2. Regression with linear neurons:

The demo code given in https://github.com/pytorch/examples/blob/master/regression/main.py is learning a randomly generated polynomial regression function with one full connected layer.

The function is already generated in code itself as *y = w4 * x^4 + w3 * x3 + w2 * x^2 + w1 * x + b*, then we're generating examples in batches of size 32 synthetically using this function and then we are passing this data to single linear layer neural network. Learning process is based on stochastic (minbatch) gradient descent algorithm with smooth L1 loss function. The result is follows with the learning rate = 0.1 (code is in demo.py)

Loss: 0.000746 after 676 batches

==> Learned function: y = +2.54 x^4 +7.38 x^3 -6.24 x^2 -4.71 x^1 +2.70

==> Actual function: y = +2.59 x^4 +7.24 x^3 -6.31 x^2 -4.65 x^1 +2.72

2.a

By replacing the parameters update logic in above code (demo.py) with torch.optim.SGD, I've created new script in Part_2a.py. These are the results with different learning rates:

i) Learning rate : 0.1

Loss: 0.000288 after 296 batches
==> Learned function: y = -3.56 x^4 -1.76 x^3 -0.16 x^2 -0.05 x^1 +1.77
==> Actual function: y = -3.53 x^4 -1.74 x^3 -0.16 x^2 -0.05 x^1 +1.76

ii) Learning rate : 0.05

Loss: 0.000909 after 734 batches
==> Learned function: y = -1.17 x^4 -11.06 x^3 -6.84 x^2 +2.47 x^1 -5.48
==> Actual function: y = -1.13 x^4 -11.16 x^3 -6.86 x^2 +2.49 x^1 -5.43

iii) Learning rate : 0.15

Loss: 0.000715 after 674 batches
==> Learned function: y = +3.15 x^4 -4.14 x^3 +2.28 x^2 +6.65 x^1 -1.13
==> Actual function: y = +3.08 x^4 -4.18 x^3 +2.32 x^2 +6.66 x^1 -1.10

Based on the findings above we can say that learning rate = 0.1 is close to the optimal value.

2.b

Fitting linear model to the given data requires one single linear layer neural network. Code was submitted for this in Part_2b.py. As the data is very less, we may not get the exact values we get when we fit linear regression model **inv(X'X) * X' * y.** The single layer in the code is trained with minibatch stochastic gradient descent algorithm till it converges loss below 0.05

The learnt parameters with single linear layer are **w1 = +0.79, w2 = +1.56 bias = +31.00.**

Note: These values change for every run of the training because we are finding local optima with perceptron training, where as in linear regression we get global minimum solution.

With the above weights, test set predictions are **41.9915, 46.7107, 54.5592**

2.c

For the same data as in 2.b when we try to fir linear regression model **inv(X'X) * X' * y,** we get global solution because we solve convex function with total least squares as error function. The code is in Part_2c.ipynb. In this we get parameters as **w1 = 0.65, w2 = 1.10, bias = 31.98** and these values are always consistent as they are global optimal solution. With the above parameters we get predictions for given test dataset **40.32, 44.03, 49.96.**

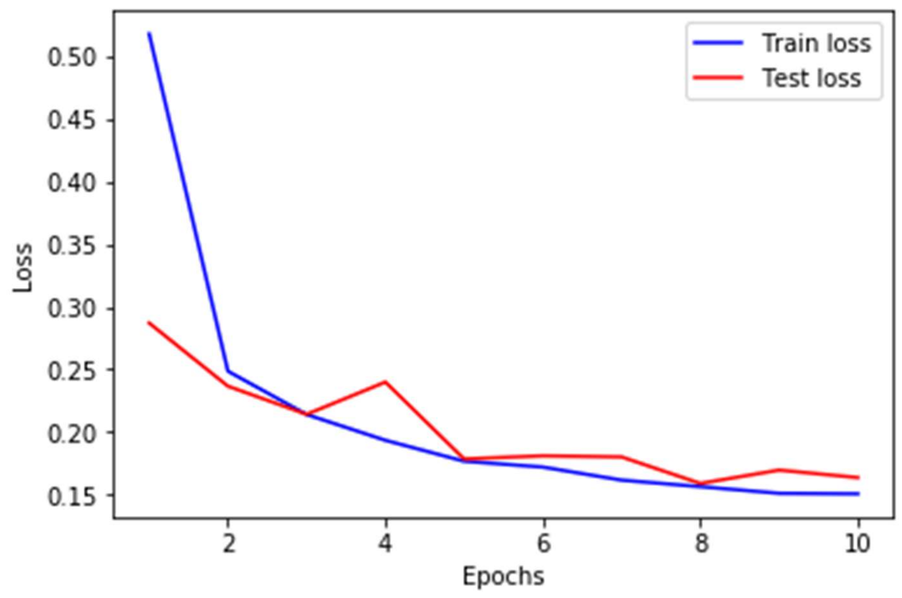
We can see significant difference in weights parameters values in perceptron training in 2.b because every time when we run, we are trying to fit a linear model to the data and it may not be the best explainable model and we are giving only local minimum solution, but with linear regression problem we are solving set equations for which close form solution is available for which we always get unique global optimal solution.

3) MNIST Image Classification:

3.a

The given 2-layer CNN was trained on MNIST data to classify the digits. Learning algorithm is based on stochastic gradient descent with training batch size as 64 for 10 total epochs. For every mini batch data we update the parameters with SGD algorithm with leaning rate 0.1. Once the model is trained, test data was used to predict the loss in batches with batch size 1000. The below graph shows the NLL loss at training at testing time for 10 epochs. (For every epoch, average NLL is used among all batches to plot).

Observation: We can see training loss dropped down after first epoch and continuously till the end, this is because if dataset is large enough then single epoch with SGD is "good enough" to learn the parameters and slowly after few epochs, model starts to overfit to the data. This is the reason why after the initial epochs, test loss started to increase.



3.b

To predict the best model I defined a metric 'classification accuracy' as number of images that are correctly classified in train and test datasets. Once every batch is trained, we get the total misclassifications in that batch and add them to existing to get total errors in whole training set. The following snippet in **Part_3b.py** returns errors in batch:

```
def get_total_misclassifications(softmax_output, target):
  '''
  Return the total classification errors in batch
  '''
  total_correctly_classified = torch.sum(torch.eq(torch.argmax(softmax_output, dim=1),
target))
  total_labels = target.shape[0]
  return total_labels - total_correctly_classified.item()
```

Once the metric is defined we need to tune hyper parameters based on it. In the given learning problem we have hyper parameters 'batch_size', 'lr', 'momentum', 'weight_decay' in which last three are parameters for optimizer optim.SGD. With multiple trail and error choices, I've seen the best metrix results at following values:

batch_size = 128
lr = 0.08
momentum = 0.2
weight_decay = 0.001

At this setting, I've observed the train and test errors as:
epoch 1, train error 0.21, test error 0.09
epoch 2, train error 0.08, test error 0.07

In other runs with different hyper parameters settings, I've observed the error to be more than this, so we can say this is one of the best hyper parameters setting.

Note: As the model is relatively less complex and data size is moderately big, we can tune hyper parameters in trial and error fashion but if model is complex with many hyper parameters then we can use grid search suing 'skorch' library.

3.c

In the above model instead of simple SGD, we can try different algorithms (https://pytorch.org/docs/0.3.0/optim.html). I've tried to algorithms

1) **Adagrad**: In this, we use adaptive gradient to update the direction in in each step by taking account cumulative gradients sum and reducing the step size in that direction so that more progress is made in less slope areas
   By using optim.Adagrad with parameters below,
   batch_size=64
   ada_lr =0.01
   ada_lr_decay = 0.001
   weight_decay = 0.001,
   model is converged with below errors
   epoch 1, train error 0.16, test error 0.09
   epoch 2, train error 0.09, test error 0.08

2) **LBFGS**: Low memory BFGS is second order derivative method which uses Quasi-Newton methods to approximate inverse Hessian matrix and then use that for optimization step
   $$\theta_{t+1} = \theta_t - \epsilon^* H^{-1} g$$
   By using optim.LBFGS with batch_size = 256 and default values for optim.LBFGS hyper-params, obtained below errors
   epoch 1, train error 1.41, test error 0.05
   epoch 2, train error 0.94, test error 0.06

   Observation: As the number of hyper parameters increase, it becomes difficult to configure the optimizer even though the algorithms increase speed of convergence with proper values. Here, we had number of hyper parameters for the optimizers like:

   Params(SGD) < Params(Adagrad) < Params(LBFGS)

   So, the model configuration also becomes difficult in the same order.