

Test Preview**TestSummary.txt: 1/1****:c3**

```
1: Test Preview: Summary for of c3
2: -----
3:
4:   Public Tests:
5:     Part 1: Linear layer:      3 / 3
6:     Part 1: Network:          2 / 2
7:     Part 1: Trainer:          3 / 3
8:     Part 1: Preprocessor:      4 / 4
9:     Part 1: Activation functions: 4 / 4
10:    Part 2: Regressor:         0 / 1
11:    Part 2: Preprocessing:      1 / 1
12:    Part 2: Performance:       0 / 1
13:
14: Git Repo: git@gitlab.doc.ic.ac.uk:lab2223_autumn/Neural_Networks_078.git
15: Commit ID: d0108
```

Test Preview

part2_house_value_regression.py: 1/7

:c3

```

1: import math
2: from sklearn.metrics import mean_squared_error
3: import torch.nn as nn
4: import torch
5: import pickle
6: import numpy as np
7: import pandas as pd
8:
9: from pandas import DataFrame, concat
10: from sklearn.compose import ColumnTransformer
11: from sklearn.preprocessing import StandardScaler, OneHotEncoder, PowerTransformer
12: from torch import Tensor, tensor, float32
13: from typing import Callable, Union, List
14:
15: def as_torch_tensor(func: Callable[..., DataFrame]):
16:     def wrapper(*args, **kwargs):
17:         df: DataFrame = func(*args, **kwargs)
18:         return tensor(df, dtype=float32)
19:     return wrapper
20:
21: class Preprocessor():
22:     __input_transformer: ColumnTransformer
23:     __target_transformer: ColumnTransformer
24:
25:     def __init__(self):
26:         self.__input_transformer = ColumnTransformer(
27:             transformers=[
28:                 ('std', StandardScaler(), [
29:                     'longitude',
30:                     'latitude',
31:                 ]),
32:                 ('pwr', PowerTransformer(method="box-cox", standardize=True), [
33:                     'housing_median_age',
34:                     'total_rooms',
35:                     'total_bedrooms',
36:                     'population',
37:                     'households',
38:                     'median_income',
39:                 ]),
40:                 ('one_hot', OneHotEncoder(), ['ocean_proximity']),
41:             ],
42:             remainder='drop',
43:         )
44:         self.__target_transformer = ColumnTransformer(
45:             transformers=[
46:                 ('std', StandardScaler(), ['median_house_value'])
47:             ],
48:             remainder='drop',
49:         )
50:
51:     def fill_missing(self, x: DataFrame):
52:         x = x.copy()
53:
54:         total_bedrooms_mean = x['total_bedrooms'].mean()
55:         x['total_bedrooms'] = x['total_bedrooms'].fillna(total_bedrooms_mean)
56:
57:         return x
58:
59:     def fit_input(self, input_df: DataFrame):
60:         self.__input_transformer.fit(input_df)
61:
62:     def fit_target(self, target_df: DataFrame):
63:         self.__target_transformer.fit(target_df)
64:
65:     @as_torch_tensor
66:     def transform_input(self, input_df: DataFrame) -> Tensor:

```

Test Preview

part2_house_value_regression.py: 2/7

:c3

```

67:         return self.__input_transformer.transform(input_df)
68:
69:     @as_torch_tensor
70:     def transform_target(self, target_df: DataFrame) -> Tensor:
71:         return self.__target_transformer.transform(target_df)
72:
73:     def inverse_transform_target(self, target_tensor: Tensor):
74:         target_np = target_tensor.detach().numpy()
75:
76:         return self.__target_transformer \
77:             .named_transformers_['std'] \
78:             .inverse_transform(target_np)
79:
80:
81: class Regressor():
82:     __preprocessor: Preprocessor
83:     __loss_history: List[float]
84:     __loss_history_eval: List[float]
85:
86:     def __init__(self, x, nb_epoch=1000, model=None, batch_size=32,
learning_rate=0.01, loss_fn=nn.MSELoss()):
87:         # You can add any input parameters you need
88:         # Remember to set them with a default value for LabTS tests
89:         """
90:         Initialise the model.
91:
92:         Arguments:
93:         - x {pd.DataFrame} -- Raw input data of shape
94:         (batch_size, input_size), used to compute the size
95:         of the network.
96:         - nb_epoch {int} -- number of epochs to train the network.
97:
98:         """
99:
100:         self.nb_epoch = nb_epoch
101:         self.batch_size = batch_size
102:         self.learning_rate = learning_rate
103:         self.loss_fn = loss_fn
104:
105:         # Initialise Loss History
106:         self.__loss_history = []
107:         self.__loss_history_eval = []
108:
109:         # Construct Preprocessor
110:         self.__preprocessor = Preprocessor()
111:
112:         # Initialise Preprocessor
113:         X, _ = self.__preprocessor(x, training=True)
114:
115:         self.input_size = X.shape[1]
116:         self.output_size = 1
117:
118:         # default configuration
119:         if model is None:
120:             self.model = nn.Sequential(
121:                 nn.Linear(self.input_size, 64),
122:                 nn.ReLU(),
123:                 nn.Linear(64, 64),
124:                 nn.ReLU(),
125:                 nn.Linear(64, self.output_size),
126:             )
127:         else:
128:             self.model = model
129:
130:     def __preprocessor(self, x, y=None, training=False):
131:         """

```

```

132: Preprocess input of the network.
133:
134: Arguments:
135:     - x {pd.DataFrame} -- Raw input array of shape
136:       (batch_size, input_size).
137:     - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1).
138:     - training {boolean} -- Boolean indicating if we are training or
139:       testing the model.
140:
141: Returns:
142:     - {torch.tensor} or {numpy.ndarray} -- Preprocessed input array of
143:       size (batch_size, input_size). The input_size does not have to be /
the same as the input_size for x above.
144:     - {torch.tensor} or {numpy.ndarray} -- Preprocessed target array of
145:       size (batch_size, 1).
146:
147: """
148:
149: # clean data
150: x = self.__preprocessor.fill_missing(x)
151:
152: # Remove rows who's age are at maximum threshold.
153: age_filter = x['housing_median_age'] <= 50
154: x = x[age_filter]
155: if y is not None: y = y[age_filter]
156:
157: # Remove rows who's price is at maximum threshold.
158: if y is not None:
159:     price_filter = y['median_house_value'] <= 500000
160:     x = x[price_filter]
161:     y = y[price_filter]
162:
163: # If training flag set, fit preprocessor to training data.
164: if training:
165:     self.__preprocessor.fit_input(x)
166:     if y is not None: self.__preprocessor.fit_target(y)
167:
168: return (
169:     self.__preprocessor.transform_input(x),
170:     self.__preprocessor.transform_target(y) if y is not None else None
171: )
172:
173: def fit(self, x, y, x_eval = None, y_eval = None):
174:     """
175:     Regressor training function
176:
177:     Arguments:
178:         - x {pd.DataFrame} -- Raw input array of shape
179:           (batch_size, input_size).
180:         - y {pd.DataFrame} -- Raw output array of shape (batch_size, 1).
181:
182:     Returns:
183:         self {Regressor} -- Trained model.
184:
185:     """
186:
187:     input_data, target_data = self.__preprocessor(
188:         x, y=y, training=True) # Do not forget
189:
190:     for epoch_n in range(self.nb_epoch):
191:         # Adam optimiser is sick
192:         optimiser = torch.optim.SGD(
193:             self.model.parameters(),
194:             lr=self.learning_rate,
195:             momentum=0.9,
196:         )

```

```

197:
198:     # self.model.parameters(), lr=self.learning_rate)
199:
200:     # shuffle data and split into batches using DataLoader
201:     rand_perm = np.random.permutation(len(input_data))
202:     shuffled_input_data = input_data[rand_perm]
203:     shuffled_target_data = target_data[rand_perm]
204:
205:     # split into self.batch_size sized batches
206:     no_batches = int(x.shape[0] / self.batch_size)
207:     input_batches = np.array_split(shuffled_input_data, no_batches)
208:     target_batches = np.array_split(shuffled_target_data, no_batches)
209:
210:     for (input_batch, target_batch) in zip(input_batches, /
target_batches):
211:         optimiser.zero_grad()
212:
213:         # Perform a forward pass through the model
214:         outputs = self.model(input_batch)
215:
216:         # Compute loss
217:         # Compute gradient of loss via backwards pass
218:         loss = self.loss_fn(outputs, target_batch)
219:         loss.backward()
220:
221:         # Change the weights via gradient decent
222:         optimiser.step()
223:
224:     error_train = self.score(x, y)
225:     self.__loss_history.append(error_train)
226:
227:     # print(f"Epoch {epoch_n} error (train): {error_train}")
228:
229:     if x_eval is not None and y_eval is not None:
230:         error_eval = self.score(x_eval, y_eval)
231:         self.__loss_history_eval.append(error_eval)
232:
233:         # print(f"Epoch {epoch_n} error (eval): {error_eval}")
234:
235:     return self.model
236:
237: #####
238: # ** END OF YOUR CODE **
239: #####
240:
241: def loss_history(self) -> List[float]:
242:     return self.__loss_history
243:
244: def loss_history_eval(self) -> List[float]:
245:     return self.__loss_history_eval
246:
247: def predict(self, x):
248:     """
249:     Output the value corresponding to an input x.
250:
251:     Arguments:
252:         x {pd.DataFrame} -- Raw input array of shape
253:           (batch_size, input_size).
254:
255:     Returns:
256:         {np.ndarray} -- Predicted value for the given input (batch_size, 1).
257:
258:     """
259:
260:     #####
261:     # ** START OF YOUR CODE **

```

```

Test Preview      part2_house_value_regression.py: 5/7      :c3
262: #####
263:
264: X, _ = self._preprocessor(x, training=False)  # Do not forget
265:
266: O = self.model.forward(X)
267:
268: return self._preprocessor.inverse_transform_target(O)
269:
270: #####
271: #                ** END OF YOUR CODE **
272: #####
273:
274: def score(self, x, y):
275:     """
276:     Function to evaluate the model accuracy on a validation dataset.
277:
278:     Arguments:
279:         - x {pd.DataFrame} -- Raw input array of shape
280:           (batch_size, input_size).
281:         - y {pd.DataFrame} -- Raw output array of shape (batch_size, 1).
282:
283:     Returns:
284:         {float} -- Quantification of the efficiency of the model.
285:
286:     """
287:
288:     X_norm, Y_norm = self._preprocessor(x, y=y, training=False)  # Do not forget
289:
290:     Y_pred_norm = self.model.forward(X_norm)
291:
292:     Y_pred = self._preprocessor.inverse_transform_target(Y_pred_norm)
293:     Y = self._preprocessor.inverse_transform_target(Y_norm)
294:
295:     return mean_squared_error(Y_pred, Y, squared=False)
296:
297:
298: def save_regressor(trained_model, model_name: Union[str, None] = None):
299:     """
300:     Utility function to save the trained regressor model in part2_model.pickle.
301:     """
302:     # If you alter this, make sure it works in tandem with load_regressor
303:     model_pickle_path = 'part2_model.pickle' if model_name is None \
304:         else f'assets/{model_name}.pickle'
305:
306:     with open(model_pickle_path, 'wb') as target:
307:         pickle.dump(trained_model, target)
308:     print(f"\nSaved model in {model_pickle_path}")
309:
310:
311: def load_regressor(model_name: Union[str, None] = None):
312:     """
313:     Utility function to load the trained regressor model in part2_model.pickle.
314:     """
315:     model_pickle_path = 'part2_model.pickle' if model_name is None \
316:         else f'assets/{model_name}.pickle'
317:
318:     # If you alter this, make sure it works in tandem with save_regressor
319:     with open(model_pickle_path, 'rb') as target:
320:         trained_model = pickle.load(target)
321:     print(f"\nLoaded model in {model_pickle_path}\n")
322:     return trained_model
323:
324:

```

```

Test Preview      part2_house_value_regression.py: 6/7      :c3
325: def RegressorHyperParameterSearch():
326:     # Ensure to add whatever inputs you deem necessary to this function
327:     """
328:     Performs a hyper-parameter for fine-tuning the regressor implemented
329:     in the Regressor class.
330:
331:     Arguments:
332:         Add whatever inputs you need.
333:
334:     Returns:
335:         The function should return your optimised hyper-parameters.
336:
337:     """
338:
339:     #####
340:     #                ** START OF YOUR CODE **
341:     #####
342:
343:     return # Return the chosen hyper parameters
344:
345:     #####
346:     #                ** END OF YOUR CODE **
347:     #####
348:
349:
350: def example_main():
351:     # Use pandas to read CSV data as it contains various object types
352:     # Feel free to use another CSV reader tool
353:     # But remember that LabTS tests take Pandas DataFrame as inputs
354:     train_data = pd.read_csv("housing.csv")
355:     eval_data = pd.read_csv("housing_eval.csv")
356:
357:     data_main(train_data, eval_data)
358:
359: def k_fold_main(k):
360:     data = pd.read_csv("housing.csv")
361:
362:     chunk_size = data.shape[0] // k
363:     data_split = [data[i : i + chunk_size] for i in range(0, data.shape[0],
364: chunk_size)]
365:
366:     total = 0
367:
368:     for i in range(0, k):
369:         # The eval data is this kth of the data.
370:         eval_data = data_split[i]
371:
372:         # The training data is all but the eval data
373:         train_data = data_split[:]
374:         del train_data[i]
375:         train_data = pd.concat(train_data)
376:
377:         total += data_main(train_data, eval_data)
378:
379:     print(f"Average score over {k} splits = {total / k}")
380:
381:     # Trains model with train_data,
382:     # Evaluates with eval_data,
383:     # Returns score
384:     def data_main(train_data, eval_data) -> float:
385:         output_label = "median_house_value"
386:
387:         # Splitting input and output
388:         x_train = train_data.loc[:, train_data.columns != output_label]
389:         y_train = train_data.loc[:, [output_label]]

```

```

390:
391:     # Training
392:     # This example trains on the whole available dataset.
393:     # You probably want to separate some held-out data
394:     # to make sure the model isn't overfitting
395:     regressor = Regressor(x_train, nb_epoch=100, learning_rate=0.01)
396:     regressor.fit(x_train, y_train)
397:     save_regressor(regressor)
398:
399:     # Error
400:     error = regressor.score(x_train, y_train)
401:     print("\nRegressor error: {}".format(error))
402:
403:     # Eval Error
404:     eval_x_train = eval_data.loc[:, eval_data.columns != output_label]
405:     eval_y_train = eval_data.loc[:, [output_label]]
406:     eval_error = regressor.score(eval_x_train, eval_y_train)
407:     print("\nRegressor error vs eval: {}\n".format(eval_error))
408:
409:     return eval_error
410:
411:
412: if __name__ == "__main__":
413:     example_main()
414:     # k_fold_main(10)
415:

```

```

1: import numpy as np
2: import pickle
3:
4:
5: def xavier_init(size, gain = 1.0):
6:     """
7:     Xavier initialization of network weights.
8:
9:     Arguments:
10:         - size {tuple} -- size of the network to initialise.
11:         - gain {float} -- gain for the Xavier initialisation.
12:
13:     Returns:
14:         {np.ndarray} -- values of the weights.
15:     """
16:     low = -gain * np.sqrt(6.0 / np.sum(size))
17:     high = gain * np.sqrt(6.0 / np.sum(size))
18:     return np.random.uniform(low=low, high=high, size=size)
19:
20:
21: class Layer:
22:     """
23:     Abstract layer class.
24:     """
25:
26:     def __init__(self, *args, **kwargs):
27:         raise NotImplementedError()
28:
29:     def forward(self, *args, **kwargs):
30:         raise NotImplementedError()
31:
32:     def __call__(self, *args, **kwargs):
33:         return self.forward(*args, **kwargs)
34:
35:     def backward(self, *args, **kwargs):
36:         raise NotImplementedError()
37:
38:     def update_params(self, *args, **kwargs):
39:         pass
40:
41:
42: class MSELossLayer(Layer):
43:     """
44:     MSELossLayer: Computes mean-squared error between y_pred and y_target.
45:     """
46:
47:     def __init__(self):
48:         self._cache_current = None
49:
50:     @staticmethod
51:     def _mse(y_pred, y_target):
52:         return np.mean((y_pred - y_target) ** 2)
53:
54:     @staticmethod
55:     def _mse_grad(y_pred, y_target):
56:         return 2 * (y_pred - y_target) / len(y_pred)
57:
58:     def forward(self, y_pred, y_target):
59:         self._cache_current = y_pred, y_target
60:         return self._mse(y_pred, y_target)
61:
62:     def backward(self):
63:         return self._mse_grad(*self._cache_current)
64:
65:
66: class CrossEntropyLossLayer(Layer):

```

```

67: """
68: CrossEntropyLossLayer: Computes the softmax followed by the negative
69: log-likelihood loss.
70: """
71:
72: def __init__(self):
73:     self._cache_current = None
74:
75: @staticmethod
76: def softmax(x):
77:     numer = np.exp(x - x.max(axis=1, keepdims=True))
78:     denom = numer.sum(axis=1, keepdims=True)
79:     return numer / denom
80:
81: def forward(self, inputs, y_target):
82:     assert len(inputs) == len(y_target)
83:     n_obs = len(y_target)
84:     probs = self.softmax(inputs)
85:     self._cache_current = y_target, probs
86:
87:     out = -1 / n_obs * np.sum(y_target * np.log(probs))
88:     return out
89:
90: def backward(self):
91:     y_target, probs = self._cache_current
92:     n_obs = len(y_target)
93:     return -1 / n_obs * (y_target - probs)
94:
95:
96: class SigmoidLayer(Layer):
97:     """
98:     SigmoidLayer: Applies sigmoid function elementwise.
99:     """
100:
101: def __init__(self):
102:     """
103:     Constructor of the Sigmoid layer.
104:     """
105:     self._cache_current = None
106:
107: def forward(self, x):
108:     """
109:     Performs forward pass through the Sigmoid layer.
110:
111:     Logs information needed to compute gradient at a later stage in
112:     '_cache_current'.
113:
114:     Arguments:
115:         x {np.ndarray} -- Input array of shape (batch_size, n_in).
116:
117:     Returns:
118:         {np.ndarray} -- Output array of shape (batch_size, n_out)
119:     """
120:
121:     self._cache_current = 1 / (1 + np.exp(-x))
122:     return self._cache_current
123:
124:
125: def backward(self, grad_z):
126:     """
127:     Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
128:     the output of this layer, performs back pass through the layer (i.e.
129:     computes gradients of loss with respect to parameters of layer and
130:     inputs of layer).
131:
132:     Arguments:

```

```

133:     grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
134:
135:     Returns:
136:         {np.ndarray} -- Array containing gradient with respect to layer
137:         input, of shape (batch_size, n_in).
138:     """
139:
140:     derivative = self._cache_current * (1 - self._cache_current)
141:     return grad_z * derivative
142:
143:
144:
145:
146: class ReluLayer(Layer):
147:     """
148:     ReluLayer: Applies Relu function elementwise.
149:     """
150:
151: def __init__(self):
152:     """
153:     Constructor of the Relu layer.
154:     """
155:     self._cache_current = None
156:
157: def forward(self, x):
158:     """
159:     Performs forward pass through the Relu layer.
160:
161:     Logs information needed to compute gradient at a later stage in
162:     '_cache_current'.
163:
164:     Arguments:
165:         x {np.ndarray} -- Input array of shape (batch_size, n_in).
166:
167:     Returns:
168:         {np.ndarray} -- Output array of shape (batch_size, n_out)
169:     """
170:
171:     self._cache_current = np.where(x <= 0, 0, x)
172:     return self._cache_current
173:
174: def backward(self, grad_z):
175:     """
176:     Given 'grad_z', the gradient of some scalar (e.g. loss) with respect to
177:     the output of this layer, performs back pass through the layer (i.e.
178:     computes gradients of loss with respect to parameters of layer and
179:     inputs of layer).
180:
181:     Arguments:
182:         grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
183:
184:     Returns:
185:         {np.ndarray} -- Array containing gradient with respect to layer
186:         input, of shape (batch_size, n_in).
187:     """
188:
189:     derivative = np.where(self._cache_current > 0, 1, self._cache_current)
190:     return grad_z * derivative
191:
192:
193: class LinearLayer(Layer):
194:     """
195:     LinearLayer: Performs affine transformation of input.
196:     """
197:
198: def __init__(self, n_in, n_out): #Å shake it all about

```

```

199: """
200: Constructor of the linear layer.
201:
202: Arguments:
203:     - n_in {int} -- Number (or dimension) of inputs.
204:     - n_out {int} -- Number (or dimension) of outputs.
205: """
206: self.n_in = n_in
207: self.n_out = n_out
208: # shake it all about
209:
210: #####
211: # ** START OF YOUR CODE **
212: #####
213:
214: """
215: Weights have the shape:
216:
217:     (w_l1, w_l2, ..., w_ln_in)
218:     ...
219:     ...
220:     (w_n_out1, w_n_out2, ..., w_n_outn_in)
221:
222: where w_ij is the weight from the i-th input to the j-th output.
223:
224: Bias are initialized to 0, as a vector of size n_out.
225:
226: """
227:
228: self.W = xavier_init((n_in, n_out)) # shake it all about
229: self.b = np.zeros((1, n_out))
230:
231: self._cache_current = None
232: self._grad_W_current = None
233: self._grad_b_current = None
234:
235: #####
236: # ** END OF YOUR CODE **
237: #####
238:
239: def forward(self, x):
240:
241:     """
242:     Performs forward pass through the layer (i.e. returns Wx + b).
243:
244:     Logs information needed to compute gradient at a later stage in
245:     `_cache_current`.
246:
247:     Arguments:
248:         x {np.ndarray} -- Input array of shape (batch_size, n_in).
249:
250:     Returns:
251:         {np.ndarray} -- Output array of shape (batch_size, n_out)
252:     """
253:
254:     # store input array in cache for backpropagation
255:     self._cache_current = x
256:     return np.dot(x, self.W) + self.b
257:
258:
259: def backward(self, grad_z):
260:     """
261:     Given `grad_z`, the gradient of some scalar (e.g. loss) with respect to
262:     the output of this layer, performs back pass through the layer (i.e.
263:     computes gradients of loss with respect to parameters of layer and
264:     inputs of layer).

```

```

265:
266: Arguments:
267:     grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).
268:
269: Returns:
270:     {np.ndarray} -- Array containing gradient with respect to layer
271:     input, of shape (batch_size, n_in).
272: """
273:
274: # Compute gradient with respect to layer input
275: self._grad_W_current = np.dot(self._cache_current.T, grad_z)
276:
277: # sum biases along columns
278: self._grad_b_current = np.sum(grad_z, axis=0, keepdims=True)
279:
280: # Compute gradient with respect to layer parameters
281: return np.dot(grad_z, self._W.T)
282:
283: def update_params(self, learning_rate):
284:     """
285:     Performs one step of gradient descent with given learning rate on the
286:     layer's parameters using currently stored gradients.
287:
288:     Arguments:
289:         learning_rate {float} -- Learning rate of update step.
290:     """
291:
292:     self._W -= learning_rate * self._grad_W_current
293:     self._b -= learning_rate * self._grad_b_current
294:
295: class MultiLayerNetwork(object):
296:     """
297:     MultiLayerNetwork: A network consisting of stacked linear layers and
298:     activation functions.
299:     """
300:
301:     def __init__(self, input_dim, neurons, activations):
302:         """
303:         Constructor of the multi layer network.
304:
305:         Arguments:
306:             - input_dim {int} -- Number of features in the input (excluding
307:             the batch dimension).
308:             - neurons {list} -- Number of neurons in each linear layer
309:             represented as a list. The length of the list determines the
310:             number of linear layers.
311:             - activations {list} -- List of the activation functions to apply
312:             to the output of each linear layer.
313:         """
314:         self.input_dim = input_dim
315:         self.neurons = neurons
316:         self.activations = activations
317:
318:         self._layers = []
319:
320:         if (len(neurons) != len(activations)):
321:             raise ValueError("The number of layers and activations must be equal")
322:
323:         for i in range(len(neurons)):
324:             if (i == 0):
325:                 self._layers.append(LinearLayer(input_dim, neurons[i]))
326:             else:
327:                 self._layers.append(LinearLayer(neurons[i-1], neurons[i]))
328:
329:

```

```

330:         match activations[i]:
331:             case "relu":
332:                 self._layers.append(ReluLayer())
333:             case "sigmoid":
334:                 self._layers.append(SigmoidLayer())
335:             case "identity":
336:                 continue
337:
338:     def forward(self, x):
339:         """
340:         Performs forward pass through the network.
341:
342:         Arguments:
343:             x {np.ndarray} -- Input array of shape (batch_size, input_dim).
344:
345:         Returns:
346:             {np.ndarray} -- Output array of shape (batch_size,
347:             #neurons_in_final_layer)
348:         """
349:
350:         for layer in self._layers:
351:             x = layer.forward(x)
352:
353:         return x
354:
355:     def __call__(self, x):
356:         return self.forward(x)
357:
358:     def backward(self, grad_z):
359:         """
360:         Performs backward pass through the network.
361:
362:         Arguments:
363:             grad_z {np.ndarray} -- Gradient array of shape (batch_size,
364:             #neurons_in_final_layer).
365:
366:         Returns:
367:             {np.ndarray} -- Array containing gradient with respect to layer
368:             input, of shape (batch_size, input_dim).
369:         """
370:
371:         for layer in reversed(self._layers):
372:             grad_z = layer.backward(grad_z)
373:
374:         return grad_z
375:
376:     def update_params(self, learning_rate):
377:         """
378:         Performs one step of gradient descent with given learning rate on the
379:         parameters of all layers using currently stored gradients.
380:
381:         Arguments:
382:             learning_rate {float} -- Learning rate of update step.
383:         """
384:
385:         for layer in self._layers:
386:             layer.update_params(learning_rate)
387:
388:
389:     def save_network(network, fpath):
390:         """
391:         Utility function to pickle 'network' at file path 'fpath'.
392:         """
393:         with open(fpath, "wb") as f:
394:             pickle.dump(network, f)
395:

```

```

396:
397: def load_network(fpath):
398:     """
399:     Utility function to load network found at file path 'fpath'.
400:     """
401:     with open(fpath, "rb") as f:
402:         network = pickle.load(f)
403:     return network
404:
405:
406: class Trainer(object):
407:     """
408:     Trainer: Object that manages the training of a neural network.
409:     """
410:
411:     def __init__(
412:         self,
413:         network,
414:         batch_size,
415:         nb_epoch,
416:         learning_rate,
417:         loss_fun,
418:         shuffle_flag,
419:     ):
420:         """
421:         Constructor of the Trainer.
422:
423:         Arguments:
424:             - network {MultiLayerNetwork} -- MultiLayerNetwork to be trained.
425:             - batch_size {int} -- Training batch size.
426:             - nb_epoch {int} -- Number of training epochs.
427:             - learning_rate {float} -- SGD learning rate to be used in training.
428:             - loss_fun {str} -- Loss function to be used. Possible values: mse,
429:             cross_entropy.
430:             - shuffle_flag {bool} -- If True, training data is shuffled before
431:             training.
432:         """
433:         self.network = network
434:         self.batch_size = batch_size
435:         self.nb_epoch = nb_epoch
436:         self.learning_rate = learning_rate
437:         self.loss_fun = loss_fun
438:         self.shuffle_flag = shuffle_flag
439:
440:         match loss_fun:
441:             case "mse":
442:                 self._loss_layer = MSELossLayer()
443:             case "cross_entropy":
444:                 self._loss_layer = CrossEntropyLossLayer()
445:
446:
447:     @staticmethod
448:     def shuffle(input_dataset, target_dataset):
449:         """
450:         Returns shuffled versions of the inputs.
451:
452:         Arguments:
453:             - input_dataset {np.ndarray} -- Array of input features, of shape
454:             (#data_points, n_features) or (#data_points,).
455:             - target_dataset {np.ndarray} -- Array of corresponding targets, of
456:             shape (#data_points, #output_neurons).
457:
458:         Returns:
459:             - {np.ndarray} -- shuffled inputs.
460:             - {np.ndarray} -- shuffled_targets.
461:         """

```



```

462:
463:     rand_perm = np.random.permutation(len(input_dataset))
464:
465:     return (input_dataset[rand_perm], target_dataset[rand_perm])
466:
467: def train(self, input_dataset, target_dataset):
468:     """
469:     Main training loop. Performs the following steps `nb_epoch` times:
470:     - Shuffles the input data (if `shuffle` is True)
471:     - Splits the dataset into batches of size `batch_size`.
472:     - For each batch:
473:         - Performs forward pass through the network given the current
474:           batch of inputs.
475:         - Computes loss.
476:         - Performs backward pass to compute gradients of loss with
477:           respect to parameters of network.
478:         - Performs one step of gradient descent on the network
479:           parameters.
480:
481:     Arguments:
482:     - input_dataset {np.ndarray} -- Array of input features, of shape
483:       (#_training_data_points, n_features).
484:     - target_dataset {np.ndarray} -- Array of corresponding targets, of
485:       shape (#_training_data_points, #output_neurons).
486:     """
487:
488:
489:
490:     for _ in range(self.nb_epoch):
491:
492:         if self.shuffle_flag:
493:             input_dataset, target_dataset = Trainer.shuffle(input_dataset,
target_dataset)
494:
495:             no_batches = int(input_dataset.shape[0] / self.batch_size)
496:             input_batches = np.array_split(input_dataset, no_batches)
497:             target_batches = np.array_split(target_dataset, no_batches)
498:
499:             for i in range(no_batches):
500:                 forward = self.network.forward(input_batches[i])
501:                 self._loss_layer.forward(forward, target_batches[i])
502:                 self.network.backward(self._loss_layer.backward())
503:                 self.network.update_params(self.learning_rate)
504:
505: def eval_loss(self, input_dataset, target_dataset):
506:     """
507:     Function that evaluate the loss function for given data. Returns
508:     scalar value.
509:
510:     Arguments:
511:     - input_dataset {np.ndarray} -- Array of input features, of shape
512:       (#_evaluation_data_points, n_features).
513:     - target_dataset {np.ndarray} -- Array of corresponding targets, of
514:       shape (#_evaluation_data_points, #output_neurons).
515:
516:     Returns:
517:     a scalar value -- the loss
518:     """
519:     forward = self.network.forward(input_dataset)
520:     return self._loss_layer.forward(forward, target_dataset)
521:
522:
523:
524: class Preprocessor(object):
525:     """
526:     Preprocessor: Object used to apply "preprocessing" operation to datasets.

```

```

527:     The object can also be used to revert the changes.
528:     """
529:
530: def __init__(self, data):
531:     """
532:     Initializes the Preprocessor according to the provided dataset.
533:     (Does not modify the dataset.)
534:
535:     Arguments:
536:     data {np.ndarray} dataset used to determine the parameters for
537:       the normalization.
538:     """
539:     self.min_range = 0
540:     self.max_range = 1
541:
542:     self.min_data = np.min(data, axis=0)
543:     self.max_data = np.max(data, axis=0)
544:
545: def apply(self, data):
546:     """
547:     Apply the pre-processing operations to the provided dataset.
548:
549:     Arguments:
550:     data {np.ndarray} dataset to be normalized.
551:
552:     Returns:
553:     {np.ndarray} normalized dataset.
554:     """
555:
556:     # Normalize the data using min-max normalization
557:
558:     return ((data - self.min_data) * (self.max_range - self.min_range)) /
(self.max_data - self.min_data)
559:
560:
561: def revert(self, data):
562:     """
563:     Revert the pre-processing operations to retrieve the original dataset.
564:
565:     Arguments:
566:     data {np.ndarray} dataset for which to revert normalization.
567:
568:     Returns:
569:     {np.ndarray} reverted dataset.
570:     """
571:
572:     return (data * (self.max_data - self.min_data)) / (self.max_range -
self.min_range) + self.min_data
573:
574:
575: def example_main():
576:     input_dim = 4
577:     neurons = [16, 3]
578:     activations = ["relu", "identity"]
579:     net = MultiLayerNetwork(input_dim, neurons, activations)
580:
581:     dat = np.loadtxt("iris.dat")
582:     np.random.shuffle(dat)
583:
584:     x = dat[:, :4]
585:     y = dat[:, 4:]
586:
587:     split_idx = int(0.8 * len(x))
588:
589:     x_train = x[:split_idx]
590:     y_train = y[:split_idx]

```

```
591:     x_val = x[split_idx:]
592:     y_val = y[split_idx:]
593:
594:     prep_input = Preprocessor(x_train)
595:
596:     x_train_pre = prep_input.apply(x_train)
597:     x_val_pre = prep_input.apply(x_val)
598:
599:     trainer = Trainer(
600:         network=net,
601:         batch_size=8,
602:         nb_epoch=1000,
603:         learning_rate=0.01,
604:         loss_fun="cross_entropy",
605:         shuffle_flag=True,
606:     )
607:
608:     trainer.train(x_train_pre, y_train)
609:     print("Train loss = ", trainer.eval_loss(x_train_pre, y_train))
610:     print("Validation loss = ", trainer.eval_loss(x_val_pre, y_val))
611:
612:     preds = net(x_val_pre).argmax(axis=1).squeeze()
613:     targets = y_val.argmax(axis=1).squeeze()
614:     accuracy = (preds == targets).mean()
615:     print("Validation accuracy: {}".format(accuracy))
616:
617:
618: if __name__ == "__main__":
619:     example_main()
```

Test Preview

testResults.txt: 1/1

:c3

```
1: ----- Test Output -----
2:
3:
4: PART 1 test output:
5:
6:
7: PART 2 test output:
8:
9: Exception thrown when creating and using an instance of Regressor.
10:
11: Loaded model in part2_model.pickle
12:
13:
14: Exception thrown when loading and evaluating the pre-trained model.
15:
16: ----- Test Errors -----
17: /lab_venv/lib/python3.10/site-packages/torch/nn/modules/loss.py:530: UserWarning: Using a target size (torch.Size([15447])) that is different to the input size (
(torch.Size([16512])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.
18:     return F.mse_loss(input, target, reduction=self.reduction)
```