ing-logistic-regression-in-pytorch

February 1, 2024

1 Identifying handwritten digits using Logistic Regression in PyTorch

Logistic Regressio allows us to predict a binary output from a set of independent variables.

1.0.1 Importing Libraries

[26]:
```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
```

1.0.2 Loading the MNIST DATASETS(images and labels)

[27]:
```
train_dataset = dsets.MNIST(root ='./data',
                            train = True,
                            transform = transforms.ToTensor(),
                            download = True)
test_dataset = dsets.MNIST(root ='./data',
                           train = False,
                           transform = transforms.ToTensor())
```

1.0.3 Defining my HyperParameters

[28]:
```
input_size = 784 # The data set is 28 * 28 thus input size will be 784
num_classes = 10 # we have 10 digitstherefor we expect 10 different outputs
num_epochs = 5
batch_size = 100
learning_rate = 0.001
```

In our dataset the image size is 28*28. Thus the input sizeis 784. also 10 digits are present in this and hense I can have 10 different outputs. Thus I set num_classes as 10 also we shall train five times the entire dataset and then we will train in small batches of 100 images a time, this is to prevent crashing due to memory overload

1.0.4 Dataset Loader (Input Pipeline)

[29]:
```
train_loader= torch.utils.data.DataLoader(dataset = train_dataset,
                                          batch_size = batch_size,
                                          shuffle = True)

test_loader= torch.utils.data.DataLoader(dataset = test_dataset,
                                          batch_size = batch_size,
                                          shuffle = False)
```

1.1 Defining the model

[30]:
```
class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out
```

I have our class I will ow insatntiate an object

[31]:
```
model = LogisticRegression(input_size, num_classes)
```

2 I will set up now the Loss function AND optimizer

i will use the cROSS ENTROPY LOSS AND FOR OPTIMIZER the STOCHASTIC GRADIENT DESCENT ALGORITHM(**SGD**) with a learning rate of 0.001

[32]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch. optim.SGD(model.parameters(), lr = learning_rate)
```

Now i will TRAIN BY DOING THE FOLLOWING:

- Reset all gradients to 0
- make a foward pass.
- Calculate the Loss.
- Perform backpropagation
- Update all weights.

[35]:
```
# Training the model
for epoch in range(num_epochs):
    for i, (images,labels) in enumerate(train_loader):
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # Forward + Backword + Optimize
```

```
        optimizer.zero_grad()
        outputs =  model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        if (i+1) % 100 ==0:
            print ('Epoch: [% d/% d], step: [% d/% d], Loss: %.4f' %(epoch+1,␣
↪num_epochs, i+1, len(train_dataset) // batch_size, loss.item()))
```

```
Epoch: [ 1/ 5], step: [ 100/ 600], Loss: 2.0439
Epoch: [ 1/ 5], step: [ 200/ 600], Loss: 1.9783
Epoch: [ 1/ 5], step: [ 300/ 600], Loss: 1.8991
Epoch: [ 1/ 5], step: [ 400/ 600], Loss: 1.8438
Epoch: [ 1/ 5], step: [ 500/ 600], Loss: 1.7745
Epoch: [ 1/ 5], step: [ 600/ 600], Loss: 1.7251
Epoch: [ 2/ 5], step: [ 100/ 600], Loss: 1.6573
Epoch: [ 2/ 5], step: [ 200/ 600], Loss: 1.5776
Epoch: [ 2/ 5], step: [ 300/ 600], Loss: 1.4355
Epoch: [ 2/ 5], step: [ 400/ 600], Loss: 1.4768
Epoch: [ 2/ 5], step: [ 500/ 600], Loss: 1.4803
Epoch: [ 2/ 5], step: [ 600/ 600], Loss: 1.3828
Epoch: [ 3/ 5], step: [ 100/ 600], Loss: 1.3586
Epoch: [ 3/ 5], step: [ 200/ 600], Loss: 1.3386
Epoch: [ 3/ 5], step: [ 300/ 600], Loss: 1.3194
Epoch: [ 3/ 5], step: [ 400/ 600], Loss: 1.2341
Epoch: [ 3/ 5], step: [ 500/ 600], Loss: 1.2819
Epoch: [ 3/ 5], step: [ 600/ 600], Loss: 1.1710
Epoch: [ 4/ 5], step: [ 100/ 600], Loss: 1.2103
Epoch: [ 4/ 5], step: [ 200/ 600], Loss: 1.1192
Epoch: [ 4/ 5], step: [ 300/ 600], Loss: 1.0727
Epoch: [ 4/ 5], step: [ 400/ 600], Loss: 1.0533
Epoch: [ 4/ 5], step: [ 500/ 600], Loss: 1.0732
Epoch: [ 4/ 5], step: [ 600/ 600], Loss: 0.9672
Epoch: [ 5/ 5], step: [ 100/ 600], Loss: 1.0434
Epoch: [ 5/ 5], step: [ 200/ 600], Loss: 0.9916
Epoch: [ 5/ 5], step: [ 300/ 600], Loss: 0.9735
Epoch: [ 5/ 5], step: [ 400/ 600], Loss: 0.8881
Epoch: [ 5/ 5], step: [ 500/ 600], Loss: 1.0379
Epoch: [ 5/ 5], step: [ 600/ 600], Loss: 1.0080
```

2.1 Now testing out the model

[37]:
```
# Model testing
correct = 0
total = 0
for images, labels in test_loader:
```

```
    images = Variable(images.view(-1, 28*28))
    outputs = model(images)
    _,predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct +=(predicted == labels).sum()

print('Accuracy of the model on the 10000 test images: % d %%'%(100* correct /␣
 ↪total))
```

```
Accuracy of the model on the 10000 test images:  83 %
```