HU, Ruoyu (rh4618)

Imperial College London

Department of Computing Academic Year **2020-2021**



Page created Fri Nov 27 23:16:04 GMT 2020

70050 mrei rh4618 v7



 ${\bf Electronic} \ \underline{\bf s} {\bf ubmission}$

Fri - 27 Nov 2020 14:24:41

rh4618

Exercise Information

Module: 70050 Introduction to Machine

Learning (Term1)

Exercise:

2 (CBC) Neural Networks Title: FAO: Rei, Marek (mrei)

Mon - 09 Nov 2020 Issued:

Fri - 27 Nov 2020 Due: **Assessment:** Group Submission: Electronic

Student Declaration - Version 7

• I declare that this final submitted version is my unaided work.

Signed: (electronic signature) Date: 2020-11-26 22:15:36

For Markers only: (circle appropriate grade)

HU, Ruoyu (rh4618) LEADER	01494690	c3	2020-11-26 22:15:36	A*	A	В	C	D	E	F
DEJL, Adam (ad5518)	01561662	c3	2020-11-26 22:16:35	A *	A	В	\mathbf{C}	D	E	\mathbf{F}
DING, Ke (kd120)	01328574	t5	2020-11-27 12:33:50	A*	A	В	\mathbf{C}	D	E	\mathbf{F}
MANGAL, Pranav (prm2418)	01487364	c3	2020-11-27 04:41:02	A *	A	В	C	D	E	F

COURSEWORK FEEDBACK FORM - Rh4618

TOTAL GRADE
94%

	GRADE	COMMENTS
Part 1	50 / 50	The submission successfully passed all the LabTS tests. The code is well-written, easily readable and efficient. Really well-done!
Implement an architecture for regression	15 / 15	A sensible regression architecture with input, hidden layers and output layer without activation function (since it is a regression task) is described in the report. The code submission successfully passed all the LabTS tests with an excellent RMSE error on the training data.
Evaluate your architecture	8/10	The preprocessing step is correct and nicely described. The NaNs are correctly filled with the mean values (an alternative would be median value since its more resilient to outliers) and one-hot encoding is used for the categorical ocean proximity. The data are correctly normalised using the MinNAwScaler. A more detailed analysis of the dataset would have shown you some highly correlated features that you could potentially combine. The RANSE as the evaluation metric and Cross-Validation as the evaluation method are correctly chosen. In the "Evaluation & Results" section, it is not clear what model architecture the experiments are conducted with. Although, the theoretical principles about dropout, regularization and early-stopping are correct, it is not clear to the reader the model / models that these findings refer to.
Fine tune your architecture	9 / 10	Generally, the hyper-parameter space is well-defined and the parameter choices are correctly justified. The step-by-step hyper-parameter space exploration is nicely done with the parameter choices to be well- justified. Instead of just stating that Adam and SGD would have no difference, some experiments could help you back this claim with empirical evidence. Excellent for including the learning curves of the final model and nicely stating future exploration paths.
Report quality	12 / 15	The report is well-structured and within the page limit. All the graphs and tables are well-presented. Some minor mistakes for example Figure 2 has no explanation about what each line represents. The three tables in the Appendix are of high importance as they contain your hyperparemeter selection exprements and they should have been in the main body of the report. In general, the to include all the important details in the main body and use the Appendix section for extra graphs, experiments, tables, proofs etc. that won't affect the reader's understanding of the discussed topic if they skip the Appendix section. Finally, some sections in the report seemed like a code walkthrough discussing the functions and methods implemented in your code which needs to be avoided. The report should be a document independent from the code where we document our approach, the steps we took to solve a problem rather than a code walkthrough.

Final Tests TestSummary.txt: 1/1 :c3

```
1: Final Tests: Summary for of c3
  2: -----
  4: Hidden Tests:
  First 1 -- Activations: relu backward: 1 / 1
First 1 -- Activations: relu forward: 1 / 1
First 1 -- Activations: sigmoid backward: 1 / 1
First 1 -- Activations: sigmoid forward: 1 / 1
First 1 -- Activations: sigmoid forward: 1 / 1
          Part 1 -- ACTIVATIONS: SIGMOID FORWARD: 1 / 1
Part 1 -- Linear layer: backward (Private only): 1 / 1
Part 1 -- Linear layer: forward (Private only): 1 / 1
Part 1 -- Linear layer: shape mismatch: 1 / 1
Part 1 -- Linear layer: smoke test: 1 / 1
  9:
10:
11:
13:
            Part 1 -- Linear layer: weight update (Private only): 1 / 1
            Part 1 -- Linear layer: zero update: 1 / 1
          Part 1 -- Linear layer: zero update: 1 / 1
Part 1 -- Network: smoke test: 1 / 1
Part 1 -- Network: zero update: 1 / 1
Part 1 -- Preprocess: different dataset: 1 / 1
Part 1 -- Preprocess: min-max scale: 1 / 1
Part 1 -- Preprocess: revert: 1 / 1
Part 1 -- Preprocess: shape mismatch: 1 / 1
Part 1 -- Trainer: 1D regression: 1 / 1
Part 1 -- Trainer: shuffle preserves pairs: 1 / 1
Part 1 -- Trainer: shuffle preserves pairs: 1 / 1
Part 1 -- Trainer: shuffle preserves pairs: 1 / 1
15:
16:
17:
18:
19:
20:
21:
22:
             Part 1 -- Trainer: shuffle removes correlations: 1 / 1
            Part 2: Regressor:
                                                                                                                    1 / 1
             Part 2: Preprocessing:
25:
                                                                                                                     1 / 1
              Part 2: Performance:
                                                                                                                      1 / 1
26:
28: Git Repo: git@gitlab.doc.ic.ac.uk:lab2021_autumn/neural_networks_65.git
29: Commit TD: 7b588
```

```
Final Tests
                         part2_house_value_regression.py: 1/8
                                                                                  :c3
                                                                                             Final Tests
                                                                                                                      part2_house_value_regression.py: 2/8
                                                                                                                                                                                :c3
                                                                                                           self.x = x
    1: import copy
                                                                                                67.
                                                                                                68.
                                                                                                           self.nb epoch = nb epoch
    3: import torch
                                                                                                69:
                                                                                                           self.n hidden = n hidden
    4: import torch.nn as nn
                                                                                                70:
                                                                                                           self.n nodes = n nodes
                                                                                                71:
    6: import pickle
                                                                                                72:
                                                                                                           self.output size = 1
    7: import numpy as np
                                                                                                73:
                                                                                                           if x is not None:
    8: import pandas as pd
                                                                                                74:
                                                                                                               X, = self. preprocessor(x, training=True)
                                                                                                75.
                                                                                                               self.input samples, self.input size = X.shape
   10: from sklearn.utils import shuffle
                                                                                                76.
                                                                                                           ٠٩٥١م
   11: from sklearn.preprocessing import LabelBinarizer, MinMaxScaler
                                                                                                77:
                                                                                                               self.input_samples = 0
   12: from sklearn.metrics import mean squared error
                                                                                                78:
                                                                                                               self.input size = 13
   13: from sklearn.model_selection import GridSearchCV, train_test_split, \
                                                                                                79:
          cross_val_score
                                                                                                80:
                                                                                                           # Pre-set or placeholder model variables
   15: from sklearn.base import BaseEstimator
                                                                                                81 •
                                                                                                           if activations is not None:
                                                                                                82:
                                                                                                               self.activations = activations
  16:
   17: from part2_network import Network
                                                                                                83:
                                                                                                           else:
   18:
                                                                                                84.
                                                                                                               self.activations = [nn.ReLU() for i in range(self.n hidden)]
   19:
                                                                                                85:
   20: class Regressor(BaseEstimator):
                                                                                                86.
                                                                                                           self.n inputs = None
                                                                                                87:
  21:
                                                                                                           self.model = None
          def __init__(self,
   22:
                                                                                                88:
                                                                                                           self.criterion = nn.MSELoss()
   23:
                                                                                                29.
                                                                                                           self.optimiser = optimiser
                      x=None.
   24:
                      nb epoch=500.
                                                                                                90:
   25:
                       n_hidden=1,
                                                                                                91:
                                                                                                           # Optimisation features
   26:
                      n_nodes=None,
                                                                                                92:
                                                                                                           self.early_stopping = early_stopping
                                                                                                93.
   27:
                       activations=None,
                                                                                                           self.earliest_stop = earliest_stop
   28:
                      optimiser=torch.optim.Adam,
                                                                                                94:
                                                                                                           self.patience = patience
   29:
                       early stopping=True,
                                                                                                95.
                                                                                                           self.dropout = dropout
                       earliest stop=100,
                                                                                                96:
   30:
                                                                                                97:
   31:
                       patience=3,
                                                                                                           98:
                                                                                                                                  ** END OF YOUR CODE **
   32:
                       dropout=False):
                                                                                               99:
                                                                                                           33.
              Initialise the model.
                                                                                               100:
  34:
                                                                                               101:
   35:
                                                                                                       def _build_optimiser(self):
   36:
                                                                                               102:
              Arguments:
                                                                                               103:
   37 •
                  - x {pd.DataFrame} -- Raw input data of shape
                                                                                                           Builds an optimiser from the set optimiser types
                                                                                               104:
   38:
                      (batch_size, input_size), used to compute the size
   39:
                      of the network.
                                                                                               105:
                                                                                                           Returns: {nn.optim} instantiated optimiser
   40:
                  - nb_epoch {int} -- number of epoch to train the network.
                                                                                               106:
                  - n_hidden {int} -- number of layers with activation functions,
                                                                                               107:
                                                                                                           # Use given model optimiser
   41:
   42:
                      includes the input layer as it requires activation function
                                                                                               108:
                                                                                                           if self.optimiser == torch.optim.Adam:
                                                                                               109.
   43:
                  - n_nodes {List[int]} -- number of nodes in each layer except
                                                                                                               return self.optimiser(self.model.parameters(),
                                                                                               110:
   44:
                      input layer, that is generated automatically
                                                                                                                                    lr=self.learning_rate,
   45:
                  - activations {List[nn.modules.activation]} -- list of activation
                                                                                               111:
                                                                                                                                    weight_decay=1e-4)
   46:
                      functions for our model
                                                                                               112:
   47:
                  - optimiser {torch.optim} -- optimiser to be used for the model
                                                                                               113:
                                                                                                           # Use classic SGD
   48:
                  - early_stopping {boolean} -- whether early stopping is enabled for
                                                                                               114:
                                                                                                           return self.optimiser(self.model.parameters(), lr=self.learning_rate)
                                                                                               115:
   49:
                      training the model
   50.
                  - earliest_stop {int} -- earliest epoch to begin to consier early
                                                                                               116:
                                                                                                       def _preprocessor(self, x, y=None, training=False):
   51:
                      stopping, used to prevent stopping too early before model starts
                                                                                               117:
   52:
                      converging
                                                                                               118:
                                                                                                           Preprocess input of the network.
   53:
                  - patience {int} -- number of consecutive epochs without a better
                                                                                               119:
                                                                                               120:
   54:
                      model before stopping
                                                                                                           Arguments:
   55:
                  - dropout {boolean} -- whether dropout is enabled during training
                                                                                               121:
                                                                                                               - x {pd.DataFrame} -- Raw input array of shape
   56:
                                                                                               122:
                                                                                                                   (batch_size, input_size).
              ....
   57:
                                                                                               123:
                                                                                                               - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1).
   58:
                                                                                               124:
                                                                                                               - training {boolean} -- Boolean indicating if we are training or
   59:
              125:
                                                                                                                   testing the model.
   60:
                                     ** START OF YOUR CODE **
                                                                                               126:
              *****************************
                                                                                               127:
   61:
                                                                                                           Returns:
   62:
                                                                                               128:
                                                                                                               - {torch.tensor} -- Preprocessed input array of size
   63:
              self.binariser labels = None
                                                                                               129:
                                                                                                                   (batch_size, input_size).
   64:
              self.v scaler = None
                                                                                               130:
                                                                                                               - {torch.tensor} -- Preprocessed target array of size
                                                                                               131:
   65:
              self.learning_rate = 0.01
                                                                                                                   (batch_size, 1).
                                                                                               132:
   66:
```

1311	Final Test	part2_house_value_regression.py: 3/8	:c3	Final Tests	part2_house_value_regression.py: 4/8 :c3
Matter M	133:	нин		199:	(batch_size, input_size).
According to Statistics Sta					- y {pd.DataFrame} Raw output array of shape (batch_size, 1).
### ### ### ### #### #################			#		
### FFFF And marked within the numerical colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the colours of the given input data The numerical colours of the given input d			. д		
### FFILE NOW values within the numerical noturns of the given input data 101		**************************************	#		self (Regressor) framed model.
numerical_x = n. depo("coen_proximity", asis=1) 101		# Fill NaN values within the numerical columns of the given input data			
Till_keys = numerical_x_noncinglit_co_inter()					
### Sommalies numerical values to exale to values between 0 and 1 131	141:				
14:		<pre>numerical_x = numerical_x.fillna(value=fill_keys)</pre>			
145: numerical_labels = numerical_x.keys()					#######################################
146: x_acair — MindawScair					
101:					
148:					
1993					
151	149:			215:	
binariar = LabolBinariar() 103:					
153: ccean_groximity = x ^ocean_proximity' 210: classes					[self.input_size for i in range(self.n_hidden)]
15th 1f training: 220; self.modal Hetwork(n_layersmeelf.m_hidden, 1. 1. 1. 1. 1. 1. 1. 1					
105: If training: ### Create new Dinaries: parameters for one-hot encoding #### Create new Dinaries: Int_Transform(ocean_proximity) ### Dinaries_data - Dinaries. Int_Transform(ocean_proximity) ### Dinaries_data - Dinaries. Int_Transform(ocean_proximity) ### Dinaries_data - Dinaries. Int_Transform(ocean_proximity) ### Spit data into train-validation if early stopping ### Dinaries. Interior		ocean_proximity = x["ocean_proximity"]			
Section of the properties of		if training.			
binarised_data = binariser_classes					n_inputs-seii.n_inputs/
Second content of the content of t					<pre>self.optimiser = self. build optimiser()</pre>
160:	158:			224:	
161:					
162:					
163:					<pre>x, val_x, y, val_y = train_test_split(x,</pre>
164:		<pre>binarised_data = binariser.transform(ocean_proximity)</pre>			y, train circ-0 9
165: op_frame = pd.DataFrame(data=pinarised_data)		# One-hot encoding parameters			
166:					
168;	166:			232:	# Preprocess training data
169;					<pre>X_train, Y_train = selfpreprocessor(x, y=y, training=True)</pre>
170:					
171:		<pre>x = pd.concat([numerical_x, op_frame], axis=1)</pre>			
172; device = torch.device('ppu') 238:		# Process on CPU, as Lab computers throw CUDA error			
173: # Create tensor from preprocessed x data 240: best_model = None 175: x_tensor = torch.tensor(x.values, device=device, requires_grad=True) 241: min_loss = 999					1var,var
174:					# Values for determining and storing optimal model
176:		# Create tensor from preprocessed x data			best_model = None
177: 178: if isinstance(y, pd.DataFrame): 244: # Set model mode to train					
178:		y_tensor = None			strikes = 0
179:		if isingtones (v. nd DataEnama).			# Cot model made to their
180:			,		
181: self.y_scaler = MinMaxScaler() 182: y = pd.DataFrame(data=self.y_scaler.fit_transform(y), 248: self.optimiser.zero_grad() 183:					
183:	181:			247:	
184: y_tensor = torch.tensor(y, values, device=device, requires_grad=True)		<pre>y = pd.DataFrame(data=self.y_scaler.fit_transform(y),</pre>			self.optimiser.zero_grad()
185: 186: # Return preprocessed x and y, return None for y if it was None 187: return x_tensor, y_tensor 188: 189: ###################################					
186: # Return preprocessed x and y, return None for y if it was None		<pre>y_tensor = torch.tensor(y.values, device=device, requires_grad=Tru</pre>	e)		
187: return x_tensor, y_tensor 188: 189: ####################################		# Deturn nyanyagagad w and w waturn Nana fan w if it was Nana			
188: 189: ####################################					
189:		recurn x_tensor, y_tensor			diopout-seri.diopout/
191:	189:	#######################################	#	255:	<pre>loss = self.criterion(y_predicted, Y_train)</pre>
192: 193:	190:	# ** END OF YOUR CODE **		256:	
193: def fit (self, x, y): 259: self.optimiser.step() 194: """ 260: 195: Regressor training function 261: if self.early_stopping: 196: 4rguments: 262: # Predict off validatation data and calculate loss 197: Arguments: 263: val_y_pred = self.model(X_val, self.activations)		#######################################	#		
194: """ 260: 195: Regressor training function 261: if self.early_stopping: 196: 262: # Predict off validatation data and calculate loss 197: Arguments: 263: val_y_pred = self.model(X_val, self.activations)					
195: Regressor training function 261: if self.early_stopping: 196: 197: Arguments: 268: # Predict off validatation data and calculate loss 198: val_y_pred = self.model(X_val, self.activations)		<pre>def fit(self, x, y):</pre>			self.optimiser.step()
196: 197: Arguments: 262: # Predict off validatation data and calculate loss 263: val_y_pred = self.model(X_val, self.activations)		Regressor training function			if self early stopping.
197: Arguments: 263: val_y_pred = self.model(X_val, self.activations)		Adjected Claiming Lanceton			
		Arguments:			
	198:	<pre>- x {pd.DataFrame} Raw input array of shape</pre>		264:	

```
Final Tests
                      part2_house_value_regression.py: 5/8
                                                                        . . . 3
                                                                                  Final Tests
                                                                                                        part2_house_value_regression.py: 6/8
                                                                                                                                                           . . . 3
 265:
                                                                                    330:
                                                                                               ....
                   # print(f'val loss: {val loss:.5f} best loss: {min loss:.5f}')
 266:
                                                                                    331:
                                                                                              Function to evaluate the model accuracy on a validation dataset.
 267:
                   # Determine if training should stop early
                                                                                    332:
 268:
                   if val loss.item() > min loss:
                                                                                    333.
                                                                                              Arguments:
 269.
                      if epoch >= self.earliest_stop:
                                                                                    331.
                                                                                                  - x {pd.DataFrame} -- Raw input array of shape
 270:
                          if strikes == self.patience:
                                                                                    335:
                                                                                                     (batch size, input size).
 271 •
                             # Early stop
                                                                                    336:
                                                                                                  - v {pd.DataFrame} -- Raw ouput array of shape (batch size, 1).
 272:
                             self.model = best model
                                                                                    337:
 273.
                             return self
                                                                                    338.
                                                                                              Returns:
 274 •
                                                                                    339:
                                                                                                  {float} -- Quantification of the efficiency of the model.
                          -1 -- .
 275:
                             strikes += 1
                                                                                    340:
 276:
                   else:
                                                                                    341:
 277:
                       # Clear strikes and save optimal model
                                                                                    342:
 278:
                      strikes = 1
                                                                                    343:
                                                                                               279:
                      min loss = val loss.item()
                                                                                    344 •
                                                                                                                   ** START OF YOUR CODE **
                      best_model = copy.deepcopy(self.model)
                                                                                    3/15 •
                                                                                               *******************************
 280:
 281:
                                                                                    346:
                                                                                    347:
 282 .
                if (epoch + 1) % 10 == 0:
                                                                                              y predicted = self.predict(x)
                   print(f'epoch {epoch + 1} / {self.nb_epoch} loss = /
                                                                                    348:
 283:
                                                                                              mse = mean_squared_error(y, y_predicted)
{loss.item():.8f}')
                                                                                    349:
                                                                                              return np.sgrt (mse)
 284 •
                                                                                    350.
 285:
            if self.early stopping:
                                                                                    351:
                                                                                               286.
                # Return optimal model if available
                                                                                    352.
                                                                                                                   ** END OF YOUR CODE **
 287:
                                                                                               *******************************
                self.model = best model
                                                                                    353:
 288:
            return self
                                                                                    354:
 289.
                                                                                    355:
            290:
                                                                                   356: def save_regressor(trained_model):
 291:
                                 ** END OF YOUR CODE **
                                                                                    357:
 292.
            358:
                                                                                           Utility function to save the trained regressor model in part2_model.pickle.
 293:
                                                                                    359:
                                                                                    360:
 294:
         def predict(self, x):
                                                                                           # If you alter this, make sure it works in tandem with load_regressor
 295:
                                                                                    361:
                                                                                           with open('part2_model.pickle', 'wb') as target:
 296:
            Ouput the value corresponding to an input x.
                                                                                    362:
                                                                                              pickle.dump(trained model, target)
 297:
                                                                                    363:
                                                                                           print("\nSaved model in part2_model.pickle\n")
 298 .
                                                                                    364:
 299:
                x {pd.DataFrame} -- Raw input array of shape
                                                                                    365:
 300:
                                                                                   366: def load_regressor():
                   (batch_size, input_size).
 301:
                                                                                    367:
 302:
                                                                                    368:
                                                                                           Utility function to load the trained regressor model in part2_model.pickle.
 303:
                {np.darray} -- Predicted value for the given input (batch_size, 1).
                                                                                    369:
 304:
                                                                                    370:
                                                                                           # If you alter this, make sure it works in tandem with save regressor
                                                                                    371:
 305:
                                                                                           with open('part2_model.pickle', 'rb') as target:
 306:
                                                                                    372:
                                                                                              trained_model = pickle.load(target)
 307:
            373:
                                                                                           print("\nLoaded model in part2_model.pickle\n")
 308:
                                 ** START OF YOUR CODE **
                                                                                    374:
                                                                                           return trained model
 309:
            375:
 310:
                                                                                    376:
 311 •
            if self.model is not None:
                                                                                    377: def RegressorHyperParameterSearch(x, y):
 312:
                                                                                    378:
               X, _ = self._preprocessor(x, training=False) # Do not forget
                                                                                           # Ensure to add whatever inputs you deem necessary to this function
 313:
                                                                                    379:
 314 •
                # Set model to evaluate mode
                                                                                    380:
                                                                                           Performs a hyper-parameter for fine-tuning the regressor implemented
 315:
                self.model.eval()
                                                                                    381:
                                                                                           in the Regressor class.
 316:
                                                                                    382:
 317:
                # Predict from input and scale result using y scaler from training
                                                                                    383:
                                                                                           Arguments:
 318:
                y_predicted = self.model(X, self.activations).detach().numpy()
                                                                                    384 •
                                                                                               - x {pd.DataFrame} -- Raw input array of shape
 319:
                y_predicted_scaled = self.y_scaler.inverse_transform(y_predicted)
                                                                                    385:
                                                                                                  (batch_size, input_size).
 320:
                                                                                    386:
                                                                                               - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1)
 321 •
                return y_predicted_scaled
                                                                                    387:
 322:
                                                                                    388.
                                                                                           Returns:
 323:
            return None
                                                                                    389:
                                                                                              - {dict{str, any}} optimised hyper-parameters.
 324:
                                                                                    390:
            391:
 325:
 326:
                                 ** END OF YOUR CODE **
                                                                                    392:
 327:
            393:
                                                                                           328:
                                                                                   394:
                                                                                                                ** START OF YOUR CODE **
                                                                                           *****************************
 329:
                                                                                    395:
         def score(self, x, y):
```

```
Final Tests
                          part2_house_value_regression.py: 7/8
                                                                                     . . . 3
  396:
  397:
          param grid = [
  398:
  399.
                   'n hidden': [1],
  400:
                   'n_nodes': [[13], [24], [32]],
  401:
                   'activations': [
  402:
                       [nn.Sigmoid()],
  403:
                       [nn.ReLU()],
  404:
                       [nn.Tanh()]
  405.
  406:
              },
  407:
                   'n_hidden': [2],
  408:
  409:
                   'n_nodes': [[13, 13], [20, 20], [24, 32], [24, 64]],
  410:
                   'activations': [
  411:
                       [nn.Sigmoid(), nn.Sigmoid()],
  412:
                       [nn.ReLU(), nn.ReLU()],
  413:
                       [nn.Tanh(), nn.Tanh()]
  414:
  415:
              },
  416:
  417:
                   'n hidden': [3],
  418:
                   'n_nodes': [[13, 13, 13], [32, 64, 32], [64, 128, 64], [512, 512, \( \mu \)
12811.
  419:
                   'activations': [
                       [nn.Sigmoid(), nn.Sigmoid(), nn.Sigmoid()],
  420:
  421:
                       [nn.Sigmoid(), nn.ReLU(), nn.ReLU()],
  422:
                       [nn.ReLU(), nn.ReLU(), nn.ReLU()],
  423:
                       [nn.ReLU(), nn.Sigmoid(), nn.ReLU()]
  424:
  425:
                   'patience': [1, 3, 5, 8],
  426:
                   'dropout': [True, False]
  427:
  428:
  429:
                   'n hidden': [4],
  430:
                   'n_nodes': [[32, 64, 64, 32]],
  431 •
                   'activations': [
  432 •
                       [nn.Sigmoid(), nn.Tanh(), nn.ReLU(), nn.Sigmoid()],
  433:
                       [nn.ReLU(), nn.ReLU(), nn.ReLU(), nn.ReLU()]
  434.
  435:
              },
  436:
  437:
  438:
           regressor = Regressor()
  439:
          grid_search = GridSearchCV(regressor,
  440:
                                     param_grid,
  441:
  442 •
                                      scoring="neg_mean_squared_error";
  443:
  444:
          grid_search.fit(x, y)
  445:
  446:
          print(grid search.best score )
  447:
          print (grid_search.best_params_)
  448:
          print (grid_search.best_estimator_)
  449:
  450:
          cvres = grid_search.cv_results_
  451:
  452 •
           for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
  453:
              print (np.sqrt (-mean_score), params)
  454:
  455:
           # Return the chosen hyper parameters
  456:
           return grid_search.best_params_
  457:
  458:
           459:
                                   ** END OF YOUR CODE **
```

```
Final Tests
                           part2_house_value_regression.py: 8/8
  461.
  462:
  463: def example main():
  464:
           output_label = "median_house_value"
  465:
  466:
           # Use pandas to read CSV data as it contains various object types
  467:
           # Feel free to use another CSV reader tool
  468:
           # But remember that LabTS tests take Pandas Dataframe as inputs
  469:
           data = pd.read csv("housing.csv")
  470:
  471:
           # Set manual seed for result replication
  472:
           torch.manual seed(42)
  473:
  474:
           # Shuffle data
  475 •
           data = shuffle(data, random_state=42)
  476:
           data.reset_index(inplace=True, drop=True)
  477:
  478:
           # Spliting input and output
  479:
           x raw = data.loc[:, data.columns != output label]
  480:
           y raw = data.loc[:, [output label]]
  481:
  482:
           x_train, x_test, y_train, y_test \
  483:
               = train test split(x raw, y raw, test size=0.2, random state=42)
  484 •
  485:
           # Training
  486:
           # Build Regressor
  487 •
           acitivations = [nn.ReLU(), nn.ReLU(), nn.ReLU()]
  488:
           n nodes = [512, 512, 128]
  489:
           regressor = Regressor(x train,
  490:
                                 nb epoch=500,
  491:
                                 n hidden=3,
  492:
                                 n_nodes=n_nodes,
  493:
                                 activations=acitivations,
  494:
                                 patience=5.
  495:
                                 dropout=False)
  496:
           # regressor.fit(x_train, y_train)
  497:
           # save_regressor(regressor)
  498 •
  499:
           # Perform cross validation and obtain average error
  500:
           nmse score = -cross val score(regressor,
  501:
                                          x raw,
  502:
                                         v raw.
  503:
                                          scoring="neg_mean_squared_error",
  504:
                                         cv=5)
  505:
  506:
           scores = np.sgrt(nmse score)
  507:
  508:
           print("Scores:", scores)
  509:
           print("Mean:", scores.mean())
  510:
           print("Standard Deviation:", scores.std())
  511:
  512:
           # RegressorHyperParameterSearch(x raw, y raw)
  513:
  514:
  515:
           # error = regressor.score(x_train, y_train)
  516:
           # print("\nRegressor error: {}\n".format(error))
  517:
  518:
           # error = regressor.score(x_test, y_test)
  519:
           # print("\nRegressor error: {}\n".format(error))
  520:
  521:
  522: if __name__ == "__main__":
           example main()
```

460:

```
Final Tests
                                part1_nn_lib.py: 1/11
                                                                                 :c3
                                                                                           Final Tests
                                                                                                                           part1_nn_lib.py: 2/11
                                                                                                                                                                            :c3
                                                                                              67:
   1: import numpy as np
   2: import pickle
                                                                                              68:
                                                                                                     CrossEntropyLossLayer: Computes the softmax followed by the negative
   3 •
                                                                                              69:
                                                                                                     log-likelihood loss.
   4:
                                                                                              70:
   5: def xavier_init(size, gain = 1.0):
                                                                                              71:
   6:
                                                                                              72:
                                                                                                     def ___init___(self):
   7:
          Xavier initialization of network weights.
                                                                                              73:
                                                                                                         self. cache current = None
   8:
                                                                                              74:
   9:
                                                                                              75:
                                                                                                     @staticmethod
          Arguments:
             - size {tuple} -- size of the network to initialise.
                                                                                              76:
   10:
                                                                                                     def softmax(x):
  11.
              - gain {float} -- gain for the Xavier initialisation.
                                                                                              77:
                                                                                                         numer = np.exp(x - x.max(axis=1, keepdims=True))
  12:
                                                                                              78:
                                                                                                         denom = numer.sum(axis=1, keepdims=True)
  13:
          Returns:
                                                                                              79:
                                                                                                         return numer / denom
              {np.ndarray} -- values of the weights.
   14:
                                                                                              80:
   15:
                                                                                              81:
                                                                                                     def forward(self, inputs, y_target):
                                                                                              82:
          low = -gain * np.sqrt(6.0 / np.sum(size))
  16:
                                                                                                         assert len(inputs) == len(y_target)
  17:
          high = gain * np.sqrt(6.0 / np.sum(size))
                                                                                              83:
                                                                                                         n_obs = len(y_target)
                                                                                              84:
   18:
          return np.random.uniform(low=low, high=high, size=size)
                                                                                                         probs = self.softmax(inputs)
   19:
                                                                                              85:
                                                                                                         self._cache_current = y_target, probs
  20:
                                                                                              86.
  21: class Layer:
                                                                                              87:
                                                                                                         out = -1 / n obs * np.sum(y target * np.log(probs))
  22:
                                                                                              88:
                                                                                                         return out
   23:
          Abstract layer class.
                                                                                              89.
   24:
                                                                                              90:
                                                                                                     def backward(self):
   25:
                                                                                              91:
                                                                                                         y_target, probs = self._cache_current
   26:
          def __init__(self, *args, **kwargs):
                                                                                              92 .
                                                                                                         n_obs = len(y_target)
                                                                                              93:
   27:
              raise NotImplementedError()
                                                                                                         return -1 / n_obs * (y_target - probs)
   28:
                                                                                              94:
   29:
          def forward(self, *args, **kwargs):
                                                                                              95:
   30:
              raise NotImplementedError()
                                                                                              96: class SigmoidLayer(Layer):
   31:
                                                                                              97:
                                                                                              98:
                                                                                                     SigmoidLayer: Applies sigmoid function elementwise.
   32:
          def __call__(self, *args, **kwargs):
   33:
              return self.forward(*args, **kwargs)
                                                                                              99:
  34:
                                                                                             100:
                                                                                                     def __init__(self):
   35:
          def backward(self, *args, **kwargs):
                                                                                             101:
   36:
              raise NotImplementedError()
                                                                                             102:
   37:
                                                                                             103:
                                                                                                         Constructor of the Sigmoid layer.
   38:
                                                                                             104:
          def update_params(self, *args, **kwargs):
   39:
              pass
                                                                                             105:
                                                                                                         self. cache current = None
   40:
                                                                                             106:
   41:
                                                                                             107:
                                                                                                     def forward(self, x):
   42: class MSELossLayer(Layer):
                                                                                             108:
                                                                                             109:
   43:
                                                                                                         Performs forward pass through the Sigmoid layer.
   44:
          MSELossLayer: Computes mean-squared error between y_pred and y_target.
                                                                                             110:
   45:
                                                                                             111:
                                                                                                         Logs information needed to compute gradient at a later stage in
   46:
                                                                                             112:
                                                                                                          ' cache current'.
   47:
          def ___init___(self):
                                                                                             113:
   48 •
              self._cache_current = None
                                                                                             114:
                                                                                                         Arguments:
                                                                                             115:
   49:
                                                                                                             x {np.ndarray} -- Input array of shape (batch_size, n_in).
   50.
          @staticmethod
                                                                                             116:
   51:
          def _mse(y_pred, y_target):
                                                                                             117:
   52:
              return np.mean((y_pred - y_target) ** 2)
                                                                                             118:
                                                                                                             {np.ndarray} -- Output array of shape (batch_size, n_out)
   53:
                                                                                             119:
   54:
                                                                                             120:
                                                                                                         @staticmethod
   55:
          def _mse_grad(y_pred, y_target):
                                                                                             121:
                                                                                                                                ** START OF YOUR CODE **
   56:
              return 2 * (y_pred - y_target) / len(y_pred)
                                                                                             122:
                                                                                                         57:
                                                                                             123:
                                                                                                         sigmoid = 1 / (1 + np.exp(-x))
   58:
          def forward(self, y_pred, y_target):
                                                                                             124:
                                                                                                         self._cache_current = sigmoid
   59:
                                                                                             125:
              self._cache_current = y_pred, y_target
   60:
              return self._mse(y_pred, y_target)
                                                                                             126:
                                                                                                         return sigmoid
   61:
                                                                                             127:
                                                                                             128:
                                                                                                         62:
          def backward(self):
   63:
              return self. mse grad(*self. cache current)
                                                                                             129:
                                                                                                                                ** END OF YOUR CODE **
                                                                                                         64:
                                                                                             130:
                                                                                             131:
   65:
                                                                                             132:
   66: class CrossEntropyLossLayer(Layer):
                                                                                                     def backward(self, grad_z):
```

7b588

```
Final Tests
                         part1_nn_lib.py: 3/11
                                                                . . . 3
                                                                        Final Tests
                                                                                                 part1_nn_lib.py: 4/11
                                                                                                                                        :c3
 133:
           ....
                                                                         199:
                                                                                   inputs of layer).
 134:
           Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
                                                                         200:
 135:
           the output of this laver, performs back pass through the laver (i.e.
                                                                         201:
                                                                                   Arguments:
 136:
           computes gradients of loss with respect to parameters of layer and
                                                                         202:
                                                                                      grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
 137:
           inputs of layer).
                                                                         203:
 138:
                                                                         204:
                                                                                   Returns:
 139:
           Arguments:
                                                                         205:
                                                                                      {np.ndarray} -- Array containing gradient with repect to layer
 140:
              grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
                                                                         206:
                                                                                         input, of shape (batch_size, n_in).
 141:
                                                                         207 •
 142:
                                                                         208.
                                                                                   143.
              {np.ndarray} -- Array containing gradient with repect to layer
                                                                         209:
                                                                                                     ** START OF YOUR CODE **
 144:
                 input, of shape (batch_size, n_in).
                                                                         210:
                                                                                   ....
 145:
                                                                         211:
                                                                                   z = self. cache current
 146:
           212:
                                                                                   relu_derivative = np.int64(z > 0)
 147:
                             ** START OF YOUR CODE **
                                                                         213:
           214:
 148 •
                                                                                   return grad_z * relu_derivative
 149:
                                                                         215:
           a = self._cache_current
 150:
           sigmoid derivative = a * (1 - a)
                                                                         216:
                                                                                   ** END OF YOUR CODE **
                                                                         217:
 151:
 152:
           return grad z * sigmoid derivative
                                                                         218.
                                                                                   153.
                                                                         219.
           154 •
                                                                         220:
 155:
                             ** END OF YOUR CODE **
                                                                         221: class LinearLayer(Layer):
 156:
           222:
 157:
                                                                         223:
                                                                                LinearLayer: Performs affine transformation of input.
 158 •
                                                                         224:
 159: class ReluLayer(Layer):
                                                                         225:
 160:
                                                                         226:
                                                                                def __init__(self, n_in, n_out):
                                                                         227:
 161:
        ReluLayer: Applies Relu function elementwise.
 162:
                                                                         228:
                                                                                   Constructor of the linear layer.
                                                                         229:
 163:
                                                                         230:
 164:
        def __init__(self):
                                                                                   Arguments:
 165:
                                                                         231:
                                                                                      - n_in {int} -- Number (or dimension) of inputs.
 166:
                                                                         232:
                                                                                      - n_out {int} -- Number (or dimension) of outputs.
           Constructor of the Relu layer.
 167:
                                                                         233:
           self._cache_current = None
 168:
                                                                         234:
                                                                                   self.n_in = n_in
 1.69:
                                                                         235:
                                                                                   self.n_out = n_out
 170:
                                                                         236:
        def forward(self, x):
 171:
                                                                         237:
                                                                                   172:
           Performs forward pass through the Relu layer.
                                                                         238:
                                                                                                     ** START OF YOUR CODE **
 173:
                                                                         239:
                                                                                   174:
           Logs information needed to compute gradient at a later stage in
                                                                         240:
                                                                                   self._W = xavier_init((n_in, n_out))
 175:
                                                                         241:
           `_cache_current`.
                                                                                   self._b = np.zeros(n_out)
 176:
                                                                         242:
 177:
           Arguments:
                                                                         243:
                                                                                   self._cache_current = None
 178:
              x {np.ndarray} -- Input array of shape (batch size, n in).
                                                                         244 •
                                                                                   self. grad W current = None
 179:
                                                                         245:
                                                                                   self._grad_b_current = None
 180:
           Returns
                                                                         246:
 181:
                                                                         247:
                                                                                   ***************************
             {np.ndarray} -- Output array of shape (batch_size, n_out)
 182:
                                                                         248:
                                                                                                     ** END OF YOUR CODE **
 183:
           249:
                                                                                   184:
                             ** START OF YOUR CODE **
                                                                         250:
 185:
           251:
                                                                                def forward(self, x):
 186:
                                                                         252:
           self._cache_current = x
 187:
                                                                         253:
                                                                                   Performs forward pass through the layer (i.e. returns Wx + b).
 188:
           return np.maximum(0, x)
                                                                         254:
 189:
                                                                         255:
                                                                                   Logs information needed to compute gradient at a later stage in
 190:
           256:
                                                                                   `_cache_current`.
 191 •
                             ** END OF YOUR CODE **
                                                                         257:
 192:
           258:
                                                                                   Arguments:
 193:
                                                                         259:
                                                                                      x {np.ndarray} -- Input array of shape (batch_size, n_in).
 194:
        def backward(self, grad_z):
                                                                         260:
 195:
                                                                         261:
 196:
           Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
                                                                         262:
                                                                                      {np.ndarray} -- Output array of shape (batch_size, n_out)
 197:
           the output of this layer, performs back pass through the layer (i.e.
                                                                         263:
 198 •
           computes gradients of loss with respect to parameters of layer and
                                                                         264:
```

Final Tests

```
265:
                          ** START OF YOUR CODE **
266:
         267:
         self. cache current = x
268:
         return x @ self. W + self. b
269.
270:
         271:
                          ** END OF YOUR CODE **
272:
         273:
274:
      def backward(self, grad z):
275:
276:
         Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
277:
         the output of this layer, performs back pass through the layer (i.e.
278:
         computes gradients of loss with respect to parameters of layer and
279.
         inputs of laver).
280:
281:
         Arguments:
282:
            grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
283:
284:
285:
            {np.ndarray} -- Array containing gradient with repect to layer
286:
               input, of shape (batch_size, n_in).
         .....
287 •
         288 .
289:
                          ** START OF YOUR CODE **
290:
         291 •
         self._grad_W_current = self._cache_current.transpose() @ grad_z
292:
         self._grad_b_current = np.ones((1, np.shape(grad_z)[0])) @ grad_z
293.
         return grad z @ self. W.transpose()
294:
         295:
296:
                          ** END OF YOUR CODE **
297:
         298:
299.
      def update_params(self, learning_rate):
300:
301:
         Performs one step of gradient descent with given learning rate on the
302:
         layer's parameters using currently stored gradients.
303:
304:
         Arguments:
305:
           learning_rate {float} -- Learning rate of update step.
306:
307:
         *****************************
308:
                          ** START OF YOUR CODE **
         309:
310:
         self._W = self._W - learning_rate * self._grad_W_current
311 •
         self._b = self._b - learning_rate * self._grad_b_current
312:
313:
         314:
                          ** END OF YOUR CODE **
315:
         316:
317:
318: class MultiLayerNetwork(object):
319:
320:
      MultiLayerNetwork: A network consisting of stacked linear layers and
321:
      activation functions.
322:
323:
324:
      def __init__(self, input_dim, neurons, activations):
325:
326:
         Constructor of the multi layer network.
327:
328:
329:
            - input_dim {int} -- Number of features in the input (excluding
330:
              the batch dimension).
```

```
331: Â Â Â Â Â A - neurons (list) -- Number of neurons in each linear layer
               represented as a list. The length of the list determines the
333:
                number of linear lavers.
334 :
            - activations {list} -- List of the activation functions to apply
335.
               to the output of each linear layer.
336:
337:
         self.input dim = input dim
338:
         self.neurons = neurons
339.
         self.activations = activations
340:
341:
          342:
                            ** START OF YOUR CODE **
343:
          344:
         ACTIVATIONS_MAP = {
3/5.
             "sigmoid": lambda size: SigmoidLayer(),
3/6.
            "relu": lambda size: ReluLayer(),
347:
             "identity": lambda size: LinearLaver(size, size)
348:
349:
350.
          self. layers = []
351:
         prev dim = input dim
352:
         for num neurons, activation in zip(neurons, activations):
353.
            self. layers.append(LinearLayer(prev dim, num neurons))
            self. lavers.append(ACTIVATIONS_MAP[activation](num_neurons))
354 •
355:
            prev_dim = num_neurons
356:
          357:
358:
                            ** END OF YOUR CODE **
359.
          360:
361:
      def forward(self, x):
362:
363:
         Performs forward pass through the network.
364:
365:
         Arguments:
366:
            x {np.ndarray} -- Input array of shape (batch_size, input_dim).
367:
368:
         Returns:
369:
             {np.ndarray} -- Output array of shape (batch_size,
370:
                #_neurons_in_final_layer)
371:
372:
          373:
                            ** START OF YOUR CODE **
374:
          375:
         prev = x
376:
          for laver in self. lavers:
377:
            prev = layer.forward(prev)
378:
         return prev
379:
380:
          381:
                            ** END OF YOUR CODE **
382:
          383:
384:
      def __call__(self, x):
385:
         return self.forward(x)
386:
387:
      def backward(self, grad_z):
388:
389:
         Performs backward pass through the network.
390:
391:
         Arguments:
392:
            grad_z {np.ndarray} -- Gradient array of shape (1,
393:
                #_neurons_in_final_layer).
394:
395:
         Returns:
396:
            {np.ndarray} -- Array containing gradient with repect to layer
```

part1_nn_lib.py: 6/11

```
Final Tests
                         part1_nn_lib.py: 7/11
                                                                 . ..3
                                                                          Final Tests
                                                                                                    part1_nn_lib.py: 8/11
                                                                                                                                            :c3
 397:
                                                                           463:
                 input, of shape (batch_size, input_dim).
 398:
                                                                           464:
                                                                                     Arguments:
 399:
           465:
                                                                                        - network {MultiLaverNetwork} -- MultiLaverNetwork to be trained.
 400:
                             ** START OF YOUR CODE **
                                                                           466:
                                                                                        - batch_size {int} -- Training batch size.
 401:
           167.
                                                                                        - nb_epoch {int} -- Number of training epochs.
 402:
           prev = grad z
                                                                           468:
                                                                                        - learning rate {float} -- SGD learning rate to be used in training.
 403:
           for layer in reversed(self._layers):
                                                                           469:
                                                                                        - loss fun {str} -- Loss function to be used. Possible values: mse,
 404.
              prev = layer.backward(prev)
                                                                           470:
 405:
           return prev
                                                                           471 •
                                                                                        - shuffle_flaq {bool} -- If True, training data is shuffled before
 406.
                                                                           472.
                                                                                           training.
 407:
           473:
 408.
                             ** END OF YOUR CODE **
                                                                           474.
                                                                                     self.network = network
 409:
           475:
                                                                                     self.batch size = batch size
 410:
                                                                           476:
                                                                                     self.nb_epoch = nb_epoch
                                                                           477:
 411:
        def update_params(self, learning_rate):
                                                                                     self.learning_rate = learning_rate
 412:
                                                                           178.
                                                                                     self.loss_fun = loss_fun
 413:
           Performs one step of gradient descent with given learning rate on the
                                                                           479:
                                                                                     self.shuffle_flag = shuffle_flag
 414:
           parameters of all layers using currently stored gradients.
                                                                           480 •
 415:
                                                                           481:
                                                                                     416:
           Arguments:
                                                                           482 .
                                                                                                        ** START OF YOUR CODE **
 417:
              learning_rate {float} -- Learning rate of update step.
                                                                           483:
                                                                                     418:
                                                                           484:
                                                                                     LOSS FUN MAP = {
 419.
           485.
                                                                                         "mse": MSELossLayer,
 420:
                             ** START OF YOUR CODE **
                                                                           486:
                                                                                         "cross_entropy": CrossEntropyLossLayer,
 421:
           487:
 422:
           for layer in self._layers:
                                                                           488 .
 423:
              layer.update_params(learning_rate)
                                                                           489:
                                                                                     self._loss_layer = LOSS_FUN_MAP[loss_fun]()
 424:
                                                                           490:
                                                                                     425 •
           491:
                                                                                                        ** END OF YOUR CODE **
 426:
                             ** END OF YOUR CODE **
                                                                           492:
                                                                                     427:
           493:
 428:
                                                                           494:
                                                                                  @staticmethod
 429:
                                                                           495:
                                                                                  def shuffle(input_dataset, target_dataset):
                                                                           496:
 430: def save_network(network, fpath):
                                                                           497:
 431:
                                                                                     Returns shuffled versions of the inputs.
 432:
        Utility function to pickle 'network' at file path 'fpath'.
                                                                           498:
 433:
                                                                           499 .
                                                                                     Arguments:
                                                                           500:
 434 •
        with open(fpath, "wb") as f:
                                                                                        - input_dataset {np.ndarray} -- Array of input features, of shape
 435:
           pickle.dump(network, f)
                                                                           501:
                                                                                            (#_data_points, n_features).
 436:
                                                                           502:
                                                                                        - target_dataset {np.ndarray} -- Array of corresponding targets, of
 437:
                                                                           503:
                                                                                            shape (#_data_points, #output_neurons).
                                                                           504:
 438: def load_network(fpath):
 439:
                                                                           505:
                                                                                     Returns:
 440:
                                                                           506:
        Utility function to load network found at file path 'fpath'.
                                                                                        - {np.ndarray} -- shuffled inputs.
 441:
                                                                           507:
                                                                                        - {np.ndarray} -- shuffled_targets.
 442 •
        with open(fpath, "rb") as f:
                                                                           508:
 443:
           network = pickle.load(f)
                                                                           509:
                                                                                      ** START OF YOUR CODE **
 444 •
        return network
                                                                           510 •
 445:
                                                                           511:
                                                                                     ***************************
 446.
                                                                           512:
                                                                                     assert len(input dataset) == len(target dataset)
 447: class Trainer (object):
                                                                           513.
                                                                                     perm = np.random.permutation(len(input dataset))
 448:
                                                                           514:
                                                                                     return input dataset[perm], target dataset[perm]
 449:
        Trainer: Object that manages the training of a neural network.
                                                                           515:
 450:
                                                                           516:
                                                                                     451:
                                                                           517:
                                                                                                        ** END OF YOUR CODE **
 452:
        def __init__(
                                                                           518:
                                                                                     453:
           self,
                                                                           519:
 454 •
           network,
                                                                           520:
                                                                                  def train(self, input_dataset, target_dataset):
 455:
                                                                           521:
           batch_size,
 456:
           nb epoch.
                                                                           522:
                                                                                     Main training loop. Performs the following steps 'nb_epoch' times:
 457:
                                                                           523:
                                                                                        - Shuffles the input data (if 'shuffle' is True)
           learning_rate,
 458:
           loss fun,
                                                                           524:
                                                                                        - Splits the dataset into batches of size 'batch_size'.
 459:
           shuffle flag,
                                                                           525:
                                                                                        - For each batch:
                                                                           526:
```

460:

461:

462:

527:

528:

batch of inputs.

- Performs forward pass through the network given the current

```
Final Tests
                         part1_nn_lib.py: 9/11
                                                               :c3
                                                                       Final Tests
                                                                                               part1_nn_lib.py: 10/11
                                                                                                                                      :c3
 529:
                                                                        595:
                 - Performs backward pass to compute gradients of loss with
 530 •
                 respect to parameters of network.
                                                                        596:
                                                                                  Arguments:
 531 •
                 - Performs one step of gradient descent on the network
                                                                        597:
                                                                                     data {np.ndarray} dataset used to determine the parameters for
 532:
                                                                        598 .
                                                                                     the normalization.
                 parameters.
                                                                        599.
 533:
 534 •
                                                                        600:
                                                                                  ********************************
           Arguments:
 535:
              - input_dataset {np.ndarray} -- Array of input features, of shape
                                                                        601:
                                                                                                    ** START OF YOUR CODE **
 536.
                 (#_training_data_points, n_features).
                                                                        602.
                                                                                  537:
              - target_dataset {np.ndarray} -- Array of corresponding targets, of
                                                                        603:
                                                                                  self.min = np.min(data, axis=0)
                                                                        604:
 538:
                 shape (#_training_data_points, #output_neurons).
                                                                                  self.max min diff = np.max(data, axis=0) - self.min
 539.
                                                                        605.
 540:
           606:
                                                                                  541:
                            ** START OF YOUR CODE **
                                                                        607:
                                                                                                    ** END OF YOUR CODE **
 542:
           608:
                                                                                  543:
           for i in range(self.nb epoch):
                                                                        609:
 544:
                                                                        610:
              if self.shuffle_flag:
                                                                               def apply(self, data):
 545:
                                                                        611:
                input_dataset, target_dataset = self.shuffle(
 546:
                                                                        612:
                   input dataset,
                                                                                  Apply the pre-processing operations to the provided dataset.
 547:
                    target dataset
                                                                        613:
 548:
                                                                        614:
 549:
                                                                        615:
                                                                                     data {np.ndarray} dataset to be normalized.
 550.
                                                                        616.
              input_batches = np.array_split(input_dataset, self.batch_size)
 551:
              target batches = np.array split(target dataset, self.batch size)
                                                                        617:
                                                                                  Returns:
 552:
                                                                        618:
                                                                                     {np.ndarray} normalized dataset.
 553:
              for input_batch, target_batch in zip(input_batches, target_batches):
                                                                        619:
 554 •
                 network_output = self.network(input_batch)
                                                                        620:
                                                                                  555:
                 self._loss_layer(network_output, target_batch)
                                                                        621:
                                                                                                   ** START OF YOUR CODE **
 556:
                 grad_loss_wrt_outputs = self._loss_layer.backward()
                                                                        622:
                                                                                  557:
                 self.network.backward(grad loss wrt outputs)
                                                                        623:
                                                                                  return (data - self.min) / self.max min diff
 558:
                 self.network.update_params(self.learning_rate)
                                                                        624:
 559:
                                                                        625:
                                                                                  ** END OF YOUR CODE **
 560:
           626:
 561:
                            ** END OF YOUR CODE **
                                                                        627:
                                                                                  628:
 562:
 563:
                                                                        629:
                                                                               def revert(self, data):
 564:
        def eval_loss(self, input_dataset, target_dataset):
                                                                        630:
 565:
                                                                        631 •
                                                                                  Revert the pre-processing operations to retreive the original dataset.
 566:
           Function that evaluate the loss function for given data.
                                                                        632:
 567:
                                                                        633:
 568:
                                                                        634:
           Arguments:
                                                                                     data {np.ndarray} dataset for which to revert normalization.
 569:
                                                                        635:
              - input_dataset {np.ndarray} -- Array of input features, of shape
 570:
                 (#_evaluation_data_points, n_features).
                                                                        636:
                                                                                  Returns:
 571:
              - target_dataset {np.ndarray} -- Array of corresponding targets, of
                                                                        637:
                                                                                     {np.ndarray} reverted dataset.
 572:
                 shape (#_evaluation_data_points, #output_neurons).
                                                                        638:
           ....
 573:
                                                                        639:
                                                                                  574 •
           640:
                                                                                                    ** START OF YOUR CODE **
 575:
                            ** START OF YOUR CODE **
                                                                        641:
                                                                                  576:
           642:
                                                                                  return data * self.max_min_diff + self.min
 577:
           network_output = self.network(input_dataset)
                                                                        643:
 578 •
           return self._loss_layer(network_output, target_dataset)
                                                                        644 •
                                                                                  579:
                                                                        645:
                                                                                                    ** END OF YOUR CODE **
 580:
           646:
                                                                                  581:
                            ** END OF YOUR CODE **
                                                                        647:
 582:
           648:
 583:
                                                                        649: def example_main():
 584:
                                                                        650:
                                                                               input dim = 4
 585: class Preprocessor(object):
                                                                        651:
                                                                               neurons = [16, 3]
 586:
                                                                        652:
                                                                               activations = ["relu", "identity"]
 587:
        Preprocessor: Object used to apply "preprocessing" operation to datasets.
                                                                        653:
                                                                               net = MultiLayerNetwork(input_dim, neurons, activations)
 588:
        The object can also be used to revert the changes.
                                                                        654:
 589:
                                                                        655:
                                                                               dat = np.loadtxt("iris.dat")
 590:
                                                                        656:
                                                                               np.random.shuffle(dat)
 591:
        def __init__(self, data):
                                                                        657:
 592:
                                                                        658:
                                                                               x = dat[:, :4]
 593:
           Initializes the Preprocessor according to the provided dataset.
                                                                        659:
                                                                               y = dat[:, 4:]
 594 •
           (Does not modify the dataset.)
                                                                        660:
```

```
Final Tests
                                part1_nn_lib.py: 11/11
  661:
           split idx = int(0.8 * len(x))
  662:
  663:
          x_train = x[:split_idx]
  664:
          y_train = y[:split_idx]
  665:
          x_val = x[split_idx:]
  666:
          y_val = y[split_idx:]
  667:
  668:
          prep_input = Preprocessor(x_train)
  669:
  670:
          x_train_pre = prep_input.apply(x_train)
  671:
          x_val_pre = prep_input.apply(x_val)
  672:
  673:
          trainer = Trainer(
  674:
              network=net,
  675:
              batch_size=8,
  676:
              nb_epoch=1000,
  677:
              learning_rate=0.01,
  678:
              loss_fun="cross_entropy",
  679:
              shuffle_flag=True,
  680:
  681:
  682:
          trainer.train(x_train_pre, y_train)
  683:
          print("Train loss = ", trainer.eval_loss(x_train_pre, y_train))
  684:
          print("Validation loss = ", trainer.eval_loss(x_val_pre, y_val))
  685:
  686:
          preds = net(x_val_pre).argmax(axis=1).squeeze()
  687:
          targets = y_val.argmax(axis=1).squeeze()
  688:
          accuracy = (preds == targets).mean()
  689:
          print("Validation accuracy: {}".format(accuracy))
  690:
  691:
  692: if __name__ == "__main__":
```

693:

example_main()

:c3

```
1: ----- Test Output -----
2:
3: PART 1 test output:
4:
5:
6: PART 2 test output:
7:
8:
9: Loaded model in part2_model.pickle
10:
11:
12: Expected RMSE error on the test data: 90000
13: Obtained RMSE error on the test data: 65548.515625
14: Succesfully reached the minimum performance threshold. Well done!
15:
16: ------ Test Errors -------
17:
```

Test Preview TestSummary.txt: 1/1 :c3

```
Test Preview
                         part2_house_value_regression.py: 1/8
                                                                                  :c3
                                                                                             Test Preview
                                                                                                                      part2_house_value_regression.py: 2/8
                                                                                                                                                                               :c3
   1: import copy
                                                                                                67.
                                                                                                           self.x = x
                                                                                                68.
                                                                                                           self.nb epoch = nb epoch
   3: import torch
                                                                                                69:
                                                                                                           self.n hidden = n hidden
   4: import torch.nn as nn
                                                                                                70:
                                                                                                           self.n nodes = n nodes
                                                                                                71:
   6: import pickle
                                                                                                72:
                                                                                                           self.output size = 1
   7: import numpy as np
                                                                                                73:
                                                                                                           if x is not None:
   8: import pandas as pd
                                                                                                74:
                                                                                                               X, = self. preprocessor(x, training=True)
                                                                                                75.
                                                                                                               self.input samples, self.input size = X.shape
  10: from sklearn.utils import shuffle
                                                                                               76.
                                                                                                           ٠٩٥١م
  11: from sklearn.preprocessing import LabelBinarizer, MinMaxScaler
                                                                                               77:
                                                                                                               self.input_samples = 0
  12: from sklearn.metrics import mean squared error
                                                                                               78:
                                                                                                               self.input size = 13
  13: from sklearn.model_selection import GridSearchCV, train_test_split, \
                                                                                               79:
          cross_val_score
                                                                                                80:
                                                                                                           # Pre-set or placeholder model variables
  15: from sklearn.base import BaseEstimator
                                                                                               81 •
                                                                                                           if activations is not None:
                                                                                               82:
                                                                                                               self.activations = activations
  16:
   17: from part2_network import Network
                                                                                                83:
                                                                                                           else:
  18:
                                                                                                84.
                                                                                                               self.activations = [nn.ReLU() for i in range(self.n hidden)]
  19:
                                                                                                85:
  20: class Regressor(BaseEstimator):
                                                                                                86.
                                                                                                           self.n inputs = None
                                                                                               87:
  21:
                                                                                                           self.model = None
          def __init__(self,
  22:
                                                                                                88:
                                                                                                           self.criterion = nn.MSELoss()
   23:
                                                                                                29.
                                                                                                           self.optimiser = optimiser
                      x=None.
   24:
                      nb epoch=500.
                                                                                                90:
   25:
                      n_hidden=1,
                                                                                               91:
                                                                                                           # Optimisation features
   26:
                      n_nodes=None,
                                                                                               92:
                                                                                                           self.early_stopping = early_stopping
   27:
                      activations=None,
                                                                                                93.
                                                                                                           self.earliest_stop = earliest_stop
   28:
                      optimiser=torch.optim.Adam,
                                                                                                94:
                                                                                                           self.patience = patience
   29:
                      early stopping=True,
                                                                                                95.
                                                                                                           self.dropout = dropout
                      earliest stop=100,
                                                                                                96:
   30:
                                                                                               97:
  31:
                      patience=3,
                                                                                                           98:
                                                                                                                                  ** END OF YOUR CODE **
   32:
                      dropout=False):
                                                                                               99:
                                                                                                           33.
              Initialise the model.
                                                                                               100:
  34:
                                                                                               101:
  35:
                                                                                                       def _build_optimiser(self):
  36:
                                                                                               102:
              Arguments:
                                                                                               103:
  37 •
                  - x {pd.DataFrame} -- Raw input data of shape
                                                                                                           Builds an optimiser from the set optimiser types
                                                                                               104:
  38:
                      (batch_size, input_size), used to compute the size
   39:
                      of the network.
                                                                                               105:
                                                                                                           Returns: {nn.optim} instantiated optimiser
   40:
                  - nb_epoch {int} -- number of epoch to train the network.
                                                                                               106:
                  - n_hidden {int} -- number of layers with activation functions,
                                                                                               107:
                                                                                                           # Use given model optimiser
   41:
   42:
                      includes the input layer as it requires activation function
                                                                                               108:
                                                                                                           if self.optimiser == torch.optim.Adam:
                                                                                               109.
   43:
                  - n_nodes {List[int]} -- number of nodes in each layer except
                                                                                                               return self.optimiser(self.model.parameters(),
                                                                                               110:
   44:
                      input layer, that is generated automatically
                                                                                                                                    lr=self.learning_rate,
   45:
                  - activations {List[nn.modules.activation]} -- list of activation
                                                                                               111:
                                                                                                                                    weight_decay=1e-4)
   46:
                      functions for our model
                                                                                               112:
   47:
                  - optimiser {torch.optim} -- optimiser to be used for the model
                                                                                               113:
                                                                                                           # Use classic SGD
   48:
                  - early_stopping {boolean} -- whether early stopping is enabled for
                                                                                               114:
                                                                                                           return self.optimiser(self.model.parameters(), lr=self.learning_rate)
                                                                                               115:
   49:
                      training the model
   50.
                  - earliest_stop {int} -- earliest epoch to begin to consier early
                                                                                               116:
                                                                                                       def _preprocessor(self, x, y=None, training=False):
   51:
                      stopping, used to prevent stopping too early before model starts
                                                                                               117:
   52:
                      converging
                                                                                               118:
                                                                                                           Preprocess input of the network.
   53:
                  - patience {int} -- number of consecutive epochs without a better
                                                                                               119:
                                                                                               120:
  54:
                      model before stopping
                                                                                                           Arguments:
  55:
                  - dropout {boolean} -- whether dropout is enabled during training
                                                                                               121:
                                                                                                               - x {pd.DataFrame} -- Raw input array of shape
  56:
                                                                                               122:
                                                                                                                   (batch_size, input_size).
              ....
   57:
                                                                                               123:
                                                                                                               - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1).
  58:
                                                                                               124:
                                                                                                               - training {boolean} -- Boolean indicating if we are training or
  59:
              125:
                                                                                                                   testing the model.
   60:
                                     ** START OF YOUR CODE **
                                                                                               126:
              *****************************
                                                                                               127:
   61:
                                                                                                           Returns:
   62:
                                                                                               128:
                                                                                                               - {torch.tensor} -- Preprocessed input array of size
   63:
              self.binariser labels = None
                                                                                               129:
                                                                                                                   (batch_size, input_size).
   64:
              self.v scaler = None
                                                                                               130:
                                                                                                               - {torch.tensor} -- Preprocessed target array of size
                                                                                               131:
   65:
              self.learning_rate = 0.01
                                                                                                                   (batch_size, 1).
                                                                                               132:
   66:
```

Test Previe	part2_house_value_regression.py: 3/8	:c3	Test Preview	<pre>part2_house_value_regression.py: 4/8</pre>	:c3
133:	11 11 11		199:	(batch_size, input_size).	
134:			200:	y {pd.DataFrame} Raw output array of shape (batch_size, 1)	•
135:	#######################################	ŧ	201:		
136:	# ** START OF YOUR CODE **		202:	Returns:	
137:	#######################################	ŧ	203:	self {Regressor} Trained model.	
138:			204:		
139:	# Fill NaN values within the numerical columns of the given input data		205:	ппп	
140:	<pre>numerical_x = x.drop("ocean_proximity", axis=1)</pre>		206:		
141:	fill_keys = numerical_x.median().to_dict()		207:	#######################################	####
142:	<pre>numerical_x = numerical_x.fillna(value=fill_keys)</pre>		208:	# ** START OF YOUR CODE **	
143:			209:	#######################################	####
144:	# Normalise numerical values to scale to values between 0 and 1		210:		
145:	<pre>numerical_labels = numerical_x.keys()</pre>		211:	# In order to match sklearn estimator API convention, the model is	built
146:	x_scaler = MinMaxScaler()		212:	# here instead of in the constructor	
147:	numerical_x = pd.DataFrame(data=x_scaler.fit_transform(numerical_x),		213:	if self.n nodes is not None:	
148:	columns=numerical_labels)		214:	self.n_inputs = [self.input_size] + self.n_nodes	
149:	columns-namerical_tabets)		215:	else:	
150:	# Donform one hat enceding on toutual values "legger provincty" replace		216:		
151:	# Perform one-hot encoding on textual values "ocean_proximity" replace		217:	<pre>self.n_inputs = [self.input_size] +\</pre>	
	# with columns of x_ij in [0, 1] for each label			<pre>[self.input_size for i in range(self.n_hidden)]</pre>	
152:	binariser = LabelBinarizer()		218:		
153:	ocean_proximity = x[" ocean_proximity "]		219:	# Neural network model	
154:			220:	self.model = Network(n_layers=self.n_hidden,	
155:	<pre>if training:</pre>		221:	n_inputs=self.n_inputs)	
156:	# Create new binariser parameters for one-hot encoding		222:		
157:	<pre>binarised_data = binariser.fit_transform(ocean_proximity)</pre>		223:	<pre>self.optimiser = selfbuild_optimiser()</pre>	
158:	self.binariser_labels = binariser.classes_		224:		
159:	else:		225:	# Split data into train-validation if early stopping	
160:	# Use existing binariser parameters for one-hot encoding		226:	<pre>if self.early_stopping:</pre>	
161:	binariser.fit(self.binariser labels)		227:	x, val_x, y, val_y = train_test_split(x,	
162:	binarised_data = binariser.transform(ocean_proximity)		228:	y,	
163:	Sinalised_data Sinalisel.elanslelm(esean_plenimie),		229:	train_size=0.8,	
164:	# One-hot encoding parameters		230:	random_state=42)	
165:	op_frame = pd.DataFrame(data=binarised_data,		231:	random_scace=42)	
166:	columns=binariser.classes)		232:	# Programme to detail and detail	
167:	columns=pinariser.classes_)			# Preprocess training data	
			233:	<pre>X_train, Y_train = selfpreprocessor(x, y=y, training=True)</pre>	
168:	# Combine numerical and one-hot encoded textual data frames		234:		
169:	<pre>x = pd.concat([numerical_x, op_frame], axis=1)</pre>		235:	<pre>if self.early_stopping:</pre>	
170:			236:	# Also preprocess validation data	
171:	# Process on CPU, as Lab computers throw CUDA error		237:	<pre>X_val, Y_val = selfpreprocessor(val_x, val_y)</pre>	
172:	<pre>device = torch.device('cpu')</pre>		238:		
173:			239:	# Values for determining and storing optimal model	
174:	# Create tensor from preprocessed x data		240:	best_model = None	
175:	<pre>x_tensor = torch.tensor(x.values, device=device, requires_grad=True)</pre>		241:	$min_loss = 999$	
176:	y_tensor = None		242:	strikes = 0	
177:	- -		243:		
178:	<pre>if isinstance(y, pd.DataFrame):</pre>		244:	# Set model mode to train	
179:	# Normalise y values if available, store scaler to be used to undo		245:	self.model.train()	
180:	# scaling		246:	<pre>for epoch in range(self.nb_epoch):</pre>	
181:	self.y_scaler = MinMaxScaler()		247:	# Clear gradient for this iteration	
182:	y = pd.DataFrame(data=self.y_scaler.fit_transform(y),		248:	self.optimiser.zero_grad()	
183:			240:	Sell.optimiser.zero_grad()	
	columns=y.keys())	,			
184:	<pre>y_tensor = torch.tensor(y.values, device=device, requires_grad=True</pre>	2)	250:	# Forward pass and loss calculation	
185:			251:	<pre>y_predicted = self.model(X_train,</pre>	
186:	# Return preprocessed x and y, return None for y if it was None		252:	self.activations,	
187:	return x_tensor, y_tensor		253:	dropout=self.dropout)	
188:			254:		
189:	#######################################	ŧ	255:	loss = self.criterion(y_predicted, Y_train)	
190:	# ** END OF YOUR CODE **		256:		
191:	#######################################	ŧ	257:	# Backpropagation	
192:			258:	loss.backward()	
	<pre>def fit(self, x, y):</pre>		259:	self.optimiser.step()	
194:	HHH		260:	·- g	
195:	Regressor training function		261:	<pre>if self.early_stopping:</pre>	
196:	Cruining Luncoton		262:	# Predict off validatation data and calculate loss	
197:	Arguments:		263:	<pre>val_y_pred = self.model(X_val, self.activations)</pre>	
198:	<pre>- x {pd.DataFrame} Raw input array of shape</pre>		264:	<pre>val_loss = self.criterion(val_y_pred, Y_val)</pre>	

```
Test Preview
                      part2_house_value_regression.py: 5/8
                                                                        . . . 3
                                                                                  Test Preview
                                                                                                        part2_house_value_regression.py: 6/8
                                                                                                                                                           . . . 3
                                                                                    330:
 265.
                   # print(f'val loss: {val loss:.5f} best loss: {min loss:.5f}')
 266:
                                                                                    331:
                                                                                              Function to evaluate the model accuracy on a validation dataset.
 267:
                   # Determine if training should stop early
                                                                                    332:
 268:
                   if val loss.item() > min loss:
                                                                                    333.
                                                                                              Arguments:
 269.
                      if epoch >= self.earliest_stop:
                                                                                    331.
                                                                                                  - x {pd.DataFrame} -- Raw input array of shape
 270:
                          if strikes == self.patience:
                                                                                    335:
                                                                                                      (batch size, input size).
 271 •
                             # Early stop
                                                                                    336:
                                                                                                  - v {pd.DataFrame} -- Raw ouput array of shape (batch size, 1).
 272:
                             self.model = best model
                                                                                    337:
 273.
                             return self
                                                                                    338.
                                                                                              Returns:
 274 •
                                                                                    339:
                                                                                                  {float} -- Quantification of the efficiency of the model.
                          -1 -- .
 275:
                             strikes += 1
                                                                                    340:
 276:
                   else:
                                                                                    341:
 277:
                       # Clear strikes and save optimal model
                                                                                    342:
 278:
                      strikes = 1
                                                                                    343:
                                                                                               279:
                      min loss = val loss.item()
                                                                                    344 •
                                                                                                                   ** START OF YOUR CODE **
                      best_model = copy.deepcopy(self.model)
                                                                                    3/15 •
                                                                                               ********************************
 280:
 281:
                                                                                    346:
                                                                                    347:
 282 .
                if (epoch + 1) % 10 == 0:
                                                                                              y predicted = self.predict(x)
                   print(f'epoch {epoch + 1} / {self.nb_epoch} loss = /
                                                                                    348:
 283:
                                                                                              mse = mean_squared_error(y, y_predicted)
{loss.item():.8f}')
                                                                                    349:
                                                                                              return np.sgrt (mse)
 284 •
                                                                                    350.
 285:
            if self.early stopping:
                                                                                    351:
                                                                                               286.
                # Return optimal model if available
                                                                                    352.
                                                                                                                   ** END OF YOUR CODE **
 287:
                                                                                               ********************************
                self.model = best model
                                                                                    353:
 288:
            return self
                                                                                    354:
 289.
                                                                                    355:
            290:
                                                                                   356: def save_regressor(trained_model):
 291:
                                 ** END OF YOUR CODE **
                                                                                    357:
 292.
            358:
                                                                                           Utility function to save the trained regressor model in part2_model.pickle.
 293:
                                                                                    359:
                                                                                    360:
 294:
         def predict(self, x):
                                                                                           # If you alter this, make sure it works in tandem with load_regressor
 295:
                                                                                    361:
                                                                                           with open('part2_model.pickle', 'wb') as target:
 296:
            Ouput the value corresponding to an input x.
                                                                                    362:
                                                                                              pickle.dump(trained model, target)
 297:
                                                                                    363:
                                                                                           print("\nSaved model in part2_model.pickle\n")
 298 .
                                                                                    364:
 299:
                x {pd.DataFrame} -- Raw input array of shape
                                                                                    365:
 300:
                                                                                   366: def load_regressor():
                   (batch_size, input_size).
 301:
                                                                                    367:
 302:
                                                                                    368:
                                                                                           Utility function to load the trained regressor model in part2_model.pickle.
 303:
                {np.darray} -- Predicted value for the given input (batch_size, 1).
                                                                                    369:
 304:
                                                                                    370:
                                                                                           # If you alter this, make sure it works in tandem with save regressor
                                                                                    371:
 305:
                                                                                           with open('part2_model.pickle', 'rb') as target:
 306:
                                                                                    372:
                                                                                              trained_model = pickle.load(target)
 307:
            373:
                                                                                           print("\nLoaded model in part2_model.pickle\n")
 308:
                                 ** START OF YOUR CODE **
                                                                                    374:
                                                                                           return trained model
 309:
            375:
 310:
                                                                                    376:
 311 •
            if self.model is not None:
                                                                                    377: def RegressorHyperParameterSearch(x, y):
 312:
                                                                                    378:
               X, _ = self._preprocessor(x, training=False) # Do not forget
                                                                                           # Ensure to add whatever inputs you deem necessary to this function
 313:
                                                                                    379:
 314 •
                # Set model to evaluate mode
                                                                                    380:
                                                                                           Performs a hyper-parameter for fine-tuning the regressor implemented
 315:
                self.model.eval()
                                                                                    381:
                                                                                           in the Regressor class.
 316:
                                                                                    382:
 317:
                # Predict from input and scale result using y scaler from training
                                                                                    383:
                                                                                           Arguments:
 318:
                y_predicted = self.model(X, self.activations).detach().numpy()
                                                                                    384 •
                                                                                               - x {pd.DataFrame} -- Raw input array of shape
 319:
                y_predicted_scaled = self.y_scaler.inverse_transform(y_predicted)
                                                                                    385:
                                                                                                  (batch_size, input_size).
 320:
                                                                                    386:
                                                                                               - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1)
 321 •
                return y_predicted_scaled
                                                                                    387:
 322:
                                                                                    388.
                                                                                           Returns:
 323:
            return None
                                                                                    389:
                                                                                              - {dict{str, any}} optimised hyper-parameters.
 324:
                                                                                    390:
            391:
 325:
 326:
                                 ** END OF YOUR CODE **
                                                                                    392:
 327:
            393:
                                                                                           328:
                                                                                   394:
                                                                                                                ** START OF YOUR CODE **
                                                                                           *****************************
 329:
                                                                                    395:
         def score(self, x, y):
```

Test Preview

```
396:
 397:
          param grid = [
 398:
 399.
                  'n hidden': [1],
 400:
                  'n_nodes': [[13], [24], [32]],
 401:
                  'activations': [
 402:
                     [nn.Sigmoid()],
 403.
                     [nn.ReLU()],
 404:
                     [nn.Tanh()]
 405:
 406:
             },
 407:
                  'n_hidden': [2],
 408:
 409:
                  'n_nodes': [[13, 13], [20, 20], [24, 32], [24, 64]],
 410:
                  'activations': [
 411:
                     [nn.Sigmoid(), nn.Sigmoid()],
 412:
                     [nn.ReLU(), nn.ReLU()],
 413:
                     [nn.Tanh(), nn.Tanh()]
 414:
 415:
             },
 416:
 417:
                  'n hidden': [3],
 418:
                  'n_nodes': [[13, 13, 13], [32, 64, 32], [64, 128, 64], [512, 512, \( \mu \)
12811.
 419:
                  'activations': [
 420:
                     [nn.Sigmoid(), nn.Sigmoid(), nn.Sigmoid()],
 421 •
                     [nn.Sigmoid(), nn.ReLU(), nn.ReLU()],
 422:
                     [nn.ReLU(), nn.ReLU(), nn.ReLU()],
 423:
                     [nn.ReLU(), nn.Sigmoid(), nn.ReLU()]
 424:
 425:
                  'patience': [1, 3, 5, 8],
 426:
                  'dropout': [True, False]
 427:
 428:
 429:
                  'n hidden': [4],
 430:
                  'n_nodes': [[32, 64, 64, 32]],
 431 •
                  'activations': [
 432 •
                     [nn.Sigmoid(), nn.Tanh(), nn.ReLU(), nn.Sigmoid()],
 433:
                     [nn.ReLU(), nn.ReLU(), nn.ReLU(), nn.ReLU()]
 434.
 435:
             },
 436:
 437:
 438:
          regressor = Regressor()
 439:
          grid_search = GridSearchCV(regressor,
 440:
                                   param_grid,
 441:
 442 •
                                   scoring="neg_mean_squared_error";
 443:
 444:
          grid_search.fit(x, y)
 445:
 446:
          print(grid search.best score )
 447:
          print (grid_search.best_params_)
 448:
          print (grid_search.best_estimator_)
 449:
 450:
          cvres = grid_search.cv_results_
 451:
 452 •
          for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
 453:
             print (np.sqrt (-mean_score), params)
 454:
 455:
          # Return the chosen hyper parameters
 456:
          return grid_search.best_params_
 457:
 458:
          459:
                                 ** END OF YOUR CODE **
          460:
```

```
part2_house_value_regression.py: 8/8
461.
462:
463: def example_main():
464:
         output_label = "median_house_value"
465:
466:
         # Use pandas to read CSV data as it contains various object types
467:
         # Feel free to use another CSV reader tool
468:
         # But remember that LabTS tests take Pandas Dataframe as inputs
469:
         data = pd.read csv("housing.csv")
470:
471:
         # Set manual seed for result replication
472:
         torch.manual seed(42)
473:
474:
         # Shuffle data
475 •
         data = shuffle(data, random_state=42)
476:
         data.reset_index(inplace=True, drop=True)
477:
478:
         # Spliting input and output
479:
         x raw = data.loc[:, data.columns != output label]
480:
         y raw = data.loc[:, [output label]]
481:
482:
         x_train, x_test, y_train, y_test \
483:
             = train test split(x raw, y raw, test size=0.2, random state=42)
484 •
485:
         # Training
486:
         # Build Regressor
487 •
         acitivations = [nn.ReLU(), nn.ReLU(), nn.ReLU()]
488:
         n nodes = [512, 512, 128]
489:
         regressor = Regressor(x train,
490:
                               nb epoch=500,
491:
                               n hidden=3,
492:
                               n_nodes=n_nodes,
493:
                               activations=acitivations,
494:
                               patience=5.
495:
                               dropout=False)
496:
         # regressor.fit(x_train, y_train)
497:
         # save_regressor(regressor)
498:
499:
         # Perform cross validation and obtain average error
500:
         nmse score = -cross val score(regressor,
501:
                                       x raw,
502:
                                       v raw.
503:
                                       scoring="neg_mean_squared_error",
504:
                                       cv=5)
505:
506:
         scores = np.sgrt(nmse score)
507:
508:
         print("Scores:", scores)
509:
         print("Mean:", scores.mean())
510:
         print("Standard Deviation:", scores.std())
511:
512:
         # RegressorHyperParameterSearch(x raw, y raw)
513:
514:
515:
         # error = regressor.score(x_train, y_train)
516:
         # print("\nRegressor error: {}\n".format(error))
517:
518:
         # error = regressor.score(x_test, y_test)
519:
         # print("\nRegressor error: {}\n".format(error))
520:
521:
522: if __name__ == "__main__":
         example main()
```

```
Test Preview
                                part1_nn_lib.py: 1/11
                                                                                 :c3
                                                                                           Test Preview
                                                                                                                           part1_nn_lib.py: 2/11
                                                                                                                                                                            :c3
                                                                                              67:
   1: import numpy as np
   2: import pickle
                                                                                              68:
                                                                                                     CrossEntropyLossLayer: Computes the softmax followed by the negative
   3 •
                                                                                              69:
                                                                                                     log-likelihood loss.
   4:
                                                                                              70:
   5: def xavier_init(size, gain = 1.0):
                                                                                              71:
   6:
                                                                                              72:
                                                                                                     def ___init___(self):
   7:
          Xavier initialization of network weights.
                                                                                              73:
                                                                                                         self. cache current = None
   8:
                                                                                              74:
   9:
                                                                                              75:
                                                                                                     @staticmethod
          Arguments:
                                                                                              76:
   10:
              - size {tuple} -- size of the network to initialise.
                                                                                                     def softmax(x):
  11.
              - gain {float} -- gain for the Xavier initialisation.
                                                                                              77:
                                                                                                         numer = np.exp(x - x.max(axis=1, keepdims=True))
  12:
                                                                                              78:
                                                                                                         denom = numer.sum(axis=1, keepdims=True)
  13:
          Returns:
                                                                                              79:
                                                                                                         return numer / denom
              {np.ndarray} -- values of the weights.
   14:
                                                                                              80:
   15:
                                                                                              81:
                                                                                                     def forward(self, inputs, y_target):
                                                                                              82:
          low = -gain * np.sqrt(6.0 / np.sum(size))
  16:
                                                                                                         assert len(inputs) == len(y_target)
  17:
                                                                                              83:
          high = gain * np.sqrt(6.0 / np.sum(size))
                                                                                                         n_obs = len(y_target)
                                                                                              84:
   18:
          return np.random.uniform(low=low, high=high, size=size)
                                                                                                         probs = self.softmax(inputs)
   19:
                                                                                              85:
                                                                                                         self._cache_current = y_target, probs
  20:
                                                                                              86.
  21: class Layer:
                                                                                              87:
                                                                                                         out = -1 / n obs * np.sum(y target * np.log(probs))
  22:
                                                                                              88:
                                                                                                         return out
   23:
          Abstract layer class.
                                                                                              89.
   24:
                                                                                              90:
                                                                                                     def backward(self):
   25:
                                                                                              91:
                                                                                                         y_target, probs = self._cache_current
   26:
          def __init__(self, *args, **kwargs):
                                                                                              92 .
                                                                                                         n_obs = len(y_target)
                                                                                              93:
   27:
              raise NotImplementedError()
                                                                                                         return -1 / n_obs * (y_target - probs)
   28:
                                                                                              94:
   29:
          def forward(self, *args, **kwargs):
                                                                                              95:
   30:
              raise NotImplementedError()
                                                                                              96: class SigmoidLayer(Layer):
   31:
                                                                                              97:
                                                                                              98:
                                                                                                     SigmoidLayer: Applies sigmoid function elementwise.
   32:
          def __call__(self, *args, **kwargs):
   33:
              return self.forward(*args, **kwargs)
                                                                                              99:
  34:
                                                                                             100:
                                                                                                     def __init__(self):
   35:
          def backward(self, *args, **kwargs):
                                                                                             101:
   36:
              raise NotImplementedError()
                                                                                             102:
   37:
                                                                                             103:
                                                                                                         Constructor of the Sigmoid layer.
   38:
                                                                                             104:
          def update_params(self, *args, **kwargs):
   39:
              pass
                                                                                             105:
                                                                                                         self. cache current = None
   40:
                                                                                             106:
   41:
                                                                                             107:
                                                                                                     def forward(self, x):
   42: class MSELossLayer(Layer):
                                                                                             108:
   43:
                                                                                             109:
                                                                                                         Performs forward pass through the Sigmoid layer.
   44:
          MSELossLayer: Computes mean-squared error between y_pred and y_target.
                                                                                             110:
   45:
                                                                                             111:
                                                                                                         Logs information needed to compute gradient at a later stage in
   46:
                                                                                             112:
                                                                                                          ' cache current'.
   47:
          def ___init___(self):
                                                                                             113:
   48 •
              self._cache_current = None
                                                                                             114:
                                                                                                         Arguments:
                                                                                             115:
   49:
                                                                                                             x {np.ndarray} -- Input array of shape (batch_size, n_in).
   50.
          @staticmethod
                                                                                             116:
   51:
          def _mse(y_pred, y_target):
                                                                                             117:
   52:
              return np.mean((y_pred - y_target) ** 2)
                                                                                             118:
                                                                                                             {np.ndarray} -- Output array of shape (batch_size, n_out)
   53:
                                                                                             119:
   54:
                                                                                             120:
                                                                                                         @staticmethod
   55:
          def _mse_grad(y_pred, y_target):
                                                                                             121:
                                                                                                                                ** START OF YOUR CODE **
   56:
              return 2 * (y_pred - y_target) / len(y_pred)
                                                                                             122:
                                                                                                         57:
                                                                                             123:
                                                                                                         sigmoid = 1 / (1 + np.exp(-x))
   58:
          def forward(self, y_pred, y_target):
                                                                                             124:
                                                                                                         self._cache_current = sigmoid
   59:
                                                                                             125:
              self._cache_current = y_pred, y_target
   60:
              return self._mse(y_pred, y_target)
                                                                                             126:
                                                                                                         return sigmoid
   61:
                                                                                             127:
                                                                                                         62:
          def backward(self):
                                                                                             128:
   63:
              return self. mse grad(*self. cache current)
                                                                                             129:
                                                                                                                                ** END OF YOUR CODE **
                                                                                                         64:
                                                                                             130:
                                                                                             131:
   65:
                                                                                             132:
   66: class CrossEntropyLossLayer(Layer):
                                                                                                     def backward(self, grad_z):
```

```
Test Preview
                         part1_nn_lib.py: 3/11
                                                                . . . 3
                                                                        Test Preview
                                                                                                 part1_nn_lib.py: 4/11
                                                                                                                                        :c3
                                                                         199:
 133.
                                                                                   inputs of layer).
 134:
           Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
                                                                         200:
 135:
           the output of this layer, performs back pass through the layer (i.e.
                                                                         201:
                                                                                   Arguments:
 136:
           computes gradients of loss with respect to parameters of layer and
                                                                         202:
                                                                                      grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
 137:
                                                                         203:
           inputs of layer).
 138:
                                                                         204:
                                                                                   Returns:
 139:
           Arguments:
                                                                         205:
                                                                                      {np.ndarray} -- Array containing gradient with repect to layer
 140:
              grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
                                                                         206:
                                                                                         input, of shape (batch_size, n_in).
 141:
                                                                         207 •
 142:
                                                                         208.
                                                                                   143.
              {np.ndarray} -- Array containing gradient with repect to layer
                                                                         209:
                                                                                                     ** START OF YOUR CODE **
 144:
                 input, of shape (batch_size, n_in).
                                                                         210:
                                                                                   ....
 145:
                                                                         211:
                                                                                   z = self. cache current
 146:
           212:
                                                                                   relu_derivative = np.int64(z > 0)
 147:
                             ** START OF YOUR CODE **
                                                                         213:
           214:
 148 •
                                                                                   return grad_z * relu_derivative
 149:
                                                                         215:
           a = self._cache_current
 150:
           sigmoid derivative = a * (1 - a)
                                                                         216:
                                                                                   217:
                                                                                                     ** END OF YOUR CODE **
 151:
 152:
           return grad z * sigmoid derivative
                                                                         218.
                                                                                   153.
                                                                         219.
           154 •
                                                                         220:
 155:
                             ** END OF YOUR CODE **
                                                                         221: class LinearLayer(Layer):
 156:
           222:
 157:
                                                                         223:
                                                                                LinearLayer: Performs affine transformation of input.
 158 •
                                                                         224:
 159: class ReluLayer(Layer):
                                                                         225:
 160:
                                                                         226:
                                                                                def __init__(self, n_in, n_out):
                                                                         227:
 161:
        ReluLayer: Applies Relu function elementwise.
 162:
                                                                         228:
                                                                                   Constructor of the linear layer.
 163:
                                                                         229:
                                                                         230:
 164:
        def __init__(self):
                                                                                   Arguments:
 165:
                                                                         231:
                                                                                      - n_in {int} -- Number (or dimension) of inputs.
 166:
                                                                         232:
                                                                                      - n_out {int} -- Number (or dimension) of outputs.
           Constructor of the Relu layer.
 167:
                                                                         233:
 168:
                                                                         234:
                                                                                   self.n_in = n_in
           self._cache_current = None
 1.69:
                                                                         235.
                                                                                   self.n_out = n_out
 170:
                                                                         236:
        def forward(self, x):
 171:
                                                                         237:
                                                                                   172:
           Performs forward pass through the Relu layer.
                                                                         238:
                                                                                                     ** START OF YOUR CODE **
 173:
                                                                         239:
                                                                                   174:
           Logs information needed to compute gradient at a later stage in
                                                                         240:
                                                                                   self._W = xavier_init((n_in, n_out))
 175:
                                                                         241:
           `_cache_current`.
                                                                                   self._b = np.zeros(n_out)
 176:
                                                                         242:
 177:
           Arguments:
                                                                         243:
                                                                                   self._cache_current = None
 178:
              x {np.ndarray} -- Input array of shape (batch size, n in).
                                                                         244 •
                                                                                   self. grad W current = None
 179:
                                                                         245:
                                                                                   self._grad_b_current = None
 180:
           Returns
                                                                         246:
 181:
                                                                         247:
                                                                                   ***************************
             {np.ndarray} -- Output array of shape (batch_size, n_out)
 182:
                                                                         248:
                                                                                                     ** END OF YOUR CODE **
 183:
           249:
                                                                                   184:
                             ** START OF YOUR CODE **
                                                                         250:
 185:
           251:
                                                                                def forward(self, x):
 186:
                                                                         252:
           self._cache_current = x
 187:
                                                                         253:
                                                                                   Performs forward pass through the layer (i.e. returns Wx + b).
 188:
           return np.maximum(0, x)
                                                                         254:
 189:
                                                                         255:
                                                                                   Logs information needed to compute gradient at a later stage in
 190:
           256:
                                                                                   `_cache_current`.
 191 •
                             ** END OF YOUR CODE **
                                                                         257:
 192:
           258:
                                                                                   Arguments:
 193:
                                                                         259:
                                                                                      x {np.ndarray} -- Input array of shape (batch_size, n_in).
 194:
        def backward(self, grad_z):
                                                                         260:
 195:
                                                                         261:
 196:
           Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
                                                                         262:
                                                                                      {np.ndarray} -- Output array of shape (batch_size, n_out)
 197:
           the output of this layer, performs back pass through the layer (i.e.
                                                                         263:
 198 •
           computes gradients of loss with respect to parameters of layer and
                                                                         264:
```

Test Preview

```
265.
                          ** START OF YOUR CODE **
266:
         267:
         self. cache current = x
268:
         return x @ self. W + self. b
269.
270:
         271:
                          ** END OF YOUR CODE **
272:
         273:
274:
      def backward(self, grad z):
275:
276:
         Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
277:
         the output of this layer, performs back pass through the layer (i.e.
278:
         computes gradients of loss with respect to parameters of layer and
279.
         inputs of laver).
280:
281:
         Arguments:
282:
            grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
283:
284:
285.
            {np.ndarray} -- Array containing gradient with repect to layer
286:
               input, of shape (batch_size, n_in).
         .....
287 •
         288 .
289:
                          ** START OF YOUR CODE **
290:
         291 •
         self._grad_W_current = self._cache_current.transpose() @ grad_z
292:
         self._grad_b_current = np.ones((1, np.shape(grad_z)[0])) @ grad_z
293.
         return grad z @ self. W.transpose()
294:
         295:
296:
                          ** END OF YOUR CODE **
297:
         298:
299.
      def update_params(self, learning_rate):
300:
301:
         Performs one step of gradient descent with given learning rate on the
302:
         layer's parameters using currently stored gradients.
303:
304:
         Arguments:
305:
           learning_rate {float} -- Learning rate of update step.
306:
307:
         ******************************
308:
                          ** START OF YOUR CODE **
         309:
310:
         self._W = self._W - learning_rate * self._grad_W_current
311 •
         self._b = self._b - learning_rate * self._grad_b_current
312:
313:
         314:
                          ** END OF YOUR CODE **
315:
         316:
317:
318: class MultiLayerNetwork(object):
319:
320:
      MultiLayerNetwork: A network consisting of stacked linear layers and
321:
      activation functions.
322:
323:
324:
      def __init__(self, input_dim, neurons, activations):
325:
326:
         Constructor of the multi layer network.
327:
328:
329:
            - input_dim {int} -- Number of features in the input (excluding
330:
              the batch dimension).
```

```
331: Â Â Â Â Â A - neurons (list) -- Number of neurons in each linear layer
               represented as a list. The length of the list determines the
                number of linear lavers.
334 :
            - activations {list} -- List of the activation functions to apply
335.
               to the output of each linear layer.
336:
337:
         self.input dim = input dim
338:
         self.neurons = neurons
339.
         self.activations = activations
340:
341:
          342:
                            ** START OF YOUR CODE **
343:
          344:
         ACTIVATIONS_MAP = {
3/5.
             "sigmoid": lambda size: SigmoidLayer(),
3/6.
            "relu": lambda size: ReluLayer(),
347:
             "identity": lambda size: LinearLaver(size, size)
348:
349:
350.
          self. layers = []
351:
         prev dim = input dim
352:
         for num neurons, activation in zip(neurons, activations):
353.
            self. layers.append(LinearLayer(prev dim, num neurons))
            self. lavers.append(ACTIVATIONS_MAP[activation](num_neurons))
354 •
355:
            prev_dim = num_neurons
356:
          357:
358:
                            ** END OF YOUR CODE **
359.
          360:
361:
      def forward(self, x):
362:
363:
         Performs forward pass through the network.
364:
365:
         Arguments:
366:
            x {np.ndarray} -- Input array of shape (batch_size, input_dim).
367:
368:
         Returns:
369:
             {np.ndarray} -- Output array of shape (batch_size,
370:
                #_neurons_in_final_layer)
371:
372:
          373:
                            ** START OF YOUR CODE **
374:
          375:
         prev = x
376:
          for laver in self. lavers:
377:
            prev = layer.forward(prev)
378:
         return prev
379:
380:
          381:
                            ** END OF YOUR CODE **
382:
          383:
384:
      def __call__(self, x):
385:
         return self.forward(x)
386:
387:
      def backward(self, grad_z):
388:
389:
         Performs backward pass through the network.
390:
391:
         Arguments:
392:
            grad_z {np.ndarray} -- Gradient array of shape (1,
393:
                #_neurons_in_final_layer).
394:
395:
         Returns:
396:
            {np.ndarray} -- Array containing gradient with repect to layer
```

part1_nn_lib.py: 6/11

Test Preview

397:

398:

399:

400:

401:

402:

403:

404.

405:

406:

407:

408.

416:

417:

418:

419.

420:

421:

422:

423:

424:

425 •

426:

427:

428:

429:

446.

462:

part1_nn_lib.py: 7/11

** END OF YOUR CODE **

** START OF YOUR CODE **

input, of shape (batch_size, input_dim).

```
** START OF YOUR CODE **
for layer in self._layers:
 layer.update_params(learning_rate)
```

```
** END OF YOUR CODE **
```

```
430: def save_network(network, fpath):
431 •
432:
433:
```

prev = grad z

return prev

for layer in reversed(self._layers):

prev = layer.backward(prev)

Utility function to pickle 'network' at file path 'fpath'.

```
434 •
         with open(fpath, "wb") as f:
435:
             pickle.dump(network, f)
436:
437:
```

439: 440: Utility function to load network found at file path 'fpath'.

```
441:
442 •
         with open(fpath, "rb") as f:
443:
             network = pickle.load(f)
444 •
         return network
445:
```

438: def load_network(fpath):

447: class Trainer (object): 448: 449: Trainer: Object that manages the training of a neural network.

450: 451: 452: def __init__(

```
453:
             self,
454 •
             network,
455:
             batch_size,
456:
             nb epoch.
457:
             learning_rate,
458:
             loss fun,
459:
             shuffle flag,
460:
461:
```

```
463:
464:
             Arguments:
                 - network {MultiLaverNetwork} -- MultiLaverNetwork to be trained.
```

465: 466: - batch_size {int} -- Training batch size. 167. - nb_epoch {int} -- Number of training epochs.

468: - learning rate {float} -- SGD learning rate to be used in training. 469: - loss fun {str} -- Loss function to be used. Possible values: mse,

470: 471 • - shuffle_flaq {bool} -- If True, training data is shuffled before 472. training.

```
473:
474.
          self.network = network
475:
          self.batch size = batch size
476:
          self.nb_epoch = nb_epoch
477:
          self.learning_rate = learning_rate
178.
          self.loss_fun = loss_fun
479:
          self.shuffle_flag = shuffle_flag
480 •
481:
          482 .
483:
```

** START OF YOUR CODE ** LOSS FUN MAP = { "mse": MSELossLayer, "cross_entropy": CrossEntropyLossLayer,

487: 488 . 489: self._loss_layer = LOSS_FUN_MAP[loss_fun]() 490: 491: ** END OF YOUR CODE ** 492: 493:

494: @staticmethod 495: def shuffle(input_dataset, target_dataset): 496:

Returns shuffled versions of the inputs.

Arguments:

- input_dataset {np.ndarray} -- Array of input features, of shape (#_data_points, n_features).
- target_dataset {np.ndarray} -- Array of corresponding targets, of shape (#_data_points, #output_neurons).

Returns:

- {np.ndarray} -- shuffled inputs.
- {np.ndarray} -- shuffled_targets.

** START OF YOUR CODE ** *************************** assert len(input dataset) == len(target dataset) perm = np.random.permutation(len(input dataset)) return input dataset[perm], target dataset[perm]

** END OF YOUR CODE **

def train(self, input_dataset, target_dataset):

Main training loop. Performs the following steps 'nb_epoch' times:

- Shuffles the input data (if 'shuffle' is True)
- Splits the dataset into batches of size 'batch_size'.
- For each batch:
- Performs forward pass through the network given the current batch of inputs.
 - Computes loss.

Constructor of the Trainer.

. . . 3

Test Preview

484:

485.

486:

497:

498:

499 .

500:

501:

502:

503:

504:

505:

506:

507:

508:

509:

510:

511:

512:

513.

514:

515:

516:

517:

518:

519:

520:

521:

522:

523:

524:

525:

526:

527:

528:

```
Test Preview
                        part1_nn_lib.py: 9/11
                                                               :c3
                                                                       Test Preview
                                                                                               part1_nn_lib.py: 10/11
                                                                                                                                      :c3
                                                                        595:
 529 •
                - Performs backward pass to compute gradients of loss with
 530 •
                respect to parameters of network.
                                                                        596:
                                                                                  Arguments:
 531 •
                - Performs one step of gradient descent on the network
                                                                        597:
                                                                                     data {np.ndarray} dataset used to determine the parameters for
 532:
                                                                        598 .
                                                                                     the normalization.
                parameters.
                                                                        599.
 533:
 534 •
                                                                        600:
                                                                                  *******************************
          Arguments:
 535:
             - input_dataset {np.ndarray} -- Array of input features, of shape
                                                                        601:
                                                                                                   ** START OF YOUR CODE **
 536.
                 (#_training_data_points, n_features).
                                                                        602.
                                                                                  537:
              - target_dataset {np.ndarray} -- Array of corresponding targets, of
                                                                        603:
                                                                                  self.min = np.min(data, axis=0)
                                                                        604:
 538:
                shape (#_training_data_points, #output_neurons).
                                                                                  self.max min diff = np.max(data, axis=0) - self.min
 539.
                                                                        605.
 540:
           606:
                                                                                  541:
                            ** START OF YOUR CODE **
                                                                        607:
                                                                                                   ** END OF YOUR CODE **
 542:
           608:
                                                                                  543:
           for i in range(self.nb epoch):
                                                                        609:
 544:
                                                                        610:
             if self.shuffle_flag:
                                                                               def apply(self, data):
 545:
                                                                        611:
                input_dataset, target_dataset = self.shuffle(
 546:
                                                                        612:
                   input dataset,
                                                                                  Apply the pre-processing operations to the provided dataset.
 547:
                   target dataset
                                                                        613:
 548:
                                                                        614:
 549:
                                                                        615:
                                                                                     data {np.ndarray} dataset to be normalized.
 550.
                                                                        616.
             input_batches = np.array_split(input_dataset, self.batch_size)
 551:
             target batches = np.array split(target dataset, self.batch size)
                                                                        617:
                                                                                  Returns:
 552:
                                                                        618:
                                                                                     {np.ndarray} normalized dataset.
 553:
             for input_batch, target_batch in zip(input_batches, target_batches):
                                                                        619:
 554 •
                network_output = self.network(input_batch)
                                                                        620:
                                                                                  555:
                self._loss_layer(network_output, target_batch)
                                                                        621:
                                                                                                   ** START OF YOUR CODE **
 556:
                grad_loss_wrt_outputs = self._loss_layer.backward()
                                                                        622:
                                                                                  557:
                self.network.backward(grad loss wrt outputs)
                                                                        623:
                                                                                  return (data - self.min) / self.max min diff
 558:
                self.network.update_params(self.learning_rate)
                                                                        624:
 559:
                                                                        625:
                                                                                  ** END OF YOUR CODE **
 560:
           626:
 561:
                            ** END OF YOUR CODE **
                                                                        627:
                                                                                  628:
 562:
 563:
                                                                        629:
                                                                               def revert(self, data):
 564:
        def eval_loss(self, input_dataset, target_dataset):
                                                                        630:
 565:
                                                                        631 •
                                                                                  Revert the pre-processing operations to retreive the original dataset.
 566:
          Function that evaluate the loss function for given data.
                                                                        632:
 567:
                                                                        633:
 568:
                                                                        634:
          Arguments:
                                                                                     data {np.ndarray} dataset for which to revert normalization.
 569:
                                                                        635:
              - input_dataset {np.ndarray} -- Array of input features, of shape
 570:
                 (#_evaluation_data_points, n_features).
                                                                        636:
                                                                                  Returns:
 571:
             - target_dataset {np.ndarray} -- Array of corresponding targets, of
                                                                        637:
                                                                                     {np.ndarray} reverted dataset.
 572:
                shape (#_evaluation_data_points, #output_neurons).
                                                                        638:
           ....
 573:
                                                                        639:
                                                                                  574 •
           640:
                                                                                                   ** START OF YOUR CODE **
 575:
                            ** START OF YOUR CODE **
                                                                        641:
                                                                                  576:
           642:
                                                                                  return data * self.max_min_diff + self.min
 577:
          network_output = self.network(input_dataset)
                                                                        643:
 578 •
          return self._loss_layer(network_output, target_dataset)
                                                                        644 •
                                                                                  579:
                                                                        645:
                                                                                                   ** END OF YOUR CODE **
 580:
           646:
                                                                                  581:
                            ** END OF YOUR CODE **
                                                                        647:
 582:
           648:
 583:
                                                                        649: def example_main():
 584:
                                                                        650:
                                                                               input dim = 4
 585: class Preprocessor(object):
                                                                        651:
                                                                               neurons = [16, 3]
 586:
                                                                        652:
                                                                               activations = ["relu", "identity"]
 587:
       Preprocessor: Object used to apply "preprocessing" operation to datasets.
                                                                        653:
                                                                               net = MultiLayerNetwork(input_dim, neurons, activations)
 588:
                                                                        654:
       The object can also be used to revert the changes.
 589:
                                                                        655:
                                                                               dat = np.loadtxt("iris.dat")
 590:
                                                                        656:
                                                                               np.random.shuffle(dat)
 591:
        def __init__(self, data):
                                                                        657:
 592:
                                                                        658:
                                                                               x = dat[:, :4]
 593:
          Initializes the Preprocessor according to the provided dataset.
                                                                        659:
                                                                               y = dat[:, 4:]
 594 •
           (Does not modify the dataset.)
                                                                        660:
```

```
Test Preview
                                part1_nn_lib.py: 11/11
 661:
          split_idx = int(0.8 * len(x))
 662:
 663:
          x_train = x[:split_idx]
 664:
          y_train = y[:split_idx]
 665:
          x_val = x[split_idx:]
 666:
          y_val = y[split_idx:]
 667:
  668:
          prep_input = Preprocessor(x_train)
 669:
 670:
          x_train_pre = prep_input.apply(x_train)
 671:
          x_val_pre = prep_input.apply(x_val)
 672:
 673:
          trainer = Trainer(
  674:
              network=net,
 675:
              batch_size=8,
 676:
              nb_epoch=1000,
 677:
              learning_rate=0.01,
  678:
              loss_fun="cross_entropy",
  679:
              shuffle_flag=True,
 680:
 681:
 682:
          trainer.train(x_train_pre, y_train)
 683:
          print("Train loss = ", trainer.eval_loss(x_train_pre, y_train))
 684:
          print("Validation loss = ", trainer.eval_loss(x_val_pre, y_val))
 685:
 686:
          preds = net(x_val_pre).argmax(axis=1).squeeze()
 687:
          targets = y_val.argmax(axis=1).squeeze()
 688:
          accuracy = (preds == targets).mean()
 689:
          print("Validation accuracy: {}".format(accuracy))
 690:
 691:
```

692: **if** __name__ == "__**main__**":

example_main()

693:

:c3

```
1: ----- Test Output -----
2:
3:
4: PART 1 test output:
5:
6:
7: PART 2 test output:
8:
9: Loaded model in part2_model.pickle
10:
11:
12: Expected RMSE error on the training data: 90000
13: Obtained RMSE error on the training data: 65367.3046875
14: Succesfully reached the minimum performance threshold. Well done!
15:
16: ------ Test Errors -------
17:
```

Introduction to ML - Artificial Neural Networks Report

rh4618, kd120, ad5518, prm2418

November 27, 2020

1 Regressor Architecture & Methodology

1.1 The Preprocessor Design

The preprocessor is designed to perform preprocessing on raw data for both training and prediction. For input raw features x, the first step is to fill missing values in numerical data in each column(feature) with the medians of the present instances of said feature, using Pandas function fillna. Then all the numerical feature values are normalised to values between 0 and 1, using sklearn.preprocessing's MinMaxScaler.

Special treatment is given to the *ocean_proximity* feature column, as it contains textual data. One-hot encoding is performed using sklearn.preprocessing's LabelBinarizer tool. First mapping each unique textual value to a list of 0s and 1s, then replacing the original column with new columns each corresponding to an encoding. The reason we do this is to dissociate the magnitude of the resulting mapped variable from the result. A possible alternative would have been to use integer mapping to map *ocean_proximity* values by distance to the ocean, though the labels in the given dataset can be a little difficult to discern an ordering and one-hot encoding was sufficient.

Finally, for input labels y, the same normalization method is applied using MinMaxScaler. We did this due to value of y being quite large by default, and we wanted to avoid having large weight values within our model, thus we down-scale the y values in our preprocessor, and up-scale the predicted value.

The values generated in the preprocessor, such as the mapping used in LabelBinarizer and the scalers used are then stored in the Regressor object to ensure consistency when training and using the model.

1.2 The Neural Network Architecture

The Network class extends the nn. Module class to allow for custom constructions of neural network models. It is designed to be flexible, allowing us to build networks of differing number of hidden layers and activation functions. Additionally, we are able to use it seamlessly with Pytorch's loss function and optimiser components to train the model.

The Network class takes a number of layers and a list of number of neurons in each layer. A torch tensor is generated for each entry, an output layer is constructed separately, before all created layers are added as the Network's parameters. The tensor layers are added using nn.ModuleList(layers) to the object to ensure that the layers are individually added as parameters of the module in adherence to Pytorch conventions.

For the forward pass, the Network class overwrites nn.Module's default forward() function, taking the input data, a list of activation functions and an optional boolean for whether we perform dropout on our input and

hidden layers. The input layer is applied to the data, and its corresponding activation function is applied, the result is then propagated through the network with each hidden layer and its activation function being applied in turn. Finally, the output layer is applied, without an explicit activation function, to preserve linearity as the given task is prediction over an unbounded space.

1.3 The Training Process

The Regressor class's fit() function includes all the training functionalities. For the sake of performing grid search, which we'll discuss further on, the model and its related loss function and optimiser are defined here, such that we create a new model every time we fit to new data.

We first call _preprocessor() on the given data, if early stopping is enabled, we split the given data into training and validation sets in a 4:1 split. The model is set to training mode, and for nb_epoch number of iterations, it performs a forward pass using the training data, calculates its loss from the resulting prediction and calls optimiser.step() to update the model weights. By default, our models use the torch.optim.Adam optimiser which implements an adaptive gradient descent strategy, additionally related to section 2.2.1.

Early stopping is also implemented during the training process, the specifics will be discussed in section 2.2.3.

1.4 The Regressor Architecture

Our Regressor class extends sklearn.base.BaseEstimator, which allows for it to be used as a sklearn estimator in both performing grid search using sklearn.model_selection.GridSearchCV or cross-validation using sklearn.model_selection.cross_val_score(). The significance of this is shown when we talk about fine tuning our model.

The impact of implementing our Regressor in this way is that our model instance is not created in the constructor as it was initially, but rather, in the fit() function. This allows the Regressor to create a new model every time it is fitted to data, which allows GridSearchCV to create new instances of the Regressor class by cloning another instance and changing the parameters. Otherwise the old model would also be copied over.

2 Evaluation & Results

2.1 Evaluating a Model

To evaluate the performance of a particular approach, we perform 5 fold cross-validation using sklearn's model_selection.cross_val_score(). To evaluate the performance of a particular model, we call score() which in turn calls predict() which performs a single forward pass through the neural network and then calculates the RMSE of the result.

2.2 Methods to Prevent Overfitting

During the initial phases of our implementation, we saw a high performance from our model on the training data set, yet a very low performance (high RMSE value) on the test data set. This is emblematic of overfitting to the training data, thus we implemented measures to prevent overfitting and allow the model to generalise better to unseen data.

2.2.1 Regularisation

The use of torch.optim.Adam as our optimiser by default implements L2 regularisation to punish large weights. This was one of the reasons we decided to apply MinMaxScaler to our y values within the preprocessor, as to avoid having large weights on the output layer.

2.2.2 Dropout

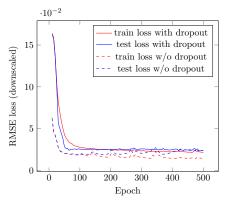


Figure 1: Effect of dropout on training and testing loss

We added this as an option the Regressor, toggled by the dropout parameter. Is not active by default for reasons we'll discuss in 2.2.3.

Dropout is implemented by passing the option as a boolean to the neural network model when calling the model's forward() function. This is only active during training, and applies nn.Dropout over each layer in the same manner as an activation function, causing 50% of outputs from a layer to be zeroed.

It can be observed from Figure 2 that the model without any significant measure to counteract overfitting begins to overfit to the training data, as indicated by the dashed lines becoming further apart. Conversely, it can also be observed that the performance of the model on both training and testing data remains fairly consistent with one another, though the training loss with dropout remains higher than without, whilst the loss on the test set without dropout increases past the test error with dropout.

2.2.3 Early Stopping

As seen in Figure 2, a standard model would overfit to the training data past a certain number of epochs, this results in the model performing badly on unseen data. Another means we have implemented to counteract this is early stopping.

This involves 3 variables passed to a Regressor instance: early_stopping - a boolean toggle for whether early stopping is enabled for the current Regressor, patience - the number of consecutive epochs without an improvement on the validation set before training is stopped, and earliest_stop - the earliest epoch at which we start considering early stopping.

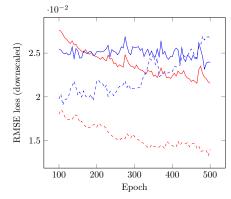


Figure 2: Final 400 Epochs

We first split the given training set into training and validation sets, at each epoch, the current model is trained on the training set, and

evaluated on the validation set, if the performance on the validation set is better than some previous optimal performance, the current model is stored as the best model. Otherwise, it increments a number strike for the number of consecutive epochs without an improvement. If strike exceeds patience, then the training process is stopped, and the best model is returned.

In Figure 3 we can observe the training error, best validation error and the testing error. Note that there are sudden spikes in the values of the training error and testing error, this is likely a result of our use of torch.optim.Adam, which when the gradient becomes small, the result often spikes in this manner. A way we can circumvent its effects is to increase the patience of the model.

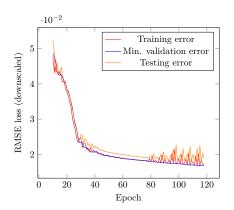


Figure 3: Effect of early stopping on train, test and validation loss

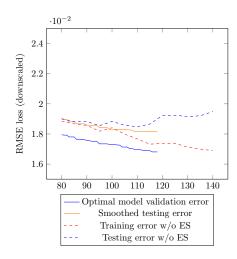


Figure 4: Smoothed results of early stopping with error values from another model without early stopping

For the sake of illustration, we've shown a smoothed version of both training and testing errors on Figure 4 along with the training and testing errors from another separately trained model, with its overfitting evident.

Through repeated cross-validation experiments, we found that effects of dropout and early stopping overlapped, in fact, it appeared that dropout slightly, though negatively, impacted the performance of early stopping, and thus we opted for early stopping to be enabled by default in lieu of dropout.

3 Hyperparameter Search

3.1 The overall search strategy

Now, there are three main sets of parameters that heavily affect a neural network's performance: the number of hidden layers, the neuron number of neurons in each layer(size of each layer), and the types of activations functions used in each layer. To obtain an optimal combination of these hyperparameters with a single set of experiments is not only computationally expensive, but also less robust. Instead, a search strategy is used, in which two major sets of experiments on two levels are conducted, one precedes the other. The first set will determine the optimal range of layer number. The second set then performs search within the optimal range of layer number, varying hyperparameters for each layer like the layer size and activations, to arrive at a best combination. For this experiment we did not run separate parameter setups for torch.optim.SGD optimisers, as its functionality was implemented in our choice of torch.optim.Adam, and we did not expect significantly different performance as a result.

Additionally for larger numbers of hidden layers, there were a significantly larger number of possible combinations of activation functions available, which would be computationally expensive and also bloat the number of samples at higher layer counts. Therefore we opted to select a few combinations for each layer count that were similar in structure across different setups to gain a general insight into the effect that the number of layers had on performance, and to narrow down a range of layer counts to explore further.

3.2 Search for optimal range of hidden layer number

Table 1 displays the results from the first set of experiments, where all the layers have the same neuron number, for a easier comparison on layer number. It can be seen that the lowerst RMSE value of all is highlighted red,

this occurs at a hidden layer number of 4. Investigating further, one can conclude that RMSE values are on average lower with hidden layer number ranging from 2 to 4, RMSE values with hidden layer number beyond 5 increases considerably. Within the same layer number, neuron number and activations are also varied, however their effect is deemed less significant. One exception is at layer number 5, when the activations at last two layers are swapped, the RMSE values rise massively. The suspected reason for the high RMSE values at higher layers counts is attributed to the small delta values within the network, as a result, past a certain number of epochs the gradient between layers effectively become zero, causing the model to not update, thereby never converging towards a lower optimum value. Our solution to this issue was therefore to use lower layer counts.

3.3 A more detailed search into the network neuron configurations

The next set of experiments is focused on varying activations and neuron number at each layer more closely, under the same hidden layer number. Table 2 shows results for a constant hidden layer count of 3, while Table 3 shows results for layer number of 4. In both tables, the lowest RMSE values have been highlighted red. In the 3 layer case (Table 2), a neuron size combination of 512-512-128 is found to be optimal, with RMSE of 67602 using all ReLU, RMSE of 67443 using all ReLU except for the middle layer using Sigmoid. Both RMSE are among the lowest in the table, and the difference between the two is not significant. Thus it is concluded that for a 3-layer network, using a 512-512-128 configuration with at least ReLU at the first or final layer can yield the best model performance.

For the 4-layer case(Table 3), it can be seen that the 128-512-512-128 configuration yields the lowest RMSE. Furthermore, activations using ReLU are deemed optimal, and substituting ReLU with Sigmoid in the middle layers has devastating effect on the RMSE (shoots up from 68183 to 84121). Therefore 128-512-512-128 with all ReLU is the best combination.

Given all above, using a large neuron count like 512 at each layer is recommended, except for the beginning and ending layers the layer size should be tapered to yield the best performance. Activations at each layer should be kept constant, and ReLU is considered the best choice.

4 Final Evaluation

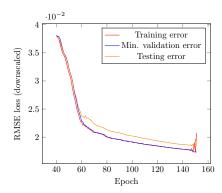


Figure 5: Performance of best model during training starting from epoch 40

As a result of our experimentation in section 3, we opted to use a 3 hidden layer neural network for our model, with 512-512-128 neuron configuration for the hidden layers. Our choice of activation functions was to use ReLU for all hidden layers, as it demonstrated the most consistently high performance across different configuration.

The model, using the methods detailed in the previous sections, performed comparatively well against our other models, with 65056 on the training data set, and 65056 on the test set, which was not used in the training of the model. This model has an RMSE error value of 65367 on LabTS.

Though this is still quite a high error rate considering the context of this task. It can be argued that perhaps a linear model is not a good fit for the relation between these features and the median house value. A potential future point of exploration is to add n-th polynomial terms to the features during preprocessing.

Appendices

A Results Tables

A.1

Varied hidden layer count, with drop-out and early-stopping											
Error in	No. of activation	Layer size	Activation at each layer								
RMSE layers	layers	Layer size	at layer 1	at layer 2	at layer 3	at layer 4	at layer 5				
72018	1	32	Sigmoid	n/a	n/a	n/a	n/a				
71710	1	64	Sigmoid	n/a	n/a	n/a	n/a				
71012	2	32	Sigmoid	Tanh	n/a	n/a	n/a				
71030	2	64	Sigmoid	Tanh	n/a	n/a	n/a				
71688	3	13	Sigmoid	Tanh	Sigmoid	n/a	n/a				
71081	3	32	Sigmoid	Tanh	Sigmoid	n/a	n/a				
71352	3	32	Sigmoid	ReLU	Sigmoid	n/a	n/a				
69988	4	13	Sigmoid	Tanh	ReLU	Sigmoid	n/a				
72907	4	24	Sigmoid	Tanh	ReLU	Sigmoid	n/a				
70948	4	32	Sigmoid	Tanh	ReLU	Sigmoid	n/a				
73178	5	13	Sigmoid	Tanh	ReLU	ReLU	Sigmoid				
77174	5	24	Sigmoid	Tanh	ReLU	ReLU	Sigmoid				
75600	5	32	Sigmoid	Tanh	ReLU	ReLU	Sigmoid				
115738	5	13	Sigmoid	Tanh	ReLU	Sigmoid	ReLU				
70092	5	24	Sigmoid	Tanh	ReLU	Sigmoid	ReLU				
115805	5	32	Sigmoid	Tanh	ReLU	Sigmoid	ReLU				

Table 1: Performances under varied hidden layer count, with constant neuron(nodes) number per layer, with drop-out and early-stopping applied.

A.2

Hidden layer count=3, with drop-out and early-stopping									
Error in RMSE	Activa	tion at eac	h layer	number of nodes at each layer					
Elloi ili itivise	at layer 1	at layer 2	at layer 3	at layer 1	at layer 2	at layer 3			
74809	ReLU	ReLU	ReLU	13	13	13			
70708	ReLU	ReLU	ReLU	32	64	32			
69054	ReLU	ReLU	ReLU	64	128	64			
67602	ReLU	ReLU	ReLU	512	512	128			
67451	ReLU	ReLU	ReLU	512	512	512			
77506	ReLU	Sigmoid	ReLU	13	13	13			
70401	ReLU	Sigmoid	ReLU	32	64	32			
68852	ReLU	Sigmoid	ReLU	64	128	64			
67443	ReLU	Sigmoid	ReLU	512	512	128			
68560	ReLU	Sigmoid	ReLU	512	512	512			

Table 2: Performances under varied neuron configurations and activations, with a constant hidden layers number of 3.

A.3

Hidden layer count=4, with drop-out and early-stopping											
Error in RMSE	A	Activation a	at each laye	number of nodes at each layer							
Error in terrise	at layer 1	at layer 2	at layer 3	at layer 4	at	at	at	at			
					layer	layer	layer	layer			
					1	2	3	4			
71129	ReLU	ReLU	ReLU	ReLU	64	128	128	64			
68183	ReLU	ReLU	ReLU	ReLU	128	512	512	128			
68874	ReLU	ReLU	ReLU	ReLU	256	512	512	256			
72115	ReLU	Sigmoid	ReLU	ReLU	64	128	128	64			
84121	ReLU	Sigmoid	ReLU	ReLU	128	512	512	128			
69699	ReLU	Sigmoid	Sigmoid	ReLU	64	128	128	64			
84121	ReLU	Sigmoid	Sigmoid	ReLU	128	512	512	128			

Table 3: Performances under varied neuron configurations and activations, with a constant hidden layers number of 4.