Autonomus Lab 3: HPC and Deep Learning

Exercise 1:

- Try GradientDescentOptimizer with different learning rates
- Check out other descent methods (look for options online)
- Represent these extra experiments plus the Gradient Descent with 0.01 learning rate

First, adapt script2launch.sh to ask for needed resources. In this case, as the task isn't resource exhaustive at all, with 40 CPUs, one GPU and 2 minutes of run would be enough. This increases our priority in the queue.

Different dependencies are used due to incompatibility problems which have been fixed using the following libraries.

```
#!/bin/bash
#SBATCH --job-name="Exercise_1_GD"
#SBATCH --qos=debug
#SBATCH -D .
#SBATCH --output=output/Exercise_1_GD_%j.out
#SBATCH --error=output/Exercise_1_GD_%j.err
#SBATCH --cpus-per-task=40
#SBATCH --gres gpu:1
#SBATCH --time=00:02:00

module purge; module load gcc/8.3.0 ffmpeg/4.2.1 cuda/10.2 cudnn/7.6.4 nccl/
                        2.4.8 tensorrt/6.0.1 openmpi/4.0.1 atlas/3.10.3 scalapack/
                        2.0.2 fftw/3.3.8 szip/2.1.1 opencv/4.1.1 python/3.7.4_ML

python $PWD/ex1_GradientDescent/gradient_descent.py
python $PWD/ex1_GradientDescent/momentum.py
python $PWD/ex1_GradientDescent/adam.py
python $PWD/ex1_GradientDescent/multioptimizer.py
```

Gradient Descent

Different learning rates have been tested for the proposed linear model using Gradient Descent. Two different situations are observed: learning rate below 0.01 and above 0.01.

In the first case, there is convergence to the optimal value and, the higher the learning rate is, the faster is the convergence. However, if this learning rate is too big, then it skips the global optimum and the loss starts to increase to infinity instead of decreasing. The fastest decrease of the loss is obtained for learning rate 0.01.

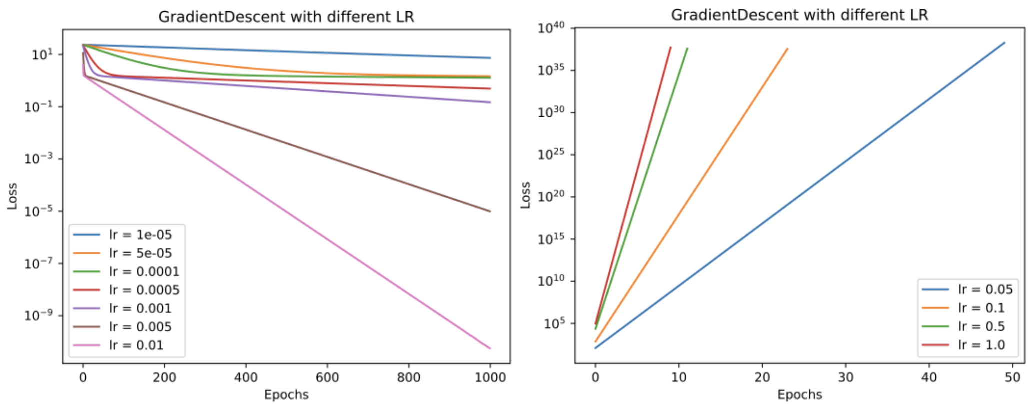


Figure 1: Gradient Descent for different learning rate values.

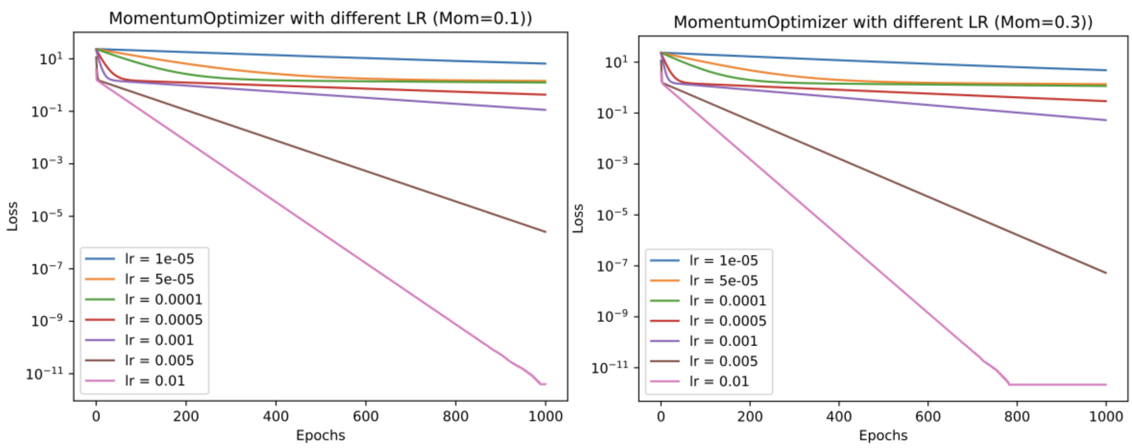
Momentum Optimizer

The Momentum optimizer, also known as SGD with momentum, is a variant of the Stochastic Gradient Descent (SGD) algorithm that incorporates momentum to accelerate the learning process. SGD is a commonly used optimization algorithm for machine learning models, but it can be slow and prone to getting stuck in local minima. Momentum helps to address these limitations by adding a momentum term to the weight updates.

The momentum term acts as a velocity, accumulating information about the direction of the gradient over time. This helps to smooth out the oscillations and prevent the algorithm from getting stuck in local minima. In essence, momentum allows the algorithm to take larger steps in the direction of the overall gradient, accelerating convergence.

Lower momentum, in this case, makes the convergence slower. This makes sense given that this problem is very simple as loss function is convex and convergence is fast and always in the same direction so information about previous steps is consistent with current steps as decreasing loss direction is always the same.

Consequently, the fastest decrease is observed for momentum 0.9 and learning rate of 0.01 (pink curve in Figure 2).



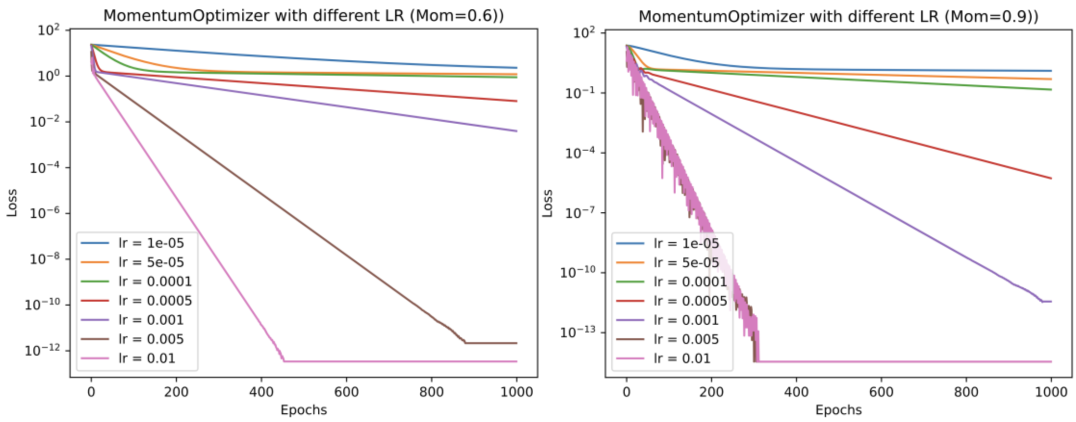


Figure 2: Convergence using Momentum Optimizer with different learning rates and momentum coefficient

Adam Optimizer

Adam employs adaptive learning rates, meaning that the learning rate is adjusted for each parameter based on its individual history of gradients. This helps to address the issue of fixed learning rates, which may be too large for some parameters and too small for others. By adaptively adjusting the learning rate, Adam can optimize the learning process for each parameter more effectively. It also includes momentum, which have been fixed to 0.9 for the reasons mentioned for the previous optimizer.

In this case, it converges even for values above 0.01. Indeed, convergence seems much faster for initial learning rate of 0.5. This can be due to posterior automatic learning rate adaptation near the optimum. This could explain the erratic behaviour observed above 0.05.

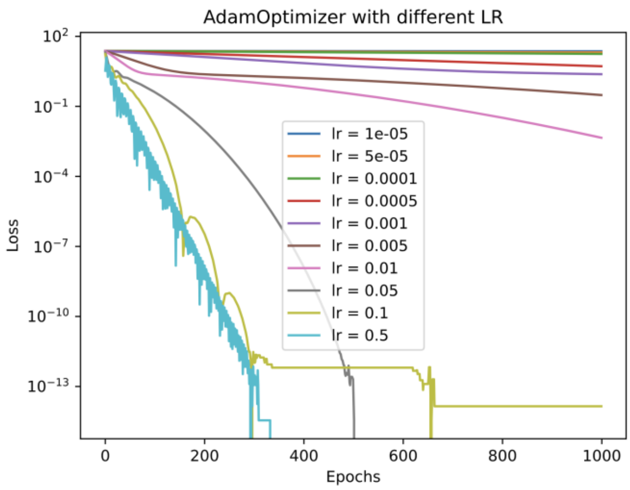


Figure 3: Convergence using Adam Optimizer with different learning rates

Comparison of three optimization algorithms for Learning Rate of 0.01

Learning rate for Momentum and Gradient Descent has been set as 0.01 based on previous observations and 0.05 in the case of Adam. Comparing the three of them, it's possible to see that the best one in terms of error minimization is Adam. Momentum optimizer reaches the minimum almost at the same time as Adam does but getting stucked in a slightly larger value. Gradient Descent Optimizer converges much slower than the others, not reaching the plateau phase in 1000 epochs. This illustrates the advantages of using more complex approaches to gradient descent optimization.

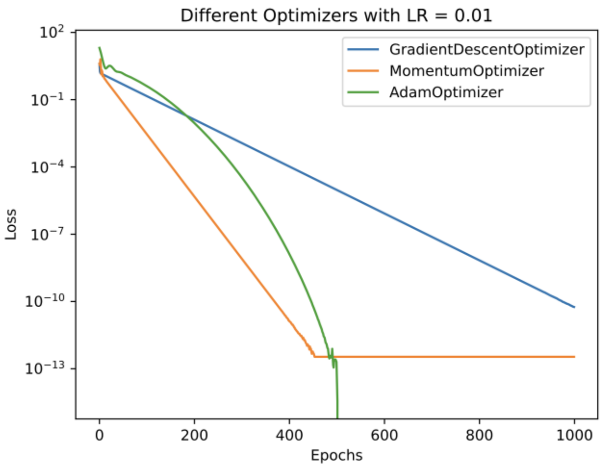


Figure 4: Comparison of all the algorithms for learning rate of 0.01

Exercise 2:

MNIST dataset

- Plot convergence rates in a similar way as the previous exercise
- Consider different optimizers and learning rates

Gradient Descent

This problem is more complex than the one observed before given that, in this case, reaching global optimum is not guaranteed anymore. The goal of choosing the best optimizer and tune their hyperparameters is finding the best local minima.

In this case, there is a significative difference in convergence rates for each of the learning rates. In this case, and for the number of epochs chosen, the best performing learning rate is 0.5.

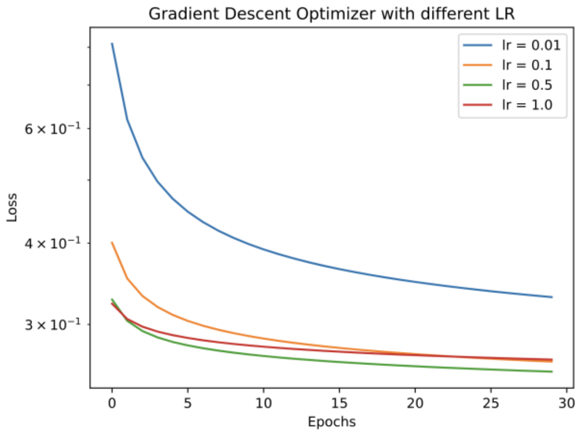


Figure 6: Gradient Descent for different learning rate values in MNIST dataset.

Momentum Optimizer

In exercise 1, momentum showed to be an important factor for the optimizer. In this case, momentum slightly affects the performance but the difference doesn't seem to be really important. The best combination obtained, in this case is learning rate of 0.05 and momentum of 0.9.

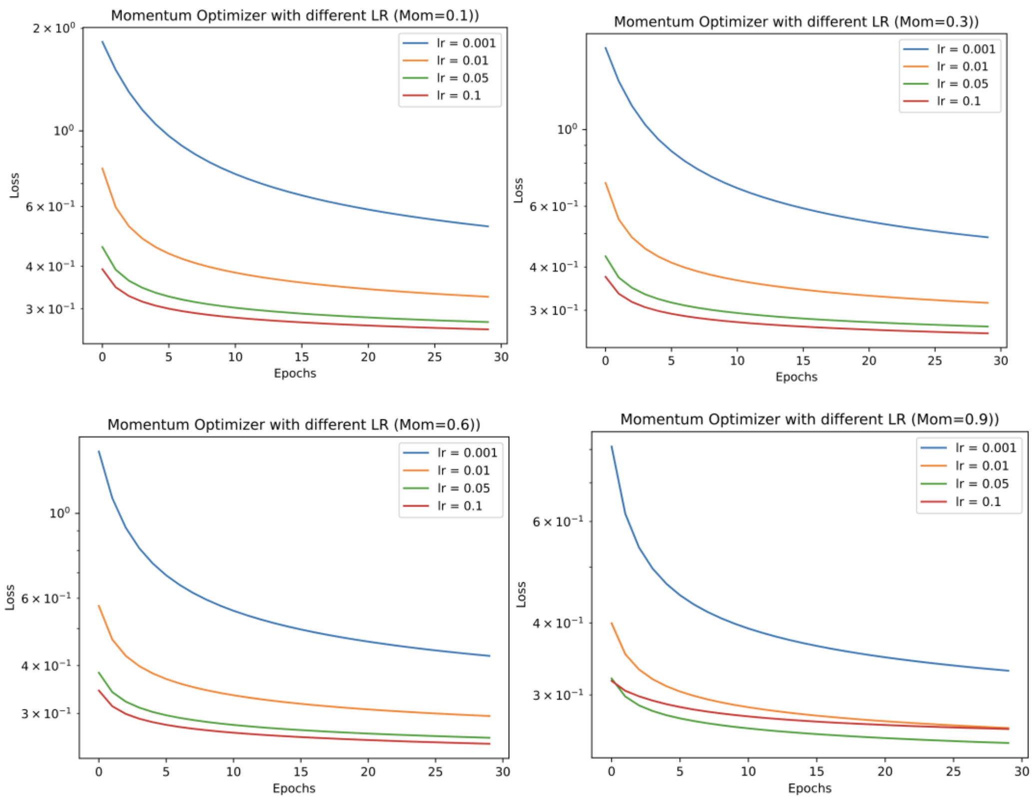


Figure 7: Momentum Optimizer for different learning rate and momentum values in MNIST dataset.

Adam Optimizer

In this case, the best learning rate seems to be 0.005.

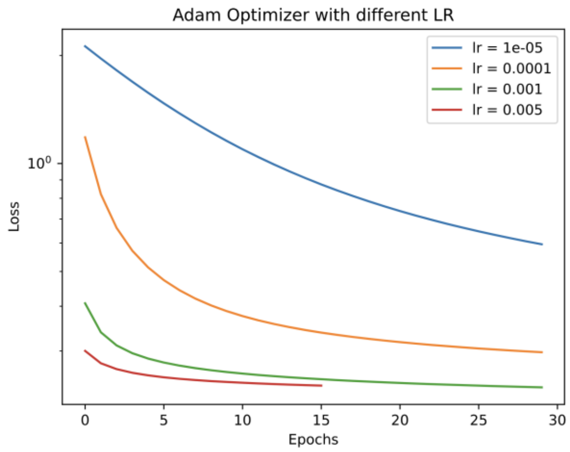


Figure 8: Adam Optimizer convergence rate for different learning rate values in MNIST dataset.

Comparison of three optimization algorithms

Finally, the best learning rates are chosen for each of the algorithms and the number of epochs is increased. This way, once again, Adam optimizer outperforms, being Momentum Optimizer in second position and Gradient Descent Optimizer in third position. However, the difference is smaller than in the case of the first problem. The main reason is the use of different loss metrics (MSE vs Categorical Cross Entropy), which in this case is not convex with respect to the parameters making more difficult to reach a global optimum.

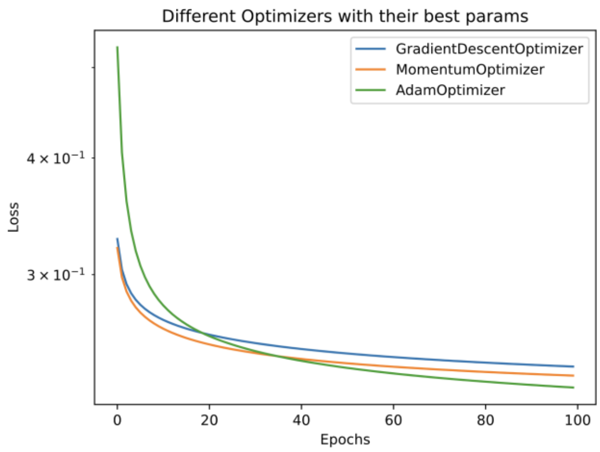


Figure 9: Comparison of all the algorithms for MNIST dataset.

Exercise 3: Increase accuracy rate as much as possible.

In the original code, only one epoch was being trained. The code has been modified (ex3_ImprovePerformance/multilayer.py) so that more epochs can be trained and validation data is used to monitor performance without incurring in overfitting.

Values for different trainings were between 0.992 and 0.993. The one reported is the one providing 0.992 accuracy in test set.

Configuration:

- Learning rate: 1e-4
- Batch Size: 50
- Epochs: 30
- Total training time: 78.419s

```
EPOCH 4/30 -- Loss 0.041344 -- Acc: 0.987140 -- Val Loss 0.054029 -- Val Acc 0.983900-- time: 2.553
EPOCH 5/30 -- Loss 0.044118 -- Acc: 0.984860 -- Val Loss 0.059687 -- Val Acc 0.981900-- time: 2.580
EPOCH 6/30 -- Loss 0.025023 -- Acc: 0.992360 -- Val Loss 0.046001 -- Val Acc 0.987500-- time: 2.509
EPOCH 7/30 -- Loss 0.023010 -- Acc: 0.992900 -- Val Loss 0.043044 -- Val Acc 0.987500-- time: 2.586
EPOCH 8/30 -- Loss 0.019184 -- Acc: 0.994400 -- Val Loss 0.041859 -- Val Acc 0.988100-- time: 2.549
EPOCH 9/30 -- Loss 0.012843 -- Acc: 0.996480 -- Val Loss 0.037061 -- Val Acc 0.988700-- time: 2.539
EPOCH 10/30 -- Loss 0.012511 -- Acc: 0.996640 -- Val Loss 0.039847 -- Val Acc 0.987900-- time: 2.533
EPOCH 11/30 -- Loss 0.009058 -- Acc: 0.997720 -- Val Loss 0.037173 -- Val Acc 0.989600-- time: 2.595
EPOCH 12/30 -- Loss 0.007599 -- Acc: 0.997940 -- Val Loss 0.035130 -- Val Acc 0.989800-- time: 2.564
EPOCH 13/30 -- Loss 0.007741 -- Acc: 0.997920 -- Val Loss 0.039236 -- Val Acc 0.989800-- time: 2.548
EPOCH 14/30 -- Loss 0.005141 -- Acc: 0.998700 -- Val Loss 0.038372 -- Val Acc 0.990300-- time: 2.504
EPOCH 15/30 -- Loss 0.004308 -- Acc: 0.998780 -- Val Loss 0.036707 -- Val Acc 0.990900-- time: 2.540
EPOCH 16/30 -- Loss 0.003883 -- Acc: 0.998980 -- Val Loss 0.037293 -- Val Acc 0.990100-- time: 2.566
EPOCH 17/30 -- Loss 0.003949 -- Acc: 0.998860 -- Val Loss 0.040933 -- Val Acc 0.989700-- time: 2.567
EPOCH 18/30 -- Loss 0.002511 -- Acc: 0.999520 -- Val Loss 0.035828 -- Val Acc 0.990700-- time: 2.540
EPOCH 19/30 -- Loss 0.002344 -- Acc: 0.999520 -- Val Loss 0.039346 -- Val Acc 0.990600-- time: 2.531
EPOCH 20/30 -- Loss 0.002355 -- Acc: 0.999500 -- Val Loss 0.035842 -- Val Acc 0.991400-- time: 2.486
EPOCH 21/30 -- Loss 0.001590 -- Acc: 0.999740 -- Val Loss 0.039414 -- Val Acc 0.991300-- time: 2.500
EPOCH 22/30 -- Loss 0.002367 -- Acc: 0.999400 -- Val Loss 0.037076 -- Val Acc 0.990900-- time: 2.491
EPOCH 23/30 -- Loss 0.001567 -- Acc: 0.999660 -- Val Loss 0.043986 -- Val Acc 0.990400-- time: 2.535
EPOCH 24/30 -- Loss 0.001099 -- Acc: 0.999800 -- Val Loss 0.038834 -- Val Acc 0.991400-- time: 2.550
EPOCH 25/30 -- Loss 0.001103 -- Acc: 0.999860 -- Val Loss 0.039401 -- Val Acc 0.991700-- time: 2.564
EPOCH 26/30 -- Loss 0.000674 -- Acc: 0.999940 -- Val Loss 0.036105 -- Val Acc 0.992100-- time: 2.533
EPOCH 27/30 -- Loss 0.000915 -- Acc: 0.999880 -- Val Loss 0.041103 -- Val Acc 0.990900-- time: 2.533
EPOCH 28/30 -- Loss 0.001036 -- Acc: 0.999700 -- Val Loss 0.037725 -- Val Acc 0.992100-- time: 2.555
EPOCH 29/30 -- Loss 0.001234 -- Acc: 0.999720 -- Val Loss 0.044305 -- Val Acc 0.991100-- time: 2.572
EPOCH 30/30 -- Loss 0.000389 -- Acc: 0.999960 -- Val Loss 0.038491 -- Val Acc 0.992000-- time: 2.560
Total Training Time: 78.419 seconds
test accuracy 0.992
```

Figure 10: Training multilayer network in MNIST dataset.

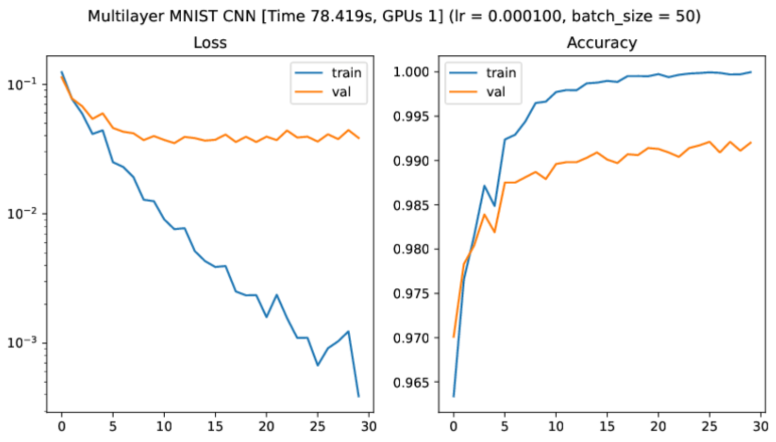


Figure 11: Training multilayer network in MNIST dataset.

Exercise 4: Use multiple GPU.

For implementing in our code the use of different GPUs, what we do is, for each epoch, train the same batch in all the gpus. This way, the training steps in one epoch are

N_GPUS*BATCH_SIZE. This way, if we use 2 GPUs, the same results than without GPU are obtained but in the half of epochs.

The piece of code responsible for this behavious is the one shown in Figure 12.

```
for epoch in range(1, N_EPOCHS+1):
  start_epoch_time = time.time()

  for i in range(len(data[0][0])//BATCH_SIZE):

    for j in range(N_GPU):
      with tf.device('/gpu:%d' %j):
        #until 1000 96,35%
        batch_ini = BATCH_SIZE*i
        batch_end = BATCH_SIZE*i+BATCH_SIZE

        batch_xs = data[0][0][batch_ini:batch_end]
        batch_ys = real_output[batch_ini:batch_end]

        train_step.run(feed_dict={x: batch_xs, y_: batch_ys, keep_prob: 0.5})
```

Figure 12: Code for multiGPU parallelization.

Increase number of GPUs

As it can be seen by comparing Figures 10 and 13, even when number of epochs is the half, the results are the same given that the number of training steps per epoch are doubled.

```
Num GPUs Available:  2
50000
50000
=====================================
TRAINING
EPOCH 1/15 -- Loss 0.090276 -- Acc: 0.972440 -- Val Loss 0.087022 -- Val Acc 0.976100-- time: 6.571
EPOCH 2/15 -- Loss 0.051206 -- Acc: 0.983180 -- Val Loss 0.057362 -- Val Acc 0.982800-- time: 4.757
EPOCH 3/15 -- Loss 0.032073 -- Acc: 0.989700 -- Val Loss 0.044583 -- Val Acc 0.986300-- time: 4.727
EPOCH 4/15 -- Loss 0.019082 -- Acc: 0.994120 -- Val Loss 0.038450 -- Val Acc 0.988700-- time: 4.695
EPOCH 5/15 -- Loss 0.016730 -- Acc: 0.994580 -- Val Loss 0.038704 -- Val Acc 0.988700-- time: 4.700
EPOCH 6/15 -- Loss 0.009990 -- Acc: 0.997160 -- Val Loss 0.033832 -- Val Acc 0.990700-- time: 4.776
EPOCH 7/15 -- Loss 0.005898 -- Acc: 0.998380 -- Val Loss 0.033163 -- Val Acc 0.991100-- time: 4.809
EPOCH 8/15 -- Loss 0.005490 -- Acc: 0.998400 -- Val Loss 0.034844 -- Val Acc 0.990900-- time: 4.870
EPOCH 9/15 -- Loss 0.003810 -- Acc: 0.999040 -- Val Loss 0.036343 -- Val Acc 0.991100-- time: 4.576
EPOCH 10/15 -- Loss 0.002554 -- Acc: 0.999320 -- Val Loss 0.033350 -- Val Acc 0.992100-- time: 4.543
EPOCH 11/15 -- Loss 0.002808 -- Acc: 0.999220 -- Val Loss 0.035369 -- Val Acc 0.992300-- time: 4.698
EPOCH 12/15 -- Loss 0.001836 -- Acc: 0.999480 -- Val Loss 0.034977 -- Val Acc 0.992400-- time: 4.653
EPOCH 13/15 -- Loss 0.001525 -- Acc: 0.999600 -- Val Loss 0.034645 -- Val Acc 0.992600-- time: 4.696
EPOCH 14/15 -- Loss 0.001180 -- Acc: 0.999760 -- Val Loss 0.038371 -- Val Acc 0.992300-- time: 4.664
EPOCH 15/15 -- Loss 0.000656 -- Acc: 0.999940 -- Val Loss 0.034928 -- Val Acc 0.992600-- time: 4.704
Total Training Time: 72.439 seconds
test accuracy 0.992
```

Figure 13: Training multilayer network in MNIST dataset with 2 GPUs.

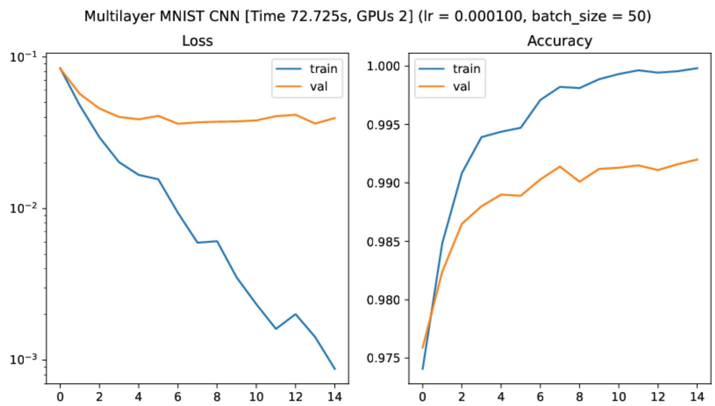
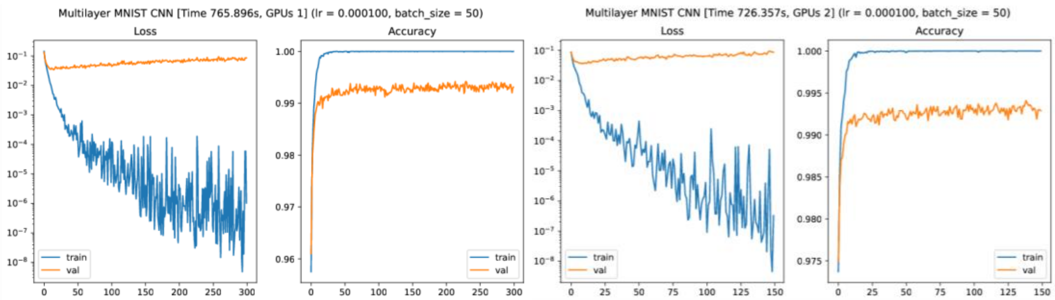


Figure 14: Training multilayer network in MNIST dataset with 2 GPUs.

Increase number of epochs to detect improvement in performance

For the case of 2 GPUs no significant improvement in performance is achieved. Probably this can be noticed in longer trainings. This is why number of epochs is increased. In Figure 15 some improvement in performance can be observed. However, when repeated for 4 GPUs,




```
EPOCH 291/300 -- Loss 0.000001 -- Acc: 1.000000 -- Val Loss 0.076347 -- Val Acc$
EPOCH 292/300 -- Loss 0.000003 -- Acc: 1.000000 -- Val Loss 0.072175 -- Val Acc$
EPOCH 293/300 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.085759 -- Val Acc$
EPOCH 294/300 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.083527 -- Val Acc$
EPOCH 295/300 -- Loss 0.000001 -- Acc: 1.000000 -- Val Loss 0.072252 -- Val Acc$
EPOCH 296/300 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.074455 -- Val Acc$
EPOCH 297/300 -- Loss 0.000001 -- Acc: 1.000000 -- Val Loss 0.072243 -- Val Acc$
EPOCH 298/300 -- Loss 0.000059 -- Acc: 0.999960 -- Val Loss 0.086586 -- Val Acc$
EPOCH 299/300 -- Loss 0.000059 -- Acc: 0.999980 -- Val Loss 0.082880 -- Val Acc$
EPOCH 300/300 -- Loss 0.000001 -- Acc: 1.000000 -- Val Loss 0.086236 -- Val Acc$
Total Training Time: 765.896 seconds
test accuracy 0.993
```

```
EPOCH 141/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.075134 -- Val Acc 0.993500-- time: 4.846
EPOCH 142/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.074855 -- Val Acc 0.993600-- time: 4.841
EPOCH 143/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.081951 -- Val Acc 0.993400-- time: 4.796
EPOCH 144/150 -- Loss 0.000004 -- Acc: 1.000000 -- Val Loss 0.079026 -- Val Acc 0.992900-- time: 4.846
EPOCH 145/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.073568 -- Val Acc 0.993200-- time: 4.869
EPOCH 146/150 -- Loss 0.000002 -- Acc: 1.000000 -- Val Loss 0.089278 -- Val Acc 0.993000-- time: 4.878
EPOCH 147/150 -- Loss 0.000052 -- Acc: 0.999980 -- Val Loss 0.095003 -- Val Acc 0.991900-- time: 4.860
EPOCH 148/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.091291 -- Val Acc 0.993000-- time: 4.977
EPOCH 149/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.088443 -- Val Acc 0.993000-- time: 4.786
EPOCH 150/150 -- Loss 0.000000 -- Acc: 1.000000 -- Val Loss 0.086184 -- Val Acc 0.992900-- time: 4.843
Total Training Time: 726.357 seconds
test accuracy 0.993
```

Figure 15: Training multilayer network in MNIST dataset with 2 GPUs for a larger amount of epochs.

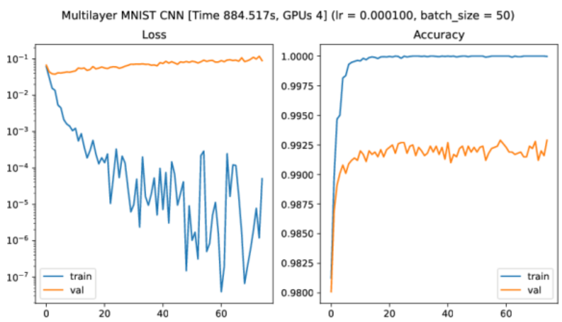


Figure 16: Training multilayer network in MNIST dataset with 4 GPUs.

Results for this configuration are 765 seconds for 1 GPU, 726 FOR 2 GPUs and 884 seconds for 4 GPUs. As a conclusion, improving the number of epochs doesn't take advantage of parallelization.

Increase batch size to detect improvement in performance

To exploit the computational advantages, larger batch sizes are needed. The advantage of multiple GPU executions is being able to accomplish complex tasks at the same time.

Increasing batch size from 50 to 5000 samples, obtained computation times are the following:

- 1 GPU: 285 seconds
- 2 GPUs: 233 seconds

- 4 GPUs: 204 seconds

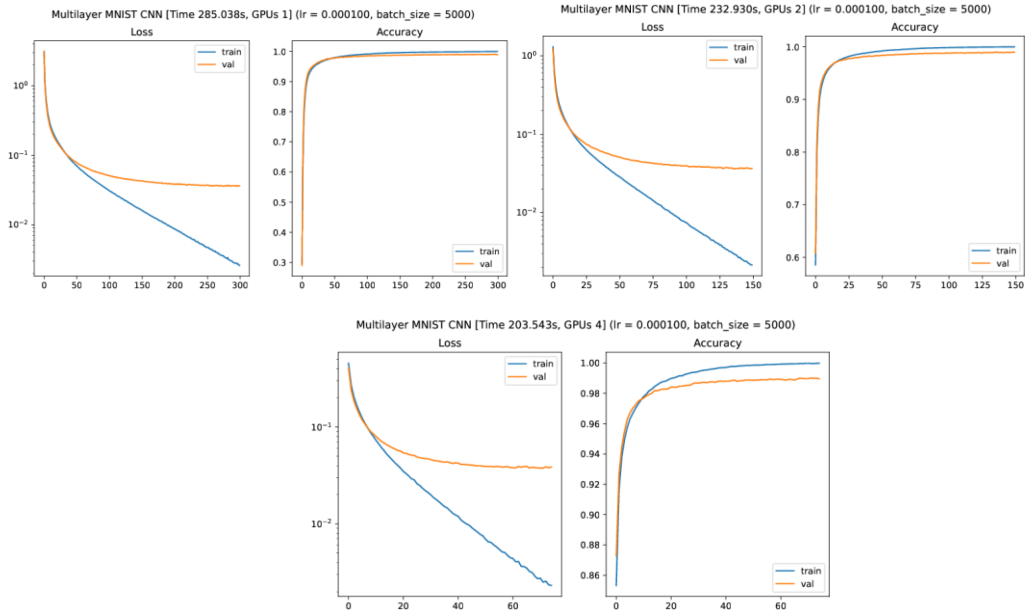


Figure 17: Training results and times for 1, 2 and 4 GPUs with a batch size of 5000.

In this case, significantly improvement is showed with the increase of the number of GPUs.

Conclusion

This study delved into high-performance computing (HPC) and deep learning, specifically exploring optimization algorithms and strategies. The research examined Gradient Descent with varying learning rates, Momentum Optimizer, and Adam Optimizer. Notably, Adam outperformed the others, showcasing adaptive learning rates and effective convergence.

The application of these algorithms to the MNIST dataset revealed nuanced results. The study emphasized the importance of selecting optimal learning rates for different optimizers. For instance, the optimal learning rate for Gradient Descent on MNIST was found to be 0.5.

Efforts were made to enhance performance, including code modifications for more epochs and validation data utilization. The achieved accuracy on the test set was 0.992, with a configuration involving a learning rate of 1e-4, batch size of 50, and 30 epochs.

The exploration extended to the use of multiple GPUs. The study demonstrated that, while utilizing more GPUs and larger batch sizes could significantly improve computational efficiency, increasing the number of epochs did not necessarily yield performance benefits.

In conclusion, the research contributes valuable insights into optimizing deep learning models. It underscores the significance of algorithm and hyperparameter selection, offering practical applications on the MNIST dataset and showcasing the advantages of parallelization strategies with multiple GPUs.