Feed-Forward NeuralNet

January 10, 2022

[1]:
```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

[2]:
```python
# MNIST
# DataLoader, Transformation
# Multilayer Neural Net, activation function
# loss and optimizer
# training loop
# model evaluation
# GPU support
```

[3]:
```python
# device config
#device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[4]:
```python
#hyper parameters
input_size = 784 #28*28 img
hidden_size = 500
num_classes = 10
num_epochs = 10
batch_size = 100
learning_rate = 0.001
```

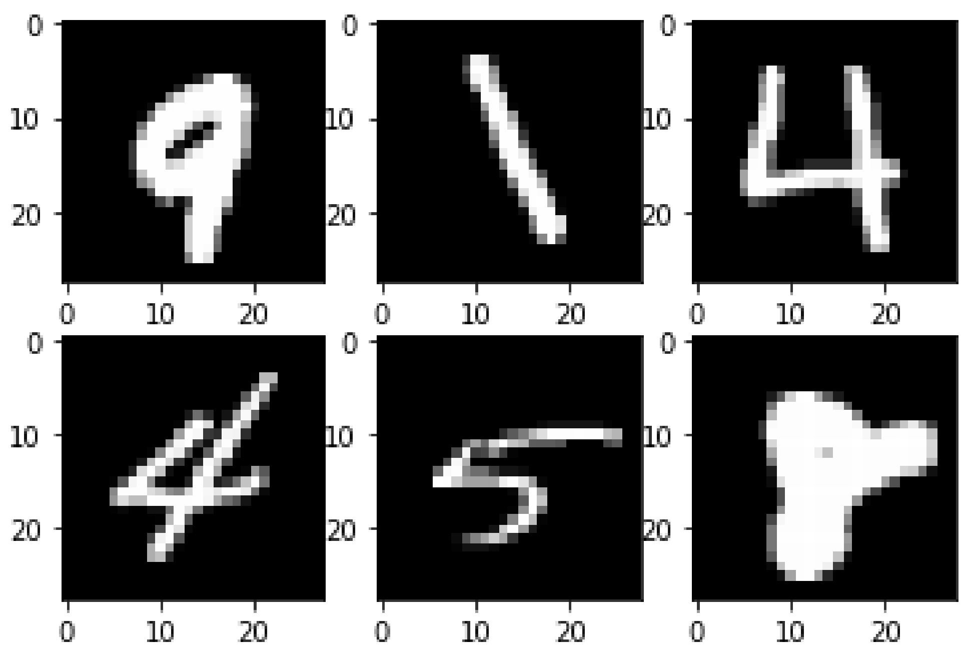
[5]:
```python
#MNIST
train_dataset = torchvision.datasets.MNIST(root='./MNIST', train=True,␣
 ↪transform=transforms.ToTensor(), download=True)
test_dataset = torchvision.datasets.MNIST(root='./MNIST', train=False,␣
 ↪transform=transforms.ToTensor(), download=False)
```

[6]:
```python
#Data Loader
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)
```

```
examples = iter(train_loader)
samples, labels = examples.next()
print(samples.shape, labels.shape)
```

```
torch.Size([100, 1, 28, 28]) torch.Size([100])
```

[7]:
```
for i in range (6):
    plt.subplot(2, 3, i+1)
    plt.imshow(samples[i][0], cmap = 'gray')
    #plt.show()
```



[8]:
```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        return out

model = NeuralNet(input_size, hidden_size, num_classes)
```

[9]:
```
# loss and optimizer
criterion = nn.CrossEntropyLoss() # applies softmax
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[10]:
```
# training loop
n_total_steps = len(train_loader)

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        #100, 1, 28, 28
        #100, 784
        images = images.reshape(-1, 28*28) #.to(device)
        labels = labels #.to(device)
    # forward
    outputs = model(images)
    loss = criterion(outputs, labels)
    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (i+1) % 100 == 0:
        print(f'Epoch [{epoch+1} / {num_epochs}], Step [{i+1}/{n_total_steps}],␣
 ↪Loss  {loss.item():.4f}')
```

```
Epoch [1 / 10], Step [600/600], Loss  2.3071
Epoch [2 / 10], Step [600/600], Loss  2.2113
Epoch [3 / 10], Step [600/600], Loss  2.1465
Epoch [4 / 10], Step [600/600], Loss  2.0621
Epoch [5 / 10], Step [600/600], Loss  1.9467
Epoch [6 / 10], Step [600/600], Loss  1.8571
Epoch [7 / 10], Step [600/600], Loss  1.7631
Epoch [8 / 10], Step [600/600], Loss  1.5891
Epoch [9 / 10], Step [600/600], Loss  1.6023
Epoch [10 / 10], Step [600/600], Loss  1.4807
```

[11]:
```
# testing loop for evaluation
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28) #.to(device)
        labels = labels #.to(device)
        outputs = model(images)

        # return value and index
        _, predictions = torch.max(outputs, 1)
```

```
        n_samples += labels.shape[0]
        n_correct += (predictions == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'accuracy = {acc}')
```

```
accuracy = 72.22
```