2B_MultiLayer_Perceptron_Assignment

May 12, 2021

1 PyTorch Assignment: Multi-Layer Perceptron (MLP)

Duke Community Standard: By typing your name below, you are certifying that you have adhered to the Duke Community Standard in completing this assignment.

Name: Ryan Hou

1.0.1 Multi-Layer Perceptrons

The simple logistic regression example we went over in the previous notebook is essentially a one-layer neural network, projecting straight from the input to the output predictions. While this can be effective for linearly separable data, occasionally a little more complexity is necessary. Neural networks with additional layers are typically able to learn more complex functions, leading to better performance. These additional layers (called "hidden" layers) transform the input into one or more intermediate representations before making a final prediction.

In the logistic regression example, the way we performed the transformation was with a fully-connected layer, which consisted of a linear transform (matrix multiply plus a bias). A neural network consisting of multiple successive fully-connected layers is commonly called a Multi-Layer Perceptron (MLP). In the simple MLP below, a 4-d input is projected to a 5-d hidden representation, which is then projected to a single output that is used to make the final prediction.

For the assignment, you will be building a MLP for MNIST. Mechanically, this is done very similary to our logistic regression example, but instead of going straight to a 10-d vector representing our output predictions, we might first transform to a 500-d vector with a "hidden" layer, then to the output of dimension 10. Before you do so, however, there's one more important thing to consider.

1.0.2 Nonlinearities

We typically include nonlinearities between layers of a neural network. There's a number of reasons to do so. For one, without anything nonlinear between them, successive linear transforms (fully connected layers) collapse into a single linear transform, which means the model isn't any more expressive than a single layer. On the other hand, intermediate nonlinearities prevent this collapse, allowing neural networks to approximate more complex functions.

There are a number of nonlinearities commonly used in neural networks, but one of the most popular is the rectified linear unit (ReLU):

$$x = \max(0, x) \tag{1}$$

There are a number of ways to implement this in PyTorch. We could do it with elementary PyTorch operations:

[1]:
```
import torch

x = torch.rand(5, 3)*2 - 1
x_relu_max = torch.max(torch.zeros_like(x),x)

print("x: {}".format(x))
print("x after ReLU with max: {}".format(x_relu_max))
```

```
x: tensor([[-0.0939,  0.2597, -0.3447],
        [ 0.8350,  0.4745, -0.3113],
        [-0.2241,  0.1941, -0.5495],
        [-0.9499,  0.4255, -0.7226],
        [-0.8520,  0.8791,  0.1414]])
x after ReLU with max: tensor([[0.0000, 0.2597, 0.0000],
        [0.8350, 0.4745, 0.0000],
        [0.0000, 0.1941, 0.0000],
        [0.0000, 0.4255, 0.0000],
        [0.0000, 0.8791, 0.1414]])
```

Of course, PyTorch also has the ReLU implemented, for example in `torch.nn.functional`:

[2]:
```
import torch.nn.functional as F

x_relu_F = F.relu(x)

print("x after ReLU with nn.functional: {}".format(x_relu_F))
```

```
x after ReLU with nn.functional: tensor([[0.0000, 0.2597, 0.0000],
        [0.8350, 0.4745, 0.0000],
        [0.0000, 0.1941, 0.0000],
        [0.0000, 0.4255, 0.0000],
        [0.0000, 0.8791, 0.1414]])
```

Same result.

1.0.3 Assignment

Build a 2-layer MLP for MNIST digit classfication. Feel free to play around with the model architecture and see how the training time/performance changes, but to begin, try the following:

Image (784 dimensions) ->
fully connected layer (500 hidden units) -> nonlinearity (ReLU) ->
fully connected (10 hidden units) -> softmax

Try building the model both with basic PyTorch operations, and then again with more object-oriented higher-level APIs. You should get similar results!

Some hints: - Even as we add additional layers, we still only require a single optimizer to learn the parameters. Just make sure to pass all parameters to it! - As you'll calculate in the Short Answer, this MLP model has many more parameters than the logisitic regression example, which makes it more challenging to learn. To get the best performance, you may want to play with the learning rate and increase the number of training epochs. - Be careful using `torch.nn.CrossEntropyLoss()`. If you look at the PyTorch documentation: you'll see that `torch.nn.CrossEntropyLoss()` combines the softmax operation with the cross-entropy. This means you need to pass in the logits (predictions pre-softmax) to this loss. Computing the softmax separately and feeding the result into `torch.nn.CrossEntropyLoss()` will significantly degrade your model's performance!

[1]:
```
### YOUR CODE HERE

%matplotlib inline

import random
import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm

import torch
import torch.nn as nn
from torchvision import datasets, transforms

class Logistic_Regression(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Flatten(1), # faltten an image to a vector
            nn.Linear(28*28, 500),
            nn.ReLU(),
            nn.Linear(500, 10)
        )

    def forward(self, x):
        return self.layers(x)

# load data and create minibatch
mnist_train = datasets.MNIST(root="./datasets", train=True,␣
 ↪transform=transforms.ToTensor(), download=True)
mnist_test = datasets.MNIST(root="./datasets", train=False,␣
 ↪transform=transforms.ToTensor(), download=True)
train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=100,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(mnist_test, batch_size=100,␣
 ↪shuffle=False)
```

```
model = Logistic_Regression()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for images, labels in tqdm(train_loader):

    optimizer.zero_grad()

    x = images.view(-1, 28*28)
    y = model(x)
    loss = criterion(y, labels)

    loss.backward()
    optimizer.step()

# Test is the same as before
correct = 0
with torch.no_grad():
    for images, labels in tqdm(test_loader):
        x = images.view(-1, 28*28)
        y = model(x)
        predictions = torch.argmax(y, dim=1)
        correct += torch.sum((predictions == labels).int())

print(f'Test accuracy: {correct/len(mnist_test)}')


# Make sure to print out your accuracy on the test set at the end.
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0), HTML(value='')))
```

```
Test accuracy: 0
```

1.0.4 Short answer

How many trainable parameters does your model have? How does this compare to the logisitic regression example?

[Your answer here]