Project 4 Writeup

Yiren Lu
50292358

November 20, 2019

Introduction

In project4, we are going to construct a 3 layer neural network to do postcode classification based on MNIST dataset.

Implementation Details

Hyper Parameters

```
# Hyper Parameters
EPOCH = 5
BATCH_SIZE = 100
LR = 0.001
```

First, the whole network consists of 2 convolution layer and a fully connected layer.

```
class CNN(nn.Module):

  def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 16, 7, 1, 3)
    self.conv2 = nn.Conv2d(16, 16, 7, 1, 3)
    self.fc = nn.Linear(16 * 28 * 28, 10)


  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.relu(self.conv2(x))
    x = x.view(-1, self.num_flat_features(x))
    x = self.fc(x)
    return x

  def num_flat_features(self, x):
    size = x.size()[1:]
        num_features = 1
```

```
        for s in size:
        num_features *= s
        return num_features
```

Prepare training and testing data

```
# prepare training and testing data
train_data = torchvision.datasets.MNIST(
        root='./mnist/',
        train=True,
        transform=torchvision.transforms.ToTensor(),
        download=False,
)

train_loader = torch.utils.data.DataLoader(
        dataset=train_data,
        batch_size=BATCH_SIZE,
        shuffle=True
)

test_data = torchvision.datasets.MNIST(
        root='./mnist/',
        train=False,
        transform=torchvision.transforms.ToTensor()
)

test_loader = torch.utils.data.DataLoader(
        dataset=test_data,
        batch_size=BATCH_SIZE,
        shuffle=True
)
```

Train the neural network and save it to local

```
def train_and_save():
  cnn = CNN()
  #print(cnn)

  optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
  loss_func = nn.CrossEntropyLoss()

  # training and testing
  for epoch in range(EPOCH):
  for batch, (b_x, b_y) in enumerate(train_loader):
  # cnn output
  output = cnn(b_x)
  # cross entropy loss
```

```
    loss = loss_func(output, b_y)
    # clear gradients for this training batch
    optimizer.zero_grad()
    # backpropagation, compute gradients
    loss.backward()
    # apply gradients
    optimizer.batch()

    torch.save(cnn, 'cnn_1(kernel_16_7_16_7_epoch5).pkl')
    print('save complete')
```

Load the model and evaluate

```
def evaluation():
    cnn = torch.load('cnn_1(kernel_16_7_16_7_epoch5).pkl')
    print("load complete")

    total = 0
    wrong = 0
    for data, target in test_loader:
    #print(target)
    test_output = cnn(data)

    pred_y = torch.max(test_output, 1)[1]
    pred_y = pred_y.data.numpy().squeeze()

    for i in range(pred_y.shape[0]):
    total += 1
    if pred_y[i] != target[i]:
    wrong += 1

    return wrong/total
```

Experiments & Results

Here are my loss of prediction.

As the structure of the whole network is fixed, what we can do is to achieve a higher performance in a limited 3 layer. I tried 3 different size of kernels: 5*5, 7*7 and 9*9.

```
PS C:\Users\luyr\Desktop\Proj4_陆弈人> python .\CNN.py
load complete
loss with 7*7 kernel and epoch=5:  0.0112
PS C:\Users\luyr\Desktop\Proj4_陆弈人> python .\CNN.py
load complete
loss with 5*5 kernel and epoch=1:  0.0189
PS C:\Users\luyr\Desktop\Proj4_陆弈人> python .\CNN.py
load complete
loss with 7*7 kernel and epoch=1:  0.0149
PS C:\Users\luyr\Desktop\Proj4_陆弈人> python .\CNN.py
load complete
loss with 9*9 kernel and epoch=1:  0.0147
```

Discussions

1. As can be seen in the result above, more iteration will result in a lower loss and higher accuracy.

2. As the number of layers is fixed, a larger kernel will give us a larger receptive field. So the 9*9 kernel has the best performance.