Convolutional Neural Networks in PyTorch

A Short Overview of How to Implement CNNs in PyTorch

Scope: This document is a summary of the "Module 4: Convolutional Neural Networks in PyTorch" notebook from the course "Deep Learning: Mastering Neural Networks" by MIT xPRO. This summary offers a quick overview of the main aspects included in the notebook.

Table of Contents

1. CNN Autoencoder with MNIST Digits

The first step will be to revisit the autoencoder implementation from the previous notebook. Recall that this used a multiple layer perceptron (MLP) neural network, treating the input input as one long vector of grey scale values.

a. Dataset Preparation and Module Import

!Keep in mind: The dataset preparation and module import is going to be the exact same as the one from the first notebook of this module.

b. Model Definition and Hyperparameters

The model definition is where the most changes will appear, now to use convolutional layers rather than just fully connected linear layers. Specifically, we see new kinds of layers being introduced, including "Conv2d", "MaxPool2d", and "Flatten" layers. However, we will continue to follow the same encoding and decoding pattern.

```
from torch.nn.modules.flatten import Flatten
class CNNAutoEncoder(nn.Module):
    # A lot of these numbers (specifically the Linear layer at the end of the encoder
    # and the beginning of the decoder) are hardcoded for our 28x28 image size
    def __init__(self):
        super(CNNAutoEncoder, self).__init__()
        # Split the Encoder and Decoder
        self.encoder = nn.Sequential(
            # Perform a convolution on the input
            # in_channels = 1 because we have a grayscale image as input
            # The size of the Conv2d layer results in our output being resized
```

```
            # to (8, 28, 28).
            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1, padding=1),
            # Perform a ReLU() just like on a linear layer
            nn.ReLU(),
            # MaxPool2d with this size will downsize our image to (8, 14, 14)
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            # Flatten the layer
            nn.Flatten(),
            # Perform a ReLU for nonlinearity
            nn.ReLU(),
            # One last Linear layer for our encoding of size 32 (like in previous version)
            nn.Linear(8*14*14, 32),
        )
        self.decoder = nn.Sequential(
            # The input to our decoder will be a linear tensor of lenght 64
            nn.Linear(32, 8*14*14),
            # Perform a ReLU for nonlinearity
            nn.ReLU(),
            # Here we will Return to our unflattened size
            nn.Unflatten(dim = 1, unflattened_size = (8, 14, 14)),
            # Perform  a ReLU for nonlinearity
            nn.ReLU(),
            # ConvTranspose2d can be thought of as an "UnConvolution" which returns us
            # to our original dimensions (1, 28, 28)
            nn.ConvTranspose2d(in_channels = 8, out_channels = 1, kernel_size = 2, stride = 2),
        )

    def forward(self, x):
        return self.decoder(self.encoder(x))

    def encode(self, x):
        return self.encoder(x)
```

```
learning_rate = 0.001
num_epochs = 10

model = CNNAutoEncoder().to(device)
```

c. Training the Model

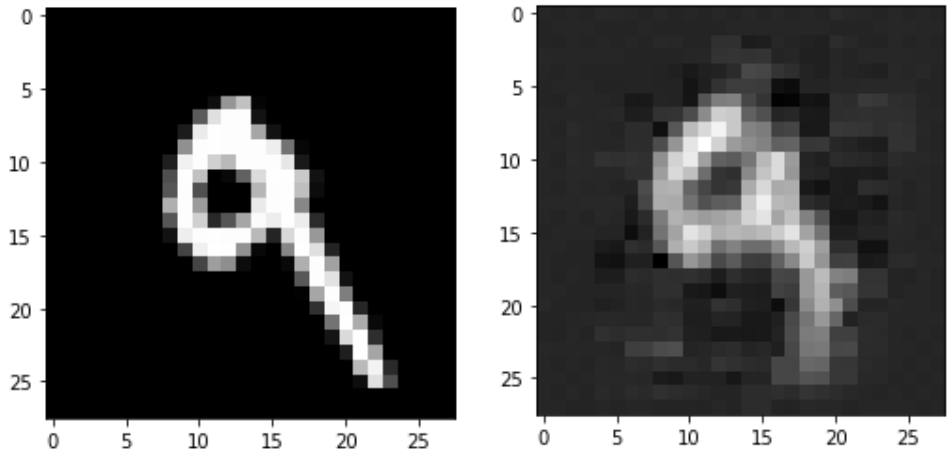
Even though the model is much more complex, we can still use the same basic autoencoder training function as before. This time, however, we won't flatten the inputs since we can now handle 2D images as the input.

d. Visualizing the Results

You can now test out the visualization of some of the different digits, using the same "plot_digit" function in the notebook. You may notice a more accurate reconstruction than in the previous autoencoder, especially for digits 8 and 9. The key idea is that the encoder part of the autoencoder has learned how to extract key "features" from the image, as encoded in the middle latent layer. This motivates us to use similar kinds of

convolutional layers in deep neural networks, to learn good feature extraction to help with other tasks, like classification.

```
plot_digit(9)
```



2. Deep CNN for MNIST Digit Classification

Now we will revisit the MNIST Digit Classification task and see how well our model performs after introducing convolutional layers, to learn good feature extractions before a final linear output layers that learn how to combine those features to make a decision about what kind of digit the image is.

a. Dataset Preparation and Module Import

The dataset preparation and imports are the same as for our CNN autoencoder of the MNIST dataset.

b. Model Definition and Hyperparameters

First, we will perform a convolution and then a max pooling operation to learn 2D features in our dataset. Next, we will perform a flattening and include two layers in order to get classification down to our 10 different classes. Most CNNs follow this pattern, in which the beginning of the network performs spatial feature extraction and the end of the network contains linear layers for classification.

```
from torch.nn.modules.flatten import Flatten
class CNNClassifier(nn.Module):
    def __init__(self):
        super(CNNClassifier, self).__init__()
        self.pipeline = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Flatten(),
            nn.Linear(8*14*14, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )
    def forward(self, x):
        return self.pipeline(x)
```

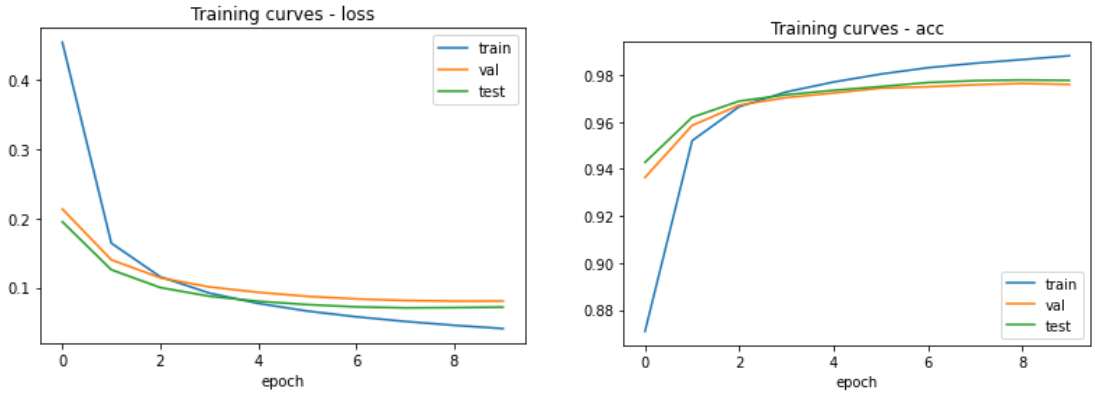
```
learning_rate = 0.001
num_epochs = 10

model = CNNClassifier().to(device)
```

c. Training the Model

The notebook provides a "train_classification_model" function to train our CNN classifier. It is important to note that this is almost identical to the previous classifier training functions we've seen, for multiple class situations. Here we have 10 different classes, 0 through 9, corresponding to the MNIST digits.

d. Visualizing Training Curves and Results

The notebook also uses the same "plot_training_curves" and confusion matrix "plot_cm" functions that we've seen in earlier notebooks. First, the training curves look good, even for only 10 epochs of training, with no indication of overfitting (the validation and test loss and accuracies are quite similar). Applying our trained model to our MNIST test split, we see that the confusion matrix contains fewer misclassified entries than before. Indeed, this simple CNN MNIST classifier is very impressive – achieving about 97% accuracy!




```
res = plot_cm(model, device, dataloaders, phase='test')
```