomy, arXiv: 1710.10686v1, available here: <https://goo.gl/wNkjXz>, last accessed 28.03.2021

1. Don’t worry if you don’t understand how Adam works. Just think of it as a smarter gradient descent. Later in the book you will find an entire chapter dedicated to advanced optimizers. [↑](#footnote-ref-1)
2. In a statistical-classification problem with two classes, a decision boundary or decision surface is a surface that partitions the underlying space into two sets, one for each class (source wikipedia: <https://goo.gl/E5nELL>). [↑](#footnote-ref-2)