

Bangladesh University of Engineering and Technology

EEE 6608

Machine Learning & Pattern Recognition

Midterm assignment: Logistic regression and fully connected neural
network from scratch

Submitted by,
Name: Md. Al-Imran Abir
Student ID: 0421062549

Date of Submission: February 12, 2022

1 Logistic Regression

The logistic regression algorithm was implemented from scratch using mainly the Python library NumPy. The SkLearn library was used only to split the training dataset into training and validation set.

1.1 Code

The `class`¹ for logistic regression is defined as below:

```
1 class LogisticRegression:
2     def __init__(self,x,y):
3         self.intercept = np.ones((x.shape[0], 1))
4         self.x = np.concatenate((self.intercept, x), axis=1)
5         self.weight = np.zeros(self.x.shape[1])
6         self.y = y
7
8     #Sigmoid function
9     def sigmoid(self, x, weight):
10        z = np.dot(x, weight)
11        #a = 1 / (1 + np.exp(-z)) #produces a warning if z is very small
12        a = .5 * (1 + np.tanh(.5 * z)) #so replaced that with this
13        return a
14
15    #calculate the Loss
16    def loss(self, h, y):
17        epsilon = 1e-10
18        J = (-y * np.log(h+epsilon) - (1-y) * np.log(1-h+epsilon)).mean()
19        return J
20
21    #calculating the gradients
22    def gradient_descent(self, X, h, y):
23        delJ = np.dot(X.T, (h - y)) / y.shape[0]
24        return delJ
25
26    def fit(self, lr , iterations):
27        J = np.zeros(iterations)
28        for i in range(iterations):
29            sigma = self.sigmoid(self.x, self.weight)
30            J[i] = self.loss(sigma,self.y)
31            delJ = self.gradient_descent(self.x , sigma, self.y)
32            self.weight -= lr * delJ
33        return J
34
35    #Prediction
36    def predict(self, x_new):
37        intercept_new = np.ones((x_new.shape[0], 1))
38        x_new = np.concatenate((intercept_new, x_new), axis=1)
39        result = self.sigmoid(x_new, self.weight)
40        y_pred = np.round(result)
41        return y_pred
```

The full code is

¹Got some help from [Logistic Regression From Scratch in Python \[Algorithm Explained\]](#)

```

1  import numpy as np # linear algebra
2  import h5py
3  import matplotlib.pyplot as plt
4  from sklearn.model_selection import train_test_split
5  import time
6  %matplotlib inline
7
8  # Random state seed
9  seed = 1234
10
11 def load_dataset():
12     train_dataset = h5py.File('/kaggle/input/happy-dataset/train_happy.h5', "r")
13     test_dataset = h5py.File('/kaggle/input/happy-dataset/test_happy.h5', "r")
14     train_set_x_orig = np.array(train_dataset["train_set_x"][:])
15     train_set_y_orig = np.array(train_dataset["train_set_y"][:])
16     test_set_x_orig = np.array(test_dataset["test_set_x"][:])
17     test_set_y_orig = np.array(test_dataset["test_set_y"][:])
18     classes = np.array(test_dataset["list_classes"][:])
19
20     train_set_y_orig = np.transpose(train_set_y_orig.reshape((1, train_set_y_orig.shape[0])))
21     test_set_y_orig = np.transpose(test_set_y_orig.reshape((1, test_set_y_orig.shape[0])))
22
23     return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes
24
25 train_set_X, train_set_y, test_set_X, test_set_y, classes = load_dataset()
26
27 def create_validation_set(train_set_X, train_set_y, test_size=0.2, random_state=seed):
28     """
29     Divides the training set into training and validation set
30
31     Input:
32     - train_set_X: Training set samples containing only features (no label)
33     - train_set_y: Training set labels
34     - test_size: (optional) % of training data to be separated as validation data
35
36     Output:
37     - X_train: Training samples
38     - y_train: Training labels
39     - X_valid: Validation samples
40     - y_valid: Validation labels
41     """
42     X_train, X_valid, y_train, y_valid = train_test_split(train_set_X, train_set_y, test_size=test_size, random_state=seed)
43     print("Train set size:", X_train.shape)
44     print("Validation set size:", X_valid.shape)
45     return X_train, y_train, X_valid, y_valid
46
47 train_X, y_train, valid_X, y_valid = create_validation_set(train_set_X, train_set_y, test_size=0.2)

```

```

48 # Preparing the data
49 X_train = train_X.reshape(480, 64*64*3)
50 X_valid = valid_X.reshape(120, 64*64*3)
51 test_set_X = test_set_X.reshape(150, 64*64*3)
52
53 def normalization(x, mu, std):
54     """
55     Normalization
56
57     Input:
58     - x: data
59     - mu: average
60     - std: standard deviation
61
62     Output:
63     - x_scaled: normalized output
64     """
65     x_scaled = (x-mu)/std
66     return x_scaled
67
68 # Normalize the data
69 mean_X = X_train.mean()
70 std_X = X_train.std()
71 X_train_scl = normalization(X_train, mean_X, std_X)
72 X_valid_scl = normalization(X_valid, mean_X, std_X)
73 X_test_scl = normalization(test_set_X, mean_X, std_X)
74
75 m = X_train.shape[0] # no of training samples
76 n = X_train.shape[1] # no of features
77
78 alpha = 1e-3 # learning rate
79 iters = 1000 # no of iterations
80
81 class LogisticRegression:
82     def __init__(self,x,y):
83         self.intercept = np.ones((x.shape[0], 1))
84         self.x = np.concatenate((self.intercept, x), axis=1)
85         self.weight = np.zeros(self.x.shape[1])
86         self.y = y
87
88     #Sigmoid function
89     def sigmoid(self, x, weight):
90         z = np.dot(x, weight)
91         # a = 1 / (1 + np.exp(-z)) #it produces a warning if z is very small
92         a = .5 * (1 + np.tanh(.5 * z)) #so replaced that with this
93         return a
94
95     #calculate the Loss

```

```

96     def loss(self, h, y):
97         epsilon = 1e-10
98         J = (-y * np.log(h+epsilon) - (1-y) * np.log(1-h+epsilon)).mean()
99         return J
100
101     #calculating the gradients
102     def gradient_descent(self, X, h, y):
103         delJ = np.dot(X.T, (h - y)) / y.shape[0]
104         return delJ
105
106
107     def fit(self, lr , iterations):
108         J = np.zeros(iterations)
109         for i in range(iterations):
110             sigma = self.sigmoid(self.x, self.weight)
111             J[i] = self.loss(sigma,self.y)
112             delJ = self.gradient_descent(self.x , sigma, self.y)
113             self.weight -= lr * delJ
114         return J
115
116     #Prediction
117     def predict(self, x_new):
118         intercept_new = np.ones((x_new.shape[0], 1))
119         x_new = np.concatenate((intercept_new, x_new), axis=1)
120         result = self.sigmoid(x_new, self.weight)
121         y_pred = np.round(result)
122         return y_pred
123
124 regressor = LogisticRegression(X_train_scl, y_train.flatten())
125 J = regressor.fit(0.002 , iters)
126 y_pred = regressor.predict(X_test_scl)
127
128 test_acc = (y_pred == test_set_y).sum()/150
129 print(test_acc)
130
131 plt.plot(range(0,iters), J)
132 plt.ylabel("Loss")
133 plt.xlabel("No of iterations")
134 plt.savefig("lr_0.002_Loss_vs_iterations.png")
135 plt.show()
136
137 # Plotting test accuracy vs learning rate
138 alpha_lr = np.array([1e-5, 5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1])
139 test_acc_lr=np.zeros(alpha_lr.shape[0])
140 idx=0
141
142 for lr in alpha_lr:
143     print("learning rate", lr)

```

```

144     regressor_lr = LogisticRegression(X_train_scl, y_train.flatten())
145     J_lr = regressor_lr.fit(lr , iters)
146     y_test_pred_lr = regressor_lr.predict(X_test_scl)
147     test_acc_lr[idx] = (y_test_pred_lr == test_set_y).sum()/150
148     print("test accuracy",test_acc_lr[idx])
149     plt.figure(idx)
150     plt.plot(range(0,iters), J_lr)
151     plt.show()
152     print("\n")
153     idx += 1
154
155     plt.plot(alpha_lr, test_acc_lr)
156     plt.xscale('log')
157     plt.ylabel("Test accuracy")
158     plt.xlabel("Learning rate")
159     plt.savefig("test_acc_vs_lr.png")
160     plt.show()
161
162     def select_subset_from_dataset(imgs, labels, ratio, shuffle=True, seed=1234):
163         """
164         Args:
165             imgs: numpy array representing the image set from which
166                   the selection is made.
167             labels: the labels associated with the provided images.
168             ratio (optional): portion of the data to be selected. Default: 0.1.
169             shuffle (optional): Whether or not to shuffle the data. Default: True.
170             seed (optional): seed of the numpy random generator: Default: 1234.
171
172         Return:
173             select_imgs: a numpy array of the selected images.
174             select_labels: labels associated with the selected images.
175
176         """
177         if shuffle:
178             np.random.seed(seed) # Set the random seed of numpy.
179             indices = np.random.permutation(imgs.shape[0])
180         else:
181             indices = np.arange(imgs.shape[0])
182         idx, _ = np.split(indices, [int(ratio*len(indices))])
183         select_imgs = imgs[idx]
184         tgt = np.array(labels)
185         select_labels = tgt[idx].tolist()
186         return select_imgs, select_labels
187
188
189     # Plotting test accuracy vs training sample size
190     percent_data = np.linspace(0.1, 1.0, 19)
191     # print(percent_data)

```

```

192 test_acc_pr=np.zeros(percent_data.shape[0])
193 ix = 0
194 for pr in percent_data:
195     print("training set size", pr)
196     pr_train_X, pr_train_y = select_subset_from_dataset(X_train_scl, y_train, ratio=pr)
197     regressor_pr = LogisticRegression(pr_train_X, pr_train_y.flatten())
198     regressor_pr.fit(alpha , iters)
199     y_test_pred_pr = regressor_pr.predict(X_test_scl)
200     test_acc_pr[ix] = (y_test_pred_pr == test_set_y).sum()/150
201     print("test accuracy:",test_acc_pr[ix],"\n")
202     ix += 1
203
204 plt.plot(percent_data, test_acc_pr, '*')
205 plt.xlabel('Fraction of training data')
206 plt.ylabel("Test accuracy")
207 plt.savefig("Percentage_Train.png")
208 plt.show()

```

1.2 Results and Discussions

1.2.1 Loss vs iteration

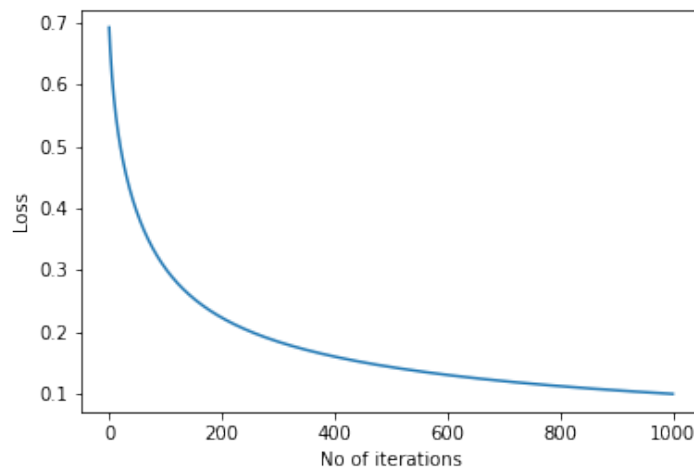


Figure 1: Plot of loss against iteration

As can be seen from the graph (Figure 1), the loss is decreasing with iteration which validates the model.

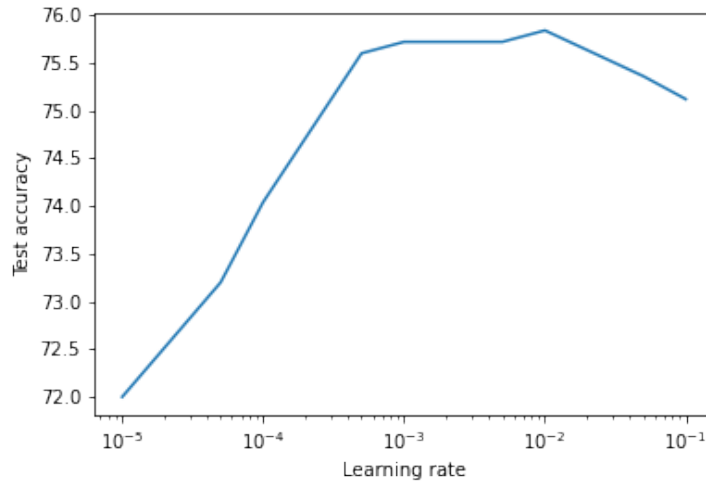


Figure 2: Plot of loss against iteration

1.2.2 Test accuracy vs learning rate

As the no of iterations were keep fixed, the test accuracy first increased with learning rate, reached a maximum and then again decreased (Figure 2). The optimum value of learning rate was 10^{-3} . Though for 0.01, the test accuracy was maximum but at that learning rate loss didn't decrease monotonically. So, that value was discarded.

1.2.3 Test accuracy vs training sample size

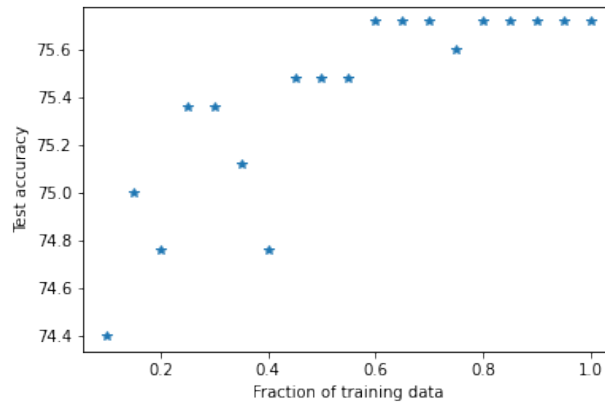


Figure 3: Plot of loss against iteration

As expected, highest accuracy was found with highest number of training samples. But the test accuracy didn't increase monotonically with the increase in training sample. To plot this graph (Figure 3), 10% to 100% of the training samples were used.

2 Fully Connected Neural Network

The fully connected neural networks were implemented using the Python framework PyTorch. But no high level APIs (Module and Sequential) of PyTorch was used. Rather barebones PyTorch was used.

The functions for defining the neural network is given below:

2.1 Code

```

1  def flatten_2d(x):
2      m = x.shape[0]
3      flat_x = x.view(-1, m) # shape n by m
4      return flat_x
5  def two_layer_fc(x, params):
6      a0 = flatten_2d(x) # n by m
7      w1, b1, w2, b2 = params
8
9      a1 = F.relu(w1.mm(a0) + b1)
10     a2 = torch.sigmoid(w2.mm(a1) + b2)
11     return a2
12 def three_layer_fc(x, params):
13     a0 = flatten_2d(x) # n by m
14     w1, b1, w2, b2, w3, b3 = params
15
16     a1 = F.relu(w1.mm(a0) + b1)
17     a2 = F.relu(w2.mm(a1) + b2)
18     a3 = torch.sigmoid(w3.mm(a2) + b3)
19     return a3
20 def four_layer_fc(x, params):
21     a0 = flatten_2d(x) # n by m
22     w1, b1, w2, b2, w3, b3, w4, b4 = params
23
24     a1 = F.relu(w1.mm(a0) + b1)
25     a2 = F.relu(w2.mm(a1) + b2)
26     a3 = F.relu(w3.mm(a2) + b3)
27     a4 = torch.sigmoid(w4.mm(a3) + b4)
28     return a4
29
30 def random_weight(shape):
31     w = torch.randn(shape, dtype=dtype) * np.sqrt(2. / connections)
32     w.requires_grad = True
33     return w
34
35 def zero_weight(shape):
36     return torch.zeros(shape, dtype=dtype, requires_grad=True)
37
38 def check_accuracy(x, y, model_fn, params):
39     with torch.no_grad():

```

```

40     preds = torch.round(model_fn(x, params))
41     acc = (preds == torch.transpose(y, 0, 1)).sum() / preds.size(1)
42     print("Accuracy", 100 * acc)
43     return preds, acc
44
45 def train_part2(X_train, y_train, X_valid, y_valid, model_fn, params, learning_rate, iters):
46     x = X_train.to(dtype=dtype)
47     y = y_train.to(dtype=dtype)
48
49     scores = model_fn(x, params)
50     loss = F.binary_cross_entropy(scores, torch.transpose(y, 0, 1))
51     loss.backward()
52
53     with torch.no_grad():
54         for w in params:
55             w -= learning_rate * w.grad
56
57             # Manually zero the gradients after running the backward pass
58             w.grad.zero_()
59
60     print('Loss = %.4f' % (loss.item()))
61     preds, acc = check_accuracy(X_valid, y_valid, model_fn, params)
62     print("Final validation accuracy", acc)
63     return J, preds, acc

```

The full code is given below:

```

1  import numpy as np
2  import h5py
3  import matplotlib.pyplot as plt
4  from sklearn.model_selection import train_test_split
5  import torch
6  import torch.nn as nn
7  import torch.nn.functional as F
8
9  %matplotlib inline
10
11 dtype = torch.float32
12
13 # Random state seed
14 seed = 1234
15 # For reproducibility
16 torch.manual_seed(seed)
17 torch.use_deterministic_algorithms(True)
18
19 def load_dataset():
20     train_dataset = h5py.File('/kaggle/input/happy-dataset/train_happy.h5', "r")
21     test_dataset = h5py.File('/kaggle/input/happy-dataset/test_happy.h5', "r")
22
23     train_set_x_orig = np.array(train_dataset["train_set_x"][:])
24     train_set_y_orig = np.array(train_dataset["train_set_y"][:])
25     test_set_x_orig = np.array(test_dataset["test_set_x"][:])
26     test_set_y_orig = np.array(test_dataset["test_set_y"][:])
27     classes = np.array(test_dataset["list_classes"][:])
28
29     train_set_y_orig = np.transpose(train_set_y_orig.reshape((1, train_set_y_orig.shape[0])))

```

```

30     test_set_y_orig = np.transpose(test_set_y_orig.reshape((1, test_set_y_orig.shape[0])))
31
32     return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes
33
34 train_set_X, train_set_y, test_set_X, test_set_y, classes = load_dataset()
35
36 def create_validation_set(train_set_X, train_set_y, test_size=0.2, random_state=seed):
37     """
38     Divides the training set into training and validation set
39
40     Input:
41     - train_set_X: Training set samples containing only features (no label)
42     - train_set_y: Training set labels
43     - test_size: (optional) % of training data to be separated as validation data
44
45     Output:
46     - X_train: Training samples
47     - y_train: Training labels
48     - X_valid: Validation samples
49     - y_valid: Validation labels
50     """
51     X_train, X_valid, y_train, y_valid = train_test_split(train_set_X, train_set_y, test_size=test_size, random_state=seed)
52     print("Train set size:", X_train.shape)
53     print("Validation set size:", X_valid.shape)
54     return X_train, y_train, X_valid, y_valid
55
56 train_X, y_train, valid_X, y_valid = create_validation_set(train_set_X, train_set_y, test_size=0.2)
57 print("Train set size:", train_X.shape)
58 print("Train label size:", y_train.shape)
59 print("Validation set size:", valid_X.shape)
60 print("Validation set size:", y_valid.shape)
61
62 # Preparing the data
63 X_train = train_X.reshape(480, 64*64*3)
64 X_valid = valid_X.reshape(120, 64*64*3)
65 test_set_X = test_set_X.reshape(150, 64*64*3)
66 print("Training data shape", X_train.shape)
67 print("Validation data shape", X_valid.shape)
68 print("Test data shape", test_set_X.shape)
69
70 def normalization(x, mu, std):
71     """
72     Normalization
73
74     Input:
75     - x: data
76     - mu: average
77     - std: standard deviation
78
79     Output:
80     - x_scaled: normalized output
81     """
82     x_scaled = (x-mu)/std
83     return x_scaled
84
85 # Normalize the data
86 mean_X = X_train.mean()
87 print(mean_X)

```

```

88     std_X = X_train.std()
89     print(std_X)
90     X_train_scl = normalization(X_train, mean_X, std_X)
91     X_valid_scl = normalization(X_valid, mean_X, std_X)
92     X_test_scl = normalization(test_set_X, mean_X, std_X)
93
94     m = X_train.shape[0] # no of training samples
95     n = X_train.shape[1] # no of features
96     X_train_app = np.c_[np.ones(m), X_train_scl] # append a column of 1
97     X_valid_app = np.c_[np.ones(120), X_valid_scl] # append a column of 1
98     X_test_app = np.c_[np.ones(150), X_test_scl] # append a column of 1
99     alpha = 1e-3 # learning rate
100    iters = 1000 # no of iterations
101
102    def flatten_2d(x):
103        m = x.shape[0]
104        flat_x = x.view(-1, m) # shape n by m
105        return flat_x
106    def two_layer_fc(x, params):
107        a0 = flatten_2d(x) # n by m
108        w1, b1, w2, b2 = params
109
110        a1 = F.relu(w1.mm(a0) + b1)
111        a2 = torch.sigmoid(w2.mm(a1) + b2)
112        return a2
113    def three_layer_fc(x, params):
114        a0 = flatten_2d(x) # n by m
115        w1, b1, w2, b2, w3, b3 = params
116
117        a1 = F.relu(w1.mm(a0) + b1)
118        a2 = F.relu(w2.mm(a1) + b2)
119        a3 = torch.sigmoid(w3.mm(a2) + b3)
120        return a3
121    def four_layer_fc(x, params):
122        a0 = flatten_2d(x) # n by m
123        w1, b1, w2, b2, w3, b3, w4, b4 = params
124
125        a1 = F.relu(w1.mm(a0) + b1)
126        a2 = F.relu(w2.mm(a1) + b2)
127        a3 = F.relu(w3.mm(a2) + b3)
128        a4 = torch.sigmoid(w4.mm(a3) + b4)
129        return a4
130
131    def random_weight(shape):
132        w = torch.randn(shape, dtype=dtype) * np.sqrt(2. / connections)
133        w.requires_grad = True
134        return w
135
136    def zero_weight(shape):
137        return torch.zeros(shape, dtype=dtype, requires_grad=True)
138
139    def check_accuracy(x, y, model_fn, params):
140        with torch.no_grad():
141            preds = torch.round(model_fn(x, params))
142            acc = (preds == torch.transpose(y, 0, 1)).sum() / preds.size(1)
143            print("Accuracy", 100 * acc)
144        return preds, acc
145

```

```

146 def train_part2(X_train, y_train, X_valid, y_valid, model_fn, params, learning_rate, iters):
147     x = X_train.to(dtype=dtype)
148     y = y_train.to(dtype=dtype)
149
150     scores = model_fn(x, params)
151     loss = F.binary_cross_entropy(scores, torch.transpose(y, 0, 1))
152     loss.backward()
153
154     with torch.no_grad():
155         for w in params:
156             w -= learning_rate * w.grad
157
158             # Manually zero the gradients after running the backward pass
159             w.grad.zero_()
160
161     print('Loss = %.4f' % (loss.item()))
162     preds, acc = check_accuracy(X_valid, y_valid, model_fn, params)
163     print("Final validation accuracy", acc)
164     return J, preds, acc
165
166 hidden_layer_size1 = 6145
167 hidden_layer_size2 = 3173
168 hidden_layer_size3 = 1587
169 learning_rate = 1e-3
170 iters = 1000 # no of iterations
171
172 # parametes for 2 level FC NN
173 # w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
174 # b1 = zero_weight((hidden_layer_size1, 1))
175 # w2 = random_weight((1, hidden_layer_size1))
176 # b2 = zero_weight((1, 1))
177 # params = [w1, b1, w2, b2]
178
179 # parametes for 3 level FC NN
180 # w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
181 # b1 = zero_weight((hidden_layer_size1, 1))
182 # w2 = random_weight((hidden_layer_size2, hidden_layer_size1))
183 # b2 = zero_weight((hidden_layer_size2, 1))
184 # w3 = random_weight((1, hidden_layer_size2))
185 # b3 = zero_weight((1, 1))
186 # params = [w1, b1, w2, b2, w3, b3]
187
188 # parametes for 4 level FC NN
189 w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
190 b1 = zero_weight((hidden_layer_size1, 1))
191 w2 = random_weight((hidden_layer_size2, hidden_layer_size1))
192 b2 = zero_weight((hidden_layer_size2, 1))
193 w3 = random_weight((hidden_layer_size3, hidden_layer_size2))
194 b3 = zero_weight((hidden_layer_size3, 1))
195 w4 = random_weight((1, hidden_layer_size3))
196 b4 = zero_weight((1, 1))
197 params = [w1, b1, w2, b2, w3, b3, w4, b4]
198
199 # convert numpy array to tensors
200 X_tr_tensor = torch.from_numpy(X_train_app)
201 Y_tr_tensor = torch.from_numpy(y_train)
202 X_val_tensor = (torch.from_numpy(X_valid_app)).to(dtype=dtype)
203 Y_val_tensor = (torch.from_numpy(y_valid)).to(dtype=dtype)

```

204

205

```
J, preds, acc = train_part2(X_tr_tensor, Y_tr_tensor, X_val_tensor, Y_val_tensor, four_layer_fc, params, learning_rate, ite
```

2.2 Results and Discussion

2.2.1 Two and three layer fully connected neural network

The test accuracy varied from 45% to 52% for two and three layer fully connected NNs. To find the optimal hyper-parameters, the learning rate and the no of nodes in the hidden layer were varied. But (most probably) due to the internal randomness of PyTorch, result couldn't be reproduced (tried to reproduce using the commands `torch.manual_seed(seed)` and `torch.use_deterministic_algorithms(True)` but didn't work).

2.2.2 Four layer fully connected neural network

For four level network, the test accuracy varied from 48% 57%. Similar to the previous cases, exact results couldn't be reproduced. 6145, 3173, 1587 nodes were used in the three hidden layers respectively.

3 Comparison

Surprisingly, the logistic regression performed better than the fully connected neural network.

Table 1: Test accuracy for logistic regression and fully connected neural network.

Metric	Logistic Regression	FC Neural Network
Accuracy	75.75%	57.82%