Image Classification with PyTorch

Data loading

```
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
from torchvision import datasets, transforms
```

```
# torchvision contains convinience functions for popular datasets
ds_train = datasets.MNIST('data', train=True, download=True)
```

```
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
100%|██████████| 9912422/9912422 [00:00<00:00, 23651401.79it/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
100%|██████████| 28881/28881 [00:00<00:00, 132533581.86it/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
100%|██████████| 1648877/1648877 [00:00<00:00, 27966611.79it/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
100%|██████████| 4542/4542 [00:00<00:00, 23260718.89it/s]Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNI
```

Each sample is a 28x28 image

```
# if we index this dataset, we get a single data point: a PIL image and an Integer
print(ds_train[0])
ds_train[0][0].resize((120,120))
```

```
(<PIL.Image.Image image mode=L size=28x28 at 0x7F0F212F4F40>, 5)
```

```
ds_train.train_data.shape
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/datasets/mnist.py:75: UserWarning: train_data has been renamed data
  warnings.warn("train_data has been renamed data")
torch.Size([60000, 28, 28])
```

Let's transform the data to something that our Pytorch models will understand for this purpose, we can supply a transform function to the datase

```
transform = transforms.Compose([
    transforms.ToTensor(),
])
ds_train = datasets.MNIST('data', train=True, download=True, transform=transform)
```

The image is now a `torch.Tensor`

```
type(ds_train[0][0])
```

```
torch.Tensor
```

```
torch.Tensor
```

The normalization is something you learned about in the lecture. Normalizing with $\mu = 0, \sigma = 1$ corresponds to no normalization. Let's compute the proper normalization constants!

```
# lets get only the images
ims_train = ds_train.data
ims_train = ims_train.float() / 255.
```

```
ims_train.shape
```

```
torch.Size([60000, 28, 28])
```

```
#########################################################################
# TODO: calculate the mean and std of MNIST images
# hint: to look for operations on pytorch tensor, refer to the official PyTorch docs
# https://pytorch.org/docs/stable/
#########################################################################
std, mu = torch.std_mean(ims_train, dim=(1,2,0))
print(std.shape, mu.shape)
print(std, mu)
```

```
torch.Size([]) torch.Size([])
tensor(0.3081) tensor(0.1307)
```

```
std = ims_train.std()
mean = ims_train.mean()
std, mean
```

```
(tensor(0.3081), tensor(0.1307))
```

We normalize the data as below.

```
transform = transforms.Compose([
    # transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize(mean=mu, std=std),
])
ds_train = datasets.MNIST('data', train=True, download=True, transform=transform)
ds_test = datasets.MNIST('data', train=False, download=True, transform=transform)
```

```
ds_train[0][0].min(), ds_train[0][0].max(), ds_train.data.float().mean()/255
```

```
(tensor(-0.4241), tensor(2.8215), tensor(0.1307))
```

Note

Something has gone wrong in calculating the mean and std, we should have values centered around 0, and std of 1.

However, we have mean value of 0.1307 with larger std than expected, and values are not centred around 0.

```
print(ds_train[0][0].shape)
plt.imshow(ds_train[0][0][0], cmap="gray")
plt.colorbar()
```

```
torch.Size([1, 28, 28])
<matplotlib.colorbar.Colorbar at 0x7f0e548c32b0>
```

Next, we want to receive mini-batches, not only single data points. We use PyTorch's DataLoader class. Build a dataloader with a batch size of 64 and 4 workers (number of subprocess that peform the dataloading). Important: you need to shuffle the training data, not the test data.

NOTE: if you encounter some unexpected errors in data loading, try setting `NUM_WORKERS = 0`

```
BATCH_SIZE = 64
NUM_WORKERS = 0
#########################################################################
# TODO: Build a dataloader for both train and test data.
#########################################################################
dl_train = DataLoader(ds_train, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS, shuffle=True)
dl_test = DataLoader(ds_test, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS)
```

MLP in Pytorch

Ok, the dataloading works. Let's build our model, PyTorch makes this very easy. We will build replicate the model from our last exercises. However, now, we add another variable called `nLayer` that indicates how many linear layers that in your network. Please adapt your code from last exercise accordingly to allow different number of layers.

```
# These are the parameters to be used
nInput = 784
nOutput = 10
nLayer = 2
nHidden = 16
act_fn = nn.ReLU()
```

```
#########################################################################
# TODO: Implement the __init__ of the MLP class.
# insert the activation after every linear layer. Important: the number of
# hidden layers should be variable!
#########################################################################

class MLP(nn.Module):
    def __init__(self, nInput, nOutput, nLayer, nHidden, act_fn):
        super(MLP, self).__init__()
        layers = []

        ##### implement this part #####
        if nLayer == 1:
            layers.append(nn.Linear(nInput, nOutput))
        else:  # nLayer >= 2
            layers.append(nn.Linear(nInput, nHidden))
            layers.append(act_fn)
            for i in range(nLayer-1):  # Already appended one
                if i+1==nLayer-1:  # if last layer
                    layers.append(nn.Linear(in_features=nHidden, out_features=nOutput))
                    print(f"{i}: Appended last layer")
                    # Don't append activation to last layer
                else:
                    layers.append(nn.Linear(in_features=nHidden, out_features=nHidden))
                    layers.append(act_fn)
                    print(f"{i}: Appended hidden")
        ###############################

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = torch.flatten(x, 1)
        return self.model(x)
```

```
# Let's test if the forward pass works
# this should print torch.Size([1, 10])
t = torch.randn(1,1,28,28)
print(t.size())
mlp = MLP(nInput, nOutput, nLayer, nHidden, act_fn)
mlp(t).shape
```

```
torch.Size([1, 1, 28, 28])
0: Appended last layer
torch.Size([1, 10])
```

```
mlp
```

```
MLP(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=10, bias=True)
  )
)
```

We already implemented the test function for you

```
def test(model, dl_test, device='cpu'):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in dl_test:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(dl_test.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.3f}%)\n'.format(
        test_loss, correct, len(dl_test.dataset),
        100. * correct / len(dl_test.dataset)))
```

Now you only need to implement the training and you are good to go

```
#########################################################################
# TODO: Implement the missing part of the training function. As a loss function we want to use cross entropy
# It can be called with F.cross_entropy().
# Hint: Pass through the model -> Backpropagate gradients -> Take gradient step
#########################################################################

def train(model, dl_train, optimizer, epoch, log_interval=100, device='cpu'):
    model.train()
    model.to(device)
    correct = 0
    for batch_idx, (data, target) in enumerate(dl_train):
        data, target = data.to(device), target.to(device)

        # first we need to zero the gradient, otherwise PyTorch would accumulate them
        optimizer.zero_grad()

        ##### implement this part #####
        # Forward
        output = model(data)

        # Backward pass & updates
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

        ###############################

        # stats
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()

        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(dl_train.dataset),
                100. * batch_idx / len(dl_train), loss.item()))

    print('\nTrain set: Average loss: {:.4f}, Accuracy: {}/{} ({:.1f}%)\n'.format(
        loss, correct, len(dl_train.dataset),
        100. * correct / len(dl_train.dataset)))
```

Ok, the setup is almost done. The onoly missing part is the optimizer. We are going to use Adam.

```
# reinitialize the mlp, so we can play with parameters right here
mlp = MLP(nInput, nOutput, nLayer, nHidden, act_fn)
optimizer = optim.Adam(mlp.parameters())
```

```
0: Appended last layer
```

```
epochs = 10
for epoch in range(1, epochs + 1):
    train(mlp, dl_train, optimizer, epoch, log_interval=100)
    test(mlp, dl_test)

print ('Training is finished.')
```

```
Train Epoch: 7 [44800/60000 (75%)]      Loss: 0.102910
Train Epoch: 7 [51200/60000 (85%)]      Loss: 0.045599
Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.301372

Train set: Average loss: 0.3170, Accuracy: 57315/60000 (95.5%)


Test set: Average loss: 0.1834, Accuracy: 9474/10000 (94.740%)

Train Epoch: 8 [0/60000 (0%)]   Loss: 0.194895
Train Epoch: 8 [6400/60000 (11%)]       Loss: 0.046270
Train Epoch: 8 [12800/60000 (21%)]      Loss: 0.089866
Train Epoch: 8 [19200/60000 (32%)]      Loss: 0.207410
Train Epoch: 8 [25600/60000 (43%)]      Loss: 0.084285
Train Epoch: 8 [32000/60000 (53%)]      Loss: 0.185919
Train Epoch: 8 [38400/60000 (64%)]      Loss: 0.187741
Train Epoch: 8 [44800/60000 (75%)]      Loss: 0.227145
Train Epoch: 8 [51200/60000 (85%)]      Loss: 0.029857
Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.176388

Train set: Average loss: 0.0822, Accuracy: 57412/60000 (95.7%)


Test set: Average loss: 0.1680, Accuracy: 9515/10000 (95.150%)

Train Epoch: 9 [0/60000 (0%)]   Loss: 0.167456
Train Epoch: 9 [6400/60000 (11%)]       Loss: 0.186205
Train Epoch: 9 [12800/60000 (21%)]      Loss: 0.022938
Train Epoch: 9 [19200/60000 (32%)]      Loss: 0.087416
Train Epoch: 9 [25600/60000 (43%)]      Loss: 0.161131
Train Epoch: 9 [32000/60000 (53%)]      Loss: 0.226196
Train Epoch: 9 [38400/60000 (64%)]      Loss: 0.127987
Train Epoch: 9 [44800/60000 (75%)]      Loss: 0.109540
Train Epoch: 9 [51200/60000 (85%)]      Loss: 0.165348
Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.093846

Train set: Average loss: 0.0197, Accuracy: 57508/60000 (95.8%)


Test set: Average loss: 0.1755, Accuracy: 9465/10000 (94.650%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.104708
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.048414
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.046636
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.115201
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.384288
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.168952
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.148816
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.118317
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.094993
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.213566

Train set: Average loss: 0.0574, Accuracy: 57560/60000 (95.9%)


Test set: Average loss: 0.1696, Accuracy: 9499/10000 (94.990%)

Training is finished.
```

After training, you should see test accuracies of > **94%** - By they way, here we report test accuracy, the last exercises reported test error. Accuracy is simply (1 - error). Both metrics are commonly reported, there is no clear preference in literature for one or the other.

Now, can you do some parameter tuning to boost the test accuracy to > **97%**?

```
##########################################################################
#TODO: modify the parameters below to see which setting that you can get to 97%
##########################################################################
nLayer = 3
nHidden = 64
act_fn = nn.ReLU()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# reinitialize the mlp, so we can play with parameters right here
mlp = MLP(nInput, nOutput, nLayer, nHidden, act_fn)
optimizer = optim.Adam(mlp.parameters(), lr=0.0005)
```

```
epochs = 15
for epoch in range(1, epochs + 1):
    train(mlp, dl_train, optimizer, epoch, log_interval=100, device=device)
    test(mlp, dl_test, device=device)

print ('Training is finished.')
```

```
Train Epoch: 12 [44800/60000 (75%)]     Loss: 0.072009
Train Epoch: 12 [51200/60000 (85%)]     Loss: 0.047842
Train Epoch: 12 [57600/60000 (96%)]     Loss: 0.034302

Train set: Average loss: 0.0244, Accuracy: 59430/60000 (99.0%)


Test set: Average loss: 0.0952, Accuracy: 9719/10000 (97.190%)

Train Epoch: 13 [0/60000 (0%)]  Loss: 0.115214
Train Epoch: 13 [6400/60000 (11%)]      Loss: 0.006994
Train Epoch: 13 [12800/60000 (21%)]     Loss: 0.007343
Train Epoch: 13 [19200/60000 (32%)]     Loss: 0.005871
Train Epoch: 13 [25600/60000 (43%)]     Loss: 0.036143
Train Epoch: 13 [32000/60000 (53%)]     Loss: 0.076983
Train Epoch: 13 [38400/60000 (64%)]     Loss: 0.067518
Train Epoch: 13 [44800/60000 (75%)]     Loss: 0.013043
Train Epoch: 13 [51200/60000 (85%)]     Loss: 0.021312
Train Epoch: 13 [57600/60000 (96%)]     Loss: 0.003378

Train set: Average loss: 0.0009, Accuracy: 59471/60000 (99.1%)


Test set: Average loss: 0.0916, Accuracy: 9747/10000 (97.470%)

Train Epoch: 14 [0/60000 (0%)]  Loss: 0.004959
Train Epoch: 14 [6400/60000 (11%)]      Loss: 0.014343
Train Epoch: 14 [12800/60000 (21%)]     Loss: 0.003528
Train Epoch: 14 [19200/60000 (32%)]     Loss: 0.020069
Train Epoch: 14 [25600/60000 (43%)]     Loss: 0.011409
Train Epoch: 14 [32000/60000 (53%)]     Loss: 0.035863
Train Epoch: 14 [38400/60000 (64%)]     Loss: 0.021117
Train Epoch: 14 [44800/60000 (75%)]     Loss: 0.005079
Train Epoch: 14 [51200/60000 (85%)]     Loss: 0.049489
Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.007104

Train set: Average loss: 0.0125, Accuracy: 59562/60000 (99.3%)


Test set: Average loss: 0.0931, Accuracy: 9752/10000 (97.520%)

Train Epoch: 15 [0/60000 (0%)]  Loss: 0.002734
Train Epoch: 15 [6400/60000 (11%)]      Loss: 0.007606
Train Epoch: 15 [12800/60000 (21%)]     Loss: 0.005781
Train Epoch: 15 [19200/60000 (32%)]     Loss: 0.001259
Train Epoch: 15 [25600/60000 (43%)]     Loss: 0.017580
Train Epoch: 15 [32000/60000 (53%)]     Loss: 0.019352
Train Epoch: 15 [38400/60000 (64%)]     Loss: 0.011877
Train Epoch: 15 [44800/60000 (75%)]     Loss: 0.011792
Train Epoch: 15 [51200/60000 (85%)]     Loss: 0.008487
Train Epoch: 15 [57600/60000 (96%)]     Loss: 0.004290

Train set: Average loss: 0.0044, Accuracy: 59608/60000 (99.3%)


Test set: Average loss: 0.1035, Accuracy: 9723/10000 (97.230%)

Training is finished.
```

- Made network deeper and wider to increase representational power, and lowered learning rate to stabilise learning

Before you move on to the next exercise, you can further play with the other parameters (learning rate, epochs, a different optimizer, etc.) to get a feeling what can improve or hamper performance.

CNN

Alright, we matched our prior performance. Let's surpass it! You will soon see the power of CNN by building a small one yourself. The structure should be as follows

CNN Architecture
Conv: $C_{in} = 1, C_{out} = 32, K = 3, S = 1, P = 0$
ReLU

CNN Architecture
Conv: $C_{in} = 32, C_{out} = 64, K = 3, S = 1, P = 0$
ReLU
MaxPool2d: $K = 2, S = 2, P = 0$
Dropout: $p = 0.25$
Linear: $C_{in} = 9216, C_{out} = 128$
ReLU
Dropout: $p = 0.5$
Linear: $C_{in} = 128, C_{out} = 10$

The layers you will need are:

`nn.Conv2d, nn.Linear, nn.Dropout, nn.MaxPool2d, nn.Flatten`

For layers without parameters you can alternatively use function in the forward pass:

`F.max_pool2d, torch.flatten`

```
#########################################################################
# TODO: Implement the __init__ and forward method of the CNN class.
# Hint: do not forget to flatten the appropriate dimension after the convolutional blocks.
# A linear layers expect input of the size (B, H) with batch size B and feature size H
#########################################################################

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=(3,3), stride=(1,1), padding=(0,0)),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(0,0)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2,2), stride=(2,2), padding=(0,0)),
            nn.Flatten(),
            nn.Dropout(p=0.25),  # Dropout after flattening
            nn.Linear(in_features=9216, out_features=128),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(in_features=128, out_features=10)
        )

    def forward(self, x):
        # print(f"x.shape: {x.shape}")
        return self.model(x)
```

```
# Let's test if the forward pass works
# this should print torch.Size([1, 10])
t = torch.randn(1,1,28,28)
cnn = CNN()
cnn(t).shape
```

```
torch.Size([1, 10])
```

```
cnn
```

```
CNN(
  (model): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=(0, 0), dilation=1, ceil_mode=False)
    (5): Flatten(start_dim=1, end_dim=-1)
    (6): Dropout(p=0.25, inplace=False)
    (7): Linear(in_features=9216, out_features=128, bias=True)
    (8): ReLU()
    (9): Dropout(p=0.5, inplace=False)
    (10): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

Alright, let's train!

```
optimizer = optim.Adam(cnn.parameters())
```

```
epochs = 5
for epoch in range(1, epochs + 1):
    train(cnn, dl_train, optimizer, epoch, log_interval=100, device=device)
    test(cnn, dl_test, device=device)
```

```
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.026496
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.167773
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.142578
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.125159

Train set: Average loss: 0.2602, Accuracy: 58425/60000 (97.4%)


Test set: Average loss: 0.0402, Accuracy: 9868/10000 (98.680%)

Train Epoch: 3 [0/60000 (0%)]   Loss: 0.049732
Train Epoch: 3 [6400/60000 (11%)]       Loss: 0.143663
Train Epoch: 3 [12800/60000 (21%)]      Loss: 0.023654
Train Epoch: 3 [19200/60000 (32%)]      Loss: 0.019535
Train Epoch: 3 [25600/60000 (43%)]      Loss: 0.027258
Train Epoch: 3 [32000/60000 (53%)]      Loss: 0.027080
Train Epoch: 3 [38400/60000 (64%)]      Loss: 0.025758
Train Epoch: 3 [44800/60000 (75%)]      Loss: 0.096416
Train Epoch: 3 [51200/60000 (85%)]      Loss: 0.013278
Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.037945

Train set: Average loss: 0.0079, Accuracy: 58828/60000 (98.0%)


Test set: Average loss: 0.0319, Accuracy: 9902/10000 (99.020%)

Train Epoch: 4 [0/60000 (0%)]   Loss: 0.031041
Train Epoch: 4 [6400/60000 (11%)]       Loss: 0.053597
Train Epoch: 4 [12800/60000 (21%)]      Loss: 0.192419
Train Epoch: 4 [19200/60000 (32%)]      Loss: 0.007639
Train Epoch: 4 [25600/60000 (43%)]      Loss: 0.210701
Train Epoch: 4 [32000/60000 (53%)]      Loss: 0.036232
Train Epoch: 4 [38400/60000 (64%)]      Loss: 0.059894
Train Epoch: 4 [44800/60000 (75%)]      Loss: 0.043570
Train Epoch: 4 [51200/60000 (85%)]      Loss: 0.008980
Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.027518

Train set: Average loss: 0.0054, Accuracy: 58962/60000 (98.3%)


Test set: Average loss: 0.0365, Accuracy: 9889/10000 (98.890%)

Train Epoch: 5 [0/60000 (0%)]   Loss: 0.017895
Train Epoch: 5 [6400/60000 (11%)]       Loss: 0.017580
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.016567
Train Epoch: 5 [19200/60000 (32%)]      Loss: 0.021148
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.013810
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.064169
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.044386
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.026764
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.055242
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.017173

Train set: Average loss: 0.0022, Accuracy: 59145/60000 (98.6%)


Test set: Average loss: 0.0318, Accuracy: 9898/10000 (98.980%)
```

This will probably take a bit longer to train, as a convolutional network is not very efficient on a CPU. The current settings should get you around **99%** accuracy. Nice! Again, you should try different hyperparameters and see how far you can push the performance.

Inline Question

If your model weight is randomly initalized, and no training is done as above. What accuracy do you think the model will get for a 10-class classification task in theory?

Your answer: The model should output a random class from 0-9 (there will be a random largest logit in the output layer) $\implies \frac{1}{10}$ chance to randomly be correct, i.e. 10% accuracy.

Training on CIFAR10

Now we are going to move to something more challenging - CIFAR10. We can reuse most of the code above. Thankfully, CIFAR is also a popular dataset, so we can again make use of a PyTorch convience function.

```
ds_train = datasets.CIFAR10(root='./data', train=True, download=True)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:01<00:00, 91924544.26it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
```

This dataset is not normalized yet, so we need to calculate the normalization constants.

```
ims_train = torch.tensor(ds_train.data)
ims_train = ims_train.float() / 255.
```

```
ims_train.std((0,1,2)), ims_train.shape
```

```
(tensor([0.2470, 0.2435, 0.2616]), torch.Size([50000, 32, 32, 3]))
```

```
#########################################################################
# TODO: calculate the mean and std of CIFAR
# hint: We want the mean and std of the channel dimension, these should
# be 3 dimensional
#########################################################################
mu = torch.mean(ims_train, dim=(0,1,2))
std = ims_train.std((0,1,2))
mu, std
```

```
(tensor([0.4914, 0.4822, 0.4465]), tensor([0.2470, 0.2435, 0.2616]))
```

```
torch.mean(ims_train, dim=(0,1,2))
```

For CIFAR we want to make use of data augmentation to improve generalization. You will find all data augmentations data are included in torchvision here:

https://pytorch.org/docs/stable/torchvision/transforms.html

```
BATCH_SIZE = 128
NUM_WORKERS = 4 # if you encounter some unexpected errors in data loading, try setting `NUM_WORKERS = 0`
#########################################################################
# TODO: Implement the proper transforms for the training and test dataloaders.
# Then build train and test dataloaders with batch size 128 and 4 workers
#
# Train:
# - Apply a random crop with size 32 on a padded version of the image with P=4
# - Flip the image horizontally with a probability of 40 %
# - Transform to a Tensor
# - Normalize with the constants calculated above
# Test:
# - Transform to a Tensor
# - Normalize with the constants calculated above
#########################################################################
transform_train = transforms.Compose([
    transforms.RandomCrop(size=32, padding=4),
    transforms.RandomHorizontalFlip(p=0.4),
    transforms.ToTensor(),
    transforms.Normalize(mean=mu, std=std)
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mu, std=std)
])

ds_train = datasets.CIFAR10('./data', train=True, download=True, transform=transform_train)
ds_test = datasets.CIFAR10('./data', train=False, download=True, transform=transform_test)

dl_train = DataLoader(ds_train, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS, shuffle=True)
dl_test = DataLoader(ds_test, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS)
```

```
Files already downloaded and verified
Files already downloaded and verified
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 w
  warnings.warn(_create_warning_msg(
```

Setting up the optimizer, this time we use SGD. The scheduler adapts the learning rate during traing (you can ignore it)

```
cnn = CNN()
optimizer = optim.SGD(cnn.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
```

```
epochs = 5
for epoch in range(1, epochs + 1):
    train(cnn, dl_train, optimizer, epoch, log_interval=100)
    test(cnn, dl_test)
    scheduler.step()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-65-ba9429f4ef5e> in <cell line: 2>()
      1 epochs = 5
      2 for epoch in range(1, epochs + 1):
----> 3     train(cnn, dl_train, optimizer, epoch, log_interval=100)
      4     test(cnn, dl_test)
      5     scheduler.step()

10 frames

/usr/local/lib/python3.10/dist-packages/torch/nn/modules/conv.py in _conv_forward(self, input, weight, bias)
    454                             weight, bias, self.stride,
    455                             _pair(0), self.dilation, self.groups)
--> 456         return F.conv2d(input, weight, bias, self.stride,
    457                         self.padding, self.dilation, self.groups)
    458

RuntimeError: Given groups=1, weight of size [32, 1, 3, 3], expected input[128, 3, 32, 32] to have 1 channels, but got 3
```

SEARCH STACK OVERFLOW

This will not work. You should see the following error message

```
Given groups=1, weight of size [32, 1, 3, 3], expected input[128, 3, 32, 32] to have 1 channels, but got 3 channels instead
```

This error is telling us that something is not right in the definition of our model. Copy the CNN class from above and make changes, so the training works.

```
#########################################################################
# TODO: Adapt the definition from the CNN class above to work on CIFAR.
# You can copy and run the following prompt for evaluation:
# CNN()(torch.randn(1,3,32,32)).shape
# It should print 'torch.Size([1, 10])'
# Hint: You need to change 2 things.
#########################################################################

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3,3), stride=(1,1), padding=(0,0)),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(0,0)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2,2), stride=(2,2), padding=(0,0)),
            nn.Flatten(),
            nn.Dropout(p=0.25),  # Dropout after flattening
            nn.Linear(in_features=12544, out_features=128),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(in_features=128, out_features=10)
        )

    def forward(self, x):
        # print(f"x.shape: {x.shape}")
        return self.model(x)
```

Let's try again

```
cnn = CNN()
optimizer = optim.SGD(cnn.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)

epochs = 10
for epoch in range(1, epochs + 1):
    train(cnn, dl_train, optimizer, epoch, log_interval=100, device=device)
```

```
    test(cnn, dl_test, device=device)
    scheduler.step()
```

```
Train Epoch: 5 [38400/50000 (77%)]      Loss: 1.693543

Train set: Average loss: 1.6703, Accuracy: 19652/50000 (39.3%)


Test set: Average loss: 1.4070, Accuracy: 4863/10000 (48.630%)

Train Epoch: 6 [0/50000 (0%)]   Loss: 1.481947
Train Epoch: 6 [12800/50000 (26%)]      Loss: 1.656423
Train Epoch: 6 [25600/50000 (51%)]      Loss: 1.522030
Train Epoch: 6 [38400/50000 (77%)]      Loss: 1.629586

Train set: Average loss: 1.5950, Accuracy: 20472/50000 (40.9%)


Test set: Average loss: 1.4753, Accuracy: 4665/10000 (46.650%)

Train Epoch: 7 [0/50000 (0%)]   Loss: 1.886494
Train Epoch: 7 [12800/50000 (26%)]      Loss: 1.770185
Train Epoch: 7 [25600/50000 (51%)]      Loss: 1.620141
Train Epoch: 7 [38400/50000 (77%)]      Loss: 1.504474

Train set: Average loss: 1.6257, Accuracy: 20556/50000 (41.1%)


Test set: Average loss: 1.3518, Accuracy: 5239/10000 (52.390%)

Train Epoch: 8 [0/50000 (0%)]   Loss: 1.711290
Train Epoch: 8 [12800/50000 (26%)]      Loss: 1.641759
Train Epoch: 8 [25600/50000 (51%)]      Loss: 1.675023
Train Epoch: 8 [38400/50000 (77%)]      Loss: 1.501010

Train set: Average loss: 1.7511, Accuracy: 21232/50000 (42.5%)


Test set: Average loss: 1.3499, Accuracy: 5129/10000 (51.290%)

Train Epoch: 9 [0/50000 (0%)]   Loss: 1.482178
Train Epoch: 9 [12800/50000 (26%)]      Loss: 1.503869
Train Epoch: 9 [25600/50000 (51%)]      Loss: 1.585268
Train Epoch: 9 [38400/50000 (77%)]      Loss: 1.637139

Train set: Average loss: 1.6004, Accuracy: 21391/50000 (42.8%)


Test set: Average loss: 1.4196, Accuracy: 4851/10000 (48.510%)

Train Epoch: 10 [0/50000 (0%)]  Loss: 1.649418
Train Epoch: 10 [12800/50000 (26%)]     Loss: 1.578653
Train Epoch: 10 [25600/50000 (51%)]     Loss: 1.628335
Train Epoch: 10 [38400/50000 (77%)]     Loss: 1.434548

Train set: Average loss: 1.6352, Accuracy: 21053/50000 (42.1%)


Test set: Average loss: 1.3469, Accuracy: 5107/10000 (51.070%)
```

This should give 40 - 50 % - and if you are not already on Colab it will give you a stressed out laptop. The performance is a lot better than random, but we can definitely do better.

Have fun with GPUs

You can already call it a day until this point because we won't grade the rest of the excecise. You can have more fun with the rest :)

If you didn't already, move to colab. To use a GPU, follow on the collaboratory menu tabs, "Runtime" => "Change runtime type" and set it to GPU. Then run the same training loop but now on GPU.

It as easy as:

```
device = 'cuda'
if device == 'cuda': torch.backends.cudnn.benchmark = True # additional speed up

cnn = CNN()
optimizer = optim.SGD(cnn.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
cnn = cnn.to(device)

epochs = 10
```

```
for epoch in range(1, epochs + 1):
    train(cnn, dl_train, optimizer, epoch, log_interval=100, device=device)
    test(cnn, dl_test, device=device)
    scheduler.step()
```

This should be way faster now. But the true advantage of the GPU is that we can use much bigger models now and still train them in a reasonable amount of time. PyTorch is again very handy. The torchvision library comes with varies state-of-the-art model architectures, some of which you have seen in the lecture.

```
from torchvision.models import resnet18
```

```
cnn = resnet18()
print(cnn)
```

Looks scary! But the only thing you need to change to make it work on CIFAR is the last layer. Currently the last layer is:

```
  (fc): Linear(in_features=512, out_features=1000, bias=True)
```

out_features is the number of classes. This models are developed for Imagenet, a dataset with 1000 classes. So this part of the model you need to adapt. Additionally, you need to add a log-softmax layer again, as we us negative log-likelihood as the training criterion.

```
#########################################################################
# TODO: Adapt the Resnet to work on CIFAR
#########################################################################
```

```
# This should print 'torch.Size([16, 10])'
cnn(torch.randn(16,3,32,32)).shape
```

```
device = 'cuda'
if device == 'cuda': torch.backends.cudnn.benchmark = True # this gives us additional speed up

optimizer = optim.SGD(cnn.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
cnn = cnn.to(device)

epochs = 50
for epoch in range(1, epochs + 1):
    train(cnn, dl_train, optimizer, epoch, log_interval=100, device=device)
    test(cnn, dl_test, device=device)
    scheduler.step()
```

This should get us well above 75%, the best we got was ~ 80%.

Now, use different torchvision architectures, different optimizers (Adam is always a good choice), data augmentation techniques, and hyperparameter search to achieve a test accuracy of >90 %