UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Deep Learning

High Performance Computing

Professors
CASAS, Marc

Team members
BEJARANO SEPULVEDA, Edison Jair
edison.bejarano@estudiantat.upc.edu

Polytechnic University of Catalonia
C/Jordi Girona Salgado,1-3 08034 BARCELONA Spain
Master's degree in Artificial Intelligence
Barcelona, 2023

Contents

Keywords: High performance computing, Deep learning, Optimization, Performance.

1 Introduction

High performance computing refers to the use of supercomputers, parallel processing techniques, and other advanced hardware and software technologies to solve complex computational problems. Deep learning, a subfield of machine learning, involves the use of artificial neural networks to analyze and model large datasets. Optimization problems involve finding the optimal solution to a particular problem or task. Graphics processing units (GPUs) are specialized chips designed to accelerate the processing of graphics and are increasingly being used for general purpose computing, including deep learning. In this report, we will explore the ways in which high performance computing, deep learning, optimization, and GPUs intersect and how they can be used together to tackle complex problems in various problems.

The development of the lab can be found in the next repository .

2 Exercise 1

2.1 Testing with different Learning rates

It was conducted an experiment in which it was used the same model and optimizer, but varied the learning rate to find the optimal value for the problem.

Learning rate: 0.01

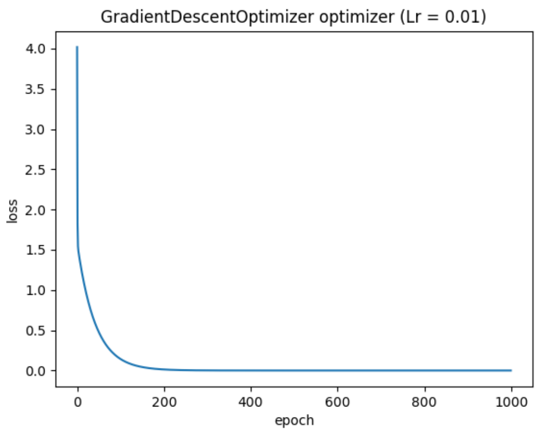


Figure 1 – Loss vs Epochs (LR: 0.01)

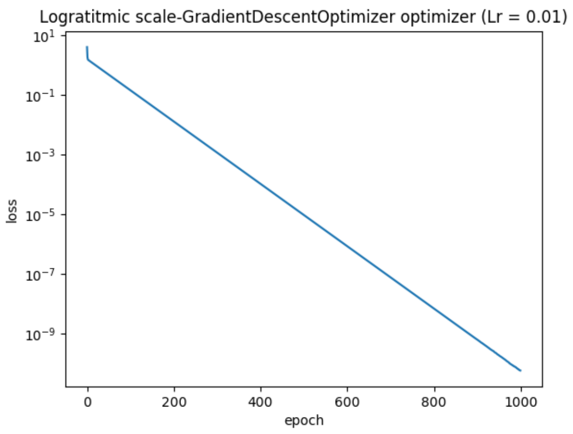


Figure 2 – Loss Log scale vs Epochs (LR: 0.01)

This configuration showed good behavior and, when plotted on a logarithmic scale, resulted in a linear line with the exception of the first few epochs. On other hand, the loss value it is reduced a lot in the first epochs.

Learning rate: 0.001

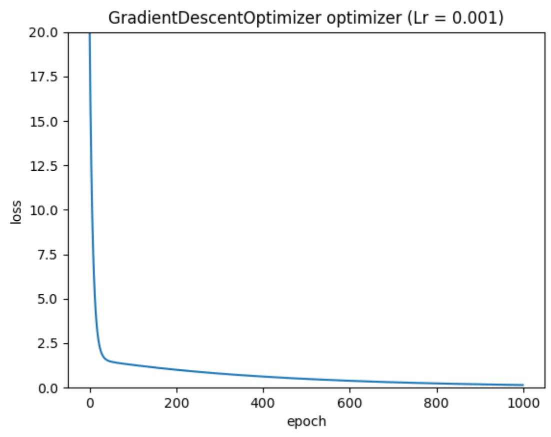


Figure 3 – Loss vs Epochs (LR: 0.01)

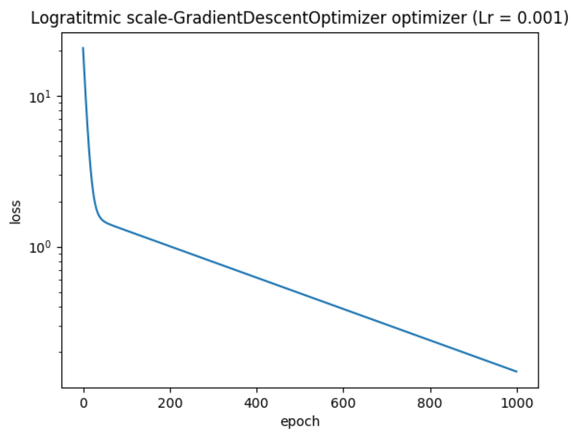


Figure 4 – Loss Log scale vs Epochs (LR: 0.01)

In this case, it was observed that the loss line rapidly descends in the first few epochs, but the accompanying plot shows that the behavior is not linear at all.

Learning rate: 0.0001

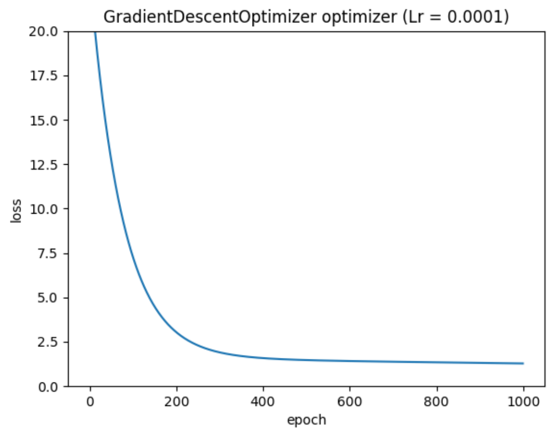


Figure 5 – Loss vs Epochs (LR: 0.01)

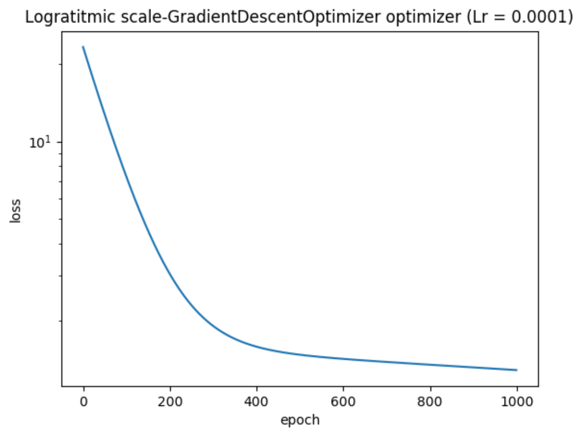


Figure 6 – Loss Log scale vs Epochs (LR: 0.01)

Compared to the previous experiment, the loss curve exhibits better behavior, although the logarithmic plot reveals that the behavior is not linear.

Learning rate: 1e-05

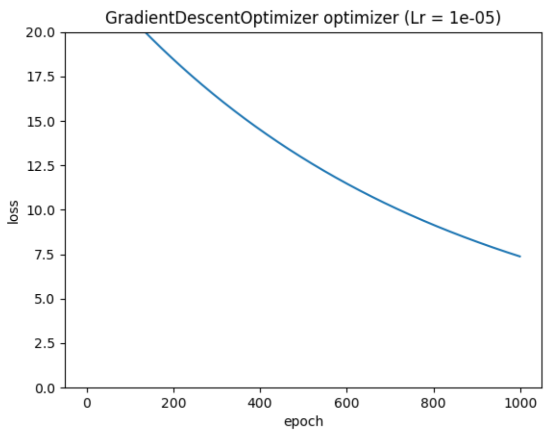


Figure 7 – Loss vs Epochs (LR: 0.01)

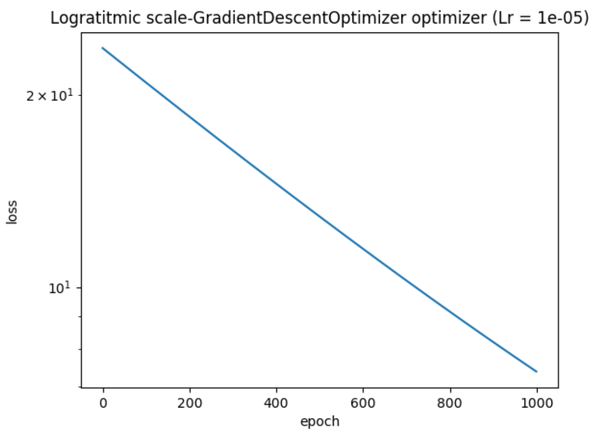


Figure 8 – Loss Log scale vs Epochs (LR: 0.01)

In this case, the logarithmic scale exhibits a linear behavior, but the loss value does not decrease below 0.75.

Learning rate: 1e-06

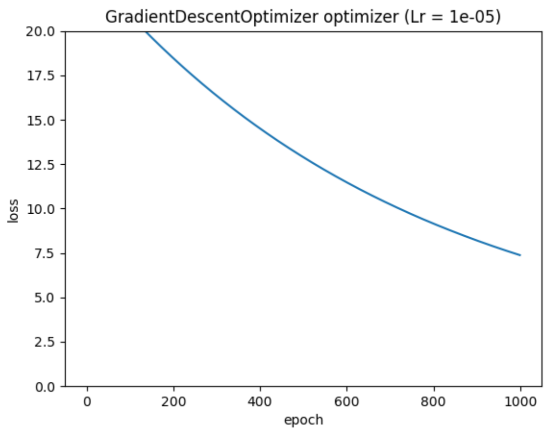


Figure 9 – Loss vs Epochs (LR: 0.01)

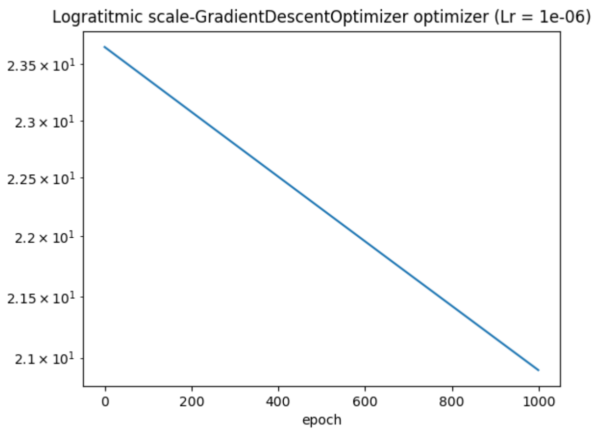


Figure 10 – Loss Log scale vs Epochs (LR: 1e-06)

Like the previous experiment, it was not possible to effectively decrease the loss, and the logarithmic scale plot showed a linear line.

Throughout these experiments, it was observed that the best behavior was consistently achieved with a learning rate of 0.01.

2.2 Testing with different Optimizers

For the following experiments, various optimizers were tested in an effort to improve the results

Optimizer: Adadelta - Learning rate: 0.01

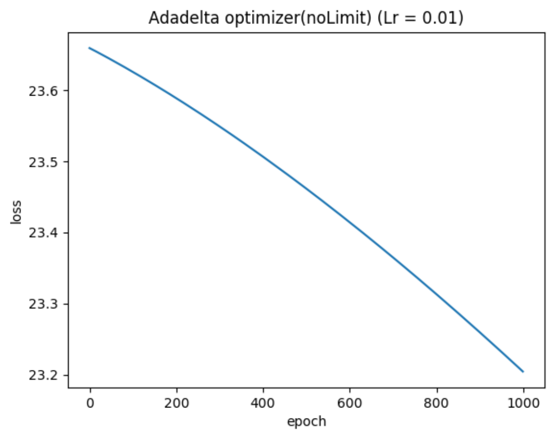


Figure 11 – Loss vs Epochs (LR: 0.01)

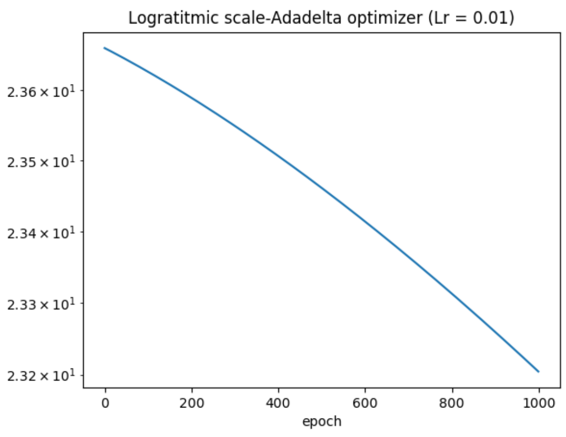


Figure 12 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Adagrad - Learning rate: 0.01

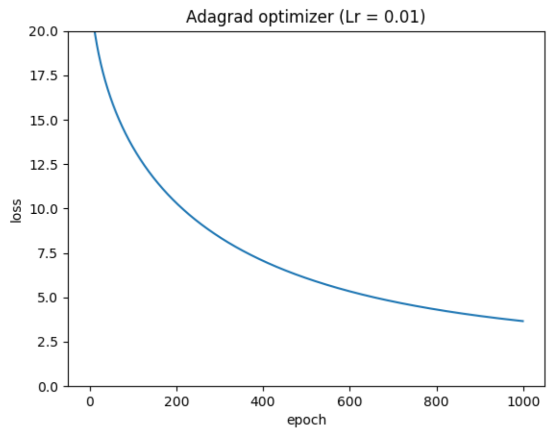


Figure 13 – Loss vs Epochs (LR: 0.01)

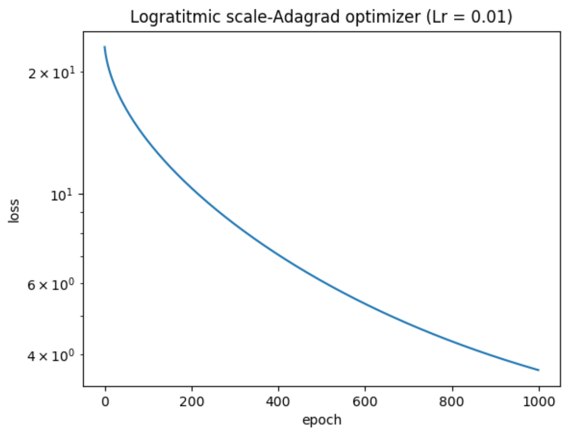


Figure 14 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Ftrl - Learning rate: 0.01

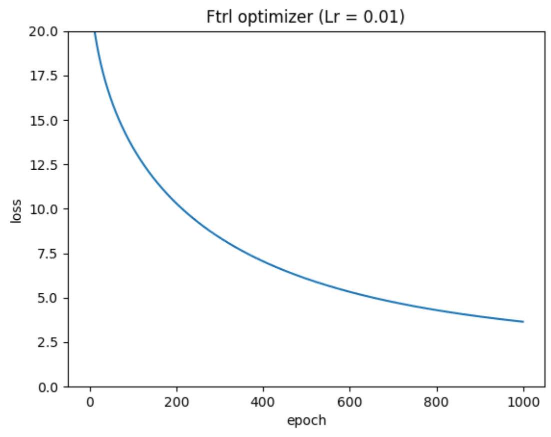


Figure 15 – Loss vs Epochs (LR: 0.01)

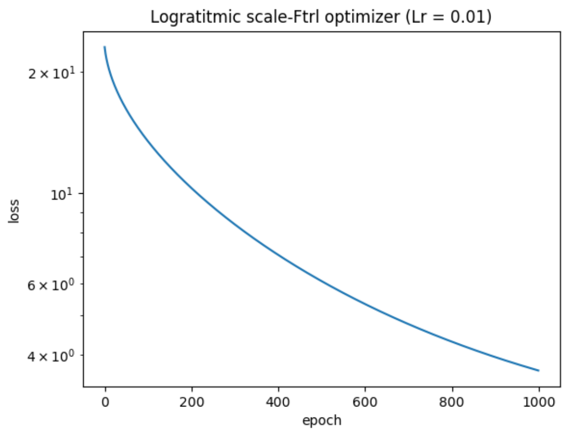


Figure 16 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Gradient Descent - Learning rate: 0.01

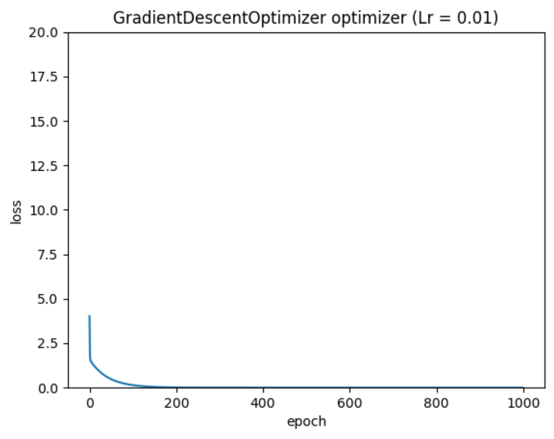


Figure 17 – Loss vs Epochs (LR: 0.01)

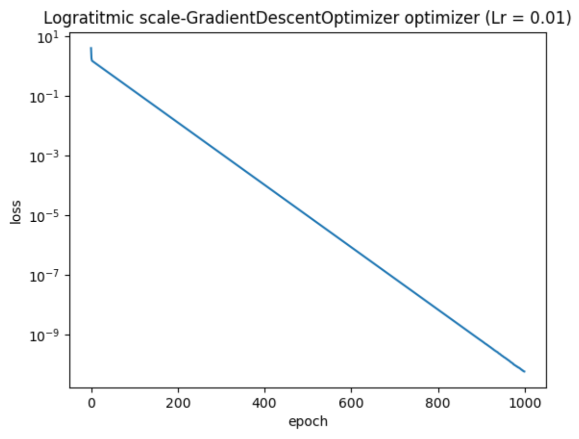


Figure 18 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Momentum with 0.9 - Learning rate: 0.01

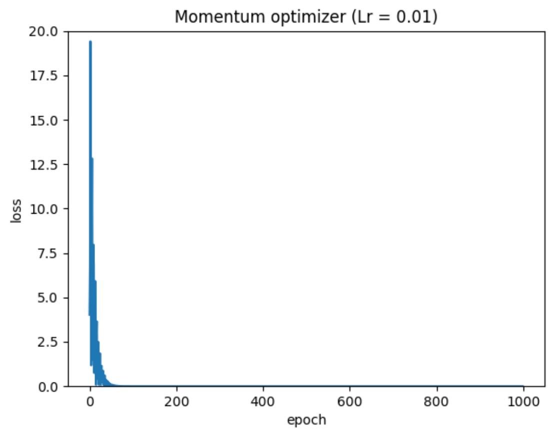


Figure 19 – Loss vs Epochs (LR: 0.01)

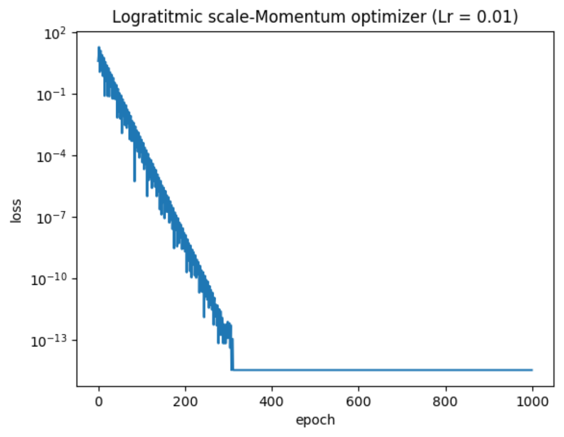


Figure 20 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Nesterov Accelerated (NAG) - Learning rate: 0.01

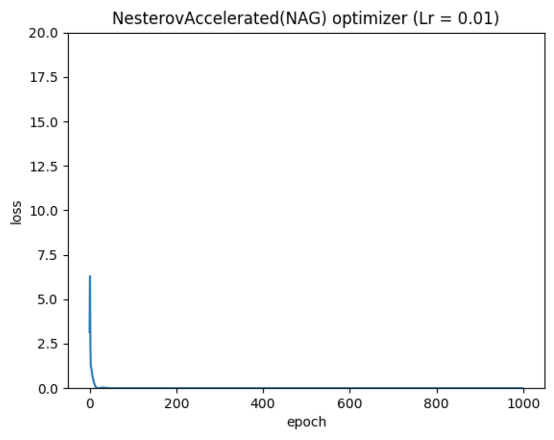


Figure 21 – Loss vs Epochs (LR: 0.01)

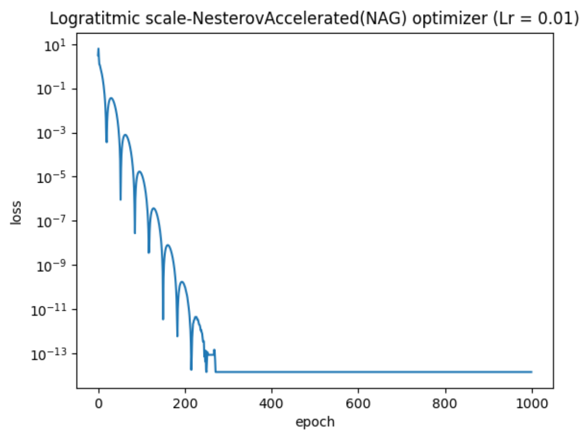


Figure 22 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Proximal Adagrad - Learning rate: 0.01

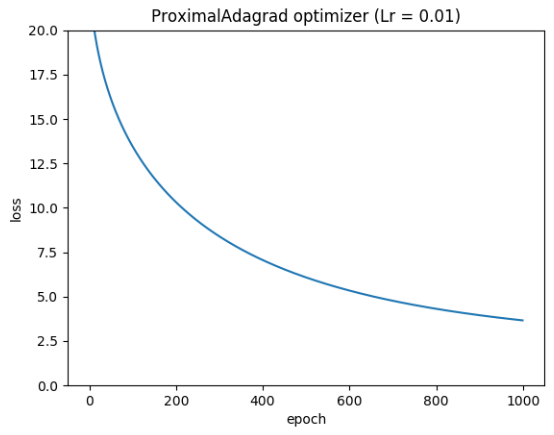


Figure 23 – Loss vs Epochs (LR: 0.01)

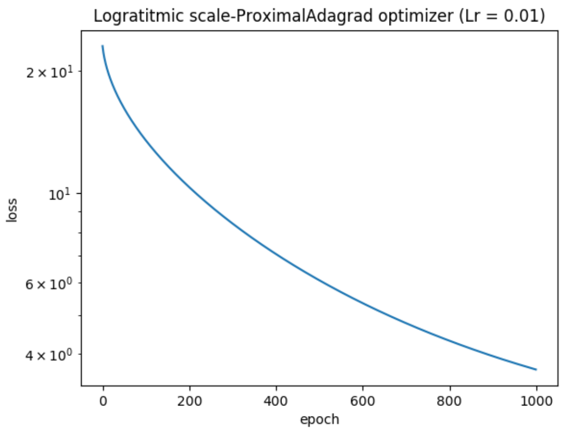


Figure 24 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: Proximal Gradient Descent - Learning rate: 0.01

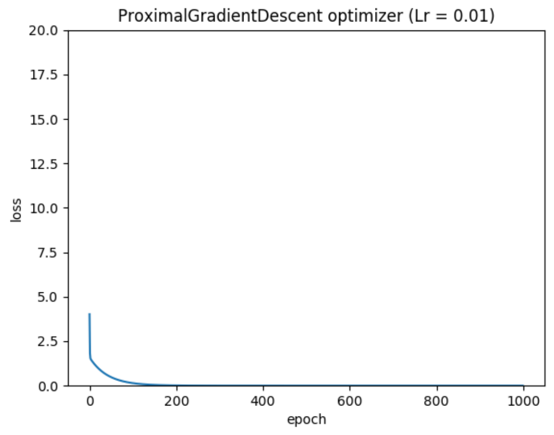


Figure 25 – Loss vs Epochs (LR: 0.01)

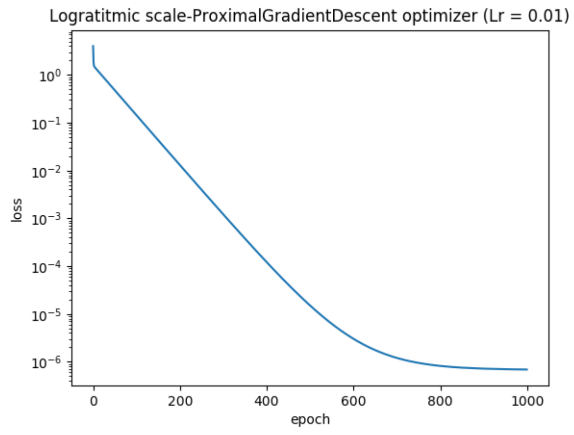


Figure 26 – Loss Log scale vs Epochs (LR: 0.01)

Optimizer: RMSProp - Learning rate: 0.01

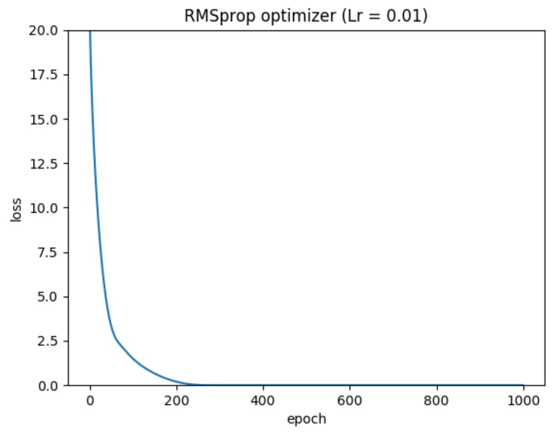


Figure 27 – Loss vs Epochs (LR: 0.01)

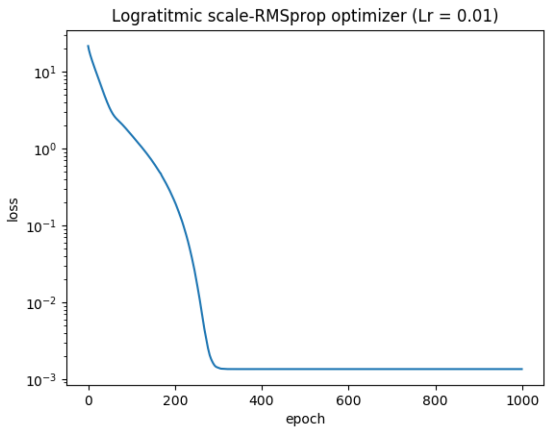


Figure 28 – Loss Log scale vs Epochs (LR: 0.01)

Throughout the previous experiments, it was observed that the best optimizers were proximal gradient descent, momentum, gradient descent, and Nesterov. The NAG optimizer was able to significantly decrease the loss value, but the linear scale plot showed occasional sudden jumps while still maintaining a general trend of improvement.

3 Exercise 2 - Single layer

It was selecting the optimizers with the highest performance from the previous exercise.

3.1 Adam optimizer vs different learning rates

Is an optimization algorithm introduced in a 2015 paper by Diederik Kingma and Jimmy Ba, and has since become a popular choice for training neural networks. Adam stands for Adaptive Moment Estimation, and is a gradient-based optimization algorithm that uses moving averages of the parameters to provide a running estimate of the second raw moments of the gradients; the term adaptive in the name refers to the fact that the algorithm ”adapts” the learning rates of each parameter based on the historical gradient information. Adam is generally considered to be an extension of the stochastic gradient descent (SGD) algorithm, and combines the benefits of SGD with those of more sophisticated optimization methods like Root Mean Square Propagation (RMSprop) and the Adaptive Learning Rate method (AdaGrad).

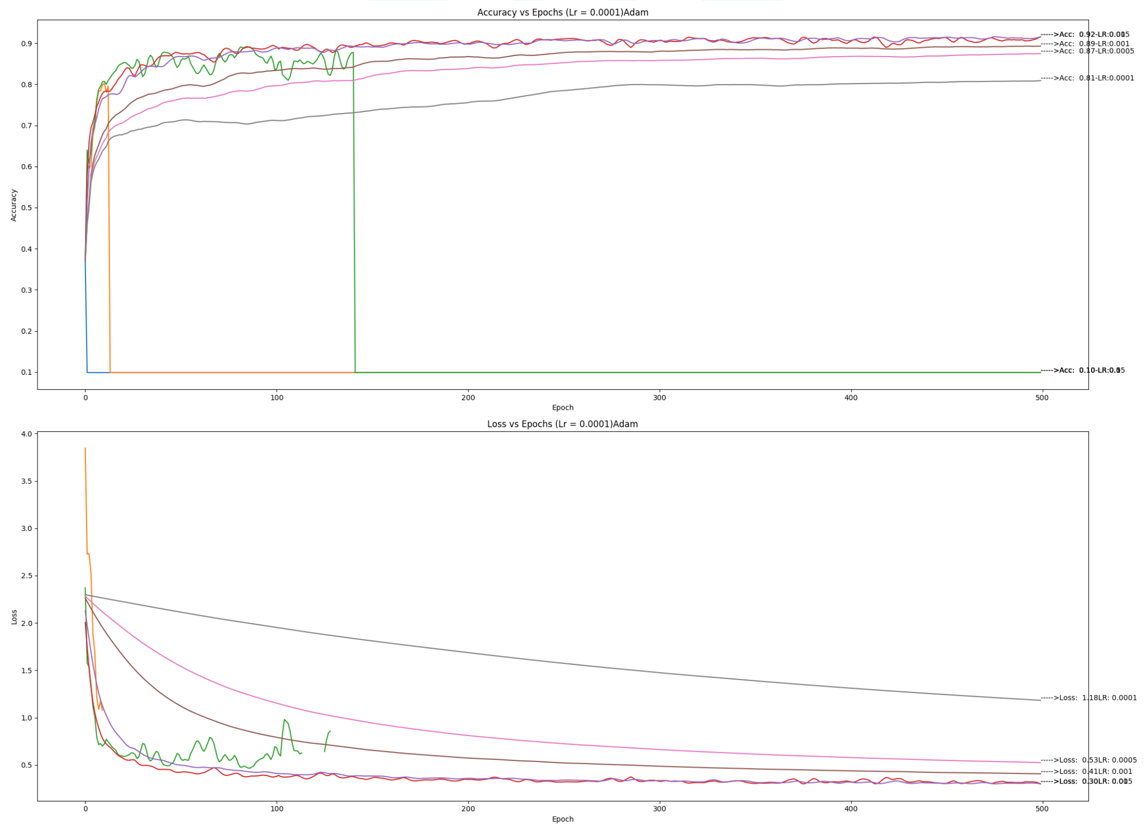


Figure 29 – Single layer example(Adam optimizer)

We observed that the best learning rate for this algorithm was 0.005, which resulted in an accuracy of 0.92. However, when using higher learning rates, the accuracy decreased to 0 after several epochs. For example, using a learning rate of 0.05, the accuracy dropped to 0.1 by epoch 140 (as shown by the green line in the top subplot).

3.2 Gradient Descent optimizer vs different learning rates

Gradient descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. It is called a gradient descent algorithm because it is based on the gradient of the function being minimized.

The algorithm starts with an initial guess for the parameters of the function and then iteratively adjusts the parameters in the direction that reduces the function's value until the minimum is found. The size of the step taken in the direction of the negative gradient is called the learning rate.

In other words, gradient descent is commonly used to train models by updating the model parameters to minimize the error between the predicted output and the ground truth label.

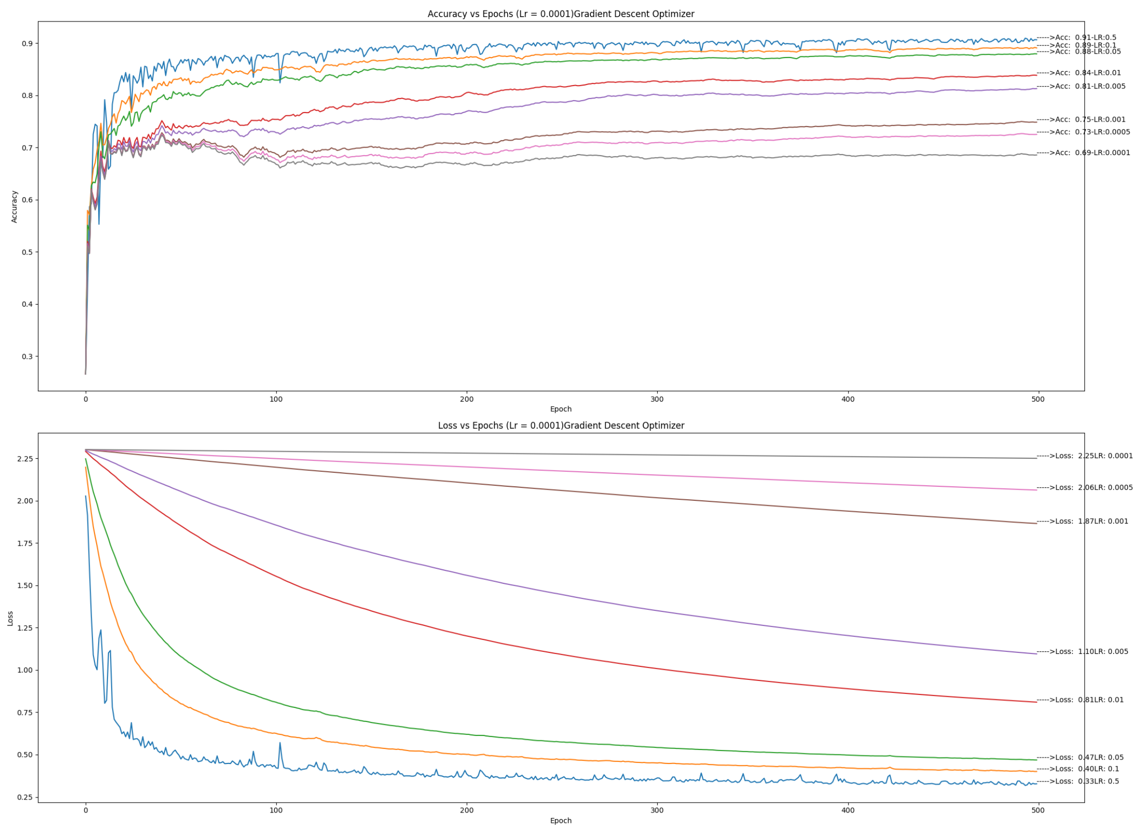


Figure 30 – Single layer example(Gradient Descent optimizer)

For the gradient descent optimization algorithm, it was observed that the best learning rate for convergence was 0.5, which resulted in an accuracy of 0.91 and a loss of 0.33. The worst performance was observed with a learning rate of 0.0001.

3.3 Momentum optimizer vs different learning rates

It is an extension of gradient descent that can accelerate the convergence of the model by allowing the model to build up velocity in the direction of the steepest descent.

In the context of neural networks, momentum can help the model to "break through" plateaus or local minima in the loss function that might otherwise slow down or halt the training process. It does this by adding a fraction of the update vector of the past time step to the current update vector. The fraction is called the momentum coefficient, and is usually set to a value between 0.5 and 0.9, and for this experiment it was defined in 0.9.

Momentum can often lead to faster convergence and better final performance of the model compared to vanilla gradient descent, but it can also make the training process more noisy and harder to debug.

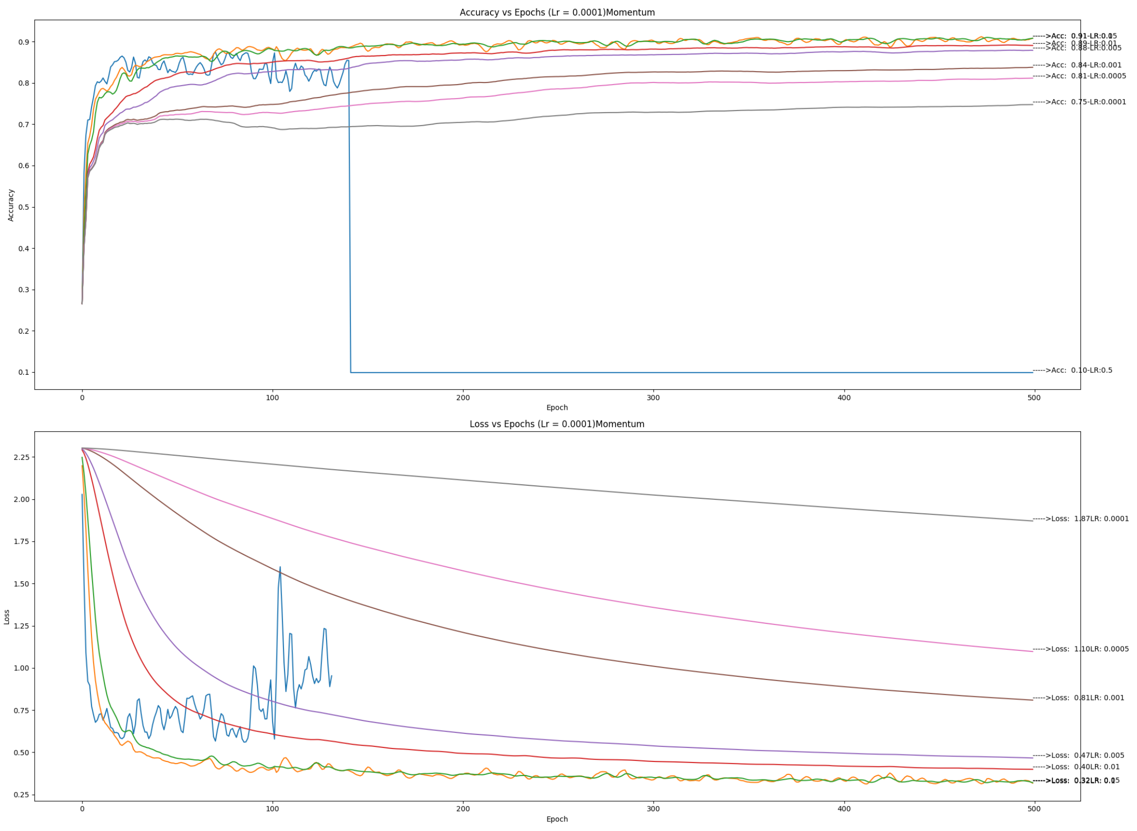


Figure 31 – Single layer example(Momentum optimizer)

In this experiment, the optimizer performed best with a learning rate of 0.05, resulting in an accuracy of 0.91 and a loss of 0.32. A learning rate of 0.5 resulted in erratic behavior, causing the accuracy to fluctuate and eventually decrease to 0.1 across all epochs.

3.4 Nesterov Accelerated optimizer vs different learning rates

Nesterov accelerated gradient (NAG) is a variant of the momentum optimization algorithm that was introduced by Yurii Nesterov in 1983. It is also an extension of gradient descent that can accelerate the convergence of the model by allowing the model to build up velocity in the direction of the steepest descent.

Like momentum, NAG uses an update vector to store a moving average of the past gradients, but it calculates the gradient at a point slightly ahead in the direction of the update vector rather than at the current position. This "lookahead" allows NAG to correct its direction earlier and leads to faster convergence compared to vanilla momentum.

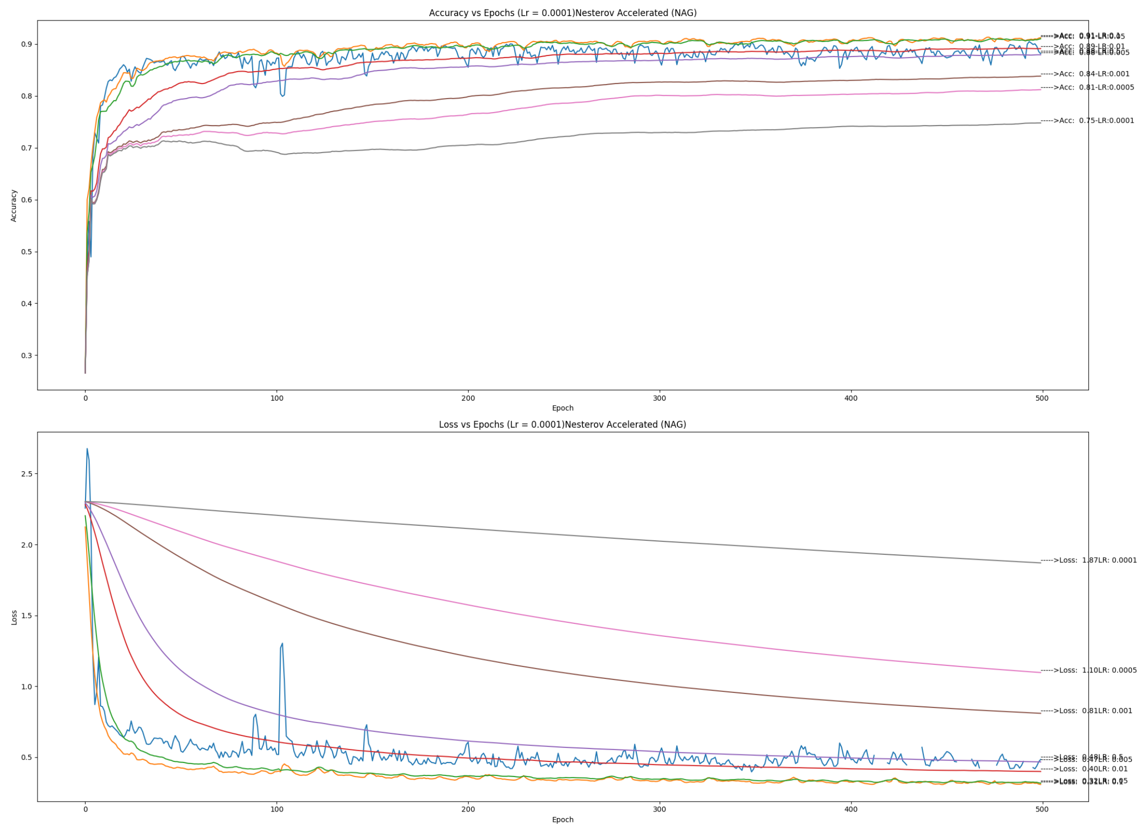


Figure 32 – Single layer example(Nesterov Accelerated optimize)

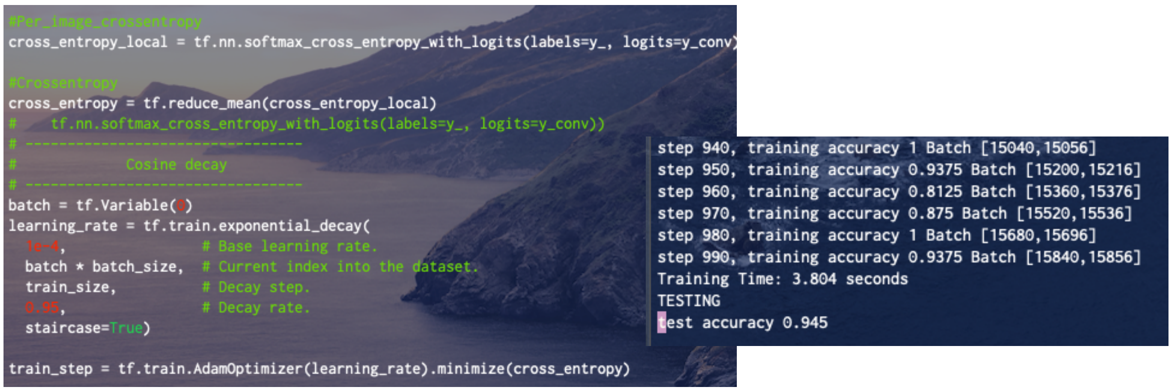
In this experiment, the best learning rate values were found to be 0.05 and 0.1, which resulted in a loss of 0.32 and an accuracy of 0.91. The convergence line for this optimizer was more pronounced, showing a rapid decrease in loss to 0.5 within 20 epochs. However, using a learning rate of 0.5 resulted in less fluctuation in the model's behavior.

4 Exercise 3 - Multi layer

In this part of the laboratory, a new script called 'multilayerModified.py' was created, which modified the original multilayer script in an attempt to increase the accuracy. The following adaptations were made in the script:

Cosine Decay

The accuracy test resulted in a score of 0.945, which did not show an improvement.



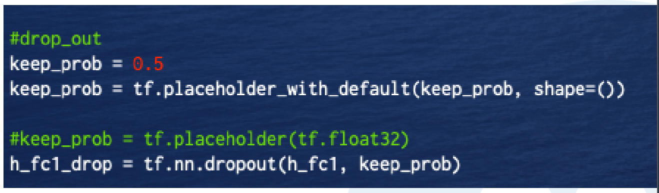
```
#Per_image_crossentropy
cross_entropy_local = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv)

#Crossentropy
cross_entropy = tf.reduce_mean(cross_entropy_local)
#    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
# --------------------------------
#              Cosine decay
# --------------------------------
batch = tf.Variable(0)
learning_rate = tf.train.exponential_decay(
  1e-4,                # Base learning rate.
  batch * batch_size,  # Current index into the dataset.
  train_size,          # Decay step.
  0.95,                # Decay rate.
  staircase=True)

train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

```
step 940, training accuracy 1 Batch [15040,15056]
step 950, training accuracy 0.9375 Batch [15200,15216]
step 960, training accuracy 0.8125 Batch [15360,15376]
step 970, training accuracy 0.875 Batch [15520,15536]
step 980, training accuracy 1 Batch [15680,15696]
step 990, training accuracy 0.9375 Batch [15840,15856]
Training Time: 3.804 seconds
TESTING
test accuracy 0.945
```

Figure 33 – Cosine decay modification and test accuracy

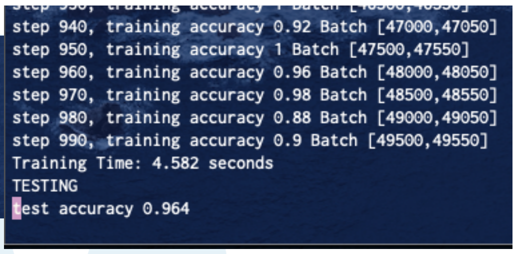
The accuracy test resulted in a score of 0.964, with the drop out including.

Drop: 0.5



```
#drop_out
keep_prob = 0.5
keep_prob = tf.placeholder_with_default(keep_prob, shape=())

#keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Figure 34 – Dropout modification in 0.5

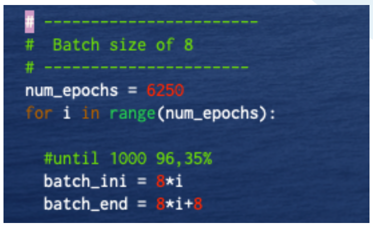


```
step 940, training accuracy 0.92 Batch [47000,47050]
step 950, training accuracy 1 Batch [47500,47550]
step 960, training accuracy 0.96 Batch [48000,48050]
step 970, training accuracy 0.98 Batch [48500,48550]
step 980, training accuracy 0.88 Batch [49000,49050]
step 990, training accuracy 0.9 Batch [49500,49550]
Training Time: 4.582 seconds
TESTING
test accuracy 0.964
```

Figure 35 – Test accuracy with drop changes

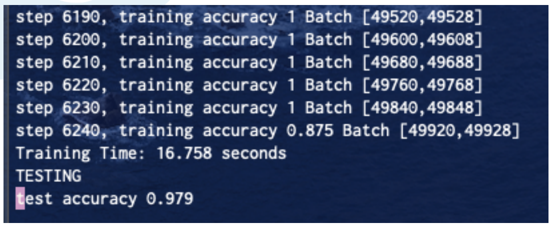
Updating the batch size for 8 it was attached increase the accuracy until 0.979.

Batch size to 8



```
# ----------------------
#  Batch size of 8
# ----------------------
num_epochs = 6250
for i in range(num_epochs):

  #until 1000 96,35%
  batch_ini = 8*i
  batch_end = 8*i+8
```

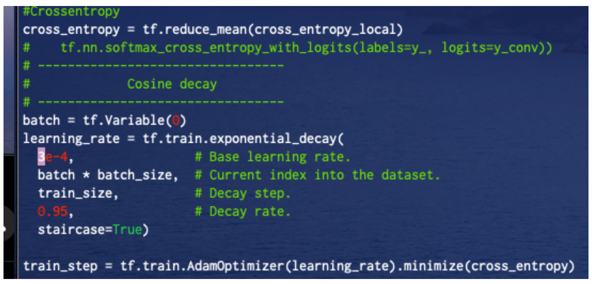
Figure 36 – Batch size modification from 50 to 8



```
step 6190, training accuracy 1 Batch [49520,49528]
step 6200, training accuracy 1 Batch [49600,49608]
step 6210, training accuracy 1 Batch [49680,49688]
step 6220, training accuracy 1 Batch [49760,49768]
step 6230, training accuracy 1 Batch [49840,49848]
step 6240, training accuracy 0.875 Batch [49920,49928]
Training Time: 16.758 seconds
TESTING
test accuracy 0.979
```

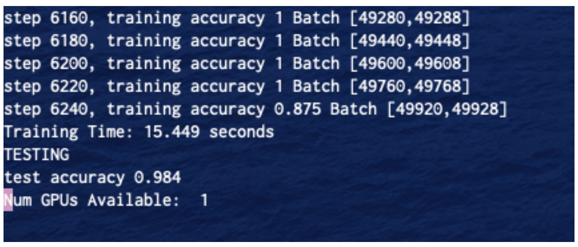
Figure 37 – Test accuracy results with batch size modified

Changing the learning rate in the cosine decay to 3e-4 resulted in an improvement in accuracy, reaching a score of 0.984. **Learning rate to 3e-4**



```
#Crossentropy
cross_entropy = tf.reduce_mean(cross_entropy_local)
#    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
# --------------------------------
#            Cosine decay
# --------------------------------
batch = tf.Variable(0)
learning_rate = tf.train.exponential_decay(
  3e-4,                # Base learning rate.
  batch * batch_size,  # Current index into the dataset.
  train_size,          # Decay step.
  0.95,                # Decay rate.
  staircase=True)

train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

Figure 38 – Modification of learning rate to 3e-4



```
step 6160, training accuracy 1 Batch [49280,49288]
step 6180, training accuracy 1 Batch [49440,49448]
step 6200, training accuracy 1 Batch [49600,49608]
step 6220, training accuracy 1 Batch [49760,49768]
step 6240, training accuracy 0.875 Batch [49920,49928]
Training Time: 15.449 seconds
TESTING
test accuracy 0.984
Num GPUs Available:  1
```

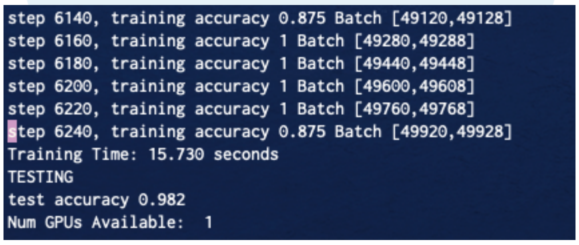
Figure 39 – Test accuracy results

The previous changes to the model resulted in a final accuracy of 0.984. However, further modifications to the batch size, learning rate, and the use of optimizers such as momentum did not yield good results. It also tried altering the percentage of the training and test sizes and found that the optimal configuration was a split of 80-20.

5 Exercise 4 - Multi layer

For the next exercise it was tested, to run the multi layer script in the next conditions:

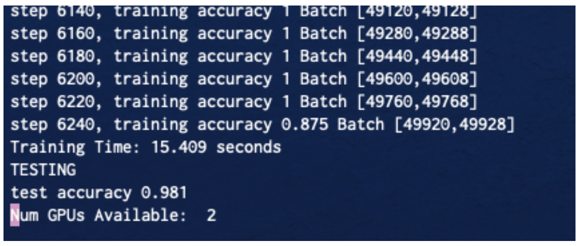
- 1 GPU
  - CPUs per task=40
  - Time:15.730
  - Test accuracy: 0.982



```
step 6140, training accuracy 0.875 Batch [49120,49128]
step 6160, training accuracy 1 Batch [49280,49288]
step 6180, training accuracy 1 Batch [49440,49448]
step 6200, training accuracy 1 Batch [49600,49608]
step 6220, training accuracy 1 Batch [49760,49768]
step 6240, training accuracy 0.875 Batch [49920,49928]
Training Time: 15.730 seconds
TESTING
test accuracy 0.982
Num GPUs Available:  1
```

Figure 40 – Multilayer.py script runned in 1 GPU

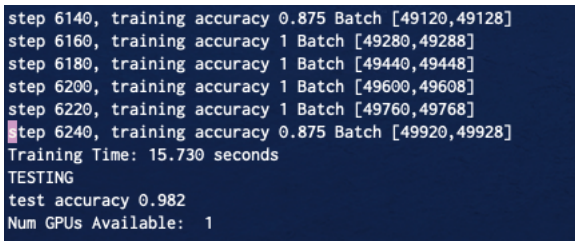
- 2 GPUs
  - CPUs per task=80
  - Time: 15.409
  - Test accuracy: 0.981



```
step 6140, training accuracy 1 Batch [49120,49128]
step 6160, training accuracy 1 Batch [49280,49288]
step 6180, training accuracy 1 Batch [49440,49448]
step 6200, training accuracy 1 Batch [49600,49608]
step 6220, training accuracy 1 Batch [49760,49768]
step 6240, training accuracy 0.875 Batch [49920,49928]
Training Time: 15.409 seconds
TESTING
test accuracy 0.981
Num GPUs Available:  2
```

Figure 41 – Multilayer.py script runned in 2 GPUs

- 4 GPUs
  - CPUs per task=160
  - Time: 15.730 s
  - Test accuracy: 0.982



```
step 6140, training accuracy 0.875 Batch [49120,49128]
step 6160, training accuracy 1 Batch [49280,49288]
step 6180, training accuracy 1 Batch [49440,49448]
step 6200, training accuracy 1 Batch [49600,49608]
step 6220, training accuracy 1 Batch [49760,49768]
step 6240, training accuracy 0.875 Batch [49920,49928]
Training Time: 15.730 seconds
TESTING
test accuracy 0.982
Num GPUs Available:  1
```

Figure 42 – Multilayer.py script runned in 4 GPUs

The speedup will depend on the type of model that are training,and also, the amount of data that is using, and the specific hardware and software configuration you are using. In general, adding more GPUs can help to parallelize the training process and make it faster, especially for larger models and datasets. However, there may also be some overhead involved in using multiple GPUs, so the actual speedup may not be exactly linear with the number of GPUs. It is also worth noting that not all models and tasks are able to take advantage of multiple GPUs, so you may not see a significant improvement in all cases.