In [11]:

```
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 [12]:

```
# 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 [13]:

```
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 [14]:

```
# 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[14]:

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

In [15]:

```
##### 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, 50)
        self.fc2 = nn.Linear(50, 50)
        self.fc3 = nn.Linear(50, 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 = self.softmax(x)
        return x
```

In [16]:

```
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 [17]:

```
# training
```

```
In [17]: # 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
    y_pred_train = net(x_train_nhts_torch.to(device))
    loss_train = criterion(y_pred_train.squeeze(), y_train_torch.to(devic
    train_losses.append(loss_train)
    _, predict_train = torch.max(y_pred_train, axis = 1)
    accuracy_train = (predict_train == y_train_torch.to(device)).sum().it
    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(device))
    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().item(
    test_accuracies.append(accuracy_test)
    # print info
    if n_epoch % 5 == 0:
        print('Epoch {}: train loss: {}; test loss: {}'.format(n_epoch, l
        print('Epoch {}: train accuracy: {}; test accuracy: {}'.format(n_
# Note: about 60% accuracy for both training and testing. (with n_epoches
```

```
Epoch 0: train loss: 1.38825523853302; test loss: 1.3879939317703247
Epoch 0: train accuracy: 0.21850393700787402; test accuracy: 0.239595
05061867267
Epoch 5: train loss: 1.381523609161377; test loss: 1.3781293630599976
Epoch 5: train accuracy: 0.3363329583802025; test accuracy: 0.3543307
086614173
Epoch 10: train loss: 1.362502098083496; test loss: 1.368055939674377
4
```

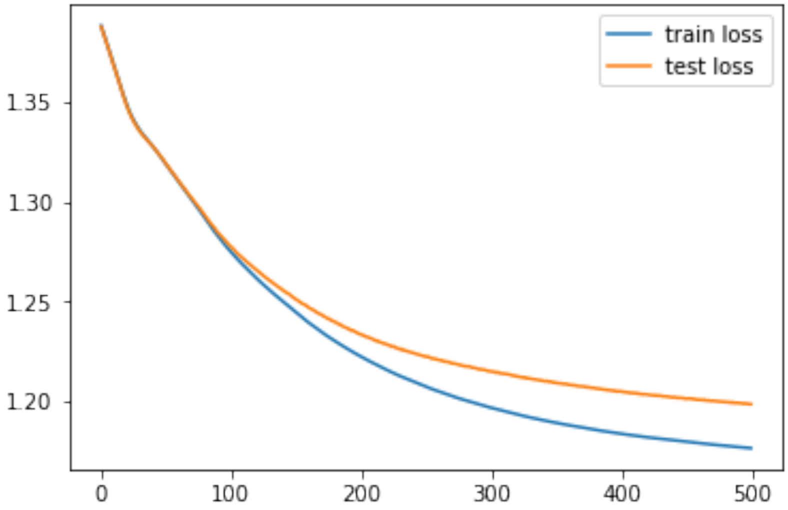
```
Epoch 10: train accuracy: 0.3363329583802025; test accuracy: 0.354330
7086614173
Epoch 15: train loss: 1.3564839363098145; test loss: 1.35739123821258
54
Epoch 15: train accuracy: 0.3363329583802025; test accuracy: 0.354330
7086614173
Epoch 20: train loss: 1.348503828048706; test loss: 1.347620964050293
Epoch 20: train accuracy: 0.3464566929133858; test accuracy: 0.364454
4431946007
Epoch 25: train loss: 1.341056227684021; test loss: 1.340260505676269
5
```

In [18]:

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

Out[18]:

```
<matplotlib.legend.Legend at 0x7f1eb12869e8>
```

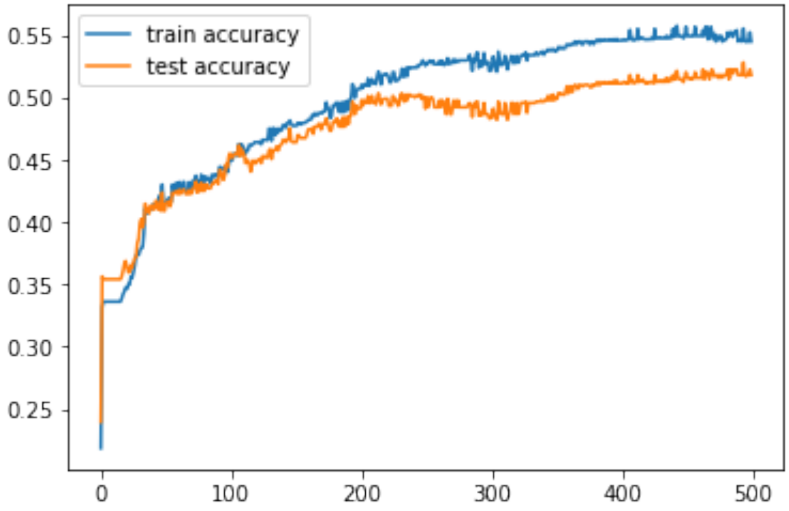


In [19]:

```
plt.plot(train_accuracies, label = "train accuracy")
plt.plot(test_accuracies, label = "test accuracy")
plt.legend()
```

Out[19]:

```
<matplotlib.legend.Legend at 0x7f1eb11e4e80>
```



In [20]:

```
torch.save(net.state_dict(), "data/nn_ADAM_3fc")
```

In [ ]: