homework3

April 13, 2025

1 Homework 3

Author: Atharva Pandhare

1.1 Imports and stuff

[1]:
```
# Load in relevant libraries, and alias where appropriate
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import time

# Define relevant variables for the ML task
batch_size = 64
learning_rate = 0.01
num_epochs = 20

# Device will determine whether to run the training on GPU or CPU.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[2]:
```
# For training data
cifar_trainset = datasets.CIFAR10(root='./data', train=True, download=True,␣
↪transform=transforms.ToTensor())

# For test data
cifar_testset = datasets.CIFAR10(root='./data', train=False, download=True,␣
↪transform=transforms.ToTensor())
```

1.2 Architectures

1.2.1 LeNet Architecture

name	output size
Input	32x32x3
conv(kernel = 5, output channels = 6)	28x28x6

name	output size
MaxPool(window = 2)	16x16x6
conv(kernel = 5, output channels = 16)	12x12x16
MaxPool(window = 2)	6x6x16
linear	120
linear	84
linear	10

[3]:
```python
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.net = nn.Sequential(
            nn.LazyConv2d(out_channels=6, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.LazyConv2d(out_channels=16, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Flatten(),

            nn.LazyLinear(out_features=120),
            nn.ReLU(),

            nn.LazyLinear(out_features=84),
            nn.ReLU(),

            nn.LazyLinear(out_features=10)
        )

    def forward(self, x):
        output = self.net(x)

        return output
```

1.2.2 LeNet Architecture with Dropout

name	output size
Input	32x32x3
conv(kernel = 5, output channels = 6)	28x28x6
MaxPool(window = 2)	16x16x6
conv(kernel = 5, output channels = 16)	12x12x16
MaxPool(window = 2)	6x6x16
linear	120

name	output size
Dropout	120
linear	84
linear	10

[4]:
```
class LeNet_v1(nn.Module):
    def __init__(self, conv_dropout_rate = 0.2, fc_dropout_rate=0.5):
        super(LeNet_v1, self).__init__()
        self.net = nn.Sequential(
            nn.LazyConv2d(out_channels=6, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.LazyConv2d(out_channels=16, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Flatten(),

            nn.LazyLinear(out_features=120),
            nn.ReLU(),

            nn.Dropout(fc_dropout_rate),

            nn.LazyLinear(out_features=84),
            nn.ReLU(),

            nn.LazyLinear(out_features=10)
        )

    def forward(self, x):
        output = self.net(x)
        return output
```

1.2.3 LeNet Architecture with Dropout and Batch Normalization

name	output size
Input	32x32x3
conv(kernel = 5, output channels = 6)	28x28x6
Batch Normalization	28x28x6
MaxPool(window = 2)	16x16x6
conv(kernel = 5, output channels = 16)	12x12x16
MaxPool(window = 2)	6x6x16
linear	120
Dropout	120

name	output size
linear	84
linear	10

[5]:
```
class LeNet_v2(nn.Module):
    def __init__(self, fc_dropout_rate=0.5):
        super(LeNet_v2, self).__init__()
        self.net = nn.Sequential(
            nn.LazyConv2d(out_channels=6, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.LazyConv2d(out_channels=16, kernel_size=5),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Flatten(),

            nn.LazyLinear(out_features=120),
            nn.ReLU(),

            nn.Dropout(fc_dropout_rate),

            nn.LazyLinear(out_features=84),
            nn.ReLU(),

            nn.LazyLinear(out_features=10)
        )

    def forward(self, x):
        output = self.net(x)
        return output
```

1.3 Train and Test Functions

[6]:
```
def train(model, trainloader, num_epochs = num_epochs):
    start_time = time.time()
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,␣
 ↪momentum=0.9)

    # Training loop
    for epoch in range(num_epochs):
        # Set model to training mode
        model.train()
```

```
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(trainloader):
            # Move data to device (CPU/GPU)
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimize
            loss.backward()
            optimizer.step()

            # Print statistics
            running_loss += loss.item()
            if i % 100 == 99:    # Print every 100 mini-batches
                print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}')
                running_loss = 0.0
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Training completed in {elapsed_time:.2f} seconds")

def test(model, testloader):
    start_time = time.time()
    # Testing the best model on test data
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f'testing finished in {elapsed_time:.2f} seconds, Accuracy: {accuracy:
 ↪.2f}%')
```

1.4 Training and Testing

1.4.1 Regular

[7]:
```
model = LeNet().to(device)
# DataLoader for training and test datasets
trainloader = torch.utils.data.DataLoader(cifar_trainset,␣
↪batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(cifar_testset, batch_size=batch_size,␣
↪shuffle=False)

train(model, trainloader)
```

```
[1, 100] loss: 2.304
[1, 200] loss: 2.300
[1, 300] loss: 2.284
[1, 400] loss: 2.196
[1, 500] loss: 2.106
[1, 600] loss: 2.075
[1, 700] loss: 2.014
[2, 100] loss: 1.891
[2, 200] loss: 1.880
[2, 300] loss: 1.792
[2, 400] loss: 1.738
[2, 500] loss: 1.696
[2, 600] loss: 1.701
[2, 700] loss: 1.647
[3, 100] loss: 1.567
[3, 200] loss: 1.567
[3, 300] loss: 1.560
[3, 400] loss: 1.544
[3, 500] loss: 1.501
[3, 600] loss: 1.472
[3, 700] loss: 1.479
[4, 100] loss: 1.431
[4, 200] loss: 1.426
[4, 300] loss: 1.387
[4, 400] loss: 1.404
[4, 500] loss: 1.389
[4, 600] loss: 1.402
[4, 700] loss: 1.342
[5, 100] loss: 1.343
[5, 200] loss: 1.288
[5, 300] loss: 1.308
[5, 400] loss: 1.304
[5, 500] loss: 1.296
[5, 600] loss: 1.284
[5, 700] loss: 1.254
```

```
[6, 100] loss: 1.252
[6, 200] loss: 1.223
[6, 300] loss: 1.227
[6, 400] loss: 1.207
[6, 500] loss: 1.200
[6, 600] loss: 1.234
[6, 700] loss: 1.241
[7, 100] loss: 1.151
[7, 200] loss: 1.174
[7, 300] loss: 1.160
[7, 400] loss: 1.154
[7, 500] loss: 1.147
[7, 600] loss: 1.184
[7, 700] loss: 1.181
[8, 100] loss: 1.107
[8, 200] loss: 1.073
[8, 300] loss: 1.127
[8, 400] loss: 1.101
[8, 500] loss: 1.126
[8, 600] loss: 1.140
[8, 700] loss: 1.104
[9, 100] loss: 1.064
[9, 200] loss: 1.055
[9, 300] loss: 1.072
[9, 400] loss: 1.085
[9, 500] loss: 1.058
[9, 600] loss: 1.058
[9, 700] loss: 1.102
[10, 100] loss: 1.028
[10, 200] loss: 1.018
[10, 300] loss: 1.020
[10, 400] loss: 1.002
[10, 500] loss: 1.028
[10, 600] loss: 1.036
[10, 700] loss: 1.022
[11, 100] loss: 0.955
[11, 200] loss: 0.980
[11, 300] loss: 0.961
[11, 400] loss: 0.975
[11, 500] loss: 0.999
[11, 600] loss: 0.998
[11, 700] loss: 1.020
[12, 100] loss: 0.902
[12, 200] loss: 0.939
[12, 300] loss: 0.956
[12, 400] loss: 0.977
[12, 500] loss: 0.962
[12, 600] loss: 0.976
```

```
[12, 700] loss: 0.963
[13, 100] loss: 0.893
[13, 200] loss: 0.903
[13, 300] loss: 0.907
[13, 400] loss: 0.933
[13, 500] loss: 0.938
[13, 600] loss: 0.980
[13, 700] loss: 0.932
[14, 100] loss: 0.858
[14, 200] loss: 0.850
[14, 300] loss: 0.855
[14, 400] loss: 0.894
[14, 500] loss: 0.898
[14, 600] loss: 0.916
[14, 700] loss: 0.916
[15, 100] loss: 0.816
[15, 200] loss: 0.837
[15, 300] loss: 0.853
[15, 400] loss: 0.850
[15, 500] loss: 0.871
[15, 600] loss: 0.895
[15, 700] loss: 0.880
[16, 100] loss: 0.811
[16, 200] loss: 0.815
[16, 300] loss: 0.837
[16, 400] loss: 0.864
[16, 500] loss: 0.834
[16, 600] loss: 0.861
[16, 700] loss: 0.866
[17, 100] loss: 0.750
[17, 200] loss: 0.780
[17, 300] loss: 0.810
[17, 400] loss: 0.811
[17, 500] loss: 0.831
[17, 600] loss: 0.872
[17, 700] loss: 0.831
[18, 100] loss: 0.745
[18, 200] loss: 0.749
[18, 300] loss: 0.773
[18, 400] loss: 0.823
[18, 500] loss: 0.803
[18, 600] loss: 0.854
[18, 700] loss: 0.831
[19, 100] loss: 0.713
[19, 200] loss: 0.740
[19, 300] loss: 0.741
[19, 400] loss: 0.768
[19, 500] loss: 0.784
```

```
[19, 600] loss: 0.809
[19, 700] loss: 0.833
[20, 100] loss: 0.695
[20, 200] loss: 0.716
[20, 300] loss: 0.721
[20, 400] loss: 0.765
[20, 500] loss: 0.752
[20, 600] loss: 0.789
[20, 700] loss: 0.793
Training completed in 66.10 seconds
```

[8]:
```
test(model, testloader)
```

```
testing finished in 0.51 seconds, Accuracy: 57.36%
```

1.4.2 With Dropout

[9]:
```
model1 = LeNet_v1().to(device)
# DataLoader for training and test datasets
trainloader = torch.utils.data.DataLoader(cifar_trainset,
 ↪batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(cifar_testset, batch_size=batch_size,
 ↪shuffle=False)

train(model1, trainloader)
```

```
[1, 100] loss: 2.304
[1, 200] loss: 2.300
[1, 300] loss: 2.268
[1, 400] loss: 2.134
[1, 500] loss: 2.056
[1, 600] loss: 1.990
[1, 700] loss: 1.939
[2, 100] loss: 1.819
[2, 200] loss: 1.743
[2, 300] loss: 1.727
[2, 400] loss: 1.718
[2, 500] loss: 1.698
[2, 600] loss: 1.642
[2, 700] loss: 1.643
[3, 100] loss: 1.597
[3, 200] loss: 1.569
[3, 300] loss: 1.547
[3, 400] loss: 1.554
[3, 500] loss: 1.586
[3, 600] loss: 1.551
[3, 700] loss: 1.544
[4, 100] loss: 1.513
```

```
[4, 200] loss: 1.482
[4, 300] loss: 1.473
[4, 400] loss: 1.465
[4, 500] loss: 1.463
[4, 600] loss: 1.453
[4, 700] loss: 1.472
[5, 100] loss: 1.410
[5, 200] loss: 1.398
[5, 300] loss: 1.430
[5, 400] loss: 1.420
[5, 500] loss: 1.414
[5, 600] loss: 1.383
[5, 700] loss: 1.392
[6, 100] loss: 1.374
[6, 200] loss: 1.371
[6, 300] loss: 1.359
[6, 400] loss: 1.351
[6, 500] loss: 1.369
[6, 600] loss: 1.364
[6, 700] loss: 1.371
[7, 100] loss: 1.339
[7, 200] loss: 1.319
[7, 300] loss: 1.294
[7, 400] loss: 1.319
[7, 500] loss: 1.312
[7, 600] loss: 1.315
[7, 700] loss: 1.299
[8, 100] loss: 1.285
[8, 200] loss: 1.270
[8, 300] loss: 1.303
[8, 400] loss: 1.286
[8, 500] loss: 1.293
[8, 600] loss: 1.289
[8, 700] loss: 1.296
[9, 100] loss: 1.266
[9, 200] loss: 1.278
[9, 300] loss: 1.270
[9, 400] loss: 1.264
[9, 500] loss: 1.275
[9, 600] loss: 1.233
[9, 700] loss: 1.269
[10, 100] loss: 1.234
[10, 200] loss: 1.222
[10, 300] loss: 1.222
[10, 400] loss: 1.252
[10, 500] loss: 1.246
[10, 600] loss: 1.227
[10, 700] loss: 1.242
```

```
[11, 100] loss: 1.207
[11, 200] loss: 1.211
[11, 300] loss: 1.210
[11, 400] loss: 1.225
[11, 500] loss: 1.241
[11, 600] loss: 1.228
[11, 700] loss: 1.223
[12, 100] loss: 1.184
[12, 200] loss: 1.222
[12, 300] loss: 1.183
[12, 400] loss: 1.210
[12, 500] loss: 1.210
[12, 600] loss: 1.198
[12, 700] loss: 1.197
[13, 100] loss: 1.165
[13, 200] loss: 1.178
[13, 300] loss: 1.227
[13, 400] loss: 1.197
[13, 500] loss: 1.199
[13, 600] loss: 1.191
[13, 700] loss: 1.175
[14, 100] loss: 1.171
[14, 200] loss: 1.172
[14, 300] loss: 1.181
[14, 400] loss: 1.181
[14, 500] loss: 1.173
[14, 600] loss: 1.173
[14, 700] loss: 1.178
[15, 100] loss: 1.168
[15, 200] loss: 1.166
[15, 300] loss: 1.139
[15, 400] loss: 1.179
[15, 500] loss: 1.168
[15, 600] loss: 1.162
[15, 700] loss: 1.162
[16, 100] loss: 1.144
[16, 200] loss: 1.135
[16, 300] loss: 1.147
[16, 400] loss: 1.136
[16, 500] loss: 1.164
[16, 600] loss: 1.152
[16, 700] loss: 1.158
[17, 100] loss: 1.129
[17, 200] loss: 1.125
[17, 300] loss: 1.140
[17, 400] loss: 1.138
[17, 500] loss: 1.158
[17, 600] loss: 1.116
```

```
[17, 700] loss: 1.182
[18, 100] loss: 1.136
[18, 200] loss: 1.108
[18, 300] loss: 1.119
[18, 400] loss: 1.118
[18, 500] loss: 1.159
[18, 600] loss: 1.125
[18, 700] loss: 1.160
[19, 100] loss: 1.116
[19, 200] loss: 1.105
[19, 300] loss: 1.092
[19, 400] loss: 1.117
[19, 500] loss: 1.123
[19, 600] loss: 1.135
[19, 700] loss: 1.161
[20, 100] loss: 1.115
[20, 200] loss: 1.125
[20, 300] loss: 1.106
[20, 400] loss: 1.068
[20, 500] loss: 1.131
[20, 600] loss: 1.116
[20, 700] loss: 1.121
Training completed in 65.90 seconds
```

[10]:
```
test(model1, testloader)
```

```
testing finished in 0.51 seconds, Accuracy: 59.54%
```

1.4.3 With Dropout and Batch Norm

[11]:
```
model2 = LeNet_v2().to(device)
# DataLoader for training and test datasets
trainloader = torch.utils.data.DataLoader(cifar_trainset,␣
 ↪batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(cifar_testset, batch_size=batch_size,␣
 ↪shuffle=False)

train(model2, trainloader)
```

```
[1, 100] loss: 2.173
[1, 200] loss: 1.873
[1, 300] loss: 1.765
[1, 400] loss: 1.705
[1, 500] loss: 1.658
[1, 600] loss: 1.638
[1, 700] loss: 1.566
[2, 100] loss: 1.535
[2, 200] loss: 1.528
```

```
[2, 300] loss: 1.513
[2, 400] loss: 1.481
[2, 500] loss: 1.493
[2, 600] loss: 1.454
[2, 700] loss: 1.429
[3, 100] loss: 1.398
[3, 200] loss: 1.391
[3, 300] loss: 1.383
[3, 400] loss: 1.424
[3, 500] loss: 1.399
[3, 600] loss: 1.378
[3, 700] loss: 1.337
[4, 100] loss: 1.356
[4, 200] loss: 1.320
[4, 300] loss: 1.314
[4, 400] loss: 1.310
[4, 500] loss: 1.301
[4, 600] loss: 1.316
[4, 700] loss: 1.293
[5, 100] loss: 1.271
[5, 200] loss: 1.260
[5, 300] loss: 1.299
[5, 400] loss: 1.262
[5, 500] loss: 1.245
[5, 600] loss: 1.255
[5, 700] loss: 1.246
[6, 100] loss: 1.222
[6, 200] loss: 1.216
[6, 300] loss: 1.233
[6, 400] loss: 1.206
[6, 500] loss: 1.223
[6, 600] loss: 1.239
[6, 700] loss: 1.189
[7, 100] loss: 1.170
[7, 200] loss: 1.187
[7, 300] loss: 1.197
[7, 400] loss: 1.183
[7, 500] loss: 1.179
[7, 600] loss: 1.192
[7, 700] loss: 1.176
[8, 100] loss: 1.147
[8, 200] loss: 1.162
[8, 300] loss: 1.168
[8, 400] loss: 1.144
[8, 500] loss: 1.167
[8, 600] loss: 1.184
[8, 700] loss: 1.138
[9, 100] loss: 1.132
```

```
[9, 200] loss: 1.141
[9, 300] loss: 1.144
[9, 400] loss: 1.139
[9, 500] loss: 1.145
[9, 600] loss: 1.127
[9, 700] loss: 1.139
[10, 100] loss: 1.113
[10, 200] loss: 1.085
[10, 300] loss: 1.121
[10, 400] loss: 1.108
[10, 500] loss: 1.109
[10, 600] loss: 1.132
[10, 700] loss: 1.122
[11, 100] loss: 1.102
[11, 200] loss: 1.108
[11, 300] loss: 1.097
[11, 400] loss: 1.080
[11, 500] loss: 1.098
[11, 600] loss: 1.101
[11, 700] loss: 1.113
[12, 100] loss: 1.080
[12, 200] loss: 1.060
[12, 300] loss: 1.126
[12, 400] loss: 1.061
[12, 500] loss: 1.072
[12, 600] loss: 1.085
[12, 700] loss: 1.111
[13, 100] loss: 1.049
[13, 200] loss: 1.056
[13, 300] loss: 1.074
[13, 400] loss: 1.054
[13, 500] loss: 1.080
[13, 600] loss: 1.071
[13, 700] loss: 1.090
[14, 100] loss: 1.042
[14, 200] loss: 1.051
[14, 300] loss: 1.063
[14, 400] loss: 1.066
[14, 500] loss: 1.056
[14, 600] loss: 1.068
[14, 700] loss: 1.080
[15, 100] loss: 1.032
[15, 200] loss: 1.031
[15, 300] loss: 1.046
[15, 400] loss: 1.049
[15, 500] loss: 1.053
[15, 600] loss: 1.095
[15, 700] loss: 1.045
```

```
[16, 100] loss: 1.018
[16, 200] loss: 1.029
[16, 300] loss: 1.035
[16, 400] loss: 1.069
[16, 500] loss: 1.032
[16, 600] loss: 1.043
[16, 700] loss: 1.035
[17, 100] loss: 1.000
[17, 200] loss: 1.002
[17, 300] loss: 1.018
[17, 400] loss: 1.020
[17, 500] loss: 1.015
[17, 600] loss: 1.037
[17, 700] loss: 1.049
[18, 100] loss: 0.987
[18, 200] loss: 1.005
[18, 300] loss: 1.016
[18, 400] loss: 1.038
[18, 500] loss: 1.043
[18, 600] loss: 1.014
[18, 700] loss: 1.040
[19, 100] loss: 0.986
[19, 200] loss: 1.000
[19, 300] loss: 0.967
[19, 400] loss: 1.008
[19, 500] loss: 1.024
[19, 600] loss: 1.026
[19, 700] loss: 1.034
[20, 100] loss: 0.994
[20, 200] loss: 1.009
[20, 300] loss: 1.005
[20, 400] loss: 0.990
[20, 500] loss: 1.031
[20, 600] loss: 1.027
[20, 700] loss: 0.998
Training completed in 67.51 seconds
```

[13]:
```
test(model2, testloader)
```

```
testing finished in 0.50 seconds, Accuracy: 64.52%
```

1.5 Conclusion

I used a batch size of 64, as it gave me better performance than 128, 256. The smaller batch size allowed for more frequent weight updates, which helped the model navigate the loss landscape more effectively. While larger batches could potentially use GPU resources more efficiently, the 64 batch size struck the right balance between computational efficiency and learning dynamics for this CIFAR-10 classification task.

The learning rate of 0.01 with SGD optimizer and momentum of 0.9 provided stable convergence without oscillation issues. Training for 20 epochs was sufficient to demonstrate the differences between the model variations while avoiding overfitting in the base model.

1.5.1 Dropout

Adding dropout to the LeNet architecture shows significant improvement in preventing overfitting: - Acts as a regularization technique by randomly "dropping" neurons during training - Forces the network to learn more robust features that don't rely on specific neurons - Creates an implicit ensemble effect, as each training iteration uses a different subset of neurons

1.5.2 Dropout + Batch Normalization

Combining dropout with batch normalization further enhances model performance: - Batch normalization stabilizes the learning process by normalizing layer inputs - Reduces internal covariate shift, allowing for higher learning rates - Works synergistically with dropout to improve both training speed and generalization - Helps mitigate vanishing/exploding gradient problems in deeper networks

These did not improve the accuracy too much, but a win is a win