Colab, Pytorch, Tensorboard

Colab (Jupyter)

Colab is, much like google docs, an online IDE for writing python code in a Jupyter Notebook hosted on a Google machine.

Python is an interpreted language; a Jupyter notebook breaks up code into cells which can be run one at a time, **preserving variables**. This is one of the main advantages of a notebook, you can do things like load data *one time*.

Helpful commands:

- First you want to connect to a Runtime (top right)
- You can choose the type of runtime (for example to access a GPU) under Runtime --> Change runtime type
- To run a cell do: **ctrl + enter**
    - **shift + enter** runs the cell and highlights the next
- You can create a new code or text cell in the top left
- When hovering over a cell, there is a small list of commands at the top right
    - Move cell up/down
    - Delete/copy/cut cell
- Double click a text cell to edit it
- Text cells are **Markdown**
- On the left you can access your filesystem, a list of active variables, search, contents, and a terminal
    - the filesystem is where you can "mount" or download data in order to access it with your code
    - instead of running things in the terminal you can run commands in cells using **!command**

Notes

- If you leave the session inactive it will timeout
- If you run something for too long it will timeout
- Your total quota for running with GPUs is limited
    - sufficient for this assignment but don't waste it, first conenct to cpu as you do first parts of the assignment

```
!ls
!pwd
# You can literally install packages
!pip install torch
```

Pytorch

Pytorch is a deep learning library.

That essentially means two things:

- like numpy it is a library for manipulating Nd-arrays --> Tensors
    - most (but not all) things you can do in numpy you can do in pytorch
    - very easy to convert between them
    - in general, numpy is better for pure array manipulation

- it provides methods for building and training neural networks

The main competitor library to pytorch is tensorflow. Why use one or the other?

- Generally accepted that tensorflow is slightly faster
  - This is because tensorflow models are "compiled" whereas pytorch ones run in-line
    - You can also compile pytorch models
- Pytorch is significantly easier to work with, to understand, and to prototype new models
  - I started with tensorflow, then eventually tried pytorch, and never looked back

```
import torch
import numpy as np
import matplotlib.pyplot as plt
```

```
# Check if GPU is available
torch.cuda.is_available()
```

```
    False
```

```
# torch.arange is just like numpy arange, similar to built-in python range()
a = torch.arange(12)
a
```

```
    tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
# Can easily convert between pytorch and numpy
b = a.numpy()
b
```

```
    array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
# reshape (numpy) --> view (pytorch)
# (turns out in recent pytorch versions you can also reshape a tensor, but it may create a copy)
print(a.view(2,6))
print(b.reshape(2,6))
```

```
    tensor([[ 0,  1,  2,  3,  4,  5],
            [ 6,  7,  8,  9, 10, 11]])
    [[ 0  1  2  3  4  5]
     [ 6  7  8  9 10 11]]
```

```
from torch import nn
```

```
class MyModel(nn.Module): # extend nn.Module
    def __init__(self, input_size, out_size):
        # call parent constructor
        super(MyModel, self).__init__()

        # define the layers in our model
        self.l1 = nn.Linear(in_features=input_size, out_features=128, bias=True)
        self.l2 = nn.Linear(in_features=128, out_features=256, bias=True)
        self.l3 = nn.Linear(in_features=256, out_features=out_size, bias=False)

    # this is where you call the model
    def forward(self, x): # x is your input features, i.e. an image
        x = nn.functional.relu(self.l1(x)) # pass x through linear layer, then apply relu activation
        x = nn.functional.relu(self.l2(x)) # do it again for the next layer
        x = self.l3(x) # last layer no activation
        return x # or torch.sigmoid(x) or softmax(x) if we want probabilities

# now we have our model, let's train it on some data
x = torch.linspace(0,10,10000).view(-1,1)
y = torch.sin(x) # let's learn the sin function


print(x.shape) # we have 10,000 datapoints of 1 feature each
print(x[1000:1010])
print(y[1000:1010])
```

```
torch.Size([10000, 1])
tensor([[1.0001],
        [1.0011],
        [1.0021],
        [1.0031],
        [1.0041],
        [1.0051],
        [1.0061],
        [1.0071],
        [1.0081],
        [1.0091]])
tensor([[0.8415],
        [0.8421],
        [0.8426],
        [0.8431],
        [0.8437],
        [0.8442],
        [0.8448],
        [0.8453],
        [0.8458],
        [0.8464]])
```

```
model = MyModel(input_size = x.shape[1], out_size=y.shape[1]) # in=1, out=1
```

```
model(VARIABLE) --> model.forward(VARIABLE)
```

```
# output shape is (N, out_size)
# grad_fn indicates that the gradients have been saved
print(model(x[:1]))
print(model.forward(x[:1]))

# can explicitly tell torch to not compute gradients...
with torch.no_grad():
    print(model(x[:1]))

# can also "detach" the gradients, meaning treat this output as a normal tensor and not model output
print(model(x[:1]).detach())
```

```
tensor([[-0.0168]], grad_fn=<MmBackward0>)
tensor([[-0.0168]], grad_fn=<MmBackward0>)
tensor([[-0.0168]])
tensor([[-0.0168]])
```

```
model(x[:1]).numpy()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
```

Some things to keep in mind:

- Why is the output not zero? --> initialization
    - What is the default initialization? --> "uniform"
- Keeping track of gradients is **memory intensive**
    - Keeping track of too many gradients often leads to memory issues
    - https://pytorch.org/docs/stable/notes/faq.html

Ok. Now let's train the model.

```
# To train a model we need an optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.001) # note we pass in the model parameters
```

```
for p in model.parameters():
    print(p.shape)
```

```
torch.Size([128, 1])
torch.Size([128])
torch.Size([256, 128])
torch.Size([256])
torch.Size([1, 256])
```

```
# we also need a loss function
criterion = nn.MSELoss()
```

```
# now we loop over our dataset
batch_size = 100
num_epochs = 1000
for epoch in range(num_epochs):
    print("\rEpoch {}".format(epoch), end="")
    for i in range(0, len(x), batch_size):
        # first thing we do is zero out the optimizer
        optimizer.zero_grad()

        batch_x = x[i:i+batch_size]
        batch_y = y[i:i+batch_size]

        # pass the batch through the model
        y_hat = model(batch_x)
        # compute the loss
        loss = criterion(y_hat, batch_y)

        # update model
        loss.backward()
        optimizer.step()
```

```
Epoch 926
```

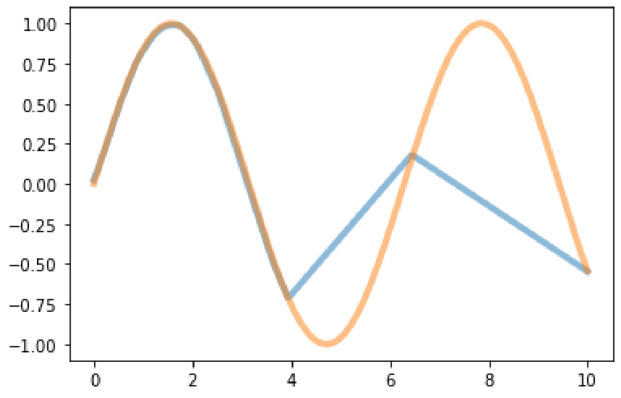
```
# let's see how we do
with torch.no_grad():
    y_pred = model(x)
```

```
print(x.shape, y_pred.shape, y.shape)
```

```
torch.Size([10000, 1]) torch.Size([10000, 1]) torch.Size([10000, 1])
```

```
plt.plot(x, y_pred, linewidth=4.0, alpha=0.5)
plt.plot(x, y, linewidth=4.0, alpha=0.5)
```

```
[<matplotlib.lines.Line2D at 0x7f6d7793d910>]
```



What can we do differently?

- Don't go through the data deterministically --> dataloaders
- Change the model architecture
- Different optimizer? Adam? Momentum?
- We don't actually need MSELoss, how can we implement the loss ourselves?

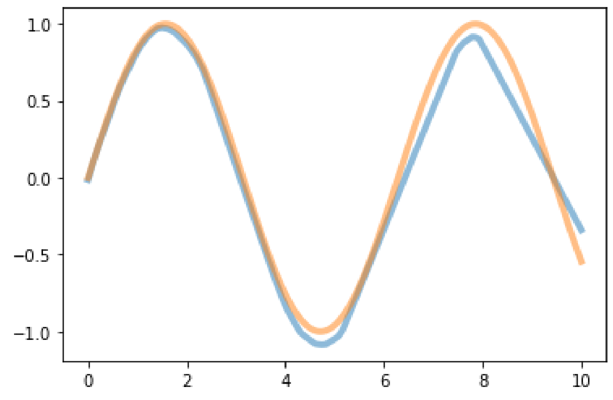
```
class MyDataset(torch.utils.data.Dataset):
    def __init__(self):
        self.x = torch.linspace(0,10,10000).view(-1,1)
        self.y = torch.sin(self.x)

    def __len__(self):
        return len(self.x)

    def __getitem__(self, index):
        # randomly choose whether to flip...
        return self.x[index], self.y[index]


def train(model, loader, optimizer, criterion, num_epochs = 1000):
    losses = []
    for epoch in range(num_epochs):
        epoch_loss = 0.0
        for batch in loader:
            batch_x, batch_y = batch
            # first thing we do is zero out the optimizer
            optimizer.zero_grad()

            # pass the batch through the model
            y_hat = model(batch_x)
            # compute the loss
            loss = criterion(y_hat, batch_y)

            # update model
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item() # this is important, do not retain the gradients
        epoch_loss /= len(loader)
        print("\rEpoch {}, Running loss {}".format(epoch, epoch_loss), end="")
        losses.append(epoch_loss)
    return losses
```

```
def mse_loss(y_hat, y):
    return ((y-y_hat)**2).mean()


my_dataset = MyDataset()
batch_size=64
data_loader = torch.utils.data.DataLoader(my_dataset, batch_size=batch_size, shuffle=True, drop_last=True)


model = MyModel(input_size=my_dataset.x.shape[1], out_size=my_dataset.y.shape[1])
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
criterion = nn.MSELoss()
# criterion = mse_loss


losses = train(model, data_loader, optimizer, criterion, num_epochs=500)
```

```
     Epoch 199, Running loss 0.00747618239480429
```

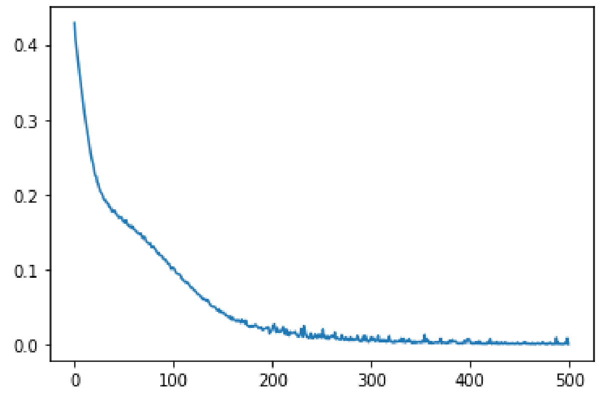
```
with torch.no_grad():
    y_pred = model(x)


plt.plot(my_dataset.x, y_pred, linewidth=4.0, alpha=0.5)
plt.plot(my_dataset.x, my_dataset.y, linewidth=4.0, alpha=0.5)
```

```
     [<matplotlib.lines.Line2D at 0x7f6d77893090>]
```



```
plt.plot(losses)
```

```
     [<matplotlib.lines.Line2D at 0x7f6d772a5750>]
```



Convolutions...

```
def conv_out_size(inp_size, kernel_size, dilation, padding, stride):
    return ((inp_size + 2*padding - dilation * (kernel_size - 1) - 1) // stride) + 1


# input is: (batch_size, number_channels, height, width)
class ExampleConv(nn.Module):
    def __init__(self, input_size=224):
        super(ExampleConv, self).__init__()

        self.input_size = input_size

        kernel_size = 3
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=kernel_size, stride=1, padding=
        self.conv1_out = conv_out_size(inp_size=self.input_size, kernel_size=kernel_size, dilation=1, padd

        self.conv2 = nn.Conv2d(16, 32, kernel_size, stride=1, padding=1, dilation=1)
        self.conv2_out = conv_out_size(inp_size=self.conv1_out, kernel_size=kernel_size, dilation=1, paddi

        self.conv3 = nn.Conv2d(32, 16, kernel_size, stride=2, padding=2, dilation=1)
        self.conv3_out = conv_out_size(inp_size=self.conv2_out, kernel_size=kernel_size, dilation=1, paddi

        self.conv4 = nn.Conv2d(16, 9, kernel_size, stride=1, padding=0, dilation=1)
        self.conv4_out = conv_out_size(inp_size=self.conv3_out, kernel_size=kernel_size, dilation=1, paddi

        # self.l1 = nn.Linear(in_features=self.conv4_out*self.conv4_out*9, out_features=2)...

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = nn.functional.relu(self.conv3(x))
        x = self.conv4(x)

        # x = self.l1(x.view(-1, self.conv4_out*self.conv4_out*9))
        return torch.sigmoid(x)


input_size = 256
conv_net = ExampleConv(input_size=input_size)


print(input_size, conv_net.conv1_out, conv_net.conv2_out, conv_net.conv3_out, conv_net.conv4_out)
```

```
256 254 254 128 126
```

```
tmp_img = torch.rand((1,3,input_size,input_size))
with torch.no_grad():
    output = conv_net(tmp_img)
    print(output.shape)
```

```
torch.Size([1, 9, 126, 126])
```

Tensorboard

The default directory is "runs" --> check files to the left

Generally (outside of colab), you would run tensorboard in a terminal which hosts a local tensorbaord instance in browser.

You can run a tensorboard instance on another machine, for example on a cluster, and then access it locally to see if your models are training properly.

[ ] ↳ *2 cells hidden*