Homework 01

Deep Learning

Name: Raja Haseeb ur Rehman

Student ID: 20194673

Problem 1

1) Make a python code for plotting some graph to help your analysis
   a) Loss graph

Code

Saving loss values. Can save in a file:

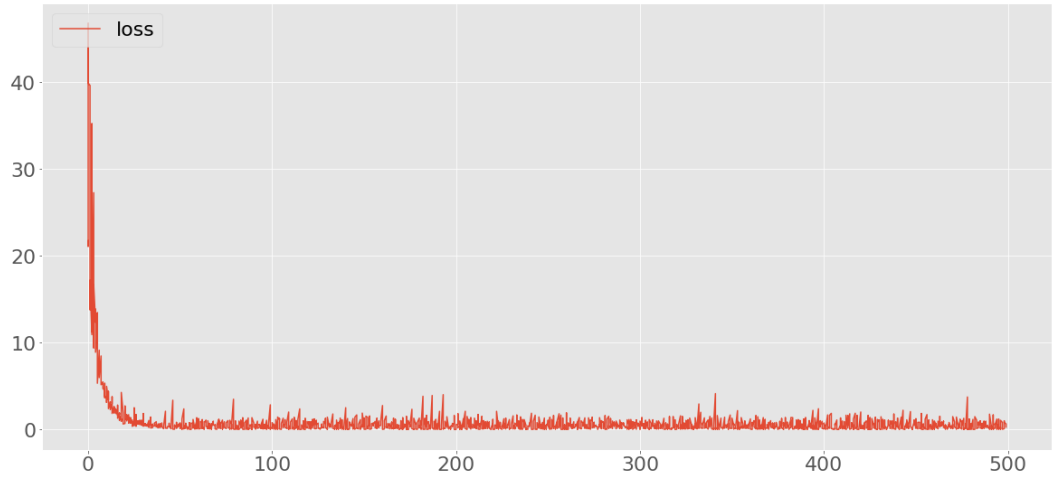
```
[45]: import time

      MODEL_NAME = f"model-{int(time.time())}"
      print(MODEL_NAME)

      def fit(model,loader,criterion,learning_rate,num_epochs):
          model.train()
          optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
          with open("model2.log", "a") as f:

              for epoch in range(num_epochs):
                  for i, data in enumerate(loader):
                      if torch.cuda.is_available():
                          x = data[0].type(torch.FloatTensor).cuda()
                          y = data[1].type(torch.FloatTensor).cuda()
                      else:
                          x = data[0].type(torch.FloatTensor)
                          y = data[1].type(torch.FloatTensor)



                      y_hat = model(x)
                      loss = criterion(y_hat, y)

                      optimizer.zero_grad()
                      loss.backward()
                      optimizer.step()

                      f.write(f"{MODEL_NAME}, {epoch}, {round(time.time(),3)},{round(float(loss), 4)}\n")

      model-1585490285
```

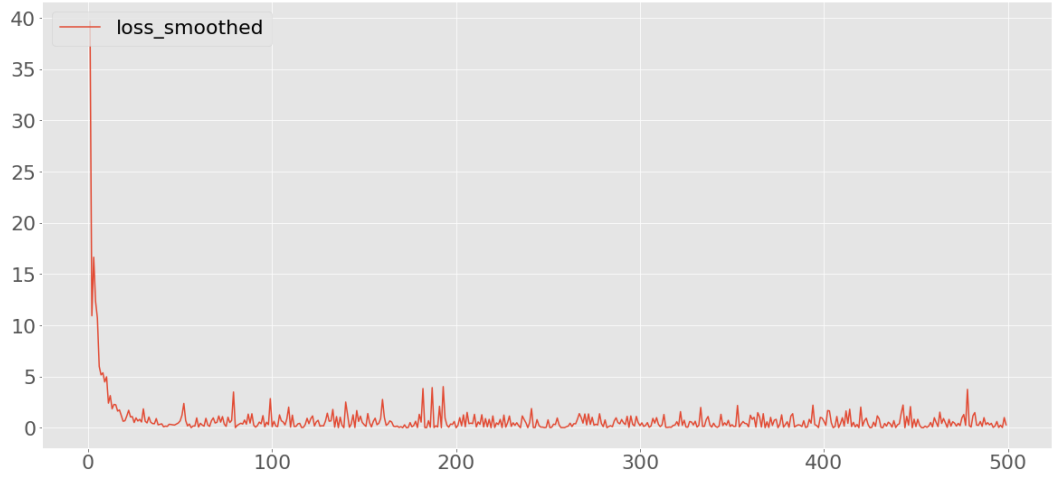
Alternatively, in an array:

```
[23]: import time

      MODEL_NAME = f"model-{int(time.time())}"
      print(MODEL_NAME)

      LOSS = []
      EPOCHS = []

      def fit(model,loader,criterion,learning_rate,num_epochs):
          model.train()
          optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
          with open("model2.log", "a") as f:

              for epoch in range(num_epochs):
                  for i, data in enumerate(loader):
                      if torch.cuda.is_available():
                          x = data[0].type(torch.FloatTensor).cuda()
                          y = data[1].type(torch.FloatTensor).cuda()
                      else:
                          x = data[0].type(torch.FloatTensor)
                          y = data[1].type(torch.FloatTensor)



                      y_hat = model(x)
                      loss = criterion(y_hat, y)

                      optimizer.zero_grad()
                      loss.backward()
                      optimizer.step()

                      LOSS.append(loss)
                      EPOCHS.append(epoch)
                      #f.write(f"{MODEL_NAME}, {epoch}, {round(time.time(),3)},{round(float(loss), 4)}\n")
```

Then plotting loss as follows.

```
[29]: #In EPOCH Form

      %matplotlib notebook
      %matplotlib inline


      import matplotlib.pyplot as plt
      from matplotlib import style
      from scipy.interpolate import interp1d
      style.use("ggplot")
      #model_name = "model-1585490285" #or MODEL_NAME



      fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
      plt.figure(figsize=(20,20))
      plt.rcParams.update({'font.size': 22})

      f = interp1d(EPOCHS, LOSS)

      #define axis, each axis is a graph
      ax1 = plt.subplot2grid((2,1), (0,0))
      ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

      ax1.plot(EPOCHS, LOSS, label="loss")
      ax1.legend(loc=2)
      ax2.plot(EPOCHS, f(EPOCHS), label="loss_smoothed")
      ax2.legend(loc=2)
      plt.show()
```





b) The real and learned function

Code

```
[29]: import time
      import numpy as np
      import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

      MODEL_NAME = f"model-{int(time.time())}"
      print(MODEL_NAME)


      def fit(model,loader,criterion,learning_rate,num_epochs):
          model.train()
          optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
          with open("model_0.1.log", "a") as f:

              for epoch in range(num_epochs):
                  for i, data in enumerate(loader):
                      if torch.cuda.is_available():
                          x = data[0].type(torch.FloatTensor).cuda()
                          y = data[1].type(torch.FloatTensor).cuda()
                      else:
                          x = data[0].type(torch.FloatTensor)
                          y = data[1].type(torch.FloatTensor)

                      y_hat = model(x)
                      loss = criterion(y_hat, y)

                      optimizer.zero_grad()
                      loss.backward()
                      optimizer.step()

                      print(y)
                      print(y_hat)
```

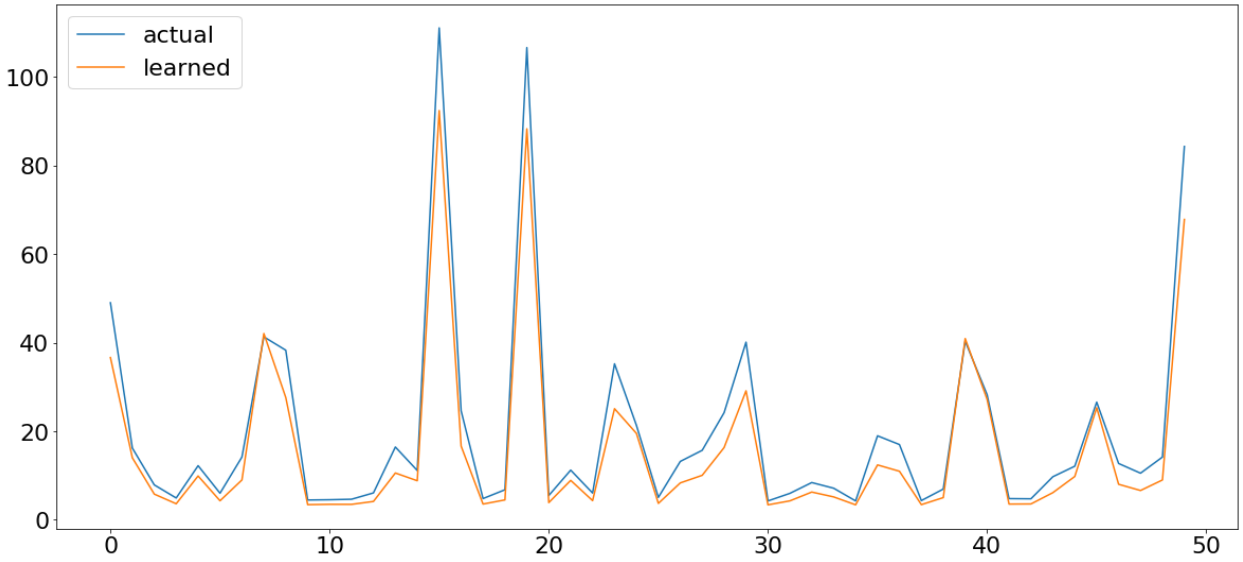
```
          y = y.cpu().detach().numpy()
          y_hat = y_hat.cpu().detach().numpy()
          fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
          plt.figure(figsize=(20,20))
          plt.rcParams.update({'font.size': 22})


          #define axis, each axis is a graph
          ax1 = plt.subplot2grid((2,1), (0,0))
          #ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

          ax1.plot(y, label="actual")
          ax1.plot(y_hat, label="learned")
          ax1.legend(loc=2)

          #ax2.plot(x, LF, label="learned")
          #ax2.legend(loc=2)

          plt.show()

                  #f.write(f"{MODEL_NAME}, {epoch}, {round(time.time(),3)},{round(float(loss), 4)}, {x}, {y}, {y_hat}\n")
model-1585898579
```

Results

After 10 epochs

==> Learned function: +7.25 x^4 -3.16 x^3 +2.56 x^2 -0.83 x^1 +3.54

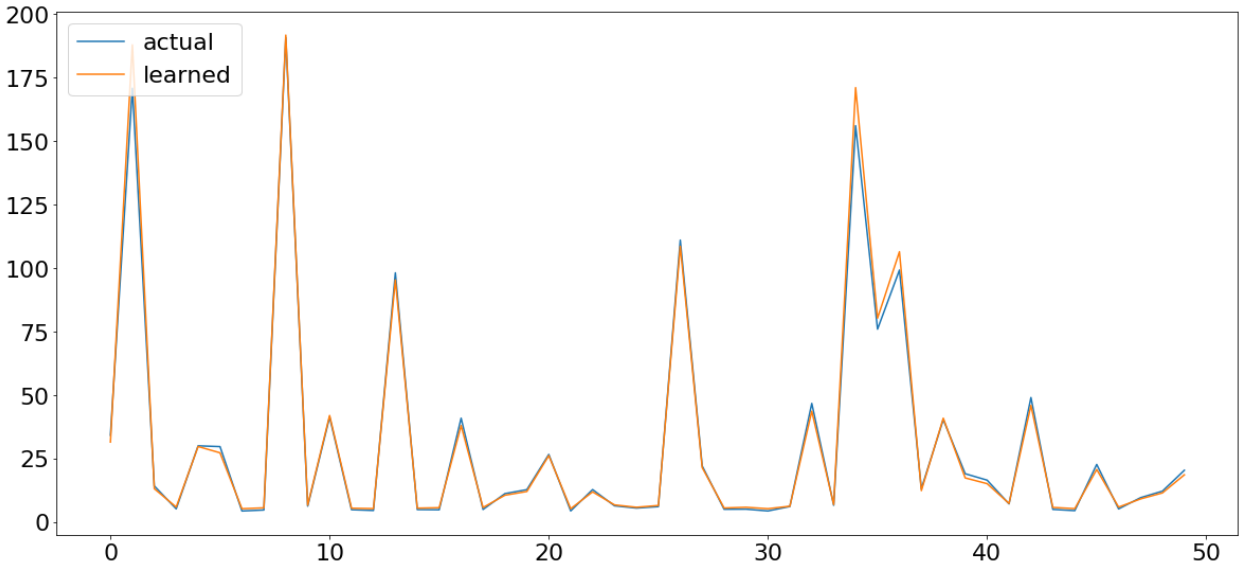
==> Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40



After 20 epochs

==> Learned function: +7.01 x^4 -4.66 x^3 +4.39 x^2 -1.57 x^1 +5.33

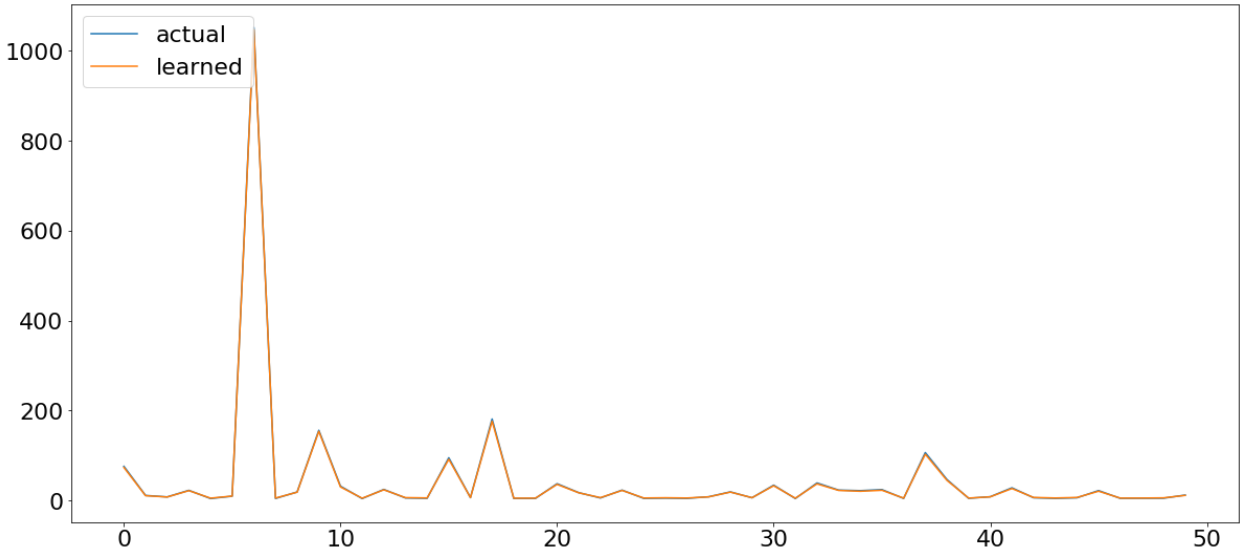
==> Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40



After 30 epochs

==> Learned function: +7.01 x^4 -4.66 x^3 +4.39 x^2 -1.57 x^1 +5.33

==> Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40



As we can see, after some epochs, both polynomials are close to each other.

To get better polynomial curves, I changed data for input and defined it in linear range instead of random.

Code

```
[13]: import time
      import re
      import numpy as np
      import matplotlib.pyplot as plt
      from matplotlib import style
      from scipy.interpolate import interp1d

      MODEL_NAME = f"model-{int(time.time())}"
      print(MODEL_NAME)

      def fit(model,loader,criterion,learning_rate,num_epochs):
          model.train()
          optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
          with open("model_0.1.log", "a") as f:

              for epoch in range(num_epochs):
                  for i, data in enumerate(loader):
                      if torch.cuda.is_available():
                          x = data[0].type(torch.FloatTensor).cuda()
                          y = data[1].type(torch.FloatTensor).cuda()
                      else:
                          x = data[0].type(torch.FloatTensor)
                          y = data[1].type(torch.FloatTensor)



                      y_hat = model(x)
                      loss = criterion(y_hat, y)

                      optimizer.zero_grad()
                      loss.backward()
                      optimizer.step()
```

```
        if epoch % 10 == 0:
            Real_function = poly_desc(W_target.view(-1), b_target)
            Learned_function = poly_desc(net.fc.weight.data.view(-1), net.fc.bias.data)
            print (Real_function)
            print (Learned_function)

            c1 = float(re.split("[( - +)]", Real_function)[-11])
            c2 = float(re.split("[( - +)]", Real_function)[-9])
            c3 = float(re.split("[( - +)]", Real_function)[-6])
            c4 = float(re.split("[( - +)]", Real_function)[-4])
            c5 = float(re.split("[( - +)]", Real_function)[-1])

            #coefficients of learned function
            C1 = float(re.split("[( - +)]", Learned_function)[-11])
            C2 = float(re.split("[( - +)]", Learned_function)[-9])
            C3 = float(re.split("[( - +)]", Learned_function)[-6])
            C4 = float(re.split("[( - +)]", Learned_function)[-4])
            C5 = float(re.split("[( - +)]", Learned_function)[-1])

            print (c1, c2, c3, c4, c5)
            print (C1, C2, C3, C4, C5)

            #Redifing functions for plotting in values form instead of string, so that we can input data

            x = np.linspace(-20, 20, num=200)

            Real_function = []
            Learned_function = []

            for i in x:
                Real_function.append(c1*i**4 -c2*i**3 +c3*i**2 -c4*i**1 +c5)
                Learned_function.append(C1*i**4 -C2*i**3 +C3*i**2 -C4*i**1 +C5)
```

```
            fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
            plt.figure(figsize=(20,20))
            plt.rcParams.update({'font.size': 22})

            #f = interp1d(epochs, losses)
            #define axis, each axis is a graph
            ax1 = plt.subplot2grid((2,1), (0,0))
            #ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

            ax1.plot(Real_function, label="real function")
            ax1.plot(Learned_function, label="learned function")
            ax1.legend(loc=2)
            #ax2.plot(epochs, f(epochs), label="loss_smoothed") #smoothed plot
            #ax2.plot(epochs,val_losses, label="val_loss")
            #ax2.legend(loc=2)

            plt.show()
model-1585898718
```

Results

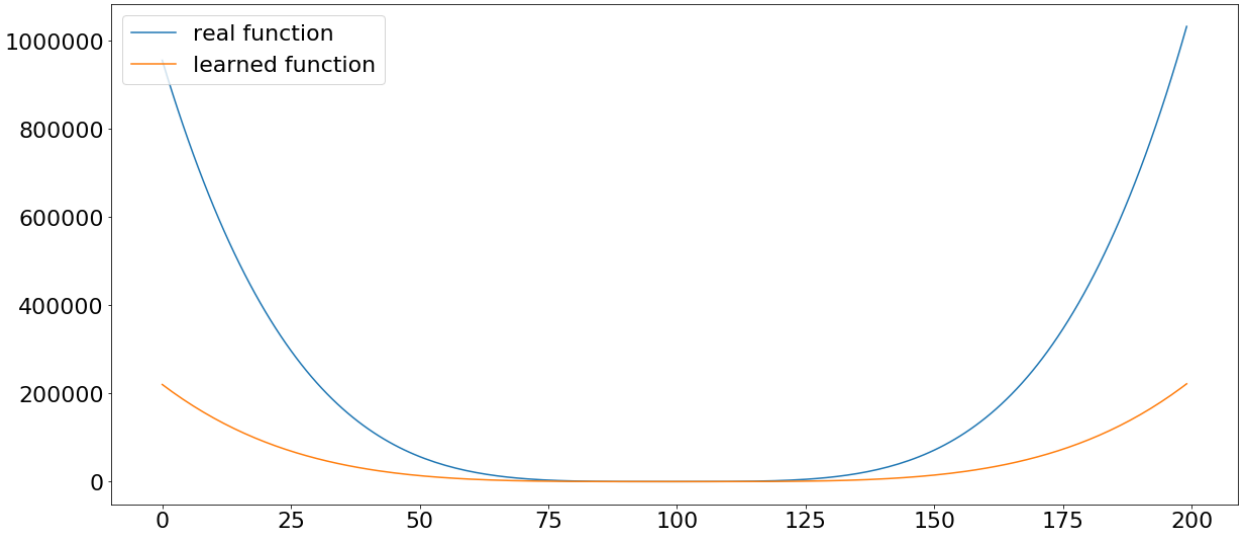
After 10 epochs

Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40

Learned function: +1.38 x^4 -0.10 x^3 +0.41 x^2 -0.01 x^1 +0.40

6.19 -4.8 7.71 -2.04 4.4

1.38 -0.1 0.41 -0.01 0.4



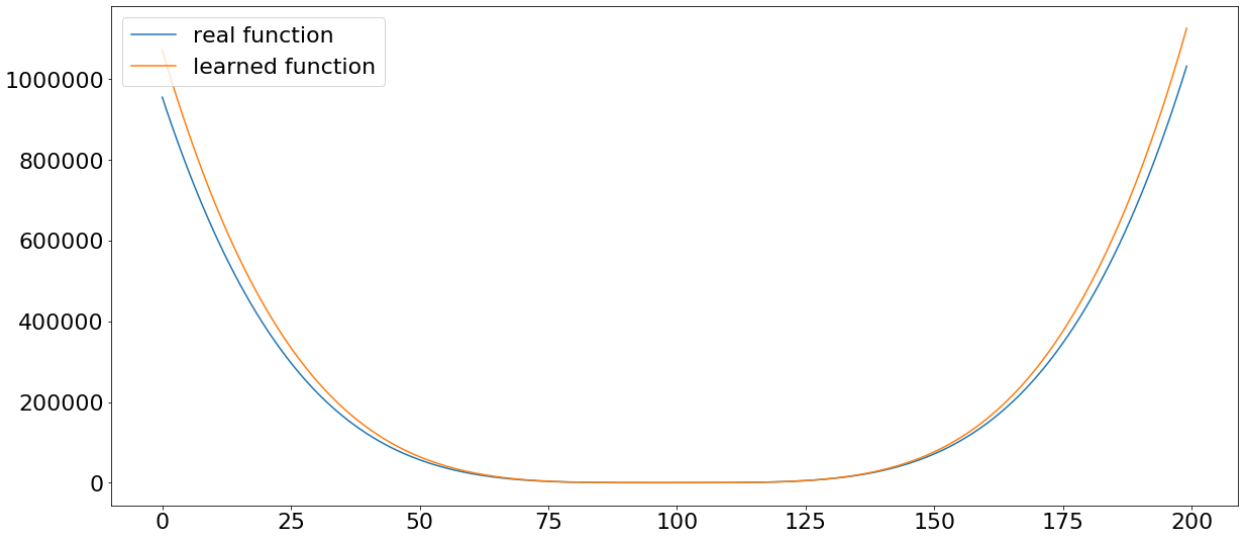
After 20 epochs

Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40

Learned function: +6.86 x^4 -3.36 x^3 +2.68 x^2 -0.93 x^1 +3.86

6.19 -4.8 7.71 -2.04 4.4

6.86 -3.36 2.68 -0.93 3.86



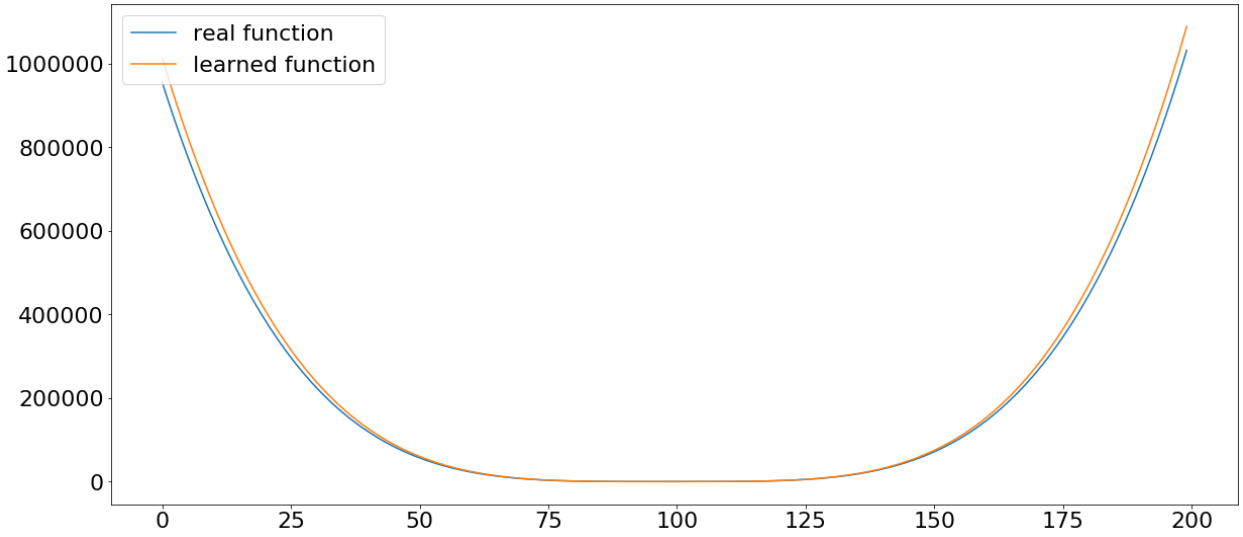
After 30 epochs

Actual function: +6.19 x^4 -4.80 x^3 +7.71 x^2 -2.04 x^1 +4.40

Learned function: s+6.56 x^4 -4.73 x^3 +4.41 x^2 -1.60 x^1 +5.36

6.19 -4.8 7.71 -2.04 4.4

6.56 -4.73 4.41 -1.6 5.36



2) Analyze about convergence according to learning rate. (Including loss graph per epoch, and the real function and the learned function per 100 epochs)

Code

```
[9]: num_epochs = 500
     batch_size = 50
     learning_rate = 0.1
     criterion = nn.SmoothL1Loss()
```

```
[21]: import time
      import numpy as np
      import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D
      from scipy.interpolate import interp1d

      MODEL_NAME = f"model-{int(time.time())}"
      print(MODEL_NAME)

      LOSS = []
      EPOCHS = []

      def fit(model,loader,criterion,learning_rate,num_epochs):
          model.train()
          optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
          with open("model_0.1.log", "a") as f:

              for epoch in range(num_epochs):
                  for i, data in enumerate(loader):
                      if torch.cuda.is_available():
                          x = data[0].type(torch.FloatTensor).cuda()
                          y = data[1].type(torch.FloatTensor).cuda()
                      else:
                          x = data[0].type(torch.FloatTensor)
                          y = data[1].type(torch.FloatTensor)

                      y_hat = model(x)
                      loss = criterion(y_hat, y)

                      optimizer.zero_grad()
                      loss.backward()
                      optimizer.step()
```

```
                LOSS.append(loss)
                EPOCHS.append(epoch)

            fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
            plt.figure(figsize=(20,20))
            plt.rcParams.update({'font.size': 22})

            f = interp1d(EPOCHS, LOSS)

            #define axis, each axis is a graph
            ax1 = plt.subplot2grid((2,1), (0,0))
            ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

            ax1.plot(EPOCHS, LOSS, label="loss")
            ax1.legend(loc=2)
            ax2.plot(EPOCHS, f(EPOCHS), label="loss_smoothed")
            ax2.legend(loc=2)
            plt.show()

            if epoch % 100 == 0:
                y = y.cpu().detach().numpy()
                y_hat = y_hat.cpu().detach().numpy()
                fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
                plt.figure(figsize=(20,20))
                plt.rcParams.update({'font.size': 22})


                #define axis, each axis is a graph
                ax1 = plt.subplot2grid((2,1), (0,0))
                #ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

                ax1.plot(y, label="actual")
                ax1.plot(y_hat, label="learned")
                ax1.legend(loc=2)
```

```
            plt.show()

            if epoch % 100 == 0:
                y = y.cpu().detach().numpy()
                y_hat = y_hat.cpu().detach().numpy()
                fig = plt.figure() #for multiple figurs, for one thing just use plt.plot(times, accuracies) and plt.show()
                plt.figure(figsize=(20,20))
                plt.rcParams.update({'font.size': 22})


                #define axis, each axis is a graph
                ax1 = plt.subplot2grid((2,1), (0,0))
                #ax2 = plt.subplot2grid((2,1), (1,0), sharex=ax1)

                ax1.plot(y, label="actual")
                ax1.plot(y_hat, label="learned")
                ax1.legend(loc=2)

                #ax2.plot(x, LF, label="learned")
                #ax2.legend(loc=2)

                plt.show()

                #f.write(f"{MODEL_NAME}, {epoch}, {round(time.time(),3)},{round(float(loss), 4)}, {x}, {y}, {y_hat}\n")
```

```
model-1585900118
```

Results

1. Loss

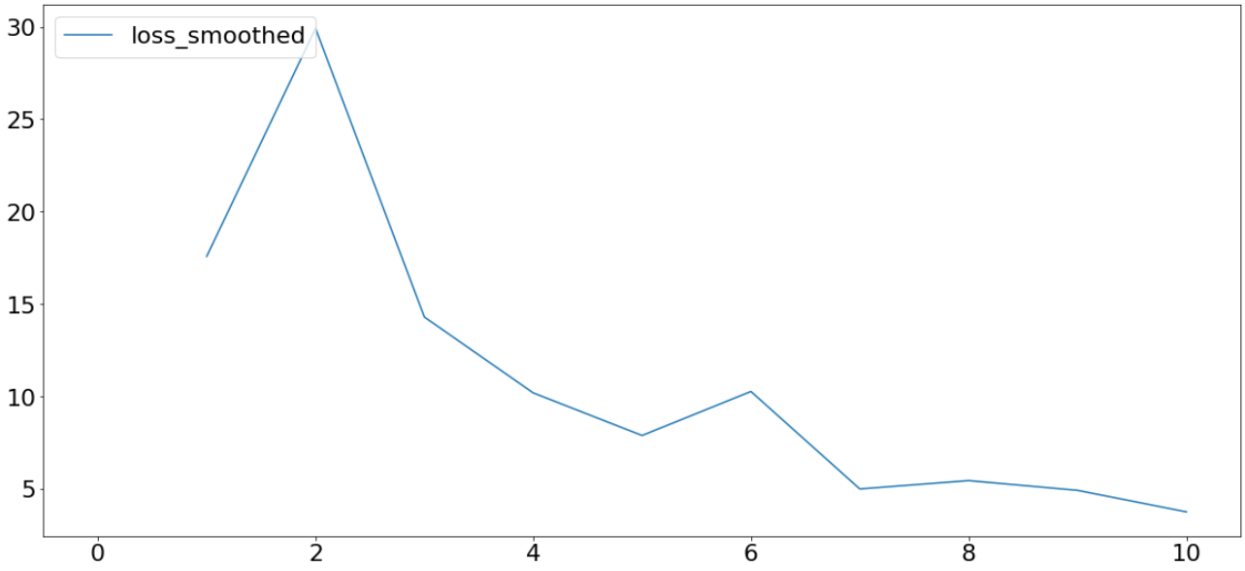


Figure 1: Loss after 10 epochs

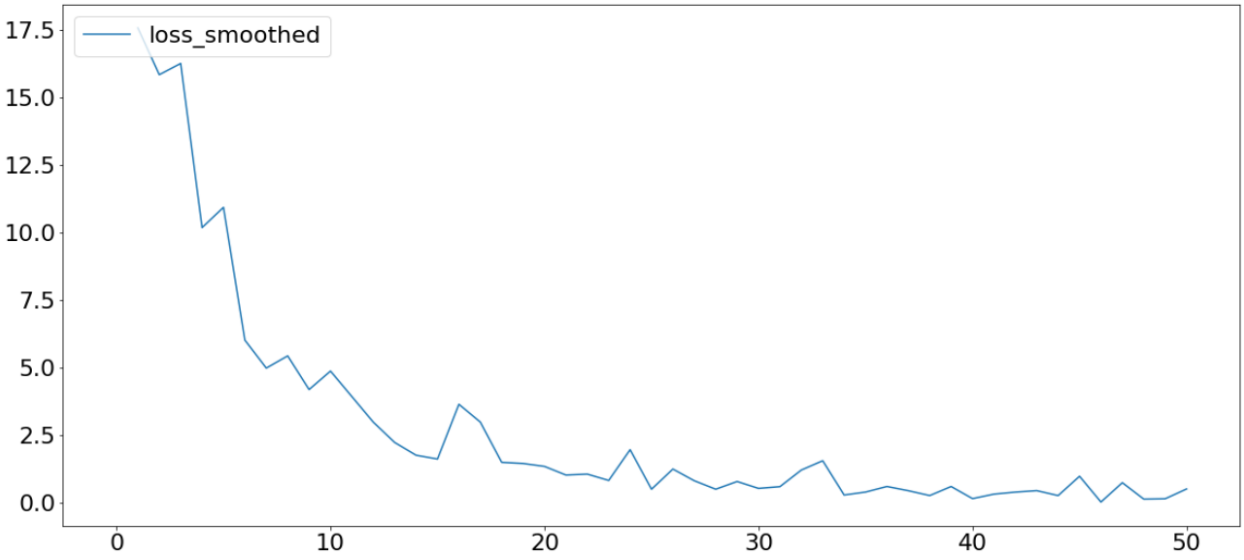


Figure 2: Loss after 50 epochs

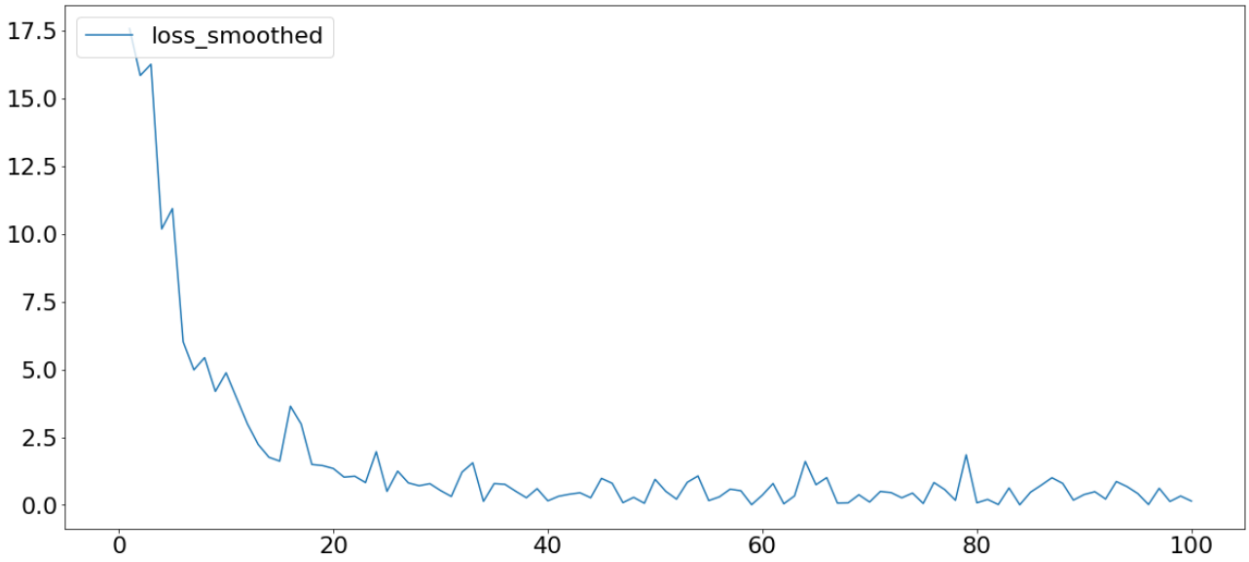


Figure 3: Loss after 100 epochs

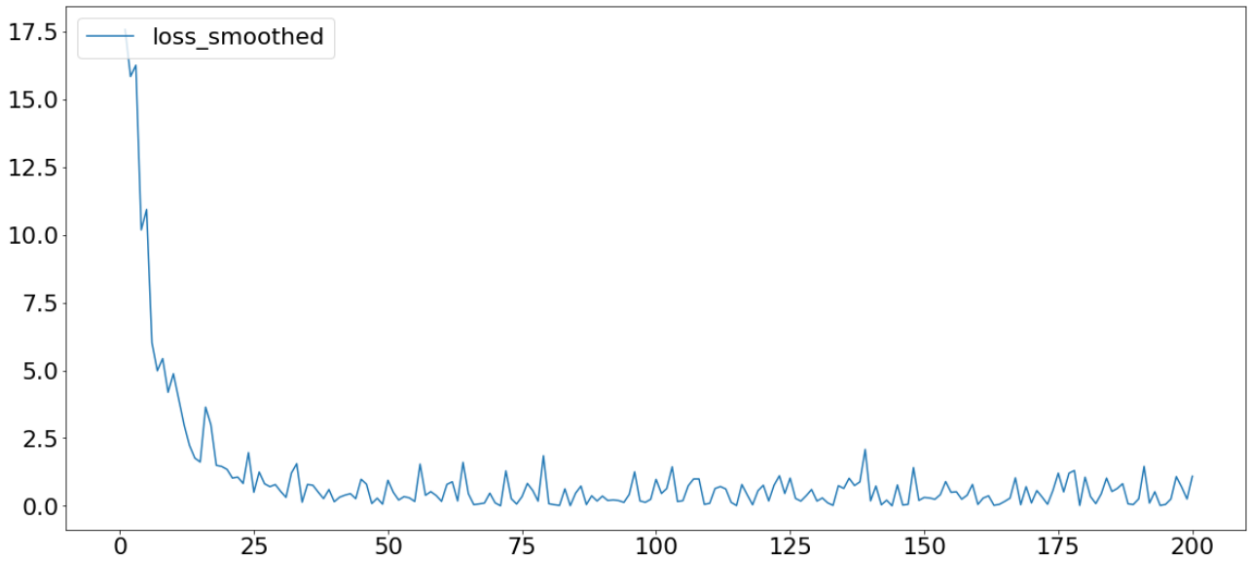


Figure 4: Loss after 200 epochs

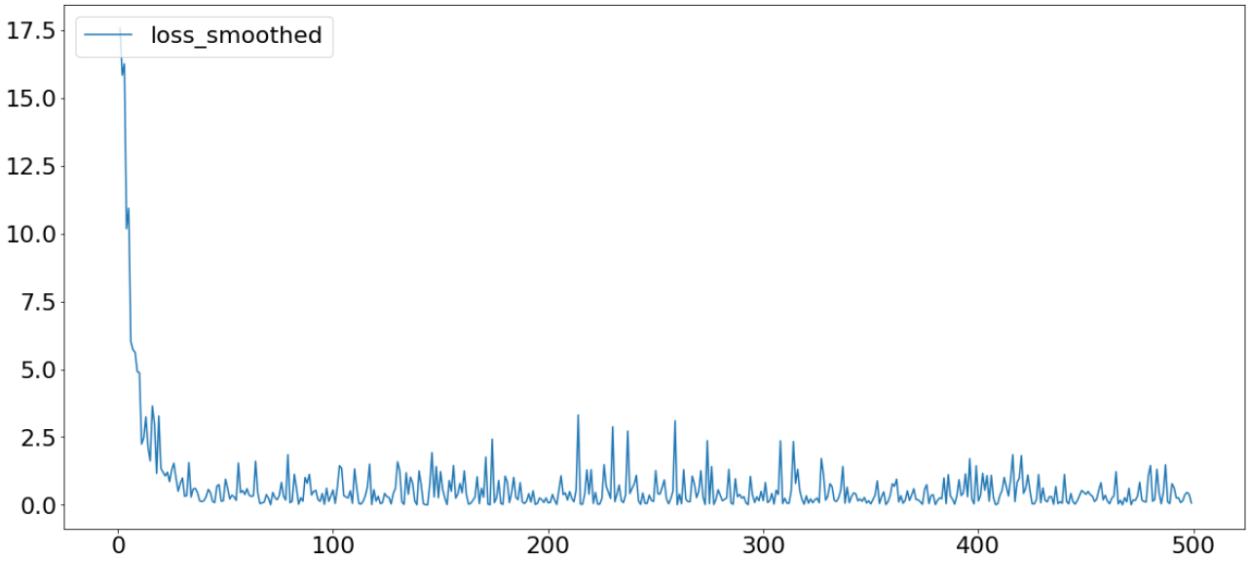


Figure 5: Loss after 500 epochs

2. Functions

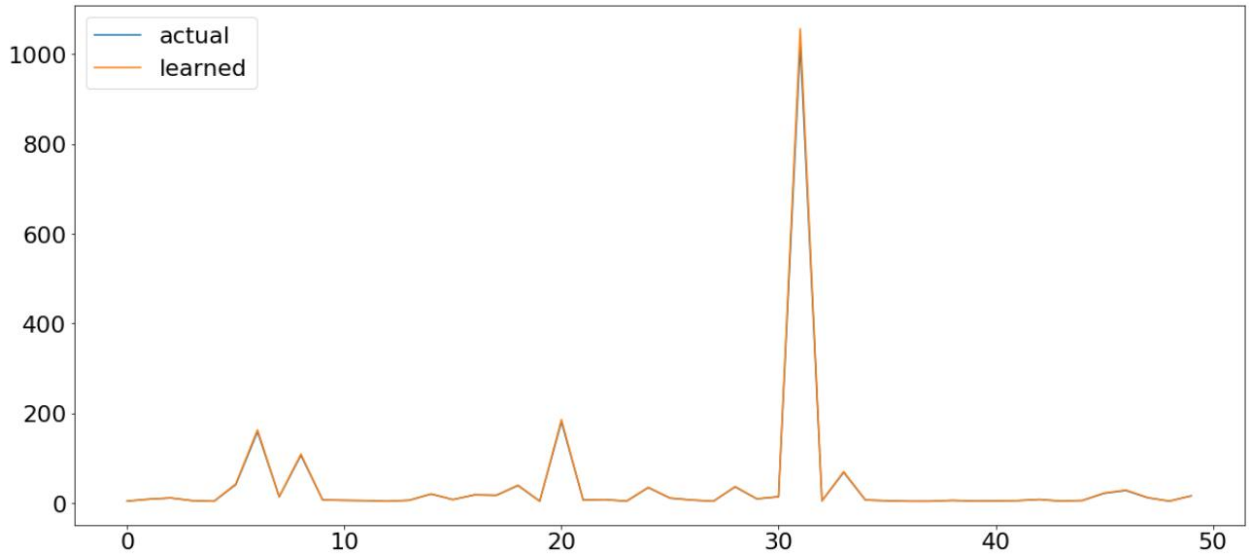


Figure 6: After 100th epoch

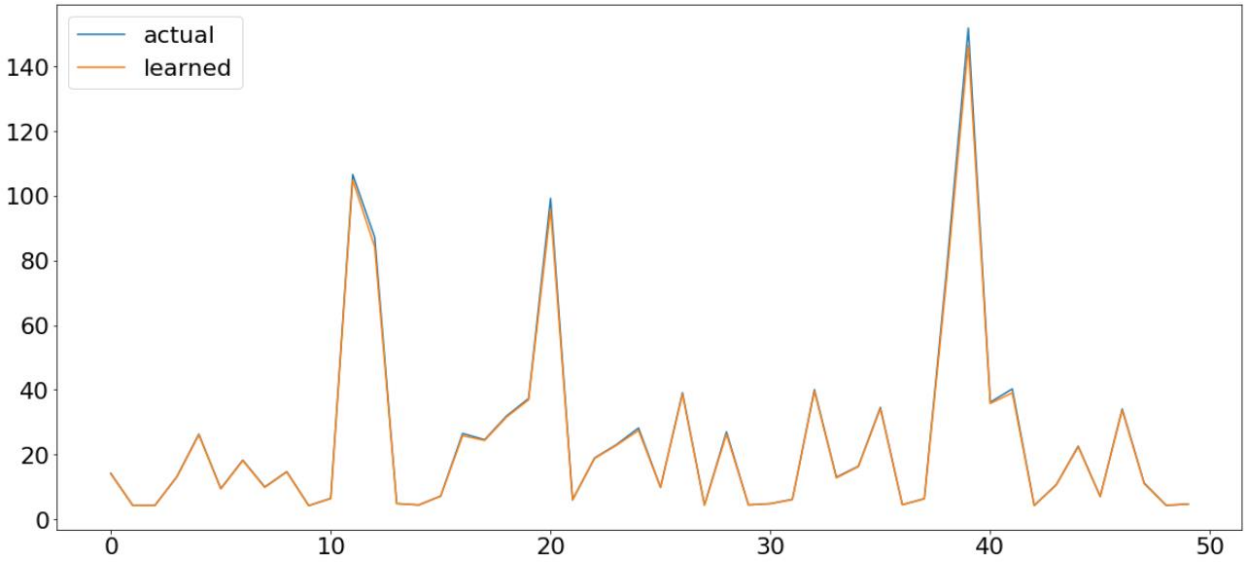


Figure 7: After 200 epochs

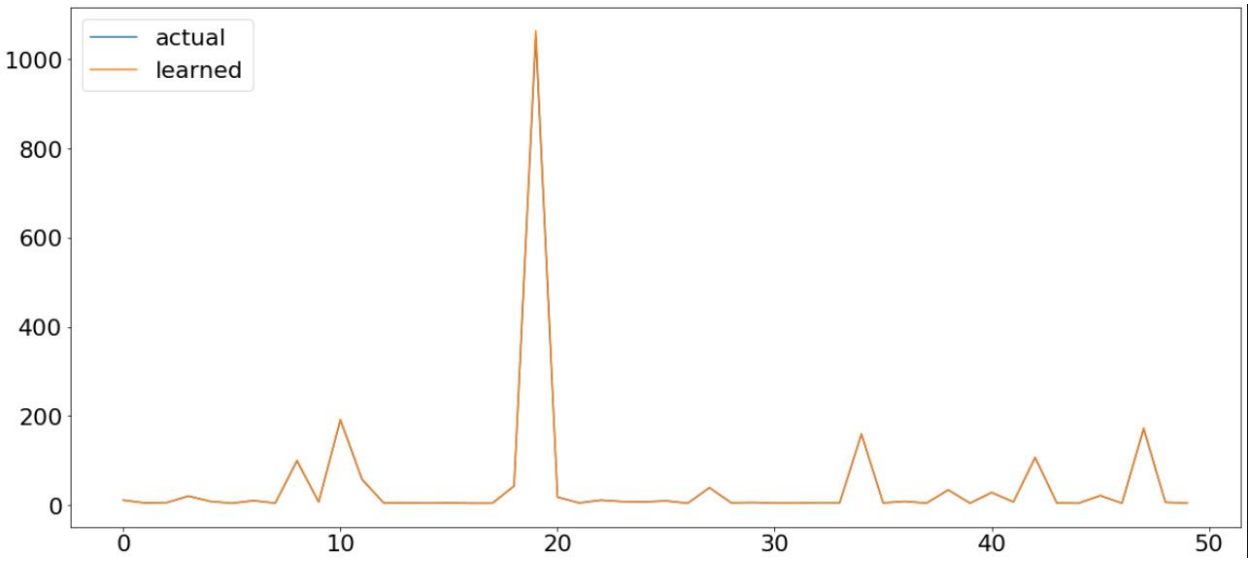


Figure 8: After 300 epochs

Problem 2

Q1. What is 'torch.cuda.is_available' used for?

Returns a bool indicating if CUDA is currently available. It shows we have access to some sort of GPU and then we can define devices which we want to use to run the different part of our network.

Q2. What is 'torch.utils.data.DataLoader' used for?

Data loader. Combines a dataset and a sampler, and provides single or multi-process iterators over the dataset. It represents a Python iterable over a dataset, with support for

- map-style and iterable-style datasets,
- customizing data loading order,
- automatic batching,
- single- and multi-process data loading,
- automatic memory pinning.

Q3. What is 'nn.Linear' used for?

This just means fully connected layer with flat input.

torch.nn.Linear (in_features, out_features, bias=True)

Applies a linear transformation to the incoming data: y = x*W^T + b

Parameters:

- **in_features** – size of each input sample (i.e. size of x)
- **out_features** – size of each output sample (i.e. size of y)
- **bias** – If set to False, the layer will not learn an additive bias. Default: True

Q4. What is 'view' used for? (4-2)What does '-1' mean?

'view' is just like numpy.reshape. But in pytorch they named it view instead of reshape. We can use it to reshape our data.

‘-1’ just specifies that this input will be of unknown shape. It will be of any size. We can also use ’1’ instead of ‘-1’.

In our case, we really just want a 1x784, and we could say that, but you will more often is -1 used in these shapings. Why? -1 suggests "any size". So it could be 1, 12, 92, 15295...etc. It's a handy way for that bit to be variable. In this case, the variable part is how many "samples" we'll pass through.

Q5. What is 'optim.SGD' used for?

torch.optim is a package implementing various optimization algorithms.

optim.SGD is used to select Stochastic Gradient Descent optimizer, and then we can also control/pass parameters like learning rate, momentum etc. using optim.SGD.

Rather than manually updating the weights of the model, we use the optim package to define an Optimizer that will update the weights for us. The optim package defines many optimization algorithms that are commonly used for deep learning, including SGD+momentum, RMSProp, Adam, etc.

Q6. What is 'nn.CrossEntropyLoss' used for?

It is used to defined which type of loss function we want to use and minimize. Here we are using CrossEntroppyLoss.

Q7. What is 'optimizer.zero_grad()' used for?

This is used to zero the gradients before each pass, otherwise gradients will start to add up. This is why it's important to do a net.zero_grad() for every step, otherwise these gradients will add up for every pass, and then we'll be re-optimizing for previous gradients that we already optimized for.

In PyTorch, we need to set the gradients to zero before starting to do backpropragation because PyTorch *accumulates the gradients* on subsequent backward passes. This is convenient while training RNNs. So, the default action is to accumulate (i.e. sum) the gradients on every loss.backward() call.

Because of this, when you start your training loop, ideally you should zero out the gradients so that you do the parameter update correctly. Else, the gradient would point in some other direction than the intended direction towards the *minimum*

Q8. What are 'out' and 'target' and what is output of criterion(out, target)?

‘target’ is our graound-truths, the actual labels of our data, and ‘out’ is predicted output label of our network. The output of criterion is the loss value between our target and predicted output and it shows how bad or good our prediction is compared to the actual class label.

Q9. What is 'loss.backward()' used for?

It back propagates the loss.

loss.backward() computes dloss/dx for every parameter x which has requires_grad=True. These are accumulated into x.grad for every parameter x.

Calling .backward() mutiple times accumulates the gradient (by addition) for each parameter. This is why you should call optimizer.zero_grad() after each .step() call.

Q10. What is 'optimizer.step()' used for?

It updates the paramets and will adjust the weights for us. optimizer.step updates the value of x using the gradient x.grad.