Homework 7 Report

Using a CNN for Cifar-10

As a bonus, I decided to implement a CNN to try and increase the accuracy. The architecture I settled on is shown below. I used the SGD optimizer with a learning rate of 0.01 and a momentum of 0.9. The performance / size ratio of the model is really good.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 128 * 4 * 4)  # Flatten
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Below are the results of training this model for 20 epochs.

Epoch [1/20], Loss: 1.5823
Epoch [2/20], Loss: 1.2962
Epoch [3/20], Loss: 1.1798

Epoch [4/20], Loss: 0.9053
Epoch [5/20], Loss: 0.9060
Epoch [6/20], Loss: 0.6863
Epoch [7/20], Loss: 0.7418
Epoch [8/20], Loss: 0.6138
Epoch [9/20], Loss: 0.7538
Epoch [10/20], Loss: 0.6263
Epoch [11/20], Loss: 0.7923
Epoch [12/20], Loss: 0.4616
Epoch [13/20], Loss: 0.5316
Epoch [14/20], Loss: 0.5991
Epoch [15/20], Loss: 0.6361
Epoch [16/20], Loss: 0.6423
Epoch [17/20], Loss: 0.3394
Epoch [18/20], Loss: 0.6074
Epoch [19/20], Loss: 0.4154
Epoch [20/20], Loss: 0.5483
took **164.86** seconds to train, and final model accuracy is **80.13%**.

GitHub link: https://github.com/Anu78/intro-to-ml-hw

Resnet-10 Model for Cifar-10

The advantage of a ResNet model compared to a regular CNN is that it addresses the vanishing gradient problem that is present in exceedingly deep networks with a lot of layers. Deep networks are unable to update gradients of parameters early in the network, and the back-propagation fails to reach the very beginning of the model. A resnet implementation in PyTorch looks like this:

```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()

        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection to downsample residual
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != self.expansion * out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, self.expansion * out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out
```

The model is built using these BasicBlocks, which are the core component of the ResNet model. They contain shortcut layers that allow the model to bypass layers.

```python
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64

        # Initial convolutional layer
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
```

```
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # Creating layers of blocks
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)

        # Average Pooling and Fully Connected Layer
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.fc(out)
        return out

# Define the ResNet-10 architecture (2+2+2+2+2 blocks)
def ResNet10():
    return ResNet(BasicBlock, [2, 2, 2, 2, 2])
```

This ResNet implementation also incorporates batch normalization, which normalizes the inputs per batch, and has trainable parameters that learn how much that normalization affects the losses per batch.

The final training lasted for 20 epochs and basically ended with 0 loss on the training dataset. It also performs better overall compared to the simple CNN, although with more tweaking it’s possible to get into the 99% range.

```
Epoch [1/20], Loss: 0.8108
Epoch [2/20], Loss: 0.4494
```

Epoch [3/20], Loss: 0.4821
Epoch [4/20], Loss: 0.4683
Epoch [5/20], Loss: 0.2822
Epoch [6/20], Loss: 0.2255
Epoch [7/20], Loss: 0.1669
Epoch [8/20], Loss: 0.2284
Epoch [9/20], Loss: 0.0571
Epoch [10/20], Loss: 0.0556
Epoch [11/20], Loss: 0.0727
Epoch [12/20], Loss: 0.0529
Epoch [13/20], Loss: 0.0063
Epoch [14/20], Loss: 0.0069
Epoch [15/20], Loss: 0.0448
Epoch [16/20], Loss: 0.0008
Epoch [17/20], Loss: 0.0002
Epoch [18/20], Loss: 0.0004
Epoch [19/20], Loss: 0.0001
Epoch [20/20], Loss: 0.0001
took 341.42 seconds to train, and final model accuracy is 85.46%

Github link: https://github.com/Anu78/intro-to-ml-hw