In [ ]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data
import time
import os

# Load the dataset
data = np.load('lab2_dataset.npz')
train_feats = torch.tensor(data['train_feats'])
test_feats = torch.tensor(data['test_feats'])
train_labels = torch.tensor(data['train_labels'])
test_labels = torch.tensor(data['test_labels'])
phone_labels = data['phone_labels']

print(train_feats.shape)
print(test_feats.shape)
print(train_labels.shape)
print(test_labels.shape)
print(phone_labels.shape)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
torch.Size([44730, 11, 40])
torch.Size([4773, 11, 40])
torch.Size([44730])
torch.Size([4773])
(48,)
cpu
```

In [ ]:
```
# Set up the dataloaders
train_dataset = torch.utils.data.TensorDataset(train_feats, train_labels)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=8, shuffle

test_dataset = torch.utils.data.TensorDataset(test_feats, test_labels)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=8, shuffle=
```

In [ ]:
```
print(phone_labels)
```

```
['sil' 's' 'ao' 'l' 'r' 'iy' 'vcl' 'd' 'eh' 'cl' 'p' 'ix' 'z' 'ih' 'sh'
 'n' 'v' 'aa' 'y' 'uw' 'w' 'ey' 'dx' 'b' 'ay' 'ng' 'k' 'epi' 'ch' 'dh'
 'er' 'en' 'g' 'aw' 'hh' 'ae' 'ow' 't' 'ax' 'm' 'zh' 'ah' 'el' 'f' 'jh'
 'uh' 'oy' 'th']
```

In [ ]:
```
# Define the model architecture
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()


        self.fc1 = nn.Linear(11 * 40, 1024)
        self.fc2 = nn.Linear(1024, 1024)
        self.fc3 = nn.Linear(1024, 1024)
        self.fc4 = nn.Linear(1024, 512)
        self.fc5 = nn.Linear(512, 256)
```

```
        self.fc6 = nn.Linear(256, 128)
        self.fc7 = nn.Linear(128, 64)
        self.fc8 = nn.Linear(64, 48)

        self.dropout = nn.Dropout(0.25)

        self.relu = nn.ReLU() # activation function

    def forward(self, x):
        x = torch.reshape(x, (-1, 11 * 40))
        #print(x.shape)

        x = self.fc1(x)
        x = self.relu(x)

        x = self.fc2(x)
        x = self.relu(x)

        x = self.fc3(x)
        x = self.relu(x)

        x = self.fc4(x)
        x = self.relu(x)

        x = self.dropout(x)

        x = self.fc5(x)
        x = self.relu(x)

        x = self.fc6(x)
        x = self.relu(x)

        x = self.fc7(x)
        x = self.relu(x)

        x = self.fc8(x)

        return x
```

In [ ]:

```
# Instantiate the model, loss function, and optimizer
model = MyModel()
modeltrained = False
try :
    model.load_state_dict(torch.load('model.pt'))
    model.eval()
    modeltrained = True
    print('Model loaded')
except:
    print('No model found')
    pass
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

def train_network(model, train_loader, criterion, optimizer):
    # TODO: fill in
    for epoch in range(10):
        # running_loss = 0.0
        time1 = time.time()
        for i, (inputs, labels) in enumerate(train_loader, 0):
```

```
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # running_loss += loss.item()
            # if i % 1000 == 999:
            #     print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_los
            #     running_loss = 0.0
            #
        time2 = time.time()
        print('Epoch %d' % (epoch + 1))
        print(time2 - time1)
        test_network(model, test_loader)


label_acc = {}

missclassifications = {}

def test_network(model, test_loader):
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            outputs = model(inputs)
            # outputs
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            for i in range(len(labels)):
                if labels[i].item() not in label_acc:
                    label_acc[labels[i].item()] = [0, 0]
                label_acc[labels[i].item()][1] += 1
                if predicted[i] == labels[i]:
                    label_acc[labels[i].item()][0] += 1
                else:
                    if labels[i].item() not in missclassifications:
                        missclassifications[labels[i].item()] = {}
                    if predicted[i].item() not in missclassifications[labels[i]
                        missclassifications[labels[i].item()][predicted[i].iten
                    missclassifications[labels[i].item()][predicted[i].item()]




    print('Test accuracy: %d %%' % (100 * correct / total))

if not modeltrained:
    print('Training model')
    time1 = time.time()
    train_network(model, train_loader, criterion, optimizer)
    time2 = time.time()
    print(time2 - time1)
    print('Finished Training')
    torch.save(model.state_dict(), 'model.pt')
```

```
No model found
Training model
Epoch 1
62.41470217704773
Test accuracy: 30 %
Epoch 2
61.731407165527344
Test accuracy: 43 %
Epoch 3
59.271251916885376
Test accuracy: 43 %
Epoch 4
63.23592805862427
Test accuracy: 48 %
Epoch 5
62.11361289024353
Test accuracy: 51 %
Epoch 6
61.719362020492554
Test accuracy: 52 %
Epoch 7
56.14547896385193
Test accuracy: 54 %
Epoch 8
54.65034103393555
Test accuracy: 55 %
Epoch 9
54.36333894729614
Test accuracy: 57 %
Epoch 10
53.94810104370117
Test accuracy: 58 %
603.7727010250092
Finished Training
```

In [ ]:

```
print('Finished Training')
```

```
Finished Training
```

In [ ]:

```
test_network(model, test_loader)
```

```
Test accuracy: 57 %
```

In [ ]:

```
# sort by accuracy
sorted_acc = sorted(label_acc.items(), key=lambda x: x[1][0] / x[1][1], reverse

print("Top 5 most accurate phones:")
for i in range(5):
    print(phone_labels[sorted_acc[i][0]], sorted_acc[i][1][0] / sorted_acc[i][1

print("Top 5 least accurate phones:")

for i in range(1, 6):
    print(phone_labels[sorted_acc[-i][0]], sorted_acc[-i][1][0] / sorted_acc[-i
```

```
Top 5 most accurate phones:
dx 0.8045454545454546
sh 0.8009090909090909
epi 0.7636363636363637
sil 0.759090909090909
s 0.7363636363636363
Top 5 least accurate phones:
uh 0.12636363636363637
zh 0.29140722291407223
th 0.32636363636363636
ax 0.3463636363636364
en 0.3663636363636364
```

In [ ]:

```
common_missclassifications = []

common_missclassifications.append(np.where(phone_labels == 'sh')[0][0])
common_missclassifications.append(np.where(phone_labels == 'p')[0][0])
common_missclassifications.append(np.where(phone_labels == 'm')[0][0])
common_missclassifications.append(np.where(phone_labels == 'r')[0][0])
common_missclassifications.append(np.where(phone_labels == 'ae')[0][0])

for i in common_missclassifications:
    sorted_miss = sorted(missclassifications[i].items(), key=lambda x: x[1], re
    print(phone_labels[i], "is commonly misclassified as:", phone_labels[sorted
```

```
sh is commonly misclassified as: s with 89 instances
p is commonly misclassified as: k with 165 instances
m is commonly misclassified as: n with 232 instances
r is commonly misclassified as: er with 288 instances
ae is commonly misclassified as: eh with 249 instances
```