Deep Learning Course
High Performance Computing Aspects of Deep Learning

Beata Baczynska

December 2021

Exercise 1: Optimization Problem

The goal of this exercise was to find linear model (W and b) that better fits a set of points.

Points:
x = [1, 2, 3, 4]
y = [0, -1, -2, -3]

Formula:
y = Wx + b

Different optimizers and learning rates were tried.
Optimizers that were tried:

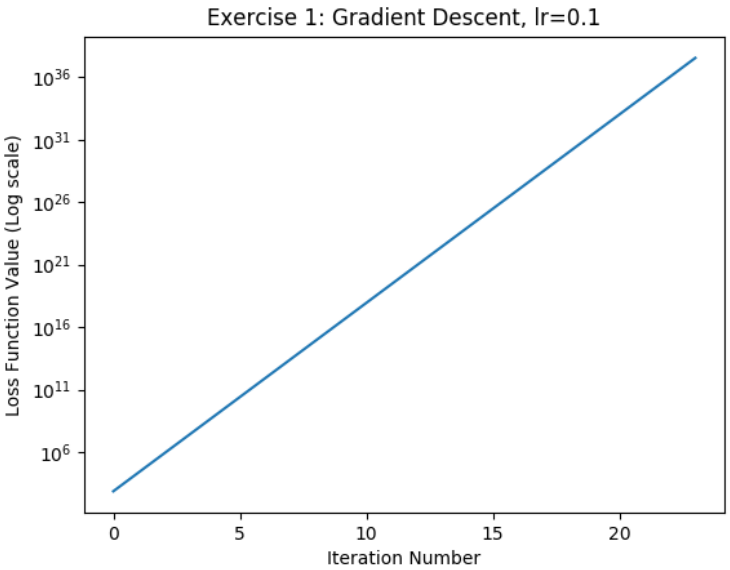
- Gradient descent
- Momentum
- Adagrad
- Adam

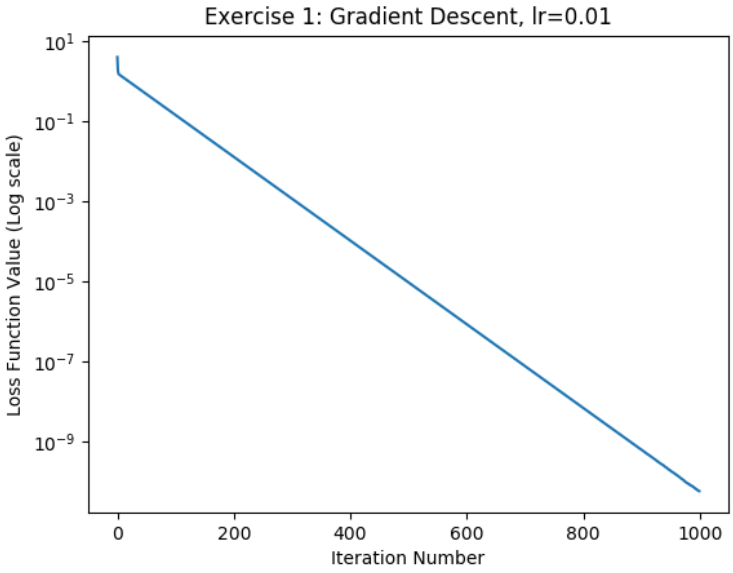
Optimizer: Gradient descent

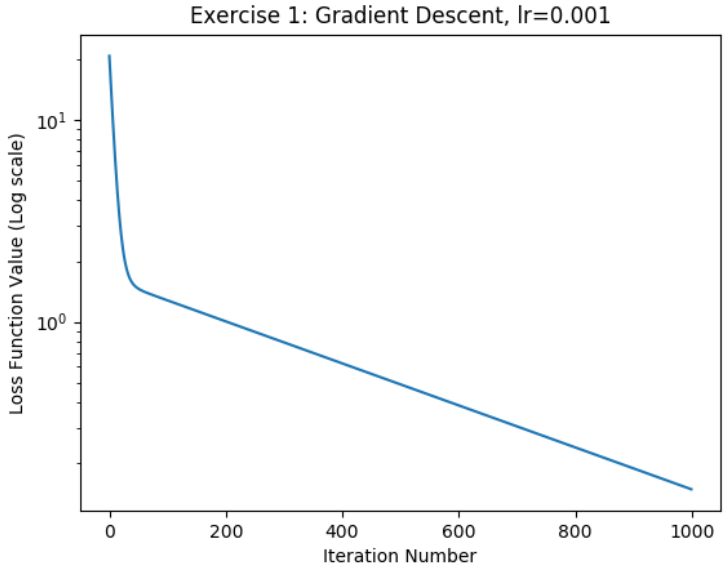
Learning rate	Loss
0.1	712
0.01	5.7e-11
0.001	0.15
0.0001	1.28

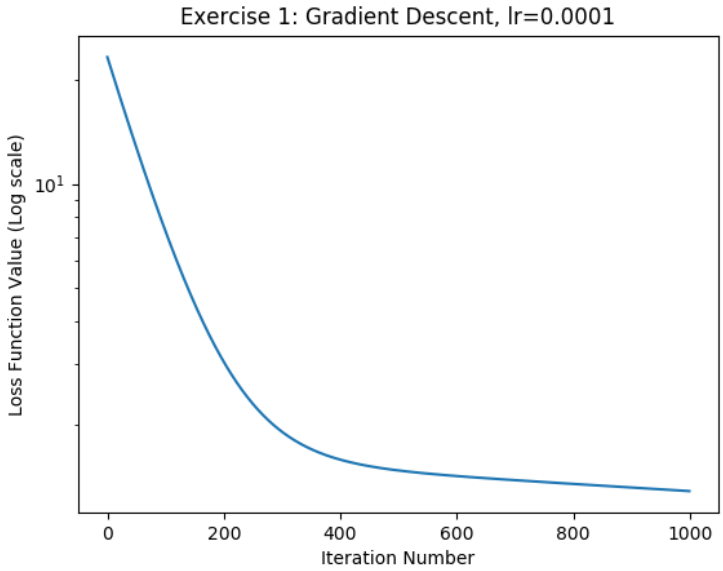
Gradient descent technique is minimizing the cost function by moving in the direction of steepest descent. The lr=0.1 was probably too big to find minimum, skipping the minimum with each change. The lr=0.001 and lr=0.0001 were too small to reach the minimum during the training. Increasing the number of training iterations would probably help. The lr=0.01 was the best choice for this problem. The value was not too big to miss minimum, and not too small that it was possible to find it during the training process.

Plots for exercise 1 - Gradient Descent




Exercise 1: Gradient Descent, lr=0.01
Loss Function Value (Log scale)
10^{1}
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
0
200
400
600
800
1000
Iteration Number


Exercise 1: Gradient Descent, lr=0.001
Loss Function Value (Log scale)
10^{1}
10^{0}
0
200
400
600
800
1000
Iteration Number

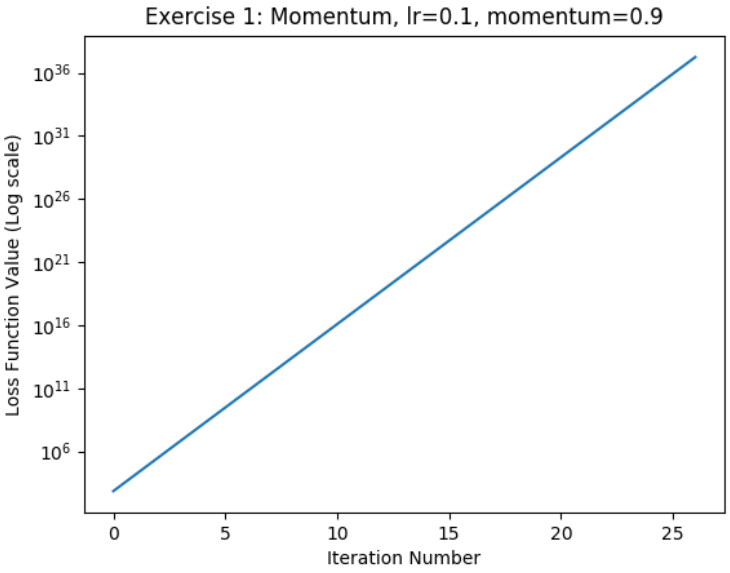


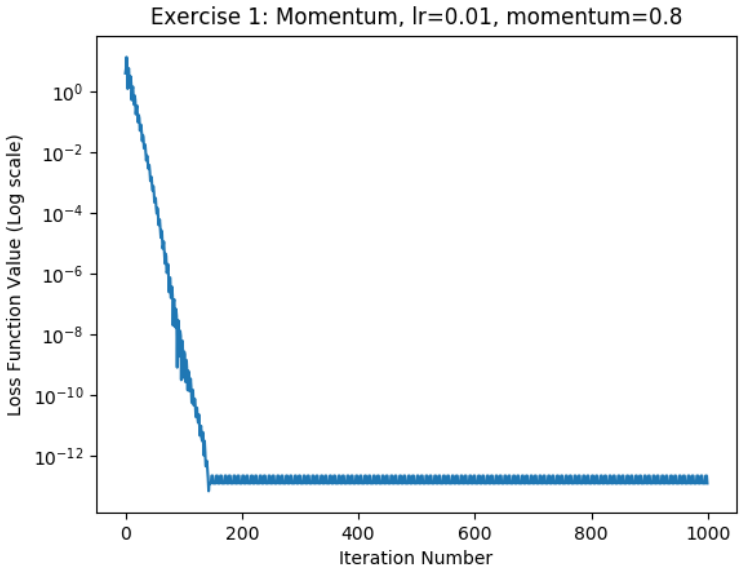
Optimizer: Momentum

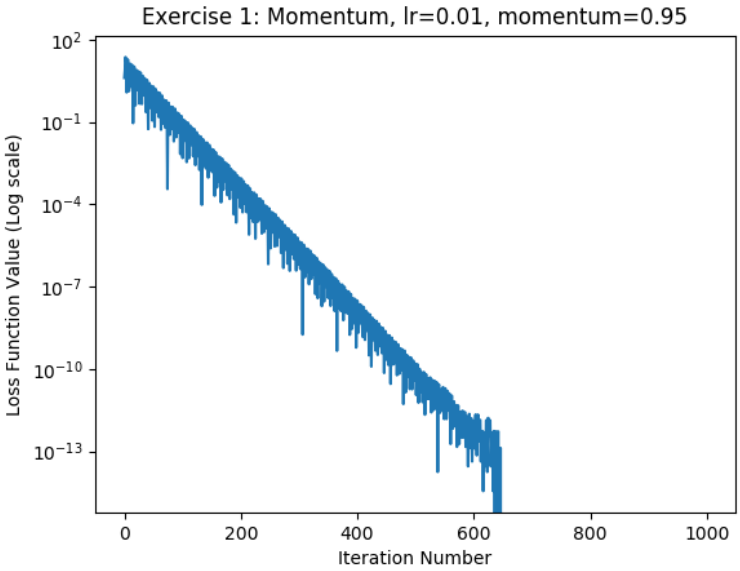
Learning rate	Momentum	Loss
0.1	0.8/0.9/0.95/0.99	larger than 100
0.01	0.8/0.9	below 10e-5
0.01	0.95	0
0.01	0.99	below 0.01
0.001	0.8/0.9/0.95/0.99	below 10e-5
0.0001	0.8/0.9	below 1
0.0001	0.95	below 0.1
0.0001	0.99	below 10e-5

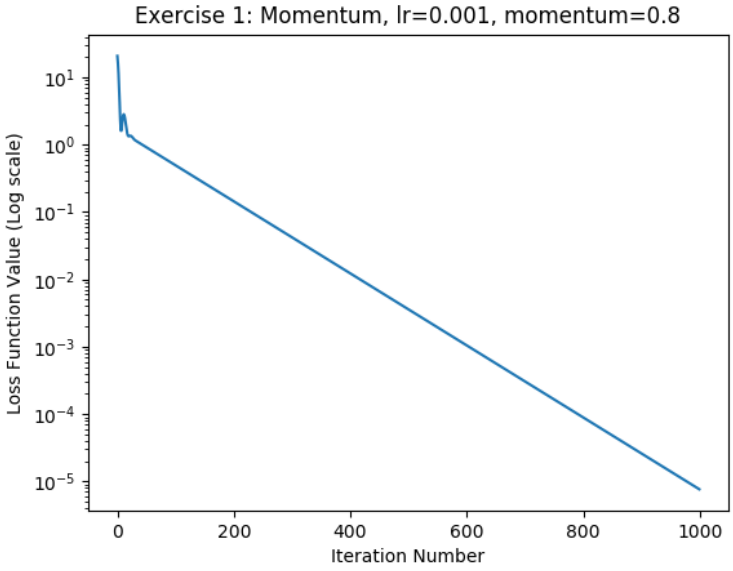
Momentum technique, instead of using only the gradient of the current step to guide the search, also accumulates the gradient of the past steps to determine the direction to go. It prevents from drastic changes when some outlier are present in current batch. We can see that the learning rate behavioral pattern is the same. The lr=0.1 is too big as it was in previous experiment. The perfect minimum was found for lr=0.01 with momentum=0.95. The average loss for lr=0.0001 was higher than for lr=0.001. We can see improvement compering to Gradient Descent technique.

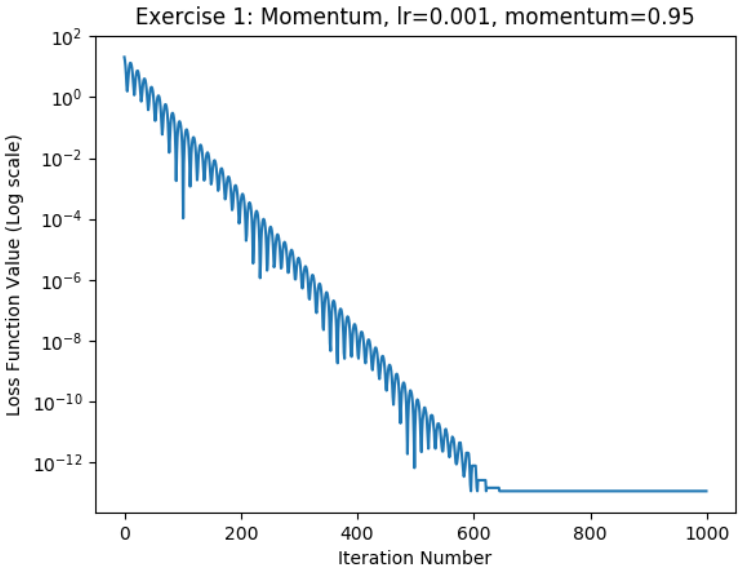
Some plots for exercise 1 - Momentum

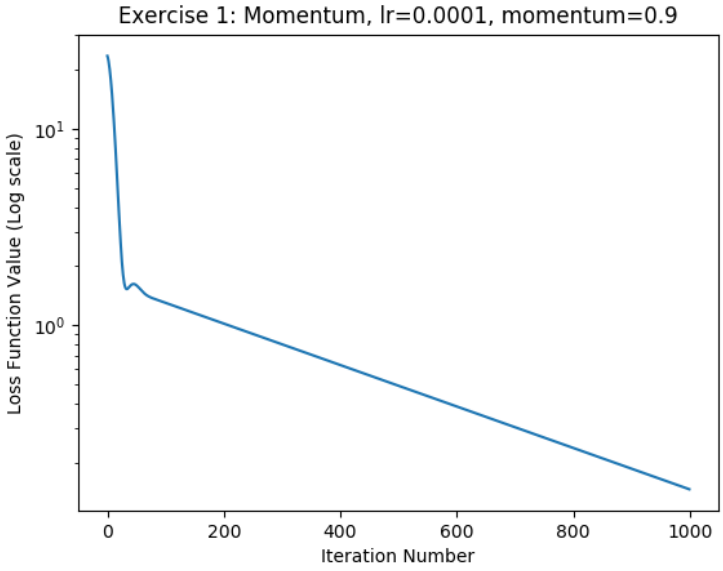


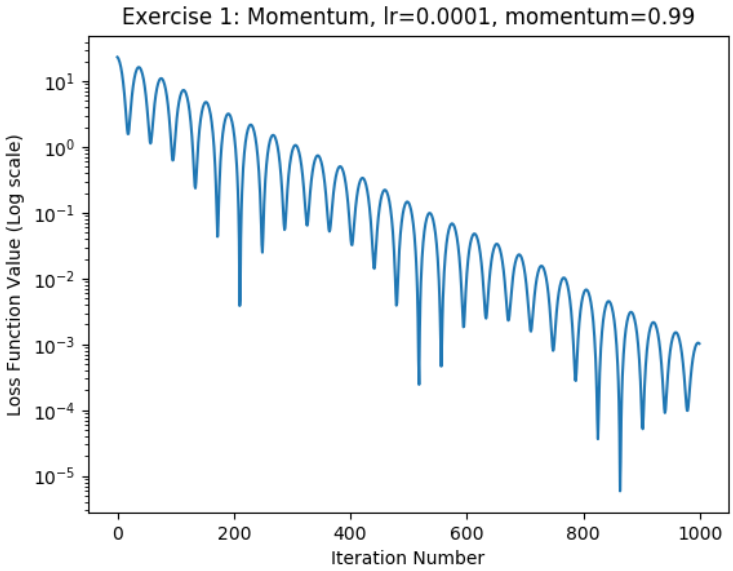



Exercise 1: Momentum, lr=0.01, momentum=0.95
Loss Function Value (Log scale)
10^{2}
10^{-1}
10^{-4}
10^{-7}
10^{-10}
10^{-13}
0
200
400
600
800
1000
Iteration Number


Exercise 1: Momentum, lr=0.001, momentum=0.8
Loss Function Value (Log scale)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
200
400
600
800
1000
Iteration Number


Exercise 1: Momentum, lr=0.001, momentum=0.95
Loss Function Value (Log scale)
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
0
200
400
600
800
1000
Iteration Number


Exercise 1: Momentum, lr=0.0001, momentum=0.9
Loss Function Value (Log scale)
10^{1}
10^{0}
0
200
400
600
800
1000
Iteration Number

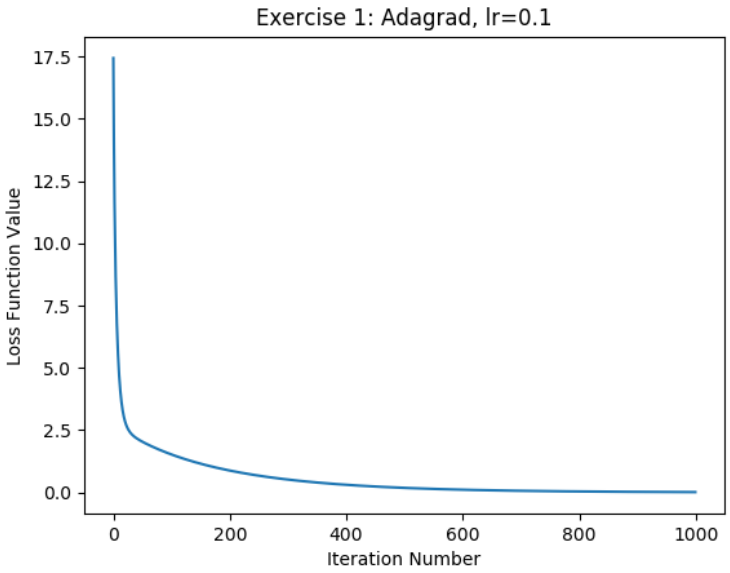

Exercise 1: Momentum, lr=0.0001, momentum=0.99
Loss Function Value (Log scale)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
200
400
600
800
1000
Iteration Number

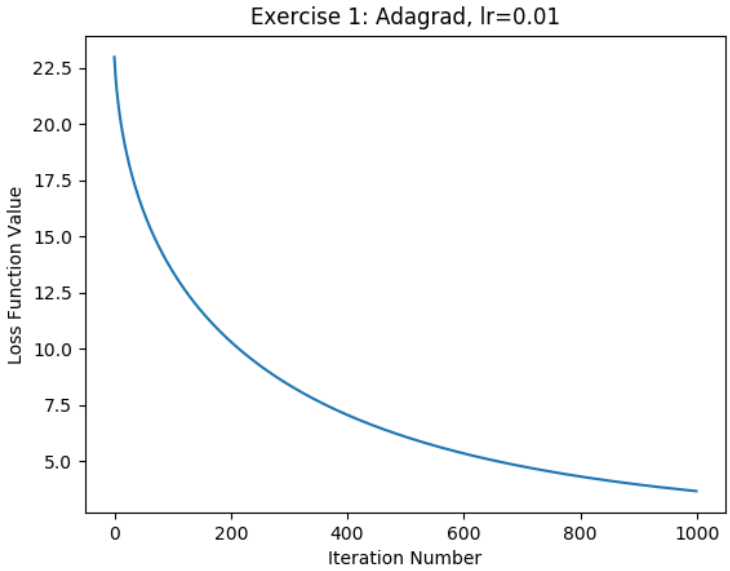
Optimizer: Adagrad

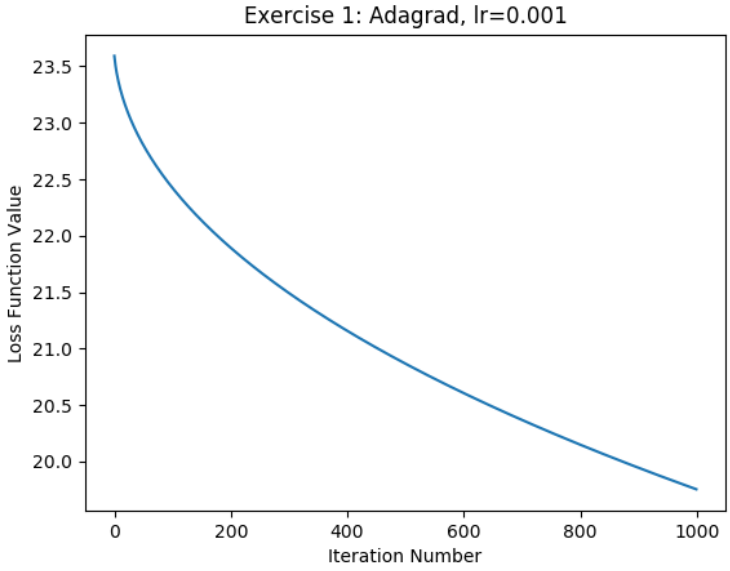
Learning rate	Loss
0.1	0.01
0.01	3.67
0.001	19.76
0.0001	23.25

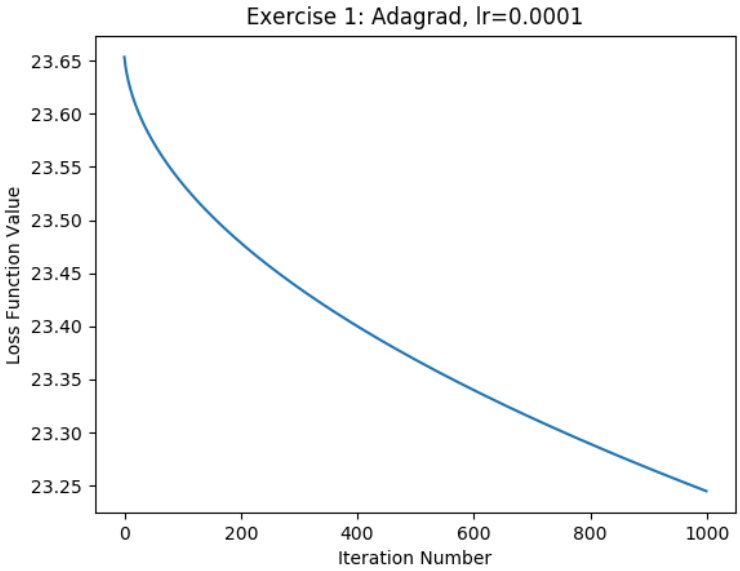
Adagrad technique allows an adaptive learning rate for each parameter. In the Tensorflow documentation we can read that this optimizer usually benefit from higher learning rates. The experiment confirmed it. We can see that when decreasing lr, loss is increasing. The best lr value is 0.1, and the worst is 0.0001.

Some plots for exercise 1 - Adagrad




Exercise 1: Adagrad, lr=0.01
Loss Function Value
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
200
400
600
800
1000
Iteration Number


Exercise 1: Adagrad, lr=0.001
Loss Function Value
23.5
23.0
22.5
22.0
21.5
21.0
20.5
20.0
0
200
400
600
800
1000
Iteration Number

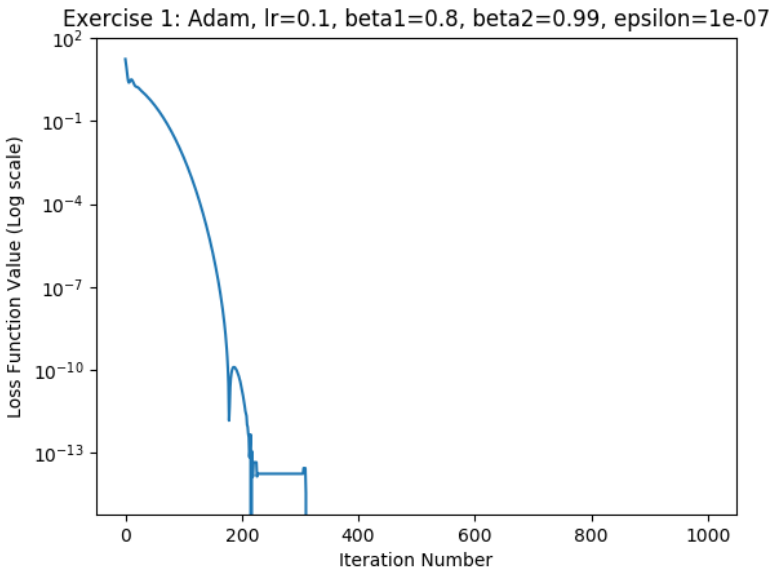

Exercise 1: Adagrad, lr=0.0001
Loss Function Value
23.65
23.60
23.55
23.50
23.45
23.40
23.35
23.30
23.25
0
200
400
600
800
1000
Iteration Number

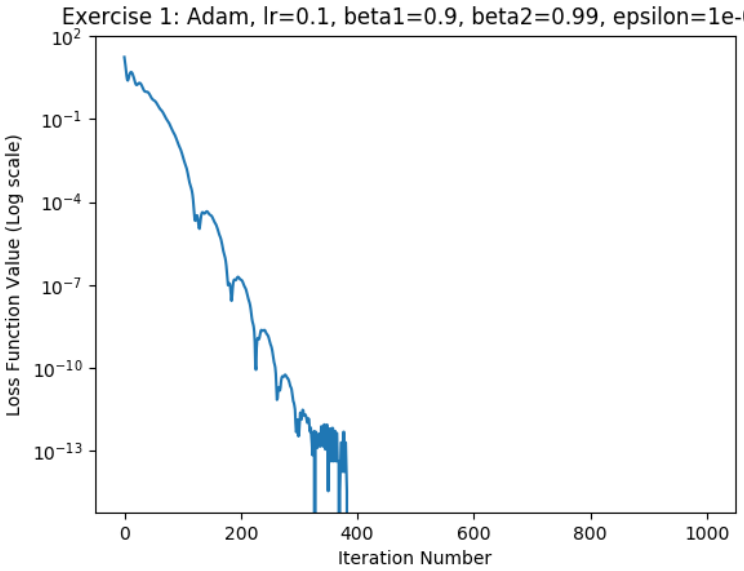
Optimizer: Adam

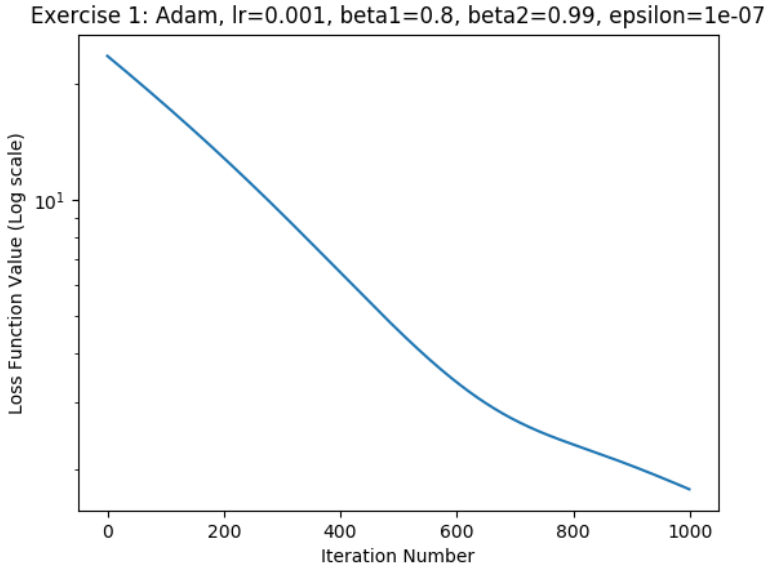
Learning rate	Beta1	Beta2	Loss
0.1	0.8/0.9/0.95	0.8/0.9/0.95	below 10e-5
0.1	0.8/0.9/0.95	0.99	0
0.1	0.99	0.8/0.9/0.95/0.99	below 10e-5
0.01	0.8/0.9/0.95	0.8/0.9/0.95/0.99	below 10e-5
0.01	0.99	0.8/0.9/0.95/0.99	around 10e-4
0.001	0.8/0.9/0.95/0.99	0.8/0.9/0.95/0.99	larger than 1
0.0001	0.8/0.9/0.95/0.99	0.8/0.9/0.95/0.99	larger than 10

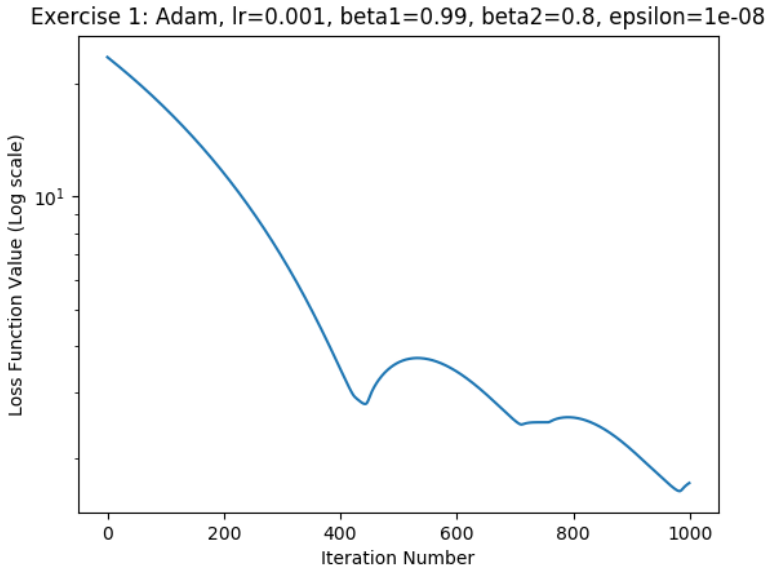
The Adam optimizer is usually most preferred technique. It is trying to combine the best properties of other optimizers by providing momentum updates for every parameter. The perfect minimum was found for lr=0.1, beta1=0.8/0.9/0.95 and beta2=0.99.

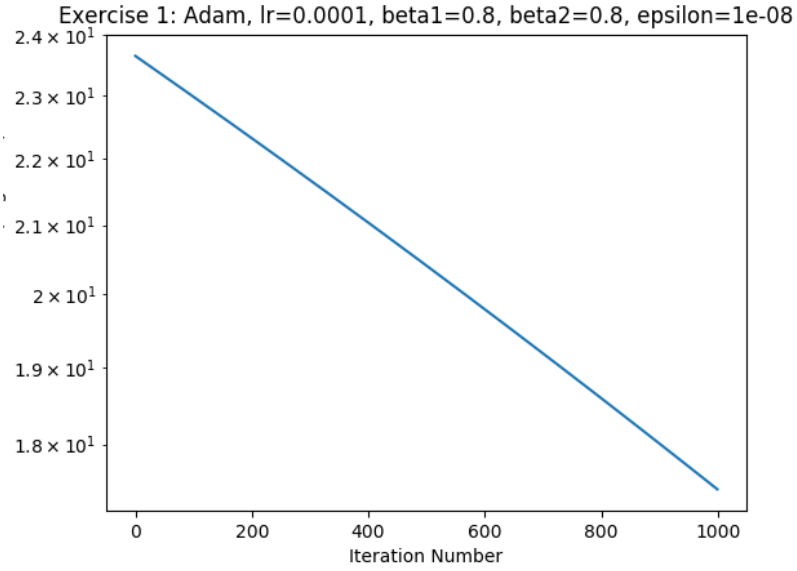
Some plots for exercise 1 - Adam




Exercise 1: Adam, lr=0.1, beta1=0.9, beta2=0.99, epsilon=1e-07
Loss Function Value (Log scale)
10^{2}
10^{-1}
10^{-4}
10^{-7}
10^{-10}
10^{-13}
0
200
400
600
800
1000
Iteration Number


Exercise 1: Adam, lr=0.001, beta1=0.8, beta2=0.99, epsilon=1e-07
Loss Function Value (Log scale)
10^{1}
0
200
400
600
800
1000
Iteration Number





Result

The solution was W=-1, b=1

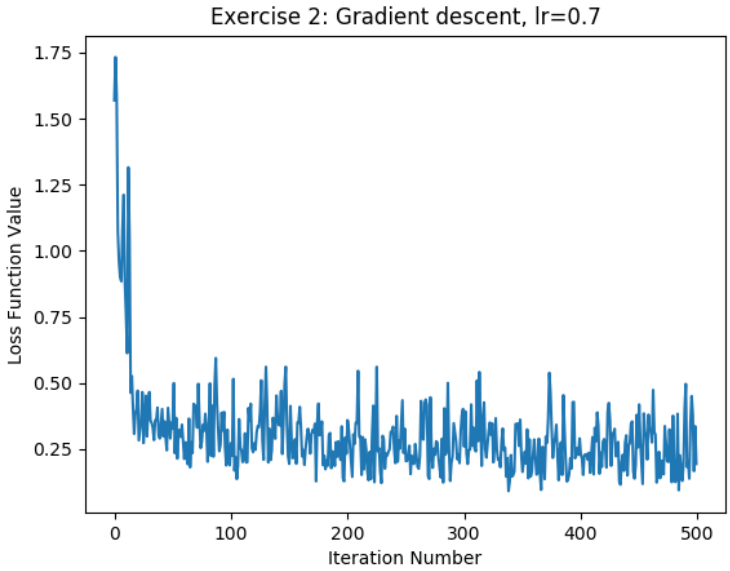
Exercise 2: Single Layer Network

The goal of this exercise was to find the values of the W and b matrices when x is the MNIST image and y is the number (0-9) it represents. In this case, we are dealing with a single-layer network. The same optimizers were tested. Only the best score for each is shown. Each training was only one epoch.

Optimizer: Gradient descent

Tested values for learning rate:

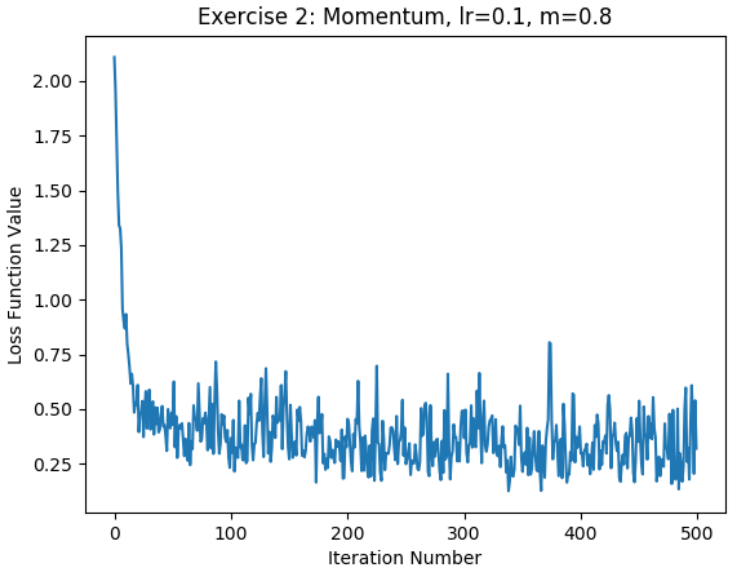
- 0.7
- 0.5
- 0.1
- 0.01
- 0.001
- 0.0001



The best result obtained for lr=0.7.
With the training loss equal 0.09 and test accuracy equal 91%.

Optimizer: Momentum

Learning rate	Momentum
0.7	0.8
0.5	0.9
0.1	0.95
0.01	0.99
0.001	
0.0001	

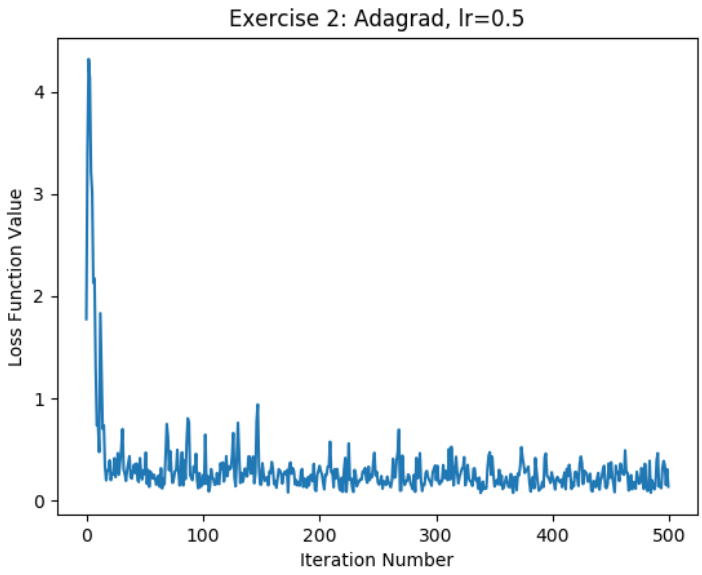


The best result obtained for lr=0.1 and momentum=0.8.
With the training loss equal 0.13 and test accuracy equal 92%.

Optimizer: Adagrad

Tested values for learning rate:

- 0.7
- 0.5
- 0.1
- 0.01
- 0.001
- 0.0001

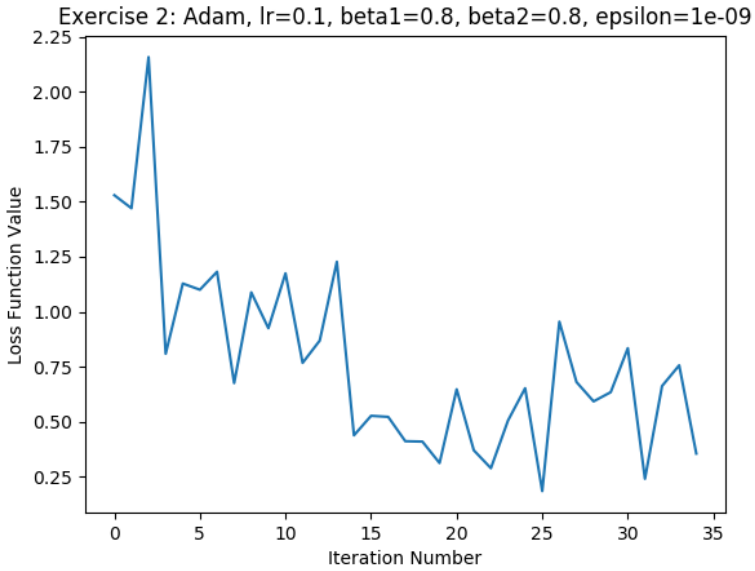


The best result obtained for lr=0.5.
With the training loss equal 0.07 and test accuracy equal 91.02%.

Optimizer: Adam

Tested values

Learning rate	Beta1	Beta2	Epsilon
0.1	0.8	0.8	1e-07
0.01	0.9	0.9	1e-08
0.001	0.95	0.95	1e-09
0.0001	0.99	0.99	



The best result obtained for lr=0.1, beta1=0.8, beta2=0.8 and epsilon=1e-09.
With the training loss equal 0.09 and test accuracy equal 91.86%.

Summary of exercise

In this exercise the best final accuracy was obtained with momentum optimizer and lr=0.5. However, compering to first exercise, this time we had test set, when we take a look only at the training loss the lowest value was obtained for Adagrad technique. Also we can notice that higher learning rates were working better for this exercise, it can be because problem was more complex this time and there were more parameters to fit (with still only one epoch of training). What we can say is that for different problem different optimizer and different learning rate can find the best solution.

Exercise 3: Multiple Layer Network

The goal of this exercise was to increase accuracy of provided model (96%).
The model was composed from:

- Convolutional layer: 32 features per each 5x5patch
- Convolutional layer: 64 features per each 5x5patch
- Densely connected Processes 64 7x7 images with 1024 neurons
- Dropout rate: 50%

Originally training consisted of 1 epoch with batch size equal 50 (1000 iterations per epoch). The Adam optimizer with learning rate equal 1e-4 was used.

The following changes were made to achieve better accuracy:

- Different batch sizes were tried (bs = [1, 2, 4, 8, 16, 250, 500, 1000]) and right number of iterations were chosen.
- Different learning rates were tried (lr = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7]

Experiments resulted with accuracy greater than 95% in one epoch

Learning rate	Batch size	Number of iterations	Test accuracy
1e-05	1	50000	96.2%
1e-05	2	25000	95.4%
1e-04	1	50000	97.6%
1e-04	2	25000	98.1%
1e-04	4	12500	98%
1e-04	8	6250	97.3%
1e-04	16	3125	97.2%
1e-03	1	50000	97.6%
1e-03	2	25000	97.1%
1e-03	4	12500	97.2%
1e-03	8	6250	98%
1e-03	16	3125	98.7%
1e-03	250	200	97.8%
1e-03	500	100	96.7%
1e-03	1000	50	95.6%
1e-02	1000	50	95.6%

When smaller learning rate the more backpropagation steps are needed (number of iterations or epochs). And by decreasing batch size, number of iterations per epoch increasing. It's visible that for smaller learning rate, the higher accuracy was achieved, when batch size was smaller.

The experiment was repeated on 40 epochs. In the next experiment learning rates 1e-06, 1e-05 and 1e-04 were chosen. As the expectation was that larger number of backpropagation steps (epochs) can help especially for smaller learning rate values. Batch size equal 8, 16, 32, 64, 128 and 256 was tried.

Experiments resulted with accuracy greater than 99% in 40 epochs

Learning rate	Batch size	Number of iterations	Test accuracy
1e-05	8	6250	99.1%
1e-05	16	3125	99.1%
1e-04	8	6250	99.3%
1e-04	16	3125	99.3%
1e-04	32	1562	99.2%
1e-04	64	781	99.2%

Figure 1: Accuracy plot for training network with lr=1e-04 and batch size=16. We can see that testing accuracy reached 99% shortly after 5th epoch.

Exercise 4: Multiple Layer Network - using multiple GPUs

The goal of this exercise was to compare training time when using more GPUs. Architecture stayed the same as in the previous exercise.
Batch size per each GPU was equal 16.
Learning rate was equal 1e-04 for Adam optimizer.
Code was written in Tensorflow library and Keras interface.
To run the code on specific number of GPUs script2launch.sh file was edited, by changing '–gres gpu:N' and '–cpus-per-task=40*N' lines and choosing the right N value.
To run the code on multiple GPUs, MirroredStrategy() function from tensorflow.distribute was used. This function was using all available GPUs (the number specified in script2launch.sh script).

Batch size given in model.fit() function was equal to 16*number_of_GPUs

Model was compiled with strategy.scope() function and later,

```
def make_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(5, 5),
                     activation='relu',
                     padding="same",
                     strides=(1, 1),
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2), padding="same"))
    model.add(Conv2D(64, (5, 5), activation='relu', padding="same", strides=(1, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2), padding="same"))
    model.add(Flatten())
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(lr=0.0001),
                  metrics=['accuracy'])

    return model
```

Figure 2: Model architecture definition in Keras interface.

compiled in this way model, was fitted as usually.

The results obtained:

Number of GPUs	Training time	Test loss	Test accuracy
1	610 seconds	0.03	99.12%
2	324 seconds	0.03	99.0%
4	156 seconds	0.03	98.91%

Using more GPUs significantly reduce training time without affecting performance of the model. By increasing number of GPUs twice we are able to train the model in half the time. By knowing how complex nowadays models can be, it's highly recommended to use multi-GPU technique.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              3212288
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                10250
=================================================================
```

Figure 3: Model summary.

```
strategy = tf.distribute.MirroredStrategy()
print("Number of accelerators: ", strategy.num_replicas_in_sync)


batch_size = 16 * strategy.num_replicas_in_sync

def make_model():...


with strategy.scope():
    model = make_model()

# print model layers
model.summary()

# TRAIN
print("TRAINING")
start_time = time.time()

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))

print("Training Time: %.3f seconds" % (time.time() - start_time))
```

Figure 4: Code listing with the most important changes. Strategy defined with MirroredStrategy() function, global batch size declaration, model creation and fitting.

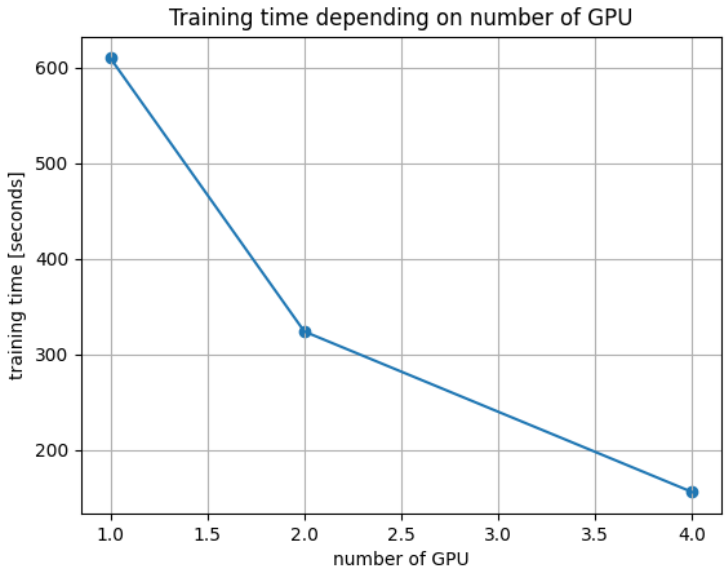


Figure 5: Training time dependency on number of GPUs.

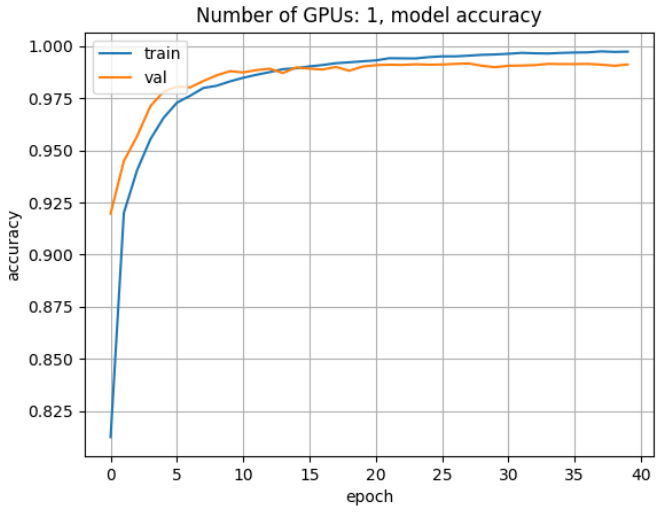


Figure 6: Accuracy plot during training on 1 GPU

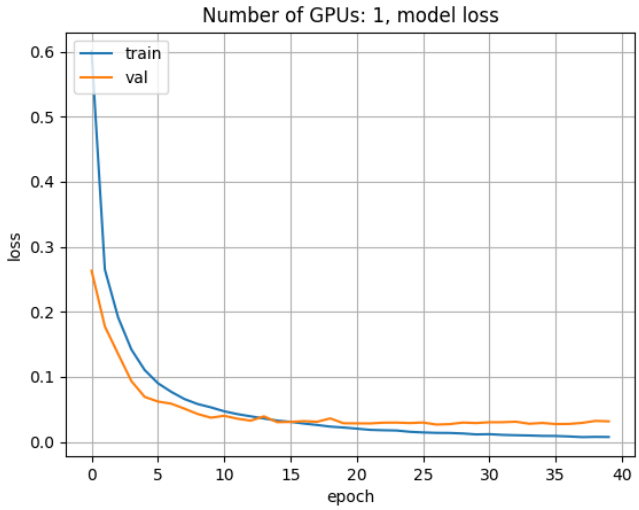


Figure 7: Loss plot during training on 1 GPU

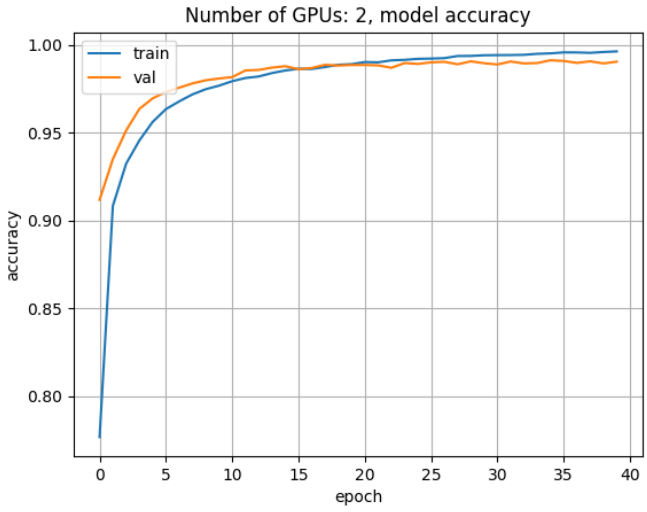


Figure 8: Accuracy plot during training on 2 GPU

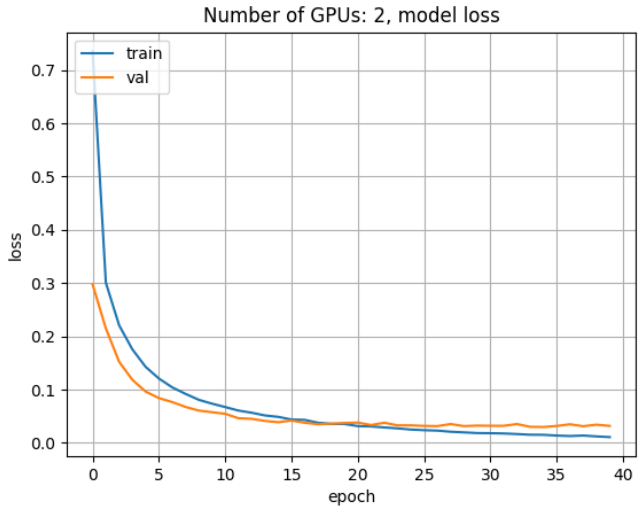


Figure 9: Loss plot during training on 2 GPU

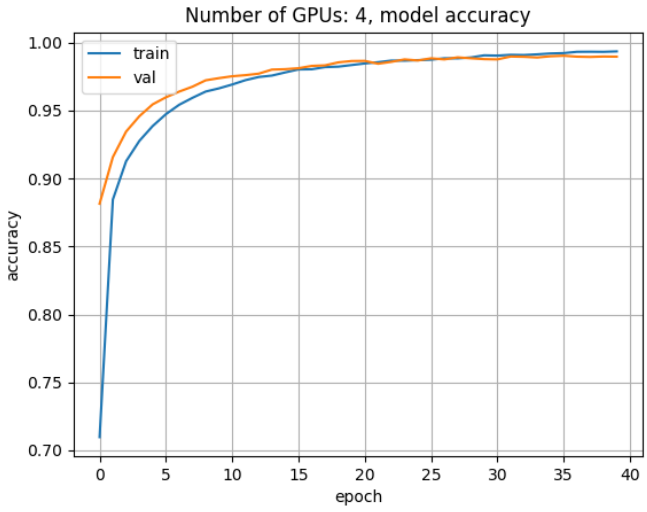


Figure 10: Accuracy plot during training on 4 GPU

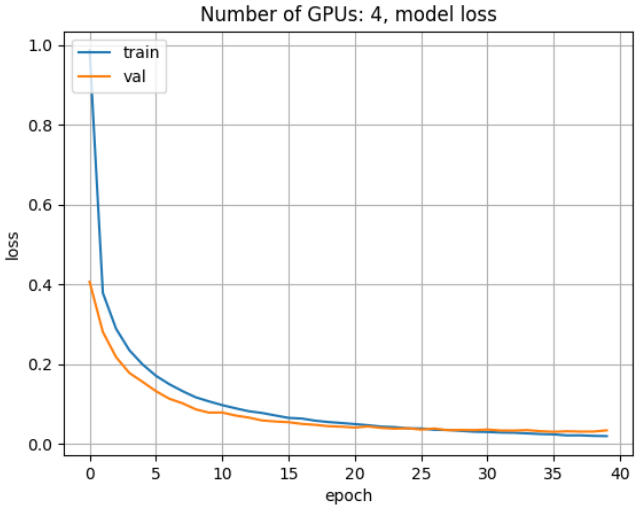


Figure 11: Loss plot during training on 4 GPU