40216952_2023_Lab2_Ex

January 28, 2023

1 Lab 2 Exercises for COMP 691 (Deep Learning)

In this lab we will learn some basics of Pytorch. - You will implement a feedforward neural network using different implementation styles. - Understand how to use torch autograd for calculating gradients. - Learn how to use GPUs for computation speed.

Save your answers for this lab as they will be used for part of Lab 3.

Start by making a **copy** of this notebook in your Google Colab.

2 Exercise 1: Loading the dataset

Below we will create a dataloader for the MNIST training data using torchvision package (following e.g. https://github.com/pytorch/examples/blob/master/mnist/main.py#L112-L120).

The dataloader iterates over the training set and will output **mini-batches of size 256** image samples.

Note: you do not need to use the image labels in the rest of this lab since you will not be doing any training.

Remarks about using GPU:

- The "device" variable allows us to select which device to place the data on. Modify your colab (or local environment) to use a GPU.
- To use GPU in Google Colab, go to Runtime then choose "change runtime type". Then choose the hardware accelerator as GPU.
- In your Google Colab notebook set the variable device to "cuda", rerunning the cell below such that the data is placed on GPU inside the for loop.

[ ]:
```
from torchvision import datasets,transforms
import torch
dataset1 = datasets.MNIST('../data', train=True, download=True,␣
 ↪transform=transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(dataset1,
                                           batch_size=256,
                                           shuffle=True,
                                           drop_last=True)
```

```
device='cuda'

for (data, target) in train_loader:
  data = data.to(device)
  target = target.to(device)
print(data.shape)
print(target.shape)
print(data.device)
```

```
torch.Size([256, 1, 28, 28])
torch.Size([256])
cuda:0
```

If you ran the code cell above, you will notice that the data is a tensor of shape ([256, 1, 28, 28]) = (batch_size, number of color channels, length of image in pixels, width of image in pixels)

3 Exercise 2: Building a neural network from the ground up!

Network Architecture: - Using only torch primitives (e.g. torch.matmul, torch._relu, etc) implement a simple feedforward neural network with 2 hidden layers that takes as input MNIST digits (28x28) and outputs **a single scalar value** i.e., the class. Avoid using any functions from torch.nn class.

- You may select the hidden layer width (greater than 20) and activations (tanh, relu, sigmoid, others) as desired.
- A typical layer will transform its inputs as follows: $y = $(Wx+b)$, where σ is the non-linear activation function.
- Initialize the weights with uniform random values in the range -1 to 1 and biases at 0.

Data:

Using the data obtained from Exercise 1, make a forward pass through the dataset in mini-batches of 256 (feed the network data). To check you are on the right track, the shape of your output should be ([256]).

Hint: Remember that the goal is to feed the MNIST images and get a class label for each image. In this exercise there is no training so do not expect that the label will be meaningful/correct!

Pay attention to the shape of the input and how it gets changed as it passes from one layer to the next in the forward pass. Ex: (256, 28*28) -> (256, hiddden_size_1) -> (256,hidden_size_2) -> (256,1). This will help you when constructing the layers of the network.

[ ]:
```
import torch

## Initialize and track the parameters using a list or dictionary (modify the␣
 ↪None)
h1_size = 50
h2_size = 50
param_dict = {
    "W0": torch.rand(28*28, h1_size)*2-1,
```

```
    "b0": torch.rand(h1_size)*2-1,
    "W1": torch.rand(h1_size, h2_size)*2-1,
    "b1": torch.rand(h2_size)*2-1,
    "W2": torch.rand(h2_size, 1)*2-1,
    "b2": torch.rand(1)*2-1,
    }



## Make sure your parameters in param_dict require gradient for training the␣
 ↪network later!

for name , param in param_dict.items():
    param.requires_grad = True
    param_dict[name] = param.to(device)
    param_dict[name].retain_grad()


## Define the network
def my_nn(input, param_dict):
    r"""Performs a single forward pass of a Neural Network with the given
    parameters in param_dict.

    Args:
        input (torch.tensor): Batch of images of shape (B, H, W), where B is
            the number of input samples,and H and W are the image height and
            width respectively.
        param_dict (dict of torch.tensor): Dictionary containing the parameters
            of the neural network. Expects dictionary keys to be of format
            "W#" and "b#" where # is the layer number.

    Returns:
        torch.tensor: Neural network output of shape (B, )
    """
    #Reshape the input image from HxW to a flat vector
    x = input.view(-1 , 28*28).to(device)
    x.requires_grad = True


    #Your code here
    #Layer 1 using relu
    x = torch.relu(x @ param_dict["W0"] + param_dict["b0"])
    #Layer 2 using relu
    x = torch.relu(x @ param_dict["W1"] + param_dict["b1"])
    #output
    x = x @ param_dict["W2"] + param_dict["b2"]
    return x.view(-1)
```

```
## Perform forward pass
#forward pass
output = my_nn(data, param_dict)
print(output.shape)
print(output.device)
```

```
torch.Size([256])
cpu
```

#Exercise 3: Implementing the same network using torch.nn.module

Implement a new torch.nn.module that performs the equivalent of the network in Exercise 2 and call it "model".

Initialize it with the same weights (ex: **nn.Linear**(in_features,out_features) so that you could have a fair comparison between the two networks. The way to do this is through **weight.data** = insert your desired weights. You can do a similar thing with the bias).

Validate the outputs of this network is the same as the one in Exercise 2 on MNIST training set.

[ ]:
```
import torch.nn as nn
class MyNN(torch.nn.Module):

    def __init__(self,h1,h2):
        super(MyNN, self).__init__()
        self.l0 = nn.Linear(28*28,h1).to(device) #input dim goes here
        self.l1= nn.Linear(h1,h2).to(device)
        self.l2 = nn.Linear(h2,1).to(device) #output dim goes here --> remember␣
 ↪you have 10 classes


    def forward(self, x):
        x = x.view(-1, 28*28)
        h0 = self.l0(x) # output will be the one specified in __init__
        x = torch.relu(h0)
        h1 = self.l1(x)
        x = torch.relu(h1)
        return self.l2(x).view(-1)

model = MyNN(h1_size,h2_size)
```

3.1 Exercise 3.1: Validating that the two implementations are equal.

First you will need to make sure the param_dict from Exercise 2 and the nn module version have the same parameters (weights and biases).

You can do this for example using: "**model.linear1.weight.data** = copy.deepcopy(param_dict['W0'].data.T)".

Note: that we do a deepcopy just to make sure this model is separate from the one in the above cell

[ ]:
```
import copy
#Copy parameters from param_dict to nn module parameters
model.l0.weight.data = copy.deepcopy(param_dict['W0'].data.T)
model.l0.bias.data = copy.deepcopy(param_dict['b0'].data)
model.l1.weight.data = copy.deepcopy(param_dict['W1'].data.T)
model.l1.bias.data = copy.deepcopy(param_dict['b1'].data)
model.l2.weight.data = copy.deepcopy(param_dict['W2'].data.T)
model.l2.bias.data = copy.deepcopy(param_dict['b2'].data)
#Run the assert statement below to check they match
for i,(data, _) in enumerate(train_loader):
  assert(((model(data)-my_nn(data, param_dict))**2).mean()<1e-4) # check that␣
 ↪all the outputs are roughly equal
print("All Clear !")
```

```
All Clear !
```

#Exercise 4: Calculating gradients.

For a single mini-batch of 256 samples (you can select any minibatch), compute the gradient of the average of the neural network outputs (over the minibatch) w.r.t to the weights.

3.1.1 Let's break this down:

First you will need to get the mean/average of the outputs. Then you need find the gradient of this mean w.r.t to the weights.

To find the gradient you can use torch autograd, which you can use simply it by calling **.backward()** on the desired variable.

Your task is to print the gradients for the first layer weight and bias. You can use either the model defined from exercise 2 or 3 for this.

Note: The network here is $f : \mathcal{R}^{HW} \rightarrow \mathcal{R}$, which means that your input layer has HW neurons (HW features) and your output layer has one output neuron (one scalar output = class). Since each batch has 256 samples, the mean can be obtained by $o = \frac{1}{256}\sum_{i=0}^{255} f(x_i)$ or simply calling **.mean()** on the output of the network. You are asked to find $\nabla_w o$ and $\nabla_b o$. To access the gradient of each parameter you can call **.grad**.

[ ]:
```
output1 = model(data).mean()
output1.backward()

output2 = my_nn(data, param_dict).mean()
output2.backward()

epsilon = 1e-4
```

```
assert(torch.norm(param_dict['W0'].grad.T - model.l0.weight.grad)<epsilon)
assert(torch.norm(param_dict['b0'].grad - model.l0.bias.grad)<epsilon)
assert(torch.norm(param_dict['W1'].grad.T - model.l1.weight.grad)<epsilon)
assert(torch.norm(param_dict['b1'].grad - model.l1.bias.grad)<epsilon)
assert(torch.norm(param_dict['W2'].grad.T - model.l2.weight.grad)<epsilon)
assert(torch.norm(param_dict['b2'].grad - model.l2.bias.grad)<epsilon)
print("All Clear !")

model.zero_grad() # for nn module

for (_,param) in param_dict.items():
  if param.grad is not None: # grad buffer doesnt exist until the first backward pass
    param.grad.detach_() # by default the gradient is in the computation graph
    param.grad.zero_()
```

```
All Clear !
```

#Exercise 5: CPU or GPU ?

Below you will find code for comparing the speed of a model on CPU and GPU as well as comparing the speed of a forward pass to a forward/backward pass. Instantiate a version of your model from exercise 3 (preferably a larger version e.g. width 100 or 500) and run the timing code.

Write 1-2 sentences to summarize your observations about the relative speed's of CPU/GPU and forward/backward

[ ]:
```
#Instantiate a model defined from (3) here
model = MyNN(1000,1000)
```

[ ]:
```
#Run on CPU
import time as timer
data = data.to('cpu')
model.cpu()

print('Running on CPU')

start = timer.time()
for _ in range(10):
  model(data)
print("Time taken forward", timer.time() - start)

start = timer.time()
for _ in range(10):
  out = model(data).mean()
  out.backward()
print("Time taken forward/backward", timer.time() - start)
```

```
Running on CPU
Time taken forward 0.1332094669342041
```

```
Time taken forward/backward 0.3723299503326416
```

[ ]:
```python
#Run on GPU
#initialize cuda
data = data.to('cuda')
model.cuda()
model(data)
print('Running on GPU')


start = timer.time()
for _ in range(10):
  model(data)
torch.cuda.synchronize()
print("Time taken", timer.time() - start)

start = timer.time()
for _ in range(10):
  out = model(data).mean()
  out.backward()
torch.cuda.synchronize()
print("Time taken forward/backward", timer.time() - start)
```

```
Running on GPU
Time taken 0.005534648895263672
Time taken forward/backward 0.017503738403320312
```

Summary of observations here:

The time it takes on GPU is much less than CPU, suggesting GPU is better at performing these tasks. The time takes for backward is around 2x - 3x of forward.