Vazirpanah-Arad-6103991982-HW3

June 21, 2023

Import everything we need such as torch and others from sklearn library.

```
import torch
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.neural_network import MLPClassifier
import pickle
from sklearn.cluster import KMeans
import random
```

this is MixMaxScaler. We use it to scale the data so we could use scaled data for calculating the loss to train the AutoEncoder we have.

```
scaler = MinMaxScaler()
```

Reading data from csv file and storing 784 columns in X_ltrain and first column in y_train. Also we scale the data set and convert it to torch array.

```
data = pd.read_csv("labeled_train_set.csv")
scaler.fit(data.iloc[:, 1:].values)
X_ltrain = torch.tensor(scaler.transform(data.iloc[:, 1:].values), dtype=torch.
 ↪float32) #all columns exept first one
y_ltrain = data['label'] #last column with all rows (the answer column)
```

Unlabeled data

```
data = pd.read_csv("unlabeled_train_set.csv")
X_ultrain = torch.tensor(scaler.transform(data.values), dtype=torch.float32)␣
 ↪#all columns exept first one
```

Test data

```
data = pd.read_csv("test_set.csv")
X_test = torch.tensor(scaler.transform(data.iloc[:, 1:].values), dtype=torch.
 ↪float32) #all columns exept first one
y_test = data['label']#last column with all rows (the answer column)
```

The class of AutoEncoder which has three layers and a forward function. The last layer has 36 neurons.

```
class AutoEncoder(torch.nn.Module):
        def __init__(self):
                super().__init__()

                self.encoder = torch.nn.Sequential(
                        torch.nn.Linear(28 * 28, 128),
                        torch.nn.ReLU(True),
                        torch.nn.Linear(128, 64),
                        torch.nn.ReLU(True),
                        torch.nn.Linear(64, 36),
                )

                self.decoder = torch.nn.Sequential(
                        torch.nn.ReLU(True),
                        torch.nn.Linear(36, 64),
                        torch.nn.ReLU(True),
                        torch.nn.Linear(64, 128),
                        torch.nn.ReLU(True),
                        torch.nn.Linear(128, 28 * 28),
                        torch.nn.Sigmoid()
                )

        def forward(self, x):
                encoded = self.encoder(x)
                decoded = self.decoder(encoded)
                return decoded
```

In these cells, we train the AutoEncoder model with labeled data set.

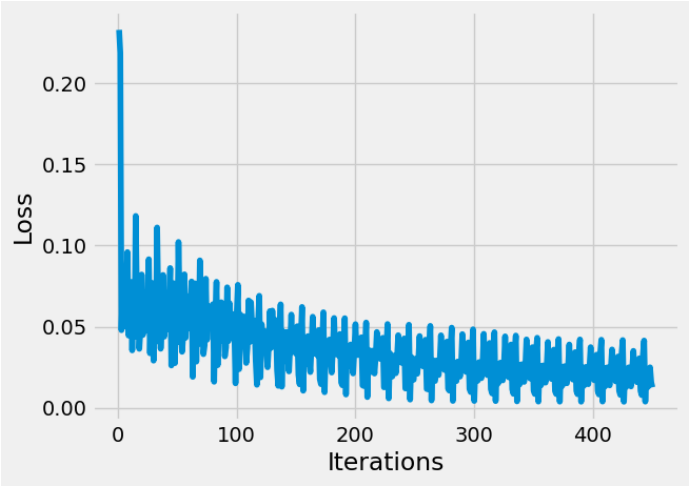
```
# Model Initialization
model = AutoEncoder()

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss()

# Using an SGD Optimizer with lr = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum= 0.9)
```

we train the AutoEncoder with 25 epochs while minimizing the loss.

```
epochs = 25
losses = np.array([], dtype=float)
loss = 0
for epoch in range(epochs):
        for i in range(len(X_ltrain)):
                # Output of Autoencoder
                reconstructed = model(X_ltrain[i])

                # Calculating the loss function
                optimizer.zero_grad()
                loss = loss_function(reconstructed, X_ltrain[i])

                # The gradients are set to zero,
                # the gradient is computed and stored.
                # .step() performs parameter update
                loss.backward()
                optimizer.step()

                # Storing the losses in a list for plotting
                losses = np.append([losses], float(loss))
```

Then plot the loss which is always decreasing.

```
# Defining the Plot Style
plt.style.use('fivethirtyeight')
plt.xlabel('Iterations')
plt.ylabel('Loss')

# Plotting the 1000 values
plt.plot([i+1 for i in range(int(len(losses)/1000))], losses[::1000])
plt.show()
```



This is the encoded data which is the output of AutoEncoder with input X_ltrain.

```
encoded_data = [model.encoder(X_ltrain[i]).detach().numpy() for i in␣
↪range(len(X_ltrain))]
```

Here are some advantages of using autoencoder features for classification:

1. Data representation:

Autoencoders learn a compressed representation of the input data by encoding it into a lower-dimensional latent space. These learned representations often capture important features and patterns in the data, making them suitable for classification tasks. By using autoencoder features, you can potentially achieve better data representation than using raw input features alone.

2. Dimensionality reduction

Autoencoders can effectively reduce the dimensionality of the input data. This is particularly useful when dealing with high-dimensional datasets, as it reduces computational complexity and helps mitigate the curse of dimensionality. By using autoencoder features, you can reduce the number of input features while preserving relevant information, which can lead to improved classification performance.

3. Noise reduction and robustness:

Autoencoders are capable of denoising the input data by learning to reconstruct clean versions of the original samples. This denoising capability can be beneficial for classification tasks, as it helps remove irrelevant or noisy features that may negatively impact the performance of classifiers. By using autoencoder features, you can potentially enhance the robustness of your classification model.

4. Feature learning:

Autoencoders can automatically learn hierarchical and abstract features from the data. As the autoencoder is trained to reconstruct the original input, the intermediate layers of the network capture increasingly complex features. These learned features can be more informative and discriminative than handcrafted features, allowing for improved classification accuracy. Autoencoders are especially advantageous when dealing with complex data distributions or when the relationship between input features and class labels is not well-defined.

5. Transfer learning:

Autoencoders can be pre-trained on large unlabeled datasets to learn useful representations of data. This pre-training can act as a form of transfer learning, where the learned features are transferred to a classification model. By utilizing the pre-trained autoencoder features, you can leverage the knowledge captured from the unlabeled data to improve the performance of the subsequent classification task, especially when labeled data is limited.

6. Non-linear transformations:

Autoencoders, especially when using deep neural network architectures, can learn non-linear transformations of the input data. This ability to capture complex relationships in the data can be particularly advantageous for classification tasks where the decision boundary between classes is non-linear. By using autoencoder features, you can potentially improve the model's ability to capture and represent these non-linear relationships, leading to better classification performance.

Here we check for the best hyper parameters on this data.

```
mlp_scores = np.zeros(12)
layers = [(100, 75), (100, 50), (75, 50)]
learning_rate = ['invscaling', 'adaptive']
activation = ['tanh', 'relu']
i = 0
for l in layers:
    for learn in learning_rate:
        for a in activation:
            score = 0
            print("layer: " + str(l) + " ,learn: " + learn + " ,activation: " +␣
 ↪a + " ,i: " + str(i+1))
            mlp = MLPClassifier(hidden_layer_sizes= l, learning_rate= learn ,␣
 ↪activation= a)
            score = cross_val_score(mlp, encoded_data, y_ltrain, cv=5)
            print("score: " + str(np.mean(score)))
            mlp_scores[i] = np.mean(score)
            i += 1
```

```
layer: (100, 75) ,learn: invscaling ,activation: tanh ,i: 1

score: 0.9455555555555556
layer: (100, 75) ,learn: invscaling ,activation: relu ,i: 2

score: 0.9443888888888889
layer: (100, 75) ,learn: adaptive ,activation: tanh ,i: 3

score: 0.9437777777777778
layer: (100, 75) ,learn: adaptive ,activation: relu ,i: 4

score: 0.9441111111111111
layer: (100, 50) ,learn: invscaling ,activation: tanh ,i: 5

score: 0.9431666666666667
layer: (100, 50) ,learn: invscaling ,activation: relu ,i: 6

score: 0.9435555555555556
layer: (100, 50) ,learn: adaptive ,activation: tanh ,i: 7

score: 0.9423333333333332
layer: (100, 50) ,learn: adaptive ,activation: relu ,i: 8

score: 0.9439444444444444
layer: (75, 50) ,learn: invscaling ,activation: tanh ,i: 9
```

```
score: 0.9407222222222222
layer: (75, 50) ,learn: invscaling ,activation: relu ,i: 10

score: 0.9397222222222222
layer: (75, 50) ,learn: adaptive ,activation: tanh ,i: 11

score: 0.9414999999999999
layer: (75, 50) ,learn: adaptive ,activation: relu ,i: 12

score: 0.9405000000000001
```

So the best parameters were layer size = (100, 50), learning rate = invscaling, activation = tanh. So we train the data for these parameters except hidden layers = (100, 75, 50).

```
mlp = MLPClassifier(hidden_layer_sizes= (100, 75, 50), learning_rate=␣
 ↪'invscaling' , activation= 'tanh')
score = cross_val_score(mlp, encoded_data, y_ltrain, cv=5)
```

After that we fit the data and calculate the accuracy for train data which is 94.46.

```
encoded_test = [model.encoder(X_test[i]).detach().numpy() for i in␣
 ↪range(len(X_test))]
mlp.fit(encoded_data, y_ltrain)
y_predict = mlp.predict(encoded_test)
acc = accuracy_score(y_test, y_predict)
acc
```

```
0.9483
```

```
per_metrics = confusion_matrix(y_test, y_predict)
per_metrics
```

this is the confusion matrix which shows the 94.83 accuracy.

```
array([[ 967,    0,    1,    1,    2,    0,    6,    1,    1,    1],
       [   0, 1119,    1,    4,    1,    1,    4,    0,    4,    1],
       [   9,    5,  973,   14,    2,    2,    3,   10,   14,    0],
       [   0,    1,   14,  942,    0,   12,    0,   10,   21,   10],
       [   0,    2,    2,    0,  941,    1,    7,    2,    4,   23],
       [   3,    1,    1,   25,    3,  817,   11,    1,   18,   12],
       [   8,    4,    2,    0,    6,    7,  930,    0,    1,    0],
       [   1,    3,   15,    5,    4,    0,    1,  985,    2,   12],
       [   3,    1,   13,   22,    7,   25,    4,    5,  889,    5],
       [   5,    6,    1,    6,   42,    6,    0,   15,    8,  920]],
      dtype=int64)
```

Calculate the accuracy on train data.

```
encoded_test = [model.encoder(X_ltrain[i]).detach().numpy() for i in␣
 ↪range(len(X_ltrain))]
y_predict = mlp.predict(encoded_test)
acc = accuracy_score(y_ltrain, y_predict)
acc
```

```
1.0
```

it has a accuracy 100 and the confusion matrix is just perfect.

```
per_metrics = confusion_matrix(y_ltrain, y_predict)
per_metrics
```

```
array([[1777,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0, 2046,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0, 1804,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0, 1832,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0, 1715,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0, 1577,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0, 1769,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0, 1873,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0, 1791,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1816]],
      dtype=int64)
```

Here we load the unlabeled data again to use it for clustering. also we call the numpy data, X.

```
data = pd.read_csv("unlabeled_train_set.csv")
X = data.values
```

Now we cluster the data, using k-means. k-means was better for clustering than DBscan because k-means parameters were only number of clusters, on the other hand, DBscan's parameters were Eps and minPt which need an information about data in space which was not sufficient to use.

we choose the clusters number, 500, so that we could have more accuracy and then we will merge them. Also map 0-499 to 0-9.

```
# Create an instance of the KMeans class
kmeans = KMeans(n_clusters=500)
# Fit the model to your data
kmeans.fit(X)
# Obtain the cluster labels
labels = kmeans.labels_
```

Now after clustering, we don't know that which label that the algorithm has given to some data, shows what number. For solving this problem, we use the mlp model and AutoEncoder that we have fitted before to correct the labels because using them, we can determine the correct one.

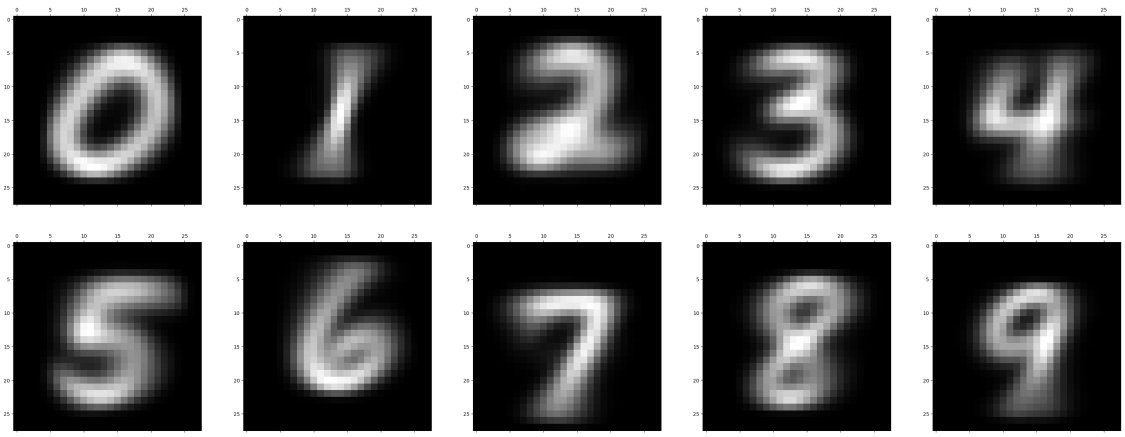
```
X_ultrain_test = scaler.transform(X)
nlabels = labels.copy()

for i in range(500):
    x = np.where(labels == i)
    rand = random.sample([j for j in range(len(x[0]))], 10)
    inp = np.array([X_ultrain_test[x[0][j]] for j in rand])
    tinp = torch.tensor(inp, dtype= torch.float32)
    encoded_out = [model.encoder(tinp[i]).detach().numpy() for i in range(10)]
    y_predict = mlp.predict(encoded_out)
    counts = np.bincount(y_predict)
    # Find the index of the maximum count
    most_common_value = np.argmax(counts)
    # print(most_common_value)
    for j in range(len(x[0])):
        nlabels[x[0][j]] = most_common_value + 500
nlabels -= 500
```

Now we map the clusters to 0-9.

```
cs = np.zeros((10, 784))
for i in range(10):
    x = np.where(nlabels == i)
    s = np.zeros(784)
    for j in range(len(x[0])):
        s += X[x[0][j]]
    s = s/len(x[0])
    cs[i] = s
```

then we will plot the centeroids of clusters which is representing the correct numbers as we can see.

```
fig, axs = plt.subplots(2, 5, figsize=(40, 15))
plt.gray()
for i in range(10):
    center = cs[i]
    axs[i//5, i%5].matshow(center.reshape(28, 28))
fig.show()
```



combine labeled train data with labeled cluster

```
new_X_train = torch.tensor(np.concatenate((X_ltrain.detach().numpy(), X_ultrain.
 ↪detach().numpy())), dtype= torch.float32)
new_y_train = np.concatenate((y_ltrain, nlabels))
```

Then we fit a model with same hyper parameters in task2 on new data.

```
encoded_train = [model.encoder(new_X_train[i]).detach().numpy() for i in␣
 ↪range(len(new_X_train))]

mlp_model = MLPClassifier(hidden_layer_sizes= (100, 75, 50), learning_rate=␣
 ↪'invscaling' , activation= 'tanh')
score = cross_val_score(mlp_model, encoded_train, new_y_train, cv=5)
print(np.mean(score))
```

```
0.9259000000000001
```

Now we test the model on combined data.

```
mlp_model.fit(encoded_train, new_y_train)

y_predict = mlp.predict(encoded_train)

acc = accuracy_score(new_y_train, y_predict)
acc
```

```
0.94395
```

Here is the confusion matrix on combined data. which shows the 94.39% accuracy.

```
per_metrics = confusion_matrix(new_y_train, y_predict)
per_metrics
```

```
array([[5821,    0,   16,    6,    2,   23,   27,    1,   35,    6],
       [   1, 6653,   16,   15,   28,   10,    9,   29,   46,   44],
       [   8,   43, 5753,   83,   36,   32,   23,  100,   99,   21],
       [  11,    2,   45, 5667,    3,  110,    3,    5,  117,   30],
       [   8,    5,   20,    5, 5347,   12,   10,   30,   24,  138],
       [  16,   11,    9,  155,   13, 5095,   40,   12,  186,   34],
       [  41,    7,   20,    4,   36,   44, 5813,   16,   30,    5],
       [   3,    7,   45,   21,   32,    8,    0, 5849,   24,  121],
       [   9,    6,   48,   90,   15,   66,   12,    3, 5248,   34],
       [   4,    8,   14,   15,  481,   16,    0,  211,   54, 5391]],
      dtype=int64)
```

Then we combine test data with labeled clusters.

```
new_X_test = torch.tensor(np.concatenate((X_test.detach().numpy(), X_ultrain.
 ↪detach().numpy())), dtype= torch.float32)
new_y_test = np.concatenate((y_test, nlabels))
```

Then we predict the encoded combine data on the previous model and file a great accuracy of 97.91%.

```
encoded_test = [model.encoder(new_X_test[i]).detach().numpy() for i in␣
 ↪range(len(new_X_test))]

y_predict = mlp_model.predict(encoded_test)

acc = accuracy_score(new_y_test, y_predict)
acc
```

```
0.9791923076923077
```

As we can see, after combining data with clusters, the accuracy of prediction got much better for test data. on the other hand, the accuracy of prediction for train data, got worse.

We can see the confusion matrix for test data on next page. Which shows that our model for combined data in more generalized and much better to use versus the model on only the data itself.

```
per_metrics = confusion_matrix(new_y_test, y_predict)
per_metrics
```

```
array([[5118,    0,    2,    2,    5,    4,    4,    2,    1,    2],
       [   0, 5915,    4,    2,    2,    2,    5,    3,    5,    2],
       [  15,   10, 5291,   38,    4,    3,    7,   32,   19,    7],
       [   1,    0,   13, 5078,    0,   33,    0,   10,   28,    8],
       [   3,    5,    5,    0, 4587,    4,   11,   11,    8,  232],
       [   6,    0,    5,   40,    1, 4782,   14,    3,   24,   11],
       [  14,    3,    5,    0,    6,    7, 5163,    2,    3,    2],
       [   0,    7,   23,    1,    4,    0,    0, 5152,    3,   75],
       [   9,    5,   17,   49,    5,   33,    4,    7, 4562,   23],
       [   4,    8,    2,    8,   32,    6,    1,   47,    9, 5270]],
      dtype=int64)
```