HW2_task1

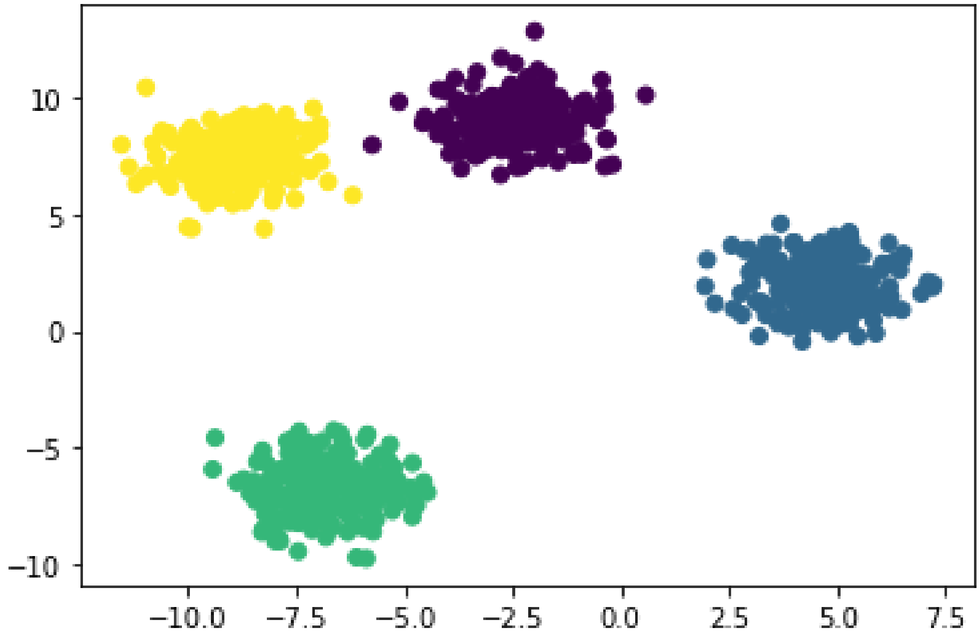
March 15, 2023

This is a implementation of a neural network using PyTorch for a classification task on a generated dataset. The dataset consists of 1000 data points with two features, divided into four clusters. The network has one hidden layer consists of 10 neurons. The input is passed through a ReLU activation function before being passed to the output layer, which has a log-softmax activation. The network is trained using stochastic gradient descent (SGD) with a learning rate of 0.01 and a negative log-likelihood loss function (NLLLoss).

The training is done for 100 epochs, with the training loss and test loss being recorded at each epoch. The training and test losses are plotted over the number of epochs. The final accuracy of the trained model is evaluated on both the training and test sets.

The model achieves a high accuracy on both the training set (99%) and the test set (100%), indicating that it is able to generalize well to unseen data. The plot of the training and test losses shows that the model is not overfitting, as the test loss does not increase while the training loss decreases.

[ ]:
```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import torch.optim as optim
import torch
import torch.nn as nn

# Generate 4 clusters of data points
X, y = make_blobs(n_samples=1000, centers=4, n_features=2, random_state=42)

# Randomly permute the dataset
perm = np.random.permutation(X.shape[0])
X = X[perm]
y = y[perm]

# Split the data into training and testing sets
X_train, y_train = X[:900], y[:900]
X_test, y_test = X[900:], y[900:]
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.show()
```



[ ]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 10)
        self.fc2 = nn.Linear(10, 4)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return torch.log_softmax(x, dim=1)

net = Net()
```

[ ]:
```
train_losses = []
test_losses = []

criterion = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

for epoch in range(100):
    running_loss = 0.0
    for i in range(0, X_train.shape[0], 10):
        inputs = torch.tensor(X_train[i:i+10], dtype=torch.float)
        labels = torch.tensor(y_train[i:i+10], dtype=torch.long)
```

```
        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    # Record the average loss per batch for the training set
    train_losses.append(running_loss / (X_train.shape[0] / 10))

    # Compute the loss on the test set
    with torch.no_grad():
        inputs = torch.tensor(X_test, dtype=torch.float)
        labels = torch.tensor(y_test, dtype=torch.long)
        outputs = net(inputs)
        test_loss = criterion(outputs, labels)
        test_losses.append(test_loss.item())

    print('[%d] train loss: %.3f, test loss: %.3f' % (epoch + 1,␣
↪train_losses[-1], test_losses[-1]))
```

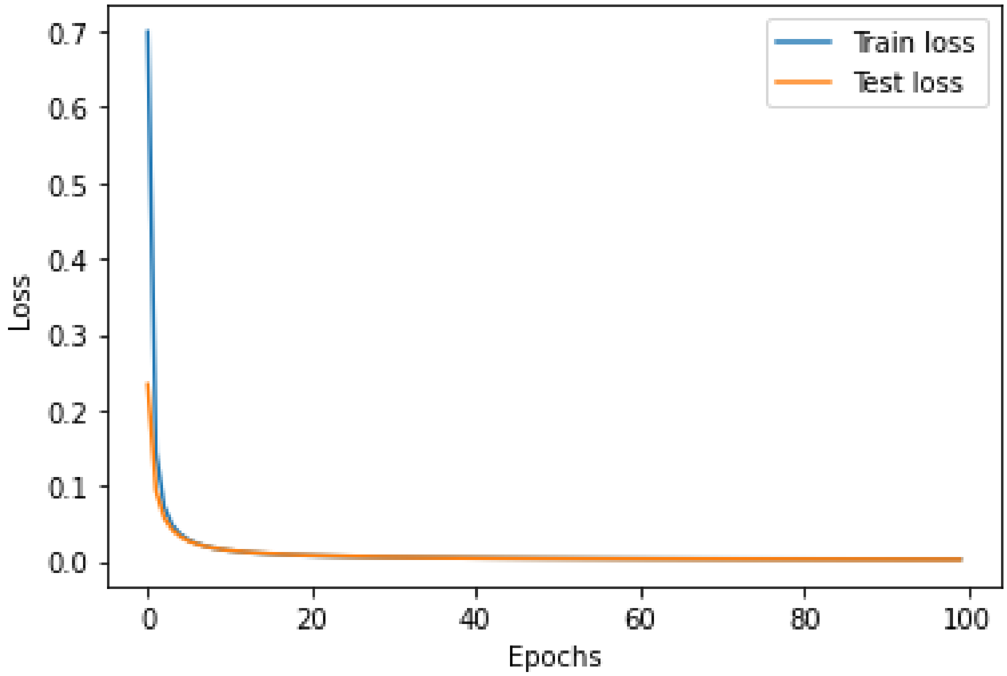
```
[1] train loss: 0.700, test loss: 0.234
[2] train loss: 0.145, test loss: 0.094
[3] train loss: 0.072, test loss: 0.058
[4] train loss: 0.048, test loss: 0.042
[5] train loss: 0.036, test loss: 0.033
[6] train loss: 0.028, test loss: 0.027
[7] train loss: 0.024, test loss: 0.023
[8] train loss: 0.020, test loss: 0.020
[9] train loss: 0.018, test loss: 0.018
[10] train loss: 0.016, test loss: 0.016
[11] train loss: 0.014, test loss: 0.015
[12] train loss: 0.013, test loss: 0.013
[13] train loss: 0.012, test loss: 0.012
[14] train loss: 0.011, test loss: 0.012
[15] train loss: 0.011, test loss: 0.011
[16] train loss: 0.010, test loss: 0.010
[17] train loss: 0.009, test loss: 0.010
[18] train loss: 0.009, test loss: 0.009
[19] train loss: 0.008, test loss: 0.009
[20] train loss: 0.008, test loss: 0.008
[21] train loss: 0.008, test loss: 0.008
[22] train loss: 0.007, test loss: 0.008
[23] train loss: 0.007, test loss: 0.007
[24] train loss: 0.007, test loss: 0.007
[25] train loss: 0.007, test loss: 0.007
```

```
[26] train loss: 0.006, test loss: 0.007
[27] train loss: 0.006, test loss: 0.006
[28] train loss: 0.006, test loss: 0.006
[29] train loss: 0.006, test loss: 0.006
[30] train loss: 0.006, test loss: 0.006
[31] train loss: 0.006, test loss: 0.006
[32] train loss: 0.005, test loss: 0.006
[33] train loss: 0.005, test loss: 0.005
[34] train loss: 0.005, test loss: 0.005
[35] train loss: 0.005, test loss: 0.005
[36] train loss: 0.005, test loss: 0.005
[37] train loss: 0.005, test loss: 0.005
[38] train loss: 0.005, test loss: 0.005
[39] train loss: 0.005, test loss: 0.005
[40] train loss: 0.005, test loss: 0.005
[41] train loss: 0.005, test loss: 0.005
[42] train loss: 0.005, test loss: 0.004
[43] train loss: 0.004, test loss: 0.004
[44] train loss: 0.004, test loss: 0.004
[45] train loss: 0.004, test loss: 0.004
[46] train loss: 0.004, test loss: 0.004
[47] train loss: 0.004, test loss: 0.004
[48] train loss: 0.004, test loss: 0.004
[49] train loss: 0.004, test loss: 0.004
[50] train loss: 0.004, test loss: 0.004
[51] train loss: 0.004, test loss: 0.004
[52] train loss: 0.004, test loss: 0.004
[53] train loss: 0.004, test loss: 0.004
[54] train loss: 0.004, test loss: 0.004
[55] train loss: 0.004, test loss: 0.004
[56] train loss: 0.004, test loss: 0.004
[57] train loss: 0.004, test loss: 0.003
[58] train loss: 0.004, test loss: 0.003
[59] train loss: 0.004, test loss: 0.003
[60] train loss: 0.004, test loss: 0.003
[61] train loss: 0.004, test loss: 0.003
[62] train loss: 0.004, test loss: 0.003
[63] train loss: 0.003, test loss: 0.003
[64] train loss: 0.003, test loss: 0.003
[65] train loss: 0.003, test loss: 0.003
[66] train loss: 0.003, test loss: 0.003
[67] train loss: 0.003, test loss: 0.003
[68] train loss: 0.003, test loss: 0.003
[69] train loss: 0.003, test loss: 0.003
[70] train loss: 0.003, test loss: 0.003
[71] train loss: 0.003, test loss: 0.003
[72] train loss: 0.003, test loss: 0.003
[73] train loss: 0.003, test loss: 0.003
```

```
[74] train loss: 0.003, test loss: 0.003
[75] train loss: 0.003, test loss: 0.003
[76] train loss: 0.003, test loss: 0.003
[77] train loss: 0.003, test loss: 0.003
[78] train loss: 0.003, test loss: 0.003
[79] train loss: 0.003, test loss: 0.003
[80] train loss: 0.003, test loss: 0.003
[81] train loss: 0.003, test loss: 0.003
[82] train loss: 0.003, test loss: 0.003
[83] train loss: 0.003, test loss: 0.003
[84] train loss: 0.003, test loss: 0.003
[85] train loss: 0.003, test loss: 0.003
[86] train loss: 0.003, test loss: 0.003
[87] train loss: 0.003, test loss: 0.003
[88] train loss: 0.003, test loss: 0.003
[89] train loss: 0.003, test loss: 0.003
[90] train loss: 0.003, test loss: 0.003
[91] train loss: 0.003, test loss: 0.003
[92] train loss: 0.003, test loss: 0.002
[93] train loss: 0.003, test loss: 0.002
[94] train loss: 0.003, test loss: 0.002
[95] train loss: 0.003, test loss: 0.002
[96] train loss: 0.003, test loss: 0.002
[97] train loss: 0.003, test loss: 0.002
[98] train loss: 0.003, test loss: 0.002
[99] train loss: 0.003, test loss: 0.002
[100] train loss: 0.003, test loss: 0.002
```

[ ]:

```
plt.plot(train_losses, label='Train loss')
plt.plot(test_losses, label='Test loss')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



[ ]:
```
with torch.no_grad():
    inputs = torch.tensor(X_train, dtype=torch.float)
    labels = torch.tensor(y_train, dtype=torch.long)
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    total = labels.size(0)
    correct = (predicted == labels).sum().item()

print('Accuracy on the training set: %d %%' % (100 * correct / total))

with torch.no_grad():
    inputs = torch.tensor(X_test, dtype=torch.float)
    labels = torch.tensor(y_test, dtype=torch.long)
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    total = labels.size(0)
    correct = (predicted == labels).sum().item()

print('Accuracy on the test set: %d %%' % (100 * correct / total))
```

```
Accuracy on the training set: 99 %
Accuracy on the test set: 100 %
```

HW2_task2

March 15, 2023

The code defines and trains a neural network model to learn the function f(x, y) = x^2 + xy + y^2 on a random set of 5000 points generated from a uniform distribution on the plane [-10, 10] x [-10, 10]. The data is split into training and testing sets with a 90/10 split. The neural network model has one hidden layer, which consists of 64 neurons, and ReLU activation. The model is trained using the mean squared error (MSE) loss function and the Adam optimizer with learning rate of 0.01. The model is trained for 200 epochs, and the batch size is set to 32. At each epoch, the model is trained on the training set and evaluated on the testing set. The training and testing losses are recorded and plotted as a function of the training time. The final training and testing losses are also reported at the end of the training.

[ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Generate 5000 random points (x, y) from a uniform distribution on [-10, 10] x␣
↪[-10, 10]
x = np.random.uniform(low=-10, high=10, size=(5000, 2))

# Define the function f(x, y) = x^2 + xy + y^2
y = x[:, 0]**2 + x[:, 0]*x[:, 1] + x[:, 1]**2

# Reshape y to a column vector
y = y.reshape(-1, 1)

# Split the data into training and testing sets with a 90/10 split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1,␣
↪random_state=42)

# Move data to GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
x_train = torch.tensor(x_train, dtype=torch.float32).to(device)
x_test = torch.tensor(x_test, dtype=torch.float32).to(device)
y_train = torch.tensor(y_train, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test, dtype=torch.float32).to(device)
```

[ ]:

```python
# Define the model
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net().to(device)

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model on GPU
train_loss = []
test_loss = []
for epoch in range(200):
    running_train_loss = 0.0
    running_test_loss = 0.0
    for i in range(0, len(x_train), 32):
        # Train
        inputs = x_train[i:i+32]
        labels = y_train[i:i+32]

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_train_loss += loss.item() * inputs.size(0)

    # Evaluate on test set
    with torch.no_grad():
        for i in range(0, len(x_test), 32):
            inputs = x_test[i:i+32]
            labels = y_test[i:i+32]

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            running_test_loss += loss.item() * inputs.size(0)
```

```
    epoch_train_loss = running_train_loss / len(x_train)
    epoch_test_loss = running_test_loss / len(x_test)
    train_loss.append(epoch_train_loss)
    test_loss.append(epoch_test_loss)

    print(f"Epoch {epoch+1:3d} | Train loss: {epoch_train_loss:.6f} | Test loss:␣
↪{epoch_test_loss:.6f}")
```

```
Epoch   1 | Train loss: 5856.777300 | Test loss: 4491.517270
Epoch   2 | Train loss: 3336.986687 | Test loss: 1976.690430
Epoch   3 | Train loss: 1346.534081 | Test loss: 773.401029
Epoch   4 | Train loss: 654.550887 | Test loss: 535.588102
Epoch   5 | Train loss: 520.628203 | Test loss: 484.840931
Epoch   6 | Train loss: 480.223928 | Test loss: 455.964436
Epoch   7 | Train loss: 454.086652 | Test loss: 432.844595
Epoch   8 | Train loss: 432.282651 | Test loss: 412.202475
Epoch   9 | Train loss: 412.338537 | Test loss: 392.910503
Epoch  10 | Train loss: 393.490713 | Test loss: 374.663848
Epoch  11 | Train loss: 375.502190 | Test loss: 357.289534
Epoch  12 | Train loss: 358.242575 | Test loss: 340.659789
Epoch  13 | Train loss: 341.623235 | Test loss: 324.652339
Epoch  14 | Train loss: 325.569345 | Test loss: 309.188141
Epoch  15 | Train loss: 310.033492 | Test loss: 294.226150
Epoch  16 | Train loss: 294.933388 | Test loss: 279.632452
Epoch  17 | Train loss: 280.207497 | Test loss: 265.407010
Epoch  18 | Train loss: 265.828967 | Test loss: 251.509280
Epoch  19 | Train loss: 251.781757 | Test loss: 237.894889
Epoch  20 | Train loss: 238.014035 | Test loss: 224.537107
Epoch  21 | Train loss: 224.479556 | Test loss: 211.394884
Epoch  22 | Train loss: 211.163343 | Test loss: 198.448694
Epoch  23 | Train loss: 198.106658 | Test loss: 185.799515
Epoch  24 | Train loss: 185.425612 | Test loss: 173.663835
Epoch  25 | Train loss: 173.280431 | Test loss: 162.040337
Epoch  26 | Train loss: 161.688923 | Test loss: 150.940518
Epoch  27 | Train loss: 150.640622 | Test loss: 140.399422
Epoch  28 | Train loss: 140.174912 | Test loss: 130.459874
Epoch  29 | Train loss: 130.331551 | Test loss: 121.172540
Epoch  30 | Train loss: 121.128282 | Test loss: 112.535347
Epoch  31 | Train loss: 112.557181 | Test loss: 104.525226
Epoch  32 | Train loss: 104.616572 | Test loss: 97.150884
Epoch  33 | Train loss: 97.302488 | Test loss: 90.414604
Epoch  34 | Train loss: 90.598669 | Test loss: 84.263615
Epoch  35 | Train loss: 84.472387 | Test loss: 78.695864
Epoch  36 | Train loss: 78.888396 | Test loss: 73.639963
Epoch  37 | Train loss: 73.805275 | Test loss: 69.054317
Epoch  38 | Train loss: 69.187067 | Test loss: 64.907319
Epoch  39 | Train loss: 64.979316 | Test loss: 61.147170
```

```
Epoch  40 | Train loss: 61.143590 | Test loss: 57.732182
Epoch  41 | Train loss: 57.645259 | Test loss: 54.617910
Epoch  42 | Train loss: 54.442421 | Test loss: 51.785044
Epoch  43 | Train loss: 51.498891 | Test loss: 49.167587
Epoch  44 | Train loss: 48.774182 | Test loss: 46.739027
Epoch  45 | Train loss: 46.251618 | Test loss: 44.471678
Epoch  46 | Train loss: 43.902727 | Test loss: 42.352100
Epoch  47 | Train loss: 41.697260 | Test loss: 40.342971
Epoch  48 | Train loss: 39.613388 | Test loss: 38.435758
Epoch  49 | Train loss: 37.637631 | Test loss: 36.609374
Epoch  50 | Train loss: 35.763128 | Test loss: 34.863918
Epoch  51 | Train loss: 33.982912 | Test loss: 33.194711
Epoch  52 | Train loss: 32.276577 | Test loss: 31.602441
Epoch  53 | Train loss: 30.638842 | Test loss: 30.088493
Epoch  54 | Train loss: 29.069500 | Test loss: 28.639283
Epoch  55 | Train loss: 27.570091 | Test loss: 27.252241
Epoch  56 | Train loss: 26.135863 | Test loss: 25.909296
Epoch  57 | Train loss: 24.754468 | Test loss: 24.612653
Epoch  58 | Train loss: 23.423801 | Test loss: 23.351341
Epoch  59 | Train loss: 22.144631 | Test loss: 22.134856
Epoch  60 | Train loss: 20.914554 | Test loss: 20.948720
Epoch  61 | Train loss: 19.731542 | Test loss: 19.812609
Epoch  62 | Train loss: 18.595496 | Test loss: 18.730342
Epoch  63 | Train loss: 17.505996 | Test loss: 17.689298
Epoch  64 | Train loss: 16.460768 | Test loss: 16.696817
Epoch  65 | Train loss: 15.462396 | Test loss: 15.752924
Epoch  66 | Train loss: 14.510775 | Test loss: 14.848936
Epoch  67 | Train loss: 13.606289 | Test loss: 13.991003
Epoch  68 | Train loss: 12.751478 | Test loss: 13.184846
Epoch  69 | Train loss: 11.945278 | Test loss: 12.417578
Epoch  70 | Train loss: 11.185289 | Test loss: 11.677351
Epoch  71 | Train loss: 10.469312 | Test loss: 10.974829
Epoch  72 | Train loss: 9.794736 | Test loss: 10.309331
Epoch  73 | Train loss: 9.155422 | Test loss: 9.677706
Epoch  74 | Train loss: 8.553867 | Test loss: 9.087158
Epoch  75 | Train loss: 7.991909 | Test loss: 8.531565
Epoch  76 | Train loss: 7.461555 | Test loss: 8.008160
Epoch  77 | Train loss: 6.963404 | Test loss: 7.518998
Epoch  78 | Train loss: 6.497920 | Test loss: 7.061846
Epoch  79 | Train loss: 6.067349 | Test loss: 6.635368
Epoch  80 | Train loss: 5.668870 | Test loss: 6.236466
Epoch  81 | Train loss: 5.298138 | Test loss: 5.866126
Epoch  82 | Train loss: 4.952974 | Test loss: 5.515210
Epoch  83 | Train loss: 4.632550 | Test loss: 5.190822
Epoch  84 | Train loss: 4.333771 | Test loss: 4.887172
Epoch  85 | Train loss: 4.055543 | Test loss: 4.603534
Epoch  86 | Train loss: 3.795846 | Test loss: 4.338200
Epoch  87 | Train loss: 3.554163 | Test loss: 4.092868
```

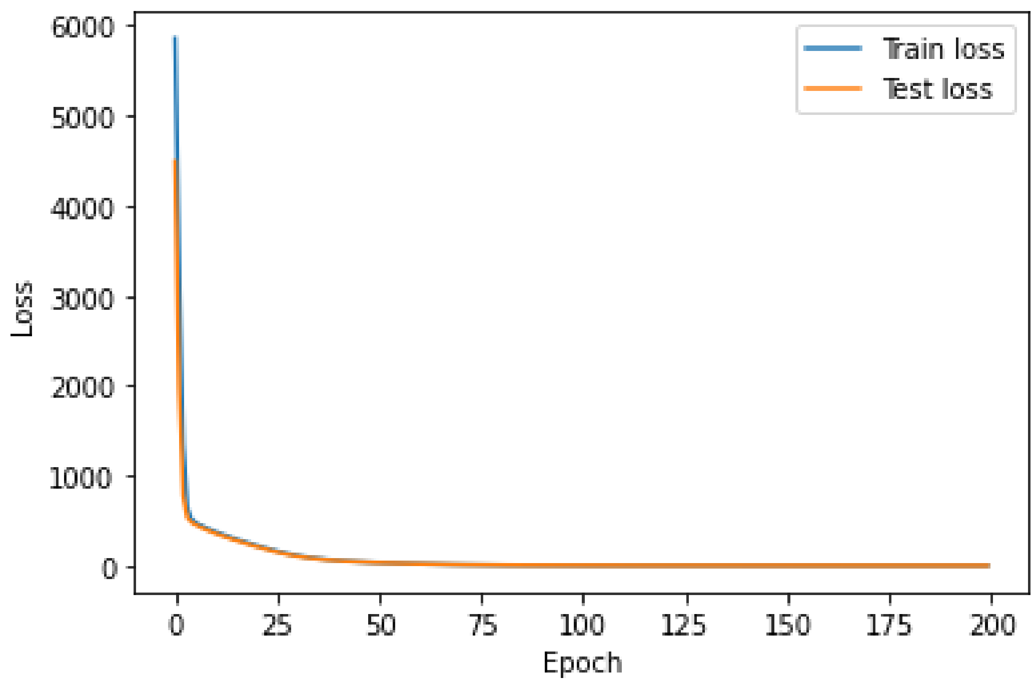
```
Epoch  88 | Train loss: 3.329803 | Test loss: 3.862264
Epoch  89 | Train loss: 3.121820 | Test loss: 3.646956
Epoch  90 | Train loss: 2.928700 | Test loss: 3.444850
Epoch  91 | Train loss: 2.749588 | Test loss: 3.259068
Epoch  92 | Train loss: 2.583342 | Test loss: 3.085827
Epoch  93 | Train loss: 2.429469 | Test loss: 2.925901
Epoch  94 | Train loss: 2.286190 | Test loss: 2.773680
Epoch  95 | Train loss: 2.152559 | Test loss: 2.629618
Epoch  96 | Train loss: 2.027891 | Test loss: 2.496135
Epoch  97 | Train loss: 1.911489 | Test loss: 2.369933
Epoch  98 | Train loss: 1.802684 | Test loss: 2.252359
Epoch  99 | Train loss: 1.702525 | Test loss: 2.140007
Epoch 100 | Train loss: 1.608868 | Test loss: 2.034733
Epoch 101 | Train loss: 1.521517 | Test loss: 1.933120
Epoch 102 | Train loss: 1.439643 | Test loss: 1.834898
Epoch 103 | Train loss: 1.363022 | Test loss: 1.740068
Epoch 104 | Train loss: 1.291177 | Test loss: 1.647117
Epoch 105 | Train loss: 1.223872 | Test loss: 1.557752
Epoch 106 | Train loss: 1.161032 | Test loss: 1.472091
Epoch 107 | Train loss: 1.101835 | Test loss: 1.392568
Epoch 108 | Train loss: 1.046317 | Test loss: 1.319446
Epoch 109 | Train loss: 0.994280 | Test loss: 1.249531
Epoch 110 | Train loss: 0.944967 | Test loss: 1.182960
Epoch 111 | Train loss: 0.898102 | Test loss: 1.118356
Epoch 112 | Train loss: 0.854062 | Test loss: 1.058363
Epoch 113 | Train loss: 0.812829 | Test loss: 1.004405
Epoch 114 | Train loss: 0.774280 | Test loss: 0.952894
Epoch 115 | Train loss: 0.738010 | Test loss: 0.902732
Epoch 116 | Train loss: 0.703632 | Test loss: 0.857586
Epoch 117 | Train loss: 0.670648 | Test loss: 0.816165
Epoch 118 | Train loss: 0.639734 | Test loss: 0.777989
Epoch 119 | Train loss: 0.610748 | Test loss: 0.741079
Epoch 120 | Train loss: 0.583565 | Test loss: 0.708820
Epoch 121 | Train loss: 0.558340 | Test loss: 0.679649
Epoch 122 | Train loss: 0.534889 | Test loss: 0.652153
Epoch 123 | Train loss: 0.512590 | Test loss: 0.626799
Epoch 124 | Train loss: 0.492039 | Test loss: 0.602061
Epoch 125 | Train loss: 0.472628 | Test loss: 0.578857
Epoch 126 | Train loss: 0.454283 | Test loss: 0.555963
Epoch 127 | Train loss: 0.436606 | Test loss: 0.535173
Epoch 128 | Train loss: 0.419995 | Test loss: 0.516468
Epoch 129 | Train loss: 0.404159 | Test loss: 0.497006
Epoch 130 | Train loss: 0.389545 | Test loss: 0.480207
Epoch 131 | Train loss: 0.375571 | Test loss: 0.463534
Epoch 132 | Train loss: 0.362134 | Test loss: 0.447794
Epoch 133 | Train loss: 0.349353 | Test loss: 0.432452
Epoch 134 | Train loss: 0.336874 | Test loss: 0.417220
Epoch 135 | Train loss: 0.325167 | Test loss: 0.400819
```

```
Epoch 136 | Train loss: 0.314104 | Test loss: 0.386807
Epoch 137 | Train loss: 0.303613 | Test loss: 0.372552
Epoch 138 | Train loss: 0.293398 | Test loss: 0.362485
Epoch 139 | Train loss: 0.283860 | Test loss: 0.351804
Epoch 140 | Train loss: 0.274696 | Test loss: 0.342148
Epoch 141 | Train loss: 0.266033 | Test loss: 0.333181
Epoch 142 | Train loss: 0.257792 | Test loss: 0.322504
Epoch 143 | Train loss: 0.250088 | Test loss: 0.312897
Epoch 144 | Train loss: 0.243173 | Test loss: 0.306793
Epoch 145 | Train loss: 0.236560 | Test loss: 0.298758
Epoch 146 | Train loss: 0.230513 | Test loss: 0.290423
Epoch 147 | Train loss: 0.224906 | Test loss: 0.284053
Epoch 148 | Train loss: 0.219685 | Test loss: 0.276969
Epoch 149 | Train loss: 0.214624 | Test loss: 0.269828
Epoch 150 | Train loss: 0.209836 | Test loss: 0.263546
Epoch 151 | Train loss: 0.205451 | Test loss: 0.257618
Epoch 152 | Train loss: 0.201117 | Test loss: 0.251383
Epoch 153 | Train loss: 0.196896 | Test loss: 0.245665
Epoch 154 | Train loss: 0.192964 | Test loss: 0.240968
Epoch 155 | Train loss: 0.189074 | Test loss: 0.236763
Epoch 156 | Train loss: 0.185386 | Test loss: 0.232548
Epoch 157 | Train loss: 0.181691 | Test loss: 0.227611
Epoch 158 | Train loss: 0.178445 | Test loss: 0.223676
Epoch 159 | Train loss: 0.175343 | Test loss: 0.219192
Epoch 160 | Train loss: 0.172319 | Test loss: 0.214863
Epoch 161 | Train loss: 0.169436 | Test loss: 0.210761
Epoch 162 | Train loss: 0.166712 | Test loss: 0.206577
Epoch 163 | Train loss: 0.164055 | Test loss: 0.203197
Epoch 164 | Train loss: 0.161630 | Test loss: 0.198900
Epoch 165 | Train loss: 0.159186 | Test loss: 0.194594
Epoch 166 | Train loss: 0.156533 | Test loss: 0.190014
Epoch 167 | Train loss: 0.153946 | Test loss: 0.185499
Epoch 168 | Train loss: 0.151695 | Test loss: 0.181589
Epoch 169 | Train loss: 0.149445 | Test loss: 0.175769
Epoch 170 | Train loss: 0.147176 | Test loss: 0.171181
Epoch 171 | Train loss: 0.144845 | Test loss: 0.165846
Epoch 172 | Train loss: 0.142833 | Test loss: 0.162096
Epoch 173 | Train loss: 0.140810 | Test loss: 0.157704
Epoch 174 | Train loss: 0.138775 | Test loss: 0.153456
Epoch 175 | Train loss: 0.136540 | Test loss: 0.148475
Epoch 176 | Train loss: 0.134247 | Test loss: 0.143552
Epoch 177 | Train loss: 0.132160 | Test loss: 0.139526
Epoch 178 | Train loss: 0.130086 | Test loss: 0.135793
Epoch 179 | Train loss: 0.128307 | Test loss: 0.132898
Epoch 180 | Train loss: 0.126734 | Test loss: 0.129659
Epoch 181 | Train loss: 0.125260 | Test loss: 0.126910
Epoch 182 | Train loss: 0.123871 | Test loss: 0.124557
Epoch 183 | Train loss: 0.122560 | Test loss: 0.122412
```

```
Epoch 184 | Train loss: 0.121229 | Test loss: 0.120414
Epoch 185 | Train loss: 0.120076 | Test loss: 0.118468
Epoch 186 | Train loss: 0.118763 | Test loss: 0.115934
Epoch 187 | Train loss: 0.117528 | Test loss: 0.114579
Epoch 188 | Train loss: 0.116572 | Test loss: 0.113097
Epoch 189 | Train loss: 0.115406 | Test loss: 0.111992
Epoch 190 | Train loss: 0.114441 | Test loss: 0.111067
Epoch 191 | Train loss: 0.113392 | Test loss: 0.109863
Epoch 192 | Train loss: 0.112567 | Test loss: 0.108369
Epoch 193 | Train loss: 0.111612 | Test loss: 0.107399
Epoch 194 | Train loss: 0.110757 | Test loss: 0.106470
Epoch 195 | Train loss: 0.109925 | Test loss: 0.105691
Epoch 196 | Train loss: 0.108963 | Test loss: 0.104535
Epoch 197 | Train loss: 0.108087 | Test loss: 0.103260
Epoch 198 | Train loss: 0.107124 | Test loss: 0.102411
Epoch 199 | Train loss: 0.106323 | Test loss: 0.101363
Epoch 200 | Train loss: 0.105617 | Test loss: 0.100108
```

[ ]:

```python
# Plot the train and test loss as a function of the training time
plt.plot(train_loss, label='Train loss')
plt.plot(test_loss, label='Test loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

# Report the final training and test loss
print(f"Final train loss: {train_loss[-1]:.6f}")
print(f"Final test loss: {test_loss[-1]:.6f}")
```




```
Final train loss: 0.105617
Final test loss: 0.100108
```

HW2_task3

March 15, 2023

1 Task 3

This code defines a Siamese Network in PyTorch. The Siamese network is a type of neural network architecture that is typically used for tasks involving image similarity estimation, such as face recognition, signature verification, and image retrieval. The Siamese network takes two input images and feeds them through two identical networks with shared weights, which produces two feature vectors. These feature vectors are concatenated and passed through a linear layer to produce a similarity score between the two input images.

This implementation of the Siamese network uses the ResNet-18 model as the feature extractor, and it is trained on only 10% of the MNIST training dataset,and use only 10% of MNIST test data set as the test data set. The code defines a PyTorch dataset class called APP_MATCHER, which groups the MNIST dataset examples based on class. For every example, the **getitem** method selects two images. For positive examples, two images are selected from the same class and the label is set to 1, while for negative examples, two images are selected from different classes, and the label is set to 0. The forward method of the SiameseNetwork class takes two input images and returns the similarity score between them.

[1]:
```
from __future__ import print_function
import argparse, random, copy
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision import transforms as T
from torch.optim.lr_scheduler import StepLR
```

[2]:
```
class SiameseNetwork(nn.Module):
    """
        Siamese network for image similarity estimation.
        The network is composed of two identical networks, one for each input.
        The output of each network is concatenated and passed to a linear layer.
        The output of the linear layer passed through a sigmoid function.
```

```
        `"FaceNet" <https://arxiv.org/pdf/1503.03832.pdf>`_ is a variant of the␣
↪Siamese network.
        This implementation varies from FaceNet as we use the `ResNet-18` model␣
↪from
        `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/
↪1512.03385.pdf>`_ as our feature extractor.
        In addition, we aren't using `TripletLoss` as the MNIST dataset is␣
↪simple, so `BCELoss` can do the trick.
    """
    def __init__(self):
        super(SiameseNetwork, self).__init__()
        # get resnet model
        self.resnet = torchvision.models.resnet18(weights=None)

        # over-write the first conv layer to be able to read MNIST images
        # as resnet18 reads (3,x,x) where 3 is RGB channels
        # whereas MNIST has (1,x,x) where 1 is a gray-scale channel
        self.resnet.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),␣
↪padding=(3, 3), bias=False)
        self.fc_in_features = self.resnet.fc.in_features

        # remove the last layer of resnet18 (linear layer which is before␣
↪avgpool layer)
        self.resnet = torch.nn.Sequential(*(list(self.resnet.children())[:-1]))

        # add linear layers to compare between the features of the two images
        self.fc = nn.Sequential(
            nn.Linear(self.fc_in_features * 2, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 1),
        )

        self.sigmoid = nn.Sigmoid()

        # initialize the weights
        self.resnet.apply(self.init_weights)
        self.fc.apply(self.init_weights)

    def init_weights(self, m):
        if isinstance(m, nn.Linear):
            torch.nn.init.xavier_uniform_(m.weight)
            m.bias.data.fill_(0.01)

    def forward_once(self, x):
        output = self.resnet(x)
        output = output.view(output.size()[0], -1)
```

```
        return output

    def forward(self, input1, input2):
        # get two images' features
        output1 = self.forward_once(input1)
        output2 = self.forward_once(input2)

        # concatenate both images' features
        output = torch.cat((output1, output2), 1)

        # pass the concatenation to the linear layers
        output = self.fc(output)

        # pass the out of the linear layers to sigmoid layer
        output = self.sigmoid(output)

        return output
```

[3]:
```
class APP_MATCHER(Dataset):
    def __init__(self, root, train, download=False):
        super(APP_MATCHER, self).__init__()

        # get MNIST dataset
        self.dataset = datasets.MNIST(root, train=train, download=download)

        # as `self.dataset.data`'s shape is (Nx28x28), where N is the number of
        # examples in MNIST dataset, a single example has the dimensions of
        # (28x28) for (WxH), where W and H are the width and the height of the␣
↪image.
        # However, every example should have (CxWxH) dimensions where C is the␣
↪number
        # of channels to be passed to the network. As MNIST contains gray-scale␣
↪images,
        # we add an additional dimension to corresponds to the number of␣
↪channels.
        self.data = self.dataset.data.unsqueeze(1).clone()

        self.group_examples()

    def group_examples(self):
        """
            To ease the accessibility of data based on the class, we will use␣
↪`group_examples` to group
            examples based on class.

            Every key in `grouped_examples` corresponds to a class in MNIST␣
↪dataset. For every key in
```

```
            `grouped_examples`, every value will conform to all of the indices␣
↪for the MNIST
            dataset examples that correspond to that key.
        """

        # get the targets from MNIST dataset
        np_arr = np.array(self.dataset.targets.clone())

        # group examples based on class
        self.grouped_examples = {}
        for i in range(0,10):
            self.grouped_examples[i] = np.where((np_arr==i))[0]

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, index):
        """
            For every example, we will select two images. There are two cases,
            positive and negative examples. For positive examples, we will have␣
↪two
            images from the same class. For negative examples, we will have two␣
↪images
            from different classes.
            Given an index, if the index is even, we will pick the second image␣
↪from the same class,
            but it won't be the same image we chose for the first class. This is␣
↪used to ensure the positive
            example isn't trivial as the network would easily distinguish the␣
↪similarity between same images. However,
            if the network were given two different images from the same class,␣
↪the network will need to learn
            the similarity between two different images representing the same␣
↪class. If the index is odd, we will
            pick the second image from a different class than the first image.
        """

        # pick some random class for the first image
        selected_class = random.randint(0, 9)

        # pick a random index for the first image in the grouped indices based␣
↪of the label
        # of the class
        random_index_1 = random.randint(0, self.grouped_examples[selected_class].
↪shape[0]-1)
```

```
        # pick the index to get the first image
        index_1 = self.grouped_examples[selected_class][random_index_1]

        # get the first image
        image_1 = self.data[index_1].clone().float()

        # same class
        if index % 2 == 0:
            # pick a random index for the second image
            random_index_2 = random.randint(0, self.
↪grouped_examples[selected_class].shape[0]-1)

            # ensure that the index of the second image isn't the same as the␣
↪first image
            while random_index_2 == random_index_1:
                random_index_2 = random.randint(0, self.
↪grouped_examples[selected_class].shape[0]-1)

            # pick the index to get the second image
            index_2 = self.grouped_examples[selected_class][random_index_2]

            # get the second image
            image_2 = self.data[index_2].clone().float()

            # set the label for this example to be positive (1)
            target = torch.tensor(1, dtype=torch.float)

        # different class
        else:
            # pick a random class
            other_selected_class = random.randint(0, 9)

            # ensure that the class of the second image isn't the same as the␣
↪first image
            while other_selected_class == selected_class:
                other_selected_class = random.randint(0, 9)


            # pick a random index for the second image in the grouped indices␣
↪based of the label
            # of the class
            random_index_2 = random.randint(0, self.
↪grouped_examples[other_selected_class].shape[0]-1)

            # pick the index to get the second image
            index_2 = self.grouped_examples[other_selected_class][random_index_2]
```

```
            # get the second image
            image_2 = self.data[index_2].clone().float()

            # set the label for this example to be negative (0)
            target = torch.tensor(0, dtype=torch.float)

        return image_1, image_2, target
```

[ ]:
```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()

    # we aren't using `TripletLoss` as the MNIST dataset is simple, so `BCELoss`␣
 ↪can do the trick.
    criterion = nn.BCELoss()

    for batch_idx, (images_1, images_2, targets) in enumerate(train_loader):
        images_1, images_2, targets = images_1.to(device), images_2.to(device),␣
 ↪targets.to(device)
        optimizer.zero_grad()
        outputs = model(images_1, images_2).squeeze()
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(images_1), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break


def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    # we aren't using `TripletLoss` as the MNIST dataset is simple, so `BCELoss`␣
 ↪can do the trick.
    criterion = nn.BCELoss()

    with torch.no_grad():
        for (images_1, images_2, targets) in test_loader:
            images_1, images_2, targets = images_1.to(device), images_2.
 ↪to(device), targets.to(device)
            outputs = model(images_1, images_2).squeeze()
```

```
            test_loss += criterion(outputs, targets).sum().item()  # sum up␣
↪batch loss
            pred = torch.where(outputs > 0.5, 1, 0)  # get the index of the max␣
↪log-probability
            correct += pred.eq(targets.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    # for the 1st epoch, the average loss is 0.0001 and the accuracy 97-98%
    # using default settings. After completing the 10th epoch, the average
    # loss is 0.0000 and the accuracy 99.5-100% using default settings.
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

[5]:

```
def main():
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch Siamese network␣
↪Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=14, metavar='N',
                        help='number of epochs to train (default: 14)')
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--no-mps', action='store_true', default=False,
                        help='disables macOS GPU training')
    parser.add_argument('--dry-run', action='store_true', default=False,
                        help='quickly check a single pass')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training␣
↪status')
    parser.add_argument('--save-model', action='store_true', default=False,
                        help='For Saving the current Model')
    parser.add_argument('-f')
    args = parser.parse_args()

    use_cuda = not args.no_cuda and torch.cuda.is_available()
    use_mps = not args.no_mps and torch.backends.mps.is_available()
```

```python
    torch.manual_seed(args.seed)

    if use_cuda:
        device = torch.device("cuda")
    elif use_mps:
        device = torch.device("mps")
    else:
        device = torch.device("cpu")

    train_kwargs = {'batch_size': args.batch_size}
    test_kwargs = {'batch_size': args.test_batch_size}
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)

    train_dataset = APP_MATCHER('../data', train=True, download=True)
    train_indices = torch.randperm(len(train_dataset))[:len(train_dataset)//10]
    train_dataset = torch.utils.data.Subset(train_dataset, train_indices)
    train_loader = torch.utils.data.DataLoader(train_dataset,**train_kwargs)

    test_dataset = APP_MATCHER('../data', train=False, download=True)
    test_indices = torch.randperm(len(test_dataset))[:len(test_dataset)//10]
    test_dataset = torch.utils.data.Subset(test_dataset, test_indices)
    test_loader = torch.utils.data.DataLoader(test_dataset, **test_kwargs)

    model = SiameseNetwork().to(device)
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    for epoch in range(1, 10):
        train(args, model, device, train_loader, optimizer, epoch)
        test(model, device, test_loader)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "siamese_network.pt")


if __name__ == '__main__':
    main()
```

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

```
  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
../data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
../data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
../data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw

Train Epoch: 1 [0/6000 (0%)]    Loss: 1.175190
Train Epoch: 1 [640/6000 (11%)] Loss: 0.682234
Train Epoch: 1 [1280/6000 (21%)]        Loss: 0.652923
Train Epoch: 1 [1920/6000 (32%)]        Loss: 0.623879
Train Epoch: 1 [2560/6000 (43%)]        Loss: 0.627442
Train Epoch: 1 [3200/6000 (53%)]        Loss: 0.501702
Train Epoch: 1 [3840/6000 (64%)]        Loss: 0.323163
Train Epoch: 1 [4480/6000 (74%)]        Loss: 0.534408
Train Epoch: 1 [5120/6000 (85%)]        Loss: 0.411011
Train Epoch: 1 [5760/6000 (96%)]        Loss: 0.383505

Test set: Average loss: 0.0005, Accuracy: 801/1000 (80%)

Train Epoch: 2 [0/6000 (0%)]    Loss: 0.448275
Train Epoch: 2 [640/6000 (11%)] Loss: 0.414833
Train Epoch: 2 [1280/6000 (21%)]        Loss: 0.232005
Train Epoch: 2 [1920/6000 (32%)]        Loss: 0.364514
Train Epoch: 2 [2560/6000 (43%)]        Loss: 0.338736
Train Epoch: 2 [3200/6000 (53%)]        Loss: 0.262378
Train Epoch: 2 [3840/6000 (64%)]        Loss: 0.345625
Train Epoch: 2 [4480/6000 (74%)]        Loss: 0.234799
Train Epoch: 2 [5120/6000 (85%)]        Loss: 0.162417
Train Epoch: 2 [5760/6000 (96%)]        Loss: 0.205213
```

```
Test set: Average loss: 0.0003, Accuracy: 906/1000 (91%)

Train Epoch: 3 [0/6000 (0%)]    Loss: 0.191834
Train Epoch: 3 [640/6000 (11%)] Loss: 0.150406
Train Epoch: 3 [1280/6000 (21%)]        Loss: 0.250022
Train Epoch: 3 [1920/6000 (32%)]        Loss: 0.151487
Train Epoch: 3 [2560/6000 (43%)]        Loss: 0.104292
Train Epoch: 3 [3200/6000 (53%)]        Loss: 0.106918
Train Epoch: 3 [3840/6000 (64%)]        Loss: 0.143443
Train Epoch: 3 [4480/6000 (74%)]        Loss: 0.064529
Train Epoch: 3 [5120/6000 (85%)]        Loss: 0.227299
Train Epoch: 3 [5760/6000 (96%)]        Loss: 0.222159

Test set: Average loss: 0.0002, Accuracy: 927/1000 (93%)

Train Epoch: 4 [0/6000 (0%)]    Loss: 0.217177
Train Epoch: 4 [640/6000 (11%)] Loss: 0.099984
Train Epoch: 4 [1280/6000 (21%)]        Loss: 0.058514
Train Epoch: 4 [1920/6000 (32%)]        Loss: 0.125665
Train Epoch: 4 [2560/6000 (43%)]        Loss: 0.195694
Train Epoch: 4 [3200/6000 (53%)]        Loss: 0.113017
Train Epoch: 4 [3840/6000 (64%)]        Loss: 0.097260
Train Epoch: 4 [4480/6000 (74%)]        Loss: 0.123328
Train Epoch: 4 [5120/6000 (85%)]        Loss: 0.116845
Train Epoch: 4 [5760/6000 (96%)]        Loss: 0.093454

Test set: Average loss: 0.0001, Accuracy: 963/1000 (96%)

Train Epoch: 5 [0/6000 (0%)]    Loss: 0.094434
Train Epoch: 5 [640/6000 (11%)] Loss: 0.196890
Train Epoch: 5 [1280/6000 (21%)]        Loss: 0.195391
Train Epoch: 5 [1920/6000 (32%)]        Loss: 0.162111
Train Epoch: 5 [2560/6000 (43%)]        Loss: 0.205471
Train Epoch: 5 [3200/6000 (53%)]        Loss: 0.321440
Train Epoch: 5 [3840/6000 (64%)]        Loss: 0.103642
Train Epoch: 5 [4480/6000 (74%)]        Loss: 0.091943
Train Epoch: 5 [5120/6000 (85%)]        Loss: 0.173878
Train Epoch: 5 [5760/6000 (96%)]        Loss: 0.075673

Test set: Average loss: 0.0001, Accuracy: 974/1000 (97%)

Train Epoch: 6 [0/6000 (0%)]    Loss: 0.078129
Train Epoch: 6 [640/6000 (11%)] Loss: 0.133610
Train Epoch: 6 [1280/6000 (21%)]        Loss: 0.078885
Train Epoch: 6 [1920/6000 (32%)]        Loss: 0.119538
Train Epoch: 6 [2560/6000 (43%)]        Loss: 0.109704
Train Epoch: 6 [3200/6000 (53%)]        Loss: 0.117903
```

```
Train Epoch: 6 [3840/6000 (64%)]        Loss: 0.041985
Train Epoch: 6 [4480/6000 (74%)]        Loss: 0.055203
Train Epoch: 6 [5120/6000 (85%)]        Loss: 0.057204
Train Epoch: 6 [5760/6000 (96%)]        Loss: 0.041579

Test set: Average loss: 0.0001, Accuracy: 977/1000 (98%)

Train Epoch: 7 [0/6000 (0%)]    Loss: 0.064444
Train Epoch: 7 [640/6000 (11%)] Loss: 0.059792
Train Epoch: 7 [1280/6000 (21%)]        Loss: 0.035186
Train Epoch: 7 [1920/6000 (32%)]        Loss: 0.135161
Train Epoch: 7 [2560/6000 (43%)]        Loss: 0.058195
Train Epoch: 7 [3200/6000 (53%)]        Loss: 0.129944
Train Epoch: 7 [3840/6000 (64%)]        Loss: 0.025679
Train Epoch: 7 [4480/6000 (74%)]        Loss: 0.213292
Train Epoch: 7 [5120/6000 (85%)]        Loss: 0.126770
Train Epoch: 7 [5760/6000 (96%)]        Loss: 0.164390

Test set: Average loss: 0.0000, Accuracy: 986/1000 (99%)

Train Epoch: 8 [0/6000 (0%)]    Loss: 0.027620
Train Epoch: 8 [640/6000 (11%)] Loss: 0.050954
Train Epoch: 8 [1280/6000 (21%)]        Loss: 0.056885
Train Epoch: 8 [1920/6000 (32%)]        Loss: 0.053830
Train Epoch: 8 [2560/6000 (43%)]        Loss: 0.029372
Train Epoch: 8 [3200/6000 (53%)]        Loss: 0.078245
Train Epoch: 8 [3840/6000 (64%)]        Loss: 0.010862
Train Epoch: 8 [4480/6000 (74%)]        Loss: 0.042567
Train Epoch: 8 [5120/6000 (85%)]        Loss: 0.018613
Train Epoch: 8 [5760/6000 (96%)]        Loss: 0.078957

Test set: Average loss: 0.0000, Accuracy: 983/1000 (98%)

Train Epoch: 9 [0/6000 (0%)]    Loss: 0.133632
Train Epoch: 9 [640/6000 (11%)] Loss: 0.012117
Train Epoch: 9 [1280/6000 (21%)]        Loss: 0.147912
Train Epoch: 9 [1920/6000 (32%)]        Loss: 0.057928
Train Epoch: 9 [2560/6000 (43%)]        Loss: 0.017033
Train Epoch: 9 [3200/6000 (53%)]        Loss: 0.027200
Train Epoch: 9 [3840/6000 (64%)]        Loss: 0.064777
Train Epoch: 9 [4480/6000 (74%)]        Loss: 0.071504
Train Epoch: 9 [5120/6000 (85%)]        Loss: 0.069630
Train Epoch: 9 [5760/6000 (96%)]        Loss: 0.007540

Test set: Average loss: 0.0001, Accuracy: 980/1000 (98%)
```

```
[ ]:
```

HW2_task4

March 14, 2023

Using the code from https://github.com/drgripa1/resnet-cifar10, ResNet model was fitted on CIFAR10 dataset with different optimizer, SGD, SGD with momentum 0.9 and Adam, also learning rates, 01, 0.3 and 0.5. The code below shows the instance of ResNet model with momentum SGD and learning rate of 0.5. I collected data of each case by replacing the "lr" value in parse_args_train function and optimzier used in class ResNetModel.

From the plots at the end, we can clearly see that learning rate of 0.1 performs best in momentum SGD and Adam, and learning rate of 0.1 performs best in SGD. Learning rate of 0.5 perform worst for all three optimzers as large learning rate leads to instability and overfitting, and may overshoot the optimal.

From the test train error recorded below, we can see that momentum sgd with learning rate of 0.1 has the lowest error, but Adam with learing rate of 0.5 has the highest test error. The reason behind is that ResNet is a deep neural network with many layers, and training it requires a significant amount of computational resources. Momentum SGD is computationally efficient because it only uses the first-order gradient information and has a simple update rule that requires fewer computations than Adam, which uses both first and second-order gradient information. CIFAR10 is also a relatively simple image classification dataset with well-defined classes and consistent patterns. Adam's adaptive learning rate and momentum can help in situations where the gradient is noisy or the loss landscape is complex, but in the case of CIFAR10, momentum SGD may be sufficient for finding a good solution.

	Test error		
	sgd	momentum sgd	Adam
0.1	41.45%	36.62%	49.04%
0.3	39.37%	48.51%	55.84%
0.5	40.22%	58.54%	60.60%

	Training error		
	sgd	momentum sgd	Adam
0.1	44.12%	35.99%	48.58%
0.3	37.19%	54.54%	56.97%
0.5	42.68%	70.17%	56.21%

[1]:
```python
import random
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import argparse
```

[2]:
```python
def parse_args_base(parser):
    parser.add_argument('--n', help='network depth', type=int, default=3)
    parser.add_argument('--batch', help='batch size', type=int, default=128)
    parser.add_argument('--dataset_dir', default='./dataset')
    return parser


def parse_args_train():
    parser = argparse.ArgumentParser()
    parser = parse_args_base(parser)
    parser.add_argument('--checkpoint_dir', default='./dataset')
    parser.add_argument('--print_freq', help='print loss freq', type=int,␣
 ↪default=10)
    parser.add_argument('--save_params_freq', help='parameters saving freq',␣
 ↪type=int, default=1000)
    parser.add_argument('--lr', help='initial learning rate', type=float,␣
 ↪default=0.5)  # change the initial learning rate to 0.3 and 0.5
    parser.add_argument('--momentum', help='optimizer momentum', type=float,␣
 ↪default=0.9)
    parser.add_argument('--weight_decay', help='optimizer weight decay (L2 reg.
 ↪)', type=float, default=0.0001)
    parser.add_argument('--decay_lr_1', help='iteration at which lr decays 1st',␣
 ↪type=int, default=400)
    parser.add_argument('--decay_lr_2', help='iteration at which lr decays 2nd',␣
 ↪type=int, default=800)
    parser.add_argument('--lr_decay_rate', help='lr *= lr_decay_rate at␣
 ↪decay_lr_i-th iteration', type=float, default=0.1)
    parser.add_argument('--n_iter', help='learning iterations', type=int,␣
 ↪default=1000)
    parser.add_argument('-f')
    args = parser.parse_args()

    return args
```

```
def parse_args_test():
    parser = argparse.ArgumentParser()
    parser = parse_args_base(parser)
    parser.add_argument('--params_path', help='path to saved model weights',␣
 ↪default='./dataset/model_final.pth')
    parser.add_argument('-f')
    args = parser.parse_args()

    return args
```

[3]:
```
def preprocess_train(tensor):
    tensor -= tensor.mean().item()
    tensor = F.pad(tensor, pad=(4, 4, 4, 4), mode='constant', value=0)
    t = random.randrange(8)
    l = random.randrange(8)
    tensor = tensor[:, t:t+32, l:l+32]
    if random.random() < 0.5:
        tensor = transforms.functional.hflip(tensor)

    return tensor


def preprocess_test(tensor):
    tensor -= tensor.mean().item()
    return tensor


def get_dataloader(is_train, batch_size, path):
    if is_train:
        return DataLoader(
            datasets.CIFAR10(path,
                             train=True,
                             download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Lambda(preprocess_train)
                             ])),
            batch_size=batch_size,
            shuffle=True
        )
    else:
        return DataLoader(
            datasets.CIFAR10(path,
                             train=False,
                             download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
```

```
                                transforms.Lambda(preprocess_test)
                            ])),
            batch_size=batch_size,
            shuffle=False
        )
```

[4]:
```
def init_weights(net, gain=0.02):
    def init_func(m):
        classname = m.__class__.__name__
        if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.
↪find('Linear') != -1):
            nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
            if hasattr(m, 'bias') and m.bias is not None:
                nn.init.constant_(m.bias.data, 0.0)
        elif classname.find('BatchNorm2d') != -1:
            nn.init.normal_(m.weight.data, 1.0, gain)
            nn.init.constant_(m.bias.data, 0.0)

    net.apply(init_func)


class ResNetModel:
    def __init__(self, opt, train=True):
        self.net = ResNetCifar(opt.n)
        if train:
            self.net.train()
        else:
            self.net.eval()
        if torch.cuda.is_available():
            self.device = torch.device('cuda')
            self.net = self.net.to('cuda')
            self.net = torch.nn.DataParallel(self.net)
        else:
            self.device = torch.device('cpu')

        init_weights(self.net)
        num_params = 0
        for param in self.net.parameters():
            num_params += param.numel()
        print(f'Total number of parameters : {num_params / 1e6:.3f} M')

        if train:
            self.checkpoint_dir = opt.checkpoint_dir
            self.optimizer = optim.SGD(      # replace with SGD , Adam
                self.net.parameters(),
                momentum=opt.momentum,
                lr=opt.lr,
```

```
                weight_decay=opt.weight_decay
                )
            self.scheduler = optim.lr_scheduler.MultiStepLR(
                self.optimizer,
                milestones=[opt.decay_lr_1, opt.decay_lr_2],
                gamma=opt.lr_decay_rate
                )
            self.criterion = nn.CrossEntropyLoss()
            self.loss = 0.0

    def optimize_params(self, x, label):
        x = x.to(self.device)
        label = label.to(self.device)
        y = self._forward(x)
        self._update_params(y, label)

    def _forward(self, x):
        return self.net(x)

    def _backward(self, y, label):
        self.loss = self.criterion(y, label)
        self.loss.backward()

    def _update_params(self, y, label):
        self.optimizer.zero_grad()
        self._backward(y, label)
        self.optimizer.step()
        self.scheduler.step()  # scheduler step in each iteration

    def test(self, x, label):
        with torch.no_grad():
            x = x.to(self.device)
            label = label.to(self.device)
            outputs = self._forward(x)
            _, predicted = torch.max(outputs.data, 1)
            total = label.size(0)
            correct = (predicted == label).sum().item()
            return correct, total, predicted

    def val(self, x, label):
        with torch.no_grad():
            x = x.to(self.device)
            label = label.to(self.device)
            y = self._forward(x)
            return self.criterion(y, label).item()

    def save_model(self, name):
```

```
        path = os.path.join(self.checkpoint_dir, f'model_{name}.pth')
        torch.save(self.net.state_dict(), path)
        print(f'model saved to {path}')

    def load_model(self, path):
        self.net.load_state_dict(torch.load(path))
        print(f'model loaded from {path}')

    def get_current_loss(self):
        return self.loss.item()
```

[5]:
```
class ResNetCifarBlock(nn.Module):
    def __init__(self, input_nc, output_nc):
        super().__init__()
        stride = 1
        self.expand = False
        if input_nc != output_nc:
            assert input_nc * 2 == output_nc, 'output_nc must be input_nc * 2'
            stride = 2
            self.expand = True

        self.conv1 = nn.Conv2d(input_nc, output_nc, kernel_size=3,␣
↪stride=stride, padding=1)
        self.bn1 = nn.BatchNorm2d(output_nc)
        self.conv2 = nn.Conv2d(output_nc, output_nc, kernel_size=3, stride=1,␣
↪padding=1)
        self.bn2 = nn.BatchNorm2d(output_nc)

    def forward(self, x):
        xx = F.relu(self.bn1(self.conv1(x)), inplace=True)
        y = self.bn2(self.conv2(xx))
        if self.expand:
            x = F.interpolate(x, scale_factor=0.5, mode='nearest')  # subsampling
            zero = torch.zeros_like(x)
            x = torch.cat([x, zero], dim=1)  # option A in the original paper
        h = F.relu(y + x, inplace=True)
        return h


def make_resblock_group(cls, input_nc, output_nc, n):
    blocks = []
    blocks.append(cls(input_nc, output_nc))
    for _ in range(1, n):
        blocks.append(cls(output_nc, output_nc))
    return nn.Sequential(*blocks)
```

```
class ResNetCifar(nn.Module):
    def __init__(self, n):
        super().__init__()

        self.conv = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.bn = nn.BatchNorm2d(16)
        self.block1 = make_resblock_group(ResNetCifarBlock, 16, 16, n)
        self.block2 = make_resblock_group(ResNetCifarBlock, 16, 32, n)
        self.block3 = make_resblock_group(ResNetCifarBlock, 32, 64, n)
        self.pool = nn.AdaptiveAvgPool2d(output_size=(1, 1))  # global average␣
↪pooling
        self.fc = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.bn(self.conv(x)), inplace=True)
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.pool(x)
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        return x
```

[6]:
```
def train(opt):
    dataloader = get_dataloader(True, opt.batch, opt.dataset_dir)
    model = ResNetModel(opt, train=True)
    print(opt)
    with open(os.path.join(opt.checkpoint_dir, 'loss_log.txt'), 'a') as f:
        f.write(str(opt) + '\n')

    total_iter = 0
    loss = 0.0

    while True:
        for batch in dataloader:
            total_iter += 1
            inputs, labels = batch
            model.optimize_params(inputs, labels)
            loss += model.get_current_loss()

            if total_iter % opt.print_freq == 0:
                txt = f'iter: {total_iter: 6d}, loss: {loss / opt.print_freq}'
                print(txt)
                txxt= str(loss / opt.print_freq)
                with open(os.path.join(opt.checkpoint_dir, 'loss_log.txt'), 'a')␣
↪as f:
                    f.write(txxt + '\n')
```

```python
                loss = 0.0

            if total_iter % opt.save_params_freq == 0:
                model.save_model(f'{total_iter // opt.save_params_freq}k')

            if total_iter == opt.n_iter:
                model.save_model('final')
                return

if __name__ == '__main__':
    args = parse_args_train()
    train(args)
```

```
Files already downloaded and verified
Total number of parameters : 0.270 M
Namespace(batch=128, checkpoint_dir='./dataset', dataset_dir='./dataset',
decay_lr_1=400, decay_lr_2=800, f='/root/.local/share/jupyter/runtime/kernel-
48e8fd21-6e3d-43b5-a0d7-0557d02feaaa.json', lr=0.5, lr_decay_rate=0.1,
momentum=0.9, n=3, n_iter=1000, print_freq=10, save_params_freq=1000,
weight_decay=0.0001)
iter:     10, loss: 5.260178208351135
iter:     20, loss: 2.386072850227356
iter:     30, loss: 2.317230987548828
iter:     40, loss: 2.3083648681640625
iter:     50, loss: 2.3303656816482543
iter:     60, loss: 2.314922499656677
iter:     70, loss: 2.3095060348510743
iter:     80, loss: 2.3009209394454957
iter:     90, loss: 2.3232123374938967
iter:    100, loss: 2.3156339168548583
iter:    110, loss: 2.3049930334091187
iter:    120, loss: 2.311630296707153
iter:    130, loss: 2.313529062271118
iter:    140, loss: 2.3088917255401613
iter:    150, loss: 2.320504069328308
iter:    160, loss: 2.3122891902923586
iter:    170, loss: 2.3100955963134764
iter:    180, loss: 2.3066614151000975
iter:    190, loss: 2.3113537549972536
iter:    200, loss: 2.312246012687683
iter:    210, loss: 2.309658408164978
iter:    220, loss: 2.303312659263611
iter:    230, loss: 2.301997923851013
iter:    240, loss: 2.2855573177337645
iter:    250, loss: 2.257064461708069
iter:    260, loss: 2.2389535188674925
iter:    270, loss: 2.2505528211593626
```

```
iter:    280, loss: 2.224766397476196
iter:    290, loss: 2.1733654499053956
iter:    300, loss: 2.173841452598572
iter:    310, loss: 2.137011480331421
iter:    320, loss: 2.157132863998413
iter:    330, loss: 2.137747859954834
iter:    340, loss: 2.09818320274353
iter:    350, loss: 2.0846671581268312
iter:    360, loss: 2.0936472058296203
iter:    370, loss: 2.0883981227874755
iter:    380, loss: 2.0420836567878724
iter:    390, loss: 1.9952819108963014
iter:    400, loss: 1.989047122001648
iter:    410, loss: 1.9662629127502442
iter:    420, loss: 1.9113463044166565
iter:    430, loss: 1.9285452604293822
iter:    440, loss: 1.891862964630127
iter:    450, loss: 1.9047509908676148
iter:    460, loss: 1.9072709918022155
iter:    470, loss: 1.9212000250816346
iter:    480, loss: 1.8909221172332764
iter:    490, loss: 1.926684558391571
iter:    500, loss: 1.9137438654899597
iter:    510, loss: 1.9010217308998107
iter:    520, loss: 1.908447551727295
iter:    530, loss: 1.903776264190674
iter:    540, loss: 1.9186978578567504
iter:    550, loss: 1.880186128616333
iter:    560, loss: 1.8685925364494325
iter:    570, loss: 1.8971842050552368
iter:    580, loss: 1.8854424595832824
iter:    590, loss: 1.8786907196044922
iter:    600, loss: 1.8689462304115296
iter:    610, loss: 1.8682750463485718
iter:    620, loss: 1.8887632846832276
iter:    630, loss: 1.8512694716453553
iter:    640, loss: 1.8586389422416687
iter:    650, loss: 1.8563894271850585
iter:    660, loss: 1.8380138993263244
iter:    670, loss: 1.8822349309921265
iter:    680, loss: 1.8472273111343385
iter:    690, loss: 1.8295931100845337
iter:    700, loss: 1.8575900197029114
iter:    710, loss: 1.861980676651001
iter:    720, loss: 1.8861384987831116
iter:    730, loss: 1.83902450799942
iter:    740, loss: 1.8680475950241089
iter:    750, loss: 1.8497129201889038
```

```
iter:    760, loss: 1.8317009091377259
iter:    770, loss: 1.8459845781326294
iter:    780, loss: 1.8697036504745483
iter:    790, loss: 1.8705100774765016
iter:    800, loss: 1.8529508590698243
iter:    810, loss: 1.8222689867019652
iter:    820, loss: 1.8134443640708924
iter:    830, loss: 1.8534798622131348
iter:    840, loss: 1.8102815628051758
iter:    850, loss: 1.8319310426712037
iter:    860, loss: 1.8184168219566346
iter:    870, loss: 1.8312573194503785
iter:    880, loss: 1.7968510746955872
iter:    890, loss: 1.8029914617538452
iter:    900, loss: 1.8617348432540894
iter:    910, loss: 1.7939058899879456
iter:    920, loss: 1.8065063953399658
iter:    930, loss: 1.8441028594970703
iter:    940, loss: 1.8144920110702514
iter:    950, loss: 1.8326905369758606
iter:    960, loss: 1.8100410938262939
iter:    970, loss: 1.8593904733657838
iter:    980, loss: 1.8339136362075805
iter:    990, loss: 1.816961658000946
iter:   1000, loss: 1.8099796295166015
model saved to ./dataset/model_1k.pth
model saved to ./dataset/model_final.pth
```

[7]:
```
def trainerror(opt):
    print(opt)
    dataloader = get_dataloader(True, opt.batch, opt.dataset_dir)
    model = ResNetModel(opt, train=False)
    model.load_model(opt.params_path)

    total_n = 0
    total_correct = 0

    for batch in dataloader:
        inputs, labels = batch
        correct, total, _ = model.test(inputs, labels)
        total_correct += correct
        total_n += total

    acc = 100 * total_correct / total_n
    err = 100 - acc
    print(f'accuracy: {acc:.2f} %')
    print(f'error: {err:.2f} %')
```

```
    print(f'{total_correct} / {total_n}')


if __name__ == '__main__':
    args = parse_args_test()
    trainerror(args)
```

```
Namespace(batch=128, dataset_dir='./dataset', f='/root/.local/share/jupyter/runt
ime/kernel-48e8fd21-6e3d-43b5-a0d7-0557d02feaaa.json', n=3,
params_path='./dataset/model_final.pth')
Files already downloaded and verified
Total number of parameters : 0.270 M
model loaded from ./dataset/model_final.pth
accuracy: 29.83 %
error: 70.17 %
14913 / 50000
```

[8]:

```
def test(opt):
    dataloader = get_dataloader(False, opt.batch, opt.dataset_dir)
    model = ResNetModel(opt, train=True)
    print(opt)
    with open(os.path.join(opt.checkpoint_dir, 'test_loss_log.txt'), 'a') as f:
        f.write(str(opt) + '\n')

    total_iter = 0
    loss = 0.0

    while True:
        for batch in dataloader:
            total_iter += 1
            inputs, labels = batch
            model.optimize_params(inputs, labels)
            loss += model.get_current_loss()

            if total_iter % opt.print_freq == 0:
                txt = f'iter: {total_iter: 6d}, loss: {loss / opt.print_freq}'
                print(txt)
                txxt= str(loss / opt.print_freq)
                with open(os.path.join(opt.checkpoint_dir, 'test_loss_log.txt'),␣
 ↪'a') as f:
                    f.write(txxt + '\n')

                loss = 0.0

            if total_iter % opt.save_params_freq == 0:
                model.save_model(f'{total_iter // opt.save_params_freq}k')
```

```python
            if total_iter == opt.n_iter:
                model.save_model('final')
                return




if __name__ == '__main__':
    args = parse_args_train()
    train(args)
```

```
Files already downloaded and verified
Total number of parameters : 0.270 M
Namespace(batch=128, checkpoint_dir='./dataset', dataset_dir='./dataset',
decay_lr_1=400, decay_lr_2=800, f='/root/.local/share/jupyter/runtime/kernel-
48e8fd21-6e3d-43b5-a0d7-0557d02feaaa.json', lr=0.5, lr_decay_rate=0.1,
momentum=0.9, n=3, n_iter=1000, print_freq=10, save_params_freq=1000,
weight_decay=0.0001)
iter:     10, loss: 5.180495095252991
iter:     20, loss: 2.5708837032318117
iter:     30, loss: 2.3134934663772584
iter:     40, loss: 2.294820785522461
iter:     50, loss: 2.305798149108887
iter:     60, loss: 2.235015892982483
iter:     70, loss: 2.202967810630798
iter:     80, loss: 2.1293555736541747
iter:     90, loss: 2.0789914965629577
iter:    100, loss: 2.070948827266693
iter:    110, loss: 2.0869250297546387
iter:    120, loss: 2.021182823181152
iter:    130, loss: 2.0224040389060973
iter:    140, loss: 2.033132183551788
iter:    150, loss: 2.012133979797363
iter:    160, loss: 1.9620522499084472
iter:    170, loss: 1.9950168967247008
iter:    180, loss: 1.9844149231910706
iter:    190, loss: 1.9728642225265502
iter:    200, loss: 1.9004098892211914
iter:    210, loss: 1.9750344157218933
iter:    220, loss: 1.9065344333648682
iter:    230, loss: 1.9135539293289185
iter:    240, loss: 1.9360596179962157
iter:    250, loss: 1.8860053777694703
iter:    260, loss: 1.8376816630363464
iter:    270, loss: 1.9220985889434814
iter:    280, loss: 1.8785597801208496
iter:    290, loss: 1.8995963215827942
iter:    300, loss: 1.8678790211677552
```

```
iter:    310, loss: 1.8353885531425476
iter:    320, loss: 1.8193363785743712
iter:    330, loss: 1.7869680166244506
iter:    340, loss: 1.8425472140312196
iter:    350, loss: 1.8245766162872314
iter:    360, loss: 1.8276852250099183
iter:    370, loss: 1.7972840666770935
iter:    380, loss: 1.87012220621109
iter:    390, loss: 1.7960012316703797
iter:    400, loss: 1.7840522289276124
iter:    410, loss: 1.8198033213615417
iter:    420, loss: 1.7096255421638489
iter:    430, loss: 1.6872313261032104
iter:    440, loss: 1.7210400342941283
iter:    450, loss: 1.6903883457183837
iter:    460, loss: 1.7191497921943664
iter:    470, loss: 1.705140483379364
iter:    480, loss: 1.6766711950302124
iter:    490, loss: 1.659067404270172
iter:    500, loss: 1.7093275427818297
iter:    510, loss: 1.6726937770843506
iter:    520, loss: 1.6716200590133667
iter:    530, loss: 1.6829920291900635
iter:    540, loss: 1.6174623489379882
iter:    550, loss: 1.676980459690094
iter:    560, loss: 1.7139034628868104
iter:    570, loss: 1.6504444837570191
iter:    580, loss: 1.5975868344306945
iter:    590, loss: 1.6268171072006226
iter:    600, loss: 1.681311798095703
iter:    610, loss: 1.6315538048744203
iter:    620, loss: 1.6422980546951294
iter:    630, loss: 1.621252679824829
iter:    640, loss: 1.6520115852355957
iter:    650, loss: 1.6609533548355102
iter:    660, loss: 1.647756004333496
iter:    670, loss: 1.6445759057998657
iter:    680, loss: 1.6381321549415588
iter:    690, loss: 1.5924819350242614
iter:    700, loss: 1.6437740325927734
iter:    710, loss: 1.59366112947464
iter:    720, loss: 1.6274990439414978
iter:    730, loss: 1.6081182718276978
iter:    740, loss: 1.6314485669136047
iter:    750, loss: 1.5685015320777893
iter:    760, loss: 1.6424452424049378
iter:    770, loss: 1.6488170385360719
iter:    780, loss: 1.641170573234558
```

```
iter:    790, loss: 1.5890581130981445
iter:    800, loss: 1.6045084834098815
iter:    810, loss: 1.6279166221618653
iter:    820, loss: 1.6227516770362853
iter:    830, loss: 1.6034451007843018
iter:    840, loss: 1.590772521495819
iter:    850, loss: 1.6050360321998596
iter:    860, loss: 1.6205818057060242
iter:    870, loss: 1.615891981124878
iter:    880, loss: 1.5819509744644165
iter:    890, loss: 1.5619788646697998
iter:    900, loss: 1.5797269225120545
iter:    910, loss: 1.6152411460876466
iter:    920, loss: 1.596661376953125
iter:    930, loss: 1.582091212272644
iter:    940, loss: 1.5766264200210571
iter:    950, loss: 1.557748556137085
iter:    960, loss: 1.5945794343948365
iter:    970, loss: 1.5818885684013366
iter:    980, loss: 1.5264253616333008
iter:    990, loss: 1.5639222264289856
iter:   1000, loss: 1.5881321787834168
model saved to ./dataset/model_1k.pth
model saved to ./dataset/model_final.pth
```

[9]:

```
def testerror(opt):
    print(opt)
    dataloader = get_dataloader(False, opt.batch, opt.dataset_dir)
    model = ResNetModel(opt, train=False)
    model.load_model(opt.params_path)

    total_n = 0
    total_correct = 0

    for batch in dataloader:
        inputs, labels = batch
        correct, total, _ = model.test(inputs, labels)
        total_correct += correct
        total_n += total

    acc = 100 * total_correct / total_n
    err = 100 - acc
    print(f'accuracy: {acc:.2f} %')
    print(f'error: {err:.2f} %')
    print(f'{total_correct} / {total_n}')
```

```
if __name__ == '__main__':
    args = parse_args_test()
    testerror(args)
```

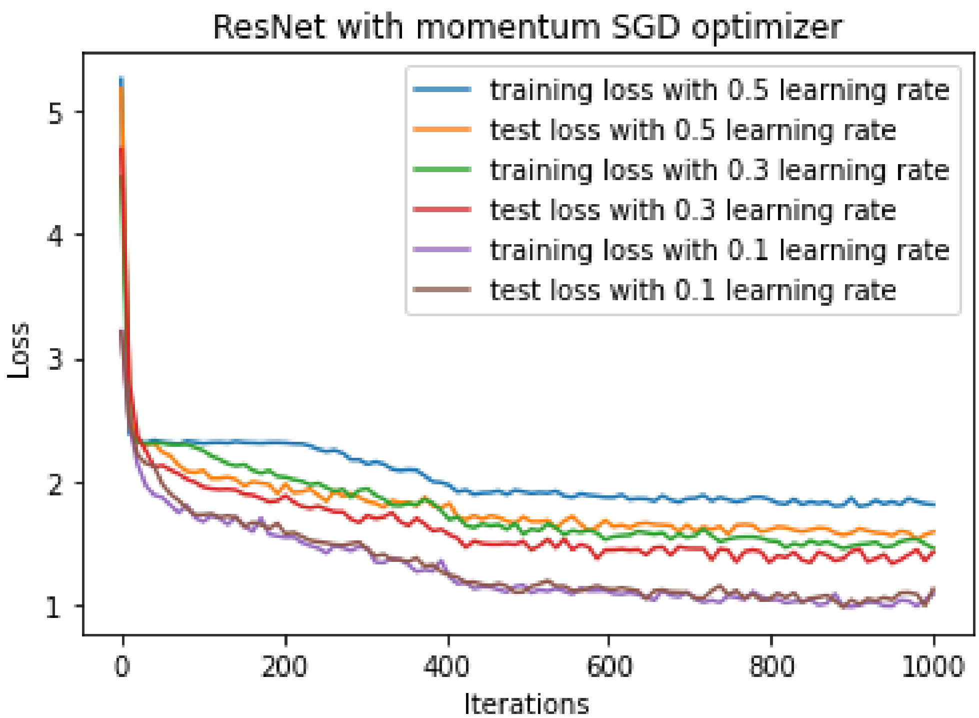
```
Namespace(batch=128, dataset_dir='./dataset', f='/root/.local/share/jupyter/runt
ime/kernel-48e8fd21-6e3d-43b5-a0d7-0557d02feaaa.json', n=3,
params_path='./dataset/model_final.pth')
Files already downloaded and verified
Total number of parameters : 0.270 M
model loaded from ./dataset/model_final.pth
accuracy: 41.46 %
error: 58.54 %
4146 / 10000
```

[10]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
split=100
df=pd.read_csv("./dataset/loss_log.txt", skiprows=[0,split+1], header=None)
np.savetxt(r'./dataset/msgd_5_train.txt', df[:split].values)
np.savetxt(r'./dataset/msgd_5_test.txt', df[split:].values)
```

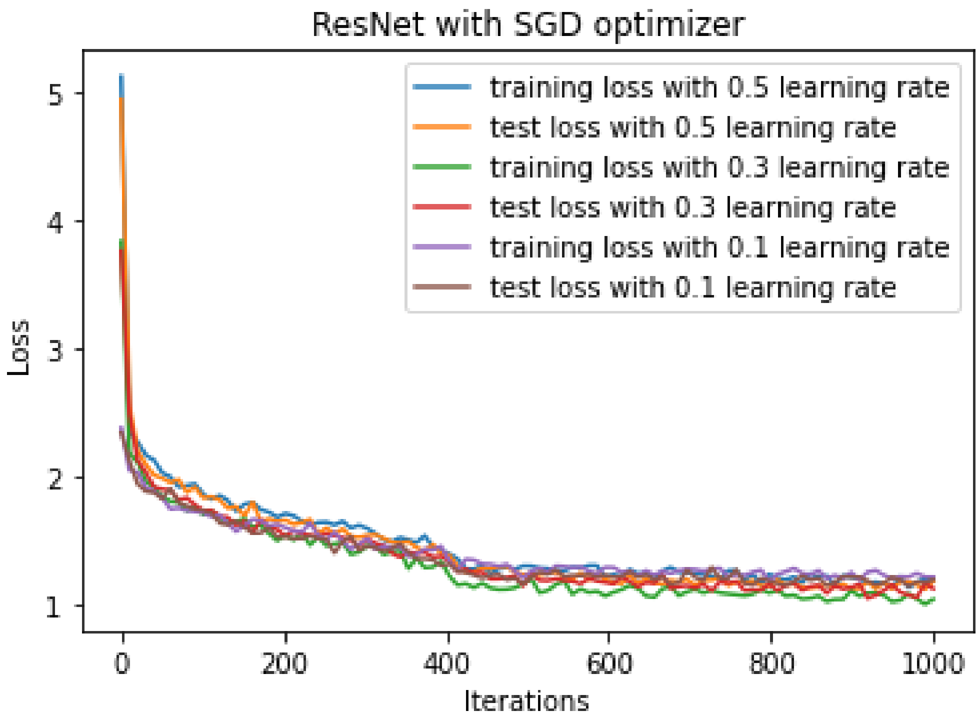
[13]:
```
msgd_5_train=pd.read_csv("./dataset/msgd_5_train.txt", header=None)
msgd_5_test=pd.read_csv("./dataset/msgd_5_test.txt", header=None)
msgd_3_train=pd.read_csv("./dataset/msgd_3_train.txt", header=None)
msgd_3_test=pd.read_csv("./dataset/msgd_3_test.txt", header=None)
msgd_1_train=pd.read_csv("./dataset/msgd_1_train.txt", header=None)
msgd_1_test=pd.read_csv("./dataset/msgd_1_test.txt", header=None)

sgd_5_train=pd.read_csv("./dataset/sgd_5_train.txt", header=None)
sgd_5_test=pd.read_csv("./dataset/sgd_5_test.txt", header=None)
sgd_3_train=pd.read_csv("./dataset/sgd_3_train.txt", header=None)
sgd_3_test=pd.read_csv("./dataset/sgd_3_test.txt", header=None)
sgd_1_train=pd.read_csv("./dataset/sgd_1_train.txt", header=None)
sgd_1_test=pd.read_csv("./dataset/sgd_1_test.txt", header=None)

adam_5_test=pd.read_csv("./dataset/adam_5_test.txt", header=None)
adam_5_train=pd.read_csv("./dataset/adam_5_train.txt", header=None)
adam_3_test=pd.read_csv("./dataset/adam_3_test.txt", header=None)
adam_3_train=pd.read_csv("./dataset/adam_3_train.txt", header=None)
adam_1_test=pd.read_csv("./dataset/adam_1_test.txt", header=None)
adam_1_train=pd.read_csv("./dataset/adam_1_train.txt", header=None)
```

[14]:
```
x=np.linspace(0,1000,split)
plt.plot(x, msgd_5_train, label='training loss with 0.5 learning rate')
plt.plot(x, msgd_5_test, label='test loss with 0.5 learning rate')
plt.plot(x, msgd_3_train, label='training loss with 0.3 learning rate')
plt.plot(x, msgd_3_test, label='test loss with 0.3 learning rate')
```

```
plt.plot(x, msgd_1_train, label='training loss with 0.1 learning rate')
plt.plot(x, msgd_1_test, label='test loss with 0.1 learning rate')
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.legend(loc='upper right')
plt.title("ResNet with momentum SGD optimizer")
plt.show()
```

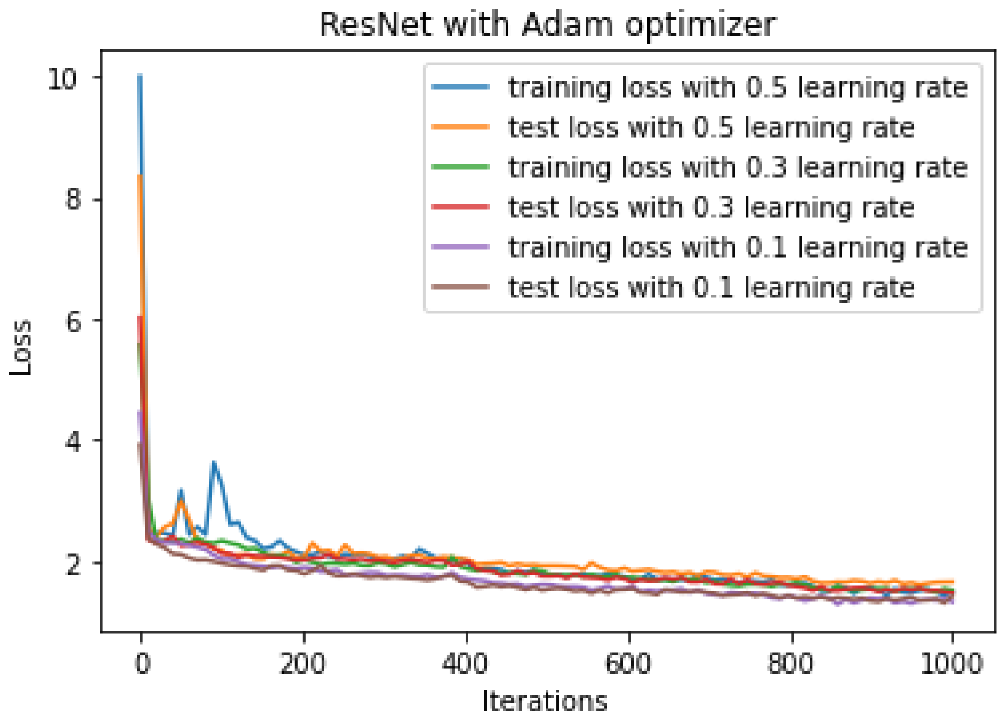


[15]:
```
plt.plot(x, sgd_5_train, label='training loss with 0.5 learning rate')
plt.plot(x, sgd_5_test, label='test loss with 0.5 learning rate')
plt.plot(x, sgd_3_train, label='training loss with 0.3 learning rate')
plt.plot(x, sgd_3_test, label='test loss with 0.3 learning rate')
plt.plot(x, sgd_1_train, label='training loss with 0.1 learning rate')
plt.plot(x, sgd_1_test, label='test loss with 0.1 learning rate')
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.legend(loc='upper right')
plt.title("ResNet with SGD optimizer")
plt.show()
```



[16]:

```
plt.plot(x, adam_5_train, label='training loss with 0.5 learning rate')
plt.plot(x, adam_5_test, label='test loss with 0.5 learning rate')
plt.plot(x, adam_3_train, label='training loss with 0.3 learning rate')
plt.plot(x, adam_3_test, label='test loss with 0.3 learning rate')
plt.plot(x, adam_1_train, label='training loss with 0.1 learning rate')
plt.plot(x, adam_1_test, label='test loss with 0.1 learning rate')
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.legend(loc='upper right')
plt.title("ResNet with Adam optimizer")
plt.show()
```


ResNet with Adam optimizer
training loss with 0.5 learning rate
test loss with 0.5 learning rate
training loss with 0.3 learning rate
test loss with 0.3 learning rate
training loss with 0.1 learning rate
test loss with 0.1 learning rate
Loss
Iterations
10
8
6
4
2
0
200
400
600
800
1000