UNIVERSITA' DEGLI STUDI DI PADOVA
DEPARTMENT OF INFORMATION ENGINEERING

ICT for Internet and Multimedia

Neural Networks and Deep Learning Course

Homework 1:Supervised Deep Learning Report

Fatıma Rabia YAPICIOĞLU
2049536

14 November 2021

CONTENTS

Instructions and warnings about running the notebook

*If you run this notebook on the Colab please make sure that you download the Skorch Library by deactivating the comment line 'pip install skorch'. I've written the code and run the notebook in my local environment, and as you can see from the outputs it compiles with all the necessary libraries successfully. Run the notebook in a vertical manner starting from the first cell to the end. Runtime takes 15-20 mins due to the GridSearchCV and CrossValidation methods. But it successfully converges at the end and prints results. Each chapter in this report has been neatly designed as in the following structure: Introduction to the tasks that will be implemented, technical implementation details, and results concerning the experiments. For some very large tables, I only provided a preview on the report, you can see all of them in the *.ipynb. Thank you.*

1. Regression Task

1.1. Showing the effect of training when optimizing weights and bias

First of all, I wanted to show how PyTorch's machine learning model progressively develops a best-fit line for a given set of data points. The first model that has been created and trained consists of a randomly initialized weight and bias, and the initial results before training can be seen as in Figure 1.1. Obviously, the fitted red line has not performed well and the difference between the predicted values and actual values is high. This result also leads to error(Mean Squared Error, MSE) and overall cost function to be high.

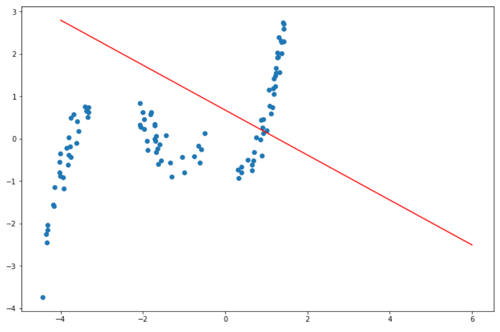


Figure 1.1

After setting the optimizer as SGD and tuning epochs as 100 steps, I trained the model and fitted the red line as in Figure 1.2 once again with the tuned weights and biases. Simple linear regression which tries to fit our data points is producing better results than the previous randomly initialized weights and biases. But, in order to get better fitting we may use polynomial regression to create a curved line over the data points we have.

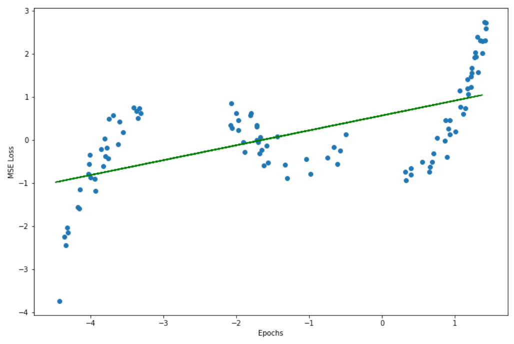


Figure 1.2

1.2. Creating a more complex model with: regularization methods & different optimizers

Observing the effects of training and understanding the difference between these two results, I've used the Skorch framework and NeuralNetRegressor in order to experiment a more complex regression model whose structure can be found as follows in the Figure 1.3. As a regularization method I've placed a DropOut. Creating a very complex model for the basic and small dataset may lead to overfit which may cause an inefficient learning experience. Therefore, for this simple dataset I didn't find it correct to add so many hidden layers with high numbers of neurons in order to avoid overfit and used the DropOut regularization method. Dropout is a simple way to regularize a network by randomly reducing the number of neurons, thus preventing the network from overfitting.

```
LinearRegressorRegularized(
  (module): Sequential(
    (0): Linear(in_features=1, out_features=10, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=10, out_features=1, bias=True)
  )
)
```

Figure 1.3.

I've also demonstrated how this linear regression model structure works with different advanced optimizers. I tried two different advanced optimizers with the Linear Regression model and I'll demonstrate four differences with the Classifier model in the following sections, and then compare them.

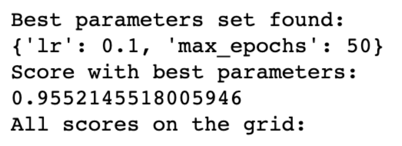
1.3. Applying Cross-Validation & GridSearch to the Regression Model

In order to apply cross validation techniques to keep track of validation loss and train loss simultaneously again we can use the Skorch framework. In the NeuralNetRegressor constructor, if we didn't indicate a specific value by default, 20% of the incoming data is reserved for validation. Moreover, in order to find the optimal hyper-parameters of a model which results in the most accurate predictions, we can use the Skorch framework and feed some cumulative parameters to the GridSearchCV. The GridSearchCV also provides cross-validation experimentation by keeping the parameter called cv(in my case cv = 4, so I splitted train data into 4 splits). I've indicated different learning rates ([0.01,0.001,0.005,0.1]) and different number of epochs ([50,100,150]), in order to find the optimal hyperparameters. Also, I've tried to create two different models with different optimizers(SGD and Adam). As a result of the GridSearchCV we may observe the results as in the following Figure 1.4.

1.4. Reporting the Results Concerning the Created Regression Models

There are two regularized complex regression models that I've created: one with the SGD optimizer and the other one is with the Adam optimizer. They both use cross-validation and GridSearchCV methods and try to find the best combination among different learning rates and epochs which are explained in the above section.

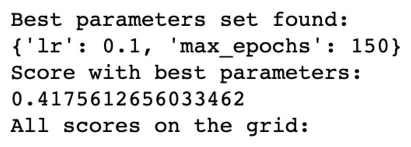
Regularized Regression Model
Optimizer: SGD



```
Best parameters set found:
{'lr': 0.1, 'max_epochs': 50}
Score with best parameters:
0.9552145518005946
All scores on the grid:
```

Figure 1.4

Regularized Regression Model
Optimizer: Adam



```
Best parameters set found:
{'lr': 0.1, 'max_epochs': 150}
Score with best parameters:
0.4175612656033462
All scores on the grid:
```

Figure 1.5

You can see the parameter grid tables in detail in my notebook, it explains which combination of parameters have been tried and which results that they produced, along with their ranking parameters at the last column. According to this table, for the SGD Optimizer the best combination for the learning rate and epoch numbers was lr: 0.1, max_epochs: 50. Also for the model with the Adam optimizer the combination which gives the best score has been reported as lr: 0.1, max_epochs: 150.

2. Classification Task

2.1. Creating a CNN, observing classification results, and visualizing the hidden layer weights

The goal here is to train a Convolutional Neural Network that maps an input image (from fashionMNIST) to one of ten classes (multi-class classification problem with mutually exclusive classes). First of all, I've tried to understand the data structure and created the following visualization as in Figure 1.6.

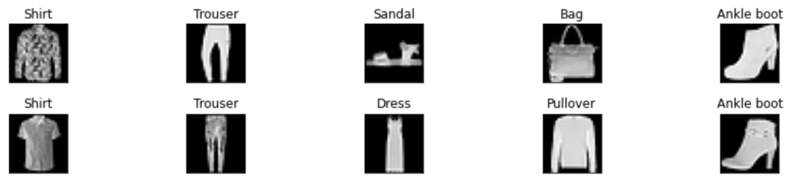
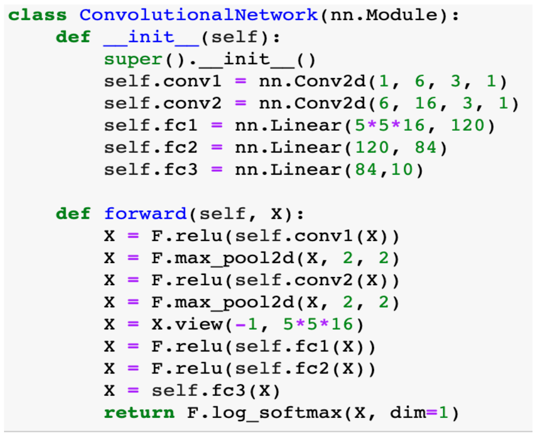


Figure 1.6

In the first CNN structure that I've created I didn't use any framework that automates the training process as Skorch but I manually wrote the code for the training process. The first CNN Model structure is as follows in Figure 1.7. This is a very basic model that has been created to compare the following more complex CNN structures and regularization techniques for comparison purposes. I set the loss function as Cross Entropy and the optimizer as SGD, then I trained the model with the 100 epochs. One can easily observe that the train loss and the validation loss reduces as we go on with the further number of iterations.



```
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.fc1 = nn.Linear(5*5*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 5*5*16)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return F.log_softmax(X, dim=1)
```

Figure 1.7

The last training loss is 0.2, validation loss is 0.3, therefore the training accuracy is around 0.78 and the validation accuracy is around 0.60. This may improve with the more number of epochs, finding optimal hyperparameters and applying different regularization methods. In order to see this model's performance on the test set again I obtained one batch of the images in the test set and demonstrated the predicted and actual values of the prediction results as it can be seen in Figure 1.7.

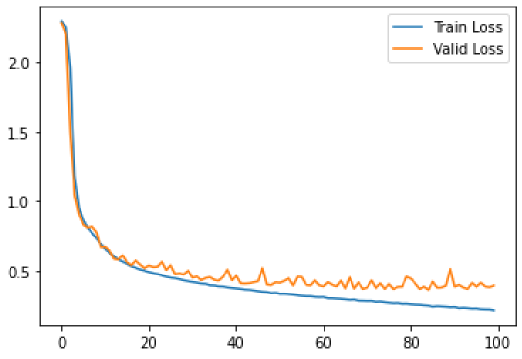


Figure 1.8

The model was in confusion with some of the categories, in order to understand this confusion better further I created a confusion matrix which will be presented in the following sections of this task.

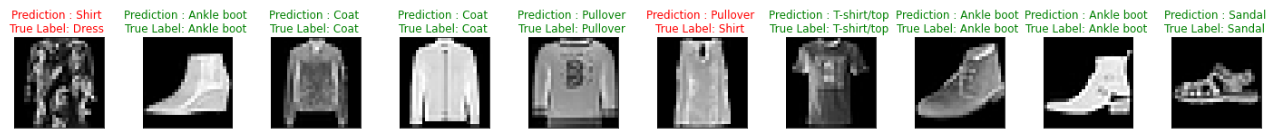


Figure 1.9

We can also visualize the weights in the hidden layers that we have created during the CNN model structure creation. As it can be seen in the following Figure 1.9. Some features in the last layer have higher weight values than the others, this shows which features are most importantly effective on the classification decision process.

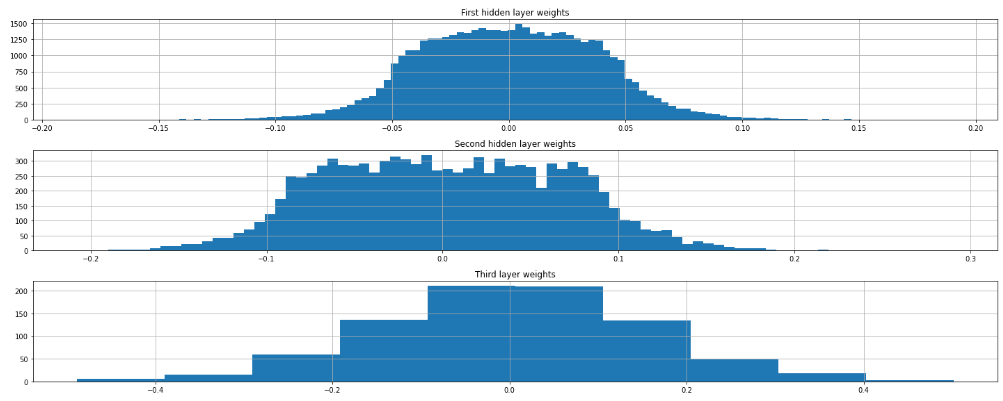
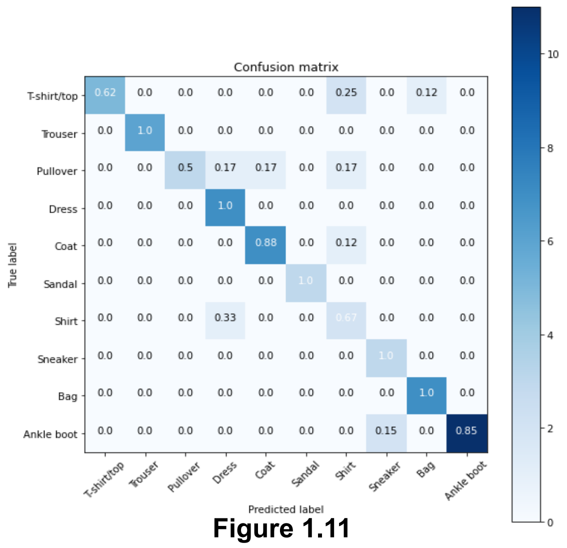


Figure 1.10

2.2. CNN Classification Results Confusion matrix

Producing a confusion matrix is important in order to understand which are the classes that confuses our model's mind, or which points the model makes wrong predictions most frequently. In this confusion matrix for instance, the model is in confusion with the classification of T-shirt/top & Shirt in the rate of 0.25. Also we can see another important rate of 0.33 where the model is in confusion with Shirt & Dress.



True \ Predicted	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	0.62	0.0	0.0	0.0	0.0	0.0	0.25	0.0	0.12	0.0
Trouser	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Pullover	0.0	0.0	0.5	0.17	0.17	0.0	0.17	0.0	0.0	0.0
Dress	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
Coat	0.0	0.0	0.0	0.0	0.88	0.0	0.12	0.0	0.0	0.0
Sandal	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
Shirt	0.0	0.0	0.0	0.33	0.0	0.0	0.67	0.0	0.0	0.0
Sneaker	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
Bag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
Ankle boot	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.15	0.0	0.85

Figure 1.11

2.3. Visualizing the CNN model convolutional layer filters and feature maps

In the first convolutional layer we set the number of filters as 6

Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1)),

and in the second convolutional layer we set the number of filters as 16

Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))

According to the complexity of the model and the dataset we may increase the number of filters by increasing the second argument of the `conv2d(inputs, filters, padding=1)` constructor. When I visualized the first convolutional layer filters I got the following 6 pictures of 3 x 3 filter visualizations as a result as in the Figure 1.12. Also, we can see 16 pictures of 3 x 3 filters which belong to the second convolutional layer as in Figure 1.13.

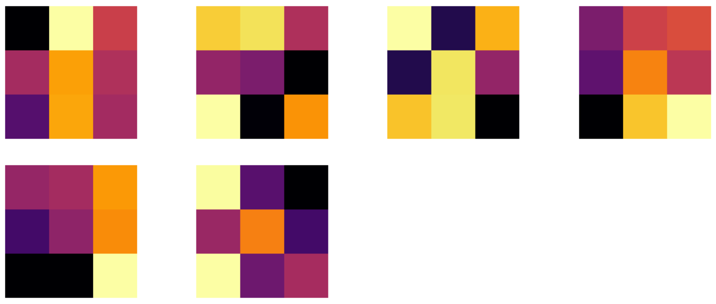


Figure 1.12

Figure 1.13

From there we can move on to visualizing the first and second layers feature maps and observe the pictures further. The following Figure 1.14 shows the visualization of the feature map of the first layer. I've used the sample data which is already extracted on the Jupyter Notebook.

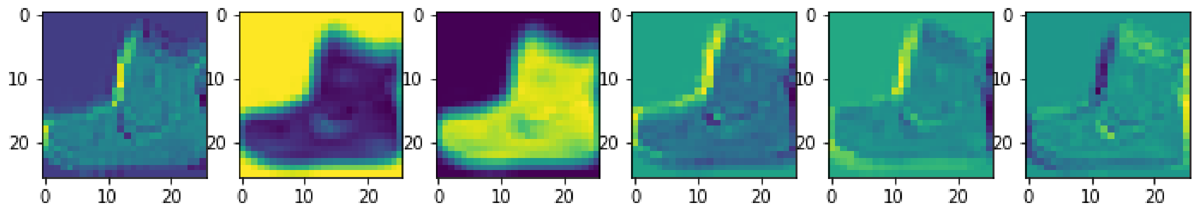


Figure 1.14

The first features are simple and basic but when we go on with the second convolutional layer the model tries to detect more detailed features. The feature map for the second convolutional layer can be investigated in detail as in the following 16 pictures.

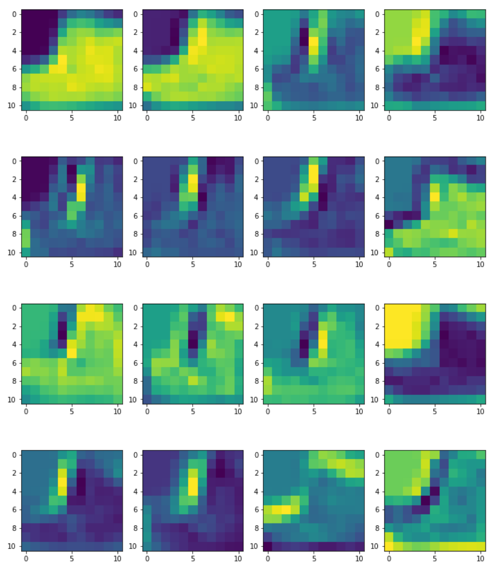


Figure 1.15

2.4. Applying Cross Validation to the CNN Model and observing different optimizers with advanced optimizers

In order to perform cross-validation on CNN model, I've created another structure which also utilizes DropOut regularization technique and then once again by using the Skorch framework and the NeuralNetClassifer. By using this Convolutional neural network Regularized I've tried 4 different advanced optimizers and compared the validation losses of each in the same plot which can be found in Figure 1.16.

```
class ConvolutionalNetworkRegularized(nn.Module):
    def __init__(self):
        super(ConvolutionalNetworkRegularized, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Figure 1.16

As it can be seen in detail in Figure 1.17, the Adam optimizer has converged to its optimal value at a better speed than the other advanced optimizers. But in terms of convergence performance the adadelta found optimal minima more efficient than the other optimizers.

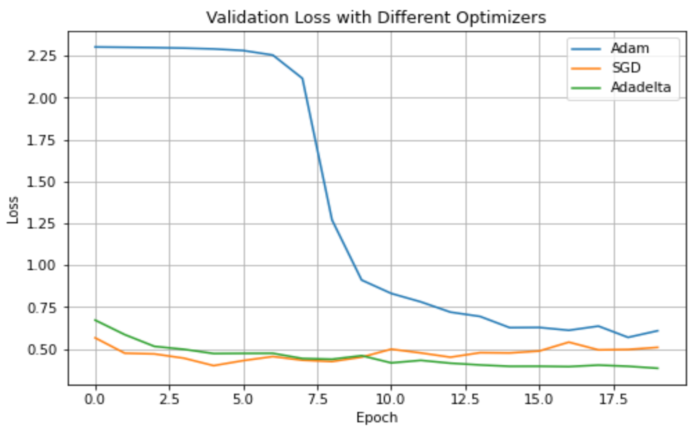
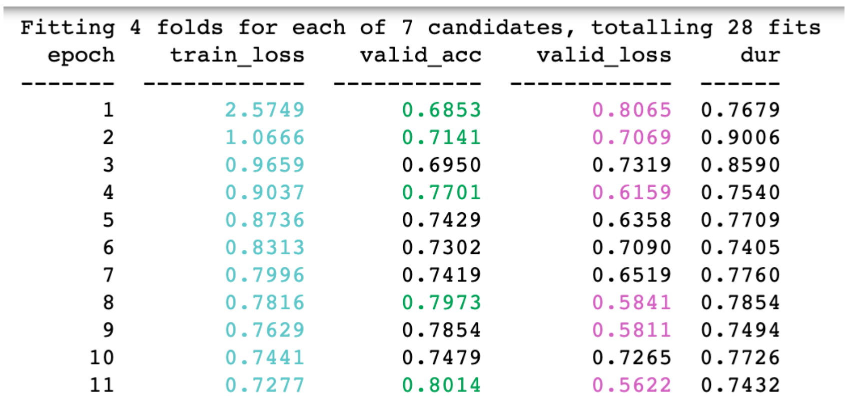


Figure 1.17

2.5. Applying GridSearch to the CNN Model

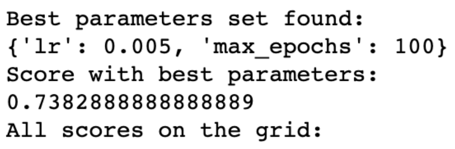
Finally in the same model structure I've experimented the cross-validation method with the GridSearchCV hyperparameter tuning technique. Overall this model produced better results than the one I've created before. During the Grid Search I fed the model with the different learning rates(7 candidates) and different number of epochs (50,100,200). Obviously the validation accuracy continues to improve and the validation loss continues to reduce as we go on with the high number of iterations. Figure 1.18 shows the Convolutional Neural Network performance.



```
Fitting 4 folds for each of 7 candidates, totalling 28 fits
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.5749       0.6853        0.8065  0.7679
      2        1.0666       0.7141        0.7069  0.9006
      3        0.9659       0.6950        0.7319  0.8590
      4        0.9037       0.7701        0.6159  0.7540
      5        0.8736       0.7429        0.6358  0.7709
      6        0.8313       0.7302        0.7090  0.7405
      7        0.7996       0.7419        0.6519  0.7760
      8        0.7816       0.7973        0.5841  0.7854
      9        0.7629       0.7854        0.5811  0.7494
     10        0.7441       0.7479        0.7265  0.7726
     11        0.7277       0.8014        0.5622  0.7432
```

Figure 1.18

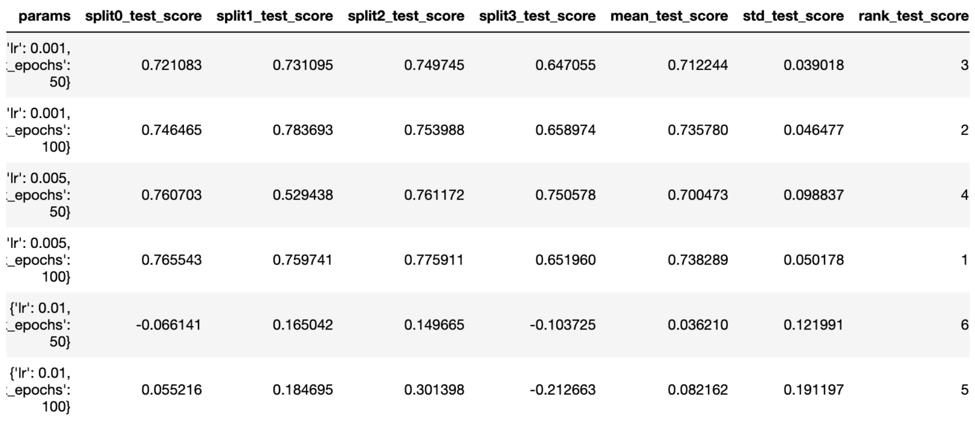
The result that the GridSearchCV produced(best parameters set found, scores with the best parameters) can be documented as follows in Figure 1.19.



```
Best parameters set found:
{'lr': 0.005, 'max_epochs': 100}
Score with best parameters:
0.7382888888888889
All scores on the grid:
```

Figure 1.19

You can also investigate all parameter combinations on the parameter grid which I presented in my notebook. It looks like in the following picture but since it is too wide it was not possible for me to include in the report here.



params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	mean_test_score	std_test_score	rank_test_score
'lr': 0.001, _epochs': 50}	0.721083	0.731095	0.749745	0.647055	0.712244	0.039018	3
'lr': 0.001, _epochs': 100}	0.746465	0.783693	0.753988	0.658974	0.735780	0.046477	2
'lr': 0.005, _epochs': 50}	0.760703	0.529438	0.761172	0.750578	0.700473	0.098837	4
'lr': 0.005, _epochs': 100}	0.765543	0.759741	0.775911	0.651960	0.738289	0.050178	1
{'lr': 0.01, _epochs': 50}	-0.066141	0.165042	0.149665	-0.103725	0.036210	0.121991	6
{'lr': 0.01, _epochs': 100}	0.055216	0.184695	0.301398	-0.212663	0.082162	0.191197	5

Figure 1.20