Parallelizing Reinforcement Learning

Gabriel S.J. Spadoni

January 2024

1 Introduction

Reinforcement Learning is an important and often used Machine Learning algorithm as it can enable an agent to learn about a task or an environment only by knowing what it can do in it. However, the greater the environment, the longer the training can take. In this report we will focus on how we can implement Reinforcement Learning and running parallel training.

First, we will explain what is Reinforcement Learning and how does it work. We will then explain how we managed to parallelize it. Finally, we will look at the results.

2 Reinforcement Learning Overview

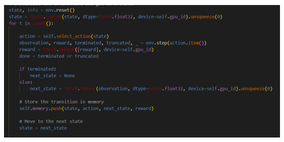
In this section, we will explain what this Reinforcement Learning and how we implemented it.

So the goal of Reinforcement Learning is to make an agent learn how to reach goal by giving it rewards when the goal is successfully reached. In this way, the agent will try to maximize the rewards it gets with the actions it does and thus reaching the goal. The agent only knows the state and the actions it can do.

We have used as the basis for the code the tutorial on Reinforcement Learning of PyTorch [1]

We are using Deep Q Learning (DQN) which is a type of Reinforcement Learning. The approach used has two models that are learning, we have the policy that is our final model and the target model which is used to keep the old of the policy net.

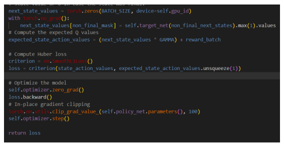
We will now describe one epoch. So during one epoch the agent will play in the environment until it either reaches the goal or fails. Failure is determined here by reaching a maximum number of actions. For each iteration in the environment, we keep in memory the state, the next state, the chosen action and the reward (see in Fig. 1). Using this, if there is enough data in the memory, we optimize the policy net. That is we optimize it for the state action values using the value of the next state times the Gamma plus the reward. The Gamma is the discount factor, it is a hyper parameter that it used to enforce



```
state, info = env.reset()
state = torch.tensor(state, dtype=torch.float32, device=self.gpu_id).unsqueeze(0)
for t in count():

    action = self.select_action(state)
    observation, reward, terminated, truncated, _ = env.step(action.item())
    reward = torch.tensor([reward], device=self.gpu_id)
    done = terminated or truncated

    if terminated:
        next_state = None
    else:
        next_state = torch.tensor(observation, dtype=torch.float32, device=self.gpu_id).unsqueeze(0)

    # Store the transition in memory
    self.memory.push(state, action, next_state, reward)

    # Move to the next state
    state = next_state
```

Figure 1: One Iteration for the training

good start as early actions will be taken more into account. We are also using the Adam as an optimizer and Huber as Loss function, we are using Huber loss function [3] as it is less sensitive to outliers (See in Fig. 2).



```
next_state_values = torch.zeros(BATCH_SIZE, device=self.gpu_id)
with torch.no_grad():
    next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1).values
# Compute the expected Q values
expected_state_action_values = (next_state_values * GAMMA) + reward_batch

# Compute Huber loss
criterion = nn.SmoothL1Loss()
loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

# Optimize the model
self.optimizer.zero_grad()
loss.backward()
# In-place gradient clipping
torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)
self.optimizer.step()

return loss
```

Figure 2: Optimization Function

Following the optimization, we update target network using the state dict of the policy network.

3 Parallelization Approach

We will now describe the approach used to parallelize the training of Reinforcement Learning.

We used `pytorch` to parallelize the learning, indeed pytorch allows us to chose on which device a model is. In device, we can specify if we have two gpus which one of them we want to use, for example:

```
policy_net = DQN(n_observations, n_actions).to(device)
```

We used as a basis for parallelizing the code of the tutorial on dpp of PyTorch [4], which we have adapted to Reinforcement Learning.

For parallelizing, we use the libraries `multiprocessing`, `nn.parallel`, `distributed` and `utils.data.distributed` from pytorch. These different libraries allows us to spawn a number of process groups equal to the number of devices we want to use. For this we use the function `init_process_group` from the `distributed` library. It takes as parameters backend which is the backend used to communicate between groups, here we use ncll which the nvidia backend library for communicated between gpus. There is also rank and world size which are respectively the rank of the current process and the number of processes. So the `init_process_group` will prepare one process at a time.

In order to have multiple processes, we use from the `multiprocessing` library the function `spawn`, that launch a function of a number of processes, it will also pass to the function the index of the process which will the rank for `init_process_group`. So if we have two devices, it will spawn two process and give them rank 0 and 1, so that one process will use device 0 and the other device 1.

After this, we create the trainer object and we pass it the rank, so that it correctly sends the model and the computation on the wanted device. We also use the function `DistributedDataParallel` from the library `nn.parallel` to synchronize the gradients of each model with all the replicas:

`self.target_net = DDP(target_net, device_ids=[gpu_id])`

and `self.policy_net = DDP(policy_net, device_ids=[gpu_id])`.

This is important such that the learning made in one process is kept for the final model. Here `gpu_id` corresponds to `rank`. After the training is finished for each process, we call the function `destroy_process_group` from the library `distributed`, this way each process is destroyed and we come back into a sequential execution.

4 Results

In this section, we will show the results and compare between a standard execution and a parallel execution the time of running and efficacy of learning. We will use the time for a full run, the loss value and also we will check if the agent is able to successfully complete the environment we trained on. In order to compare, we have run both version on the hpc, for a different number of iterations each. For the parallel, we used two gpus. For running it with different number of iterations we created a bash file, that launches the program for each different set up. So we launched with 100, 200, 500 and 1000 iterations.

Training with only one gpu took:

- For 100 iterations: 6.37 seconds
- For 200 iterations: 19.54 seconds
- For 500 iterations: 108.31 seconds
- For 1000 iterations: 735.39 seconds.

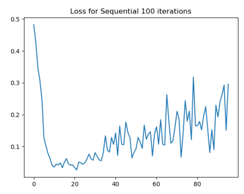


Figure 3: Loss for 100 iterations with one GPU

Here we see that loss function for 100 and 200 indeed in 100 we see that the agent is quickly learning and then it stops learning. We have the same for 200 and 500 iterations.

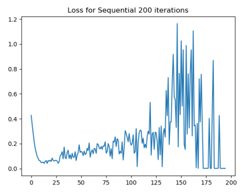


Figure 4: Loss for 200 iterations with one GPU

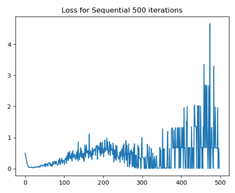


Figure 5: Loss for 500 iterations with one GPU

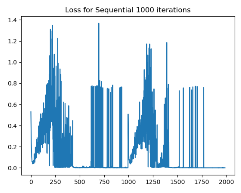
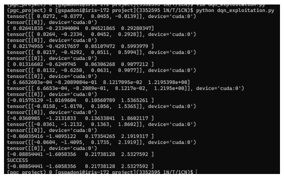


Figure 6: Loss for 1000 iterations with one GPU

However regarding exploitation, the best performing model is the one with 200 iterations, indeed in Fig.7:



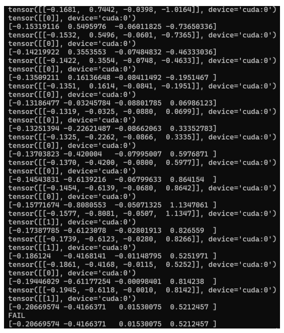
```
(pgc_project) 0 [gspadoni@iris-172 project](3352595 1N/T/1CN)$ vim dqn_exploitation.py
(pgc_project) 0 [gspadoni@iris-172 project](3352595 1N/T/1CN)$ python dqn_exploitation.py
tensor([[ 0.0272, -0.0377,  0.0455, -0.0139]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.02641835 -0.23344004  0.04521865  0.29280347]
tensor([[ 0.0264, -0.2334,  0.0452,  0.2928]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.02174955 -0.42917657  0.05107472  0.5993979 ]
tensor([[ 0.0217, -0.4292,  0.0511,  0.5994]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.01316602 -0.6249745   0.06306268  0.9077212 ]
tensor([[ 0.0132, -0.6250,  0.0631,  0.9077]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 6.6652603e-04 -8.2089084e-01  8.1217095e-02  1.2195398e+00]
tensor([[ 6.6653e-04, -8.2089e-01,  8.1217e-02,  1.2195e+00]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.01575129 -1.0169604   0.10560789  1.5365261 ]
tensor([[-0.0158, -1.0170,  0.1056,  1.5365]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.0360905  -1.2131833   0.13633841  1.8602117 ]
tensor([[-0.0361, -1.2132,  0.1363,  1.8602]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.06035416 -1.4095122   0.17354265  2.1919317 ]
tensor([[-0.0604, -1.4095,  0.1735,  2.1919]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.08854441 -1.6058356   0.21738128  2.5327592 ]
SUCCESS
[-0.08854441 -1.6058356   0.21738128  2.5327592 ]
(pgc_project) 0 [gspadoni@iris-172 project](3352595 1N/T/1CN)$
```

Figure 7: Exploitation for the model with 200 iterations

And the one that performs the worst is the one with 1000 iterations, indeed in Fig. 8:

Here we have the plot for the loss functions of the 100 iterations parallel Reinforcement Learning, see in Fig. 9:

This loss curve is good as we see it is quickly going down. However, the way



```
tensor([[-0.1681,  0.7442, -0.0398, -1.0164]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.15319116  0.5495976  -0.06011825 -0.73650336]
tensor([[-0.1532,  0.5496, -0.0601, -0.7365]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.14219922  0.3553553  -0.07484832 -0.46333036]
tensor([[-0.1422,  0.3554, -0.0748, -0.4633]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.13509211  0.16136648 -0.08411492 -0.1951467 ]
tensor([[-0.1351,  0.1614, -0.0841, -0.1951]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.13186477 -0.03245784 -0.08801785  0.06986123]
tensor([[-0.1319, -0.0325, -0.0880,  0.0699]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.13251394 -0.22621487 -0.08662063  0.33352783]
tensor([[-0.1325, -0.2262, -0.0866,  0.3335]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.13703823 -0.420004   -0.07995007  0.5976871 ]
tensor([[-0.1370, -0.4200, -0.0800,  0.5977]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.14543831 -0.6139216  -0.06799633  0.864154  ]
tensor([[-0.1454, -0.6139, -0.0680,  0.8642]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.15771674 -0.8080553  -0.05071325  1.1347061 ]
tensor([[-0.1577, -0.8081, -0.0507,  1.1347]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.17387785 -0.6123078  -0.02801913  0.826559  ]
tensor([[-0.1739, -0.6123, -0.0280,  0.8266]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.186124   -0.4168141  -0.01148795  0.5251971 ]
tensor([[-0.1861, -0.4168, -0.0115,  0.5252]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.19446029 -0.61177254 -0.00098401  0.814238  ]
tensor([[-0.1945, -0.6118, -0.0010,  0.8142]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.20669574 -0.4166371   0.01530075  0.5212457 ]
FAIL
[-0.20669574 -0.4166371   0.01530075  0.5212457 ]
```

Figure 8: Exploitation for the model with 1000 iterations

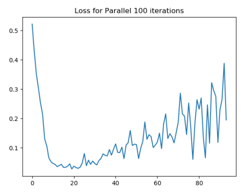
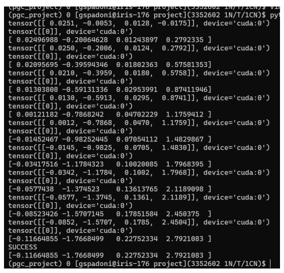


Figure 9: Loss values for 100 iterations in Parallel

we have set up the learning makes that we have a curve for only one process but the model is for two, which gives the double of total iterations. This explains the good performing model (See in Fig. 10):

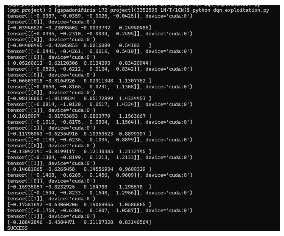


```
(pgc_project) 0 [gspadoni@iris-176 project](3352602 1N/T/1CN)$ vi
(pgc_project) 0 [gspadoni@iris-176 project](3352602 1N/T/1CN)$ py
tensor([[ 0.0251, -0.0053,  0.0128, -0.0175]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.02496988 -0.20064628  0.01243897  0.2792335 ]
tensor([[ 0.0250, -0.2006,  0.0124,  0.2792]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.02095695 -0.39594346  0.01802363  0.57581353]
tensor([[ 0.0210, -0.3959,  0.0180,  0.5758]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.01303808 -0.59131336  0.02953991  0.87411946]
tensor([[ 0.0130, -0.5913,  0.0295,  0.8741]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.00121182 -0.7868242   0.04702229  1.1759412 ]
tensor([[ 0.0012, -0.7868,  0.0470,  1.1759]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.01452467 -0.98252445  0.07054112  1.4829867 ]
tensor([[-0.0145, -0.9825,  0.0705,  1.4830]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.03417516 -1.1784323   0.10020085  1.7968395 ]
tensor([[-0.0342, -1.1784,  0.1002,  1.7968]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.0577438  -1.374523    0.13613765  2.1189098 ]
tensor([[-0.0577, -1.3745,  0.1361,  2.1189]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.08523426 -1.5707145   0.17851584  2.450375  ]
tensor([[-0.0852, -1.5707,  0.1785,  2.4504]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.11664855 -1.7668499   0.22752334  2.7921083 ]
SUCCESS
[-0.11664855 -1.7668499   0.22752334  2.7921083 ]
(pgc_project) 0 [gspadoni@iris-176 project](3352602 1N/T/1CN)$ |
```

Figure 10: Exploitation for 100 parallel iterations

Compared to the exploitation for 100 sequential iterations which has more steps to do before success, see in Fig. 11:

Here we have the curve of the loss for the parallel 200 iterations training (Fig. 13):



```
(pgc_project) 0 [gspadoni@iris-172 project](3352595 1N/T/1CN)$ python dqn_exploitation.py
tensor([[-0.0387, -0.0359, -0.0025, -0.0425]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.03946525 -0.23098502 -0.0033792   0.24940486]
tensor([[-0.0395, -0.2310, -0.0034,  0.2494]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.04408495 -0.42605853  0.0016089   0.54102   ]
tensor([[-0.0441, -0.4261,  0.0016,  0.5410]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.05260612 -0.62120306  0.0124293   0.83420944]
tensor([[-0.0526, -0.6212,  0.0124,  0.8342]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.06503018 -0.8164926   0.02911348  1.1307752 ]
tensor([[-0.0650, -0.8165,  0.0291,  1.1308]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.08136003 -1.0119834   0.05172899  1.4324453 ]
tensor([[-0.0814, -1.0120,  0.0517,  1.4324]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.1015997  -0.81753653  0.0803779   1.1563667 ]
tensor([[-0.1016, -0.8175,  0.0804,  1.1564]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.11795043 -0.62354916  0.10350523  0.8899307 ]
tensor([[-0.1180, -0.6235,  0.1035,  0.8899]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.13042141 -0.8199117   0.12130385  1.2132745 ]
tensor([[-0.1304, -0.8199,  0.1213,  1.2133]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.14681965 -0.6265458   0.14556934  0.9609329 ]
tensor([[-0.1468, -0.6265,  0.1456,  0.9609]], device='cuda:0')
tensor([[0]], device='cuda:0')
[-0.15935057 -0.8232925   0.164788    1.295578  ]
tensor([[-0.1594, -0.8233,  0.1648,  1.2956]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.17581642 -0.63060206  0.19069955  1.0586865 ]
tensor([[-0.1758, -0.6306,  0.1907,  1.0587]], device='cuda:0')
tensor([[1]], device='cuda:0')
[-0.18842846 -0.4384471   0.21187328  0.83140564]
SUCCESS
```

Figure 11: Exploitation for 100 iterations

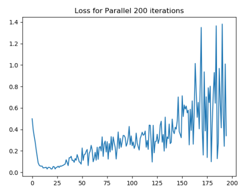
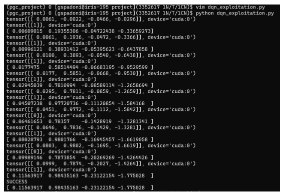


Figure 12: Loss values for 200 parallel iterations



```
(pgc_project) 0 [gspadoni@iris-195 project](3352617 1N/T/1CN)$ vim dqn_exploitation.py
(pgc_project) 0 [gspadoni@iris-195 project](3352617 1N/T/1CN)$ python dqn_exploitation.py
tensor([[ 0.0061, -0.0022, -0.0466, -0.0296]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.00609015  0.19355306 -0.04722438 -0.33659273]
tensor([[ 0.0061,  0.1936, -0.0472, -0.3366]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.00996121  0.38931412 -0.05395623 -0.6437858 ]
tensor([[ 0.0100,  0.3893, -0.0540, -0.6438]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.0177475   0.58514494 -0.06683195 -0.9529599 ]
tensor([[ 0.0177,  0.5851, -0.0668, -0.9530]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.02945039  0.7810994  -0.08589114 -1.2658694 ]
tensor([[ 0.0295,  0.7811, -0.0859, -1.2659]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.04507238  0.97720736 -0.11120854 -1.584168  ]
tensor([[ 0.0451,  0.9772, -0.1112, -1.5842]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.06461653  0.78357    -0.1428919  -1.3281341 ]
tensor([[ 0.0646,  0.7836, -0.1429, -1.3281]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.08028793  0.9801766  -0.16945457 -1.6619058 ]
tensor([[ 0.0803,  0.9802, -0.1695, -1.6619]], device='cuda:0')
tensor([[0]], device='cuda:0')
[ 0.09989146  0.7873854  -0.20269269 -1.4264426 ]
tensor([[ 0.0999,  0.7874, -0.2027, -1.4264]], device='cuda:0')
tensor([[1]], device='cuda:0')
[ 0.11563917  0.98435163 -0.23122154 -1.775028  ]
SUCCESS
[ 0.11563917  0.98435163 -0.23122154 -1.775028  ]
```

Figure 13: Exploitation for 200 parallel iterations

And the result of the exploitation for 200 iterations in parallel (See in Fig. 13):

As we see the number of steps used to atteign a successful state is really low.

We unfortunately don't have the run time for the parallel training.

5 Conclusion

As a conclusion, we have implemented parallel training for Reinforcement Learning. It seems to give correct results and help for the training of Reinforcement Learning.

For improving this project, we can apply parallel reinforcement learning on more complex environments as this environment is simple enough that standard RL models will learn it quickly. Indeed, we used a simple environment to help us having quick training times and to speed up the testing. The aim of this project was to get familiar with PyTorch especially for parallelization and to apply to Reinforcement Learning0.

References

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html https://github.com/pytorch/tutorials/blob/main/intermediate_source/reinforcement_q_learning.py

[2] https://en.wikipedia.org/wiki/Q-learning

[3] https://en.wikipedia.org/wiki/Huber_loss

[4] https://github.com/pytorch/examples/blob/main/distributed/ddp-tutorial-series/multigpu.py