2B_MultiLayer_Perceptron_Assignment

August 10, 2024

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:

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:

[15]:
```
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([[ 0.6884,  0.8074,  0.8749],
        [ 0.2012, -0.2841,  0.7612],
        [-0.1938,  0.6472, -0.1155],
        [ 0.9222,  0.9166, -0.7331],
        [-0.9161, -0.1950, -0.5097]])
x after ReLU with max: tensor([[0.6884, 0.8074, 0.8749],
        [0.2012, 0.0000, 0.7612],
        [0.0000, 0.6472, 0.0000],
        [0.9222, 0.9166, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

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

[16]:
```
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([[0.6884, 0.8074, 0.8749],
        [0.2012, 0.0000, 0.7612],
        [0.0000, 0.6472, 0.0000],
        [0.9222, 0.9166, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

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!

[18]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import time

# Define a function to retry downloading the dataset
def download_data():
    for _ in range(5):  # Retry up to 5 times
        try:
            train_dataset = datasets.MNIST(root='./data', train=True,␣
 ↪transform=transform, download=True)
            test_dataset = datasets.MNIST(root='./data', train=False,␣
 ↪transform=transform)
            return train_dataset, test_dataset
        except Exception as e:
            print(f"Error downloading dataset: {e}")
            time.sleep(5)  # Wait for 5 seconds before retrying
    raise RuntimeError("Failed to download dataset after several attempts")

# Transform to normalize the data and flatten the images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    transforms.Lambda(lambda x: x.view(-1))
])

# Download MNIST dataset with retries
train_dataset, test_dataset = download_data()

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)
```

```python
class BasicMLP:
    def __init__(self):
        self.W1 = torch.randn(784, 500, requires_grad=True) * 0.01
        self.b1 = torch.zeros(500, requires_grad=True)
        self.W2 = torch.randn(500, 10, requires_grad=True) * 0.01
        self.b2 = torch.zeros(10, requires_grad=True)

    def forward(self, x):
        z1 = x @ self.W1 + self.b1
        a1 = torch.relu(z1)
        z2 = a1 @ self.W2 + self.b2
        return z2  # We return the logits here

    def parameters(self):
        return [self.W1, self.b1, self.W2, self.b2]

# Instantiate the model
model = BasicMLP()

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(784, 500)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x  # Again, we return the logits

# Instantiate the model
model = MLP()


criterion = nn.CrossEntropyLoss()  # Combines softmax and cross-entropy
optimizer = optim.SGD(model.parameters(), lr=0.01)

num_epochs = 5
for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
```

```
        if batch_idx % 100 == 0:
            print(f'Epoch {epoch+1}/{num_epochs}, Step {batch_idx}/
 ↪{len(train_loader)}, Loss: {loss.item():.4f}')


correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f'Accuracy: {100 * correct / total:.2f}%')
```

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

HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))


Error downloading dataset: Remote end closed connection without response
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))


Error downloading dataset: Remote end closed connection without response
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz


HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))


Error downloading dataset: Remote end closed connection without response
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))


Error downloading dataset: Remote end closed connection without response
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Error downloading dataset: Remote end closed connection without response
```

```
      ␣
↪---------------------------------------------------------------------------

       RuntimeError                              Traceback (most recent call␣
↪last)

       <ipython-input-18-b6c4fe7328e1> in <module>
        26
        27 # Download MNIST dataset with retries
   ---> 28 train_dataset, test_dataset = download_data()
        29
        30 train_loader = DataLoader(dataset=train_dataset, batch_size=64,␣
↪shuffle=True)


       <ipython-input-18-b6c4fe7328e1> in download_data()
        16             print(f"Error downloading dataset: {e}")
        17             time.sleep(5)  # Wait for 5 seconds before retrying
   ---> 18     raise RuntimeError("Failed to download dataset after several␣
↪attempts")
        19
        20 # Transform to normalize the data and flatten the images


       RuntimeError: Failed to download dataset after several attempts
```

1.0.4 Short answer

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

1.0.5 Trainable Parameters Calculation:

Model Architecture:

1. **Input Layer to Hidden Layer (784 -> 500)**
   - **Weights:** 784 input units * 500 hidden units = 392,000 parameters
   - **Biases:** 500 hidden units = 500 parameters
2. **Hidden Layer to Output Layer (500 -> 10)**
   - **Weights:** 500 hidden units * 10 output units = 5,000 parameters
   - **Biases:** 10 output units = 10 parameters

Total Trainable Parameters:

- **Total:** 392,000 (weights) + 500 (biases) + 5,000 (weights) + 10 (biases) = **397,510** parameters

1.0.6 Comparison to Logistic Regression Example:

- **Logistic Regression Model (784 -> 10):**
  - **Weights:** 784 input units * 10 output units = 7,840 parameters
  - **Biases:** 10 output units = 10 parameters
  - **Total:** 7,840 + 10 = **7,850** parameters

1.0.7 Summary:

- Your 2-layer MLP model has **397,510** trainable parameters.
- The logistic regression model has **7,850** trainable parameters.

Thus, the MLP model has significantly more parameters than the logistic regression model, which makes it more complex and potentially more powerful, but also more challenging to train effectively.

[ ]: