Table of Contents

[More Information About Skorch 1](#_Toc127877268)

[Linear Regression Model 2](#_Toc127877269)

[Grid and Random Searching Linear Regressions 5](#_Toc127877270)

[Creating a Parameter Grid 5](#_Toc127877271)

[Skorch for Handwriting Classification 9](#_Toc127877272)

## More Information About Skorch

A fair amount of information for this section about skorch is taken from:

<https://skorch.readthedocs.io/en/stable/user/neuralnet.html>

**NeuralNet** and the derived classes enable *skorch*. These classes wrap a custom PyTorch Module while providing an interface that is familiar for sklearn users. In particular, the interface allows us to use GridSearchCV and RandomizedSearchCV. Last day **NeuralNetClassifier** from skorch was used. Today, **NeuralNetRegressor** from skorch will be used.

Once the PyTorch Module is defined, it can then be passed to NeuralNet, in conjunction with a PyTorch criterion. Finally, you can call fit() and predict(), as with an sklearn estimator. The finished code could look something like this:

|  |
| --- |
| class MyModule(torch.nn.Module):  ...  net = NeuralNet(  module=MyModule,  criterion=torch.nn.NLLLoss,  )  net.fit(X, y)  y\_pred = net.predict(X\_valid) |

skorch:

* wraps the PyTorch Module in an sklearn interface
* converts numpy.ndarrays to PyTorch Tensors if necessary
* abstracts away the fit loop
* takes care of batching the data

You therefore have a lot less boilerplate code, letting you focus on what matters. At the same time, skorch is very flexible and can be extended with ease, getting out of your way as much as possible.

## Linear Regression Model

Example : Linear Regression

This example demonstrates how to create a neural network in PyTorch for linear regression with

skorch. The detail for how the PyTorch code works is in the comments. Exercises later will offer hands-on practice for a better understanding

The model gives quite different results from one run to the next. Part of the problem is that there is not much data so each training session uses a very different subset of data when the data is randomized.

|  |  |  |
| --- | --- | --- |
| RMSE: 43.60546 | RMSE: 88.813866 | RMSE: 32.51236 |
|  |  |  |

Notice that the networks parameter dictionary is passed to the object during instantiation with the \*\*params parameter. The forward() function the network receives the dictionary with the \*\*kwargs parameter.

Here is the code:

|  |
| --- |
| from sklearn.datasets import make\_regression  from skorch import NeuralNetRegressor  import torch.nn as nn  import numpy as np  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  import torch.nn.functional as F  from torch import optim  # This class could be any name.  # nn.Module is needed to enable grid searching of parameters  # with skorch later.  from skorch.callbacks import EarlyStopping  class MyNeuralNet(nn.Module):  def \_\_init\_\_(self, num\_units=10, nonlin=F.relu):  super(MyNeuralNet, self).\_\_init\_\_()  self.num\_units = num\_units  self.nonlin = nonlin  self.dense0 = nn.Linear(4, num\_units)  self.nonlin = nonlin  self.dense1 = nn.Linear(num\_units, 10)  self.output = nn.Linear(10, 1)  def forward(self, X, \*\*kwargs):  X = self.nonlin(self.dense0(X))  X = F.relu(self.dense1(X))  X = self.output(X)  return X  def buildModel(x, y):  # Trains the Neural Network with fixed hyperparameters  # The Neural Net is initialized with fixed hyperparameters  myNetwork = MyNeuralNet(num\_units=10)    # Define learning rate, max\_epochs and momentum  # separately from the network.  params = {  'lr': 0.001, # Learning rate  'max\_epochs': 1000, # Maximum number of epochs  'optimizer':optim.SGD,  'optimizer\_\_momentum': 0.9,  }  net\_regr = NeuralNetRegressor(  myNetwork,  \*\*params,  callbacks=[EarlyStopping(patience=60)],  )  model = net\_regr.fit(x, y)  return model  def evaluateModel(model, X\_test, y\_test, scalerY):  print(model)  y\_pred\_scaled = model.predict(X\_test)  y\_pred = scalerY.inverse\_transform(y\_pred\_scaled)  mse = mean\_squared\_error(y\_test, y\_pred)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))  # Prep the data.  # This is a toy dataset for regression, 1000 data points with 20 features each  import torch  import pandas as pd  df = pd.read\_csv('/users/pm/desktop/daydocs/data/petrol\_consumption.csv')  X = df.copy()  del X['Petrol\_Consumption']  y = df['Petrol\_Consumption']  X\_train, X\_test, y\_train, y\_test =\  train\_test\_split(X, y, test\_size=0.2)  from sklearn.preprocessing import StandardScaler  scalerX = StandardScaler()  scaledXTrain = scalerX.fit\_transform(X\_train)  scaledXTest = scalerX.transform(X\_test)  scalerY = StandardScaler()  scaledYTrain = scalerY.fit\_transform(np.array(y\_train).reshape(-1,1))  # Must convert the data to PyTorch tensors  X\_train\_tensor = torch.tensor(scaledXTrain, dtype=torch.float32)  X\_test\_tensor = torch.tensor(scaledXTest, dtype=torch.float32)  y\_train\_tensor = torch.tensor(list(scaledYTrain), dtype=torch.float32)  y\_test\_tensor = torch.tensor(list(y\_test), dtype=torch.float32)  # Build the model.  model = buildModel(X\_train\_tensor, y\_train\_tensor)  print(model.get\_params())  # Evaluate the model.  evaluateModel(model, X\_test\_tensor, y\_test\_tensor, scalerY)  import matplotlib.pyplot as plt  def drawLossPlot(net):  plt.plot(net.history[:, 'train\_loss'], color='blue', label='train')  plt.plot(net.history[:, 'valid\_loss'], color='orange', label='val')  plt.legend()  plt.show()  drawLossPlot(model) |

Exercise (4 marks)

Create an additional hidden layer for the network in Example 1.

Show a screenshot of your model which displays the different layers of the network. However include your name and be sure that there is an extra later in the network.



Show your screenshot here:

|  |
| --- |
|  |

Exercise (4 marks)

Use the USA\_Housing.csv data set to build a PyTorch network with EarlyStopping to predict Price. Use all available features except ‘Address’. Show your loss curve and RMSE here.

|  |
| --- |
| RMSE: 100926.91 |

Show the code as text for your complete program here:

|  |
| --- |
| from sklearn.datasets import make\_regression from skorch import NeuralNetRegressor import torch.nn as nn import numpy as np from sklearn.model\_selection import train\_test\_split from sklearn.metrics import mean\_squared\_error import torch.nn.functional as F from torch import optim  # This class could be any name. # nn.Module is needed to enable grid searching of parameters # with skorch later. from skorch.callbacks import EarlyStopping   class MyNeuralNet(nn.Module):  def \_\_init\_\_(self, num\_units=10, nonlin=F.relu):  super(MyNeuralNet, self).\_\_init\_\_()  self.num\_units = num\_units  self.nonlin = nonlin   self.dense0 = nn.Linear(5, num\_units)  self.nonlin = nonlin  self.dense1 = nn.Linear(num\_units, 10)  self.output = nn.Linear(10, 1)   def forward(self, X, \*\*kwargs):  X = self.nonlin(self.dense0(X))  X = F.relu(self.dense1(X))  X = self.output(X)  return X   def buildModel(x, y):  # Trains the Neural Network with fixed hyperparameters  # The Neural Net is initialized with fixed hyperparameters  myNetwork = MyNeuralNet(num\_units=128)   # Define learning rate, max\_epochs and momentum  # separately from the network.  params = {  'lr': 0.001, # Learning rate  'max\_epochs': 1000, # Maximum number of epochs  'optimizer': optim.SGD,  'optimizer\_\_momentum': 0.9,  }   net\_regr = NeuralNetRegressor(  myNetwork,  \*\*params,  callbacks=[EarlyStopping(patience=60)],  )  model = net\_regr.fit(x, y)  return model   def evaluateModel(model, X\_test, y\_test, scalerY):  print(model)  y\_pred\_scaled = model.predict(X\_test)  y\_pred = scalerY.inverse\_transform(y\_pred\_scaled)  mse = mean\_squared\_error(y\_test, y\_pred)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))   # Prep the data. # This is a toy dataset for regression, 1000 data points with 20 features each import torch import pandas as pd from pathlib import Path PATH = Path("C:/datasets/USA\_Housing.csv")  df = pd.read\_csv(PATH) # Show all columns. pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line. pd.set\_option('display.width', 250)  print(df.head()) X = df.copy() del X['Price'] del X['Address'] y = df['Price']  X\_train, X\_test, y\_train, y\_test = \  train\_test\_split(X, y, test\_size=0.2)  from sklearn.preprocessing import StandardScaler  scalerX = StandardScaler() scaledXTrain = scalerX.fit\_transform(X\_train) scaledXTest = scalerX.transform(X\_test)  scalerY = StandardScaler() scaledYTrain = scalerY.fit\_transform(np.array(y\_train).reshape(-1, 1))  # Must convert the data to PyTorch tensors X\_train\_tensor = torch.tensor(scaledXTrain, dtype=torch.float32) X\_test\_tensor = torch.tensor(scaledXTest, dtype=torch.float32) y\_train\_tensor = torch.tensor(list(scaledYTrain), dtype=torch.float32) y\_test\_tensor = torch.tensor(list(y\_test), dtype=torch.float32)  # Build the model. model = buildModel(X\_train\_tensor, y\_train\_tensor) print(model.get\_params()) # Evaluate the model. evaluateModel(model, X\_test\_tensor, y\_test\_tensor, scalerY)  import matplotlib.pyplot as plt   def drawLossPlot(net):  plt.plot(net.history[:, 'train\_loss'], color='blue', label='train')  plt.plot(net.history[:, 'valid\_loss'], color='orange', label='val')  plt.legend()  plt.show()   drawLossPlot(model) |

## Grid and Random Searching Linear Regressions

Sklearns’s GridSearchCV class can be used to grid search PyTorch network parameters. As you may have guessed, to implement a random search on a subset of options RandomizedSearchCV can also be used to save time.

### Creating a Parameter Grid

When searching parameters, options are stored in a dictionary:

|  |
| --- |
| params = {  'nn\_\_max\_epochs': [30,50,60],  'nn\_\_lr': [0.01, 0.015, 0.007],  'nn\_\_module\_\_num\_neurons': [15,20,25],  'nn\_\_optimizer': [optim.Adam, optim.SGD, optim.RMSprop]} |

An odd difference between GridSearchCV and RandomizedSearchCV is that **GridSearchCV** refers to the parameter dictionary with the **param\_grid** attribute:

|  |
| --- |
| gs = GridSearchCV(pipeline, param\_grid=params, refit=True, cv=3,  scoring='neg\_mean\_squared\_error', verbose=1) |

**RandomizedSearchCV** refers to the parameter dictionary with the **param\_distributions** attribute:

|  |
| --- |
| gs = RandomizedSearchCV(pipeline, param\_distributions=params, refit=True, cv=3,  scoring='neg\_mean\_squared\_error', verbose=1) |

To enable the grid search with ***skorch*** I am using nn.Module to organize the network. You will notice where the search parameters are defined that ***num\_neurons*** and ***dropout*** are prefixed with ***nn\_\_module\_\_***.

'nn\_\_module\_\_num\_neurons': [5, 10],

'nn\_\_module\_\_dropout': [0.1, 0.5],

These nn\_\_module\_\_ parameters can be passed into the constructor to enable their grid search. You can pass whatever variables you want to the constructor of the neural network as long as they are prefixed with **nn\_\_module\_\_**.

class MyNeuralNet(nn.Module):

# Define network objects.

# Defaults are set for number of neurons and the

# dropout rate.

def \_\_init\_\_(self, num\_neurons=10, dropout=0.1):

super(MyNeuralNet, self).\_\_init\_\_()

You will also notice other parameters in the grid search that are not prefixed with **nn\_\_module\_\_**. These are special case variables which do not need to be passed to the constructor of the network.

'nn\_\_max\_epochs': [10, 20],

'nn\_\_lr': [0.1, 0.01],

'nn\_\_optimizer': [optim.Adam, optim.SGD, optim.RMSprop]

A pipeline is used to assemble the scaling and network objects since GridSearchCV needs to work with a Pipeline object. Also note that the network class is not directly instantiated when referenced in the NeuralNetClassifier. However, the parameters for instantiation of the network are passed in the GridSearchCV function.

nn = NeuralNetClassifier(MyNeuralNet, verbose=0, train\_split=False)

pipeline = Pipeline([('scale', StandardScaler()), ('nn', nn)])

params = {

'nn\_\_max\_epochs': [10, 20],

'nn\_\_lr': [0.1, 0.01],

'nn\_\_module\_\_num\_neurons': [5, 10],

'nn\_\_module\_\_dropout': [0.1, 0.5],

'nn\_\_optimizer': [optim.Adam, optim.SGD, optim.RMSprop]}

# The grid search module is instantiated

gs = GridSearchCV(pipeline, params, refit=True, cv=3,

scoring='balanced\_accuracy', verbose=1)

Example : Grid Search

This example shows the full code needed to grid search multiple parameters. These are the model parameters that were chosen for the best estimator:

|  |
| --- |
| Best parameters:  {'nn\_\_lr': 0.01, 'nn\_\_max\_epochs': 20, 'nn\_\_module\_\_dropout': 0.1,  'nn\_\_module\_\_num\_neurons': 10, 'nn\_\_optimizer': <class  'torch.optim.rmsprop.RMSprop'>} |

Here is the full code solution:

|  |
| --- |
| from sklearn.datasets import make\_regression  from sklearn.pipeline import Pipeline  from skorch import NeuralNetRegressor  import torch.nn as nn  import numpy as np  from sklearn.model\_selection import train\_test\_split, GridSearchCV  from sklearn.metrics import mean\_squared\_error  import torch.nn.functional as F  # This class could be any name.  # nn.Module is needed to enable grid searching of parameters  # with skorch later.  from torch import optim  class MyNeuralNet(nn.Module):  def \_\_init\_\_(self, num\_neurons):  super(MyNeuralNet, self).\_\_init\_\_()  self.num\_units = num\_neurons  self.dense0 = nn.Linear(4, num\_neurons)  self.dense1 = nn.Linear(num\_neurons, 10)  self.output = nn.Linear(10, 1)  def forward(self, X, \*\*kwargs):  X = F.relu(self.dense0(X))  X = F.relu(self.dense1(X))  X = self.output(X)  return X  def buildModel(x, y):  nn = NeuralNetRegressor(MyNeuralNet, verbose=1, train\_split=False)  # Trains the Neural Network with fixed hyperparameters  pipeline = Pipeline([ ('nn', nn)])  params = {  'nn\_\_max\_epochs': [30,50,60],  'nn\_\_lr': [0.01, 0.015, 0.007],  'nn\_\_module\_\_num\_neurons': [15,20,25],  'nn\_\_optimizer': [optim.Adam, optim.SGD, optim.RMSprop]}  # The grid search module is instantiated  gs = GridSearchCV(pipeline, param\_grid=params, refit=True, cv=3,  scoring='neg\_mean\_squared\_error', verbose=1)  return gs.fit(x, y)  def evaluateModel(model, X\_test, y\_test, scalerY):  print(model)  y\_pred\_scaled = model.predict(X\_test)  y\_pred = scalerY.inverse\_transform(y\_pred\_scaled)  mse = mean\_squared\_error(y\_test, y\_pred)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))  # Prep the data.  # This is a toy dataset for regression, 1000 data points with 20 features each  import torch  import pandas as pd  df = pd.read\_csv('/users/pm/desktop/daydocs/data/petrol\_consumption.csv')  X = df.copy()  del X['Petrol\_Consumption']  y = df['Petrol\_Consumption']  X\_train, X\_test, y\_train, y\_test =\  train\_test\_split(X, y, test\_size=0.2)  from sklearn.preprocessing import StandardScaler  scalerX = StandardScaler()  scaledXTrain = scalerX.fit\_transform(X\_train)  scaledXTest = scalerX.transform(X\_test)  scalerY = StandardScaler()  scaledYTrain = scalerY.fit\_transform(np.array(y\_train).reshape(-1,1))  # Must convert the data to PyTorch tensors  X\_train\_tensor = torch.tensor(scaledXTrain, dtype=torch.float32)  X\_test\_tensor = torch.tensor(scaledXTest, dtype=torch.float32)  y\_train\_tensor = torch.tensor(list(scaledYTrain), dtype=torch.float32)  y\_test\_tensor = torch.tensor(list(y\_test), dtype=torch.float32)  # Build the model.  model = buildModel(X\_train\_tensor, y\_train\_tensor)  print("Best parameters:")  print(model.best\_params\_)  # Evaluate the model.  evaluateModel(model.best\_estimator\_, X\_test\_tensor, y\_test\_tensor, scalerY) |

Exercise (5 Marks)

Grid search for the **USA\_Housing.csv** file. Show your code. You can use RandomizedSearchCV if you want. Note that the parameter dictionary attribute is **param\_grid** forGridSearchCV and **param\_distributions** forRandomizedSearchCV.

Display a screenshot of the recommended parameters here:

|  |
| --- |
|  |

Show your code as text here:

|  |
| --- |
| from sklearn.datasets import make\_regression from sklearn.pipeline import Pipeline from skorch import NeuralNetRegressor import torch.nn as nn import numpy as np from sklearn.model\_selection import train\_test\_split, GridSearchCV from sklearn.metrics import mean\_squared\_error import torch.nn.functional as F  # This class could be any name. # nn.Module is needed to enable grid searching of parameters # with skorch later. from torch import optim  class MyNeuralNet(nn.Module):  def \_\_init\_\_(self, num\_neurons):  super(MyNeuralNet, self).\_\_init\_\_()   self.num\_units = num\_neurons  self.dense0 = nn.Linear(4, num\_neurons)  self.dense1 = nn.Linear(num\_neurons, 10)  self.output = nn.Linear(10, 1)   def forward(self, X, \*\*kwargs):  X = F.relu(self.dense0(X))  X = F.relu(self.dense1(X))  X = self.output(X)  return X  def buildModel(x, y):  nn = NeuralNetRegressor(MyNeuralNet, verbose=1, train\_split=False)  # Trains the Neural Network with fixed hyperparameters  pipeline = Pipeline([ ('nn', nn)])   params = {  'nn\_\_max\_epochs': [30,50,60],  'nn\_\_lr': [0.01, 0.015, 0.007],  'nn\_\_module\_\_num\_neurons': [15,20,25],  'nn\_\_optimizer': [optim.Adam, optim.SGD, optim.RMSprop]}   # The grid search module is instantiated  gs = GridSearchCV(pipeline, param\_grid=params, refit=True, cv=3,  scoring='neg\_mean\_squared\_error', verbose=1)   return gs.fit(x, y)  def evaluateModel(model, X\_test, y\_test, scalerY):  print(model)  y\_pred\_scaled = model.predict(X\_test)  y\_pred = scalerY.inverse\_transform(y\_pred\_scaled)  mse = mean\_squared\_error(y\_test, y\_pred)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))  # Prep the data. # This is a toy dataset for regression, 1000 data points with 20 features each import torch import pandas as pd from pathlib import Path PATH = Path("C:/datasets/USA\_Housing.csv") df = pd.read\_csv(PATH) X = df.copy() del X['Price'] del X['Address'] y = df['Price']  X\_train, X\_test, y\_train, y\_test =\  train\_test\_split(X, y, test\_size=0.2)  from sklearn.preprocessing import StandardScaler scalerX = StandardScaler() scaledXTrain = scalerX.fit\_transform(X\_train) scaledXTest = scalerX.transform(X\_test)  scalerY = StandardScaler() scaledYTrain = scalerY.fit\_transform(np.array(y\_train).reshape(-1,1))  # Must convert the data to PyTorch tensors X\_train\_tensor = torch.tensor(scaledXTrain, dtype=torch.float32) X\_test\_tensor = torch.tensor(scaledXTest, dtype=torch.float32) y\_train\_tensor = torch.tensor(list(scaledYTrain), dtype=torch.float32) y\_test\_tensor = torch.tensor(list(y\_test), dtype=torch.float32)  # Build the model. model = buildModel(X\_train\_tensor, y\_train\_tensor)  print("Best parameters:") print(model.best\_params\_)  # Evaluate the model. evaluateModel(model.best\_estimator\_, X\_test\_tensor, y\_test\_tensor, scalerY) |

## Skorch for Handwriting Classification

<https://github.com/skorch-dev/skorch/blob/master/notebooks/MNIST.ipynb>

Example : MINST Image Classification with Skorch

This example uses a Skorch classifier network to train for identifying handwriting digits from 0 to 9.



Google Colab is recommended for this tutorial. If you are running in colab, you should install the dependencies and download the dataset by running the following cell:

|  |
| --- |
| import subprocess  # Installation on Google Colab  try:  import google.colab  subprocess.run(['python', '-m', 'pip', 'install', 'skorch' , 'torch'])  except ImportError:  pass  from sklearn.datasets import fetch\_openml  from sklearn.model\_selection import train\_test\_split  import numpy as np  import matplotlib.pyplot as plt |

Using SciKit-Learns fetch\_openml to load MNIST data.

|  |
| --- |
| mnist = fetch\_openml('mnist\_784', as\_frame=False, cache=False) |

Each image of the MNIST dataset is encoded in a 784 dimensional vector, representing a 28 x 28 pixel image. Each pixel has a value between 0 and 255, corresponding to the grey-value of a pixel. The above featch\_mldata method to load MNIST returns data and target as uint8 which we convert to float32 and int64 respectively.

|  |
| --- |
| X = mnist.data.astype('float32')  y = mnist.target.astype('int64') |

To avoid big weights that deal with the pixel values from between [0, 255], we scale X down. A commonly used range is [0, 1].

|  |
| --- |
| X /= 255.0 |

Split the data.

|  |
| --- |
| X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42) |

Plot data samples with labels to better understand the data.

|  |
| --- |
| def plot\_example(X, y):  """Plot the first 5 images and their labels in a row."""  for i, (img, y) in enumerate(zip(X[:5].reshape(5, 28, 28), y[:5])):  plt.subplot(151 + i)  plt.imshow(img)  plt.xticks([])  plt.yticks([])  plt.title(y)  plot\_example(X\_train, y\_train) |

The output demonstrates how the images and labels are aligned.



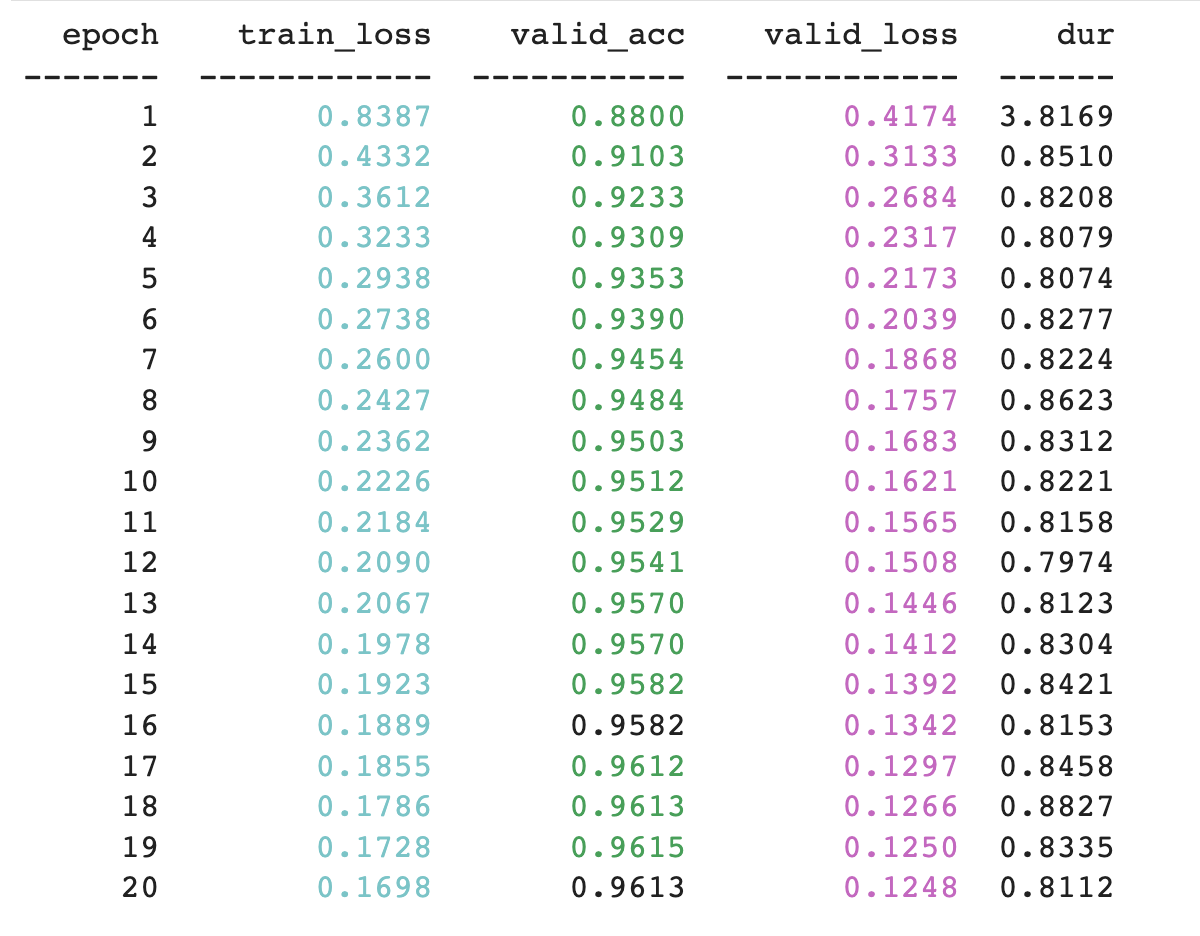
This next step builds a simple, fully connected neural network with one hidden layer. The input layer has 784 dimensions (28x28) hidden layer has 98 (= 784 / 8) and output layer 10 neurons, representing digits 0 - 9.

|  |
| --- |
| import torch  from torch import nn  import torch.nn.functional as F  device = 'cuda' if torch.cuda.is\_available() else 'cpu'  mnist\_dim = X.shape[1]  hidden\_dim = int(mnist\_dim/8)  output\_dim = len(np.unique(mnist.target))  mnist\_dim, hidden\_dim, output\_dim  # outputs: (784, 98, 10)  class ClassifierModule(nn.Module):  def \_\_init\_\_(  self,  input\_dim=mnist\_dim,  hidden\_dim=hidden\_dim,  output\_dim=output\_dim,  dropout=0.5,  ):  super(ClassifierModule, self).\_\_init\_\_()  self.dropout = nn.Dropout(dropout)  self.hidden = nn.Linear(input\_dim, hidden\_dim)  self.output = nn.Linear(hidden\_dim, output\_dim)  def forward(self, X, \*\*kwargs):  X = F.relu(self.hidden(X))  X = self.dropout(X)  X = F.softmax(self.output(X), dim=-1)  return X |

skorch allows to use PyTorch's networks in the SciKit-Learn setting:

|  |
| --- |
| from skorch import NeuralNetClassifier  torch.manual\_seed(0)  net = NeuralNetClassifier(  ClassifierModule,  max\_epochs=20,  lr=0.1,  device=device,  )  net.fit(X\_train, y\_train); |

Running the code will give the ouput:



|  |
| --- |
| from sklearn.metrics import accuracy\_score  y\_pred = net.predict(X\_test)  accuracy\_score(y\_test, y\_pred) |

Exercise (3 marks)

There is not much to do here except to verify the code example is working. Run the code and add the code below with your name in it to get the mark. The sample output should look like the following but with your name:



Show your screenshot here:

|  |
| --- |
|  |