EE435_PyTorch_Assignment

May 18, 2024

1 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.

[1]:
```
!pip install pytorch
!pip install torchvision
```

```
Collecting pytorch
  Downloading pytorch-1.0.2.tar.gz (689 bytes)
  Preparing metadata (setup.py) … done
Building wheels for collected packages: pytorch
  error: subprocess-exited-with-error

  × python setup.py bdist_wheel did not run successfully.
   exit code: 1
   > See above for output.

  note: This error originates from a subprocess, and is likely not a
problem with pip.
  Building wheel for pytorch (setup.py) … error
  ERROR: Failed building wheel for pytorch

  Running setup.py clean for pytorch
Failed to build pytorch
```

```
ERROR: Could not build wheels for pytorch, which is required to install
pyproject.toml-based projects
Requirement already satisfied: torchvision in
/usr/local/lib/python3.10/dist-packages (0.17.1+cu121)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from torchvision) (1.25.2)
Requirement already satisfied: torch==2.2.1 in /usr/local/lib/python3.10/dist-
packages (from torchvision) (2.2.1+cu121)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.10/dist-packages (from torchvision) (9.4.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch==2.2.1->torchvision) (3.14.0)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.10/dist-packages (from torch==2.2.1->torchvision)
(4.11.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch==2.2.1->torchvision) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch==2.2.1->torchvision) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch==2.2.1->torchvision) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch==2.2.1->torchvision) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch==2.2.1->torchvision)
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(23.7 MB)
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch==2.2.1->torchvision)
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(823 kB)
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch==2.2.1->torchvision)
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(14.1 MB)
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch==2.2.1->torchvision)
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7
MB)
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch==2.2.1->torchvision)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6
MB)
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch==2.2.1->torchvision)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6
MB)
Collecting nvidia-curand-cu12==10.3.2.106 (from torch==2.2.1->torchvision)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl
(56.5 MB)
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch==2.2.1->torchvision)
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl
```

```
(124.2 MB)
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch==2.2.1->torchvision)
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl
(196.0 MB)
Collecting nvidia-nccl-cu12==2.19.3 (from torch==2.2.1->torchvision)
  Using cached nvidia_nccl_cu12-2.19.3-py3-none-manylinux1_x86_64.whl (166.0 MB)
Collecting nvidia-nvtx-cu12==12.1.105 (from torch==2.2.1->torchvision)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
Requirement already satisfied: triton==2.2.0 in /usr/local/lib/python3.10/dist-
packages (from torch==2.2.1->torchvision) (2.2.0)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-
cu12==11.4.5.107->torch==2.2.1->torchvision)
  Using cached nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl
(21.1 MB)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch==2.2.1->torchvision)
(2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch==2.2.1->torchvision) (1.3.0)
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-
nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu12,
nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-
cusparse-cu12, nvidia-cudnn-cu12, nvidia-cusolver-cu12
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-
cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime-cu12-12.1.105
nvidia-cudnn-cu12-8.9.2.26 nvidia-cufft-cu12-11.0.2.54 nvidia-curand-
cu12-10.3.2.106 nvidia-cusolver-cu12-11.4.5.107 nvidia-cusparse-cu12-12.1.0.106
nvidia-nccl-cu12-2.19.3 nvidia-nvjitlink-cu12-12.4.127 nvidia-nvtx-cu12-12.1.105
```

[44]:
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt


device = torch.device('cuda')

print(device)
```

```
cuda
```

1.0.1 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?

This reduces overfitting of data by exposing it to additional variations of the same image. This assumes that we can get more information from the original dataset by using augmentation which can also be helpful if we are training with a limited dataset. This overall improves the model accuracy.

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

Normalization can help training of our neural networks as this step gets the different features are on a similar scale, which helps to stabilize the gradient descent step, allowing the use of larger learning rates or helping models converge faster for a given learning rate.

[45]:
```
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import random_split
from math import ceil

BATCH_SIZE = 100


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
```

```
Files already downloaded and verified
Files already downloaded and verified
40000 10000
```

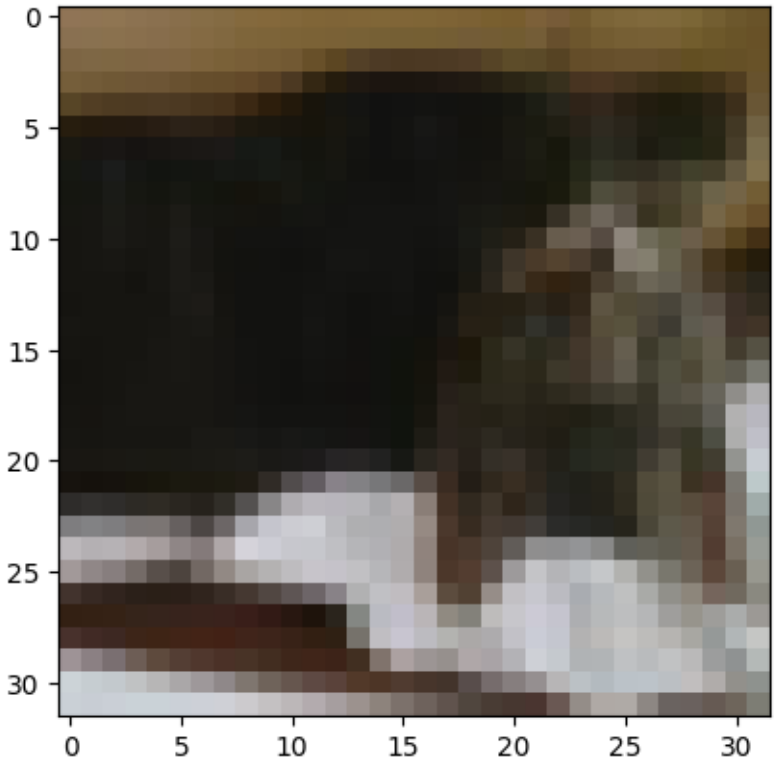
[45]:
```
400
```

[46]:
```
train_loader = torch.utils.data.DataLoader(train_ds, BATCH_SIZE, shuffle=True,␣
↪drop_last = True)
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.

[47]:
```
img, label = trainset[9]
plt.imshow((img.permute((1, 2, 0))+1)/2)
print('Label (numeric):', label)#`vbn. z wertjkwertyu;
print('Label (textual):', classes[label])
```

```
Label (numeric): 3
Label (textual): cat
```



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.

[48]:
```
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(4096,100), #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.

[49]:
```
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)
```

[49]:
```
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=100, bias=True)
  )
```

```
)
```

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

Dropout layers are adapted to prevent the issue of overfitting which is where the model may learn statistical noise. Without dropouts, units may learn to fix mistakes of other units thereby creating co-adaptations leading to our model not getting generalized. Addition of dropout layers forces layers to take more or less responsibility for the inputs with a probabilistic approach therebt making the model more general resulting in a better performance with test dataset it has never seen before.

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

[50]:
```
import torch.optim as optim

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    net.parameters(),
    lr=0.0002,
    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.

[53]:
```
for epoch in range(10):  # 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)#TODDO     #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:
            #TODO: load images and labels from validation loader
            images, labels = data
            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:1, Train Loss:1.4220407459139823
Accuracy of 10000 val images: 0.5092
Val Loss: 0.3378035110235214
E:2, Train Loss:1.2951746878027917
Accuracy of 10000 val images: 0.5216
Val Loss: 0.32890733271837236
E:3, Train Loss:1.1976687783002853
Accuracy of 10000 val images: 0.5603
Val Loss: 0.3034729914367199
E:4, Train Loss:1.1180827513337135
Accuracy of 10000 val images: 0.5948
Val Loss: 0.2820093309879303
E:5, Train Loss:1.0553837290406227
Accuracy of 10000 val images: 0.6151
Val Loss: 0.26997333616018293
```

```
E:6, Train Loss:0.9979504333436489
Accuracy of 10000 val images: 0.6319
Val Loss: 0.2598070977628231
E:7, Train Loss:0.9434045785665512
Accuracy of 10000 val images: 0.6278
Val Loss: 0.256942468136549
E:8, Train Loss:0.891601822078228
Accuracy of 10000 val images: 0.6537
Val Loss: 0.2474515427649021
E:9, Train Loss:0.8506807227432728
Accuracy of 10000 val images: 0.6483
Val Loss: 0.25523970171809196
E:10, Train Loss:0.8076005019247532
Accuracy of 10000 val images: 0.6567
Val Loss: 0.2526149518787861
Finished Training
```

2.1 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 for too long, we again run into the issue of overfitting. The network could end up hard encoding some biases and weights by learning the training data to the point it stops being able to generalize and ends up performing poorly in the validation tests. The data augmentation and dropout method we used are few of the tools used to counter overfitting. Another thing we could do is monitor the accuracy in both testing and training data as a function of epochs and see that it performs well in both. We could also set an upper bound on the number of epochs once a desired value of accuracy is reached. If not, we might have to readjust our model, maybe relying on an ensemble of models so there is less chance of hard encoding some of the biases into one specific model.

2.2 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?

We can see the performance on the testing set now.

Changing learning rate from 0.0002 to 0.01.

Since ADAM computes individual adaptive learning rates with the learning rate sent while calling the function as the bound, we want to start with the smallest possible learning rate and increase it while the validation loss as a function of learning rate decreses. The optimal learning rate would be a rate a little below the point the validation loss starts climbing up with increase in learning rate.

In the run below with learning rate 0.01, we see that the rate is too high and since the step size is too large, the model might be missing the points at which the cost is minimized. Since the learning rate is already too high, one way to find the optimal learning rate would be to restart the model, and look for the optimal value we just discussed about by plotting validation loss as a function of learning rate.

[67]:
```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    net.parameters(),
    lr=0.01,
    betas = (0.5, 0.999)
)
```

[68]:
```
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: 0.1
```

[60]:
```
for epoch in range(10):  # 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)#TODDO     #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:
            #TODO: load images and labels from validation loader
            images, labels = data
            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:1, Train Loss:71.25373828664422
Accuracy of 10000 val images: 0.1589
Val Loss: 0.5949645668268204
E:2, Train Loss:2.192418440878391
Accuracy of 10000 val images: 0.1903
Val Loss: 0.5358748480677604
E:3, Train Loss:2.2002488535642626
Accuracy of 10000 val images: 0.2025
Val Loss: 0.525718737244606
E:4, Train Loss:2.0801099979877473
Accuracy of 10000 val images: 0.1983
Val Loss: 0.5208989408612251
E:5, Train Loss:2.0226137349009514
Accuracy of 10000 val images: 0.1658
Val Loss: 0.5497078382968903
E:6, Train Loss:3.3257417610287665
Accuracy of 10000 val images: 0.1022
Val Loss: 0.5867826229333878
```

```
E:7, Train Loss:2.3256529533863066
Accuracy of 10000 val images: 0.0959
Val Loss: 0.5786735653877259
E:8, Train Loss:2.3115513783693316
Accuracy of 10000 val images: 0.0956
Val Loss: 0.5797243237495422
E:9, Train Loss:2.308466332554817
Accuracy of 10000 val images: 0.0966
Val Loss: 0.5768873572349549
E:10, Train Loss:2.3086473774909972
Accuracy of 10000 val images: 0.0966
Val Loss: 0.5764306473731995
Finished Training
```

[ ]: