CMSC 491/691: Computer Vision **Fall 2023**

Lecture 10: 2024-03-11 Pytorch Tutorial

Lecturer: Tejas Gokhale *Scribe: Kathleen Koerner*

The notes below are an example of what I am expecting. They were taken from a random graduate class. They illustrate some uses of various LaTeX macros. Take a look at this and imitate.

10.1 Recap of Last Lecture

In the last lecture, we talked about ConvNet, which is a combination of convolutional layers that use functions in order to transform inputs to outputs. It is a series of relationships between the various layers.

Convolution involves sliding a filter over an image and computing the dot products between the filter and the image. Images are 3 dimensions, and these filters will always match the depth of the image. Each position of the filter combined with the image results in one dot product, and thus the output of this function will have differing length and height dimensions, based on the size of the filter and image, and a depth of 1.

This function results in an activation map. This can be done with several filters, to output several different activation maps, such as 8 filters would result in 8 activation maps. Each map would have a depth of 1, so the depth of the resulting image would be 8 (1 for each map).

10.2 Pytorch

Pytorch is a machine-learning library that is helpful for creating deep learning models.

The tutorial starts with initializing tensors, which are data structures that can be compared to arrays and matrices. One way these can be created is by using the torch.tensor() method. They can also be created from a NumPy array, or from another tensor. Tensors have attributes associated with them as well, such as their shape, their datatype, and the device that they are stored on.

Various operations can be used on tensors, but first they must be moved to the GPU using the tensor.to() method. Tensors can be indexed, sliced, and concatenated. Other arithmetic operations can be performed, such as multiplication and transposition.

With the Pytorch library, you can also load and view datasets to be used for model training. Refer to the .ipynb file provided during class. There is even an example of a simple one-layer neural network that can be built as a starting point. This also include how to compute gradients and optimization.

I have included below the examples for the training loop, testing loop, and full implementation of this one-layer neural network:

Finally, you can use Pytorch to save and load the model weights, so that the model can be used again.

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Figure 10.1: The implementation written in python for the training loop.

```python
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Figure 10.2: The implementation written in python for the testing loop.

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Figure 10.3: The implementation written in python for the full implementation of the simple one-layer neural network.