Task 2: Stacked Autoencoder based Pre-Training of DNN and Fine Tuning

In [ ]:

```
import torch
import random
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
```

In [ ]:

```
animals = ['cavallo', 'pecora', 'gatto', 'farfalla', 'cane']
datasetDir = "Dataset/"
```

Generating the Dataset

The dataset is generated by combining all csvs, and randomly shuffling the data and then doing an 80-20 train-test split

In [ ]:

```
def generateDatasetCSV():
    newlines = []
    for i,animal in enumerate(animals):

        with open(f"{datasetDir+animal}.csv","r") as f:
            lines = f.readlines()
            for line in lines:
                newlines.append(','.join(line.strip().split(",")[1:49])+f",{i}")

    random.shuffle(newlines)
    train_lines = newlines[0:int(0.8*len(newlines))]
    test_lines = newlines[int(0.8*len(newlines)):len(newlines)]

    with open(f"{datasetDir}train.csv","w") as f:
        for line in train_lines:
            f.write(line)
            f.write("\n")
    with open(f"{datasetDir}test.csv","w") as f:
        for line in test_lines:
            f.write(line)
            f.write("\n")
```

In [ ]:

```
generateDatasetCSV()
```

In [ ]:

```
class AnimalDataset(Dataset):
    def __init__(self,data_source):
        with open(data_source,"r") as f:
            self.lines = f.readlines()
        self.inputs = []
        self.outputs = []
        for line in self.lines[0:]:
            self.linedata = tuple(map(float,line.strip().split(",")))
            self.inputs.append(torch.tensor(self.linedata[0:48])/189000.0)
            self.outputs.append(torch.tensor(self.linedata[48]))
    def __len__(self):
        return len(self.outputs)
    def __getitem__(self,idx):
        return self.inputs[idx],self.outputs[idx]
```

In [ ]:

```
class AANN1(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(in_features=48,out_features=64),nn.Tanh(),nn
        self.decoder = nn.Sequential(nn.Linear(in_features=24,out_features=64),nn.Tanh(),nn
    def forward(self,x,path="all"):
        if(path == "all"):
            return self.decoder(self.encoder(x))
        else:
            return self.encoder(x)
```

In [ ]:

```
class AANN2(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(in_features=24,out_features=32),nn.Tanh(),nn
        self.decoder = nn.Sequential(nn.Linear(in_features=12,out_features=32),nn.Tanh(),nn
    def forward(self,x,path="all"):
        if(path == "all"):
            return self.decoder(self.encoder(x))
        else:
            return self.encoder(x)
```

In [ ]:

```
class AANN3(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(in_features=12,out_features=18),nn.Tanh(),nn
        self.decoder = nn.Sequential(nn.Linear(in_features=6,out_features=18),nn.Tanh(),nn.
    def forward(self,x,path="all"):
        if(path == "all"):
            return self.decoder(self.encoder(x))
        else:
            return self.encoder(x)
```

In [ ]:

```
trainDataset = AnimalDataset(data_source=f"{datasetDir}train.csv")
```

In [ ]:

```
batch_size = 25
trainLoader  = torch.utils.data.DataLoader(trainDataset, batch_size=batch_size,shuffle=True
```

In [ ]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
aann1 = AANN1().to(device)
aann2 = AANN2().to(device)
aann3 = AANN3().to(device)
```

Training AANN1

In [ ]:

```
# Training AANN1

aann1_optimizer = optim.Adam(aann1.parameters(),lr=4e-4)
criterion = nn.KLDivLoss(reduction="batchmean")
epochs = 1000

for epoch in range(epochs):
    total_loss = 0
    cnt = 0
    for i,(data,target) in enumerate(trainLoader):
        data = data.to(device)

        out = aann1(data)

        loss = criterion(nn.functional.log_softmax(out),data)
        cnt+=1
        total_loss += loss.item()
        aann1_optimizer.zero_grad()
        loss.backward()

        aann1_optimizer.step()

    print(f"Epoch {epoch} average loss: {total_loss/cnt}")
```

In [ ]:

```
torch.save(aann1.state_dict(), "aann1.pth")
```

In [ ]:

```
trainDataset[0][0]*189000.0
```

In [ ]:

```
torch.nn.functional.softmax(aann1(trainDataset[0][0].to(device)))*189000.0
```

Training AANN2

In [ ]:

```
# Training AANN2

aann2_optimizer = optim.Adam(aann2.parameters(),lr=4e-4)
criterion = nn.MSELoss()
epochs = 1000

for epoch in range(epochs):
    total_loss = 0
    cnt = 0
    for i,(data,target) in enumerate(trainLoader):
        data = aann1(data.to(device),path="encoder")

        out = aann2(data)

        loss = criterion(out,data)
        cnt+=1
        total_loss += loss.item()
        aann2_optimizer.zero_grad()
        loss.backward()

        aann2_optimizer.step()

    print(f"Epoch {epoch} average loss: {total_loss/cnt}")
```

In [ ]:

```
torch.save(aann2.state_dict(), "aann2.pth")
```

In [ ]:

```
aann1(trainDataset[0][0].to(device),path="encoder")
```

In [ ]:

```
aann2(aann1(trainDataset[0][0].to(device),path="encoder"))
```

Training AANN3

In [ ]:

```
# Training AANN3

aann3_optimizer = optim.Adam(aann3.parameters(),lr=4e-4)
criterion = nn.MSELoss()
epochs = 1000

for epoch in range(epochs):
    total_loss = 0
    cnt = 0
    for i,(data,target) in enumerate(trainLoader):
        data = aann2(aann1(data.to(device),path="encoder"),path="encoder")

        out = aann3(data)

        loss = criterion(out,data)
        cnt+=1
        total_loss += loss.item()
        aann3_optimizer.zero_grad()
        loss.backward()

        aann3_optimizer.step()

    print(f"Epoch {epoch} average loss: {total_loss/cnt}")
```

In [ ]:

```
torch.save(aann3.state_dict(), "aann3.pth")
```

Fine Tuning Stacked Autoencoder by adding an Output Softmax Layer

In [ ]:

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

        self.aann1 = AANN1()
        self.aann1.load_state_dict(torch.load("aann1.pth"))

        self.aann2 = AANN2()
        self.aann2.load_state_dict(torch.load("aann2.pth"))

        self.aann3 = AANN3()
        self.aann3.load_state_dict(torch.load("aann3.pth"))

        self.output = nn.Sequential(nn.Linear(in_features=6,out_features=5))
    def forward(self,x):

        return self.output(self.aann3(self.aann2(self.aann1(x,path="encoder"),path="encoder
```

In [ ]:

```
# Fine Tuning Stacked Autoencoder
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
sae = StackedAE().to(device)

sae_optimizer = optim.Adam(sae.parameters(),lr=6e-3)
sae_scheduler = optim.lr_scheduler.ReduceLROnPlateau(sae_optimizer,"min")
criterion = nn.CrossEntropyLoss()
epochs = 1000

for epoch in range(epochs):
    total_loss = 0
    cnt = 0
    for i,(data,target) in enumerate(trainLoader):
        input_ = sae(data.to(device))



        loss = criterion(input_,target.long().to(device))
        cnt+=1
        total_loss += loss.item()
        sae_optimizer.zero_grad()
        loss.backward()

        sae_optimizer.step()
    sae_scheduler.step(total_loss/cnt)
    print(f"Epoch {epoch} average loss: {total_loss/cnt}")
```

In [ ]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

In [ ]:

```
torch.save(sae.state_dict(), "sae.pth")
```

In [ ]:

```
sae = StackedAE()
sae.load_state_dict(torch.load("sae.pth"))
```

In [ ]:

```
testDataset = AnimalDataset(data_source=f"{datasetDir}test.csv")
```

In [ ]:

```
import numpy as np
device = 'cpu'
y_actual_train = []
y_pred_train = []

y_actual_test = []
y_pred_test = []


cnt = 0
tot = 0
for j in range(len(trainDataset)):
        y_pred_train.append(torch.argmax(nn.functional.softmax(sae(trainDataset[j][0].to(de
        y_actual_train.append(trainDataset[j][1].item())
for j in range(len(testDataset)):
        y_pred_test.append(torch.argmax(nn.functional.softmax(sae(testDataset[j][0].to(devi
        y_actual_test.append(testDataset[j][1].item())

y_actual_train = np.array(y_actual_train)
y_pred_train = np.array(y_pred_train)

y_actual_test = np.array(y_actual_test)
y_pred_test = np.array(y_pred_test)

for j in range(len(y_actual_train)):
    tot+=1
    if(y_actual_train[j] == y_pred_train[j]):
        cnt+=1
print("Train Set Accuracy: ", cnt/tot)

tot = 0
cnt = 0

for j in range(len(y_actual_test)):
    tot+=1
    if(y_actual_test[j] == y_pred_test[j]):
        cnt+=1

print("Test Set Accuracy: ", cnt/tot)
```

In [ ]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

cm = confusion_matrix(y_actual_test, y_pred_test, labels=[0,1,2,3,4])
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=animals)

disp.plot()
plt.title("Test Set Confusion Matrix")
plt.savefig("test_confusion.png")
plt.show()
```

In [ ]:

```
cm = confusion_matrix(y_actual_train, y_pred_train, labels=[0,1,2,3,4])
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=animals)

disp.plot()
plt.title("Train Set Confusion Matrix")
plt.savefig("train_confusion.png")
plt.show()
```

In [ ]:

```
freqs = [0 for i in range(5)]
for j in range(len(y_actual_train)):
    freqs[int(y_actual_train[j].item())]+=1
```

In [ ]:

```
plt.xlabel("Class")
plt.ylabel("Frequency")
plt.hist(y_actual_train)
plt.xticks([0,1,2,3,4])
plt.title("Data Distribution in the Training Set")
plt.savefig("data.jpg")
```

In [ ]:

```

```