Coding CNNs from Scratch with Pytorch

In this assignment you will code a famous CNN architecture AlexNet (https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html) to classify images from the CIFAR10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html), which consists of 10 classes of natural images such as vehicles or animals. AlexNet is a landmark architecture because it was one of the first extremely deep CNNs trained on GPUs, and achieved state-of-the-art performance in the ImageNet challenge in 2012.

A lot of code will already be written to familiarize yourself with PyTorch, but you will have to fill in parts that will apply your knowledge of CNNs. Additionally, there are some numbered questions that you must answer either in a separate document, or in this notebook. Some questions may require you to do a little research. To type in the notebook, you can insert a text cell.

Let's start by installing PyTorch and the torchvision package below. Due to the size of the network, you will have to run on a GPU. So, click on the Runtime dropdown, then Change Runtime Type, then GPU for the hardware accelerator.

```
!pip install pytorch
!pip install torchvision
```

```
Collecting pytorch
  Downloading pytorch-1.0.2.tar.gz (689 bytes)
Building wheels for collected packages: pytorch
  Building wheel for pytorch (setup.py) ... error
  ERROR: Failed building wheel for pytorch
  Running setup.py clean for pytorch
Failed to build pytorch
Installing collected packages: pytorch
    Running setup.py install for pytorch ... error
ERROR: Command errored out with exit status 1: /usr/bin/python3 -u -c 'import io, os, sy
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages (0
Requirement already satisfied: pillow!=8.3.0,>=5.3.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: torch==1.10.0 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from tor
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packag
```

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

```
device = torch.device('cuda')

print(device)

    cuda
```

1. In the following cell, we are employing something called "data augmentation" with random horizontal and vertical flips. So when training data is fed into the network, it is ranadomly transformed. What are advantages of this?

1.increase dataset While we are training a CNN model with high accuracy and good generalization, the most important things are that we need enough data to train the model. However, collecting data such as image recognition is hard to have thousands or millions of data. Therefore, we can rotate, flip, cut, add noise, whiten, and change color to increase dataset.

2.Avoid overfitting When training a classifier, it should be easy to encounter overfitting. Therefore, with utilizing data augmentation, we can reduce the possibility of overfitting.

Data augmentation is a commonly used image preprocessing, but we should avoid some wrong situation. For example, when training digital models, if we use vertical flipping which will cause training problems between 6 and 9. Moreover, if the input image is in size of 256*256, augmentation is randomly cropped 128 pixels, it might importantly content and the model cannot learn useful information.

2. We normalize with the line transforms.Normalize((0.5,), (0.5,)). What are the benefits of normalizing data?

1.Reduce steps of gradient descent In many learning algorithms, the objective function is assumpted zero mean and have same order of squared differences. If the squared difference of a feature's orders of is larger than other features, then it will dominate the learning algorithm which will cause the model no learn from the other features as we say expected.

2.Increase accuracy For linear model, after normalize the data, the optimization process will become smoother, and it is faster to converge to the optimal solution. Therefore, the accuracy rate will increase.

```
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import random_split
from math import ceil
```

```
BATCH_SIZE = 1000


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.RandomHorizontalFlip(p=0.5),
     transforms.RandomVerticalFlip(p=0.5),
     transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

torch.manual_seed(43)
val_size = 10000
train_size = len(trainset) - val_size


train_ds, val_ds = random_split(trainset, [train_size, val_size])
print(len(train_ds), len(val_ds))


classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

num_steps =  ceil(len(train_ds) / BATCH_SIZE)
num_steps
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-
                                        170499072/? [00:06<00:00, 59300359.62it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
40000 10000
40
```

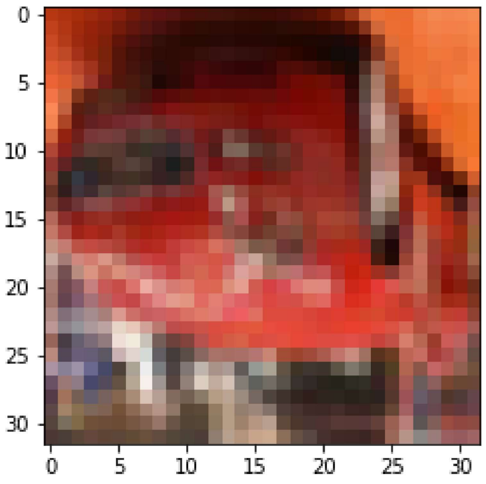
```
train_loader = torch.utils.data.DataLoader(train_ds, BATCH_SIZE, shuffle=True, drop_last = Tr
val_loader = torch.utils.data.DataLoader(val_ds, BATCH_SIZE)
test_loader = torch.utils.data.DataLoader(testset, BATCH_SIZE)
```

You can insert an integer into the code trainset[#insert integer] to visualize images from the training set. Some of the images might look weird because they have been randomly flipped according to our data augmentation scheme.

```
img, label = trainset[5]#insert integer]
plt.imshow((img.permute((1, 2, 0))+1)/2)
```

```
print('Label (numeric):', label)
print('Label (textual):', classes[label])
```

```
    Label (numeric): 1
    Label (textual): car
```



Now comes the fun part. You will have to put in the correct parameters into different torch.nn functions in order to convolve and downsample the image into the correct dimensionality for classification. Think of it as a puzzle. You will insert the parameters where there is a comment #TODO.

```
class Discriminator(torch.nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels = 3, #TODO,
                      out_channels = 64, #TODO,
                      kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3,
                      padding=1),#TODO),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192,#TODO,
                      384,#TODO,
                      kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )
        #Fully connected layers
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(1024,4096),#TODO),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),#TODO),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 10),#TODO),
        )

    def forward(self, x):
        x = self.features(x)
        #we must flatten our feature maps before feeding into fully connected layers
        x = x.contiguous().view(x.size(0), -1)#TODO)
        x = self.classifier(x)
        return x
```

Below we are initializing our model with a weight scheme.

```
net = Discriminator()

def weights_init(m):

    classname = m.__class__.__name__

    if classname.find('Conv') != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)

    elif classname.find('BatchNorm') != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0)


# Initialize Models
net = net.to(device)

net.apply(weights_init)
```

```
Discriminator(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
```

```
      (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (9): ReLU(inplace=True)
      (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): Dropout(p=0.5, inplace=False)
      (1): Linear(in_features=1024, out_features=4096, bias=True)
      (2): ReLU(inplace=True)
      (3): Dropout(p=0.5, inplace=False)
      (4): Linear(in_features=4096, out_features=4096, bias=True)
      (5): ReLU(inplace=True)
      (6): Linear(in_features=4096, out_features=10, bias=True)
    )
  )
```

3. Notice above in our network architecture, we have what are called "Dropout" layers. What is the point of these?

Dropout is a method that randomly selecting neurons not participate in training. Therfore, the model does not rely too much on certain features and enhance the generalization of the model. These method is commonly used in machine learning to prevent overfitting.

Defining our cost/loss function, which is cross-entropy loss. We also define our optimizer with hyperparameters: learning rate and betas.

```
import torch.optim as optim

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    net.parameters(),
    lr=0.0003,
    betas = (0.5, 0.999)
)
```

Below we actually train our network. Run for just 10 epochs. It takes some time. Wherever there is the comment #TODO, you must insert code.

```
for epoch in range(50):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)#TODO

        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)#TODO     #pass input data into network to get outputs
        loss = criterion(outputs, labels)#TODO)
        loss.backward()  #calculate gradients
        optimizer.step() #take gradient descent step


        running_loss += loss.item()


    print("E:{}, Train Loss:{}".format(
                epoch+1,
                running_loss / num_steps
            )
        )




    #validation
    correct = 0
    total = 0
    val_loss = 0.0
    with torch.no_grad():
        for data in val_loader:
            images, labels = data
            #TODO: load images and labels from validation loader
            images = images.to(device)
            labels = labels.to(device)
            outputs = net(images)#TODO  #run forward pass
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            loss = criterion(outputs, labels)#TODO       #calculate validation loss
            val_loss += loss.item()
    val_loss /=num_steps
    print('Accuracy of 10000 val images: {}'.format( correct / total))
    print('Val Loss: {}'.format( val_loss))

print('Finished Training')
```

```
E:32, Train Loss:0.6306983558688165
Accuracy of 10000 val images: 0.6685
Val Loss: 0.24880201518535613
E:33, Train Loss:0.6070372790098191
Accuracy of 10000 val images: 0.6866
```

```
Val Loss: 0.2346355676651001
E:34, Train Loss:0.5837077304720879
Accuracy of 10000 val images: 0.675
Val Loss: 0.23879897892475127
E:35, Train Loss:0.5927547723054886
Accuracy of 10000 val images: 0.6712
Val Loss: 0.243065445125103
E:36, Train Loss:0.5745449908077717
Accuracy of 10000 val images: 0.6695
Val Loss: 0.256869874894619
E:37, Train Loss:0.5452909663319587
Accuracy of 10000 val images: 0.6632
Val Loss: 0.25845731645822523
E:38, Train Loss:0.5402073502540589
Accuracy of 10000 val images: 0.6834
Val Loss: 0.2491868942975998
E:39, Train Loss:0.5211011230945587

Accuracy of 10000 val images: 0.6827
Val Loss: 0.25029719471931455
E:40, Train Loss:0.5353323854506016
Accuracy of 10000 val images: 0.6837
Val Loss: 0.24668078571558
E:41, Train Loss:0.4941793270409107
Accuracy of 10000 val images: 0.677
Val Loss: 0.24830785840749742
E:42, Train Loss:0.49641330912709236
Accuracy of 10000 val images: 0.676
Val Loss: 0.252286022901535
E:43, Train Loss:0.4649282030761242
Accuracy of 10000 val images: 0.6625
Val Loss: 0.2712598517537117
E:44, Train Loss:0.4716526098549366
Accuracy of 10000 val images: 0.6857
Val Loss: 0.24938022792339326
E:45, Train Loss:0.5181257799267769
Accuracy of 10000 val images: 0.6732
Val Loss: 0.25867445319890975
E:46, Train Loss:0.4244796626269817
Accuracy of 10000 val images: 0.6846
Val Loss: 0.2591854125261307
E:47, Train Loss:0.40056391954422
Accuracy of 10000 val images: 0.6863
Val Loss: 0.2622984886169434
E:48, Train Loss:0.39125835299491885
Accuracy of 10000 val images: 0.6827
Val Loss: 0.2698840230703354
E:49, Train Loss:0.4045791484415531
Accuracy of 10000 val images: 0.6355
Val Loss: 0.3111805945634842
E:50, Train Loss:0.3853556074202061
Accuracy of 10000 val images: 0.6912
Val Loss: 0.2618514895439148
Finished Training
```

4. If we train for more epochs, our accuracy/performance will increase. What happens if we train for too long though? What method can be employed to mitigate this?

If we train more epochs, the accuracy will increase. We can add more hidden layer to increase the accuracy, and we can add some dropout layer to reduce overfitting. However, we cannot train for more epoch, and we need to choose moderate epoch size so that the training model will not be overfitting and achieve high accuracy.

5. Try increasing learning rate and look at the metrics for training and validation data? What do you notice? Why do think this is happening?

If I increase the learning rate, the validation loss will increase and cannot converge. The reason why it will not converge is that the step is too big for gradient descent and it won't find the minimum point so that the value will vibrate or even become bigger. Therefore, we should choose small learning rate for training model and observe the validation loss if it converges.

We can see the performance on the testing set now.

```
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of 10000 test images: {}'.format( correct / total))
```

```
Accuracy of 10000 test images: 0.6885
```

✓ 4s completed at 7:53 PM