Table of Contents

[GridSearchCV 1](#_Toc125962291)

[RandomSearchCV 3](#_Toc125962292)

[Visualizing Loss and Accuracy for a Classification Model 4](#_Toc125962293)

[Viewing Losses for Linear Models 9](#_Toc125962294)

[MLPClassifier and MLPRegressor from Scikit-learn 12](#_Toc125962295)

[GridSearchCV with MLPClassifier 14](#_Toc125962296)

[MLPRegressor 15](#_Toc125962297)

## GridSearchCV

Since many parameters are not linearly related it can help to grid search many at once. That way it is possible to have a better sense of which parameter settings perform well together.

Example : GridSearch

GridSearch offers an approach to search multiple determine the best parameters for adding a new layer. The process may take between 20 minutes to an hour. When finished the output will

show the best performing model with parameters at the top. The other models with their average score and standard deviation will appear after.

|  |
| --- |
| Best: -19.212292 using {'numNeurons': 25, 'initializer': 'glorot\_uniform', 'activation': 'softplus'}  -26.609111 (8.702621) with: {'numNeurons': 15, 'initializer': 'zero', 'activation': 'linear'}  -47.283887 (14.076808) with: {'numNeurons': 15, 'initializer': 'glorot\_uniform', 'activation': 'softsign'}  -23.229303 (4.183168) with: {'numNeurons': 15, 'initializer': 'lecun\_uniform', 'activation': 'relu'}  -19.212292 (2.931735) with: {'numNeurons': 25, 'initializer': 'glorot\_uniform', 'activation': 'softplus'}  -25.405355 (9.082541) with: {'numNeurons': 30, 'initializer': 'he\_normal', 'activation': 'softplus'}  … |

Here is the code:

|  |
| --- |
| import pandas as pd  import tensorflow  from sklearn.model\_selection import train\_test\_split  PATH = "/users/pm/desktop/daydocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  dataset = df.values  # split into input (X) and output (Y) variables  X = dataset[:,0:13]  y = dataset[:,13]  ROW\_DIM = 0  COL\_DIM = 1  x\_arrayReshaped = X.reshape(X.shape[ROW\_DIM],  X.shape[COL\_DIM])  # Convert DataFrame columns to vertical columns of target variables values.  y\_arrayReshaped = y.reshape(y.shape[ROW\_DIM],1)  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.3, random\_state=0)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp,  y\_temp, test\_size=0.5, random\_state=0)  n\_features = X\_train.shape[1]  from keras.models import Sequential  from keras.layers import Dense  from keras.wrappers.scikit\_learn import KerasRegressor  from sklearn.model\_selection import GridSearchCV  # Define the model.  def create\_model(numNeurons=5, initializer='uniform', activation='softplus'):  # create model  model = Sequential()  model.add(Dense(25, kernel\_initializer='uniform',  input\_dim=n\_features, activation='softplus'))  model.add(Dense(numNeurons, kernel\_initializer=initializer,  activation=activation))  model.add(Dense(1, kernel\_initializer='he\_normal', activation='softplus'))  opt = tensorflow.keras.optimizers.Adam(learning\_rate=0.005)  # Compile model  model.compile(loss='mse', optimizer=opt)  return model  ### Grid Building Section #######################  # Define the parameters to try out  params = { 'activation' : ['softmax', 'softplus', 'softsign', 'relu', 'tanh',  'sigmoid', 'hard\_sigmoid', 'linear'],  'numNeurons':[10, 15, 20, 25, 30, 35],  'initializer': ['uniform', 'lecun\_uniform', 'normal', 'zero',  'glorot\_normal', 'glorot\_uniform', 'he\_normal', 'he\_uniform']  }  model = KerasRegressor(build\_fn=create\_model, epochs=100,  batch\_size=9, verbose=1)  grid = GridSearchCV(estimator=model, param\_grid=params, n\_jobs=-1, cv=3)  #################################################  grid\_result = grid.fit(X\_train, y\_train)  # summarize results  print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))  means = grid\_result.cv\_results\_['mean\_test\_score']  stds = grid\_result.cv\_results\_['std\_test\_score']  params = grid\_result.cv\_results\_['params']  for mean, stdev, param in zip(means, stds, params):  print("%f (%f) with: %r" % (mean, stdev, param)) |

## RandomSearchCV

To save time you can use RandomSearchCV which implements a random subset of combinations of the GridSearchCV algorithm. RandomSearchCV() is faster than GridSearchCV() but RandomSearchCV is less accurate. Still, during a first pass you might consider using RandomSearchCV() to discover suitable parameter combinations and ranges before running a more time consuming grid search.

Note: Neural networks are sensitive to scaling so ideally scaling should be performed or at least tried as an option. I have left scaling out to avoid complexity but really it probably should be performed.

Example : RandomSearchCV

This example converts Example 1 to use a much quicker search for suitable parameters.

Switching to using a RandomSearchCV() function instead of the GridSearchCV() function only involves swapping out two lines of code. To make the change, replace:

|  |
| --- |
| grid = GridSearchCV(estimator=model, param\_grid=params, n\_jobs=-1, cv=3) |

with:

|  |
| --- |
| from sklearn.model\_selection import RandomizedSearchCV  grid = RandomizedSearchCV(model, param\_distributions = params, cv = 3) |

Exercise (2 marks)

Explain two advantages of RandomizedSearchCV here:

|  |
| --- |
|  |

Exercise (6 marks)

Modify Example 2 to perform a random search on the following parameters:

|  |
| --- |
| params = { 'activation' : ['relu', 'sigmoid'],  'numNeurons': [50, 100, 200],  'numHiddenLayers':[1,2,3],  'initializer': ['normal']  } |

Hints:

* You will need to adjust the parameters of the create\_model() function.
* The hidden layers will need to be added in a loop.

Show your revised code as text here:

|  |
| --- |
|  |

Show a screenshot which displays the output including the best model. Ensure that all four parameters are visible in the screenshot.

|  |
| --- |
|  |

Exercise (4 marks)

Calculate the root mean square error with your best model from the grid search in Example 2.

predictions = grid.best\_estimator\_.predict(X\_test)

Show your revised code here:

|  |
| --- |
|  |

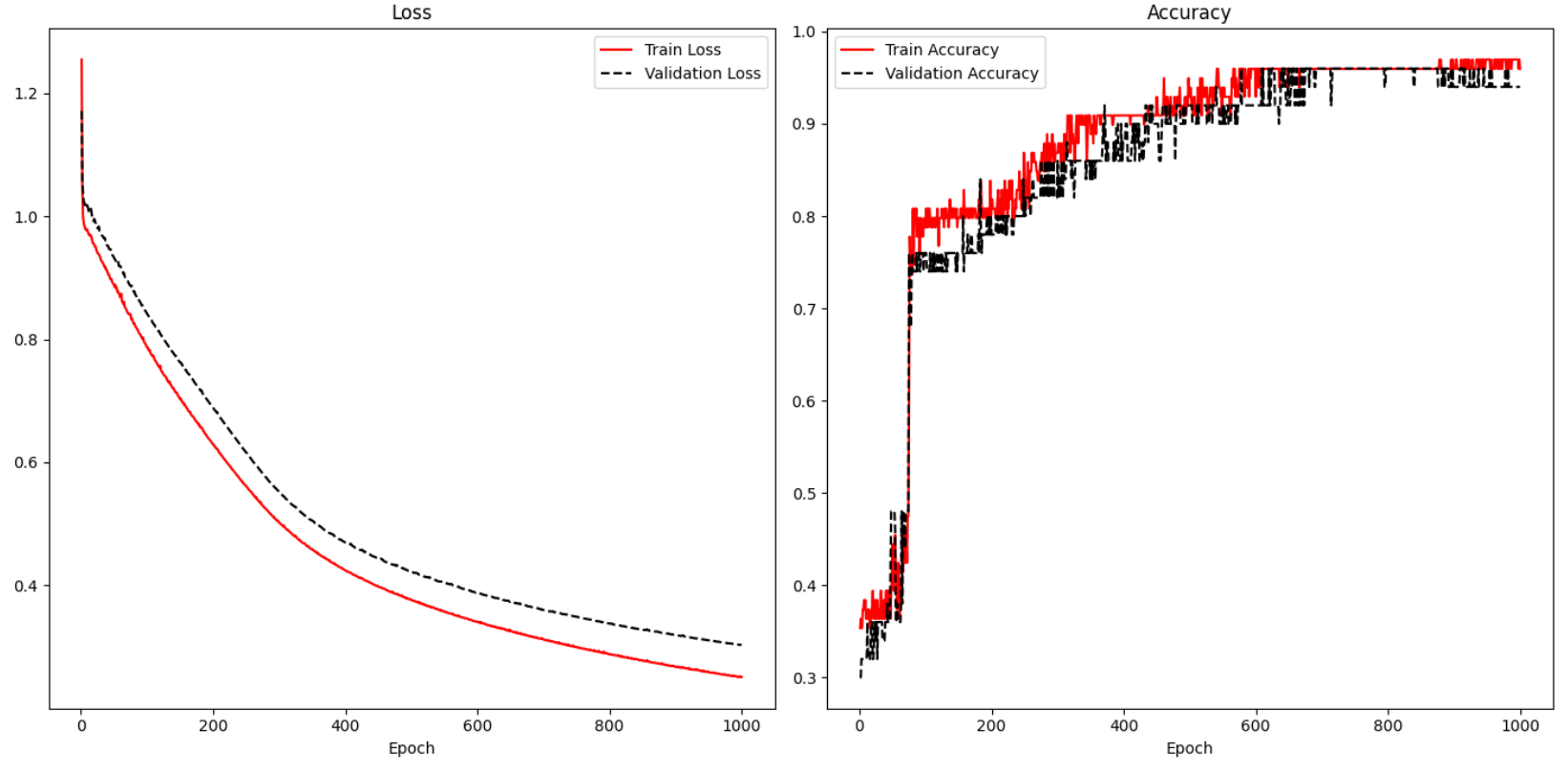
Show a screenshot of the output here:

|  |
| --- |
|  |

## Visualizing Loss and Accuracy for a Classification Model

Being able to visualize loss and accuracy during training and testing helps to understand the effectiveness of the hyperparameters. Figure 1 shows an ideal scenario for loss and accuracy for a classification model. Loss decline is smooth. The accuracy peaks at approximately 200 epochs. There is some fluctuation in accuracy. The information suggests that potentially 200 to 400 epochs are required for reaching peak performance during training with the current set of parameters. You will also notice that validation (test) sets have slightly higher losses and the tests sets are also slightly less accurate than the training sets.

Figure : Visualizing Loss and Accuracy Parameters for a Classification Model



To enable display of train and validation results, validation\_data arguments must be supplied in the model.fit() instruction. Note that the data has already been split into three data sets so that the model can be fit with training data and validated during training with validation data. The test data set is unseen data which is then used after the model has been trained for a final evaluation.

|  |
| --- |
| X\_train, **X\_temp**, y\_train, **y\_temp** = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.3)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(**X\_temp**,  **y\_temp**, test\_size=0.5) |

A history variable is needed to store the loss and accuracy results.

|  |
| --- |
| history = model.fit(X\_train, y\_train, epochs=1000, batch\_size=28, verbose=1,  validation\_data=(X\_val, y\_val)) |

To track the accuracy of the result, metrics=['accuracy'] is included in the compile statement.

|  |
| --- |
| model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy',  metrics=['accuracy']) |

Example : Visualizing Loss and Accuracy

Here is the code that is used to display the accuracy and losses in Figure 1 when fitting an iris data set prediction model. The locations where the history variable is set and referenced are highlighted to show how the losses and accuracy are tracked.

A detailed classification report shows how well the model performs with test data which the model did not see during training or validation.

|  |
| --- |
| precision recall f1-score support  0 1.00 1.00 1.00 9  1 1.00 0.89 0.94 9  2 0.88 1.00 0.93 7 |

Here is the code:

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  PATH = "/Users/pm/Desktop/DayDocs/data/"  df = pd.read\_csv(PATH + 'iris\_old.csv')  df.columns = ['Sepal L', 'Sepal W', 'Petal L', 'Petal W', 'Iris Type']  print(df)  # Convert text to numeric category.  # 0 is setosa, 1 is versacolor and 2 is virginica  df['y'] = LabelEncoder().fit\_transform(df['Iris Type'])  # Prepare the data.  dfX = df.iloc[:, 0:4] # Get X features only from columns 0 to 3  dfY = df.iloc[:, 5:6] # Get X features only from column 5  ROW\_DIM = 0  COL\_DIM = 1  # Create vertical array of features.  x\_array = dfX.values  x\_arrayReshaped = x\_array.reshape(x\_array.shape[ROW\_DIM],  x\_array.shape[COL\_DIM])  y\_array = dfY.values  y\_arrayReshaped = y\_array.reshape(y\_array.shape[ROW\_DIM],  y\_array.shape[COL\_DIM])  # Split into train, validation and test data sets.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(  x\_arrayReshaped, y\_arrayReshaped, test\_size=0.33)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(  X\_temp, y\_temp, test\_size=0.50)  n\_features = X\_train.shape[COL\_DIM]  # Define the model.  model = Sequential()  # Hidden layer 1 (also receives the input layer)  model.add(Dense(2, activation='relu', input\_shape=(n\_features,)))  # Output layer  model.add(Dense(3, activation='softmax'))  # Compile the model.  model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])  # Fit the model.  history = model.fit(X\_train, y\_train, epochs=1000, batch\_size=28, verbose=1,  validation\_data=(X\_val, y\_val))  # Evaluate the model with unseen data.  loss, acc = model.evaluate(X\_test, y\_test, verbose=0)  print('Test Accuracy: %.3f' % acc)  # make a prediction  row = [5.1, 3.5, 1.4, 0.2]  yhat = model.predict([row])  print('Predicted: s (class=d)' + str(yhat))  import matplotlib.pyplot as plt  def showLoss(history):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 1)  # Visualize loss history for training data.  plt.plot(epoch\_count, training\_loss, label='Train Loss', color='red')  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, 'r--', label='Validation Loss',  color='black')  plt.xlabel('Epoch')  plt.legend(loc="best")  plt.title("Loss")  def showAccuracy(history):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  # Visualize loss history for training data.  plt.plot(epoch\_count, training\_loss, label='Train Accuracy', color='red')  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, 'r--',  label='Validation Accuracy', color='black')  plt.xlabel('Epoch')  plt.legend(loc="best")  plt.title('Accuracy')  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  showLoss(history)  showAccuracy(history)  plt.show()  from sklearn.metrics import classification\_report  # Provide detailed evaluation with unseen data.  y\_probability = model.predict(X\_test)  import numpy as np  # Convert probability arrays to whole numbers.  # eg. [0.0003, 0.01, 0.9807] becomes 2.  predictions = np.argmax(y\_probability, axis=-1)  print(classification\_report(y\_test, predictions)) |

Exercise (6 marks)

Starting with the following code which loads a bill fraud data set, create a neural network to predict if the bill is fraudulent. Track losses and accuracy and show the results at the end.

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  import numpy as np  PATH = "/Users/pm/Desktop/DayDocs/data/"  df = pd.read\_csv(PATH + 'bill\_authentication.csv')  # Convert text to numeric category.  # 0 is setosa, 1 is versacolor and 2 is virginica  y = df['Class']  X = df  del X['Class']  ROW\_DIM = 0  COL\_DIM = 1  # Create vertical array of features.  x\_array = X.values  x\_arrayReshaped = x\_array.reshape(x\_array.shape[ROW\_DIM],  x\_array.shape[COL\_DIM])  y\_array = np.array(y.values)  y\_arrayReshaped = y\_array.reshape(len(y\_array),1) |

Show the loss curve at the end when the model is finished training.

|  |
| --- |
|  |

Even if the precision, recall and accuracy are perfect, what improvement does the plot above suggest is possible?

|  |
| --- |
|  |

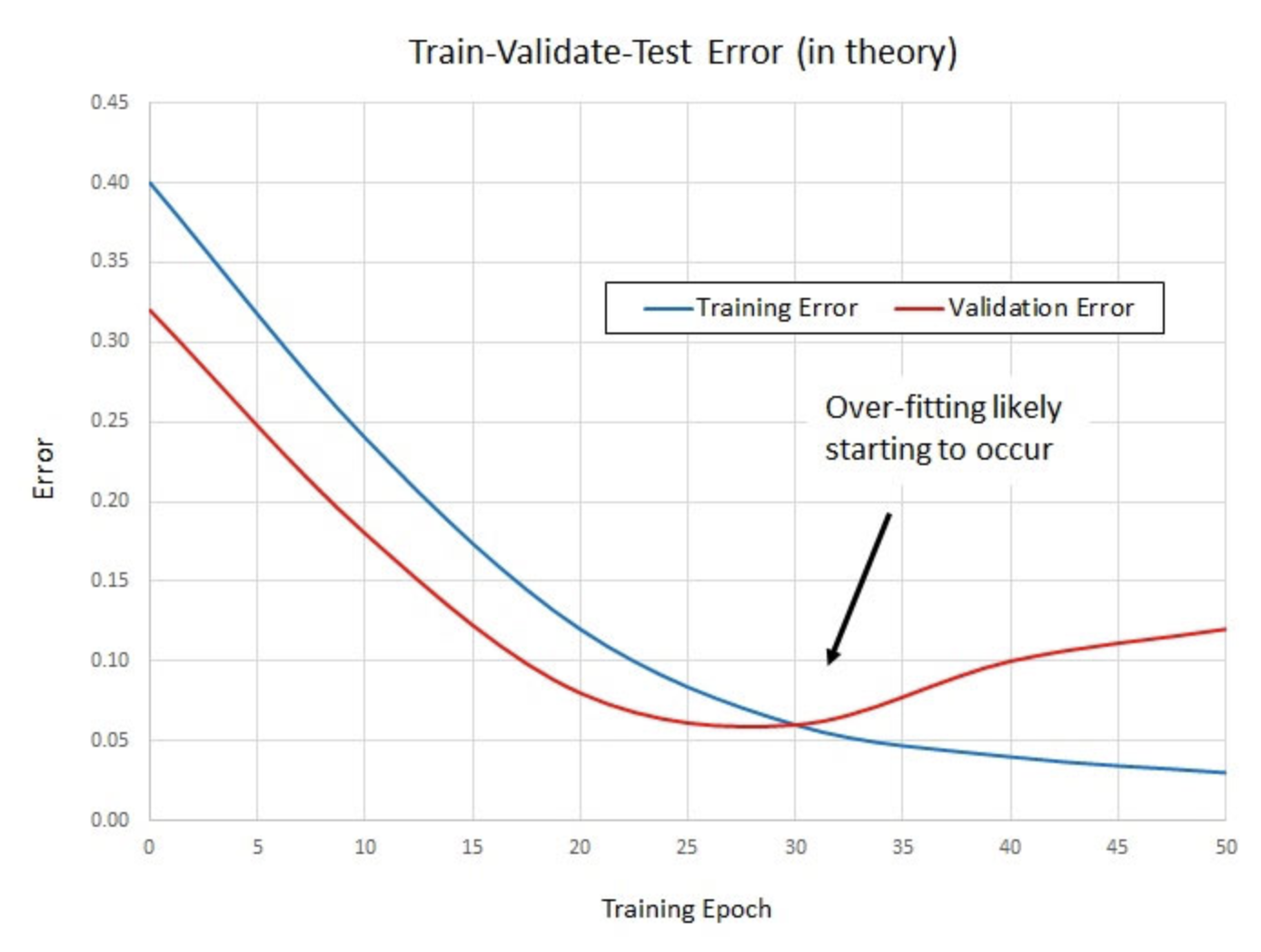
## Viewing Losses for Linear Models

Since linear models use measures such as RMSE to judge the fit of a model, accuracy is not plotted for linear models like in Figure 1. However, the loss plot is still useful for linear models.

The loss plot on the left of Figure 2 shows a situation where overfitting occurs. The overfitting is observed when the test data suddenly demonstrates rising losses while the training losses continue to decline.

The graph at the right of Figure 2 suggests underfitting. For this case, it appears as if the minimum loss has not been reached. The model may be improved by increasing the number of epochs.

Figure : Overfitting and Underfitting

Example : Predicting Housing Price

Last day, the following code for predicting housing prices was discussed. This code implements a neural network that was previously optimized with a grid search.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  dataset = df.values  # split into input (X) and output (Y) variables  X = dataset[:, 0:13]  y = dataset[:, 13]  ROW\_DIM = 0  COL\_DIM = 1  x\_arrayReshaped = X.reshape(X.shape[ROW\_DIM],  X.shape[COL\_DIM])  # Convert DataFrame columns to vertical columns of target variables values.  y\_arrayReshaped = y.reshape(y.shape[ROW\_DIM], 1)  # Splitting data into 3 data sets.  # Train and val are used while model is being fit.  # Test is held back until the end for the final evaluation.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.3, random\_state=0)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp,  y\_temp, test\_size=0.3, random\_state=0)  from keras.models import Sequential  from keras.layers import Dense  from keras.wrappers.scikit\_learn import KerasRegressor  from sklearn.model\_selection import cross\_val\_score  from sklearn.model\_selection import KFold  # define base model  from keras.optimizers import Adam #for adam optimizer  def baseline\_model():  model = Sequential()  model.add(Dense(25, input\_dim=13, kernel\_initializer='uniform',  activation='softplus'))  model.add(Dense(10, kernel\_initializer='lecun\_uniform', activation='softplus'))  model.add(Dense(1, kernel\_initializer='uniform'))  # Use Adam optimizer with the given learning rate  opt = Adam(lr=0.005)  model.compile(loss='mean\_squared\_error')  return model  # evaluate model  estimator = KerasRegressor(build\_fn=baseline\_model, epochs=100,  batch\_size=9, verbose=1)  kfold = KFold(n\_splits=10)  results = cross\_val\_score(estimator, X\_train, y\_train, cv=kfold)  print("Baseline Mean (%.2f) MSE (%.2f) " % (results.mean(), results.std()))  print("Baseline RMSE: " + str(np.sqrt(results.std())))  # So then we build the model.  model = baseline\_model()  history = model.fit(X\_train, y\_train, epochs=100,  batch\_size=9, verbose=1,  validation\_data=(X\_val, y\_val))  predictions = model.predict(X\_test)  mse = metrics.mean\_squared\_error(y\_test, predictions)  print("Neural network MSE: " + str(mse))  print("Neural network RMSE: " + str(np.sqrt(mse))) |

Exercise (1 mark)

Explain why it does not make sense to draw an accuracy plot for the code in Example 4.

|  |
| --- |
|  |

Exercise (6 marks)

Plot the loss function plot that appears when the model in Example 4 is fit. Show your revised program here. Refer to Example 3 for code that plots the loss function. Show a screenshot of the loss plot here:

|  |
| --- |
|  |

It is typical for the validation data to show higher losses than the training losses because the model learns with the training data. With this in mind, explain if you think the loss function shows a reasonably good fit, underfitting or overfitting.

|  |
| --- |
|  |

## MLPClassifier and MLPRegressor from Scikit-learn

The MLPClassifier and MLPRegressor machine learning pipeline classes from scikit-learn offer

a simple and fast way to generate neural networks while hiding the details. MLP stands for multi-layer perceptron which essentially is a sequential neural network with back propagation.

Not only are the MLPClassifier and MLPRegressor useful classes from an educational perspective, they can be used for simple ANN (artificial neural network) implementations and they can also be used to explore suitable ANN parameters which could then be used with a different framework.

At a very basic level, the MLPClassifier (or MLPRegressor) can be instantiated without any parameters.

# Create and fit model.

model = MLPClassifier()

model.fit(trainX\_scaled, y\_train)

print(model)

# Evaluate model.

predicted\_y = model.predict(testX\_scaled)

The basic instantiations of these models are actually very competitive and often perform just as well as models that are developed with basic grid searching.

Example : MLPClassifier

In this example, the MLPClassifier is initialized without any parameters. The results show a reasonable learning rate and decent metrics (results will vary from run to run):

Figure : Loss Plot for MLPClassifier



|  |
| --- |
| precision recall f1-score support  0 1.00 0.93 0.96 14  1 0.86 0.86 0.86 14  2 0.89 0.94 0.91 17 |

The default model often is very competitive with grid searched models. When the model is fitted, the parameters of the model can be displayed with the get\_params() function.

|  |
| --- |
| print(model.get\_params()) |

These are the parameters that were chosen for the default instantiation. You will notice some parameters such as warm\_start, early\_stopping, and momentum which we will talk about in a future lesson.

|  |
| --- |
| {'activation': 'relu', 'alpha': 0.0001, 'batch\_size': 'auto', 'beta\_1': 0.9, 'beta\_2': 0.999, 'early\_stopping': False, 'epsilon': 1e-08, 'hidden\_layer\_sizes': (100,), 'learning\_rate': 'constant', 'learning\_rate\_init': 0.001, 'max\_fun': 15000, 'max\_iter': 200, 'momentum': 0.9, 'n\_iter\_no\_change': 10, 'nesterovs\_momentum': True, 'power\_t': 0.5, 'random\_state': None, 'shuffle': True, 'solver': 'adam', 'tol': 0.0001, 'validation\_fraction': 0.1, 'verbose': False, 'warm\_start': False} |

Here is the code for the example:

|  |
| --- |
| from sklearn import metrics  from sklearn.neural\_network import MLPClassifier  from sklearn.model\_selection import train\_test\_split, GridSearchCV  import matplotlib.pyplot as plt  from sklearn.preprocessing import StandardScaler  import pandas as pd  plt.style.use('ggplot')  # Create numeric target for iris type.  dataset = pd.read\_csv('/Users/pm/Downloads/iris\_v2.csv')  dataset.iris\_type = pd.Categorical(dataset.iris\_type)  # Prepare x and y.  dataset['flowertype'] = dataset.iris\_type.cat.codes  del dataset['iris\_type']  y = dataset['flowertype']  X = dataset  del X['flowertype']  # Split X and y.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,  y, test\_size=0.30)  # Scale X and Y.  scX=StandardScaler()  scalerX = scX.fit(X\_train)  trainX\_scaled = scalerX.transform(X\_train)  testX\_scaled = scalerX.transform(X\_test)  # Create and fit model.  model = MLPClassifier()  model.fit(trainX\_scaled, y\_train)  print(model.get\_params()) # Show model parameters.  # Evaluate model.  predicted\_y = model.predict(testX\_scaled)  print(metrics.classification\_report(y\_test, predicted\_y))  print(metrics.confusion\_matrix(y\_test, predicted\_y))  def showLosses(model):  plt.plot(model.loss\_curve\_)  plt.title("Loss Curve")  plt.xlabel('Iterations')  plt.ylabel('Cost')  plt.show()  showLosses(model) |

### GridSearchCV with MLPClassifier

Example : Grid searching the MLPClassifier

This example shows how to fine tune and ideally improve the MLPClassifier by grid searching different parameters. The custom classification report shows good results.

|  |
| --- |
| precision recall f1-score support  0 1.00 1.00 1.00 14  1 1.00 0.93 0.96 14  2 0.94 1.00 0.97 17 |

To build this example, add the following code to Example 5. Note, the **max\_iter** parameter specifies the number of **epochs**. When grid searching you will see warnings when the algorithm does not reach an optimum. My understanding is that it is not possible to eliminate these warnings because the algorithm is grid searching with a fixed set of iteration options.

We are only grid searching a subset of parameters so often the model that is built with a default initialization in Example 5 will do better.

|  |
| --- |
| parameters={  'solver': ['adam', 'sgd'],  'learning\_rate': ['constant', 'adaptive', 'invscaling'],  'hidden\_layer\_sizes': [(200,200), (300,200), (150,150)],  'activation': ["logistic", "relu", "tanh"]  }  model2 = GridSearchCV(estimator=model, param\_grid=parameters,  scoring='accuracy', # average='macro'),  n\_jobs=-1, cv=4, verbose=1,  return\_train\_score=False)  model2.fit(trainX\_scaled, y\_train)  print("Best parameters: ")  print(model2.best\_params\_)  y\_pred = model2.predict(testX\_scaled)  print("Report with grid: ")  print(metrics.classification\_report(y\_test, y\_pred))  print(metrics.confusion\_matrix(y\_test, y\_pred))  showLosses(model2.best\_estimator\_) |

Exercise (2 marks)

Run the code in Example 6. Show a screenshot which displays the parameters for the best model and be sure to include the hidden\_layer\_sizes in the screenshot.

|  |
| --- |
|  |

Explain what the hidden\_layer\_sizes parameter means (do not over think this).

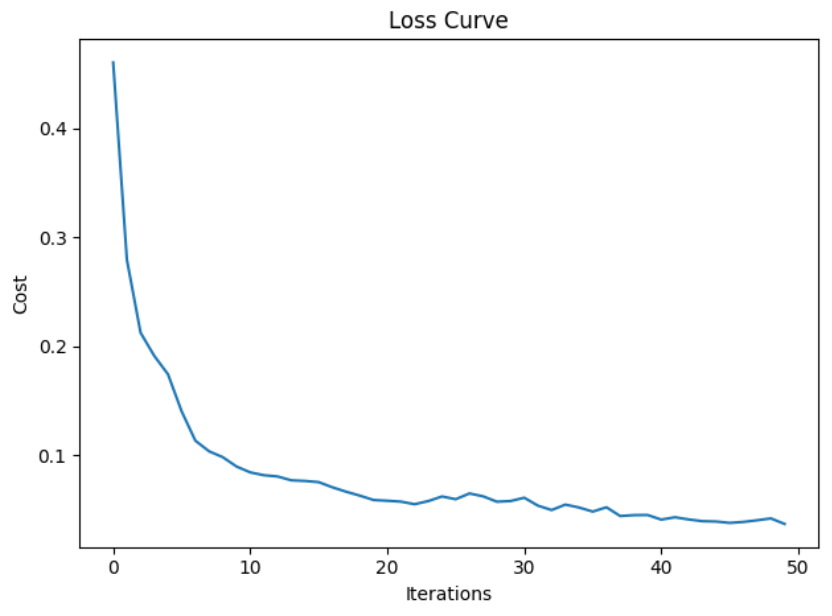
|  |
| --- |
|  |

### MLPRegressor

The MLPRegressor is the regressor version of the simplified MLP family from scikit-learn.

Example : MLP Regressor

This code fits a model for predicting house prices and evaluates a default MLPRegressor object without specifying any parameters. The results are quite good for the data set provided.

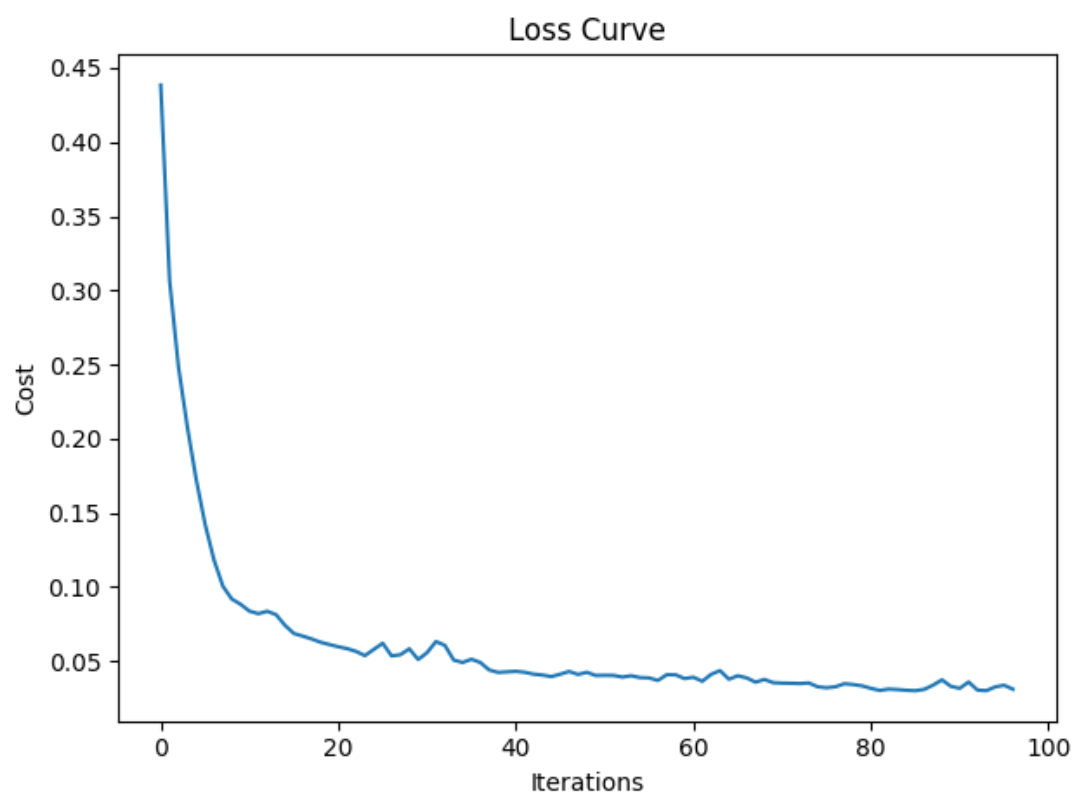


RMSE: 3.8574

|  |
| --- |
| import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  from sklearn.neural\_network import MLPRegressor  from sklearn import metrics  import warnings  warnings.filterwarnings(action='once')  PATH = "/users/pm/desktop/daydocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(PATH + CSV\_DATA, header=None)  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  print(df.head())  print(df.tail())  print(df.describe())  dataset = df.values  # split into input (X) and output (Y) variables  X = dataset[:,0:13]  y = dataset[:,13]  trainX, temp\_X, trainY, temp\_y = train\_test\_split(X, y, train\_size = 0.7)  valX, testX, valY, testY = train\_test\_split(temp\_X, temp\_y, train\_size = 0.5)  # Scale X and Y.  scX = StandardScaler()  scalerX = scX.fit(trainX)  trainX\_scaled = scalerX.transform(trainX)  valX\_scaled = scalerX.transform(valX)  testX\_scaled = scalerX.transform(testX)  scY=StandardScaler()  trainY\_scaled = scY.fit\_transform(np.array(trainY).reshape(-1,1))  testY\_scaled = scY.transform(np.array(testY).reshape(-1,1))  valY\_scaled = scY.transform(np.array(valY).reshape(-1,1))  # Build basic multilayer perceptron.  model1 = MLPRegressor(  # 3 hidden layers with 150 neurons, 100, and 50.  hidden\_layer\_sizes = (150,100,50),  max\_iter = 50, # epochs  activation = 'relu',  solver = 'adam', # optimizer  verbose=1)  model1.fit(trainX\_scaled, trainY\_scaled)  def showLosses(model):  plt.plot(model.loss\_curve\_)  plt.title("Loss Curve")  plt.xlabel('Iterations')  plt.ylabel('Cost')  plt.show()  def evaluateModel(model, testX\_scaled, testY\_scaled, scY):  showLosses(model)  scaledPredictions = model.predict(testX\_scaled)  y\_pred = scY.inverse\_transform(  np.array(scaledPredictions).reshape(-1,1))  mse = metrics.mean\_squared\_error(testY\_scaled, y\_pred)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))  evaluateModel(model1, valX\_scaled, valY\_scaled, scY)  # here is the new part.  param\_grid = {  'hidden\_layer\_sizes': [(150,100,50), (120,80,40), (100,50,30)],  'max\_iter': [50, 100],  'activation': ['tanh', 'relu'],  'solver': ['sgd', 'adam'],  'alpha': [0.0001, 0.05],  'learning\_rate': ['constant','adaptive'],  } |

Example : Grid Searching the MLP Regressor

This example shows how to grid search the MLP Regressor parameters of the model from Example 7. It turns out that the base model tends to do a good job of fitting a neural network.



Sometimes, but not always, the grid search will find a better fit.

RMSE: 3.268

To build this example, add the following code to the end of Example 7.

|  |
| --- |
| # here is the new part.  param\_grid = {  'hidden\_layer\_sizes': [(150,100,50), (120,80,40), (100,50,30)],  'max\_iter': [50, 100],  'activation': ['tanh', 'relu'],  'solver': ['sgd', 'adam'],  'alpha': [0.0001, 0.05],  'learning\_rate': ['constant','adaptive'],  }  from sklearn.model\_selection import GridSearchCV  # n\_jobs=-1 means use all processors.  # Run print(metrics.get\_scorer\_names()) for scoring choices.  model2 = MLPRegressor()  gridModel = GridSearchCV(model2, param\_grid, n\_jobs= -1, cv=10,  scoring='neg\_mean\_squared\_error')  gridModel.fit(trainX\_scaled, trainY\_scaled)  print("Best parameters")  print(gridModel.best\_params\_)  evaluateModel(gridModel.best\_estimator\_, valX\_scaled, valY\_scaled, scY)  # Evaluate both models with test (unseen) data.  print("\n\*\*\* Base model with test data: ")  evaluateModel(model1, testX\_scaled, testY\_scaled, scY)  print(model1.get\_params())  print("\n\*\*\* Grid searched model with test data: ")  evaluateModel(gridModel.best\_estimator\_, testX\_scaled, testY\_scaled, scY)  print(gridModel.get\_params()) |

Exercise (2 marks)

Advanced parameter settings for learning rate provide algorithms for slowing down the learning rate as the model approaches an optimum. The scikit-learn model enables an adjusting learning rate with the ‘adaptive’ setting.

|  |
| --- |
| 'learning\_rate': ['constant','adaptive'], |

Starting with Example 8, add an option to the grid search to include a range of starting learning rates. Then change the grid search so it uses a random search instead.

|  |
| --- |
| 'learning\_rate\_init': [0.0001, 0.001, 0.005, 0.01, 0.1, 0.2] |

Show your code here after making the changes:

|  |
| --- |
|  |

Run the random search after making the changes. Show a screenshot of the best model here.

|  |
| --- |
|  |