AI_Assgn_2_Q3

November 10, 2021

1 Import the packages

[ ]:
```
import torch
import torchvision
from torch import nn, optim
from torchsummary import summary
```

2 Declare variables for the CNN

- **Epoch** is the number of passes of the entire training dataset through the neural network. A pair of forward and backward propagation indicates a single pass.
- **Batch Size** is the number of samples to work through before updating the weights and biases associated with the model.
- **Learning Rate** controls how much to change the model parameters in response to the prediction error each time the model weights are updated.

[ ]:
```
batch_size = 32
epoch = 30
learning_rate = 0.01
```

3 Load the training set and validation set using Dataset and DataLoader

[ ]:
```
trans = torchvision.transforms.ToTensor()
train_data = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST(
    'mnist_data', train=True, download=True, transform=trans
    ), batch_size=batch_size
    )
val_data = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST(
    'mnist_data', train=False, download=True, transform=trans
    ), batch_size=batch_size)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
mnist_data/MNIST/raw/train-images-idx3-ubyte.gz
  0%|          | 0/9912422 [00:00<?, ?it/s]


Extracting mnist_data/MNIST/raw/train-images-idx3-ubyte.gz to
mnist_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
mnist_data/MNIST/raw/train-labels-idx1-ubyte.gz
  0%|          | 0/28881 [00:00<?, ?it/s]


Extracting mnist_data/MNIST/raw/train-labels-idx1-ubyte.gz to
mnist_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
mnist_data/MNIST/raw/t10k-images-idx3-ubyte.gz
  0%|          | 0/1648877 [00:00<?, ?it/s]


Extracting mnist_data/MNIST/raw/t10k-images-idx3-ubyte.gz to
mnist_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
mnist_data/MNIST/raw/t10k-labels-idx1-ubyte.gz
  0%|          | 0/4542 [00:00<?, ?it/s]


Extracting mnist_data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
mnist_data/MNIST/raw


/usr/local/lib/python3.7/dist-packages/torchvision/datasets/mnist.py:498:
UserWarning: The given NumPy array is not writeable, and PyTorch does not
support non-writeable tensors. This means you can write to the underlying
(supposedly non-writeable) NumPy array using the tensor. You may want to copy
the array to protect its data or make it writeable before converting it to a
tensor. This type of warning will be suppressed for the rest of this program.
(Triggered internally at  /pytorch/torch/csrc/utils/tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

4 Define the CNN using ReLU function for image classification

[ ]:
```
class ConvNet(nn.Module):
  def __init__(self):
    super(ConvNet, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=3,␣
 ↪stride=1, padding=1)
    self.conv2 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3,␣
 ↪stride=1)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    self.relu = nn.ReLU()
    self.linear1 = nn.Linear(6*6*6, 10)
  def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.pool(x)
    x = self.relu(self.conv2(x))
    x = self.pool(x)
    x = x.view(x.shape[0], -1)
    x = self.linear1(x)
    return x
```

5 Define a function for validating the model

[ ]:
```
def validate(model, data):
  total = 0
  correct = 0
  for i, (images, labels) in enumerate(data):
    images = images.cuda()
    labels = labels.cuda()
    y_pred = model(images)
    value, pred = torch.max(y_pred, 1)
    total += y_pred.size(0)
    correct += torch.sum(pred == labels)
  return correct * 100 / total
```

6 Initialize the neural network and optimizer

[ ]:
```
convnet = ConvNet().cuda()
optimizer = optim.Adam(convnet.parameters(), lr=learning_rate)
cross_entropy = nn.CrossEntropyLoss()
```

7 Print the Model Summary

[ ]:
```
summary(convnet, (1, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 3, 224, 224]              30
              ReLU-2          [-1, 3, 224, 224]               0
         MaxPool2d-3          [-1, 3, 112, 112]               0
            Conv2d-4          [-1, 6, 110, 110]             168
              ReLU-5          [-1, 6, 110, 110]               0
         MaxPool2d-6            [-1, 6, 55, 55]               0
            Linear-7                  [-1, 10]           2,170
================================================================
Total params: 2,368
Trainable params: 2,368
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 3.83
Params size (MB): 0.01
Estimated Total Size (MB): 4.03
----------------------------------------------------------------

/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning:
Named tensors and all their associated APIs are an experimental feature and
subject to change. Please do not use them for anything important until they are
released as stable. (Triggered internally at
/pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
ceil_mode)
```

8 Display the validation accuracy on each epoch

[ ]:
```
for n in range(epoch):
  for i, (images, labels) in enumerate(train_data):
    images = images.cuda()
    labels = labels.cuda()
    optimizer.zero_grad()
    prediction = convnet(images)
    loss = cross_entropy(prediction, labels)
    loss.backward()
    optimizer.step()
  accuracy = float(validate(convnet, val_data))
  print("Epoch:", n+1, "Loss: ", float(loss.data), "Accuracy:", accuracy)
```

```
Epoch: 1 Loss:  0.0835513323545456 Accuracy: 88.06999969482422
Epoch: 2 Loss:  0.1373533010482788 Accuracy: 89.77999877929688
Epoch: 3 Loss:  0.23270569741725922 Accuracy: 90.91999816894531
Epoch: 4 Loss:  0.2337682545185089 Accuracy: 93.18999481201172
Epoch: 5 Loss:  0.08620554953813553 Accuracy: 94.07999420166016
Epoch: 6 Loss:  0.05838495492935181 Accuracy: 94.33999633789062
Epoch: 7 Loss:  0.10215658694505692 Accuracy: 94.7699966430664
Epoch: 8 Loss:  0.10404697060585022 Accuracy: 94.72000122070312
Epoch: 9 Loss:  0.09468015283346176 Accuracy: 94.3499984741211
Epoch: 10 Loss:  0.10854273289442062 Accuracy: 94.18999481201172
Epoch: 11 Loss:  0.1658000349998474 Accuracy: 93.55999755859375
Epoch: 12 Loss:  0.18976950645446777 Accuracy: 93.3699951171875
Epoch: 13 Loss:  0.1318638026714325 Accuracy: 94.32999420166016
Epoch: 14 Loss:  0.11216580867767334 Accuracy: 94.8499984741211
Epoch: 15 Loss:  0.09310924261808395 Accuracy: 94.91999816894531
Epoch: 16 Loss:  0.0903550460934639 Accuracy: 95.2699966430664
Epoch: 17 Loss:  0.12142740190029144 Accuracy: 95.18000030517578
Epoch: 18 Loss:  0.18455654382705688 Accuracy: 93.91999816894531
Epoch: 19 Loss:  0.10916927456855774 Accuracy: 95.55999755859375
Epoch: 20 Loss:  0.1532885730266571 Accuracy: 94.79000091552734
Epoch: 21 Loss:  0.11319854110479355 Accuracy: 95.40999603271484
Epoch: 22 Loss:  0.1639622151851654 Accuracy: 94.68999481201172
Epoch: 23 Loss:  0.23376189172267914 Accuracy: 93.2699966430664
Epoch: 24 Loss:  0.19606859982013702 Accuracy: 92.93999481201172
Epoch: 25 Loss:  0.23199202120304108 Accuracy: 91.88999938964844
Epoch: 26 Loss:  0.19728757441043854 Accuracy: 93.12999725341797
Epoch: 27 Loss:  0.1515316665172577 Accuracy: 92.97999572753906
Epoch: 28 Loss:  0.15251077711582184 Accuracy: 92.64999389648438
Epoch: 29 Loss:  0.10096003115177155 Accuracy: 93.61000061035156
Epoch: 30 Loss:  0.11824318021535873 Accuracy: 91.8499984741211
```

9 Observations

Using ReLU as an Activation function increased the accuracy of the Convolutional Neural Networks.

This is because **non-saturation of its gradient**, which greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid / tanh functions