Duckietown_dt_intro_nn_with_pytorch

December 31, 2023

1 Duckietown Intro to Neural Nets with PyTorch

1.1 Prerequisites

Step 1: install pytorch.

Important: the following cell is only needed for this notebook. If you're installing pytorch on your own computer (which you totally can, even without a NVIDIA GPU), then please use the instructions from here: https://pytorch.org/get-started/locally/

Note that this will take a few seconds.

[ ]:
```
# http://pytorch.org/
!pip install torch==1.13.0 torchvision==0.14.0
```

```
Collecting torch==1.13.0
  Downloading torch-1.13.0-cp310-cp310-manylinux1_x86_64.whl (890.1 MB)
                             890.1/890.1

MB 980.6 kB/s eta 0:00:00
Collecting torchvision==0.14.0
  Downloading torchvision-0.14.0-cp310-cp310-manylinux1_x86_64.whl (24.3 MB)
                             24.3/24.3 MB
40.5 MB/s eta 0:00:00
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from torch==1.13.0) (4.5.0)
Collecting nvidia-cuda-runtime-cu11==11.7.99 (from torch==1.13.0)
  Downloading nvidia_cuda_runtime_cu11-11.7.99-py3-none-manylinux1_x86_64.whl
(849 kB)
                             849.3/849.3

kB 62.5 MB/s eta 0:00:00
Collecting nvidia-cudnn-cu11==8.5.0.96 (from torch==1.13.0)
  Downloading nvidia_cudnn_cu11-8.5.0.96-2-py3-none-manylinux1_x86_64.whl (557.1
MB)
                             557.1/557.1

MB 3.0 MB/s eta 0:00:00
Collecting nvidia-cublas-cu11==11.10.3.66 (from torch==1.13.0)
  Downloading nvidia_cublas_cu11-11.10.3.66-py3-none-manylinux1_x86_64.whl
(317.1 MB)
```

```
                                        317.1/317.1
MB 4.3 MB/s eta 0:00:00
Collecting nvidia-cuda-nvrtc-cu11==11.7.99 (from torch==1.13.0)
  Downloading nvidia_cuda_nvrtc_cu11-11.7.99-2-py3-none-manylinux1_x86_64.whl
(21.0 MB)
                                        21.0/21.0 MB
47.6 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-
packages (from torchvision==0.14.0) (1.23.5)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from torchvision==0.14.0) (2.31.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.10/dist-packages (from torchvision==0.14.0) (9.4.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from nvidia-cublas-cu11==11.10.3.66->torch==1.13.0) (67.7.2)
Requirement already satisfied: wheel in /usr/local/lib/python3.10/dist-packages
(from nvidia-cublas-cu11==11.10.3.66->torch==1.13.0) (0.42.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->torchvision==0.14.0)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->torchvision==0.14.0) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->torchvision==0.14.0)
(2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->torchvision==0.14.0)
(2023.11.17)
Installing collected packages: nvidia-cuda-runtime-cu11, nvidia-cuda-nvrtc-cu11,
nvidia-cublas-cu11, nvidia-cudnn-cu11, torch, torchvision
  Attempting uninstall: torch
    Found existing installation: torch 2.1.0+cu121
    Uninstalling torch-2.1.0+cu121:
      Successfully uninstalled torch-2.1.0+cu121
  Attempting uninstall: torchvision
    Found existing installation: torchvision 0.16.0+cu121
    Uninstalling torchvision-0.16.0+cu121:
      Successfully uninstalled torchvision-0.16.0+cu121
```

```
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
torchaudio 2.1.0+cu121 requires torch==2.1.0, but you have torch 1.13.0 which is
incompatible.
torchdata 0.7.0 requires torch==2.1.0, but you have torch 1.13.0 which is
incompatible.
torchtext 0.16.0 requires torch==2.1.0, but you have torch 1.13.0 which is
incompatible.
Successfully installed nvidia-cublas-cu11-11.10.3.66 nvidia-cuda-nvrtc-
cu11-11.7.99 nvidia-cuda-runtime-cu11-11.7.99 nvidia-cudnn-cu11-8.5.0.96
torch-1.13.0 torchvision-0.14.0
```

Step 2: import and check that torch works

[ ]:
```
import torch

torch.rand(2)
```

[ ]:
```
tensor([0.5343, 0.1011])
```

Step 3: Convert from numpy to torch and back

[ ]:
```
import numpy as np

a_npy = np.eye(4)
print (a_npy)

a_tor = torch.from_numpy(a_npy) # convert numpy to torch Tensor
print (a_tor)

b_npy = a_tor.numpy() # convert Tensor back to numpy
print (b_npy)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], dtype=torch.float64)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
```

```
[0. 0. 0. 1.]]
```

Step 4: Notice that if you change the data EITHER in numpy or torch, you're changing it in both

[ ]:
```
a_npy[0,3] = 99 # we're only changing the numpy array

a_tor[1,3] = 123 # we're only changing the torch Tensor

print (a_npy) # no we're not
print (a_tor)
print (b_npy)
```

```
[[  1.   0.   0.  99.]
 [  0.   1.   0. 123.]
 [  0.   0.   1.   0.]
 [  0.   0.   0.   1.]]
tensor([[  1.,   0.,   0.,  99.],
        [  0.,   1.,   0., 123.],
        [  0.,   0.,   1.,   0.],
        [  0.,   0.,   0.,   1.]], dtype=torch.float64)
[[  1.   0.   0.  99.]
 [  0.   1.   0. 123.]
 [  0.   0.   1.   0.]
 [  0.   0.   0.   1.]]
```

Step 5: Other than that, PyTorch is just like Numpy… but all the things are called different names

[ ]:
```
# These things do the same thing, but have different names and one is a␣
 ↪property,
# the other one is a function.

print (a_npy.shape)
print (a_tor.size())

# Most (not all) things that you can do in Numpy, you can also do in PyTorch,
# but you have to google what they are called
```

```
(4, 4)
torch.Size([4, 4])
```

1.2 Neural Nets

Step 6: This is how you define a neural net in PyTorch

IMPORTANT: read the comments. It's crucial that you understand what's happening. If something is unclear, ask on stack overflow with the label #neural-nets. It is easy to make simple mistakes when working with neural networks, so it is better to ask for help as early as possible.

[ ]:
```
import torch # we already did this, but just in case you want to
# copy-paste this code block
```

```
import torch.nn as nn # functions for neural nets, like layers and loss function
import torch.nn.functional as F #  these are helper functions liks sigmoid, relu, etc

# what's missing here is the import for the optimizer


# you need to create a class that inherits from nn.Module

class Net(nn.Module):
    def __init__(self):
        # Here in the init function you only create the layers that you want to
        # use later on. You're not actually connecting them to anything here.
        # So you can create them in any order.

        super(Net, self).__init__()

        # 3 color channels input, 6 convolutional kernels -> 6 channels output,
        # and also 5x5 square kernel
        self.conv1 = nn.Conv2d(3, 6, 5)
        # after applying this to a 3x32x32 image without padding or stride,
        # the result will be 6x28x28

        # max pooling with a 2x2 moving window
        self.pool = nn.MaxPool2d(2, 2)
        # after applying this to the 6x28x28 image, the result will be
        # 6x14x14 (the channels are not affected)

        # 6 channels input, 16 convolutional kernels -> 16 channels output,
        # and also 5x5 square kernel
        self.conv2 = nn.Conv2d(6, 16, 5)
        # after applying this to a 6x14x14 image without padding or stride,
        # the result will be 16x10x10

        # Later in the actual forward pass, we will apply the maxpooling twice,
        # but we only have to define it once, because it doesn't have any para-
        # meters that we're backpropping through.
        # So we know that we will apply MaxPool2d(2,2) again to the 16x10x10 image.
        # Therefore the output of the convolutional layers will be 16x5x5.


        # This layer definition requires that you did the convolution math.
        # The final "image" will be 5 by 5 pixels and 16 channels deep, therefore
        # the input is 16 * 5 * 5.
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 120)

        # The sizes of these layers are _completely_ arbitrary.
        self.fc2 = nn.Linear(120, 84)

        # ultimately our ouput will be a 10-element vector for each input image
        # which corresponds to a one-hot encoding of a 0-9 integer
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # This function is for a single forward pass through the network.
        # So you get the input, pass it through all the layers you defined above
        # (!important, don't define any new layers here) and then return the␣
↪final
        # result.

        # Apply, in that order: convolutional layer 1 (3,6,5), ReLU,␣
↪MaxPool2d(2,2)
        x = self.pool(F.relu(self.conv1(x)))

        # Apply, in that order: convolutional layer 2 (6,16,5), ReLU,␣
↪MaxPool2d(2,2)
        x = self.pool(F.relu(self.conv2(x)))

        # The input is still 3-dimensional (shape: 16x5x5). Here we transform it
        # into a vector of size (16*5*5 = 400)
        x = x.view(-1, 16 * 5 * 5)

        # Pass it through fully connected layer 1 and then through ReLU
        x = F.relu(self.fc1(x))

        # Pass it through fully connected layer 2 and then through ReLU
        x = F.relu(self.fc2(x))

        # Pass it through the last layer WITHOUT RELU and return it
        x = self.fc3(x)
        return x


# here we just instantiate the network, so we can go use it.
net = Net()

# and make sure it's using 32-bit floats ("Float"), not 64-bit floats ("Double")
net = net.float()
```

Step 7 (optional): print the network

[ ]:
```
print (net)
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, 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)
)
```

Step 8: Test the network with random input

[ ]:
```
# create a single image (note how the color channel is in the front)
dummy_input = np.random.uniform(low=0, high=1, size=(3,32,32)).astype(np.
 ↪float32)

# convert to Tensor:
dummy_tensor = torch.from_numpy(dummy_input)

print ("old size:", dummy_tensor.size())

# IMPORTANT. torch works in batches. You can have a batch size of 1 (if it's␣
 ↪only)
# a single image, but you need to add the dimension (which you want to become␣
 ↪the
# new axis 0):

dummy_tensor = dummy_tensor.unsqueeze(0)

print ("new size:", dummy_tensor.size(),"<--- see? There's a new first␣
 ↪dimension!")

# now we can feed it into the network:
prediction = net(dummy_tensor)

print ("prediction size:",prediction.size(),"<-- The output has the " \
       "same first dimension. That's the batch size!")

print ("")
print (prediction)
```

```
old size: torch.Size([3, 32, 32])
new size: torch.Size([1, 3, 32, 32]) <--- see? There's a new first dimension!
prediction size: torch.Size([1, 10]) <-- The output has the same first
dimension. That's the batch size!
```

```
tensor([[ 0.0138, -0.0613,  0.0471, -0.0862, -0.1121,  0.0693,  0.1024, -0.0680,
          0.0424,  0.0940]], grad_fn=<AddmmBackward0>)
```

Step 9: Start optimizing!

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

# instead of Mean Squared Error (MSE or "L2") loss we use CE loss here because
# this has great performance if your output is categorical and ideally in range
 ↪[0,1]
criterion = nn.CrossEntropyLoss()

# stochastic gradient descent... (There are better options)
# and feed the net.parameters() to the optimizer - that's all the optimizable
# parameters in the netwoprk
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Step 10: Do a trial run of the optimization

[ ]:
```
# let's stack ("concatenate" - "torch.cat()") 4 images into a minibatch:
inputs = torch.cat([dummy_tensor, dummy_tensor, dummy_tensor, dummy_tensor],
 ↪dim=0).float()
print ("inputs.size()", inputs.size())

# let's make some dummy labels - notice how it's only the batch dimension
labels = torch.zeros(4).long()
print ("labels.size()",labels.size())

# zero the parameter gradients (always do this during learning before every
 ↪forward pass)
optimizer.zero_grad()

# predict the outputs
outputs = net(inputs)
print ("outputs.size()", outputs.size())

# now we have a problem: our labels are unidimensional, but the prediction is
# 10-dimensional (on purpose)... what do?
# Answer: you can either spread out the ground truth into one-hot encoding
# Or: you can use a loss function that can accept both: CrossEntropy.

loss = criterion(outputs, labels) # apply the loss function, notice the format
 ↪of
# loss(prediction, ground_truth) <-- that's important

# calculate the backpropagation values
loss.backward()
```

```
# apply the backprop values according to the optimizer
optimizer.step()

# And print loss - very important - PLOT THIS! If this doesn't go a lot lower
# then you are done and the network is converged
print ("Loss:", loss.item())

# Now run this cell couple of times and watch the loss go down.
```

```
inputs.size() torch.Size([4, 3, 32, 32])
labels.size() torch.Size([4])
outputs.size() torch.Size([4, 10])
Loss: 2.2957229614257812
```

1.3 Let's train on real data

Step 11: Load the CIFAR-10 dataset

(It's very small and you can download it directly through torch, without manually downloading it)

[ ]:
```
# torchvision has some helper functions for image-based dataset
import torchvision

# we want to apply certain things to all of our images - that's what transforms␣
 ↪do
import torchvision.transforms as transforms


# we would like all of our incoming images to: become a torch Tensor
# (instead of a numpy array) and we want to normalize all images
# normalization: img = (img-mean)/standard_deviation
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) # parameters for
# the normalization are mean and std for each channel


# download the training part of the CIFAR-10 dataset and apply transforms
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

# wrap that into a multi-threaded data loader for quicker multi-CPU data loading
# and for being able to shuffle the data and sample whole batches and not just
# single elements
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=0)

# same thing for the test dataset
```

```
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=0)

# these are the names of the labels, corresponding to 0,1,2,3, etc.
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

  0%|          | 0/170498071 [00:00<?, ?it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

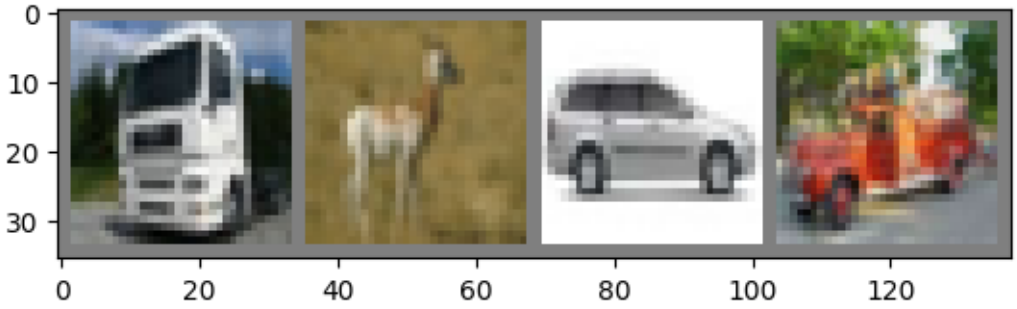
Step 12: ALWAYS verify your data!

Before running *any* experiment, always look at your data and make sure it's what you expect it to be

[ ]:
```
print ("dataset length:",len(trainset))

image, label = trainset[0]

print ("single image (size):",image.size()) # <-- this is a tensor
print ("single label:",label) # <-- this is not, but it will be if we use the␣
 ↪"trainloader" from above


### now let's actually -look- at the images

import matplotlib.pyplot as plt

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images using the trainloader
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images - make a grid of 4
imshow(torchvision.utils.make_grid(images))
```

```
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
dataset length: 50000
single image (size): torch.Size([3, 32, 32])
single label: 6
truck  deer   car truck
```



[ ]:
```
print (images.min(), images.mean(), images.max()) # important to verify your␣
↪distribution
```

```
tensor(-0.9922) tensor(0.0301) tensor(1.)
```

1.4 Let's glue it all together

Step 13: Let's actually run the optimization in a loop with the dataset

[ ]:
```
print ("[epoch, line of data]")

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):

        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        # print statistics
        running_loss += loss.item()

        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[{}, {}] \tloss: {}'.format(
                epoch + 1,
                i + 1,
                running_loss / 2000)
                )

            running_loss = 0.0

print('Finished Training')
```

```
[epoch, line of data]
[1, 2000]       loss: 2.200274342060089
[1, 4000]       loss: 1.8947220413684844
[1, 6000]       loss: 1.6740154998898507
[1, 8000]       loss: 1.5822681087106467
[1, 10000]      loss: 1.527068680100143
[1, 12000]      loss: 1.478550386980176
[2, 2000]       loss: 1.4029825597032906
[2, 4000]       loss: 1.3704184580296277
[2, 6000]       loss: 1.3622120897397398
[2, 8000]       loss: 1.3592307803183794
[2, 10000]      loss: 1.3203937190771102
[2, 12000]      loss: 1.2817288416475057
Finished Training
```

… Now go to the cells bellow (Step 14), and inspect the results. They're not very good, right?

Then, copy the entire previous cell below and run again (to see how the loss converges to something if you run it long enough). Run Step 14 again. See how much better that was? Not perfect, but better.

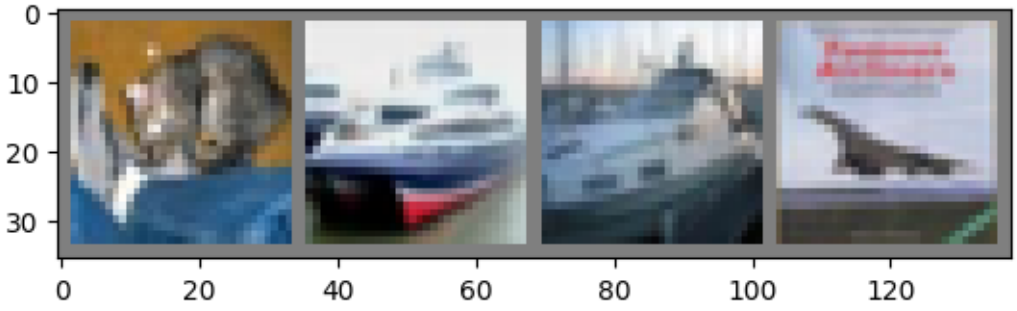
(Obviously, in a real setting, you'd use some kind of loop instead of copying the cell.)

[ ]:
```
# insert code here
```

Step 14: Inspection - manually check your predictions

[ ]:
```
dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))

print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
GroundTruth:    cat  ship  ship plane
```




```
[ ]:
```

```python
# now let's get the net's predictions
outputs = net(images)

# argmax the 10 dimensions into one
_, predicted = torch.max(outputs, 1)

# get the names of the labels for each int label
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:    cat  ship   car  ship
```

Step 15: Actually get a numerical evaluation

```
[ ]:
```

```python
correct = 0
total = 0

# If we're not learning anything then we use the torch.no_grad() environment.
# In this environment no gradients are ever calculated.
with torch.no_grad():
    for data in testloader:

        images, labels = data
        outputs = net(images)

        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print("Accuracy of the network on the " \
      "10000 test images: {}%".format(100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 54.27%
```

~60% is not very accurate. You can do much better.

1.5 "Homework"

Goal: improve the score, i.e. the test set accuracy

We won't verify how you do on this, so this is just so you can learn about neural networks.

Ideas: - try out a different optimizer - try out more training epochs - try out a bigger neural network (more hidden nodes or more layers) - try plotting the loss over time and stop training when the loss converges - (if you're fancy) try out dataset augmentation - that means applying more transformations to your images before feeding them to the network - like random rotation, random noise, etc. Here's a ist of all transforms: https://pytorch.org/docs/stable/torchvision/transforms.html

2 Contributors

- Florian Golemo
- Bhairav Mehta
- Charlie Gauthier