In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

In [2]:
```
# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
if torch.cuda.is_available():
    print("Using the GPU!")
else:
    print("WARNING: Could not find GPU! Using CPU only")
```

```
Using the GPU!
```

In [3]:
```
x_train_nhts = np.load("data/x_train_nhts.npy")
x_test_nhts = np.load("data/x_test_nhts.npy")

x_train_images = np.load("data/x_train_images.npy")
x_test_images = np.load("data/x_test_images.npy")


y_train = np.load("data/y_train.npy")
y_test = np.load("data/y_test.npy")
print("The sample size of training set is: ", x_train_nhts.shape[0])
print("The sample size of testing set is: ", x_test_nhts.shape[0])
```

```
The sample size of training set is:  3556
The sample size of testing set is:  889
```

In [4]:

```
# bridge numpy to torch
x_train_nhts_torch = torch.as_tensor(x_train_nhts).float() # specify floa
x_train_images_torch = torch.as_tensor(x_train_images).float()
x_test_nhts_torch = torch.as_tensor(x_test_nhts).float()
x_test_images_torch = torch.as_tensor(x_test_images).float()
y_train_torch = torch.as_tensor(y_train[:,0])
y_test_torch = torch.as_tensor(y_test[:,0])
n_train = x_train_nhts.shape[0]
n_test = x_test_nhts.shape[0]
# inputs: x_train_nhts, x_train_images, x_test_nhts, x_test_images, y_tra
K = len(np.unique(y_train))
x_dim = x_train_nhts.shape[1]
#
pd.value_counts(y_train[:,0])/y_train.shape[0]
```

Out[4]:

```
2    0.336333
1    0.325928
3    0.251969
0    0.085771
dtype: float64
```

In [5]:
```
##### Type 1: with only NHTS dataset.
class NN(nn.Module):  # subclass nn.Module
    def __init__(self):
        super(NN, self).__init__()
        self.fc1 = nn.Linear(x_dim, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 128)
        self.fc4 = nn.Linear(128, 128)
        self.fc5 = nn.Linear(128, 256)
        self.fc6 = nn.Linear(256, 256)
        self.fc7 = nn.Linear(256, 256)
        self.fc8 = nn.Linear(256, 512)
        self.fc9 = nn.Linear(512, K)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        x = self.fc1(x)
        x = x.relu()
        x = self.fc2(x)
        x = x.relu()
        x = self.fc3(x)
        x = x.relu()
        x = self.fc4(x)
        x = x.relu()
        x = self.fc5(x)
        x = x.relu()
        x = self.fc6(x)
        x = x.relu()
        x = self.fc7(x)
        x = x.relu()
        x = self.fc8(x)
        x = x.relu()
        x = self.fc9(x)
        x = x.relu()
        x = self.softmax(x)
        return x
```

In [6]:
```
net = NN().float().to(device)
print(type(net))
optim = torch.optim.Adam(net.parameters(), lr=0.0001)
criterion = nn.CrossEntropyLoss()
#
n_epoches = 500 # so many?
batch_size = 200
```

```
<class '__main__.NN'>
```

In [7]:
```
# training
train_losses = []
test_losses = []
```

```
train_accuracies = []
test_accuracies = []
for n_epoch in range(n_epoches):
    # create permutation for batch training
    permutation = torch.randperm(x_train_nhts_torch.size()[0])
    for i in range(0, x_train_nhts_torch.size()[0], batch_size):
        # clear gradients first (for each iteration!)!
        optim.zero_grad()
        # forward pass
        indices = permutation[i:i+batch_size]
        batch_x, batch_y = x_train_nhts_torch[indices].to(device), y_trai
        batch_y_pred_train = net(batch_x).to(device)
        # loss
        loss = criterion(batch_y_pred_train.squeeze(), batch_y)
        # compute gradients
        loss.backward()
        # one step optim
        optim.step()

    # eval training accuracy
    with torch.no_grad():
        y_pred_train = net(x_train_nhts_torch.to(device))
        loss_train = criterion(y_pred_train.squeeze(), y_train_torch.to(d
        train_losses.append(loss_train)
        _, predict_train = torch.max(y_pred_train, axis = 1)
        accuracy_train = (predict_train == y_train_torch.to(device)).sum(
        train_accuracies.append(accuracy_train)
        # evaluate testing sets step-wise
        net.eval()
        y_pred_test = net(x_test_nhts_torch.to(device))
        loss_test = criterion(y_pred_test.squeeze(), y_test_torch.to(devi
        test_losses.append(loss_test)
        _, predict_test = torch.max(y_pred_test.to(device), axis = 1)
        accuracy_test = (predict_test == y_test_torch.to(device)).sum().i
        test_accuracies.append(accuracy_test)
        # print info
        if n_epoch % 5 == 0:
            print('Epoch {}: train loss: {}; test loss: {}'.format(n_epoc
            print('Epoch {}: train accuracy: {}; test accuracy: {}'.forma
    # Note: about 60% accuracy for both training and testing. (with n_epo
```

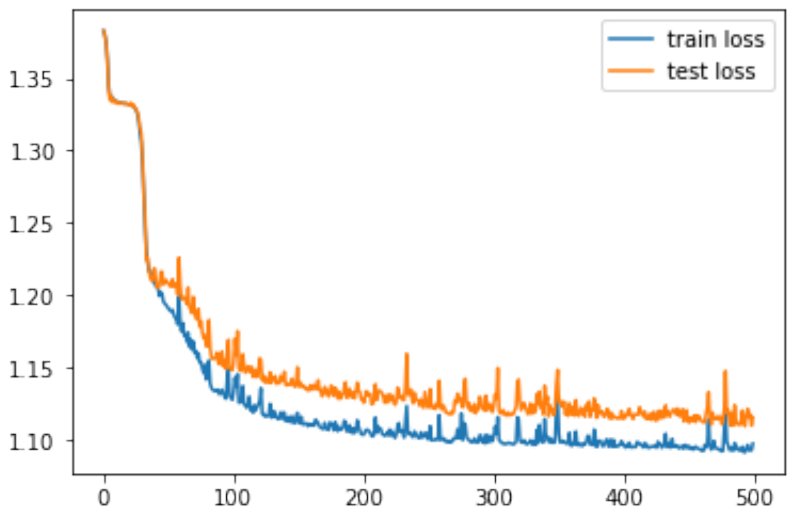
```
Epoch 0: train loss: 1.3834277391433716; test loss: 1.382644653320312
5
Epoch 0: train accuracy: 0.3363329583802025; test accuracy: 0.3543307
086614173
Epoch 5: train loss: 1.3317162990570068; test loss: 1.334838271141052
2
Epoch 5: train accuracy: 0.3363329583802025; test accuracy: 0.3543307
086614173
Epoch 10: train loss: 1.3473947048187256; test loss: 1.33404099941253
66
```

```
66
Epoch 10: train accuracy: 0.3467379077615298; test accuracy: 0.366704
16197975253
Epoch 15: train loss: 1.3273825645446777; test loss: 1.33260667324066
16
Epoch 15: train accuracy: 0.3363329583802025; test accuracy: 0.354330
7086614173
Epoch 20: train loss: 1.3150571584701538; test loss: 1.33140265941619
87
Epoch 20: train accuracy: 0.3363329583802025; test accuracy: 0.354330
7086614173
```

In [8]:

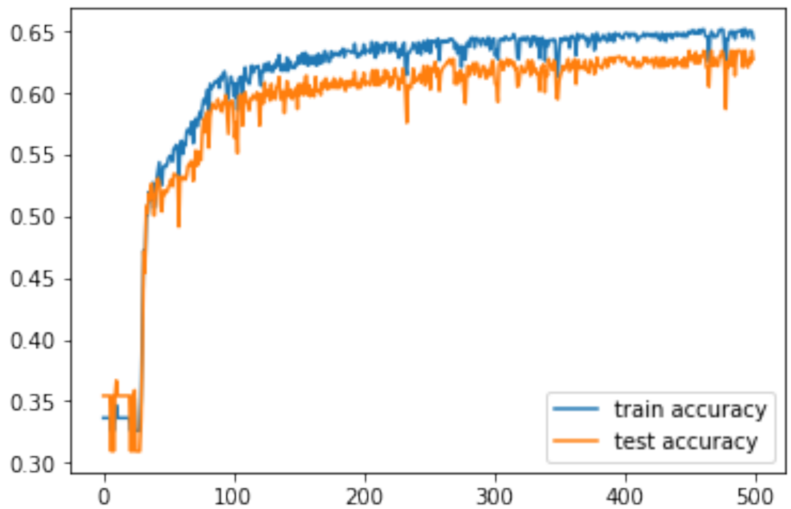
```python
plt.plot(train_losses, label = "train loss")
plt.plot(test_losses, label = "test loss")
plt.legend()
```

Out[8]: <matplotlib.legend.Legend at 0x7fc2103c2128>



In [9]:
```
plt.plot(train_accuracies, label = "train accuracy")
plt.plot(test_accuracies, label = "test accuracy")
plt.legend()
```

Out[9]:
```
<matplotlib.legend.Legend at 0x7fc210363a58>
```



In [10]:
```
torch.save(net.state_dict(), "data/nn_ADAM_8fc")
```

In [ ]: