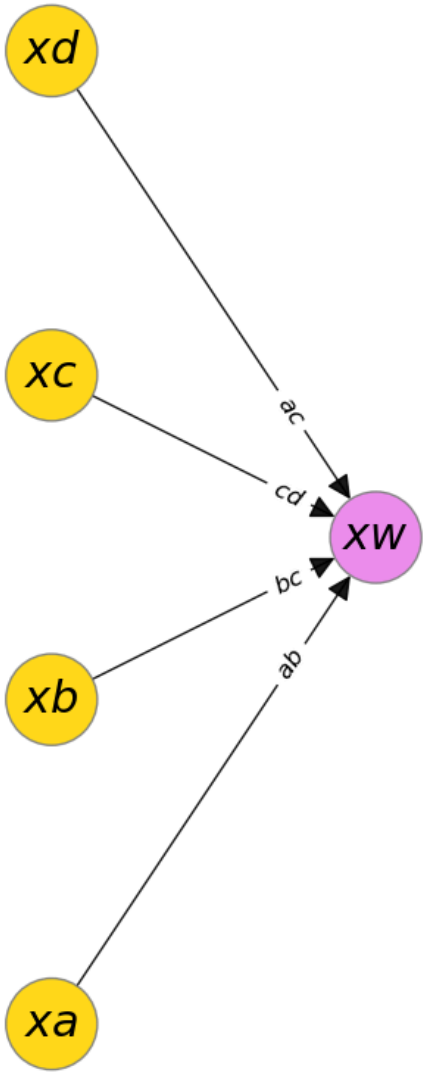
ECE60146: Homework 3 - Optimization

Pranay Chimmani, PUID: 0037630649

1 Using ComputationalGraphPrimer

1.1 Comparing one-neuron with CGP vs torch.nn

All of the below code is borrowed from ComputationalGraphPrimer package: https://engineering.purdue.edu/kak/distCGP/



Last ten lines of output: **CPG one-neuron** vs **torch.nn one-neuron**

```
[iter=39001]  loss = 0.1796
[iter=39101]  loss = 0.1812
[iter=39201]  loss = 0.1828
[iter=39301]  loss = 0.1855
[iter=39401]  loss = 0.1880
[iter=39501]  loss = 0.1763
[iter=39601]  loss = 0.1733
[iter=39701]  loss = 0.1829
[iter=39801]  loss = 0.1738
[iter=39901]  loss = 0.1762
```

CPG one-neuron; last 10 losses

```
[iter=39001]  loss = 0.8921
[iter=39101]  loss = 0.9144
[iter=39201]  loss = 1.0346
[iter=39301]  loss = 0.9973
[iter=39401]  loss = 0.9522
[iter=39501]  loss = 0.9266
[iter=39601]  loss = 0.8729
[iter=39701]  loss = 1.0250
[iter=39801]  loss = 0.9344
[iter=39901]  loss = 0.9151
```

torch.nn one-neuron; last 10 losses

Display of the training loss versus the training iterations: **CPG one-neuron** vs **torch.nn one-neuron**

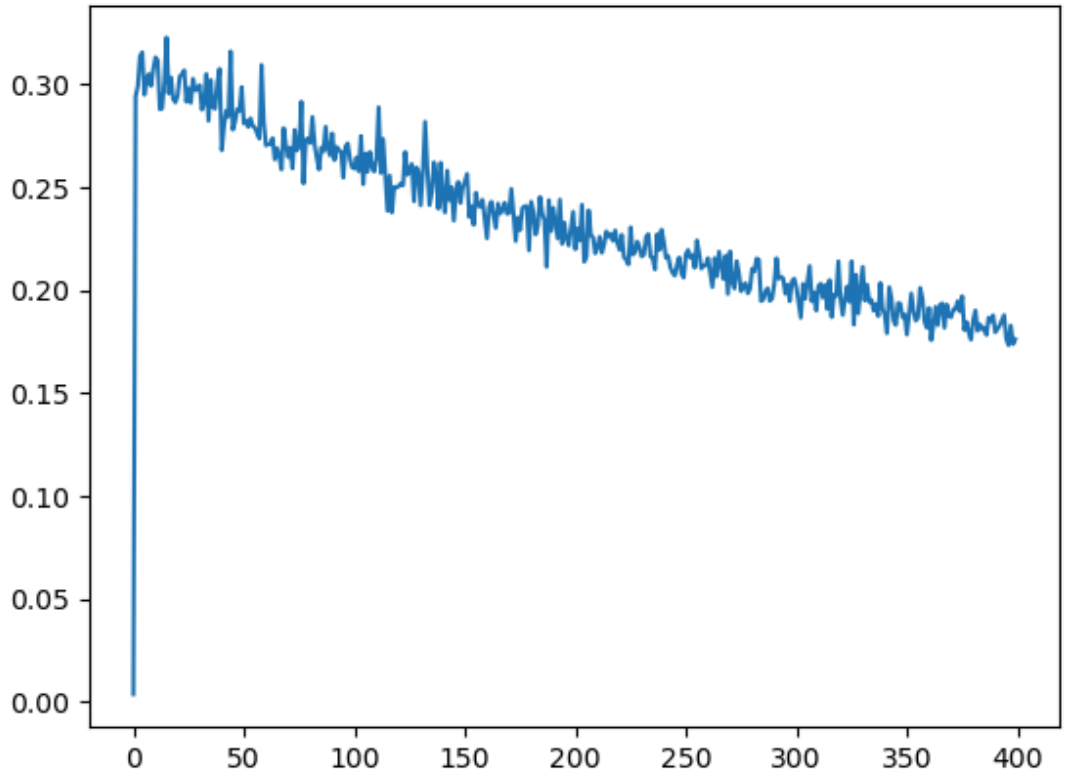


Fig1. Loss vs Iteration plot for CPG One - Neuron one_neuron_classifier.py

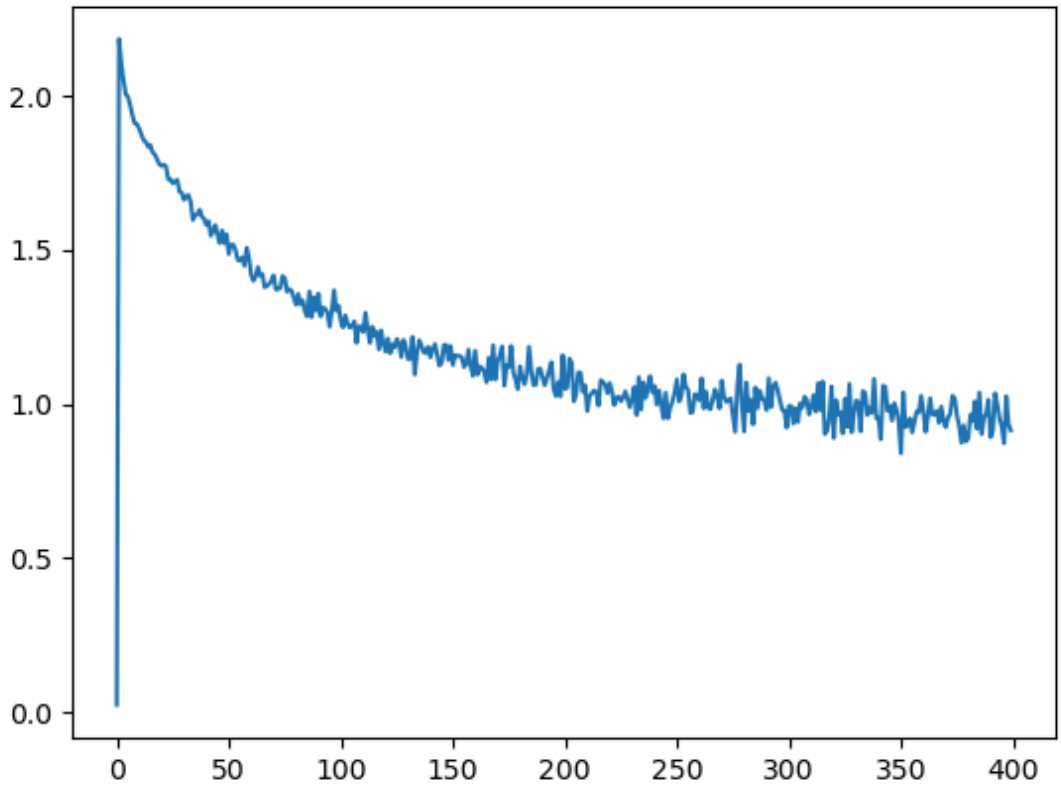
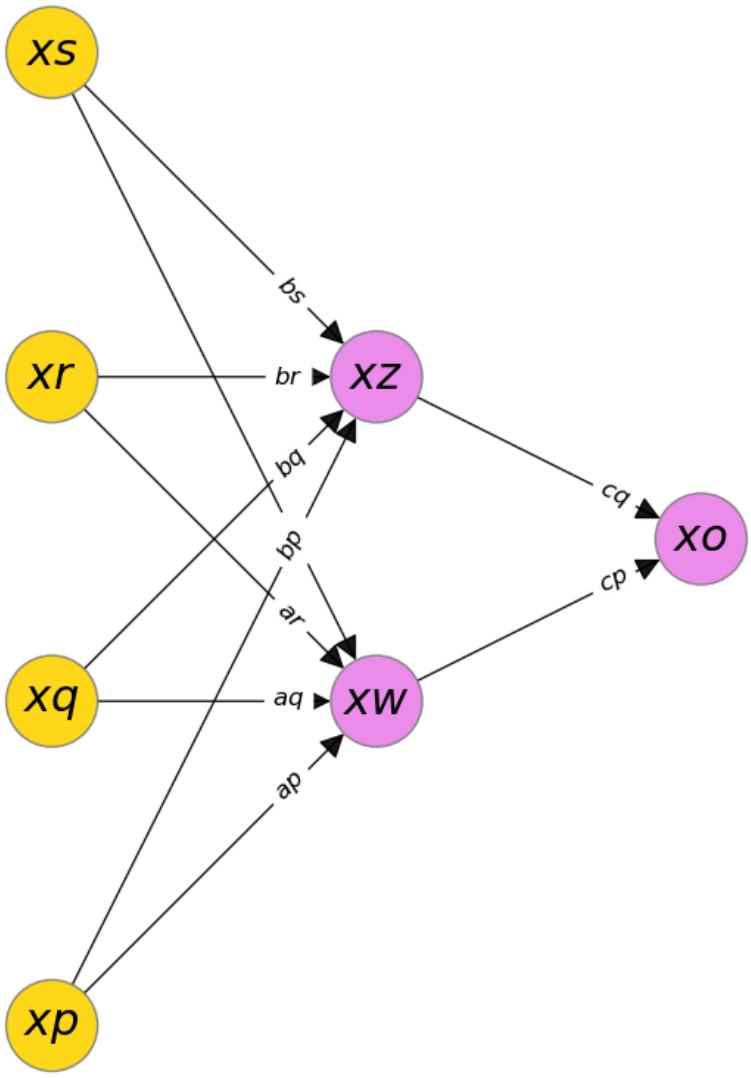


Fig2. Loss vs Iteration plot for torch.nn One - Neuron verify_with_torchnn.py

Fig 1 shows the CPG/ custom made one-neuron loss vs itteration graph and Fig 2. shows the torch.nn one-neuron loss vs itteration grpah. Both fig1 and fig2 have decresing loss, which shows that the error is decreasing over time/itterations and that the gradient optimizers are working, i.e. model are "learning". In fig1 the loss reduces linearly, but where as in fig2 it drops rapidly initially, then flattens and ossilates around the end behaving sort of like exponentially. Fig 2 converges slower than the Fig1 probably cause of a better gradient optimizer. Fig2 reaches to a loss of under ~1.0 which is less than half where as fig1 reached only ~1.7 > half by the end of training.

1.2 Comparing multi-neuron with CGP vs torch.nn



Last ten lines of output: **CPG multi-neuron** vs **torch.nn multi-neuron**

```
[iter=39001]  loss = 5.0162
[iter=39101]  loss = 5.2571
[iter=39201]  loss = 5.1446
[iter=39301]  loss = 5.0601
[iter=39401]  loss = 4.9013
[iter=39501]  loss = 5.1261
[iter=39601]  loss = 5.0779
[iter=39701]  loss = 5.2043
[iter=39801]  loss = 5.1673
[iter=39901]  loss = 5.1952
```

CPG multi-neuron last 10 losses

```
[iter=19001]  loss = 0.0869
[iter=19101]  loss = 0.0908
[iter=19201]  loss = 0.1015
[iter=19301]  loss = 0.0971
[iter=19401]  loss = 0.1010
[iter=19501]  loss = 0.1280
[iter=19601]  loss = 0.0970
[iter=19701]  loss = 0.0879
[iter=19801]  loss = 0.0989
[iter=19901]  loss = 0.0771
```

torch.nn multi-neuron last 10 losses

Display of the training loss versus the training iterations: **CPG multi-neuron** vs **torch.nn multi-neuron**

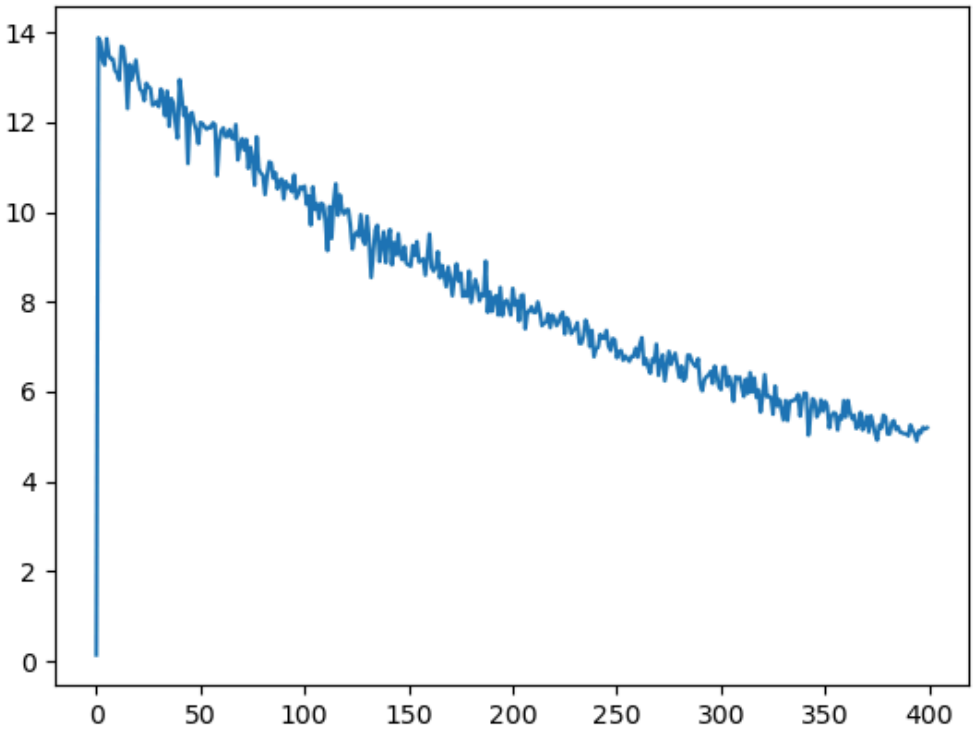


Fig3. Loss vs Iteration plot for CPG multi-neuron multi_neuron_classifier.py

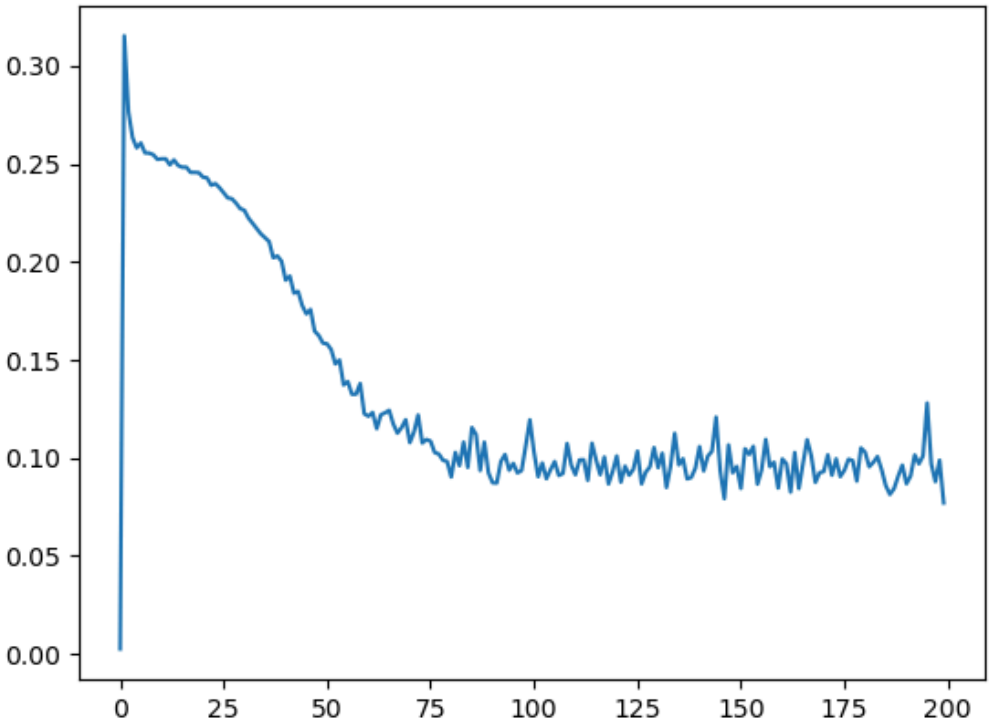
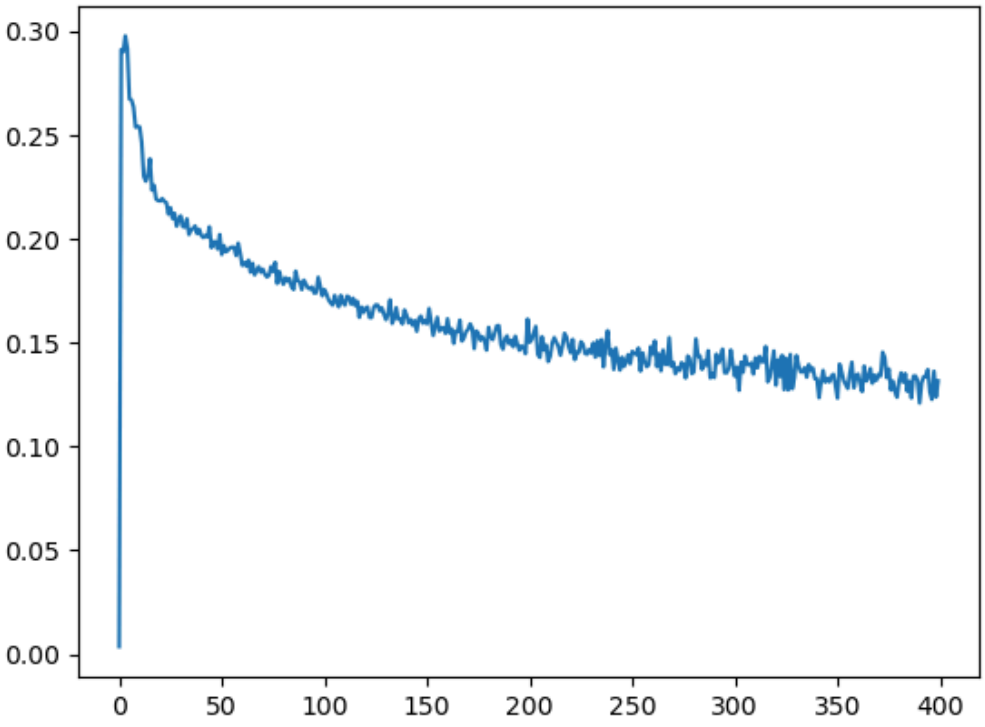


Fig4. Loss vs Iteration plot for torch multi-neuron multi_neuron_classifier.py

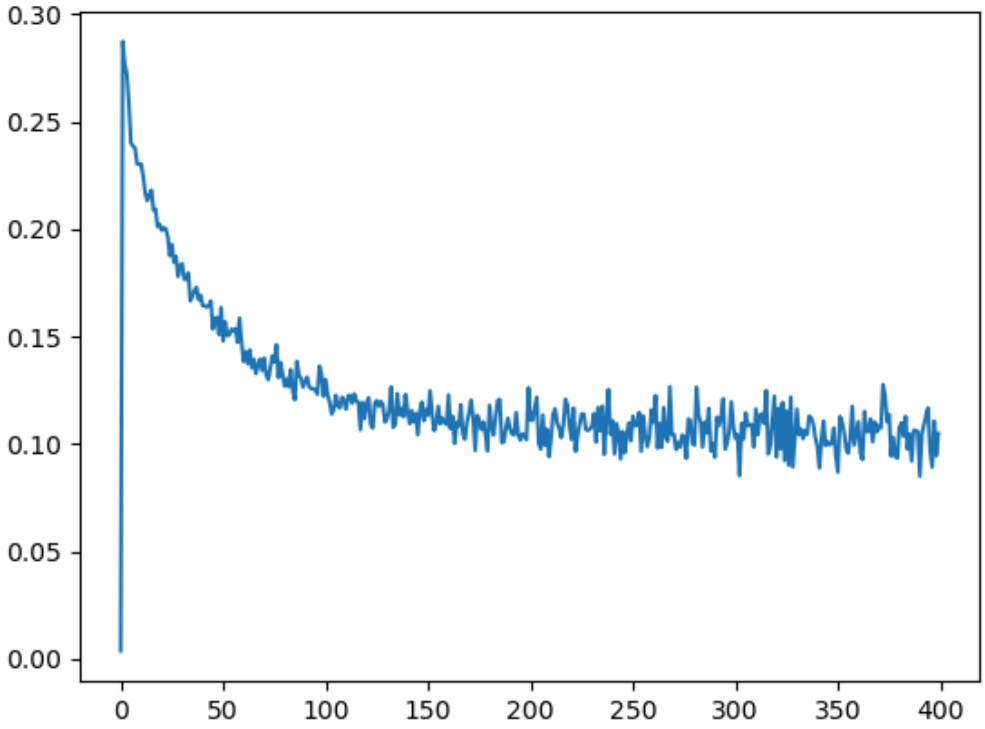
Fig 3 shows the plot for custom multi_neuron and fig 4 shows for the torch multi_neuron loss vs itteration plots. Just like fig 1 and fig 2 we observe similar trend in case of multi-neuron network also. Fig 2 converges faster than fig 1 and also reachs a lower loss value.

Below was my first trail for one_neuron SGD with momentum and Adam optimizers

SGD+ with momentum


0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
300
350
400

Adam


0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
300
350
400

2 ProgrammingTask

2.1 Implementing SGD+ and Adam

One-Neuron Classifiers Analysis and Visualization

Majority of the code has been borrowed from ComputationalGraphPrimer package by Prof. Kak: https://engineering.purdue.edu/kak/distCGP/ any edits have been made in a new file called UpdatedComputationalGraphPrimer.py file, where I have created a new class called **UpdatedComputationalGraphPrimer** which inherits the properties from the original **ComputationalGraphPrimer** class.

I have edited **run_training_loop_one_neuron_model** to redirect to the appropriate optimizer function based on the **Optimizer** ENUM.

New fuction added are **backprop_and_update_params_one_neuron_model_sgd_plus** which impliments the SGD+ Momentum and **backprop_and_update_params_one_neuron_model_adam** which impliments the Adam Optimizer both the functions are based on the original function **backprop_and_update_params_one_neuron_model**.

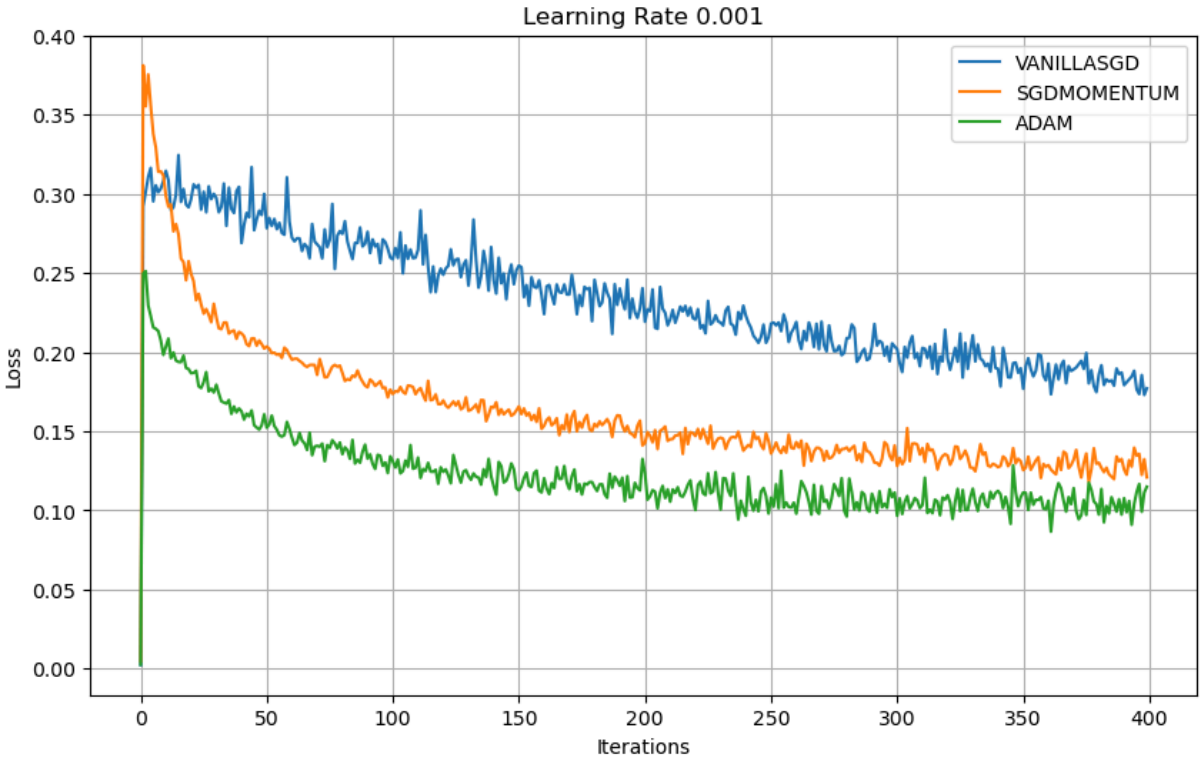
In [ ]:
```
import random
import numpy

seed = 0
random.seed(seed)
numpy.random.seed(seed)

from UpdatedComputationalGraphPrimer import *
```

In [ ]:
```
def compute_one_neuron(learning_rate):
      plt.figure(figsize=(10, 6))
      for optimizer in Optimizer:
            cgp = UpdatedComputationalGraphPrimer(
                        one_neuron_model = True,
                        expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                        output_vars = ['xw'],
                        dataset_size = 5000,
                        learning_rate = learning_rate,
                        training_iterations = 40000,
                        batch_size = 8,
                        display_loss_how_often = 100,
                        debug = False,
                        momentum = 0.9,
                        beta1 = 0.9  ,
                        beta2 = 0.999,
                        epsilon = 1e-8,  # Small constant to prevent division by ze
                        optimizer = optimizer,        #Enum to switch between optim
                  )
```

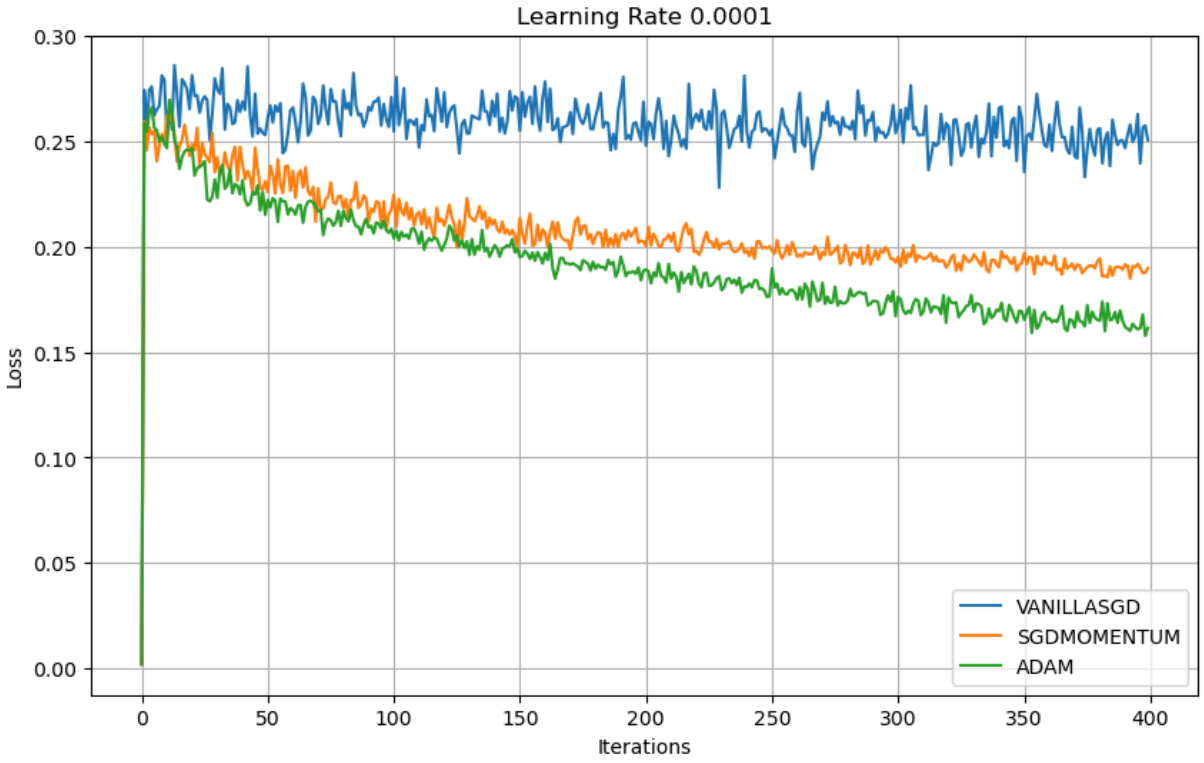
```
            cgp.parse_expressions()
            training_data = cgp.gen_training_data()
            loss_running_record = cgp.run_training_loop_one_neuron_model( training_
            plt.plot(loss_running_record, label=optimizer.name)

        # Plot Labels
        plt.xlabel("Iterations")
        plt.ylabel("Loss")
        plt.title(f"Learning Rate {learning_rate}")
        plt.legend()
        plt.grid()
        plt.show()
```

In [ ]:
```
if __name__ == '__main__':
    learning_rates = [1e-3,1e-4,3e-3]
    for x in learning_rates:
        compute_one_neuron(x)
```

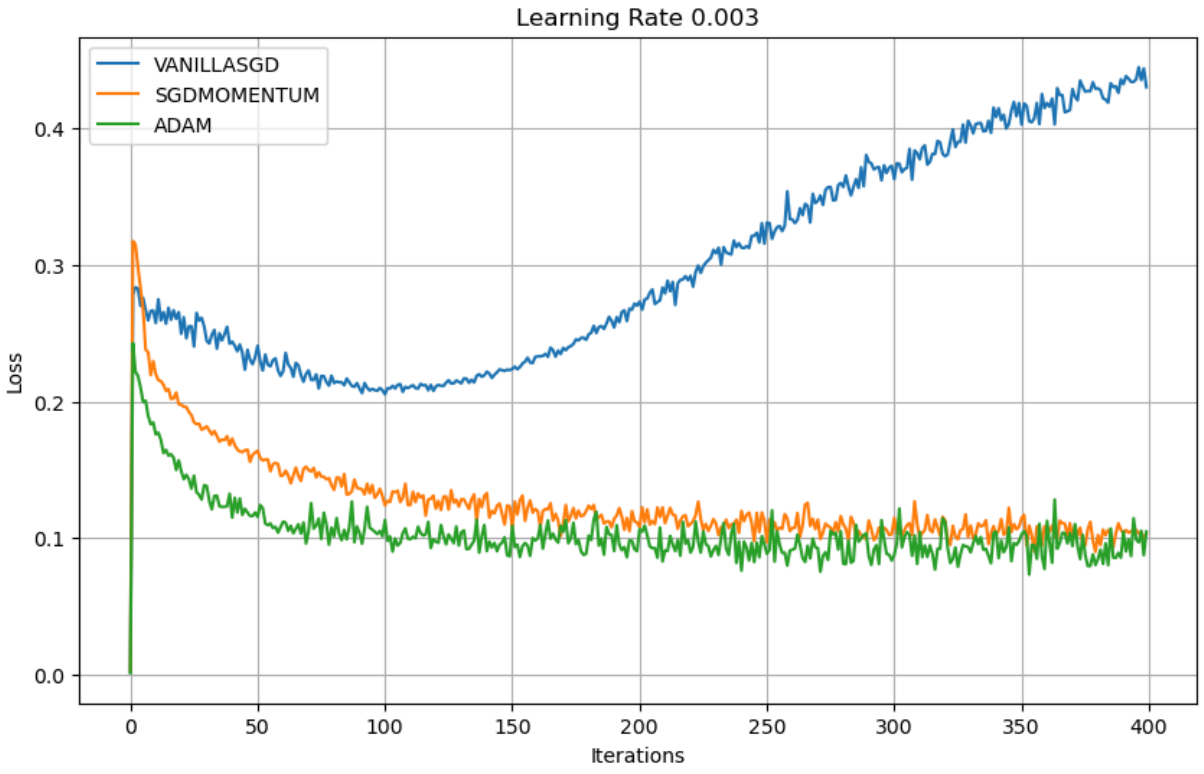
```
0
0
0
```




```
0
0
0
```




```
0
0
0
```



As you can see from the above figures, which have the plots for SGD, SGD+ and Adam Optimizers for the learning rates **1e-3(0.001), 1e-4(0.0001), 3e-3(0.003)** respectively. For all the 3 learning rates Adam converges fastst followed by SGD+ and SGD in the last.

In case one with a relatively "good" learning rate of 1e-3, once Global minimum is reached Adam osillates around it the most right after SGD+ folowed by the SGD being the smoothest of all.

In the second case where the **learning rate was 1e-4 which was too low**, since the SGD has only one parameter the learning rate so it could not move from the initial position and was just osilating there, since the SGD+ has the other parameter momentum and Adam has betas. They were able to move closer to the global minimum as they didnt just depend on the *learning rate*. Adam is the smoothest here.

In the thirs case where the **learning rate was 3e-3 which was too high**, SGD algorithm probably changed the sign once it got closer to the local minimum and then went the opposite direction, basically it overshot itself. Here we can see the advantage SGD+ has over the SGD. Again SGD+ and Adam performed as expected. In this case the **Adam classifier actually beats the torch.nn convergence rate** and the SGD+ is too close to call visually.

We can clearly see that as the **learning rate increased so did the convergence rate**, it converged the fastest when the learning rate was 3e-3 and the slowest when it was 1e-4. (Ignoring the Vanilla SGD). (This might not always be the case, this is just speculation based on the analysis I did with around 15 learning rate out of which 3 are presented here)

Multi-Neuron Analysis and Visualization

In [ ]:
```
from UpdatedComputationalGraphPrimer import *
```

In [ ]:
```
def compute_multi_neuron(learning_rate):
      plt.figure(figsize=(10, 6))
      for optimizer in Optimizer:
          cgp = UpdatedComputationalGraphPrimer(
                        num_layers = 3,
                        layers_config = [4,2,1],                          # num of n
                        expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                                       'xz=bp*xp+bq*xq+br*xr+bs*xs',
                                       'xo=cp*xw+cq*xz'],
                        output_vars = ['xo'],
                        dataset_size = 5000,
                        learning_rate = learning_rate,
                        training_iterations = 20000,
                        batch_size = 8,
                        display_loss_how_often = 100,
                        debug = False,
                        momentum = 0.9,
                        beta1 = 0.9  ,
                        beta2 = 0.999,
                        epsilon = 1e-8,  # Small constant to prevent division by ze
                        optimizer = optimizer,        #Enum to switch between optim
                  )
          cgp.parse_multi_layer_expressions()
          training_data = cgp.gen_training_data()
          loss_running_record= cgp.run_training_loop_multi_neuron_model( training_d
```

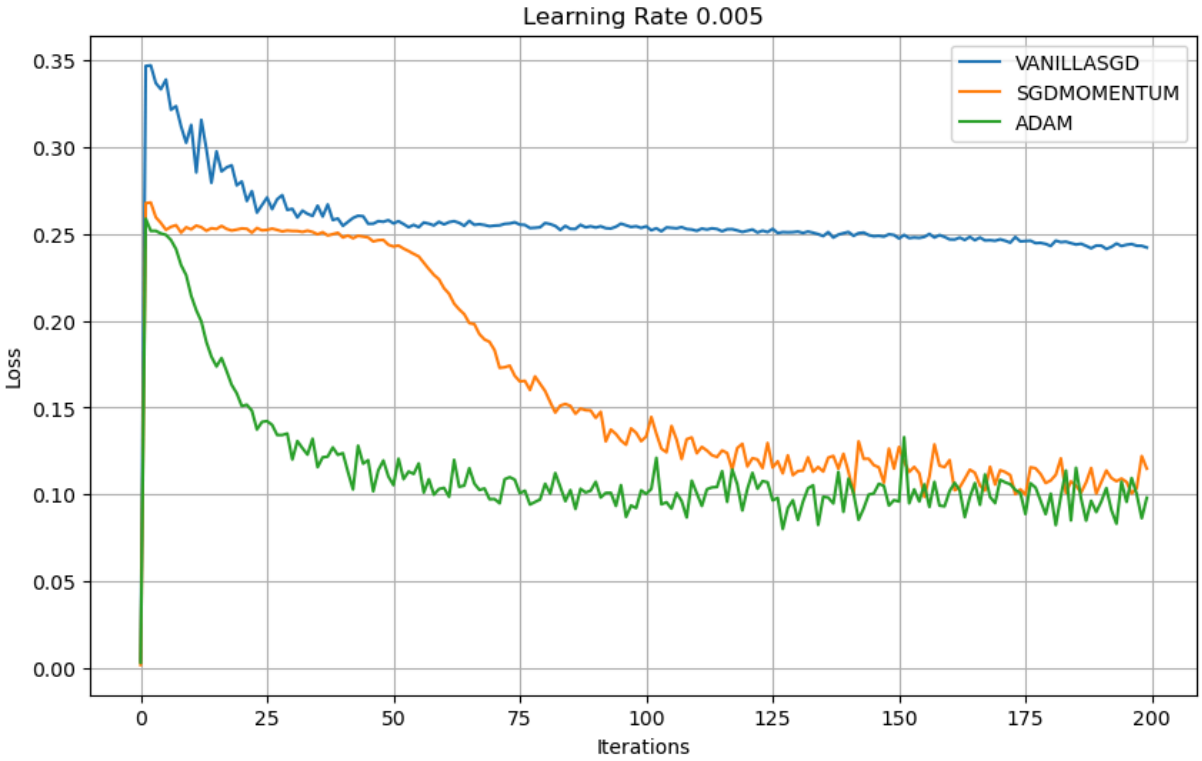
```
            plt.plot(loss_running_record, label=optimizer.name)

        # Plot Labels
        plt.xlabel("Iterations")
        plt.ylabel("Loss")
        plt.title(f"Learning Rate {learning_rate}")
        plt.legend()
        plt.grid()
        plt.show()
```

In [ ]:
```
learning_rates = [1e-2, 5e-3, 1e-3]
for x in learning_rates:
    compute_multi_neuron(x)
```

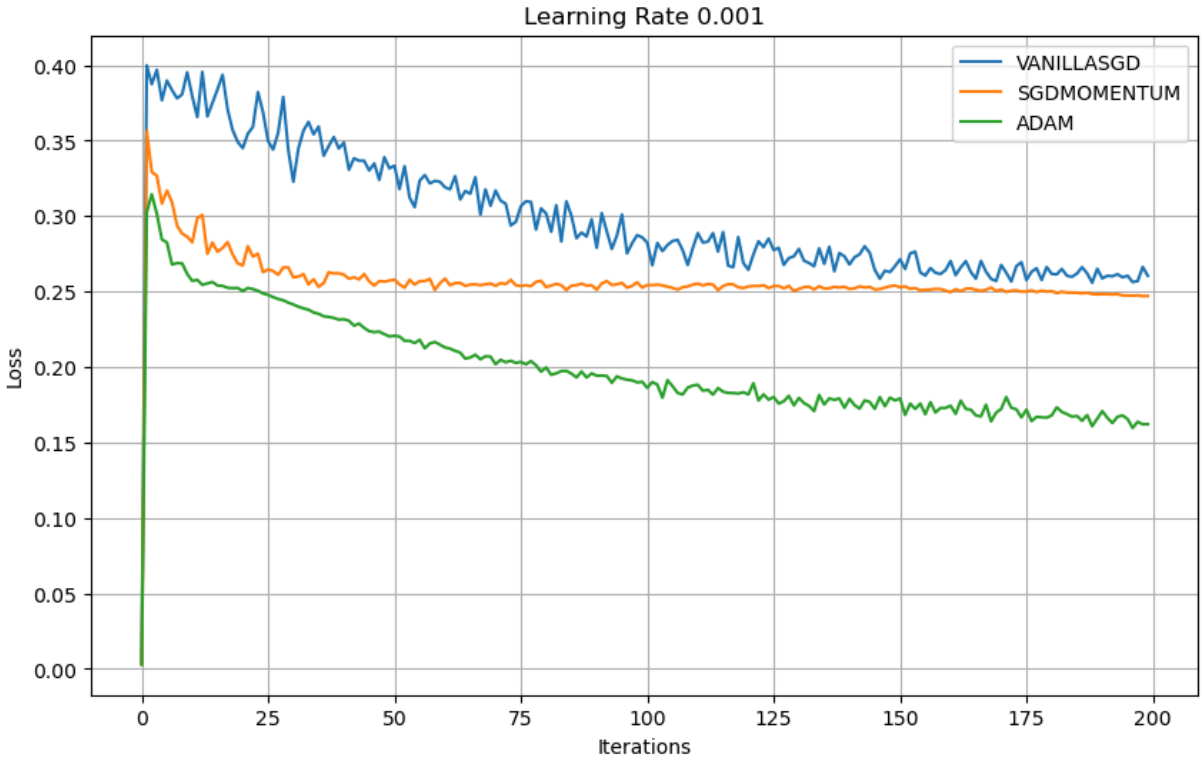
```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A6F0C10>, <ComputationalGraphPrimer.Exp object at 0x000002348A6F2CB0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A6F2D10>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A6F2CE0>, <ComputationalGraphPrimer.Exp object at 0x000002348A6F2CB0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A77F100>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A6F2CB0>, <ComputationalGraphPrimer.Exp object at 0x000002348A6F2D70>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A9AFA30>]}
```


Learning Rate 0.01
Loss
Iterations
0.00
0.05
0.10
0.15
0.20
0.25
0
25
50
75
100
125
150
175
200
VANILLASGD
SGDMOMENTUM
ADAM

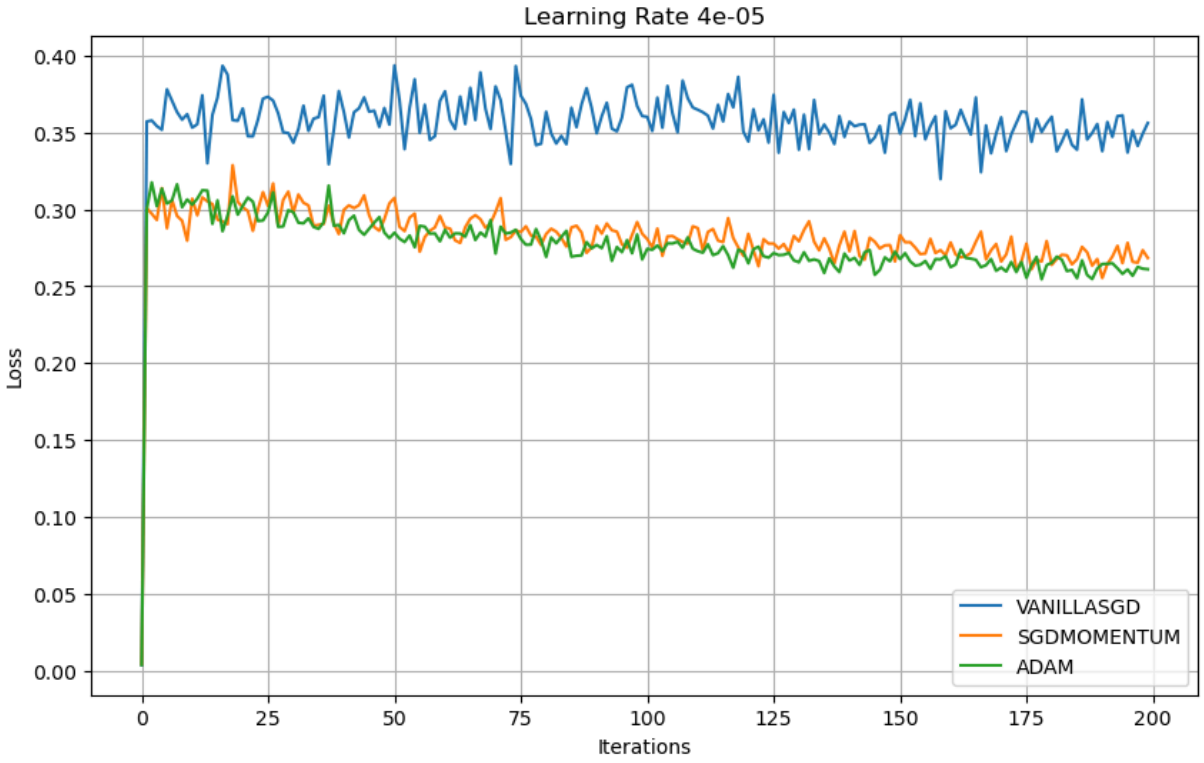
```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02349104FA90>, <ComputationalGraphPrimer.Exp object at 0x000002349104FAC0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002349104FB20>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02349104FAF0>, <ComputationalGraphPrimer.Exp object at 0x000002349104FAC0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234910BC1C0>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02349104FAC0>, <ComputationalGraphPrimer.Exp object at 0x000002349104FBB0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234910C4C40>]}
```


Learning Rate 0.005
VANILLASGD
SGDMOMENTUM
ADAM
Loss
Iterations
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
25
50
75
100
125
150
175
200

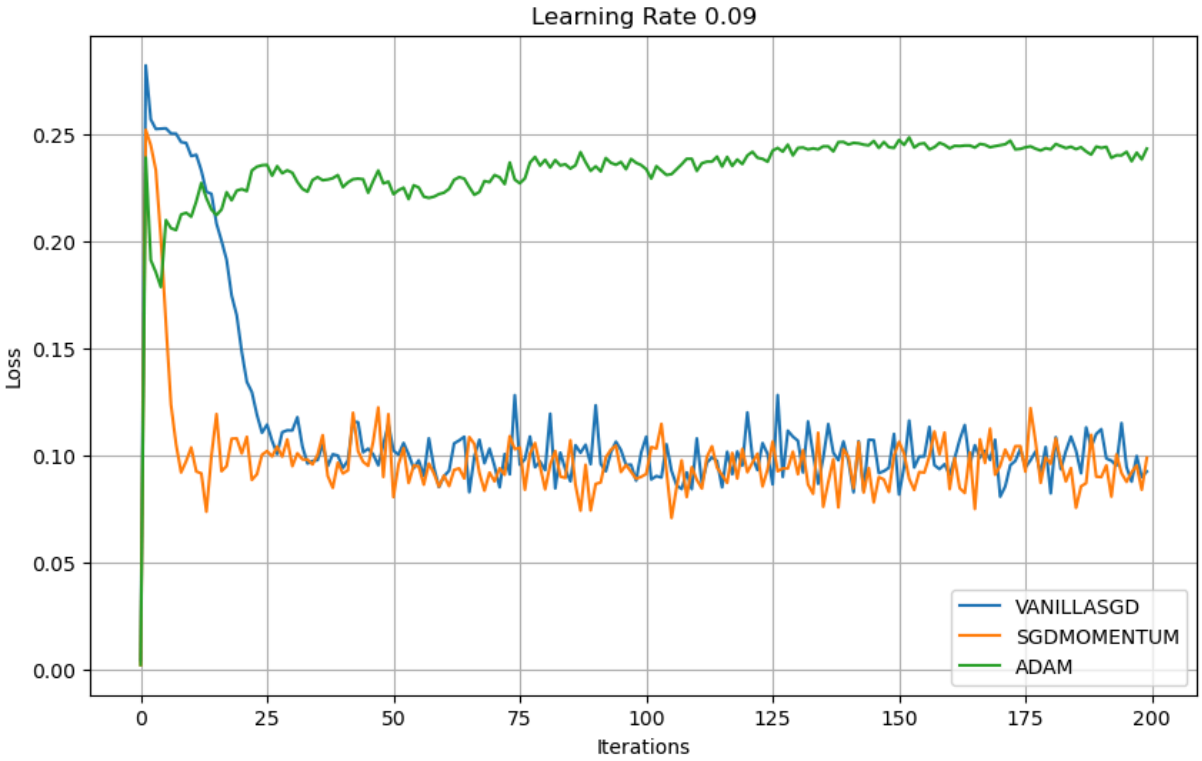
```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02349111FEB0>, <ComputationalGraphPrimer.Exp object at 0x000002349111FFD0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234910C8070>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
0234910C8100>, <ComputationalGraphPrimer.Exp object at 0x00000234910C8040>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A7E57B0>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
0234910C8040>, <ComputationalGraphPrimer.Exp object at 0x000002349111FFD0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x0000023491178700>]}
```


Learning Rate 0.001
VANILLASGD
SGDMOMENTUM
ADAM
Loss
Iterations
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
25
50
75
100
125
150
175
200

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
0234911F4610>, <ComputationalGraphPrimer.Exp object at 0x00000234911F44C0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234911F4670>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
0234911F4640>, <ComputationalGraphPrimer.Exp object at 0x00000234911F44C0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234911F4BE0>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
0234911F46A0>, <ComputationalGraphPrimer.Exp object at 0x00000234911F44C0>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x00000234910E5540>]}
```


Learning Rate 4e-05
Loss
Iterations
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
25
50
75
100
125
150
175
200
VANILLASGD
SGDMOMENTUM
ADAM

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A7FA4D0>, <ComputationalGraphPrimer.Exp object at 0x000002348A7FBA30>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A7FAD70>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A7FA350>, <ComputationalGraphPrimer.Exp object at 0x000002348A7FBA30>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002348A7FBE80>]}


self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs
*xs'], 2: ['xo=cp*xw+cq*xz']}


[Final] independent vars: {'xr', 'xp', 'xs', 'xq'}


[Final] self.layer_vars:  {0: ['xp', 'xq', 'xr', 'xs'], 1: ['xw', 'xz'], 2: ['xo']}


[Final] self.layer_params:  {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'b
s']], 2: [['cp', 'cq']]}


[Final] self.layer_exp_objects:  {1: [<ComputationalGraphPrimer.Exp object at 0x0000
02348A7FBA30>, <ComputationalGraphPrimer.Exp object at 0x000002348A7FB880>], 2: [<Co
mputationalGraphPrimer.Exp object at 0x000002349111E380>]}
```



In case of multi-neuron unlike one-neuron, Adam optimizer did not always perform the best.

As we observe from the above plots as the learning rate increased, the performance of the Adam optimizer reduced which is opposite of what we observed in the one-neuron, it isn't the expected behaviour. But our implementation of the optimizers and the multi-neural network is very basic. Another factor could be the Bais, incase of one-neuron we used only bais for the whole network/layer where as here in case of multi-neuron we used different Bais for every node in every layer. This could have made the optimizer more volatile compared to the single-neuron.

When the learning rate was low, Adam outperformed the SGD+ and SGD, converged faster as it used both RMS and momentum for faster convergence. It is followed by SGD+ then SGD, very similar to one-neuron.

Comparing Effects of Hyperparameters:

In [ ]:
```
import time
from prettytable import PrettyTable
```

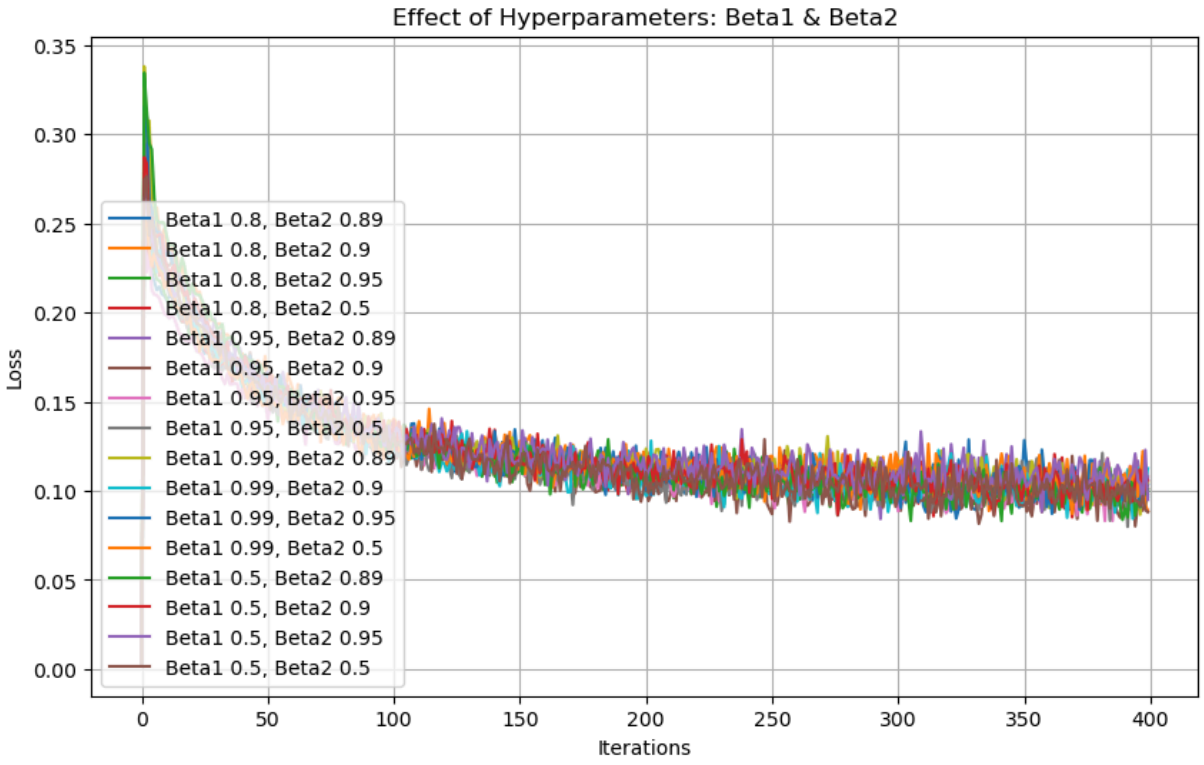
In [ ]:
```
def compute_one_neuron_betas(b1,b2):
      # results = []
      cgp = UpdatedComputationalGraphPrimer(
                  one_neuron_model = True,
                  expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                  output_vars = ['xw'],
                  dataset_size = 5000,
                  learning_rate = 1e-3,
                  training_iterations = 40000,
```

```
                batch_size = 8,
                display_loss_how_often = 100,
                debug = False,
                momentum = 0.9,
                beta1 = b1,
                beta2 = b2,
                epsilon = 1e-8,  # Small constant to prevent division by zero
                optimizer = Optimizer.ADAM,        #Enum to switch between optimi
            )

        cgp.parse_expressions()
        training_data = cgp.gen_training_data()

        start_time = time.time()  # Start timing
        loss_running_record = cgp.run_training_loop_one_neuron_model( training_data )
        end_time = time.time()  # End timing
        # results.append()
        # print("resultsssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
        plt.plot(loss_running_record, label=f"Beta1 {b1}, Beta2 {b2}")

        return ([b1, b2, np.round(end_time-start_time,2), np.round(loss_running_recor
```

In [ ]:

```
b1 = [0.8, 0.95, 0.99, 0.5]
b2 = [0.89, 0.9, 0.95, 0.5]
# b1 = [0.99, 0.5]
# b2 = [0.95, 0.5]
plt.figure(figsize=(10, 6))
results = []
for x in b1:
    for y in b2:
        results.append(compute_one_neuron_betas(x,y))
# print("rrrrrrrrrrrrrrrrrresults")
# print(results)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.title(f"Effect of Hyperparameters: Beta1 & Beta2")
plt.legend()
plt.grid()
plt.show()

# Display the table from the data
table = PrettyTable()
table.title = "Effect of Hyperparameters: Beta1 & Beta2"
table.field_names = ["Beta1", "Beta2", "Time Taken (s)", "Final Loss", "Min Loss"]
table.add_rows(results)
table.sortby = "Final Loss"
print(table)
```

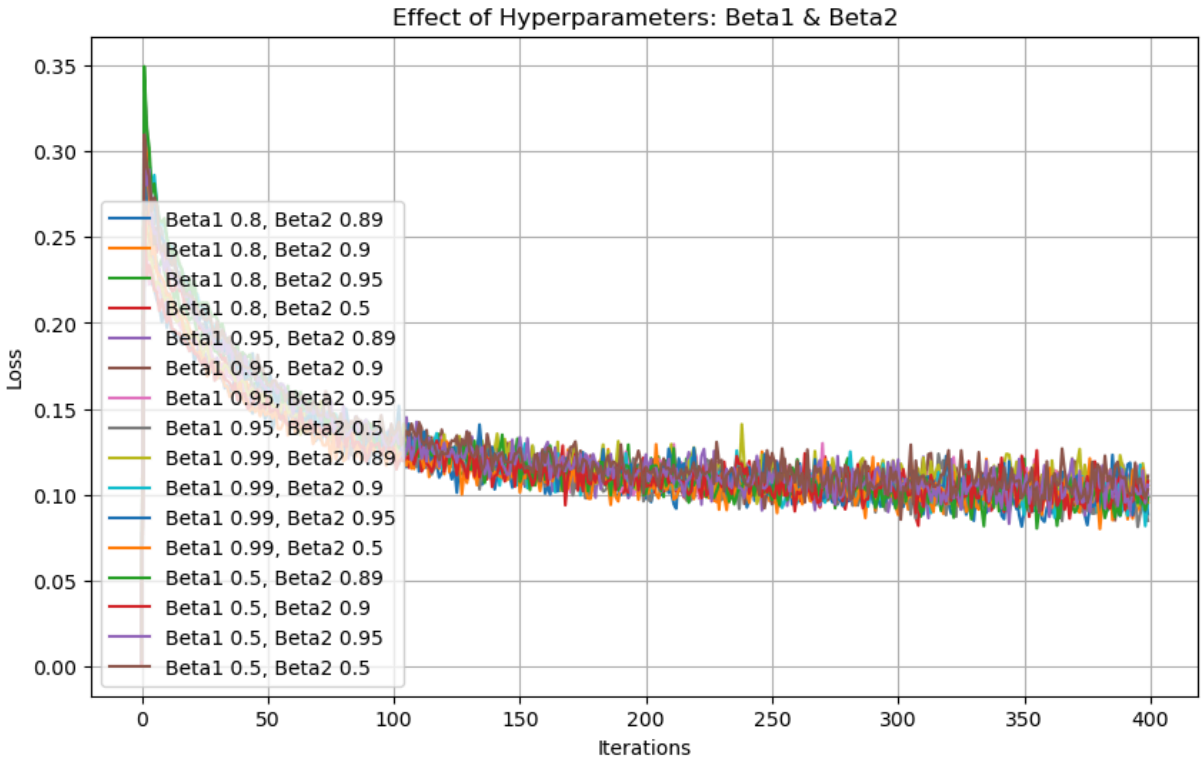
```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```



```
+-------------------------------------------------------+
|        Effect of Hyperparameters: Beta1 & Beta2       |
+-------+-------+----------------+------------+----------+
| Beta1 | Beta2 | Time Taken (s) | Final Loss | Min Loss |
+-------+-------+----------------+------------+----------+
|  0.5  |  0.5  |     11.24      |   0.0881   |  0.0021  |
|  0.99 |  0.5  |      11.4      |   0.0886   |  0.0021  |
|  0.95 |  0.95 |     11.15      |   0.0891   |  0.0023  |
|  0.5  |  0.95 |     11.49      |   0.095    |  0.0032  |
|  0.99 |  0.95 |     11.37      |   0.097    |  0.0032  |
|  0.95 |  0.9  |     11.31      |   0.0984   |  0.0048  |
|  0.8  |  0.95 |     11.18      |   0.0993   |  0.002   |
|  0.95 |  0.5  |     11.19      |   0.1001   |  0.0041  |
|  0.5  |  0.89 |      11.3      |   0.1008   |  0.0059  |
|  0.8  |  0.9  |     10.97      |   0.101    |  0.0035  |
|  0.8  |  0.89 |     11.57      |   0.1032   |  0.0027  |
|  0.99 |  0.89 |     11.21      |   0.1046   |  0.0039  |
|  0.5  |  0.9  |     11.35      |   0.1059   |  0.0035  |
|  0.95 |  0.89 |     11.03      |   0.1064   |  0.0032  |
|  0.8  |  0.5  |     11.13      |   0.1077   |  0.0022  |
|  0.99 |  0.9  |     11.61      |   0.1125   |  0.0017  |
+-------+-------+----------------+------------+----------+
```

In [ ]:
```python
b1 = [0.8, 0.95, 0.99, 0.5]
b2 = [0.89, 0.9, 0.95, 0.5]
# b1 = [0.99, 0.5]
# b2 = [0.95, 0.5]
plt.figure(figsize=(10, 6))
results = []
for x in b1:
    for y in b2:
        results.append(compute_one_neuron_betas(x,y))
# print("rrrrrrrrrrrrrrrrresults")
# print(results)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.title(f"Effect of Hyperparameters: Beta1 & Beta2")
plt.legend()
plt.grid()
plt.show()

# Display the table from the data
table = PrettyTable()
table.title = "Effect of Hyperparameters: Beta1 & Beta2"
table.field_names = ["Beta1", "Beta2", "Time Taken (s)", "Final Loss", "Min Loss"]
table.add_rows(results)
print(table)
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```



Effect of Hyperparameters: Beta1 & Beta2				
Beta1	Beta2	Time Taken (s)	Final Loss	Min Loss
0.8	0.89	11.58	0.0887	0.0019
0.8	0.9	11.27	0.097	0.0024
0.8	0.95	11.47	0.1077	0.0031
0.8	0.5	11.08	0.0983	0.0023
0.95	0.89	11.33	0.1007	0.0003
0.95	0.9	11.34	0.111	0.0024
0.95	0.95	11.22	0.105	0.002
0.95	0.5	11.24	0.0849	0.0078
0.99	0.89	11.3	0.1053	0.0039
0.99	0.9	11.49	0.0957	0.004
0.99	0.95	11.13	0.0983	0.0025
0.99	0.5	11.56	0.0955	0.0041
0.5	0.89	11.31	0.098	0.0042
0.5	0.9	11.74	0.1077	0.0018
0.5	0.95	12.12	0.1057	0.0045
0.5	0.5	11.63	0.1029	0.004

Incase of Adam, beta 1 is used for the momentum. It helps to keep moving more and more if the direction of the gradient is same,it helps to converge faster and get out of loccal minimum. Beta 2 is used for squared gradient depending on the previous.

The running times, the convergence are very close to each other as we can see from the plot and the table. The final lose is very very slightly different, which again can just be margin or error. Its too close to say.

I have sorted it according to the final loss, to maybe get any insights as we observe we got least Final loss when both B1 and b2 were the least 0.5 and 0.5 which is not expected. and the seconf highest when b2 was 0.5 and b1 was 0.95, maybe b2 being less has more effect than b1? as the third least is when b1 n b2 are 0.95.

When sorted according to B1, I dont really observe any trends that are on your face visible.

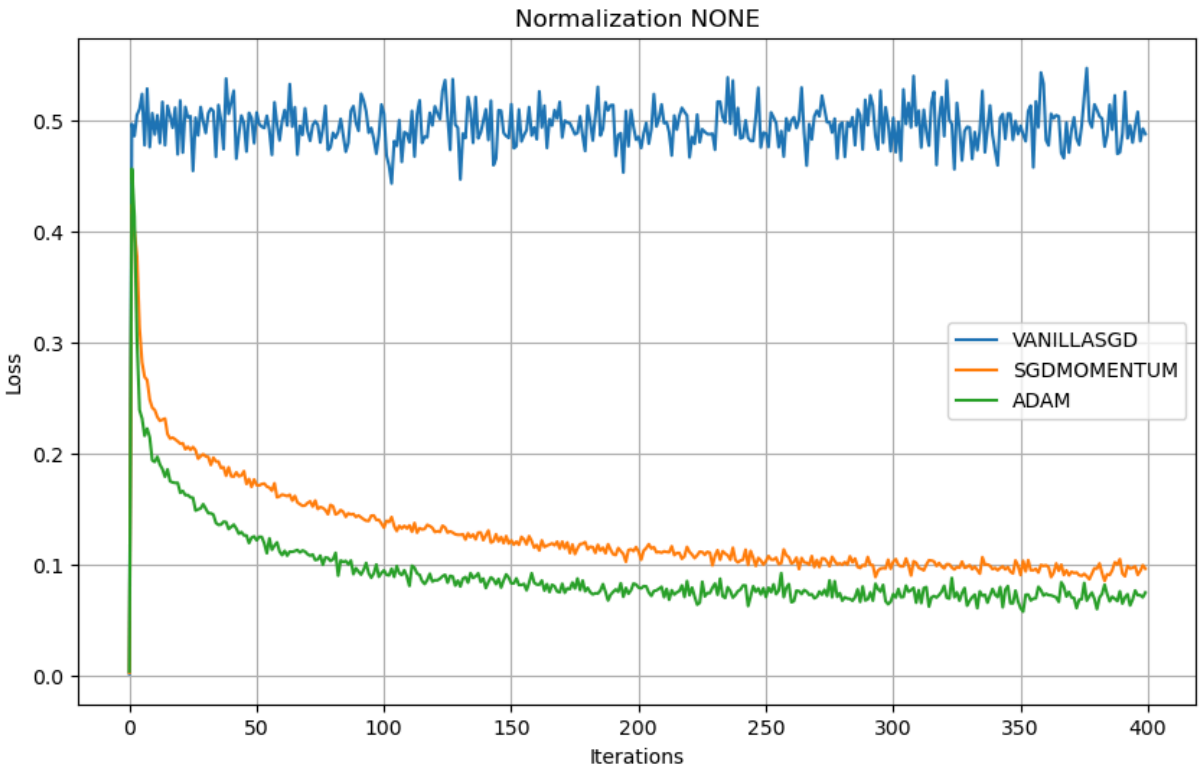
3. Extra credit

In [ ]:
```
def compute_one_neuron( normalization: Normalization):
      plt.figure(figsize=(10, 6))
      for optimizer in Optimizer:
            cgp = UpdatedComputationalGraphPrimer(
                        one_neuron_model = True,
                        expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                        output_vars = ['xw'],
                        dataset_size = 5000,
                        learning_rate = 1e-3,
                        training_iterations = 40000,
                        batch_size = 8,
                        display_loss_how_often = 100,
```

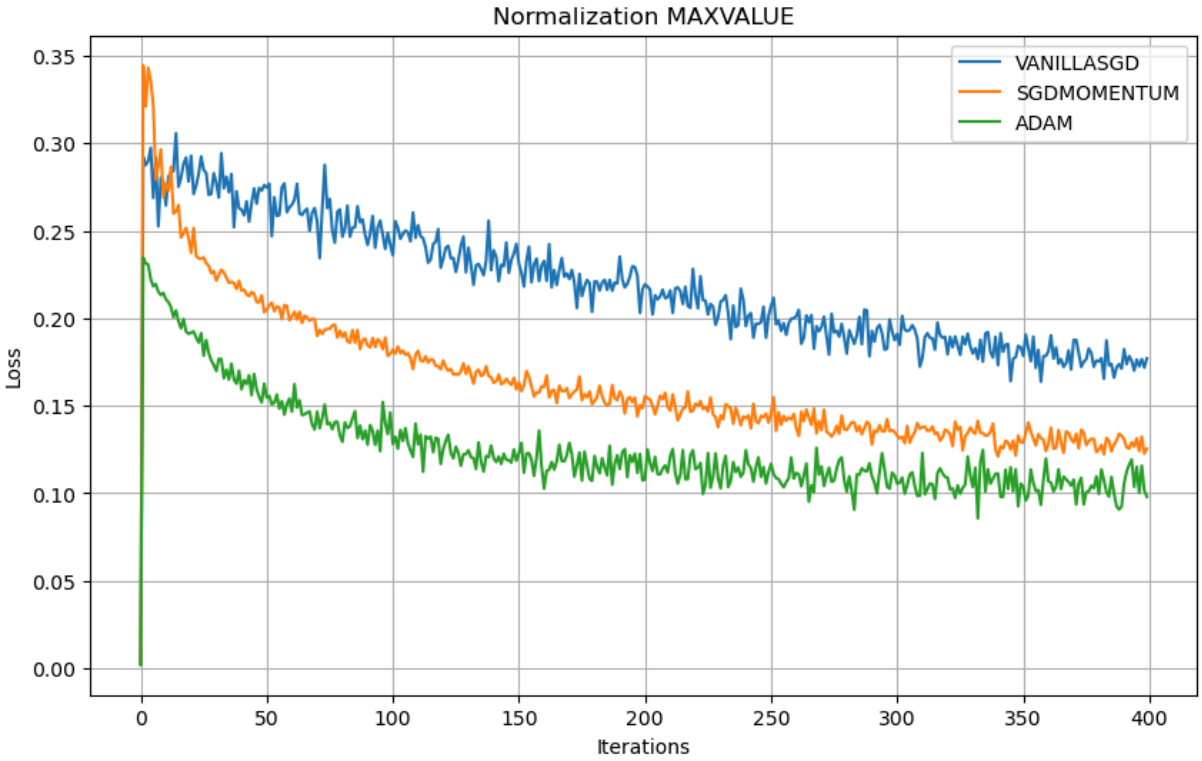
```
                        debug = False,
                        momentum = 0.9,
                        beta1 = 0.9  ,
                        beta2 = 0.999,
                        epsilon = 1e-8,  # Small constant to prevent division by ze
                        optimizer = optimizer,        #Enum to switch between optim
                    )

            cgp.parse_expressions()
            training_data = cgp.gen_training_data()
            loss_running_record = cgp.run_training_loop_one_neuron_model( training_
            plt.plot(loss_running_record, label=optimizer.name)

      # Plot Labels
      plt.xlabel("Iterations")
      plt.ylabel("Loss")
      plt.title(f"Normalization {normalization.name}")
      plt.legend()
      plt.grid()
      plt.show()
```

In [ ]:
```
if __name__ == '__main__':
#     learning_rates = [1e-3,1e-4,3e-3]
#     learning_rates = [1e-3]
      for x in Normalization:
            compute_one_neuron(x)
```

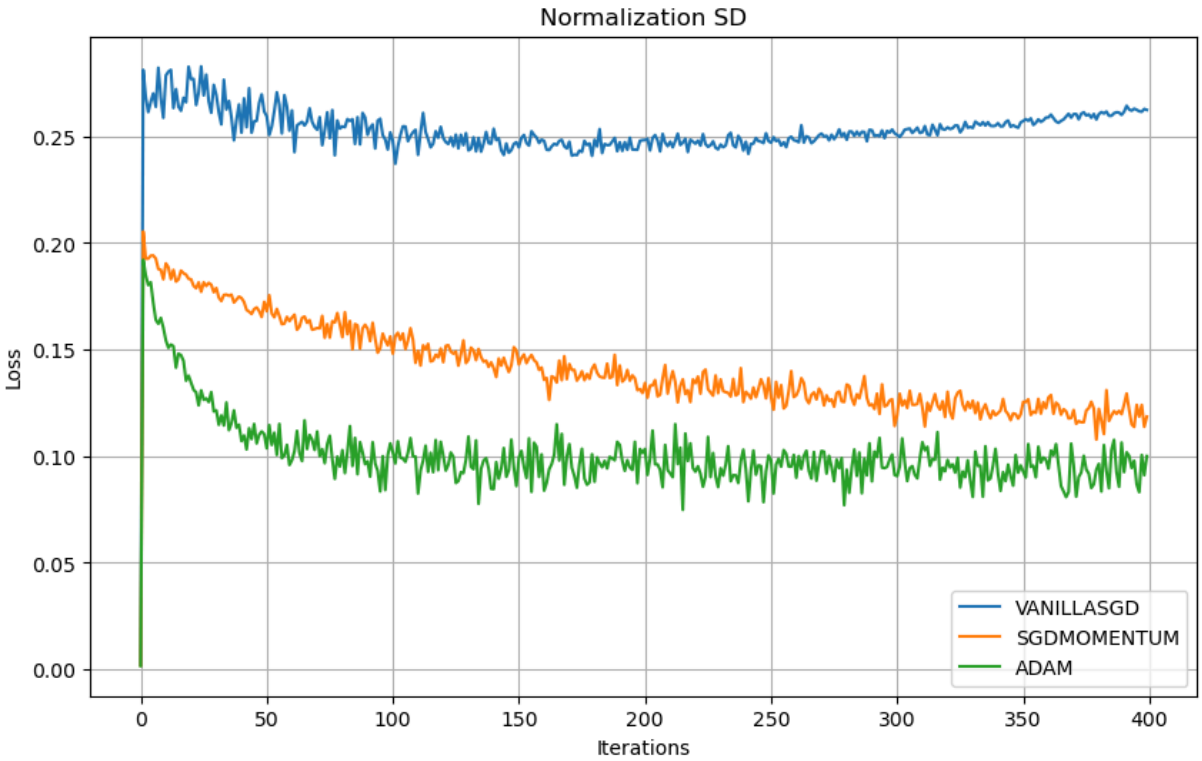
```
0
0
0
```




```
0
0
0
```



0
0
0



No Normalization vs Normalization vs Truncated Standard Deviation Remap Normalization

Case 1 No Normalization: Since the training data was already Gaussian input it was already around the mean with the infinite interval range $(-\infty,\infty)$. Since the data is already clean but

with extreme values, it should work pretty good. From the plot we can see that It performed very well in case of SGD+ and Adam as the data does not have any irregularities, it converged to the min loss faster than max normalization and was also very smooth.

Case 2 Max Value Normalization: Here we are just diving the batch by its max value, each batch will have different batch values, also the extreme values are very extreme might also be infinite. So, the it will cause alot of irregularities cause of varying batch max values and the scaling done by just diving by max value. We can observe that the plot becomes more volatile cause the irregularies and also its the slowest convergence.

Case 2 Trncated SD Normalization: Here we are truncating to the interval $(\mu-5\sigma,\mu+5\sigma)$ remove all the extreme outliers which once you cross 5 SD are irrelavant anyways. Then we are Rescaling it to (-1,1). Since this also changes the distribution slightly it became more volatile. But it is smoother than the Max Value and also converges the fastest among the three, The scaling is very rudimetry but effective.

Source Code for the new class

Majority again borrowed from Prof. Kak's work.

In [ ]:
```
from ComputationalGraphPrimer import *
from enum import Enum

class Optimizer(Enum):
    VANILLASGD = 1
    SGDMOMENTUM = 2
    ADAM = 3

class Normalization(Enum):
    NONE = 1
    MAXVALUE = 2
    SD = 3

class UpdatedComputationalGraphPrimer(ComputationalGraphPrimer):

    def __init__(self, *args, **kwargs):
        if 'momentum' in kwargs    :   momentum = kwargs.pop('momentum')
        if 'beta1' in kwargs:
            beta1 = kwargs.pop('beta1')
        if 'beta2' in kwargs:
            beta2 = kwargs.pop('beta2')
        if 'epsilon' in kwargs:
            epsilon = kwargs.pop('epsilon')
        if 'optimizer' in kwargs    :   optimizer = kwargs.pop('optimizer')

        if momentum:
            self.momentum = momentum
        else:
            self.momentum = 0.9
        if beta1:
            self.beta1 = beta1
```

```
        else:
            self.beta1 = 0.9

        if beta2:
            self.beta2 = beta2
        else:
            self.beta2 = 0.999

        if epsilon:
            self.epsilon = epsilon
        else:
            self.epsilon = 1e-8
        if optimizer:
            self.optimizer = optimizer
        else:
            self.optimizer = Optimizer.VANILLASGD
        super().__init__(*args, **kwargs)

        # Initialize first and second moment estimates for Adam Optimizer
        self.t = 1  # Time step counter


    ##########################################################################
    ######################################### one neuron model ###############
    def run_training_loop_one_neuron_model(self, training_data,normalization = Norm
        """
        The training loop must first initialize the learnable parameters.  Remember
        symbolic names in your input expressions for the neural layer that do not b
        letter 'x'.  In this case, we are initializing with random numbers from a u
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in s
        self.velocity_bias = 0.0
        self.velocity = {param: 0.0 for param in self.vals_for_learnable_params}

        self.bias = random.uniform(0,1)                    ## Adding the bias improv
                                                           ##   We initialize it to
        self.m_t = {param: 0.0 for param in self.vals_for_learnable_params}
        self.v_t = {param: 0.0 for param in self.vals_for_learnable_params}
        self.m_t_bias = 0.0  # For bias
        self.v_t_bias = 0.0  # For bias

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first u
            the training dataset is created.  Search for the following function in

                                gen_training_data(self)

            As you will see in the implementation code for this method, the trainin
            consists of a Python dict with two keys, 0 and 1, the former points to
            all Class 0 samples and the latter to a list of all Class 1 samples.  I
            the data samples are drawn from a multi-dimensional Gaussian distributi
            classes have different means and variances.  The dimensionality of each
            is set by the number of nodes in the input layer of the neural network.
```

```
            The data loader's job is to construct a batch of samples drawn randomly
            lists mentioned above.  And it mush also associate the class label with
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                                 ## When
                                                                                ##   sam
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                                 ## First
                maxval = 0.0                                                    ## For a
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                if(normalization == Normalization.MAXVALUE):
                    batch_data = [item/maxval for item in batch_data]           ## N
                #  Truncates data to (μ - 5σ, μ + 5σ) and rescales to (-1,1).
                elif normalization == Normalization.SD:
                    mean = np.mean(batch_data)
                    sigma = np.std(batch_data)
                    # Define truncation range
                    lower_bound = mean - (5 * sigma)
                    upper_bound = mean + (5 * sigma)
                    # Truncating the values to remove outliers
                    batch_data = np.clip(batch_data, lower_bound, upper_bound)
                    # Rescale to (-1,1)
                    batch_data = 2 * (batch_data - lower_bound) / (upper_bound - lo
                batch = [batch_data, batch_labels]
                return batch

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        data = data_loader.getbatch()
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                          ##  AvSer
                                                                                ##    eve
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
```

```
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tupl
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2 for i in ra
            avg_loss_over_iterations += loss / float(len(class_labels_in_batch))
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                # print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations)
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels_in_batch, y_pre
            # Adam Optimizer
            if self.optimizer == Optimizer.ADAM:
                self.backprop_and_update_params_one_neuron_model_adam(data_tuples_i
            # SGD+ momentum
            elif self.optimizer == Optimizer.SGDMOMENTUM:
                self.backprop_and_update_params_one_neuron_model_sgd_plus(data_tupl
            # Vanilla SGD
            else:
                self.backprop_and_update_params_one_neuron_model(data_tuples_in_bat
        return loss_running_record


    def backprop_and_update_params_one_neuron_model_sgd_plus(self, data_tuples_in_b
        """
        This function implements the equations shown on Slide 61 of my Week 3 prese
        class at Purdue.  All four parameters defined above are lists of what was e
        forward prop function or calculated by it for each training data sample in
        """
        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.
        vals_for_learnable_params = self.vals_for_learnable_params
        # Compute gradients and apply momentum updates for weights
        for param in self.vals_for_learnable_params:
            partial_of_loss_wrt_param = 0.0

            for j in range(self.batch_size):
                vals_for_input_vars_dict = dict(zip(input_vars, list(data_tuples_in
                partial_of_loss_wrt_param += - y_errors_in_batch[j] * vals_for_inpu

            partial_of_loss_wrt_param /= float(self.batch_size)

            # Apply momentum update rule

            # formula from youtube video: https://www.youtube.com/watch?v=k8fTYJPd3
            # self.velocity[param] = self.momentum * self.velocity[param] + (1 - se

            # Formula form the Class Lecture pdf:  pk+1 = pk - 2·α·JF(pk)·ϵk
            # self.velocity[param] = self.momentum * self.velocity[param] +  2 * se

            # formula from the HW pdf: vt+1 = βvt +gt
            self.velocity[param] = self.momentum * self.velocity[param] +  partial_

            # Update weight using velocity: wt+1 = wt −ηvt+1
            self.vals_for_learnable_params[param] -= self.learning_rate * self.velo

        # Compute gradients and apply momentum updates for bias
```

```
        y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
        deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)

        gradient_bias = y_error_avg * deriv_sigmoid_avg

        # Apply momentum update rule for bias
        # self.velocity_bias = self.momentum * self.velocity_bias + (1 - self.momen
        self.velocity_bias = self.momentum * self.velocity_bias +  gradient_bias

        # Update bias using velocity
        self.bias += self.learning_rate * self.velocity_bias
        # self.bias += self.velocity_bias

    def backprop_and_update_params_one_neuron_model_adam(self, data_tuples_in_batch
        """
        Implements backpropagation using the Adam Optimizer.
        """
        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
        param_to_vars_map = {v: k for k, v in input_vars_to_param_map.items()}  # C

        # Compute gradients and apply Adam updates for weights
        for param in self.vals_for_learnable_params:
            partial_of_loss_wrt_param = 0.0

            for j in range(self.batch_size):
                vals_for_input_vars_dict = dict(zip(input_vars, list(data_tuples_in
                partial_of_loss_wrt_param += - y_errors_in_batch[j] * vals_for_inpu

            partial_of_loss_wrt_param /= float(self.batch_size)

            # Update biased first moment estimate
            # mt+1 = β1 * mt +(1-β1)*gradt,
            self.m_t[param] = self.beta1 * self.m_t[param] + (1 - self.beta1) * par

            # Update biased second raw moment estimate:
            # vt+1 = β2 * vt +(1-β2)*(gradt)2
            self.v_t[param] = self.beta2 * self.v_t[param] + (1 - self.beta2) * (pa

            # Compute bias-corrected moment estimates
            # ^mt+1=  mt / 1 - β1^t
            m_t_hat = self.m_t[param] / (1 - self.beta1 ** self.t)
            # ^vt+1 = vt / 1 - β2^t
            v_t_hat = self.v_t[param] / (1 - self.beta2 ** self.t)

            # Apply Adam update rule
            #  pt+1 = pt - (lr * (^mt+1 / (^vt+1 +ϵ) ^ -1/2))
            self.vals_for_learnable_params[param] -= self.learning_rate * (m_t_hat

        # Compute bias gradient
        y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
        deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)
        gradient_bias = y_error_avg * deriv_sigmoid_avg

        # Using the same formulas as above but for bias
        # Update biased first moment estimate for bias
```

```
        # mt_b = β1 * mt_b +(1-β1)*gradt
        self.m_t_bias = self.beta1 * self.m_t_bias + (1 - self.beta1) * gradient_bi

        # Update biased second moment estimate for bias
        # vt_b = β2 * vt_b +(1-β2)*(gradt)2
        self.v_t_bias = self.beta2 * self.v_t_bias + (1 - self.beta2) * (gradient_b

        # Compute bias-corrected estimates for bias
        # ^mt_b=  mt_b / 1 - β1^t
        m_t_bias_hat = self.m_t_bias / (1 - self.beta1 ** self.t)
        # ^vt_B = vt_b / 1 - β2^t
        v_t_bias_hat = self.v_t_bias / (1 - self.beta2 ** self.t)

        # Apply Adam update rule for bias
        #  bt+1 = bt + (lr * (^mt_b / (^vt_b + ϵ) ^ -1/2))
        self.bias += self.learning_rate * (m_t_bias_hat / (v_t_bias_hat + self.epsi
        # Increment time step
        self.t += 1

    ##########################################################################################



    ##########################################################################################
    ######################################### multi neuron model ###################

    def run_training_loop_multi_neuron_model(self, training_data):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first u
            the training dataset is created.  Search for the following function in

                                gen_training_data(self)

            As you will see in the implementation code for this method, the trainin
            consists of a Python dict with two keys, 0 and 1, the former points to
            all Class 0 samples and the latter to a list of all Class 1 samples.  I
            the data samples are drawn from a multi-dimensional Gaussian distributi
            classes have different means and variances.  The dimensionality of each
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly
            lists mentioned above.  And it mush also associate the class label with
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
```

```
            cointoss = random.choice([0,1])                                 ## When
                                                                            ##   sam
            if cointoss == 0:
                return random.choice(self.class_0_samples)
            else:
                return random.choice(self.class_1_samples)

        def getbatch(self):
            batch_data,batch_labels = [],[]                                 ## First
            maxval = 0.0                                                    ## For a
            for _ in range(self.batch_size):
                item = self._getitem()
                if np.max(item[0]) > maxval:
                    maxval = np.max(item[0])
                batch_data.append(item[0])
                batch_labels.append(item[1])
            batch_data = [item/maxval for item in batch_data]               ## Norma
            batch = [batch_data, batch_labels]
            return batch

    ##  The training loop must first initialize the learnable parameters.  Reme
    ##  symbolic names in your input expressions for the neural layer that do n
    ##  letter 'x'.  In this case, we are initializing with random numbers from
    ##  over the interval (0,1):
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in s
    ##  In the same  manner, we must also initialize the biases at each node th
    ##  propagating data:
    self.bias =   {i : [random.uniform(0,1) for j in range( self.layers_config[
    data_loader = DataLoader(training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0                                          ##
                                                                            ##
    # momentum parameters
    self.velocity = {param: 0.0 for param in self.all_params}
    self.velocity_bias = {i: [0.0 for _ in range(self.layers_config[i])]
                     for i in range(1, self.num_layers)}
    ## For estimating the changes to the bias to be made on the basis of the de
    self.m_t = {param: 0.0 for param in self.vals_for_learnable_params}
    self.v_t = {param: 0.0 for param in self.vals_for_learnable_params}
    self.m_t_bias = {i: [0.0 for _ in range(self.layers_config[i])]
                     for i in range(1, self.num_layers)}
    self.v_t_bias = {i: [0.0 for _ in range(self.layers_config[i])]
                     for i in range(1, self.num_layers)}
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = data[0]
        class_labels = data[1]
        self.forward_prop_multi_neuron_model(data_tuples)
        predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_lay
        y_preds =  [item for sublist in  predicted_labels_for_batch  for item i
        loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(c
        loss_avg = loss / float(len(class_labels))
        avg_loss_over_iterations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
```

```
                loss_running_record.append(avg_loss_over_iterations)
                # print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations)
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels, y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds, y_errors_in
            # Adam Optimizer
            if self.optimizer == Optimizer.ADAM:
                self.backprop_and_update_params_multi_neuron_model_adam(y_preds, y_
            # SGD+ momentum
            elif self.optimizer == Optimizer.SGDMOMENTUM:
                self.backprop_and_update_params_multi_neuron_model_sgd_plus(y_preds
            # Vanilla SGD
            else:
                self.backprop_and_update_params_multi_neuron_model(y_preds, y_error
        return loss_running_record


    def backprop_and_update_params_multi_neuron_model_sgd_plus(self, predictions, y
        """
        First note that loop index variable 'back_layer_index' starts with the inde
        the last layer.  For the 3-layer example shown for 'forward', back_layer_in
        starts with a value of 2, its next value is 1, and that's it.

        In the code below, the outermost loop is over the data samples in a batch.
        on Slide 73 of my Week 3 lecture, in order to calculate the partials of Los
        respect to the learnable params, we need to backprop the prediction errors
        the gradients of the Sigmoid.  For the purpose of satisfying the requiremen
        SGD, the backprop of the prediction errors and the gradients needs to be ca
        out separately for each training data sample in a batch.  That's what the o
        loop is for.

        After we exit the outermost loop, we average over the results obtained from
        training data sample in a batch.

        Pay attention to the variable 'vars_in_layer'.  These store the node variab
        the current layer during backpropagation.
        """
        ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to store backprop
        pred_err_backproped_at_layers =   [ {i : [None for j in range( self.layers_
                                                                for i in range(se
        ## This will store "\delta L / \delta w" you see at the LHS of the equation
        partial_of_loss_wrt_params = {param : 0.0 for param in self.all_params}
        ## For estimating the changes to the bias to be made on the basis of the de
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i] ) ]  for i
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [ y_errors[b] ]
            for back_layer_index in reversed(range(1,self.num_layers)):
                input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
                deriv_sigmoids =  self.gradient_vals_for_layers[back_layer_index]
                vars_in_layer  =  self.layer_vars[back_layer_index]
                vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]
                vals_for_input_vars_dict =  dict(zip(vars_in_next_layer_back, self.
                ## For the next statement, note that layer_params are stored in a d
                ##       {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']],
                ## "layer_params[idx]" is a list of lists for the link weights in l
                layer_params = self.layer_params[back_layer_index]
```

```
                transposed_layer_params = list(zip(*layer_params))
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][back_layer_index - 1][k] =


                for j,var in enumerate(vars_in_layer):
                    layer_params = self.layer_params[back_layer_index][j]
                    input_vars_to_param_map = self.var_to_var_param[var]
                    param_to_vars_map = {param : var for var, param in input_vars_t

                    ##  Update the partials of Loss wrt to the learnable parameters
                    ##  and the previous layer. You are accumulating these partials
                    ##  data samples in the batch being processed.  For each traini
                    ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 sl
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param]   +=   pred_err_backprope
                                                                    vals_for_in
                ##  We will now estimate the change in the bias that needs to be ma
                ##  from the derivatives the sigmoid at the nodes in the current la
                ##  backproped to the previous layer nodes:
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] += pred_err_backpro

        ## Now update the learnable parameters.  The loop shown below carries out S
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] /  float(
            self.velocity[param] = self.momentum * self.velocity[param] + partial_o
            # self.velocity[param] = self.momentum * self.velocity[param] + (1 - se
            self.vals_for_learnable_params[param] += self.learning_rate * self.velo
            # self.vals_for_learnable_params[param] -=  self.velocity[param]

            # step = self.learning_rate * partial_of_loss_wrt_param
            # self.vals_for_learnable_params[param] += step

        ##  Finally we update the biases at all the nodes that aggregate data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                # self.bias[layer_index][k]  +=  self.learning_rate * ( bias_change
                bias_gradient = bias_changes[layer_index][k] / float(self.batch_siz
                self.velocity_bias[layer_index][k] = self.momentum * self.velocity_
                self.bias[layer_index][k] += self.learning_rate * self.velocity_bia
                # self.velocity_bias[layer_index][k] = self.momentum * self.velocit
                # self.bias[layer_index][k] -= self.velocity_bias[layer_index][k]

    def backprop_and_update_params_multi_neuron_model_adam(self, predictions, y_err
        """
        First note that loop index variable 'back_layer_index' starts with the inde
        the last layer.  For the 3-layer example shown for 'forward', back_layer_in
        starts with a value of 2, its next value is 1, and that's it.

        In the code below, the outermost loop is over the data samples in a batch.
        on Slide 73 of my Week 3 lecture, in order to calculate the partials of Los
        respect to the learnable params, we need to backprop the prediction errors
```

```
        the gradients of the Sigmoid.  For the purpose of satisfying the requiremen
        SGD, the backprop of the prediction errors and the gradients needs to be ca
        out separately for each training data sample in a batch.  That's what the o
        loop is for.

        After we exit the outermost loop, we average over the results obtained from
        training data sample in a batch.

        Pay attention to the variable 'vars_in_layer'.  These store the node variab
        the current layer during backpropagation.
        """
        ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to store backprop
        pred_err_backproped_at_layers =   [ {i : [None for j in range( self.layers_
                                                                for i in range(se
        ## This will store "\delta L / \delta w" you see at the LHS of the equation
        partial_of_loss_wrt_params = {param : 0.0 for param in self.all_params}
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i] ) ]  for i

        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [ y_errors[b] ]
            for back_layer_index in reversed(range(1,self.num_layers)):
                input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
                deriv_sigmoids =  self.gradient_vals_for_layers[back_layer_index]
                vars_in_layer  =  self.layer_vars[back_layer_index]
                vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]
                vals_for_input_vars_dict =  dict(zip(vars_in_next_layer_back, self.
                ## For the next statement, note that layer_params are stored in a d
                ##       {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']],
                ## "layer_params[idx]" is a list of lists for the link weights in l
                layer_params = self.layer_params[back_layer_index]
                transposed_layer_params = list(zip(*layer_params))
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][back_layer_index - 1][k] =


                for j,var in enumerate(vars_in_layer):
                    layer_params = self.layer_params[back_layer_index][j]
                    input_vars_to_param_map = self.var_to_var_param[var]
                    param_to_vars_map = {param : var for var, param in input_vars_t

                    ##  Update the partials of Loss wrt to the learnable parameters
                    ##  and the previous layer. You are accumulating these partials
                    ##  data samples in the batch being processed.  For each traini
                    ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 sl
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param]   +=   pred_err_backprope
                                                                 vals_for_in
                ##  We will now estimate the change in the bias that needs to be ma
                ##  from the derivatives the sigmoid at the nodes in the current la
                ##  backproped to the previous layer nodes:
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] += pred_err_backpro
```

```
        ## Now update the learnable parameters.  The loop shown below carries out S
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] /  float(
            # Update biased first moment estimate
            # mt+1 = β1 * mt +(1-β1)*gradt,
            self.m_t[param] = self.beta1 * self.m_t[param] + (1 - self.beta1) * par

            # Update biased second raw moment estimate:
            # vt+1 = β2 * vt +(1-β2)*(gradt)2
            self.v_t[param] = self.beta2 * self.v_t[param] + (1 - self.beta2) * (pa

            # Compute bias-corrected moment estimates
            # ^mt+1=  mt / 1 - β1^t
            m_t_hat = self.m_t[param] / (1 - self.beta1 ** self.t)
            # ^vt+1 = vt / 1 - β2^t
            v_t_hat = self.v_t[param] / (1 - self.beta2 ** self.t)

            # Apply Adam update rule
            #  pt+1 = pt + (lr * (^mt+1 / (^vt+1 +ϵ) ^ -1/2))
            self.vals_for_learnable_params[param] -= self.learning_rate * (m_t_hat
            # step = self.learning_rate * partial_of_loss_wrt_param
            # self.vals_for_learnable_params[param] += step

        ##  Finally we update the biases at all the nodes that aggregate data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):

                # self.bias[layer_index][k]  +=  self.learning_rate * ( bias_change
                # self.bias[layer_index][k] -= self.velocity_bias[layer_index][k]
                gradient_bias = bias_changes[layer_index][k] / float(self.batch_siz
                # Using the same formulas as above but for bias
                # Update biased first moment estimate for bias
                # mt_b = β1 * mt_b +(1-β1)*gradt
                self.m_t_bias[layer_index][k] = self.beta1 * self.m_t_bias[layer_in

                # Update biased second moment estimate for bias
                # vt_b = β2 * vt_b +(1-β2)*(gradt)2
                self.v_t_bias[layer_index][k] = self.beta2 * self.v_t_bias[layer_in

                # Compute bias-corrected estimates for bias
                # ^mt_b=  mt_b / 1 - β1^t
                m_t_bias_hat = self.m_t_bias[layer_index][k] / (1 - self.beta1 ** s
                # ^vt_B = vt_b / 1 - β2^t
                v_t_bias_hat = self.v_t_bias[layer_index][k] / (1 - self.beta2 ** s
                # Apply Adam update rule for bias
                #  bt+1 = bt + (lr * (^mt_b / (^vt_b + ϵ) ^ -1/2))
                self.bias[layer_index][k] += self.learning_rate * (m_t_bias_hat / (
        self.t += 1

    ##########################################################################
```