CS 7150: Deep Learning — Spring 2021— Paul Hand

HW 2 — revised
Due: Friday March 5, 2021 at 5:00 PM Eastern time via Gradescope

Names: [Sonal Jain, Gourang Patel, Sanjan Vijayakumar]

You will submit this homework in groups of 2. You may consult any and all resources. Note that some of these questions are somewhat vague by design. Part of your task is to make reasonable decisions in interpreting the questions. Your responses should convey understanding, be written with an appropriate amount of precision, and be succinct. Where possible, you should make precise statements. For questions that require coding, you may either type your results with figures into this tex file, or you may append a pdf of output of a Jupyter notebook that is organized similarly. You may use code available on the internet as a starting point.

Question 1. *Perform experiments that show that BatchNorm can (a) accelerate convergence of gradient based neural network training algorithms, (b) can permit larger learning rates, and (c) can lead to better neural network performance. You may select any context you like for your experiments.*

Clearly explain your experimental setup, results, and interpretation.

Response:

Training procedure/ Experimental Setup:

In this particular problem we have created two models first one is a 2 layer fully connected neural network with ReLU activation functions at the end of both the layers with 10 classes at the output layer. We are training this neural network on MNIST data which is a very famous dataset for hand writing recognition. The another version is a 2 layer fully connected neural network with Batch Norm layer and ReLU activation functions at the end of both the layers with 10 classes at the output layer. We are training this neural network on MNIST data which is a very famous dataset for hand writing recognition.

The training procedure includes:

1. Load Data: MNSIT data-set is downloaded from Torchvision Library and splitted into train and test dataset.

2. Model Definition: In this step, we defined the necessary parameters such as input size, hidden layer size, batch size, number of epochs, learning rate and output classes. After this, the train and test data was divided into batches and we built the neural network. The Neural network has two variations one consists of input layer, 2 fully connected hidden layer with ReLu activation in between and a output layer. We chose Stochastic gradient descent as optimizer and Cross entropy loss function. In another version consists of input layer, 2 fully connected hidden layer with BatchNorm layer and ReLu activation in between and a output layer. We chose Stochastic gradient descent as optimizer and Cross entropy loss function.

3. Training Phase: We start with the forward propagation defined by forward function (forward(self, x)) in the code where the input data passes through the network in such a way that all the neurons apply their transformation to the information they receive from the neurons

```
NeuralNet(
  (classifier): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

Figure 1: Model parameters with No Batch Normalization

```
BatchNormNet(
  (classifier): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

Figure 2: Model parameters with Batch Normalization

of the previous layer and sending it to the neurons of the next layer. When the data passes through all the neurons, final layer will have the results of label prediction.

We will use a loss function to estimate the loss and compare the true and predicted labels. Once loss is calculated (criterion(output, labels)), information is propagated back, this is called backward propagation defined by loss.backward() in the code. Starting from the output layer, that loss information propagates to all the neurons in the hidden layer that contribute directly to the output.

We then adjust the weights of connections between the neurons to make this loss as close to zero. We have used Stochastic gradient descent for this in the code denoted by optim.SGD(). This is done in batches of data in the successive iterations (epochs) of all the dataset that we pass to the net work in each iteration.

4. We stored the loss and accuracy while training for the number of iterations and saved the both the models with the best accuracy.

After performing hyperparameter tuning we obtained the best train and test accuracy for:

$$INPUTSIZE = 784$$

$$HIDDENSIZE_1 = 128$$

$$HIDDENSIZE_2 = 64$$

$$BATCHSIZE = 60$$

$$NUMEPOCHS = 15$$

We will be testing our model variations on two learning rates with and without BatchNorm

$$LEARNINGRATE = 0.01\&0.5$$

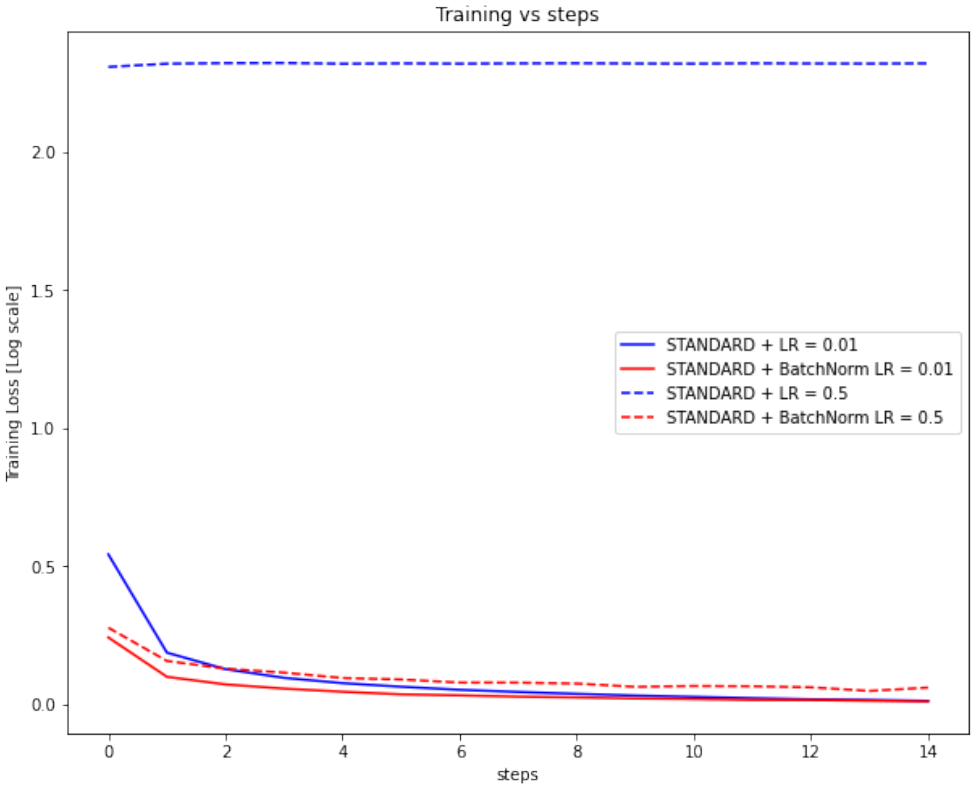


Figure 3: Training Loss VS steps

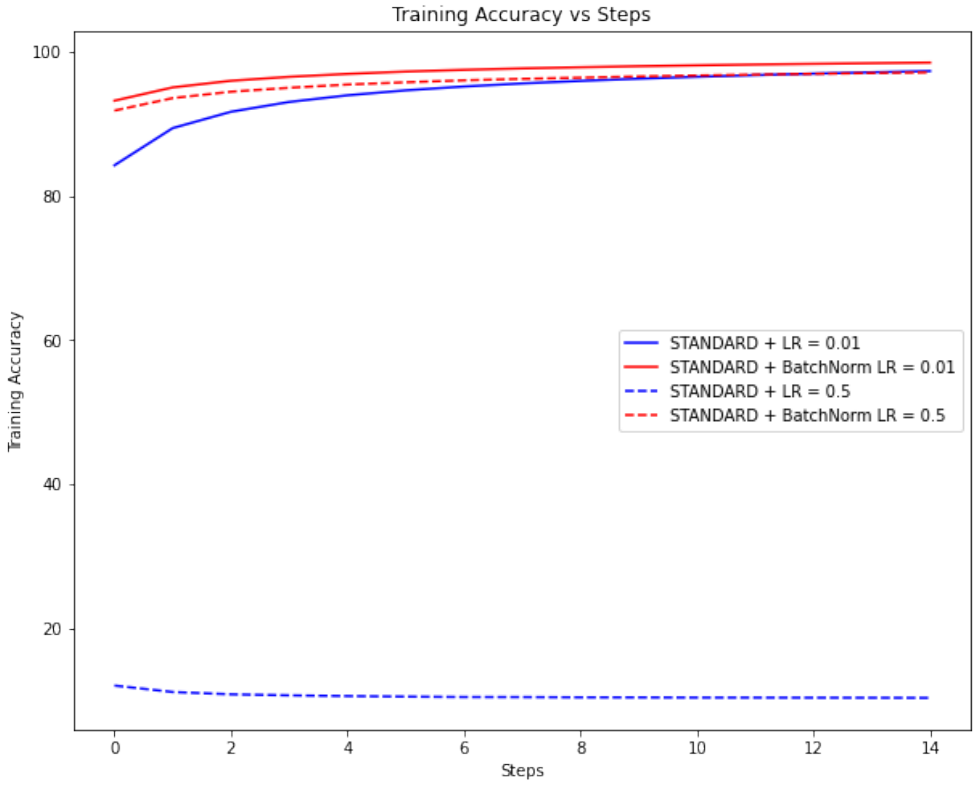


Figure 4: Training Accuracy vs Steps

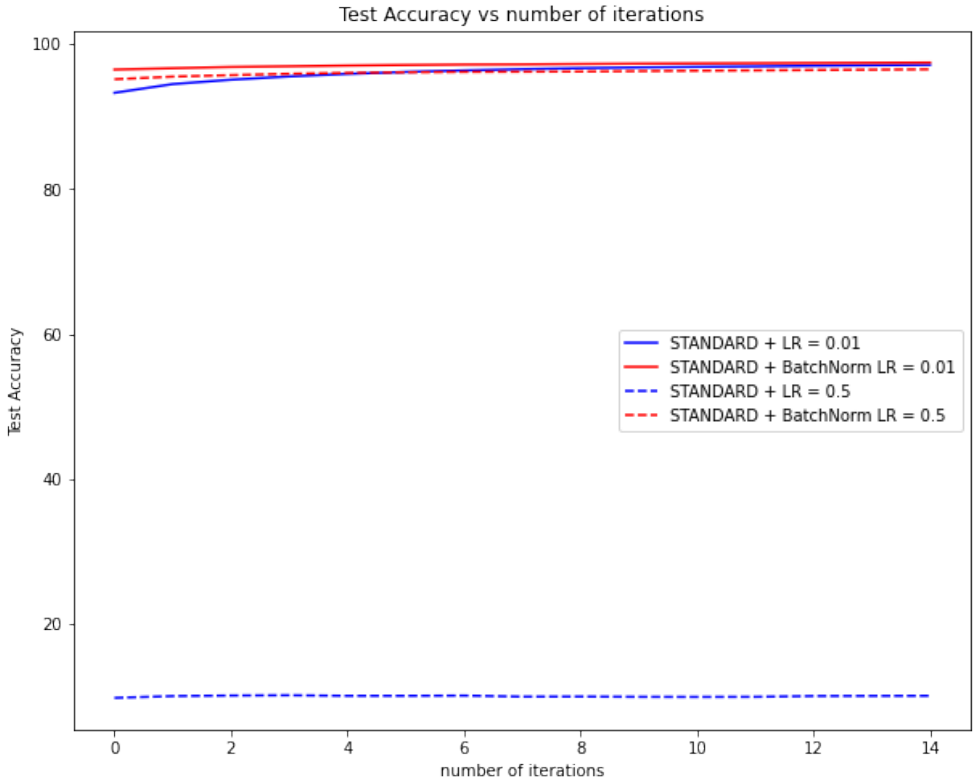


Figure 5: Test Accuracy vs Steps

Results: The best train and test accuracy we got for two variations for 15 epochs are:
Model with No Batch Normalization and Learning rate - 0.01
Epoch 12 Training Loss: 0.013566769078955985
Training Accuracy 97.31333333333333
Testing Accuracy 97.13733333333333

Model with Batch Normalization and Learning rate - 0.01
Training Loss: 0.011879017250656035
Training Accuracy 98.493
Testing Accuracy 97.45066666666666

Model with No Batch Normalization and Learning rate - 0.5
Training Loss: 2.320054642915726
Training Accuracy 10.359
Testing Accuracy 10.079333333333333

Model with Batch Normalization and Learning rate - 0.5
Training Loss: 0.061732807612632314
Training Accuracy 97.12777777777778
Testing Accuracy 96.526

Plotting Training Loss, Training Accuracy, Test Accuracy for Standard Model and Standard Model + Batch Normalization with two learning rates – 0.01, 0.5

1.Training Loss with Number of Steps Refer to the figure 3

2.Training Accuracy with Number of Steps Refer to the figure 4

3.Test Accuracy with Number of Steps Refer to the figure 5

Observation :
1.Training Loss with Number of Steps We can observe in figure 3 that the training loss doesn't decrease for Standard + LR = 0.5, and it is the best for Standard + BatchNorm = 0.01. We expect that our SGD won't perform good with higher learning rate which is true in this case when LR is 0.5. We can say that BatchNorm helps in making the model perform good with higher learning rate as opposed to case when Standard model perform badly with higher learning rate.

2.Training Accuracy with Number of Steps We can observe in figure 4 that the training accuracy is highest for the standard NN with BatchNorm and Learning rate 0.01. Then after that for Standard NN with BatchNorm and Learning rate 0.5 performs good. The worst accuracy us observed in case of standard model with learning rate 0.5 as expected. The Training accuracy for Standard + LR = 0.01 significantly reaches a good level but it will gradually take more time for it to reach to that mark.

3.Test Accuracy with Number of Steps

We can observe in the figure 5 the same pattern in the case of test accuracy where standard model with BatchNorm and LR 0.01 gives the best test accuracy. Second best test accuracy we can get is in the case of Standard model with batch and Learning rate 0.5. The worst test accuracy we can get for is for standard model with learning rate 0.5 with the test accuracy decrease continuously then increase a slightly and again decreasing.

Interpretation : Based on our observations we can conclude that adding Batch Normalization would enhance our training performance, as we see a slight improvement in Training Accuracy and Test Accuracy as compared to the model with No Batch Normalization, also we are able to achieve the performance upgrade as Batch Normalization relatively lesser number of epochs to train the model.

We also can interpret based on our findings that Batch Normalization can permit higher learning rates, as when we increased the learning rate the model performed didn't get affected, in-turn it reached to the mark of Standard Model with less learning rate in less number of iterations. Thus, BatchNorm not only increases the performance of the model but also permit higher learning rates.

Question 2. *Stochastic Gradient Descent*

In this problem, you will build your own solver of Stochastic Gradient Descent. Do not use built-in solvers from any deep learning packages. In this problem, you will use stochastic gradient descent to solve

$$\min_y \frac{1}{n}\sum_{i=1}^{n}|y-x_i|^2, \tag{1}$$

where x_i is a real number for $i = 1 \dots n$.

(a) Using calculus, derive a closed-form expression for the minimizer y^*.

Response:

To find the closed form solution and find the expression of minimizer y^*, we will take a derivative w.r.t y and equate it to 0. Doing this we get,

$$\frac{df}{dy} = \frac{2}{n}\sum_{i=1}^{n}\frac{(y-x_i)*|y-x_i|}{|y-x_i|}$$

$$0 = \frac{2}{n}\sum_{i=1}^{n}\frac{(y-x_i)*|y-x_i|}{|y-x_i|}$$

This gives,

$$\sum_{i=1}^{n}\frac{(y-x_i)*|y-x_i|}{|y-x_i|} = 0$$

Which gives,

$$\sum_{i=1}^{n}(y - x_i) = 0$$

$$\sum_{i=1}^{n} y = \sum_{i=1}^{n} x_i$$

$$n * y = \sum_{i=1}^{n} x_i$$

Finally, we get the closed form solution of y.

$$y = \frac{1}{n}\sum_{i=1}^{n} x_i$$

(b) Generate points $x_i \sim \text{Uniform}[0,1]$ for $i = 1 \ldots 100$. Use Stochastic Gradient Descent with a constant learning rate to solve (1). Use $G(y) = \frac{d}{dy}|y - x_i|^2$ for a randomly chosen $i \in \{1 \ldots n\}$. Create a plot of MSE error (relative to y^*) versus iteration number for two different learning rates. Make sure your plot clearly shows that SGD with the larger learning rate leads to faster initial convergence and a larger terminal error range than SGD with the smaller learning rate.

Response:
The plot is on the next page Fig.6
We have plotted the Training loss with respect to number of steps for two variations of learning rate i.e 0.01, represented by the blue line in the plot and 0.10 represented by the orange line in the plot and run the SGD for 200 iterations.
We observe that for LR = 0.10 at the initial stage the training loss drops very quickly as compared to the training loss for LR = 0.01, as we can a steep drop in the orange line. however when we reach the termination stage, the loss with LR = 0.10 is very unstable and that represents larger terminal error as compared to the LR = 0.01 which is relatively stable as it reaches the termination.

Question 3. *Momentum, RMSProp, and Adam — revised*

Define

$$f_1(x, y) = x^2 + 0.1y^2,$$
$$f_2(x, y) = \frac{(x-y)^2}{2} + 0.1\frac{(x+y)^2}{2}.$$

In this problem you will minimize these two functions using four optimizers: GD, GD with momentum 0.9, RMSProp, and Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. For each, use learning rate 10^{-3}. When optimizing f_1, initialize at $(1, 1)$. When optimizing f_2, initialize at $(\sqrt{2}, 0)$.

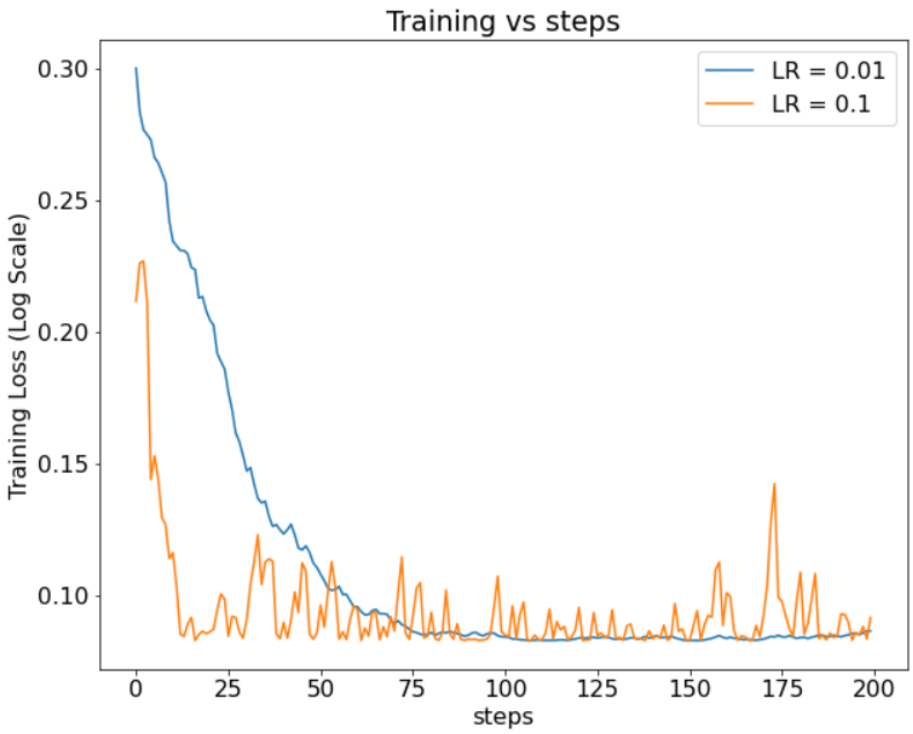


Figure 6: Training loss vs steps

(a) Before numerically solving these 8 optimization problems, guess the ranking (from fastest to slowest) of the rates of convergence of the 8 solutions. Explain your reasoning. You might guess that some converge at the same rate as others. (This subproblem will be graded only on effort and not on correctness).

Response:

Based on our intuition and understanding of various Optimizers we can comment on the convergence rate.

For a start, we observed the Hessian matrices for both the functions and observed that for both the functions, the hessian matrices are positive semidefinite, hence we can say that we are solving a convex optimization problem in both the cases, that is a local or global minimum exists. We also observed the curvature of F1 is steep then F2

We then observed all this 4 optimizers, and compared them for both F1 and F2, from our therotical understanding from the papers, since SGD with momentum works on a phenomenon of momentum [ref context: rolling a ball from a hill]. We believe that it would converge the fastest than all the other three methods. And for both F1 and F2 it will converge at the same rate as there are no adaptive learning rates in GD and GD with Momentum.

Comparing the other three – SGD, RMSProp, ADAM:

If we compare RMSProp with SGD, in RMSProp we compute the running average of the most recent gradients and then normalize the running gradient with the average, thus giving more weights to the recent gradients. Thus, RMSProp will easily converge and

reach towards minima before GD. So we believe, ConvergenceRate(GD) < ConvergenceRate(RMSProp)
Also, since both Adam and RMSProp have adaptive learning rates, so both will be able to converge to minima relatively quickly then GD. However,conceptually we believe that the convergence rate for F1 and F2 will vary for both these cases, as in steep curvatures, we need lower learning rates for F1 would converge faster for both Adam and RMSprop.

So the order of convergence would be,
(GD with Momentum (F1) = SGD with Momentum (F2)) > [(RMSProp(F1) > RMSProp(F2) = (Adam(F1) = Adam(F2))] > [GD(F2) = GD(F1)]

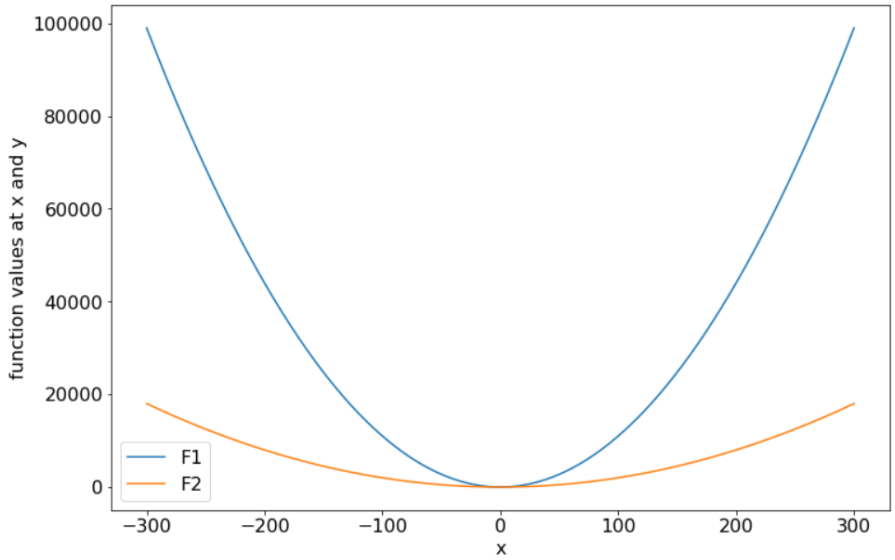


Figure 7: Curvature of f_1 and f_2

(b) Numerically solve the 8 optimization problems described above. Plot the objective as a function of iteration count. Perform at least 2000 iterations. You are encouraged to use the built-in optimizers for these algorithms in PyTorch.

Response:

(c) For each optimizer, comment on whether convergence rate was the same for f_1 and f_2. Why was it the same or why was it different? Explain. You might find it useful break up the explanation in terms of a first phase and a second phase.

From the above part, we can conclude that the convergence rate of the F1 and F2 for SGD and SGD with momentum is exactly same but it differs in the case of RMSProp and ADAM
Response:

We plotted the curvature of F1 and F2 in fig.7, and we see that F1(blue line) has more steep curvature the F2(orange line), we will try to analyze the given plots in terms of curvature shape and adaptivity of learning rates for RMSProp and ADAM.

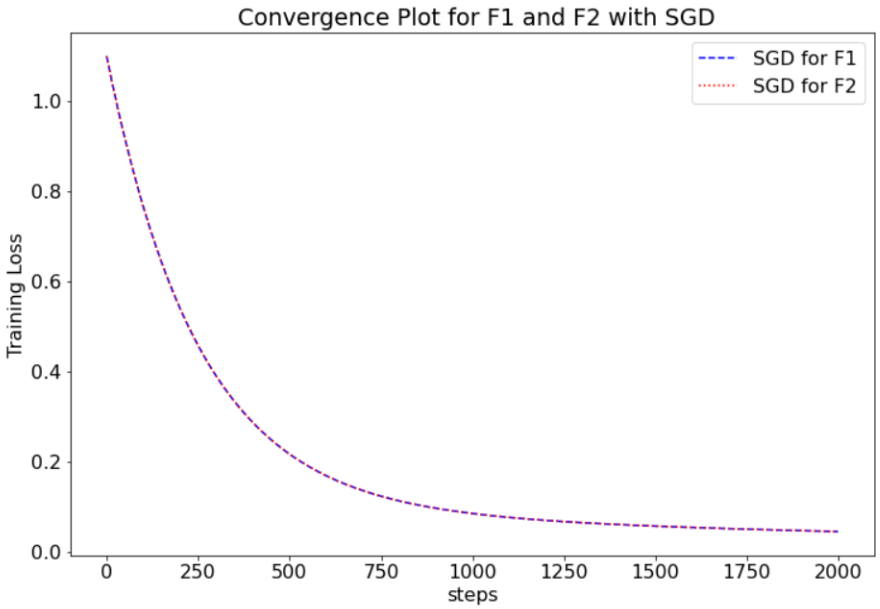


Figure 8: Training Loss vs steps

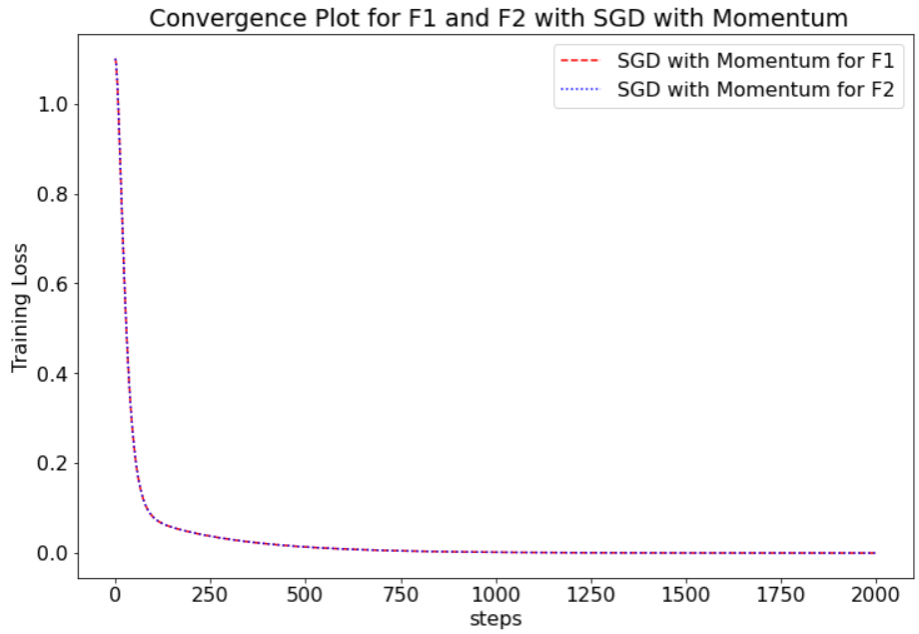


Figure 9: Training Loss vs steps

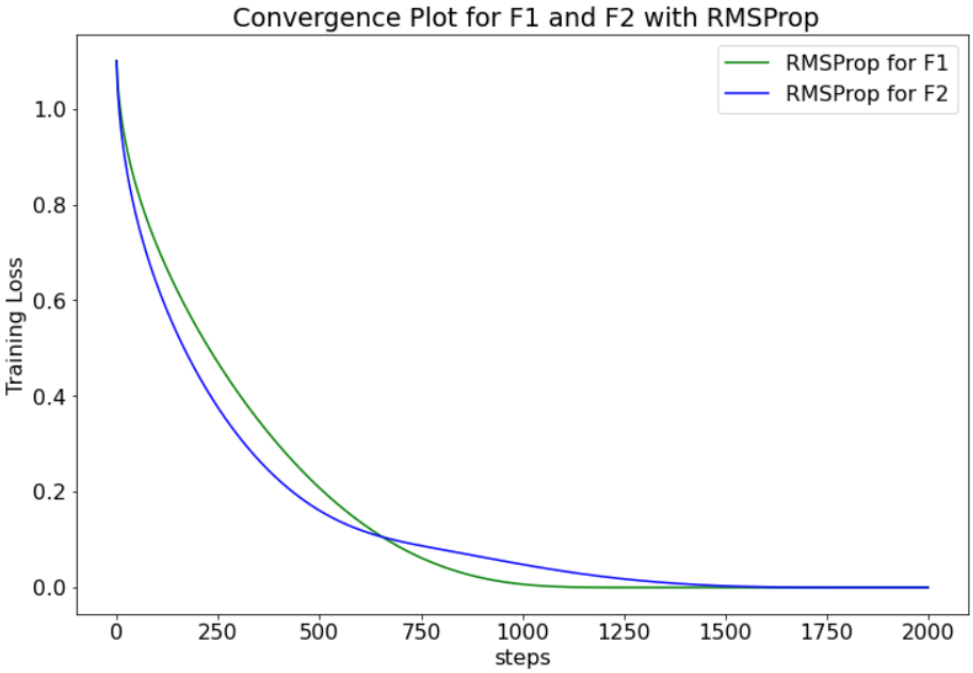


Figure 10: Training Loss vs steps

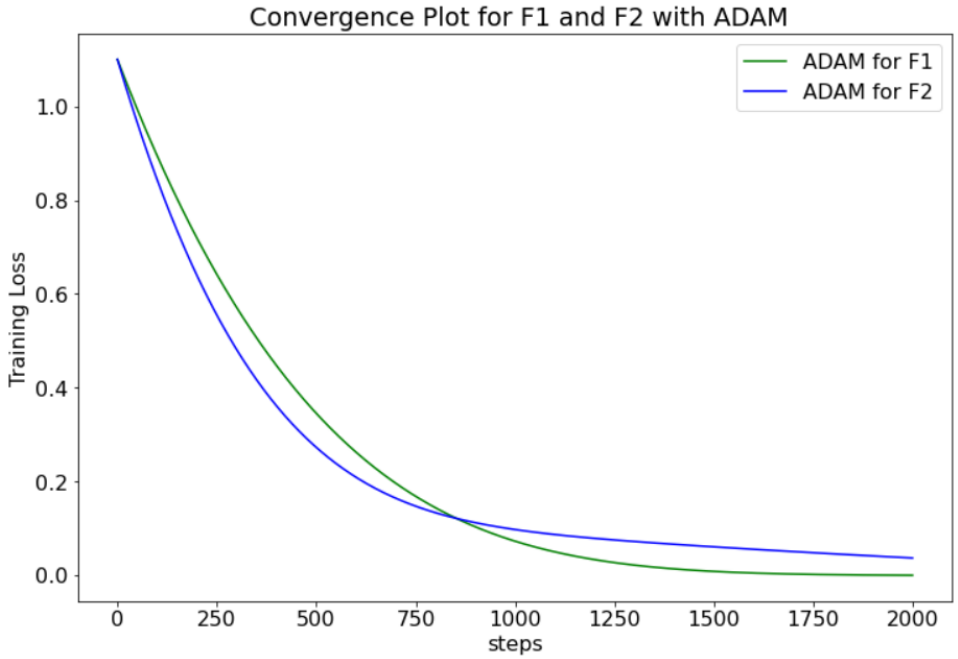


Figure 11: Training Loss vs steps

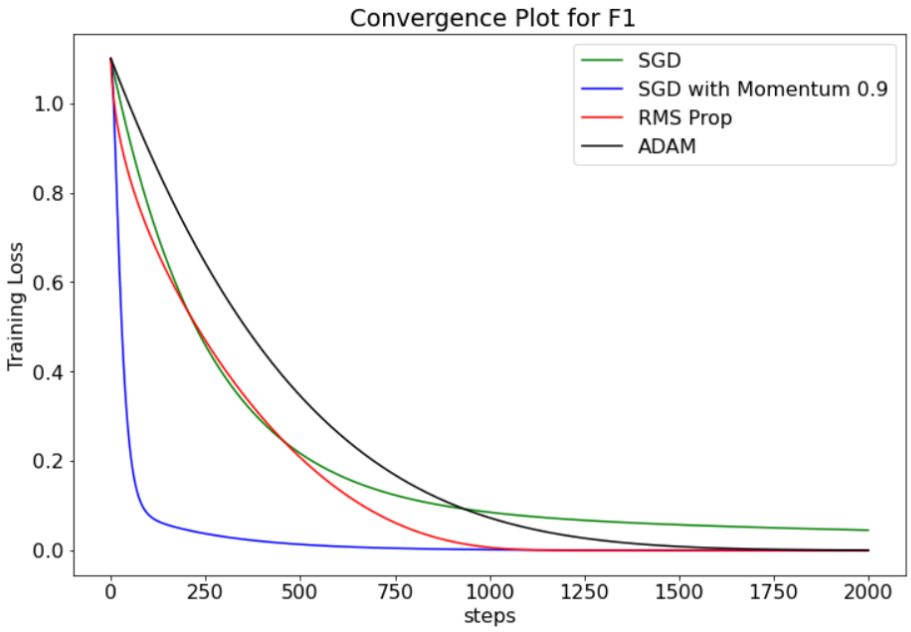


Figure 12: Training Loss vs steps

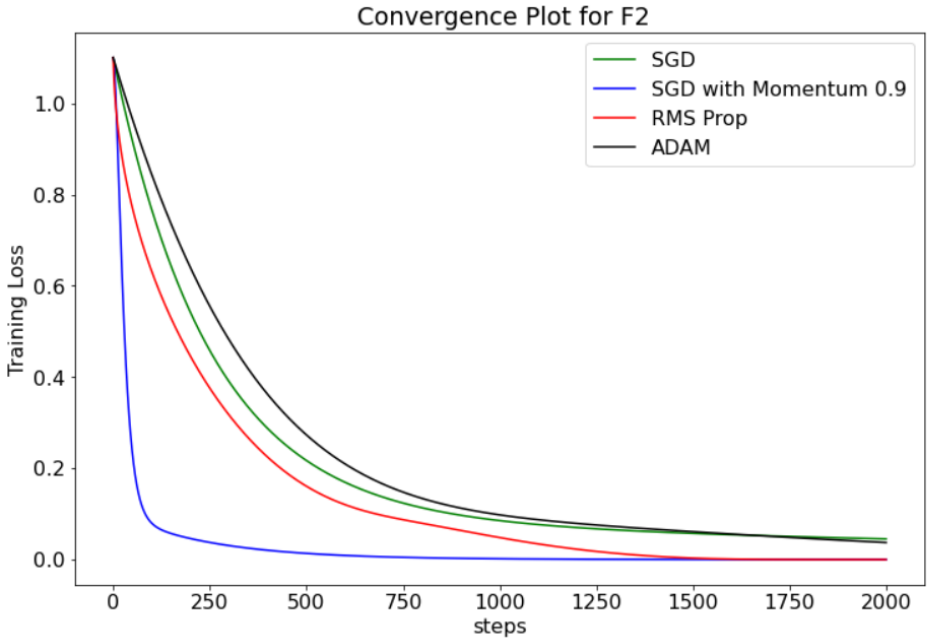


Figure 13: Training Loss vs steps

1. We observed same convergence rate for GD and SGD + Momentum = 0.9 in Fig 8 and Fig 9 - Since the starting points are same for both f1 and f2 as f(1,1) = 1.1 and $f2(\sqrt{2}, 0)$ = 1.1, and there is no updates in learning rate in SGD and SGD with momentum, therefore we believe that it will follow the rolling ball concept, and the curvature for both F1 and F2 being the same, will lead to same convergence rate in both GD and SGD + Momentum.
Also, in GD convergence rate is same for both F1 and F2 as the curvature of F1 and F2 is convex in nature and there are no updates in the learning rate as well. However, SGD + Momentum will converge faster than GD, due to the introduced momentum concept as it will follow the direction of gradient.

2. We observe different convergence rate for RMSProp and Adam for F1 and F2 in Fig 10 and Fig 11, and the pattern is similar as F2 initially drops quickly, but with increasing iterations F1 will perform better and converges faster. Similar pattern is observed in both F1 and F2.
We see this difference mainly due to the shape of the curvature, as F1 is more steep than F2 and we know that for steep curvatures we need lower learning rates. Therefore, during the start since F2 is not steep it works well with the higher step size and performs well, and as the iterations increase the step size decrease gradually, due to adaptive learning rate in RMSProp and Adam, this decrease in learning rate aids F1 to perform better and converge quickly eventually than F2. As F2 would perform well for higher learning rates, as the curvature is not that steep.

3. Fig 12 and Fig 13, represents the combined plot for all the optimizers for F1 and F2 respectively. We see that SGD with Momentum converges the fastest. Adam and RMSProp gradually follows the same pattern for both F1 and F2.

Perform experiments that show that BatchNorm can
(a) accelerate convergence of gradient based neural network training algorithms, (b) can permit larger learning rates, and
(c) can lead to better neural network performance. You may select any context you like for your experiments.

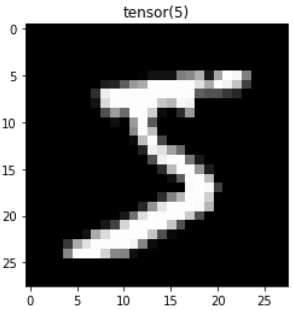
```
import torchvision
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torchvision import transforms
import torch.utils.data as Data
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_squared_error
```

```
from six.moves import urllib
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
```

```
if not(os.path.exists('./mnist/')) or not os.listdir('./mnist/'):
    DOWNLOAD_MNIST = True

train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=True,
)

test_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=False,
    transform=torchvision.transforms.ToTensor(),
    download=True,
)
```

```
print("Training Data Size: {}".format(train_data.data.size()))
print("Training Data Labels Size : {}".format(train_data.targets.size()))
plt.imshow(train_data.data[0].numpy(), cmap = 'gray')
plt.title(train_data.targets[0])
plt.show()
```

```
Training Data Size: torch.Size([60000, 28, 28])
Training Data Labels Size : torch.Size([60000])
```



Parameters for the Model

```
## CNN with two hidden layers and Relu Activation for MNIST
## training parameters

INPUT_SIZE = 784
HIDDEN_SIZE_1 = 128
HIDDEN_SIZE_2 = 64
BATCH_SIZE = 60
NUM_EPOCHS = 50
# LEARNING_R = 0.1
NUM_CLASSES = 10
```

```
seed = 1234
```

```
torch.manual_seed(seed)
```

```
<torch._C.Generator at 0x7f034c87d510>
```

```
train_data_loader = Data.DataLoader(
    dataset = train_data,
    batch_size = BATCH_SIZE,
    shuffle = True
    )
test_data_loader = Data.DataLoader(
    dataset = test_data,
    batch_size = BATCH_SIZE,
    shuffle = True
)
```

```
## Visualizing the test data
batch = next(iter(test_data_loader))
samples = batch[0][:5]
y_true = batch[1]

for i,sample in enumerate(samples):
  plt.subplot(1,5,i+1)
  plt.title("Number %i" %y_true[i])
  plt.imshow(sample.numpy().reshape((28,28)), cmap = 'gray')
  plt.axis("off")
```

Neural Network Model Architecture without Batch Normalization

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size1,hidden_size2, num_classes):
        super(NeuralNet, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(input_size, hidden_size1),  # 28 x 28 = 784 flatten the input image
            nn.ReLU(),
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU(),
            nn.Linear(hidden_size2, num_classes)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

Neural Network Model Architecture with Batch Normalization

```
class BatchNormNet(nn.Module):
    def __init__(self, input_size, hidden_size1,hidden_size2, num_classes):
        super(BatchNormNet, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.BatchNorm1d(hidden_size1), #applying batch norm
            nn.ReLU(),
            nn.Linear(hidden_size1, hidden_size2),
            nn.BatchNorm1d(hidden_size2),
            nn.ReLU(),
            nn.Linear(hidden_size2, num_classes)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

```
model_1 = NeuralNet(INPUT_SIZE, HIDDEN_SIZE_1, HIDDEN_SIZE_2, NUM_CLASSES)
model_1
```

```
NeuralNet(
  (classifier): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
```

```
          (2): Linear(in_features=128, out_features=64, bias=True)
          (3): ReLU()
          (4): Linear(in_features=64, out_features=10, bias=True)
        )
      )
```

```
model_2 = BatchNormNet(INPUT_SIZE, HIDDEN_SIZE_1, HIDDEN_SIZE_2, NUM_CLASSES)
model_2
```

```
BatchNormNet(
  (classifier): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

```
def train_test(NUM_EPOCHS, MODEL_TYPE, LEARNING_R):

  loss_list = []
  accuracy = []
  iterations = []
  running_loss = 0.0
  total_train = 0.0
  total_test = 0.0
  test_accuracy_list = []
  correct_test = 0.0
  correct_train = 0.0
  test_accuracy = 0.0
  optimizer = optim.SGD(MODEL_TYPE.parameters(), lr = LEARNING_R, momentum=0.9)
  criterion = torch.nn.CrossEntropyLoss()

  for epoch in range(NUM_EPOCHS):
    for i, (image, labels) in enumerate(train_data_loader):
      train = image.reshape(-1, 28*28)
      labels = labels
      # print(train, labels)
      output = MODEL_TYPE(train)
      optimizer.zero_grad()
      loss = criterion(output, labels)

      loss.backward()
      optimizer.step()

      # print statistics
      running_loss += loss.item()
      _, predicted = torch.max(output.data,1)
      total_train += labels.size(0)
      correct_train += predicted.eq(labels.data).sum().item()
      train_accuracy = 100 * correct_train / total_train

    for i, (image, labels) in enumerate(test_data_loader):
      images = image.reshape(-1, 28*28)
      output = MODEL_TYPE(images)
      _, pred = torch.max(output,1)
      total_test += labels.size(0)
      correct_test += pred.eq(labels.data).sum().item()
    test_accuracy = 100 * correct_test / total_test
    print (f"\nEpoch {epoch+1}\nTraining Loss: {running_loss/1000}\nTraining Accuracy {train_accuracy}\nTesting Accuracy {test_accuracy}")
    loss_list.append(running_loss/1000)
    accuracy.append(train_accuracy)
    test_accuracy_list.append(test_accuracy)
    running_loss = 0.0

  return loss_list, accuracy, test_accuracy_list



loss_list, accuracy, test_accuracy_list = train_test(15, model_1, 0.01)
```

```
Epoch 1
Training Loss: 0.5435802626833319
Training Accuracy 84.245
Testing Accuracy 93.28

Epoch 2
Training Loss: 0.18813127533718943
Training Accuracy 89.40416666666667
Testing Accuracy 94.485

Epoch 3
Training Loss: 0.12843033196032047
Training Accuracy 91.67166666666667
```

```
Testing Accuracy 95.08666666666667

Epoch 4
Training Loss: 0.09702961960341781
Training Accuracy 93.03208333333333
Testing Accuracy 95.5375

Epoch 5
Training Loss: 0.07783916562329977
Training Accuracy 93.95466666666667
Testing Accuracy 95.89

Epoch 6
Training Loss: 0.06489027580828406
Training Accuracy 94.64305555555555
Testing Accuracy 96.16166666666666

Epoch 7
Training Loss: 0.05375874708965421
Training Accuracy 95.1745238095238
Testing Accuracy 96.36714285714285

Epoch 8
Training Loss: 0.0459937264820328
Training Accuracy 95.59958333333333
Testing Accuracy 96.54125

Epoch 9
Training Loss: 0.03917964295600541
Training Accuracy 95.95166666666667
Testing Accuracy 96.67888888888889

Epoch 10
Training Loss: 0.0329050537487492
Training Accuracy 96.2595
Testing Accuracy 96.782

Epoch 11
Training Loss: 0.02826120852230815
Training Accuracy 96.52090909090909
Testing Accuracy 96.86636363636363

Epoch 12
Training Loss: 0.023591460424591787
Training Accuracy 96.75111111111111
```

```
loss_list_batch, accuracy_batch, test_accuracy_list_batch = train_test(15, model_2, 0.01)
```

```
Epoch 1
Training Loss: 0.24308478444442153
Training Accuracy 93.20666666666666
Testing Accuracy 96.49

Epoch 2
Training Loss: 0.10070806024177
Training Accuracy 95.05166666666666
Testing Accuracy 96.675

Epoch 3
Training Loss: 0.0733083789232187
Training Accuracy 95.97055555555555
Testing Accuracy 96.85

Epoch 4
Training Loss: 0.05804960775561631
Training Accuracy 96.52
Testing Accuracy 96.94

Epoch 5
Training Loss: 0.04674879469280131
Training Accuracy 96.924
Testing Accuracy 97.044

Epoch 6
Training Loss: 0.03734981960919686
Training Accuracy 97.24694444444444
Testing Accuracy 97.12166666666667

Epoch 7
Training Loss: 0.033665731187793424
Training Accuracy 97.48666666666666
Testing Accuracy 97.18428571428572

Epoch 8
Training Loss: 0.028936308094649577
Training Accuracy 97.68708333333333
Testing Accuracy 97.20125

Epoch 9
Training Loss: 0.026172564322128892
Training Accuracy 97.84611111111111
Testing Accuracy 97.2611111111111

Epoch 10
```

```
Training Loss: 0.022291810103750323
Training Accuracy 97.992
Testing Accuracy 97.308

Epoch 11
Training Loss: 0.020365376599278534
Training Accuracy 98.11287878787878
Testing Accuracy 97.33636363636364

Epoch 12
Training Loss: 0.01726678554539103
Training Accuracy 98.22527777777778
```

```
loss_list_lr, accuracy_lr, test_accuracy_list_lr = train_test(15, model_1, 0.5)
```

```
Epoch 4
Training Loss: 2.321709380865097
Training Accuracy 10.705833333333333
Testing Accuracy 10.16

Epoch 5
Training Loss: 2.3187411880493163
Training Accuracy 10.603
Testing Accuracy 10.076

Epoch 6
Training Loss: 2.3200936992168426
Training Accuracy 10.530277777777778
Testing Accuracy 10.078333333333333

Epoch 7
Training Loss: 2.319007211923599
Training Accuracy 10.470476190476191
Testing Accuracy 10.112857142857143

Epoch 8
Training Loss: 2.3200471324920655
Training Accuracy 10.462916666666667
Testing Accuracy 9.96375

Epoch 9
Training Loss: 2.3206279332637787
Training Accuracy 10.427037037037037
Testing Accuracy 9.97888888888889

Epoch 10
Training Loss: 2.3195667493343355
Training Accuracy 10.419166666666667
Testing Accuracy 9.939

Epoch 11
Training Loss: 2.3189584126472473
Training Accuracy 10.398181818181818
Testing Accuracy 9.926363636363636

Epoch 12
Training Loss: 2.320414979696274
Training Accuracy 10.393888888888888
Testing Accuracy 9.94

Epoch 13
Training Loss: 2.31963569688797
Training Accuracy 10.382948717948718
Testing Accuracy 10.048461538461538

Epoch 14
Training Loss: 2.319215399980545
Training Accuracy 10.376190476190477
Testing Accuracy 10.065

Epoch 15
Training Loss: 2.320054642915726
Training Accuracy 10.359
Testing Accuracy 10.079333333333333
```

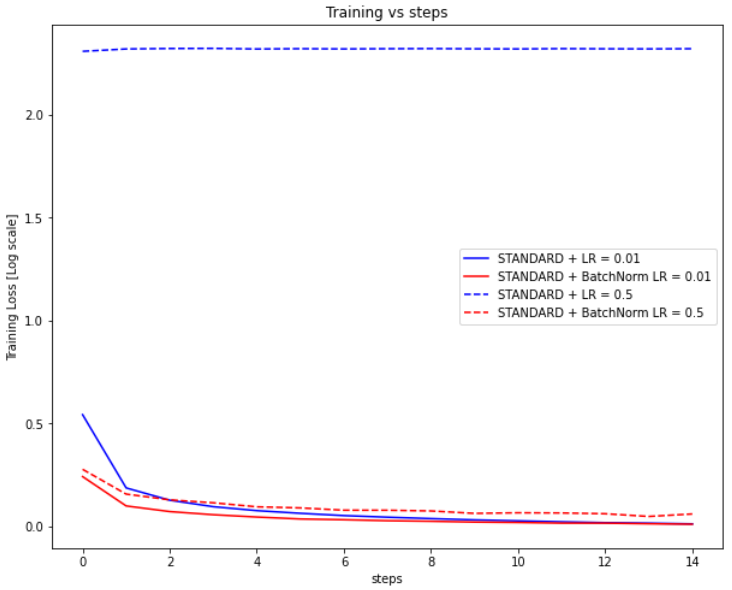
```
loss_list_batch_lr, accuracy_batch_lr, test_accuracy_list_batch_lr = train_test(15, model_2, 0.5)
```

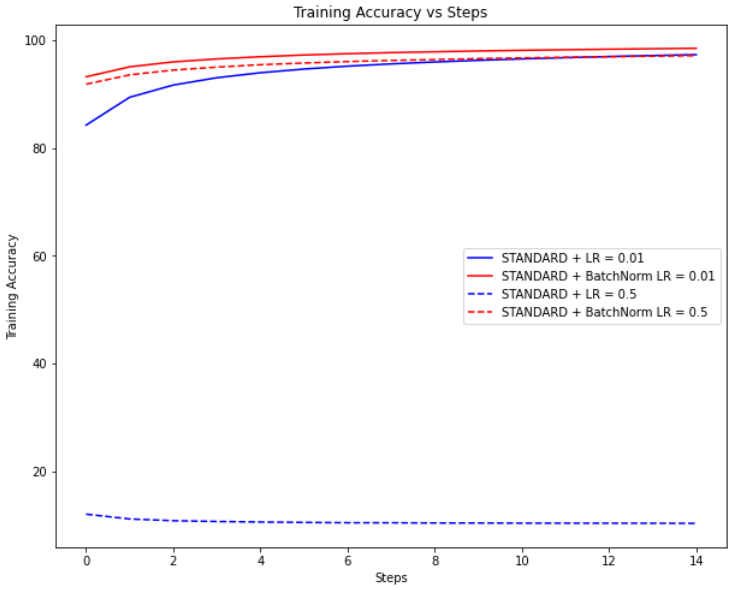
```
Epoch 1
Training Loss: 0.2784119394924492
Training Accuracy 91.83666666666667
Testing Accuracy 95.15

Epoch 2
Training Loss: 0.15816073708515616
Training Accuracy 93.565
Testing Accuracy 95.515

Epoch 3
Training Loss: 0.1302526795999147
Training Accuracy 94.45277777777778
Testing Accuracy 95.72

Epoch 4
```

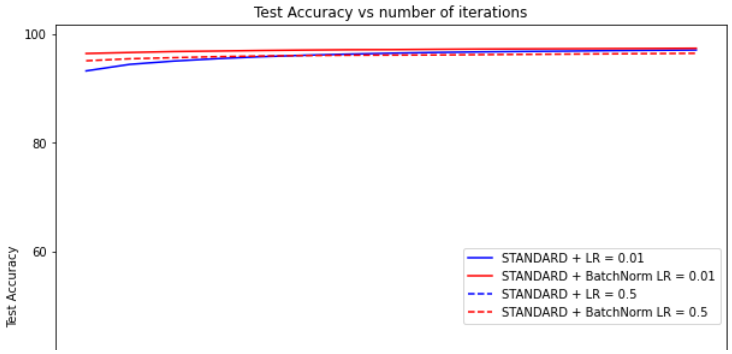
```
Training Loss: 0.11589078436011914
Training Accuracy 94.99875
Testing Accuracy 95.905

Epoch 5
Training Loss: 0.09642964972840855
Training Accuracy 95.448
Testing Accuracy 96.034

Epoch 6
Training Loss: 0.09108214095447328
Training Accuracy 95.75944444444444
Testing Accuracy 96.10833333333333

Epoch 7
Training Loss: 0.07996909010774107
Training Accuracy 96.03285714285714
Testing Accuracy 96.16714285714286

Epoch 8
Training Loss: 0.07975507021520753
Training Accuracy 96.234375
Testing Accuracy 96.19125

Epoch 9
Training Loss: 0.07634779434568009
Training Accuracy 96.40185185185184
Testing Accuracy 96.21777777777778

Epoch 10
Training Loss: 0.06454593713971553
Training Accuracy 96.57216666666666
Testing Accuracy 96.271

Epoch 11
Training Loss: 0.06750927054710337
Training Accuracy 96.70681818181818
Testing Accuracy 96.32545454545455

Epoch 12
Training Loss: 0.06653158761604482
Training Accuracy 96.8236111111111
```

Plotting Training Loss, Training Accuracy, Test Accuracy for Standard Model and Standard Model + Batch Normalization with two learning rates -- 0.01 & 0.5

```
plt.subplots(figsize = (10,8))
plt.plot(np.array(loss_list), 'b', label='STANDARD + LR = 0.01')
plt.plot(np.array(loss_list_batch), 'r', label = 'STANDARD + BatchNorm LR = 0.01')
plt.plot(np.array(loss_list_lr), 'b', label = 'STANDARD + LR = 0.5', linestyle='--')
plt.plot(np.array(loss_list_batch_lr), 'r', label = 'STANDARD + BatchNorm LR = 0.5', linestyle='--')
plt.title('Training vs steps')
plt.xlabel('steps')
plt.ylabel('Training Loss [Log scale]')
plt.legend()
# plt.show()
plt.subplots(figsize = (10,8))
plt.plot( np.array(accuracy), 'b', label='STANDARD + LR = 0.01')
plt.plot( np.array(accuracy_batch), 'r', label = 'STANDARD + BatchNorm LR = 0.01')
plt.plot( np.array(accuracy_lr), 'b', label = 'STANDARD + LR = 0.5', linestyle='--')
plt.plot( np.array(accuracy_batch_lr), 'r', label = 'STANDARD + BatchNorm LR = 0.5' ,linestyle='--')
plt.title('Training Accuracy vs Steps')
plt.xlabel('Steps')
plt.ylabel('Training Accuracy')
plt.legend()
plt.subplots(figsize = (10,8))
plt.plot(np.array(test_accuracy_list), 'b', label='STANDARD + LR = 0.01')
plt.plot(np.array(test_accuracy_list_batch), 'r', label='STANDARD + BatchNorm LR = 0.01')
plt.plot(np.array(test_accuracy_list_lr), 'b', label='STANDARD + LR = 0.5', linestyle='--')
plt.plot(np.array(test_accuracy_list_batch_lr), 'r', label='STANDARD + BatchNorm LR = 0.5', linestyle='--')
plt.title('Test Accuracy vs number of iterations')
plt.xlabel('number of iterations')
plt.ylabel('Test Accuracy')
plt.legend()
plt.show()
```







Problem 2: Stochastic Gradient Descent

In this problem, you will build your own solver of Stochastic Gradient Descent. Do not use built-in solvers from any deep learning packages. In this problem, you will use stochastic gradient descent to solve

$\min_y \frac{1}{n} \sum_{i=1}^{n} |y - x_i|^2$,

where x_i is a real number for $i = 1 \dots n$.

number of iterations

Generate points $x_i \sim \text{Uniform}[0, 1]$ for $i = 1 \dots 100$. Use Stochastic Gradient Descent with a constant learning rate to solve . Use $G(y) = \frac{d}{dy}|y - x_i|^2$ for a randomly chosen $i \in \{1 \dots n\}$. Create a plot of MSE error (relative to y^*) versus iteration number for two different learning rates. Make sure your plot clearly shows that SGD with the larger learning rate leads to faster initial convergence and a larger terminal error range than SGD with the smaller learning rate.

```
x = np.random.uniform(0, 1, 100)
x
```

```
array([0.81242289, 0.49224719, 0.50022526, 0.03673682, 0.38437515,
       0.3301572 , 0.85307694, 0.43176919, 0.15553839, 0.8711398 ,
       0.24368733, 0.88113004, 0.7915877 , 0.15809865, 0.43641085,
       0.09947189, 0.16596453, 0.26499223, 0.67069352, 0.80045897,
       0.98175961, 0.80429001, 0.39187074, 0.38444551, 0.17915751,
       0.59141489, 0.29719122, 0.37586049, 0.14586543, 0.35315941,
       0.47115241, 0.02880125, 0.95989842, 0.27436739, 0.02325197,
       0.13087981, 0.07000763, 0.86305129, 0.56284311, 0.01021442,
       0.04896813, 0.99408011, 0.84028199, 0.19169223, 0.5881976 ,
       0.6608769 , 0.43081494, 0.47946213, 0.07956622, 0.83177287,
       0.17002237, 0.80474146, 0.67965427, 0.08120251, 0.57684593,
       0.48577558, 0.96518871, 0.68625098, 0.7595727 , 0.57482186,
       0.35926645, 0.71059353, 0.40776774, 0.04389297, 0.84894579,
       0.05554327, 0.84987062, 0.18584273, 0.9240408 , 0.71613412,
       0.44361592, 0.55939728, 0.29919322, 0.47680932, 0.64446494,
       0.92575525, 0.71379881, 0.37108754, 0.27419059, 0.45438238,
       0.06040569, 0.09723543, 0.30522647, 0.80983967, 0.37319997,
       0.65785208, 0.88482931, 0.56327411, 0.78251049, 0.6075572 ,
       0.69552948, 0.49670241, 0.56691243, 0.8179933 , 0.14411382,
       0.20488686, 0.85409818, 0.17894463, 0.27378486, 0.2040781 ])
```

Stochastic Gradient Descent code to minimize the given function

```
import random
def sgd(X, learning_rate, n_epochs=200):
    y = 0  # Randomly initializing weights
    epoch = 1
    loss = []
    while epoch <= n_epochs:
        temp = random.choice(X)
        Ly = 2*(y-temp)
        y = y - learning_rate * Ly
        error = np.sum((y - X)**2)/len(X)
        loss.append(error)
        print("Epoch: %d, Loss: %.3f" %(epoch, error))
        epoch+=1
    return loss
```

```
loss = sgd(x, 0.01)
loss_1 = sgd(x, 0.1)
print("Loss with LR = 0.01", loss)
print("Loss with LR = 0.1", loss_1)
```

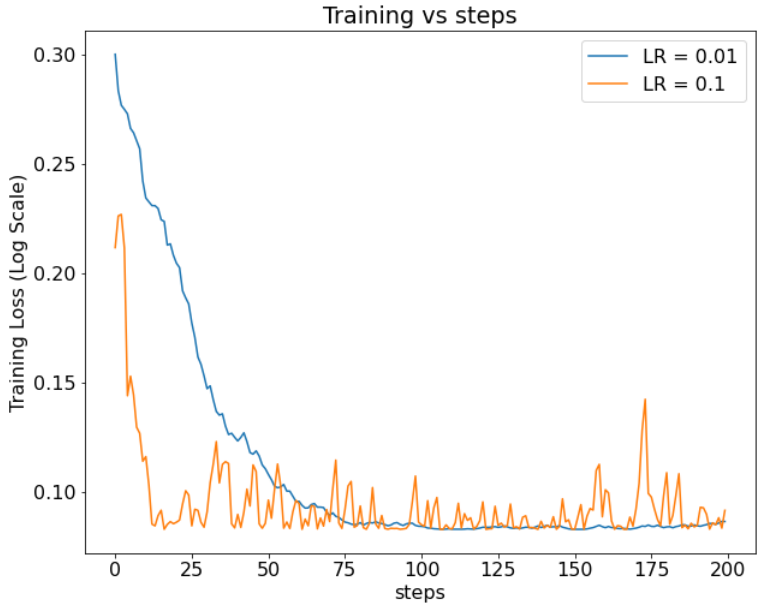
```
Epoch: 1, Loss: 0.300
Epoch: 2, Loss: 0.283
Epoch: 3, Loss: 0.277
Epoch: 4, Loss: 0.275
Epoch: 5, Loss: 0.273
Epoch: 6, Loss: 0.266
Epoch: 7, Loss: 0.264
Epoch: 8, Loss: 0.260
Epoch: 9, Loss: 0.257
Epoch: 10, Loss: 0.242
Epoch: 11, Loss: 0.234
Epoch: 12, Loss: 0.233
Epoch: 13, Loss: 0.231
Epoch: 14, Loss: 0.231
Epoch: 15, Loss: 0.230
Epoch: 16, Loss: 0.224
Epoch: 17, Loss: 0.224
Epoch: 18, Loss: 0.213
Epoch: 19, Loss: 0.213
Epoch: 20, Loss: 0.208
Epoch: 21, Loss: 0.204
Epoch: 22, Loss: 0.203
Epoch: 23, Loss: 0.192
Epoch: 24, Loss: 0.189
Epoch: 25, Loss: 0.186
Epoch: 26, Loss: 0.177
Epoch: 27, Loss: 0.171
Epoch: 28, Loss: 0.162
Epoch: 29, Loss: 0.158
Epoch: 30, Loss: 0.153
Epoch: 31, Loss: 0.147
Epoch: 32, Loss: 0.148
Epoch: 33, Loss: 0.142
Epoch: 34, Loss: 0.137
Epoch: 35, Loss: 0.135
Epoch: 36, Loss: 0.136
Epoch: 37, Loss: 0.130
Epoch: 38, Loss: 0.126
Epoch: 39, Loss: 0.127
Epoch: 40, Loss: 0.125
Epoch: 41, Loss: 0.123
Epoch: 42, Loss: 0.125
```

```
Epoch: 43, Loss: 0.127
Epoch: 44, Loss: 0.123
Epoch: 45, Loss: 0.118
Epoch: 46, Loss: 0.117
Epoch: 47, Loss: 0.119
Epoch: 48, Loss: 0.116
Epoch: 49, Loss: 0.112
Epoch: 50, Loss: 0.110
Epoch: 51, Loss: 0.108
Epoch: 52, Loss: 0.105
Epoch: 53, Loss: 0.103
Epoch: 54, Loss: 0.102
Epoch: 55, Loss: 0.102
Epoch: 56, Loss: 0.103
Epoch: 57, Loss: 0.100
```

Plotting Training Loss vs number of steps for two variations of Learning rate {0.04, 0.2}

```
plt.figure(figsize=(10,8))
plt.rcParams.update({'font.size': 16})
plt.plot(loss,  Label='LR = 0.01')
plt.plot(loss_1,  Label='LR = 0.1')
plt.title('Training vs steps')
plt.xlabel('steps')
plt.ylabel('Training Loss (Log Scale)')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f334ef9c390>
```



Problem 3

Defining function f_1

$f_1(x, y) = x^2 + 0.1y^2$

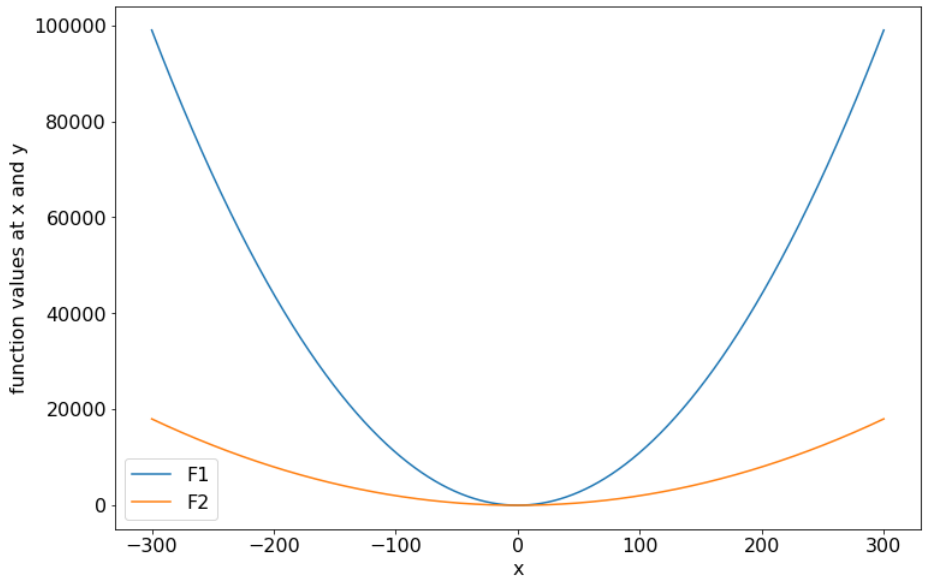
Defining function f_2

$f_2(x, y) = \frac{(x-y)^2}{2} + 0.1\frac{(x+y)^2}{2}$

We can observe that both the functions are convex in nature.

```
# Objective function
def f_1(x, y):
    return (x) ** 2 + 0.1 * (y) ** 2

def f_2(x, y):
  return ((x - y)** 2)/2 + 0.1 * ((x + y)**2)/2


x = np.arange(-300, 300, 0.01)
y = np.arange(-300, 300, 0.01)
plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(x, f_1(x,y), label = "F1")
plt.plot(x, f_2(x,y), label = 'F2')
```

```
plt.legend()
plt.xlabel("x")
plt.ylabel("function values at x and y")
plt.show()
```



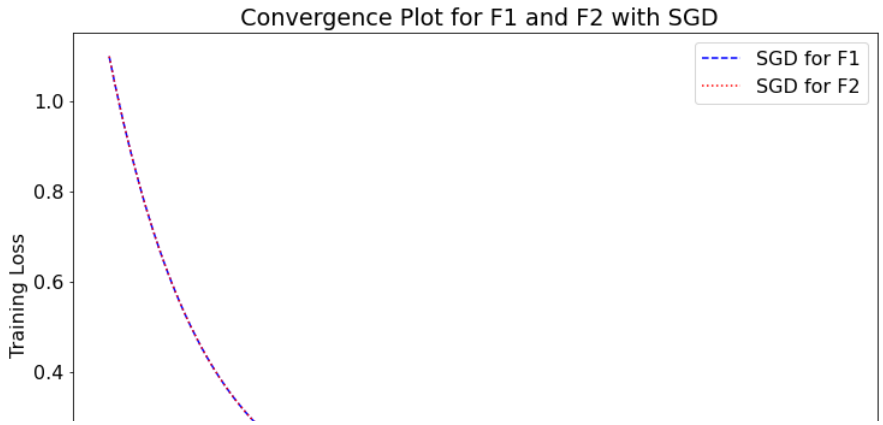
Minimizer Function

The function takes in the function, the initial values and the optimizer as arguments and returns the list that contains loss values for 2000 iterations

```
def minimizer(f, hp, optimizer):
  for param in hp:
    param.requires_grad = True
  obj_vals = []
  for i in range(2000):
      optimizer.zero_grad()
      loss = f(*hp)
      loss.backward()
      optimizer.step()
      obj_vals.append(loss)
  for param in hp:
      param.requires_grad = False
  return obj_vals
```

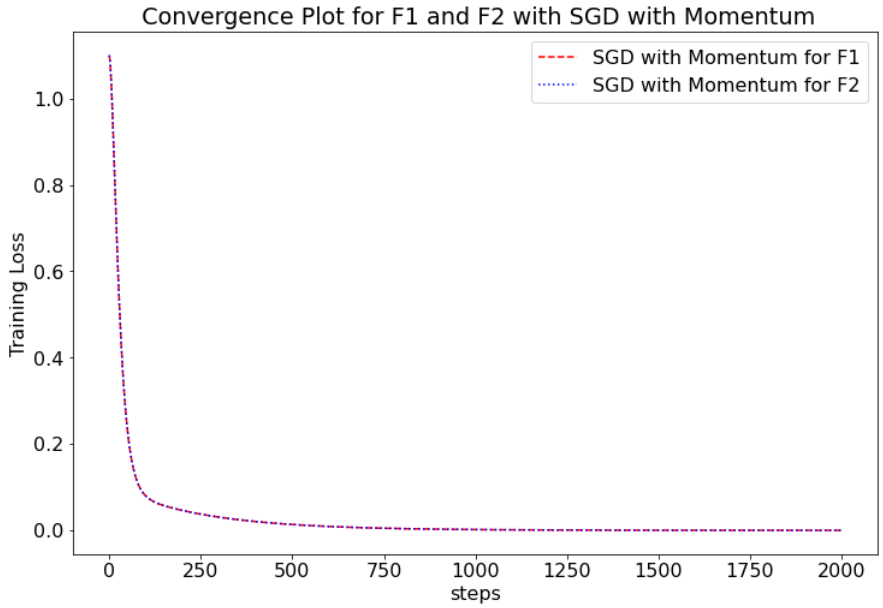
Minimizing using the SGD

```
initial_1 = torch.Tensor([1.0]), torch.Tensor([1.0])
optimizer = torch.optim.SGD([{"params": [initial_1[0]], "lr": 1e-3}, {"params": [initial_1[1]], "lr": 1e-3}])
sgd_f1_vals =  minimizer(f_1, initial_1, optimizer)

initial_2 = torch.Tensor([np.sqrt(2)]), torch.Tensor([0.0])
optimizer = torch.optim.SGD([{"params": [initial_2[0]], "lr": 1e-3}, {"params": [initial_2[1]], "lr": 1e-3}])
sgd_f2_vals =  minimizer(f_2, initial_2, optimizer)

plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(sgd_f1_vals, 'b', label = "SGD for F1", linestyle = '--')
plt.plot(sgd_f2_vals, 'r', label = "SGD for F2", linestyle = ":")
plt.title(" Convergence Plot for F1 and F2 with SGD")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



Minimizing the SGD with Momentum 0.9

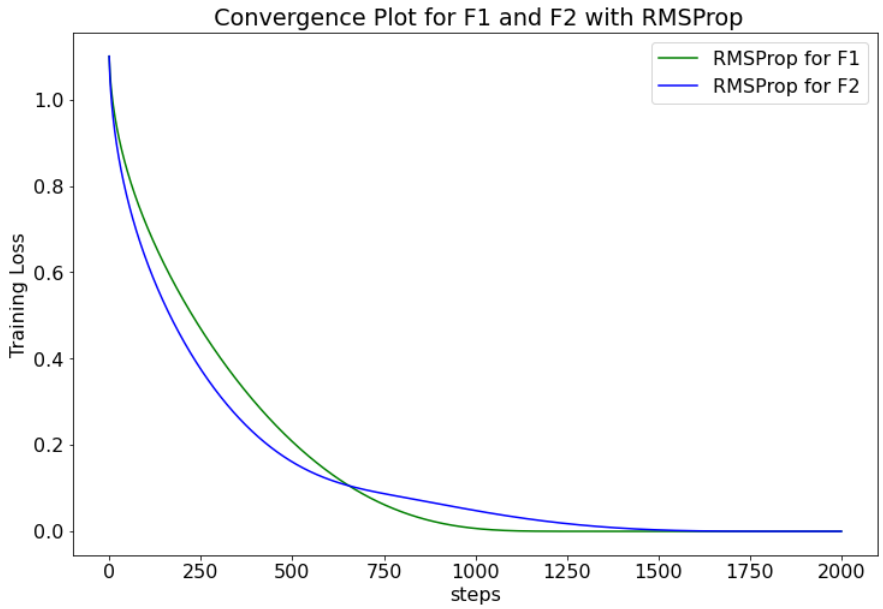
```
initial_1 = torch.Tensor([1.0]), torch.Tensor([1.0])
optimizer_1 = torch.optim.SGD([{"params": [initial_1[0]], "lr": 1e-3}, {"params": [initial_1[1]], "lr": 1e-3}], momentum = 0.9)
sgd_momentum_f1_vals =  minimizer(f_1, initial_1, optimizer_1)

initial_2 = torch.Tensor([np.sqrt(2)]), torch.Tensor([0.0])
optimizer_1 = torch.optim.SGD([{"params": [initial_2[0]], "lr": 1e-3}, {"params": [initial_2[1]], "lr": 1e-3}], momentum = 0.9)
sgd_momentum_f1_vals =  minimizer(f_2, initial_2, optimizer_1)

plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(sgd_momentum_f1_vals, 'r', label = "SGD with Momentum for F1", linestyle = '--')
plt.plot(sgd_momentum_f1_vals, 'b', label = "SGD with Momentum for F2", linestyle = ':')
plt.title(" Convergence Plot for F1 and F2 with SGD with Momentum")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



Minimizing the RMSProp

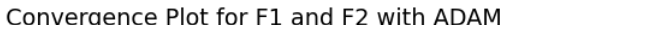
```
initial_1 = torch.Tensor([1.0]), torch.Tensor([1.0])
optimizer_2 = torch.optim.RMSprop([{"params": [initial_1[0]], "lr": 1e-3}, {"params": [initial_1[1]], "lr": 1e-3}])
rmsProp_f1_vals =  minimizer(f_1, initial_1, optimizer_2)

initial_2 = torch.Tensor([np.sqrt(2)]), torch.Tensor([0.0])
optimizer_2 = torch.optim.RMSprop([{"params": [initial_2[0]], "lr": 1e-3}, {"params": [initial_2[1]], "lr": 1e-3}])
rmsProp_f2_vals =  minimizer(f_2, initial_2, optimizer_2)

plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(rmsProp_f1_vals, 'g', label = "RMSProp for F1")
```

```
plt.plot(rmsProp_f2_vals, 'b', label = "RMSProp for F2")
plt.title(" Convergence Plot for F1 and F2 with RMSProp")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



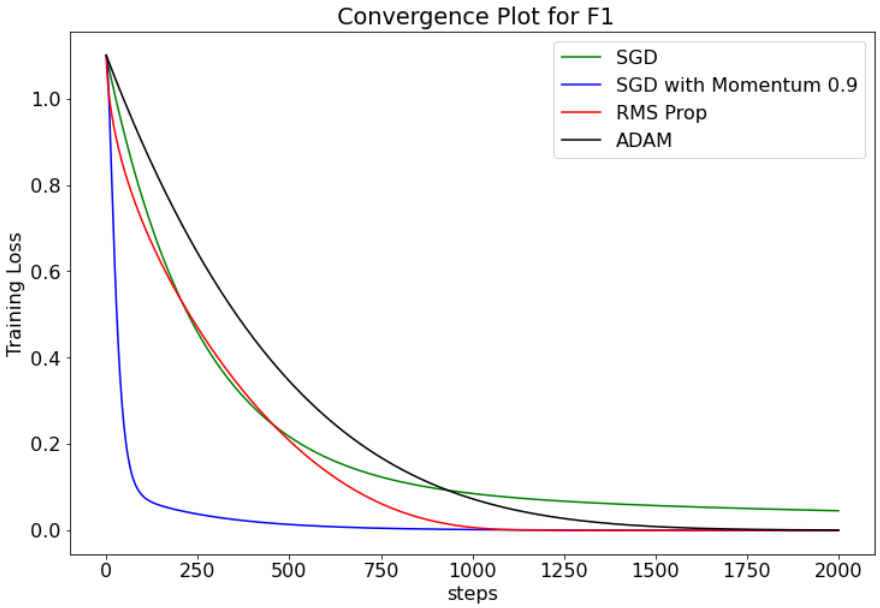
Minimizing the Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$

```
initial_1 = torch.Tensor([1.0]), torch.Tensor([1.0])
optimizer_3 = torch.optim.Adam([{"params": [initial_1[0]], "lr": 1e-3}, {"params": [initial_1[1]], "lr": 1e-3}], betas=(0.9, 0.999))
adam_f1_vals =  minimizer(f_1, initial_1, optimizer_3)

initial_2 = torch.Tensor([np.sqrt(2)]), torch.Tensor([0.0])
optimizer_3 = torch.optim.Adam([{"params": [initial_2[0]], "lr": 1e-3}, {"params": [initial_2[1]], "lr": 1e-3}], betas=(0.9, 0.999))
adam_f2_vals =  minimizer(f_2, initial_2, optimizer_3)

plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(adam_f1_vals, 'g', label = "ADAM for F1")
plt.plot(adam_f2_vals, 'b', label = "ADAM for F2")
plt.title(" Convergence Plot for F1 and F2 with ADAM")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



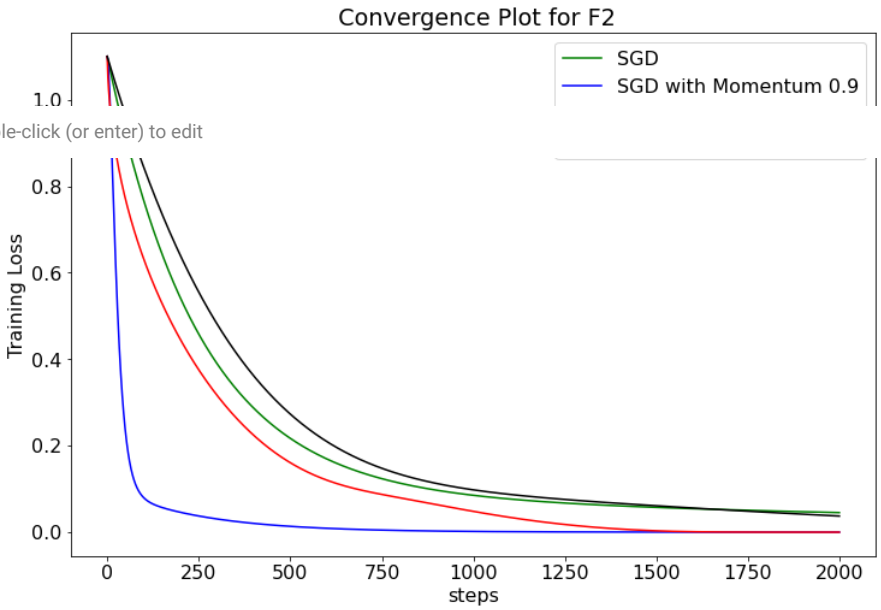
Plotting the $f_1(x, y) = x^2 + 0.1y^2$ with various varitions in optimizer

```
plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(sgd_f1_vals, 'g', label = 'SGD')
plt.plot(sgd_momentum_f1_vals, 'b', label = "SGD with Momentum 0.9")
plt.plot(rmsProp_f1_vals, 'r', label = "RMS Prop")
plt.plot(adam_f1_vals, 'black', label = "ADAM")
plt.title(" Convergence Plot for F1")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



Plotting the $f_2(x, y) = \frac{(x-y)^2}{2} + 0.1\frac{(x+y)^2}{2}$ with various varitions of optimizer

```
plt.figure(figsize=(12,8))
plt.rcParams.update({'font.size': 16})
plt.plot(sgd_f2_vals, 'g', label = "SGD")
plt.plot(sgd_momentum_f2_vals, 'b', label = "SGD with Momentum 0.9")
plt.plot(rmsProp_f2_vals, 'r' , label = "RMS Prop")
plt.plot(adam_f2_vals, 'black', label = "ADAM")
plt.title(" Convergence Plot for F2")
plt.xlabel('steps')
plt.ylabel('Training Loss')
plt.legend()
plt.show()
```



Double-click (or enter) to edit