Compression of neural networks

Kian Anvari Hamedani

Consider the dataset MNIST. In order to classify these images, train a deep neural network so that the accuracy of the implemented architecture of the test set data is at least 85%. It is suggested to use multilayer neural network architecture.

Import Necessary Libraries

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import torch.nn.utils.prune as prune

from sklearn import metrics

import matplotlib.pyplot as plt

import copy
```

At first, we present the libraries needed to solve the question into the project:

- Pytorch library for forming neural network and importing different operation functions and neural network training

- Adding datasets to import the MNIST dataset into the project to use its images

- Sklearn library metrics section to calculate auc value after neural network compression

- Matplotlib library for displaying dataset images along with model accuracy changes and auc plots

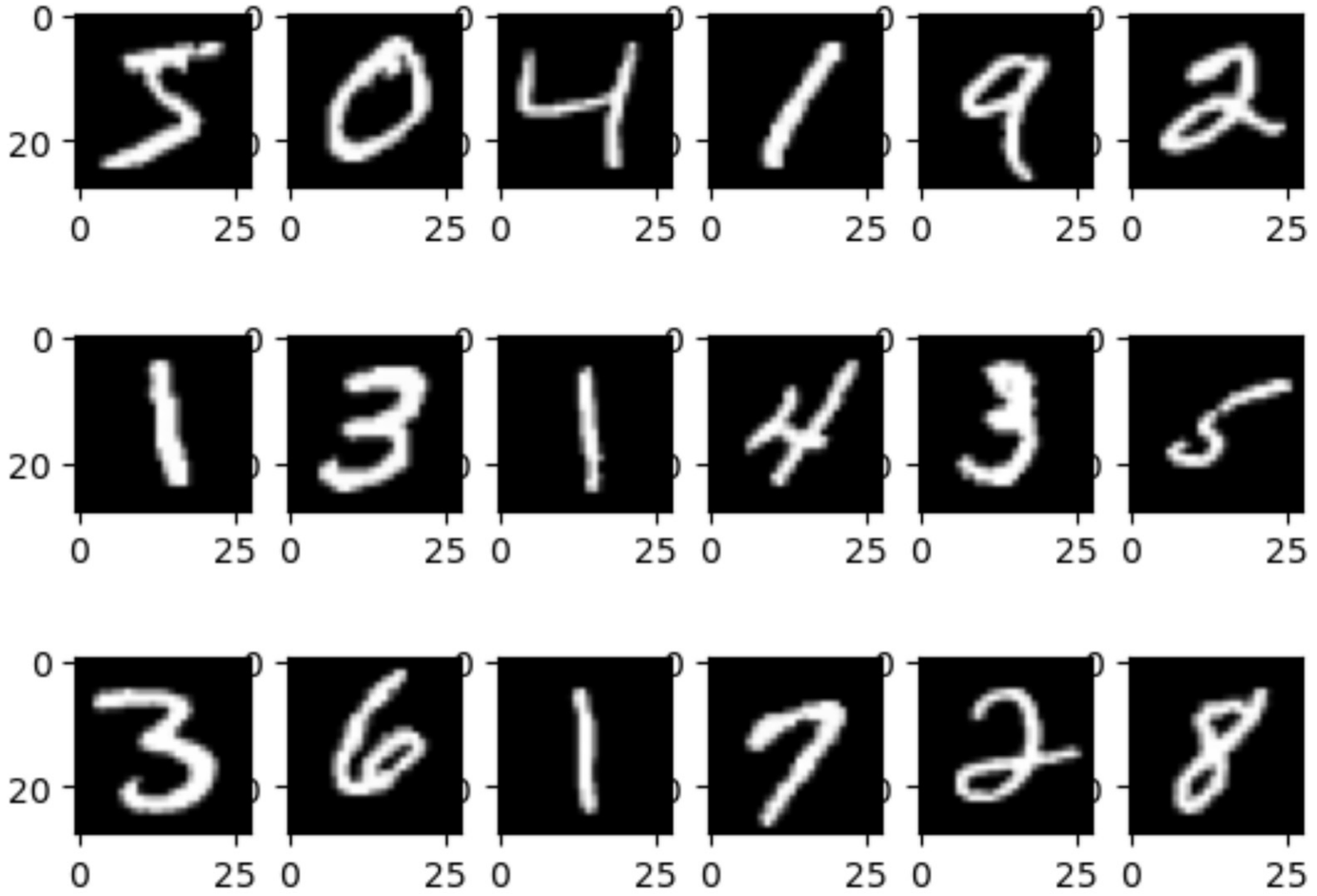
Show 18 Images of MNIST Dataset

```
mnist_images = datasets.MNIST('data', train=True, download=True)

for k, (image, label) in enumerate(mnist_images):
    if k >= 18:
        break
    plt.subplot(3, 6, k+1)
    plt.imshow(image, cmap="gray")
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
```

```
9913344/? [00:28<00:00, 644132.63it/s]
```

we download and enter the MNIST dataset into the project and then show its 18 images .



18 images from the MNIST dataset

Split dataset images into train/validation images

```
mnist_data = datasets.MNIST('data', train=True, transform=transforms.ToTensor())

mnist_data = list(mnist_data)
mnist_train = mnist_data[:4096]
mnist_val   = mnist_data[4096:5120]
```

We obtain the MNIST dataset and store it in the form of tensor in the mnist _ data variable .

We consider the first 4096 images as training images and other images as test images .

Class/Structure of fully-connected classifier model

```
class MNISTClassifier(nn.Module):

    def __init__(self):
        super(MNISTClassifier, self).__init__()

#         input is 28*28 and we have 256 neurons in layer1

        self.layer1 = nn.Linear(28 * 28, 256)

#         freeze 15% of neurons to avoid over-fitting

        self.drop = nn.Dropout(0.15)
        self.layer2 = nn.Linear(256, 10)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.sigmoid(self.layer1(flattened))
        drop1 = self.drop(activation1)
        output = self.layer2(drop1)
        return output
```

Then we vectorize the input image so that all the pixels of the image are placed together and entered into the model.
We set the action function of the first layer as sigmoid and the action function of the last layer as SoftMax.

In this part, we specify the main structure of the fully-connected neural network:

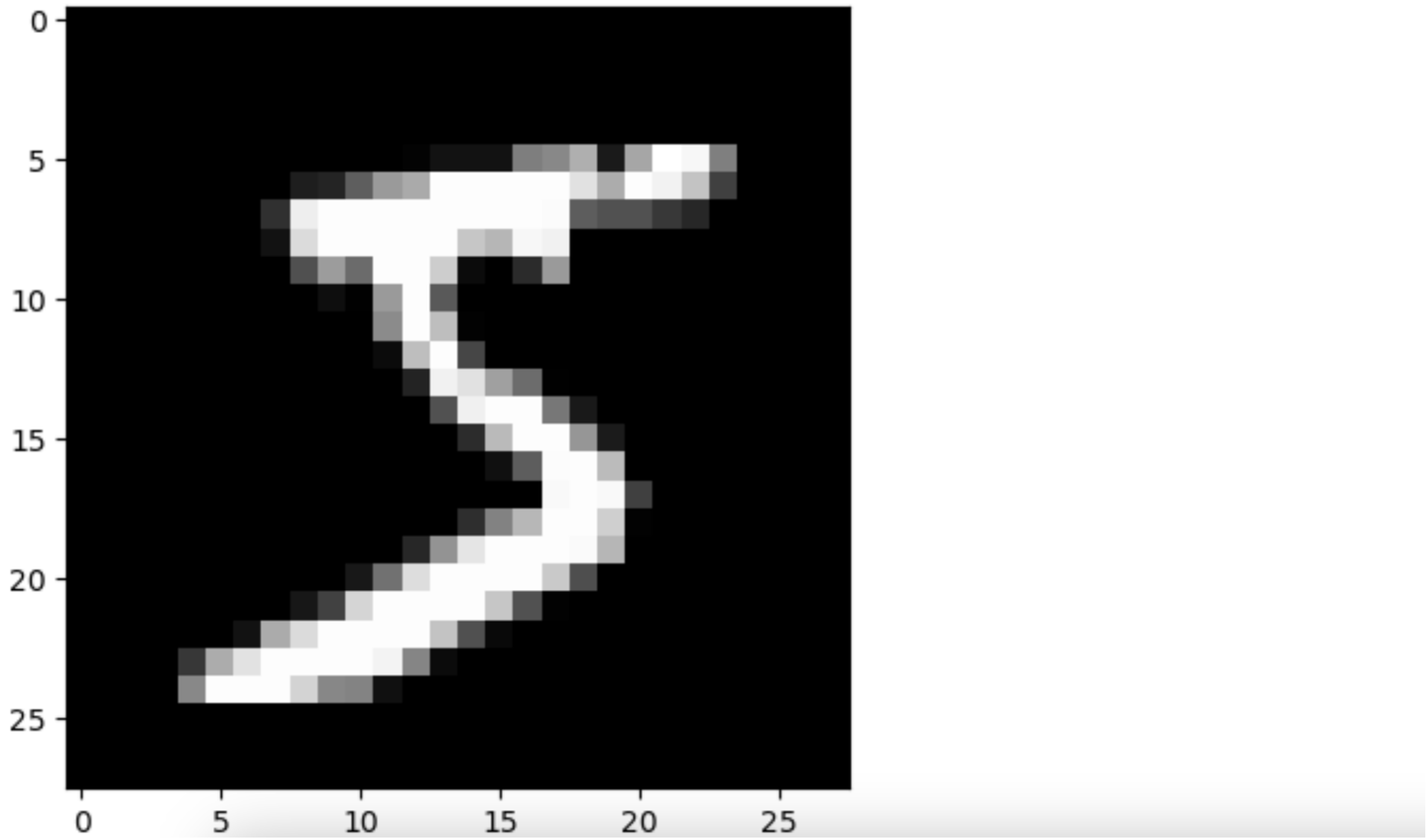
The size and number of pixels of input images to the model are 28 x 28 and we considered 256 neurons for the first layer.

Also, to prevent the weights of the first layer from being overfitting, we used dropout in the weights of the first layer to freeze and deactivate 15% of them.

Then, since there are 10 classes to be classified, in the last layer, we considered 10 neurons that estimate the probability of the input belonging to each class.

```python
model = MNISTClassifier()
prob = F.softmax(output, dim=1)
first_img , first_label = mnist_train[0]
output = model(first_img)
plt.imshow(first_img[0], cmap="gray")
print(output)
print(output.shape)
```

```
tensor([[-0.2993,  0.0415, -0.3139,  0.4649, -0.1052, -0.4542,  0.6010,  0.0272,
          0.6462,  0.1412]], grad_fn=<AddmmBackward>)
torch.Size([1, 10])
```



As a test, we display the first image of the training dataset along with its label.

Define train function

```
import torch.optim as optim

def train(model, data, batch_size=32, num_epochs=1):

    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size)
    criterion = nn.CrossEntropyLoss()

    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    iters, losses, train_acc, val_acc = [], [], [], []


    n = 0
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):
            out = model(imgs)
            loss = criterion(out, labels)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            iters.append(n)
            losses.append(float(loss)/batch_size)
            train_acc.append(get_accuracy(model, train=True))
            val_acc.append(get_accuracy(model, train=False))
            n += 1
```

We define the training function as follows:
As the input of the model and the dataset and the size of each packet and the number of iterations of training.
Error function: Cross entropy
We used gradient descent with a learning rate of one percent and a momentum of 0.9 to train the model.
We have created lists for repetition and error and accuracy in training data and accuracy in test data to display them in the graph in the future.
In this loop, which is repeated as many repetitions as the training, we optimize our model using the functions of the Torch library using the error propagation rule and then add the corresponding values in the corresponding lists.

```
# plotting
plt.title("Training Curve")
plt.plot(iters, losses, label="Train")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()

plt.title("Training Curve")
plt.plot(iters, train_acc, label="Train")
plt.plot(iters, val_acc, label="Validation")
plt.xlabel("Iterations")
plt.ylabel("Training Accuracy")
plt.legend(loc='best')
plt.show()

print("Final Training Accuracy: {}".format(train_acc[-1]))
print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

We draw the model accuracy in training and test data.

Define get accuracy function

```
train_acc_loader = torch.utils.data.DataLoader(mnist_train, batch_size=4096)
val_acc_loader = torch.utils.data.DataLoader(mnist_val, batch_size=1024)

def get_accuracy(model, train=False):
    if train:
        data = mnist_train
    else:
        data = mnist_val

    correct = 0
    total = 0

    for imgs, labels in torch.utils.data.DataLoader(data, batch_size=64):
        output = model(imgs)
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += imgs.shape[0]

    return correct / total
```

We use this function to obtain the accuracy of the model in the training and test datasets. This function takes the model and the dataset type as input.

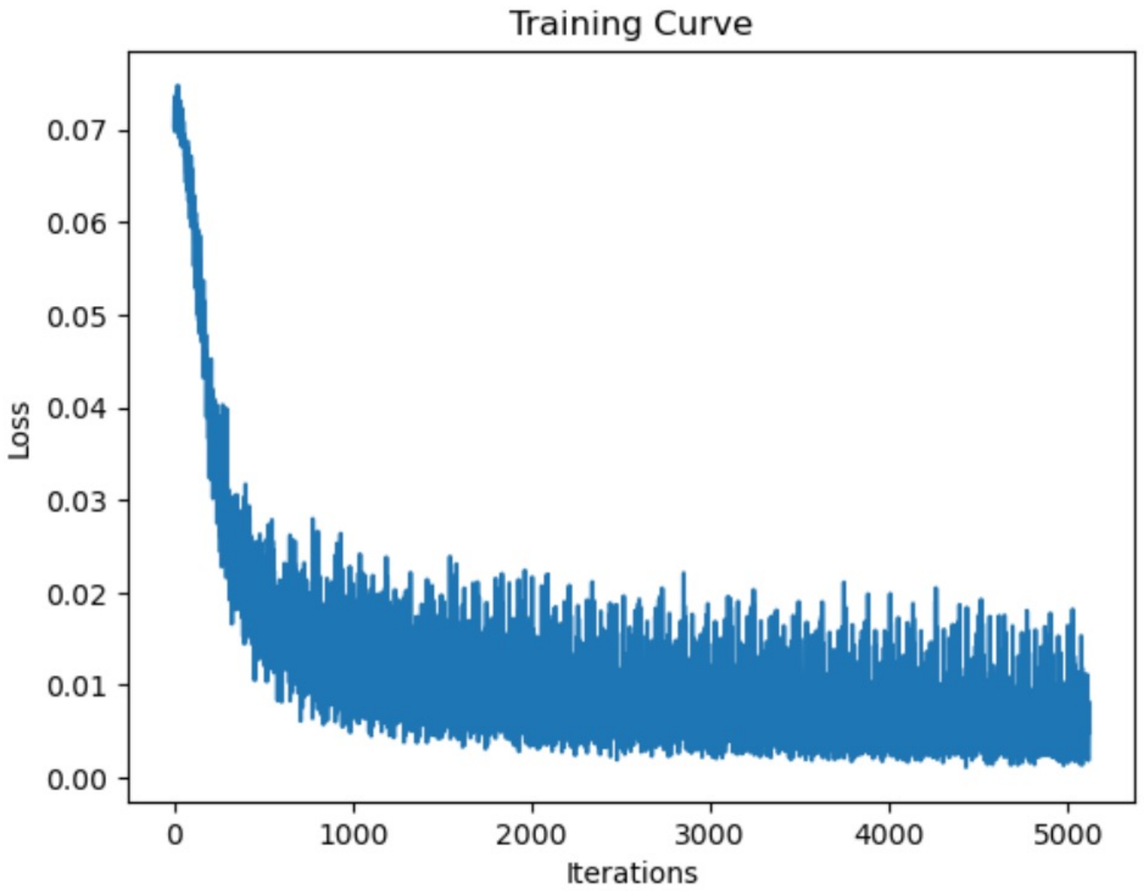
According to the type of dataset entered in the input, the function compares the accuracy value of the output obtained by the model with their real value or label and returns the accuracy value in the output.

Train Model

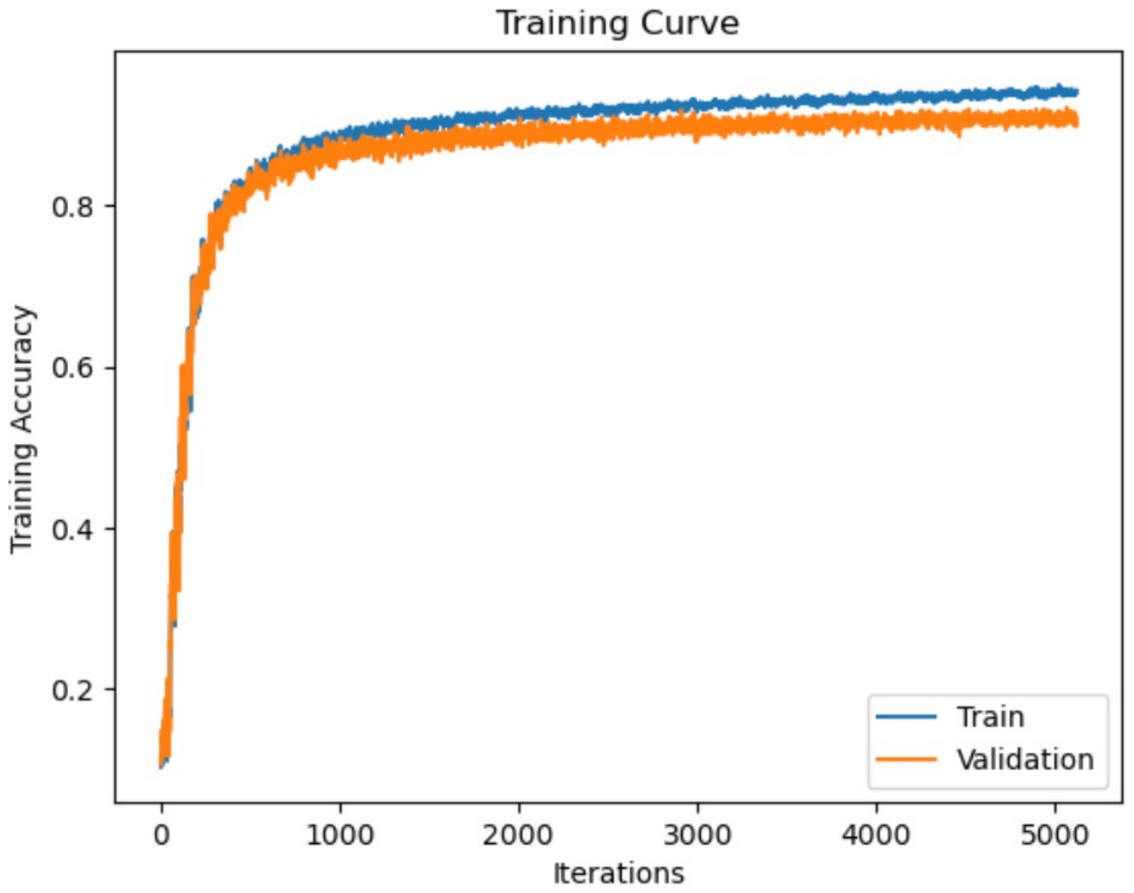
```
model = MNISTClassifier()
```

```
train(model, mnist_train, num_epochs=40)

torch.save(model.state_dict(), "saved_model1")
```

We create a sample neural network and train it with the training dataset and 32 packets and 40 iterations and store the obtained model for future use.



Model error in the training data graph



Model accuracy in training data graph

```
Final Training Accuracy: 0.943359375
Final Validation Accuracy: 0.904296875
```

The accuracy of the trained model in training and test data

Implement the selected compression method in step 1 on the neural network of the second part. Next, for different values of the compression rate (from zero to 90 percent), draw the accuracy graph according to the compression rate and report the area under the curve (AUC).

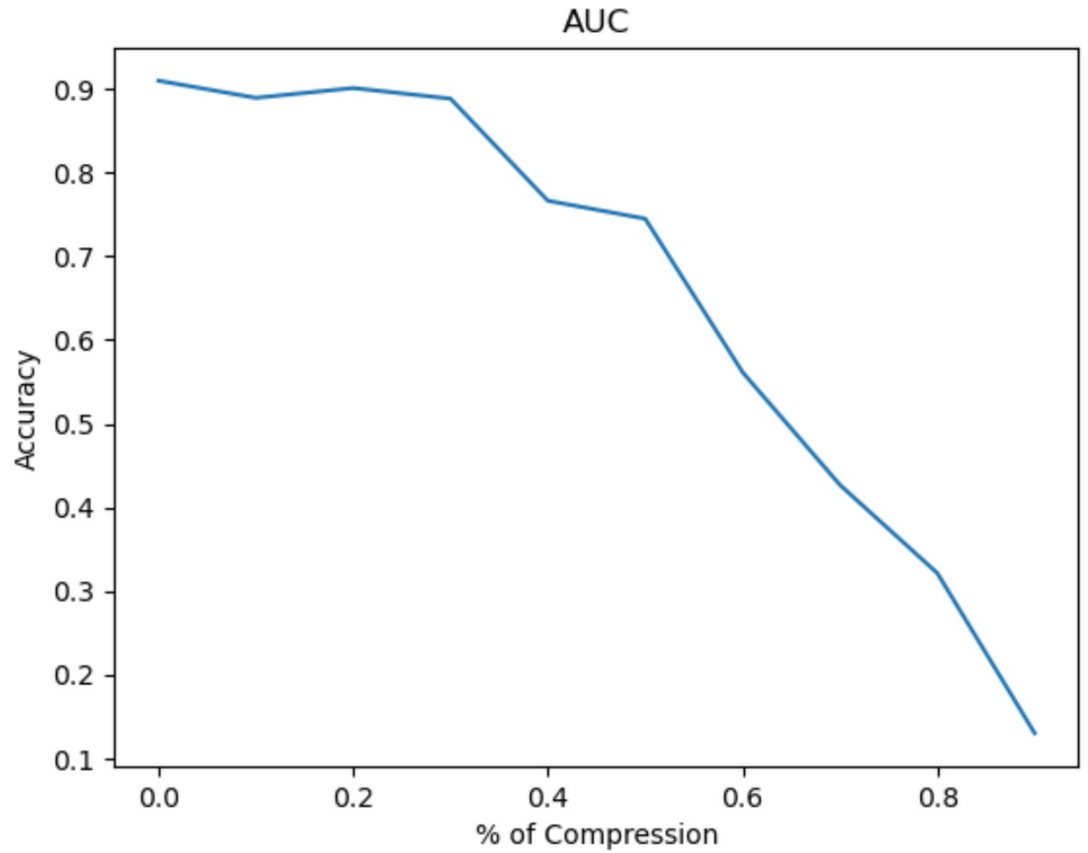
Random Weight Pruning For Compress Neural Networkk

```
accuracy = []

for i in range(10):

  model = MNISTClassifier()
  model.load_state_dict(torch.load('saved_model12'))

  prune.random_unstructured(model.layer1,name ='weight', amount=(i/10))
  prune.random_unstructured(model.layer2,name ='weight', amount=(i/10))

  get_accuracy(model, train=False)
  accuracy.append(get_accuracy(model, train=False))


x = [0,.1,.2,.3,.4,.5,.6,.7,.8,.9]
```

In the third part, to compress the neural network, we use the idea of pruning weights:
We use a loop to compress the neural network at different compression rates.
In each compression rate, we randomly prune the value corresponding to the same rate from the weights of each layer and train the model and save the accuracy of the model to finally print the AUC value related to this type of compression.

```
plt.subplot(1,1,1)
plt.title("AUC")
plt.xlabel("% of Compression")
plt.ylabel("Accuracy")
plt.plot(x, accuracy)

print(f"AUC = {metrics.auc(x, accuracy)}")
```

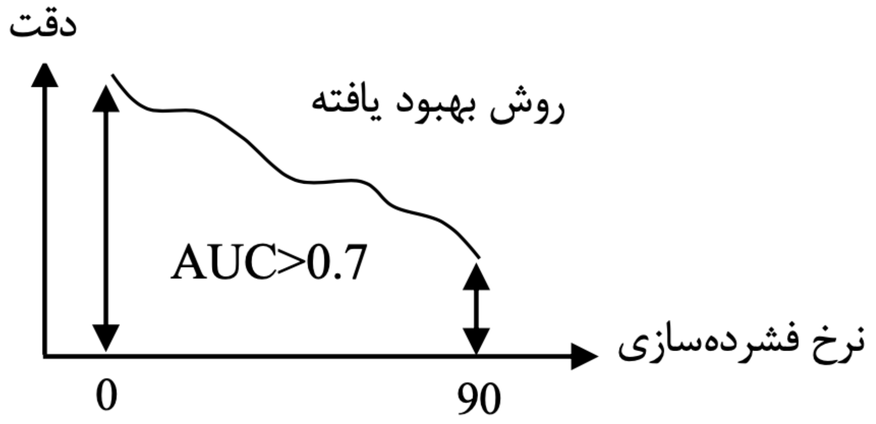
```
AUC = 0.6020996093750001
```



As shown in the following diagram, we observe the accuracy obtained at each compression rate :

The value of the AUC in this method is 0.6.

4. Make a change to the chosen method so that after a 90% compression rate, the AUC of the new method is at least 7.0.



Implementation of customized-pruning method

```
def fnd(arg, row):
    rr = int(arg/row)
    col = arg - (rr * row)
    return rr, col
```

In part 4, the idea of weight pruning is implemented with a new and useful change.

In the following, we have used the torch.argmin function to obtain the minimum weight value. The output of this function gives us the index in the linear space. To get the exact minimum weight index in the weights matrix, we use the find function to get the row and column of the minimum weight accurately in weights matrix.

```
accuracy1 = []

for i in range(10):

  model = MNISTClassifier()
  model.load_state_dict(torch.load('saved_model12'))

#   Map all weights into + area

  layer1 = torch.abs(copy.deepcopy(model.layer1.weight))
  layer2 = (copy.deepcopy(model.layer2.weight))

#   28 * 28 *256

  a = int((i/10)*200704)

#  256 * 10

  b = int((i/10)*2560)
```

The idea of weight pruning in this method:
At first, we take the absolute value of all the weights of the first layer and map all the weights to the positive space.
In the following, relative to the compression rate in question, which increases from ۰ to ۹۰٪ in a loop, we remove minimum weights mapped to the positive space or set it to ۰.
Then we do the same thing on the weights of the second layer, but this time without mapping to the space of positives.

```
  for _ in range(a):

#    find argmin(Minimum weight) in layer 1 in layer1
    index = torch.argmin(layer1)

#    rows : number of neurons
    rr = model.layer1.weight.shape[1]

    row , col = fnd(index ,rr)

    with torch.no_grad():
      layer1[row][col] = 5000
      model.layer1.weight[row][col] = 0



  for j_ in range(b):
    index=torch.argmin(layer2)
    rr = model.layer2.weight.shape[1]
    row , col = fnd(index ,rr )

    with torch.no_grad():
      layer2[row][col]= 5000
      model.layer2.weight[row][col] = 0


  print(f"accuracy with {int((x[i])*100)}% compression is {get_accuracy(model, train=False)}")

  accuracy1.append(get_accuracy(model, train=False))


x = [0,.1,.2,.3,.4,.5,.6,.7,.8,.9]
```

First, we make a copy of the weights of each layer and implement the explained idea on those weights.

According to the explanations mentioned in the previous slide, we identify the location of the element or the minimum weight using the find function and set it to 0 in the weights matrix.

After setting the corresponding weight to 0 in the model weights, we put its value in the copied weights of 5000 so that it will not be identified as the minimum weight again in the next iterations.

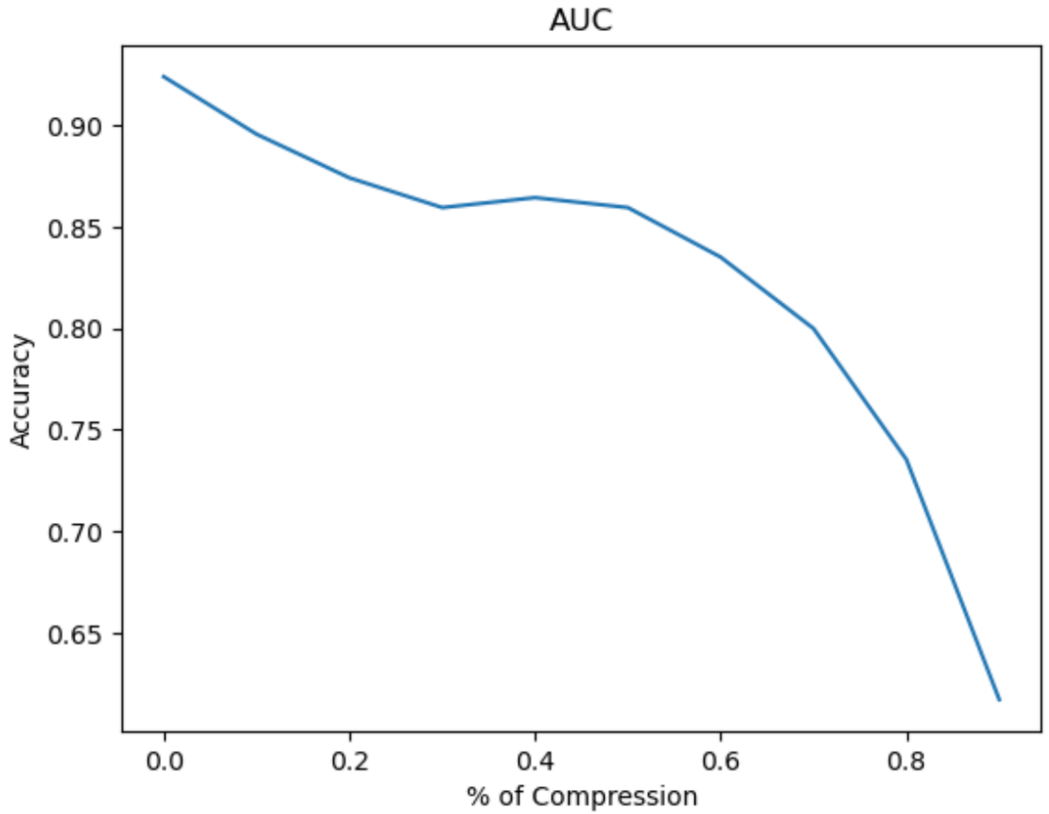
After performing the appropriate pruning with the compression rate, we retrain the model with the pruned weights in each stage and calculate and save the obtained accuracies in different compression rates so that we can draw its AUC diagram at the end. So that we can check the success rate of this model.

```
accuracy with 0% compression is 0.919921875
accuracy with 10% compression is 0.8935546875
accuracy with 20% compression is 0.8681640625
accuracy with 30% compression is 0.859375
accuracy with 40% compression is 0.8720703125
accuracy with 50% compression is 0.8701171875
accuracy with 60% compression is 0.84765625
accuracy with 70% compression is 0.7939453125
accuracy with 80% compression is 0.7412109375
accuracy with 90% compression is 0.62890625
```

Accuracy of the model in different compression rates after applying and implementing the new pruning idea

```
plt.subplot(1,1,1)
plt.title("AUC")
plt.xlabel("% of Compression")
plt.ylabel("Accuracy")
plt.plot(x, accuracy1)

print(f"AUC = {metrics.auc(x, accuracy1)}")
```

```
AUC = 0.74931640625
```



AUC diagram in different compression rates after applying and implementing the new pruning idea:

As you can see, the AUC value obtained in this method is 0.74, which is more than the acceptable value of 0.7.