Table of Contents

[Timing Algorithms 1](#_Toc62034904)

[GridSearchCV 3](#_Toc62034905)

[RandomSearchCV 5](#_Toc62034906)

[Visualizing Loss and Accuracy for a Classification Model 6](#_Toc62034907)

[Viewing Losses for Classification and Linear Models 10](#_Toc62034908)

## Timing Algorithms

By now you likely have noticed how computationally expensive neural network algorithms are. Timing is a significant factor now when evaluating the feasibility of a model. Because of potential time constraints, you increasingly will want to consider the execution time of your algorithms as a limiting factor.

Example : Timing Functions

This example shows a simple way to record the execution time of a function. This code can easily be modified as each case requires.

|  |
| --- |
| import time  def currentTimeInMilliseconds():  newTime = int(round(time.time() \* 1000))  return newTime  def doSomething():  for i in range(0,10000):  print(i)  startTime = currentTimeInMilliseconds()  doSomething()  endTime = currentTimeInMilliseconds()  time\_diff = endTime - startTime  print("Time taken is: " + str(time\_diff) + " ms") |

The output from this example is:

|  |
| --- |
| Time taken is: 70 ms |

Exercise (1 mark)

Add a second calling instruction for the doSomething() function. Pad the second calling instruction with code to enable additional logging and output of execution times. Show your revised program here.

|  |
| --- |
| *import* time  def currentTimeInMilliseconds():      newTime = int(round(time.time() \* 1000))  *return* newTime  def doSomething():      time\_set = currentTimeInMilliseconds()  *for* i *in* range(0,10000):          print(i)      time\_finished = currentTimeInMilliseconds()    *return* f"Total Time: {time\_finished - time\_set}"  print(doSomething()) |

Example 1 offers a simple and flexible approach to log execution times for code routines. Sometimes it will provide exactly what you need. You can easily customize this code as each case requires. A disadvantage with the timing routine in Example 1 is that the extra timing code adds clutter.

Example : Clutter-Free Time Logging

This example shows a cleaner alternative to logging execution times in larger applications. In this example, a timeit() method allows us to implement a @timeit decorator. The @timeit decorator can be placed automatically above any function to record its execution time. A showTimeResults() function allows us to display the execution times of each function once they have executed.

|  |
| --- |
| import time  import pandas as pd  #-------------------------------------------------------------------  # Logs method execution times to a data frame.  #-------------------------------------------------------------------  timeItList = []  def showTimeResults():  timeItDf = pd.DataFrame()  for i in range(0, len(timeItList)):  timeItDf = timeItDf.append(timeItList[i], ignore\_index=True)  print("\n")  print(timeItDf)  return timeItDf  #-------------------------------------------------------------------  # Enables @timeit decorator.  #-------------------------------------------------------------------  def timeit(method):  def timed(\*args, \*\*kw):  ts = time.time()  result = method(\*args, \*\*kw)  te = time.time()  if 'log\_time' in kw:  name = kw.get('log\_name', method.\_\_name\_\_.upper())  kw['log\_time'][name] = int((te - ts) \* 1000)  else:  diff = (te - ts) \* 1000  timeItList.append({ 'Method':method.\_\_name\_\_, 'Time (ms)':diff})  print('%r %2.2f ms' % (method.\_\_name\_\_, diff))  return result  return timed  @timeit # Prefix any function with this code to automatically log execution time.  def doSomething():  for i in range(0,1000):  print(i)  for x in range(0,3):  doSomething()  showTimeResults() |

Exercise (2 marks)

Add a new method called doSomethingElse(), in addition to doSomething(). In doSomethingElse(), print out your first name 100 times. Call your new function. Use the @timeit decorator with the function to show how long it takes to run your new function. Show your revised program here:

|  |
| --- |
| *import* time  *import* pandas *as* pd  *#-------------------------------------------------------------------*  *# Logs method execution times to a data frame.*  *#-------------------------------------------------------------------*  timeItList = []  def showTimeResults():      timeItDf = pd.DataFrame()  *for* i *in* range(0, len(timeItList)):          timeItDf = timeItDf.append(timeItList[i], *ignore\_index*=True)      print("\n")      print(timeItDf)  *return* timeItDf  *#-------------------------------------------------------------------*  *# Enables @timeit decorator.*  *#-------------------------------------------------------------------*  def timeit(*method*):      def timed(\**args*, \*\**kw*):          ts     = time.time()          result = method(\*args, \*\*kw)          te     = time.time()  *if* 'log\_time' *in* kw:              name = kw.get('log\_name', method.\_\_name\_\_.upper())              kw['log\_time'][name] = int((te - ts) \* 1000)  *else*:              diff = (te - ts) \* 1000              timeItList.append({ 'Method':method.\_\_name\_\_, 'Time (ms)':diff})              print('%r  %2.2f ms' % (method.\_\_name\_\_, diff))  *return* result  *return* timed  @timeit *# Prefix any function with this code to automatically log execution time.*  def doSomething():  *for* i *in* range(0,1000):          print(i)    @timeit  def doSomethingElse():  *for* i *in* range(0, 100):          print("Sean")  *for* x *in* range(0,3):      doSomething()      doSomethingElse()  showTimeResults() |

Show your DataFrame output after adding doSomethingElse(). The output must show execution times for each time the doSomething() and doSomethingElse() methods are invoked.

|  |
| --- |
| This program ran in 3 loops |

## GridSearchCV

You may have noticed that performing the GridSearch last week was exhausting in terms of time.

Example : GridSearch

Last week this code was used to perform a GridSearch to determine the best parameters for adding a new layer. The process may take between 20 minutes to an hour.

|  |
| --- |
| import pandas as pd  import tensorflow  from sklearn.model\_selection import train\_test\_split  PATH = "C:\\datasets\\"  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\_test, y\_train, y\_test = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.2, 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(lr=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

If you are short on 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.

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) |

Example : Using RandomSearchCV() and Timing While Tuning

This example shows how to implement a random search and time recording while fitting the neural network that is presented in Example 3. Unfortunately, the @timeit decorator does not work GridSearchCV() function so a more basic approach is used to calculate and display the execute time duration.

To enable the random search, starting with Example 3, replace the following line:

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

with this code:

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

Next, to record the execution time, replace the instruction:

|  |
| --- |
| grid\_result = grid.fit(X\_train, y\_train) |

with this basic set of instructions to record the time:

|  |
| --- |
| import time  def currentTimeInMilliseconds():  newTime = int(round(time.time() \* 1000))  return newTime  startTime = currentTimeInMilliseconds()  grid\_result = grid.fit(X\_train, y\_train)  endTime = currentTimeInMilliseconds()  time\_diff = endTime - startTime  print("The time to perform the search is: " + str(time\_diff) + " ms.") |

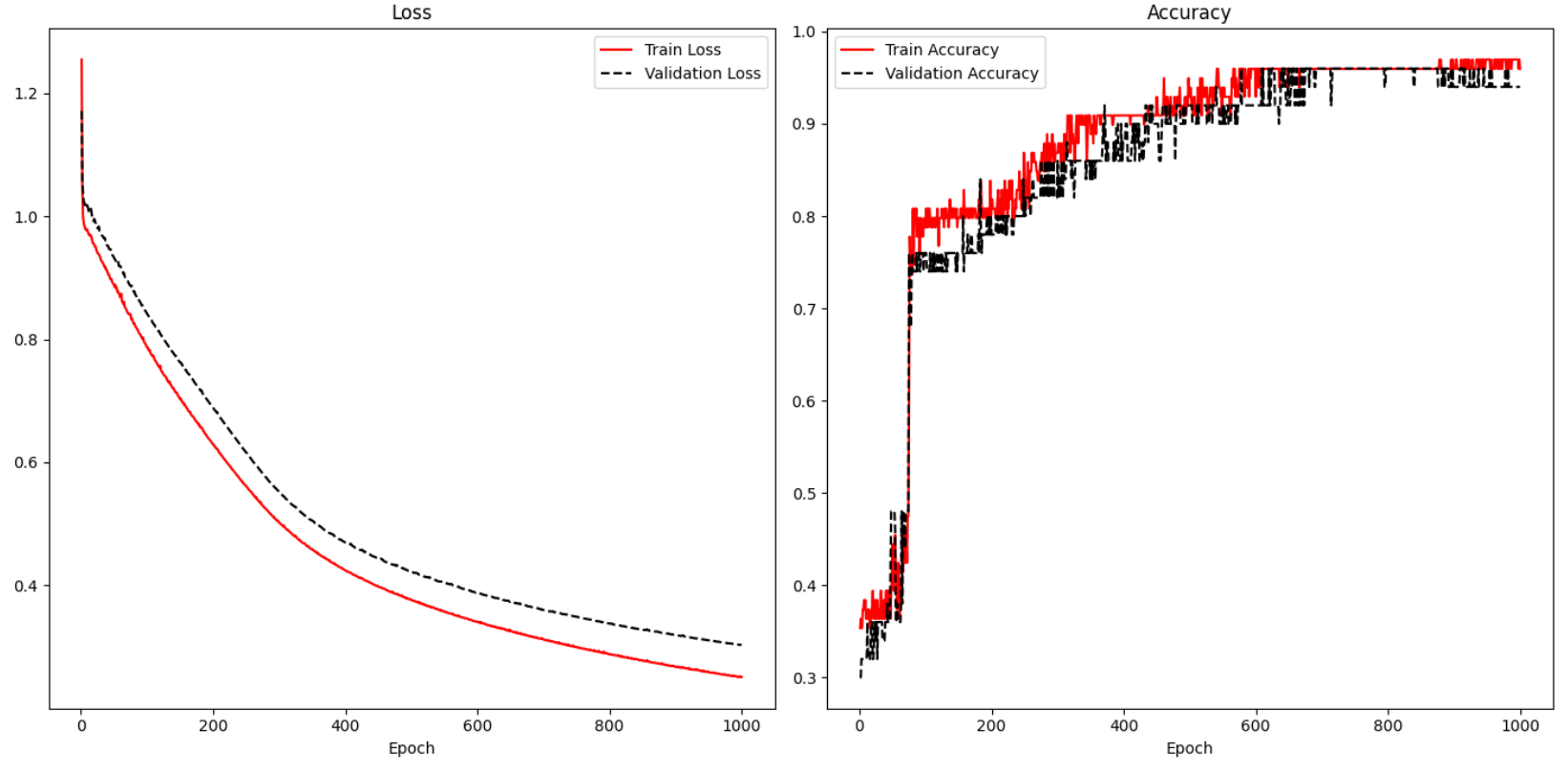
The output confirms that approximately 83 seconds were taken to perform a random search. A grid search likely would find a better set of parameters but the random search is faster.

|  |
| --- |
| The time to perform the search is: 82835 ms.  Best: -19.642832 using {'numNeurons': 15, 'initializer': 'zero', 'activation': 'linear'}  -85.678594 (15.195062) with: {'numNeurons': 30, 'initializer': 'glorot\_normal', 'activation': 'hard\_sigmoid'}  -22.580643 (4.320906) with: {'numNeurons': 10, 'initializer': 'uniform', 'activation': 'softplus'}  -86.138295 (15.683223) with: {'numNeurons': 25, 'initializer': 'uniform', 'activation': 'tanh'}  -19.642832 (2.034633) with: {'numNeurons': 15, 'initializer': 'zero', 'activation': 'linear'}  … |

## 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 test results, validation\_data arguments must be supplied in the model.fit() instruction. 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\_test, y\_test)) |

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 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.

|  |
| --- |
| 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 and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  x\_arrayReshaped, y\_arrayReshaped, test\_size=0.33)  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\_test, y\_test))  # Evaluate the model.  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() |

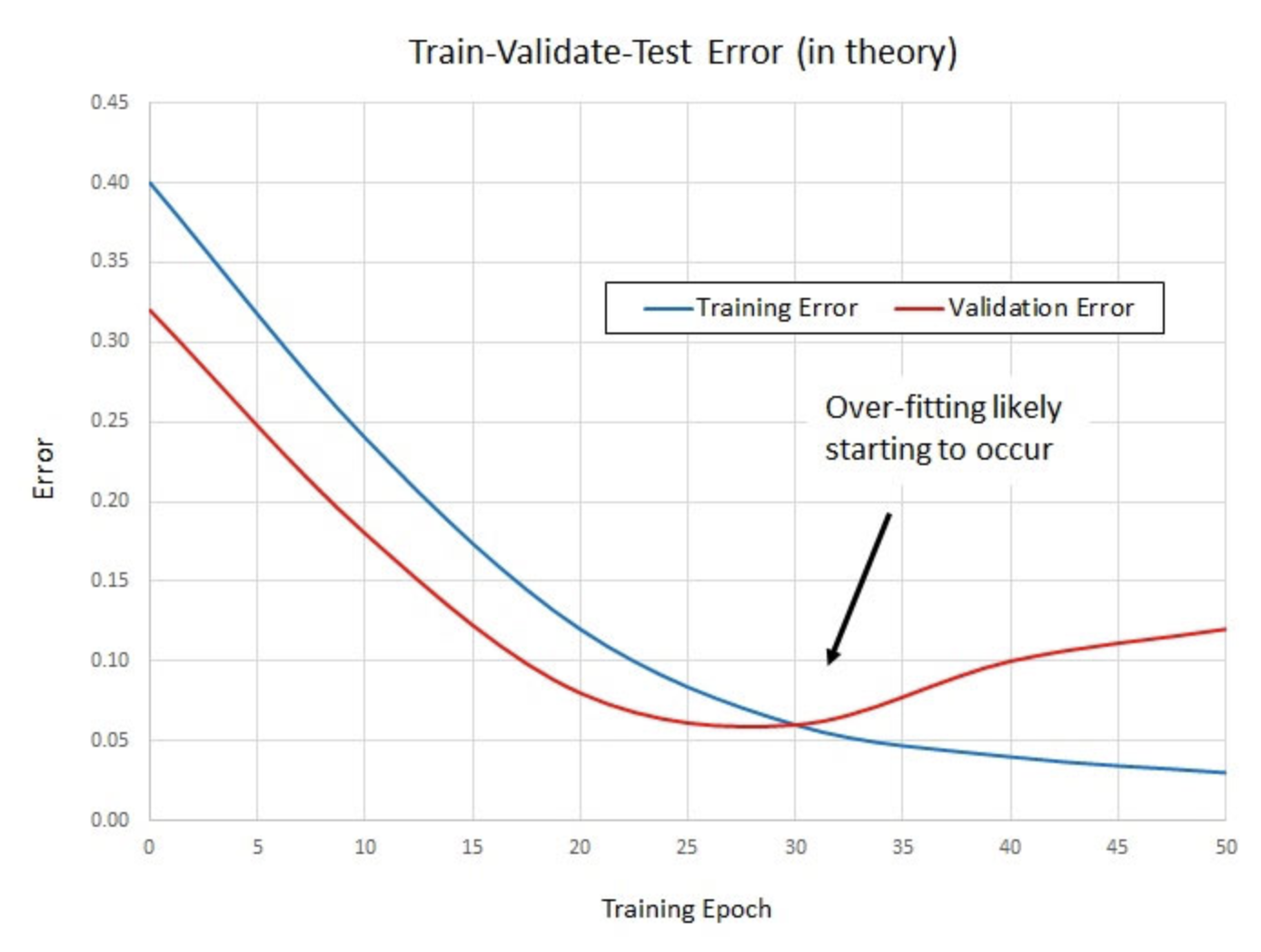
## Viewing Losses for Classification and Linear Models

Since linear models use measures such as RMSE to judge accuracy, 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 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)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(x\_arrayReshaped,  y\_arrayReshaped, test\_size=0.2, 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\_test, y\_test))  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 6.

|  |
| --- |
| linear models use RMSE as a judge of accuracy so there is no need to draw an accuracy plot. We can just look at the printed RMSE value. |

Exercise (4 marks)

Add code to time the duration needed to fit the model. You may want to implement the routine discussed in Example 1. Show your revised program here:

|  |
| --- |
| *import* pandas *as* pd  *import* numpy  *as* np  *from* sklearn                 *import* metrics  *from* sklearn.model\_selection *import* train\_test\_split  *import* time  PATH     = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(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\_test, y\_train, y\_test = train\_test\_split(x\_arrayReshaped,                       y\_arrayReshaped, *test\_size*=0.2, *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 timeit(*method*):      def timed(\**args*, \*\**kw*):          ts     = time.time()          result = method(\*args, \*\*kw)          te     = time.time()  *if* 'log\_time' *in* kw:              name = kw.get('log\_name', method.\_\_name\_\_.upper())              kw['log\_time'][name] = int((te - ts) \* 1000)  *else*:              diff = (te - ts) \* 1000              timeItList.append({ 'Method':method.\_\_name\_\_, 'Time (ms)':diff})              print('%r  %2.2f ms' % (method.\_\_name\_\_, diff))  *return* result  *return* timed  *# #-------------------------------------------------------------------*  *# # Logs method execution times to a data frame.*  *# #-------------------------------------------------------------------*  timeItList = []  def showTimeResults():      timeItDf = pd.DataFrame()  *for* i *in* range(0, len(timeItList)):          timeItDf = timeItDf.append(timeItList[i], *ignore\_index*=True)      print("\n")      print(timeItDf)  *return* timeItDf  @timeit  def fitModel(*X\_train*, *y\_train*):      model.fit(X\_train, y\_train, *epochs*=100,  *batch\_size*=9, *verbose*=1,  *validation\_data*=(X\_test, y\_test))    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 = fitModel(X\_train, y\_train)  showTimeResults()  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))) |

Show the time duration here:

|  |
| --- |
|  |

Exercise (6 marks)

Plot the loss function output that occurs while the model in Example 6 is fit. Show your revised program here. Refer to Example 5 for code that plots the loss function.

|  |
| --- |
| *import* pandas *as* pd  *import* numpy  *as* np  *from* sklearn                 *import* metrics  *from* sklearn.model\_selection *import* train\_test\_split  *import* time  PATH     = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "housing.data"  df = pd.read\_csv(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\_test, y\_train, y\_test = train\_test\_split(x\_arrayReshaped,                       y\_arrayReshaped, *test\_size*=0.2, *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*  *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')  def timeit(*method*):      def timed(\**args*, \*\**kw*):          ts     = time.time()          result = method(\*args, \*\*kw)          te     = time.time()  *if* 'log\_time' *in* kw:              name = kw.get('log\_name', method.\_\_name\_\_.upper())              kw['log\_time'][name] = int((te - ts) \* 1000)  *else*:              diff = (te - ts) \* 1000              timeItList.append({ 'Method':method.\_\_name\_\_, 'Time (ms)':diff})              print('%r  %2.2f ms' % (method.\_\_name\_\_, diff))  *return* result  *return* timed  *# #-------------------------------------------------------------------*  *# # Logs method execution times to a data frame.*  *# #-------------------------------------------------------------------*  timeItList = []  def showTimeResults():      timeItDf = pd.DataFrame()  *for* i *in* range(0, len(timeItList)):          timeItDf = timeItDf.append(timeItList[i], *ignore\_index*=True)      print("\n")      print(timeItDf)  *return* timeItDf  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', *metrics*=['accuracy'])  *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\_test, y\_test))  loss, acc = model.evaluate(X\_test, y\_test, *verbose*=0)  print('Test Accuracy: %.3f' % acc)  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)))    plt.subplots(*nrows*=1, *ncols*=2,  *figsize*=(14,7))  showLoss(history)  showAccuracy(history)  plt.show() |

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.

|  |
| --- |
| Looking at the loss function, I would say that it shows a reasonably good fit. The validation curve doesn’t depart from the train loss like it does in overfitting.  Paper Review: A Disciplined Approach To Neural Network Hyper-Parameters:  Part I - Learning Rate, Batch Size, Momentum, And Weight Decay |