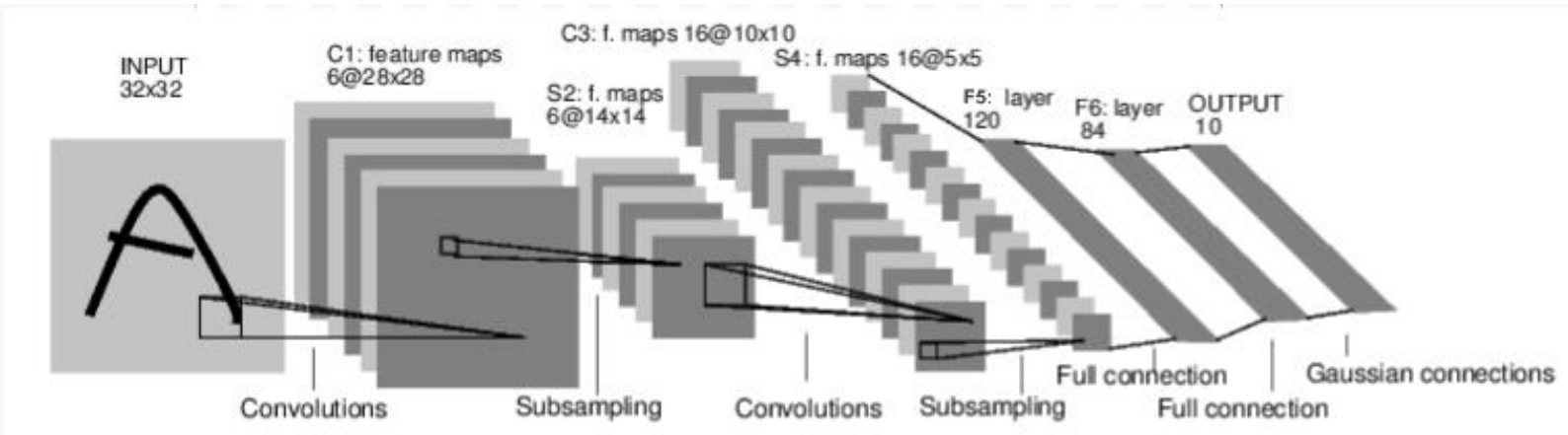
CS405 Machine Learning

Lab #6 Neural Network

(100 points)

1. Introduction

PyTorch is an open source machine learning framework that commonly used for research prototyping and production deployment. In this lab, we will try to train a image classifier using neural networks(NN) under PyTorch. A simple NN as below can be constructed using the *torch.nn* package



It is a simple feed-forward network.

It takes the input, feeds it through several layers one after the other, and then finally gives the output.

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)

- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: weight = weight - learning_rate * gradient

2. Define the network

In this section, we will present an example about how to define a neural network.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Fig. 1

Here, we just have to define the *forward* function, and the *backward* function (where gradients are computed) will be automatically defined using autograd. You can use any of the Tensor operations in the *forward* function.

The learnable parameters of a model are returned by *net.parameters()*

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

```
10
torch.Size([6, 1, 3, 3])
```

Fig. 2

Then, we can try to input a random 32*32 data, Zero the gradient buffers of all parameters and backprops with random gradients:

```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[-0.0685, -0.0604,  0.0296, -0.1328,  0.1131,  0.0436, -0.0095,  0.1085,
         0.1188,  0.0949]], grad_fn=<AddmmBackward>)
```

Fig. 3

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

Fig. 4

At this point, we covered:

- Defining a neural network
- Processing inputs and calling backward

Still Left:

- Computing the loss
- Updating the weights of the network

3. Loss Function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

There are several different loss functions under the *nn* package . A simple loss is: *nn.MSELoss* which computes the mean-squared error between the input and the target.

For example:

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(0.8297, grad_fn=<MseLossBackward>)
```

Fig. 5

4. Backprop

To backpropagate the error all we have to do is to use *loss.backward()*. You need to clear the existing gradients though, else gradients will be accumulated to existing gradients. Now we shall call *loss.backward()*, and have a look at conv1's bias gradients before and after the backward.

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([-0.0113,  0.0111, -0.0221, -0.0237,  0.0054, -0.0059])
```

Fig. 6

5. Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We can implement this with very simple python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

Fig. 7

However, under PyTorch framework, various of different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc are implemented. To use these rules, we can use a small package call *torch.optim* .

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Fig. 8

Exercise 01:

Follow the Image Classifier Training tutorial of PyTorch(1) to train your own image classifier.

6. Lab requirement

Please answer the Exercise 01 questions and wright a report.

7. References

(1) https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py