Bangladesh University of Engineering and Technology

EEE 6608

Machine Learning & Pattern Recognition

Midterm assignement: Logistic regression and fully connectted 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:

```
class LogisticRegression:
        def __init__(self,x,y):
            self.intercept = np.ones((x.shape[0], 1))
            self.x = np.concatenate((self.intercept, x), axis=1)
            self.weight = np.zeros(self.x.shape[1])
            self.y = y

        #Sigmoid function
        def sigmoid(self, x, weight):
            z = np.dot(x, weight)
            #a = 1 / (1 + np.exp(-z)) #produces a warning if z is very small
            a = .5 * (1 + np.tanh(.5 * z)) #so replaced that with this
            return a

        #calculate the Loss
        def loss(self, h, y):
            epsilon = 1e-10
            J = (-y * np.log(h+epsilon) - (1-y) * np.log(1-h+epsilon)).mean()
            return J

        #calculating the gradients
        def gradient_descent(self, X, h, y):
            delJ = np.dot(X.T, (h - y)) / y.shape[0]
            return delJ

        def fit(self, lr , iterations):
            J = np.zeros(iterations)
            for i in range(iterations):
            sigma = self.sigmoid(self.x, self.weight)
            J[i] = self.loss(sigma,self.y)
            delJ = self.gradient_descent(self.x , sigma, self.y)
            self.weight -= lr * delJ
            return J

        #Prediction
        def predict(self, x_new):
            intercept_new = np.ones((x_new.shape[0], 1))
            x_new = np.concatenate((intercept_new, x_new), axis=1)
            result = self.sigmoid(x_new, self.weight)
            y_pred = np.round(result)
            return y_pred
```

The full code is

[1] Got some help from Logistic Regression From Scratch in Python [Algorithm Explained]

```python
import numpy as np # linear algebra
import h5py
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import time
%matplotlib inline

# Random state seed
seed = 1234

def load_dataset():
    train_dataset = h5py.File('/kaggle/input/happy-dataset/train_happy.h5', "r")
    test_dataset = h5py.File('/kaggle/input/happy-dataset/test_happy.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:])
    train_set_y_orig = np.array(train_dataset["train_set_y"][:])
    test_set_x_orig = np.array(test_dataset["test_set_x"][:])
    test_set_y_orig = np.array(test_dataset["test_set_y"][:])
    classes = np.array(test_dataset["list_classes"][:])

    train_set_y_orig = np.transpose(train_set_y_orig.reshape((1, train_set_y_orig.shape[0])))
    test_set_y_orig = np.transpose(test_set_y_orig.reshape((1, test_set_y_orig.shape[0])))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes

train_set_X, train_set_y, test_set_X, test_set_y, classes = load_dataset()

def create_validation_set(train_set_X, train_set_y, test_size=0.2, random_state=seed):
    """
    Divides the training set into training and validation set

    Input:
    - train_set_X: Training set samples containing only features (no label)
    - train_set_y: Training set labels
    - test_size: (optional) % of training data to be separated as validation data

    Output:
    - X_train: Training samples
    - y_train: Training labels
    - X_valid: Validation samples
    - y_valid: Validation labels
    """
    X_train, X_valid, y_train, y_valid = train_test_split(train_set_X, train_set_y, test_size=test_size, random_state=seed)
    print("Train set size:", X_train.shape)
    print("Validation set size:", X_valid.shape)
    return X_train, y_train, X_valid, y_valid

train_X, y_train, valid_X, y_valid = create_validation_set(train_set_X, train_set_y, test_size=0.2)
```

```python
# Preparing the data
X_train = train_X.reshape(480, 64*64*3)
X_valid = valid_X.reshape(120, 64*64*3)
test_set_X = test_set_X.reshape(150, 64*64*3)

def normalization(x, mu, std):
    """
    Normalization

    Input:
    - x: data
    - mu: average
    - std: standard deviation

    Output:
    - x_scaled: normalized output
    """
    x_scaled = (x-mu)/std
    return x_scaled

# Normalize the data
mean_X = X_train.mean()
std_X = X_train.std()
X_train_scl = normalization(X_train, mean_X, std_X)
X_valid_scl = normalization(X_valid, mean_X, std_X)
X_test_scl = normalization(test_set_X, mean_X, std_X)

m = X_train.shape[0] # no of training samples
n = X_train.shape[1] # no of features

alpha = 1e-3 # learning rate
iters = 1000 # no of iterations

class LogisticRegression:
    def __init__(self,x,y):
        self.intercept = np.ones((x.shape[0], 1))
        self.x = np.concatenate((self.intercept, x), axis=1)
        self.weight = np.zeros(self.x.shape[1])
        self.y = y

    #Sigmoid function
    def sigmoid(self, x, weight):
        z = np.dot(x, weight)
#         a = 1 / (1 + np.exp(-z)) #it produces a warning if z is very small
        a = .5 * (1 + np.tanh(.5 * z)) #so replaced that with this
        return a

    #calculate the Loss
```

```
    def loss(self, h, y):
        epsilon = 1e-10
        J = (-y * np.log(h+epsilon) - (1-y) * np.log(1-h+epsilon)).mean()
        return J

    #calculating the gradients
    def gradient_descent(self, X, h, y):
        delJ = np.dot(X.T, (h - y)) / y.shape[0]
        return delJ


    def fit(self, lr , iterations):
        J = np.zeros(iterations)
        for i in range(iterations):
            sigma = self.sigmoid(self.x, self.weight)
            J[i] = self.loss(sigma,self.y)
            delJ = self.gradient_descent(self.x , sigma, self.y)
            self.weight -= lr * delJ
        return J

    #Prediction
    def predict(self, x_new):
        intercept_new = np.ones((x_new.shape[0], 1))
        x_new = np.concatenate((intercept_new, x_new), axis=1)
        result = self.sigmoid(x_new, self.weight)
        y_pred = np.round(result)
        return y_pred

regressor = LogisticRegression(X_train_scl, y_train.flatten())
J = regressor.fit(0.002 , iters)
y_pred = regressor.predict(X_test_scl)

test_acc = (y_pred == test_set_y).sum()/150
print(test_acc)

plt.plot(range(0,iters), J)
plt.ylabel("Loss")
plt.xlabel("No of iterations")
plt.savefig("lr_0.002_Loss_vs_iterations.png")
plt.show()

# Plotting test accuracy vs learning rate
alpha_lr = np.array([1e-5, 5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1])
test_acc_lr=np.zeros(alpha_lr.shape[0])
idx=0

for lr in alpha_lr:
    print("learning rate", lr)
```

```
    regressor_lr = LogisticRegression(X_train_scl, y_train.flatten())
    J_lr = regressor_lr.fit(lr , iters)
    y_test_pred_lr = regressor_lr.predict(X_test_scl)
    test_acc_lr[idx] = (y_test_pred_lr == test_set_y).sum()/150
    print("test accuracy",test_acc_lr[idx])
    plt.figure(idx)
    plt.plot(range(0,iters), J_lr)
    plt.show()
    print("\n")
    idx += 1

plt.plot(alpha_lr, test_acc_lr)
plt.xscale('log')
plt.ylabel("Test accuracy")
plt.xlabel("Learning rate")
plt.savefig("test_acc_vs_lr.png")
plt.show()

def select_subset_from_dataset(imgs, labels, ratio, shuffle=True, seed=1234):
    """
    Args:
     imgs: numpy array representing the image set from which
        the selection is made.
     labels: the labels associated with the provided images.
     ratio (optional): portion of the data to be selected. Default: 0.1.
     shuffle (optional): Whether or not to shuffle the data. Default: True.
     seed (optional): seed of the numpy random generator: Default: 1234.

    Return:
        select_imgs: a numpy array of the selected images.
        select_labels: labels associated with the selected images.

    """
    if shuffle:
        np.random.seed(seed)  # Set the random seed of numpy.
        indices = np.random.permutation(imgs.shape[0])
    else:
        indices = np.arange(imgs.shape[0])
    idx, _ = np.split(indices, [int(ratio*len(indices))])
    select_imgs = imgs[idx]
    tgt = np.array(labels)
    select_labels = tgt[idx]#.tolist()
    return select_imgs, select_labels


# Plotting test accuracy vs training sample size
percent_data = np.linspace(0.1, 1.0, 19)
# print(percent_data)
```

```python
test_acc_pr=np.zeros(percent_data.shape[0])
ix = 0
for pr in percent_data:
    print("training set size", pr)
    pr_train_X, pr_train_y = select_subset_from_dataset(X_train_scl, y_train, ratio=pr)
    regressor_pr = LogisticRegression(pr_train_X, pr_train_y.flatten())
    regressor_pr.fit(alpha , iters)
    y_test_pred_pr = regressor_pr.predict(X_test_scl)
    test_acc_pr[ix] = (y_test_pred_pr == test_set_y).sum()/150
    print("test accuracy:",test_acc_pr[ix],"\n")
    ix += 1

plt.plot(percent_data, test_acc_pr, '*')
plt.xlabel('Fraction of training data')
plt.ylabel("Test accuracy")
plt.savefig("Percentage_Train.png")
plt.show()
```

1.2 Results and Discussions

1.2.1 Loss vs iteration

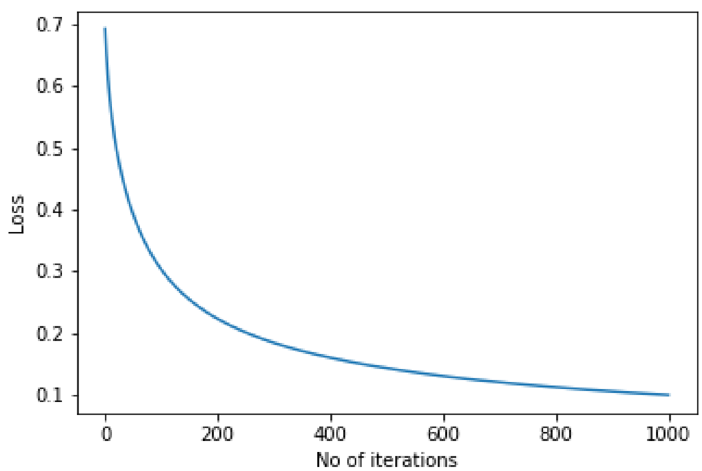

Figure 1: Plot of loss against iteration

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

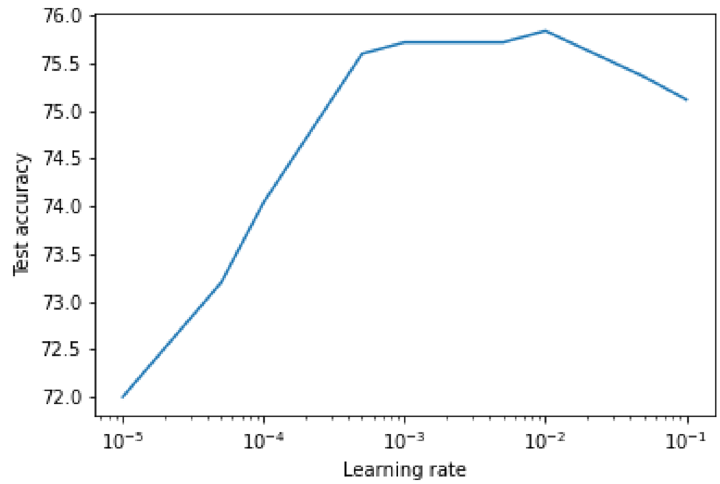

Figure 2: Plot of loss against iteration

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

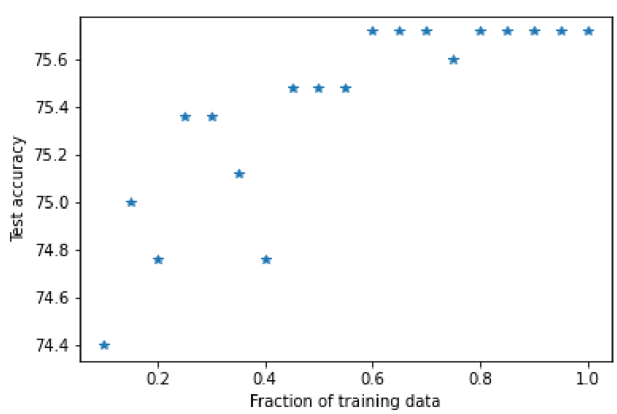

Figure 3: Plot of loss against iteration

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

```
def flatten_2d(x):
    m = x.shape[0]
    flat_x = x.view(-1, m) # shape n by m
    return flat_x
def two_layer_fc(x, params):
    a0 = flatten_2d(x)   # n by m
    w1, b1, w2, b2 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = torch.sigmoid(w2.mm(a1) + b2)
    return a2
def three_layer_fc(x, params):
    a0 = flatten_2d(x)   # n by m
    w1, b1, w2, b2, w3, b3 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = F.relu(w2.mm(a1) + b2)
    a3 = torch.sigmoid(w3.mm(a2) + b3)
    return a3
def four_layer_fc(x, params):
    a0 = flatten_2d(x)   # n by m
    w1, b1, w2, b2, w3, b3, w4, b4 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = F.relu(w2.mm(a1) + b2)
    a3 = F.relu(w3.mm(a2) + b3)
    a4 = torch.sigmoid(w4.mm(a3) + b4)
    return a4

def random_weight(shape):
    w = torch.randn(shape, dtype=dtype) * np.sqrt(2. / connections)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, dtype=dtype, requires_grad=True)

def check_accuracy(x, y, model_fn, params):
    with torch.no_grad():
```

```
        preds = torch.round(model_fn(x, params))
        acc = (preds == torch.transpose(y, 0, 1)).sum() / preds.size(1)
        print("Accuracy", 100 * acc)
    return preds, acc

def train_part2(X_train, y_train, X_valid, y_valid, model_fn, params, learning_rate, iters):
    x = X_train.to(dtype=dtype)
    y = y_train.to(dtype=dtype)

    scores = model_fn(x, params)
    loss = F.binary_cross_entropy(scores, torch.transpose(y, 0, 1))
    loss.backward()

    with torch.no_grad():
        for w in params:
            w -= learning_rate * w.grad

            # Manually zero the gradients after running the backward pass
            w.grad.zero_()

    print('Loss = %.4f' % (loss.item()))
    preds, acc = check_accuracy(X_valid, y_valid, model_fn, params)
    print("Final validation accuracy", acc)
    return J, preds, acc
```

The full code is given below:

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.nn.functional as F

%matplotlib inline

dtype = torch.float32

# Random state seed
seed = 1234
# For reproducibility
torch.manual_seed(seed)
torch.use_deterministic_algorithms(True)

def load_dataset():
    train_dataset = h5py.File('/kaggle/input/happy-dataset/train_happy.h5', "r")
    test_dataset = h5py.File('/kaggle/input/happy-dataset/test_happy.h5', "r")

    train_set_x_orig = np.array(train_dataset["train_set_x"][:])
    train_set_y_orig = np.array(train_dataset["train_set_y"][:])
    test_set_x_orig = np.array(test_dataset["test_set_x"][:])
    test_set_y_orig = np.array(test_dataset["test_set_y"][:])
    classes = np.array(test_dataset["list_classes"][:])

    train_set_y_orig = np.transpose(train_set_y_orig.reshape((1, train_set_y_orig.shape[0])))
```

```
    test_set_y_orig = np.transpose(test_set_y_orig.reshape((1, test_set_y_orig.shape[0])))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes

train_set_X, train_set_y, test_set_X, test_set_y, classes = load_dataset()

def create_validation_set(train_set_X, train_set_y, test_size=0.2, random_state=seed):
    """
    Divides the training set into training and validation set

    Input:
    - train_set_X: Training set samples containing only features (no label)
    - train_set_y: Training set labels
    - test_size: (optional) % of training data to be separated as validation data

    Output:
    - X_train: Training samples
    - y_train: Training labels
    - X_valid: Validation samples
    - y_valid: Validation labels
    """
    X_train, X_valid, y_train, y_valid = train_test_split(train_set_X, train_set_y, test_size=test_size, random_state=seed)
    print("Train set size:", X_train.shape)
    print("Validation set size:", X_valid.shape)
    return X_train, y_train, X_valid, y_valid

train_X, y_train, valid_X, y_valid = create_validation_set(train_set_X, train_set_y, test_size=0.2)
print("Train set size:", train_X.shape)
print("Train label size:", y_train.shape)
print("Validation set size:", valid_X.shape)
print("Validation set size:", y_valid.shape)

# Preparing the data
X_train = train_X.reshape(480, 64*64*3)
X_valid = valid_X.reshape(120, 64*64*3)
test_set_X = test_set_X.reshape(150, 64*64*3)
print("Training data shape", X_train.shape)
print("Validation data shape", X_valid.shape)
print("Test data shape", test_set_X.shape)

def normalization(x, mu, std):
    """
    Normalization

    Input:
    - x: data
    - mu: average
    - std: standard deviation

    Output:
    - x_scaled: normalized output
    """
    x_scaled = (x-mu)/std
    return x_scaled

# Normalize the data
mean_X = X_train.mean()
print(mean_X)
```

```
std_X = X_train.std()
print(std_X)
X_train_scl = normalization(X_train, mean_X, std_X)
X_valid_scl = normalization(X_valid, mean_X, std_X)
X_test_scl = normalization(test_set_X, mean_X, std_X)

m = X_train.shape[0] # no of training samples
n = X_train.shape[1] # no of features
X_train_app = np.c_[np.ones(m), X_train_scl] # append a column of 1
X_valid_app = np.c_[np.ones(120), X_valid_scl] # append a column of 1
X_test_app = np.c_[np.ones(150), X_test_scl] # append a column of 1
alpha = 1e-3 # learning rate
iters = 1000 # no of iterations

def flatten_2d(x):
    m = x.shape[0]
    flat_x = x.view(-1, m) # shape n by m
    return flat_x
def two_layer_fc(x, params):
    a0 = flatten_2d(x)    # n by m
    w1, b1, w2, b2 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = torch.sigmoid(w2.mm(a1) + b2)
    return a2
def three_layer_fc(x, params):
    a0 = flatten_2d(x)    # n by m
    w1, b1, w2, b2, w3, b3 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = F.relu(w2.mm(a1) + b2)
    a3 = torch.sigmoid(w3.mm(a2) + b3)
    return a3
def four_layer_fc(x, params):
    a0 = flatten_2d(x)    # n by m
    w1, b1, w2, b2, w3, b3, w4, b4 = params

    a1 = F.relu(w1.mm(a0) + b1)
    a2 = F.relu(w2.mm(a1) + b2)
    a3 = F.relu(w3.mm(a2) + b3)
    a4 = torch.sigmoid(w4.mm(a3) + b4)
    return a4

def random_weight(shape):
    w = torch.randn(shape, dtype=dtype) * np.sqrt(2. / connections)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, dtype=dtype, requires_grad=True)

def check_accuracy(x, y, model_fn, params):
    with torch.no_grad():
        preds = torch.round(model_fn(x, params))
        acc = (preds == torch.transpose(y, 0, 1)).sum() / preds.size(1)
        print("Accuracy", 100 * acc)
    return preds, acc

```

```python
def train_part2(X_train, y_train, X_valid, y_valid, model_fn, params, learning_rate, iters):
    x = X_train.to(dtype=dtype)
    y = y_train.to(dtype=dtype)

    scores = model_fn(x, params)
    loss = F.binary_cross_entropy(scores, torch.transpose(y, 0, 1))
    loss.backward()

    with torch.no_grad():
        for w in params:
            w -= learning_rate * w.grad

            # Manually zero the gradients after running the backward pass
            w.grad.zero_()

    print('Loss = %.4f' % (loss.item()))
    preds, acc = check_accuracy(X_valid, y_valid, model_fn, params)
    print("Final validation accuracy", acc)
    return J, preds, acc

hidden_layer_size1 = 6145
hidden_layer_size2 = 3173
hidden_layer_size3 = 1587
learning_rate = 1e-3
iters = 1000 # no of iterations

# parametes for 2 level FC NN
# w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
# b1 = zero_weight((hidden_layer_size1, 1))
# w2 = random_weight((1, hidden_layer_size1))
# b2 = zero_weight((1, 1))
# params = [w1, b1, w2, b2]

# parametes for 3 level FC NN
# w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
# b1 = zero_weight((hidden_layer_size1, 1))
# w2 = random_weight((hidden_layer_size2, hidden_layer_size1))
# b2 = zero_weight((hidden_layer_size2, 1))
# w3 = random_weight((1, hidden_layer_size2))
# b3 = zero_weight((1, 1))
# params = [w1, b1, w2, b2, w3, b3]

# parametes for 4 level FC NN
w1 = random_weight((hidden_layer_size1, 3 * 64 * 64 + 1))
b1 = zero_weight((hidden_layer_size1, 1))
w2 = random_weight((hidden_layer_size2, hidden_layer_size1))
b2 = zero_weight((hidden_layer_size2, 1))
w3 = random_weight((hidden_layer_size3, hidden_layer_size2))
b3 = zero_weight((hidden_layer_size3, 1))
w4 = random_weight((1, hidden_layer_size3))
b4 = zero_weight((1, 1))
params = [w1, b1, w2, b2, w3, b3, w4, b4]

# convert numpy array to tensors
X_tr_tensor = torch.from_numpy(X_train_app)
Y_tr_tensor = torch.from_numpy(y_train)
X_val_tensor = (torch.from_numpy(X_valid_app)).to(dtype=dtype)
Y_val_tensor = (torch.from_numpy(y_valid)).to(dtype=dtype)
```

```
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%