

Simple PyTorch Project

Overview

This guide will walk you through a very simple PyTorch training pipeline. Accompanying code for this article can be found here:

<https://git.arts.ac.uk/ipavlov/WikiMisc/blob/main/SimpleCNN.ipynb>

Loading Libraries

Every Python project starts by loading all the relevant libraries. In our case, the code for that is:

```
import torch
from glob import glob
import cv2
import albumentations
from albumentations.pytorch import ToTensorV2
from torch.utils.data import Dataset, DataLoader
```

Reading the Dataset

For this example, we will use the MNIST dataset, containing 70,000 images of handwritten digits from 0 to 9. The specific version of MNIST used in this example can be found here:

<https://www.kaggle.com/datasets/alexanderyyy/mnist-png>

Download the dataset and unpack it in the same directory your Jupyter Notebook is located. The unpacked dataset will consist of train and test folders containing images for model training and evaluation. Both test and train folders will have 10 sub-folders for each digit.

Python_download

To train our model, we need to know the file names and labels of all images in the dataset. A simple way to do this is demonstrated in the code below:

```
def readMnist(folder):
    filenames = [] #List for image filenames
    labels = [] #List for image labels

    folderNameLen = len(folder)
```

```

#Reads all the filenames in a given folder recursively
for filename in glob(folder + '/*/*.*.png', recursive=True):
    filenames += [filename]

    #Get the label of the image from it's filepath
    labels += [int(filename[folderNameLen:folderNameLen+1])]

return filenames, labels

trainFiles, trainLabels = readMnist('./mnist_png/train/')
testFiles, testLabels = readMnist('./mnist_png/test/')

```

Note: Different datasets will require different approaches.

Dataset Class

The Dataset class will provide necessary functionality to our training and evaluation pipeline, like loading images and labels, image transformations, and others.

```

class MnistDataset(Dataset):
    def __init__(self, filepaths, labels, transform):
        self.labels = labels
        self.filepaths = filepaths
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image = cv2.imread(self.filepaths[idx], 0)
        h,w = image.shape

        image = self.transform(image=image)["image"]/255.
        label = self.labels[idx]

        return image, label

# Usually transformations would include data augmentation tricks,
# but for this example we will limit ourselves to just converting image data from
# NumPy array to PyTorch tensor.
transform = albumentations.Compose(

```

```
[
    ToTensorV2()
]
```

To allow for mini-batch use we need to introduce a `DataLoader` to our pipeline.

```
#Instantiate Dataset objects for train and test datasets.
trainDataset = MnistDataset(trainFiles, trainLabels, transform)
testDataset = MnistDataset(testFiles, testLabels, transform)
```

Model Class

Our model classifies the input images between 10 different classes. Below is the code for our simple convolutional neural network. The comments in the code will provide additional explanation.

```
class CNN(torch.nn.Module):
    # Inside of __init__ we define the structure of our neural network.
    # Thinks of this as a collection of all potential layers and modules
    # that we will use during the feedforward process.
    def __init__(self):
        super().__init__() #Needed to initialize torch.nn.Module correctly

        # Our first convolutional block. torch.nn.Sequential is container
        # that will execute modules inside of it sequentially.
        # This convolutional block consists of a simple convolutional layer,
        # ReLU activation functions, and Max Pooling operation.
        self.conv1 = torch.nn.Sequential
            (torch.nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )
        # Our second onvolutional block.
        self.conv2 = torch.nn.Sequential(
```

```

        torch.nn.Conv2d(16, 32, 5, 1, 2),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(2),
    )
    # Fully connected layer that outputs 10 classes
    self.out = torch.nn.Linear(32 * 7 * 7, 10)

# forward is a function that is used for feedforward operation of our model.
# Input arguments are input data for our model. In this case x would be a batch of images from the MNIST
dataset.
# Inside of this function we apply modules we defined in __init__ to input images.
def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    #This line flattens tensors from 4 dimensions to 2.
    x = torch.flatten(x, start_dim=1)
    output = self.out(x)
    return output

# This line creates an object of our convolutional neural network class.
# We use .cuda() to send our model to GPU.
model = CNN().cuda()

```

Training and Validation

Below is the code for our training and validation procedure.

```

#Here we define cross entropy loss functions, which we will use for loss calculation.
loss_fn = torch.nn.CrossEntropyLoss()

#This is our optimizer algorithm. In this example we use Stochastic gradient descent
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

num_epochs = 25

#We will execute our training inside of a loop. Each iteration is a new epoch.
for epoch in range(num_epochs):
    print('Epoch:', epoch)

    running_loss = 0
    model = model.train() #sets our model to training mode.

```

```

for i, data in enumerate(train_dataloader): #we will iterate over our dataloader to get batched data.
    x, y = data
    #Don't forget to send your images and labels to the same device as your model. In our case it's a GPU.
    x = x.cuda()
    y = y.cuda()

    #Resets gradients
    optimizer.zero_grad()
    #output of our CNN model
    outputs = model(x)
    #Here we calculate loss values
    loss = loss_fn(outputs, y)

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

    running_loss += loss.item()

print(running_loss/len(train_dataloader)) #average training loss for current epoch

model = model.eval() #sets our model to evaluation mode.
test_acc = 0
test_running_loss = 0
for i, data in enumerate(test_dataloader):
    x, y = data
    x = x.cuda()
    y = y.cuda()

    outputs=model(x)
    loss = loss_fn(outputs, y)

    test_running_loss += loss.item()
    #We apply softmax here to get the probabilities for each class
    probs = torch.nn.functional.softmax(outputs, dim=1)
    #We select the highest probability as our final predication
    pred = torch.argmax(probs, dim=1)
    test_acc += torch.sum(pred == y)

#Average evaluation loss and evaluation accuracy for this epoch
print(test_running_loss/len(testDataset), test_acc/len(testDataset))

```

Revision #3

Created 20 March 2024 16:55:52 by Ilia Pavlov

Updated 29 April 2024 22:22:38 by Ilia Pavlov