Practical 5 MNIST and CNNs in PyTorch

Johan Neethling (24739286) 2024-04-29

Introduction

In this practical we utilised PyTorch to implement a convolutional neural network (CNN). We were given a notebook containing the relevant skeleton code and MNIST dataset. We had to define, instantiate, and work with the model.

Report

Question 1

We used the *read_npz* function to read in the datasets as training and test sets. This function was defined in the *helper_functions* script and implemented for us.

```python
# read datasets
train_X,train_Y = read_npz("data/train.npz")
test_X,test_Y = read_npz("data/test.npz")

print("+" + 40 * '-')
print(f"| Train X: {train_X.shape}")
print(f"| Train Y: {train_Y.shape}")
print(f"| Test X: {test_X.shape}")
print(f"| Test Y: {test_Y.shape}")
print("+ " + 40 * '-')
```

✓ 1.1s Python

FIGURE 1: CODE FROM QUESTION 1

Question 2

We defined the CNN layers and forward pass in **2.1** and **2.2**. We then instantiated the model in **2.3**.

Layer	Type	Input	Kernel	Num (C)	Stride	Activation	Layer Output
1	CNN	(28 x 28 x 1)	(8 x 8 x ▢)	10	2	ReLU	(▢ x ▢ x 10)
2	Dense	(▢)	-	-	-	-	(10)

FIGURE 2: PROVIDED MODEL ARCHITECTURE

Each input is a 28x28x1 image. We put that input through a two dimensional convolutional later which use 10 (C = number of kernels) 8x8x1 kernels ($M_3 = P_3 = 1$) with a stride of 2 and a ReLU activation function. Our first layer outputs an 11x11x10 stack.

Our second layer puts that output stack through a linear layer with an input of a 1x1210 vector and an output of a 1x10 vector.

$$R_{q_1 q_2} = r_{uh}[q_1 - 1, q_2 - 1] = \sum_{i_1=q_1}^{q_1+N_{h_1}-1} \sum_{i_2=q_2}^{q_2+N_{h_2}-1} U_{i_1 i_2}.H_{(i_1-q_1+1)(i_2-q_2+1)}$$

$$Z_{q_1 q_2} = b + Rq_1 q_2$$

$$V_{q_1 q_2} = \sigma(Z_{q_1 q_2})$$

FIGURE 3: MATHEMATICAL BREAKDOWN OF A GENERAL CONVOLUTIONAL LAYER

R represents the results of convolution; we then add the bias and perform an activation function (in this case ReLU) to end up with **V**.

```
#************************** YOUR CODE HERE ***********************#

P_3 = 1
kernel = (M_1, M_2)

self.conv_1 = torch.nn.Conv2d(P_3, M_3, kernel, stride)
self.relu = torch.nn.ReLU()
self.dense = torch.nn.Linear(1210, K)
# self.softmax = torch.nn.Softmax(dim=-1)

#*************************************************************#
```

FIGURE 4: CODE FROM QUESTION 2.1 - DEFINITION OF CNN LAYERS

We set P_3 = 1 to match the provided input dimensions. We define the convolutional, ReLU and dense layers for the model as requested. The softmax exclusion will be discuss later.

```
#************************** YOUR CODE HERE **********************#

new_shape = (-1, 1, 28, 28)
# print("x: ", x.shape)

x = torch.reshape(x, new_shape)
# print("x: ", x.shape)
z_1 = self.conv_1(x)
# print("z_1: ", z_1.shape)
v_1 = self.relu(z_1)
# print("v_1: ", v_1.shape)

u_2 = torch.reshape(v_1, (v_1.shape[0], -1))
# print("u_2: ", u_2.shape)

y_hat = self.dense(u_2)
# print("y_hat: ", y_hat.shape)
# y_hat = self.softmax(y_hat)
# print("y_hat: ", y_hat.shape)

#****************************************************************#
```

FIGURE 5: CODE FROM QUESTION 2.2 - DEFINITION OF CNN FORWARD PASS

We reshape our data, then put it through our predefined layers. We used the print statements to debug our model and ensure stack shape throughout model forward pass.

```
x:  torch.Size([784])
x:  torch.Size([1, 1, 28, 28])
z_1:  torch.Size([1, 10, 11, 11])
v_1:  torch.Size([1, 10, 11, 11])
u_2:  torch.Size([1, 1210])
y_hat:  torch.Size([1, 10])
```

FIGURE 6: SHAPES OF DATA THROUGH FORWARD PASS

```
#************************** YOUR CODE HERE **********************#

model = CNN(8, 8, 10, 10, 2)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), 0.01)

#****************************************************************#
```

FIGURE 7: CODE FROM QUESTION 2.3 - INSTANTIATION OF CNN MODEL AND LOSS FUNCTION

We instantiated the CNN model for the provided architecture (as seen in figure 2). We implemented a cross entropy loss function and SGD optimizer as requested.

The loss function compares our probabilities per input to the expected output and returns the loss of our model. The *CrossEntropyLoss()* function from the PyTorch library does not require normalised predicted outputs. After experimentation, we found that the softmax layer drops the model accuracy from approximately 93 to approximately 90.

The optimizer is used to perform gradient descent and update the model's learnt weights with ease. It is a predefined function from the PyTorch library and abstracts the complexities of process to provide the user with a streamlined experience. The SGD optimizer minimises the loss function during the training phase when using batched data.

Question 3

We implemented the model's forward pass, model loss, and backward pass to train and test the model in **3.1** and **3.3**. We also update the model's parameters during the training process in **3.2**.

```
#************************** YOUR CODE HERE **********************#

y_hat,z_1,v_1 = model(X_i)
loss = loss_fn(y_hat, torch.nn.functional.one_hot(y_i,10).float())
loss.backward()

#************************************************************#
```

FIGURE 8: CODE FROM QUESTION 3.1 - FORWARD PASS, LOSS CALCULATION AND BACKWARD PASS OF TRAINING PHASE

To allow the loss function to compute, we needed to reformat the true outputs into one-hot encoded PyTorch tensors.

```
tensor([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9, 4, 0, 9, 1,
        1, 2, 4, 3, 2, 7])
tensor([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
...
```

FIGURE 9: COMPARISON BETWEEN Y_I BEFORE AND AFTER REFORMATTING

```
#************************** YOUR CODE HERE ***********************#

optimizer.step()
optimizer.zero_grad()

#*****************************************************************#
```

FIGURE 10: CODE FROM QUESTION 3.2 - UPDATE OF MODEL PARAMETERS

We update the model weights by gradient in the *step()* function and reset the gradients in *zero_grad()* function to prevent gradient accumulation.

```
#************************** YOUR CODE HERE ***********************#

y_hat,z_1,v_1 = model(X_i)
loss = loss_fn(y_hat,nn.functional.one_hot(y_i,10).float())
# loss.backward()

#*****************************************************************#
```

FIGURE 11: CODE FROM QUESTION 3.3 - FORWARD PASS & LOSS CALCULATION FOR TEST PHASE

We implement the forward pass and loss function on unseen data to test our model's prediction capability as we train it over 10 epochs of 30 observations each. We had initially implemented another backward pass on the test data (as requested), but that broke the code and did not make sense.

```
Epoch 1 Loss: 0.0105 Accuracy: 9.52 Test Loss: 0.0047 Test Accuracy: 85.19
Epoch 2 Loss: 0.0039 Accuracy: 9.52 Test Loss: 0.0042 Test Accuracy: 84.14
Epoch 3 Loss: 0.0028 Accuracy: 9.52 Test Loss: 0.0038 Test Accuracy: 85.12
Epoch 4 Loss: 0.0022 Accuracy: 9.52 Test Loss: 0.0034 Test Accuracy: 85.96
Epoch 5 Loss: 0.0017 Accuracy: 9.52 Test Loss: 0.0031 Test Accuracy: 90.11
Epoch 6 Loss: 0.0014 Accuracy: 9.52 Test Loss: 0.0028 Test Accuracy: 90.26
Epoch 7 Loss: 0.0011 Accuracy: 9.52 Test Loss: 0.0025 Test Accuracy: 91.39
Epoch 8 Loss: 0.0009 Accuracy: 9.52 Test Loss: 0.0023 Test Accuracy: 92.34
Epoch 9 Loss: 0.0008 Accuracy: 9.52 Test Loss: 0.0021 Test Accuracy: 92.46
Epoch 10 Loss: 0.0007 Accuracy: 9.52 Test Loss: 0.0019 Test Accuracy: 93.90
```

FIGURE 12: ACCURACY OF MODEL PER EPOCH OF TRAINING

Question 4

We were tasked to plot the data using a predefined function *plot_filters()* from the provided helper_functions script.

```
#************************** YOUR CODE HERE **********************#

y_hat,z_1,v_1 = model(X)
plot_filters(X, model.conv_1.weight, z_1, v_1)

#****************************************************************#
```

FIGURE 13: CODE FROM QUESTION 4 - IMPLEMENTATION AND PLOTTING OF MODEL

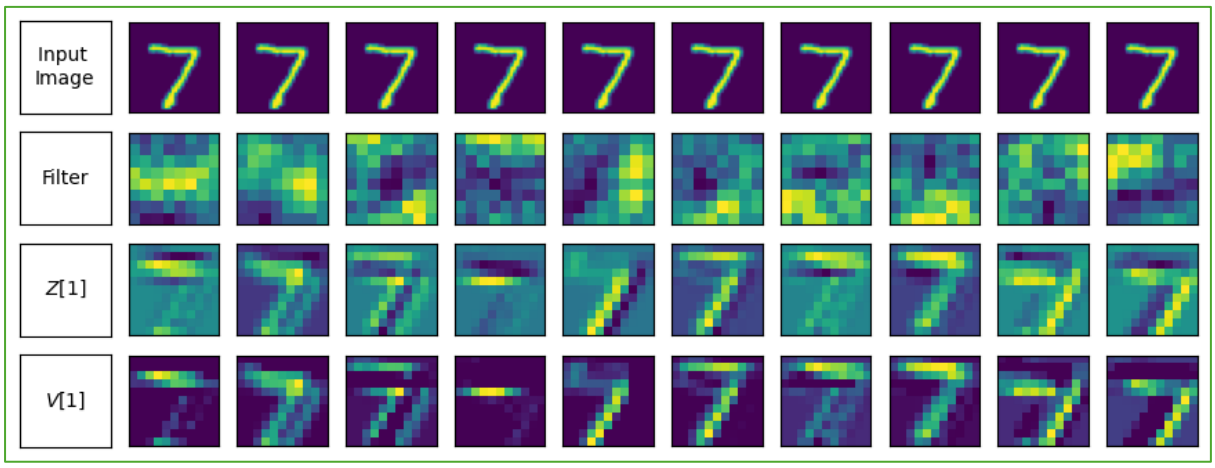

FIGURE 14: OUTPUT OF THE PLOT_FILTER() FUNCTION PERFORMED

Conclusion

This practical provided hands-on experience with implementing a convolutional neural network (CNN) in PyTorch for image classification. The key steps involved defining the CNN architecture with convolutional and dense layers, instantiating the model, defining the loss function and optimizer, and training the model on the training data while evaluating on the test set.

Through this process, several important concepts were reinforced, such as tensor reshaping, interpreting model outputs, calculating model loss, applying the model forward pass, and back-propagating gradients to update learnable parameters during optimization. Plotting utilities allowed visualising the learned convolutional filters.

Ultimately, the implemented CNN model achieved around 93% accuracy on the MNIST test set after 10 training epochs with stochastic gradient descent optimization. While simple, this hands-on exercise provided valuable experience with the key components involved in building, training, and evaluating CNNs using the PyTorch for an image classification task. The skills gained here can be extended to more complex CNN architectures and computer vision problems.