**SARSA algorithm report**

Ericka Bermudez. 202230093

The following report describes the algorithm “SARSA”. SARSA stands for “State-Action-Reward-State-Action”, the things that should be done before performing an update. It was invented by Rummery and Niranjan in 1994. SARSA is one of the older reinforcement learning algorithms, but still incorporates many of the important ideas of value-based methods and is a good example of them, even if it is not widely used due to its inefficiency during training.

Value-based algorithms evaluate state-action pairs by learning one of the value functions and use these evaluations to select actions. They are typically more sample-efficient than policy-based algorithms because they have lower variance and make better use of data gathered from the environment. However, there are no guarantees that these algorithms will converge to an optimum.

The SARSA algorithm consists of two core ideas. (1) Technique for learning the Q-function known as temporal difference (TD) learning. (2) Method for generating actions using the Q-function. We will see the code and interpret what we are asking our computer to train our SARSA agent.

{

"sarsa\_epsilon\_greedy\_cartpole": {

"agent": [{

"name": "SARSA",

"algorithm": {

"name": "SARSA",

"action\_pdtype": "Argmax",

"action\_policy": "epsilon\_greedy",

"explore\_var\_spec": {

"name": "linear\_decay",

"start\_val": 1.0,

"end\_val": 0.05,

"start\_step": 0,

"end\_step": 10000

},

"gamma": 0.99,

"training\_frequency": 5

},

The first line indicates the name of the file we are working on, in this case sarsa\_epsilon\_greedy\_cartpole. The agent is a list to allow for multiple agents, but in this scenario there is only one agent.

The epsilon greedy determines how the SARSA agent will act in an environment. The epsilon greedy policy mitigates the effect in which an agent may not explore the entire state-action space sufficiently, because the agent never experiences many (s, a) pairs as if it selects always the same action in state s.

The epsilon helps us to decide if the agent will act randomly or greedily. If epsilon is more than a random number, then it will act randomly. by calling the random action function. Else, if acting greedily, a Q-value will be estimated fr each action in the state.

Linear decay is the strategy we chose to update epsilon (though in the case of a different policy it could be another variable) in the explore\_var\_spec. They control the trade off between exploration and exploitation. It is initialized to 1.0 and annealed to 0.05 (end\_val, the final value for epsilon) between steps 0 and 10,000.

Gamma is the discount factor or rate. It controls the problem horizon of an agent. It is a crucial parameter and determines how much an agent weights rewards received in the future compared to the current time step. To choose this number, we should think about how delayed rewards typically are in an environment. 0.99 is the default value a lot of times.

The training frequency. In the next lines of code, we will see that we have selected OnPolicyBatchReplay memory. Then, our training is batch-wise. and we use the training\_frequency variable to establish a batch size. In the original settings, the default batch size was 32 but we changed it to 5. This batch size means that the network will be trained every 5 steps.

"memory": {

"name": "OnPolicyBatchReplay"

},

Here is where we select the type of memory. OnPolicyReplay allows us to use Batch Memory Update. Batch Memory allows us to add directly the current experience to the main memory containers.

"net": {

"type": "MLPNet",

"hid\_layers": [64],

"hid\_layers\_activation": "selu",

"clip\_grad\_val": 0.5,

"loss\_spec": {

"name": "MSELoss"

},

"optim\_spec": {

"name": "RMSprop",

"lr": 0.01

},

"lr\_scheduler\_spec": null

}

}],

This part of the code refers to the network architecture. It is a Multilayer Precepton. This function calculates the output values of the neural network for the given data. It will be used to learn the Q function. It has 64 hidden layers and uses SeLU activation function. SeLU stands for Scaled Exponential Linear Unit. SeLU self normalizes the neural network.

optim\_spec is the configuration of the optimizer. For our scenario, we chose RMSprop, with a learning rate of 0.01.

clip\_grad\_val is the maximum norm of the gradient, it helps us to avoid exploding loss. In this case, it is 0.5, but a good number could be anywhere between 0.5 and 1.0 as this helps to avoid large parameter updates. The learning rate scheduler spec is not used.

"env": [{

"name": "CartPole-v0",

"max\_t": null,

"max\_frame": 100000

}],

These lines allow us to configure the environment (or environments, we can extend the list). As with REINFORCE, we are using the CartPole example, from OpenAi’s Gym. We don’t define the optional maximum step per episode, but the total time steps (frames) is configured. The training consists of 100,000 time steps, defined by max\_frame.

"body": {

"product": "outer",

"num": 1

},

Body specifies how multiagent connects to multi environments. We are using the default value 1, as it is a single-agent single-environment use case.

"meta": {

"distributed": false,

"eval\_frequency": 2000,

"max\_trial": 1,

"max\_session": 4

},

In meta we find information about how the lab is run. The meta distributed as false means that we won’t use async parallelization. We declare that there will be 4 episodes (sessions), 1 trial, and that the agent will be evaluated every 2,000 time steps. During evaluation, epsilon will be set to its final value (set previously on end\_val). After the episodes are run, the mean total rewards are calculated, as reported in the graphs in the following section of this report.

"search": {

"agent": [{

"net": {

"optim\_spec": {

"lr\_\_grid\_search": [0.0005, 0.001, 0.001, 0.005, 0.01, 0.05, 0.1]

}

}

}]

}

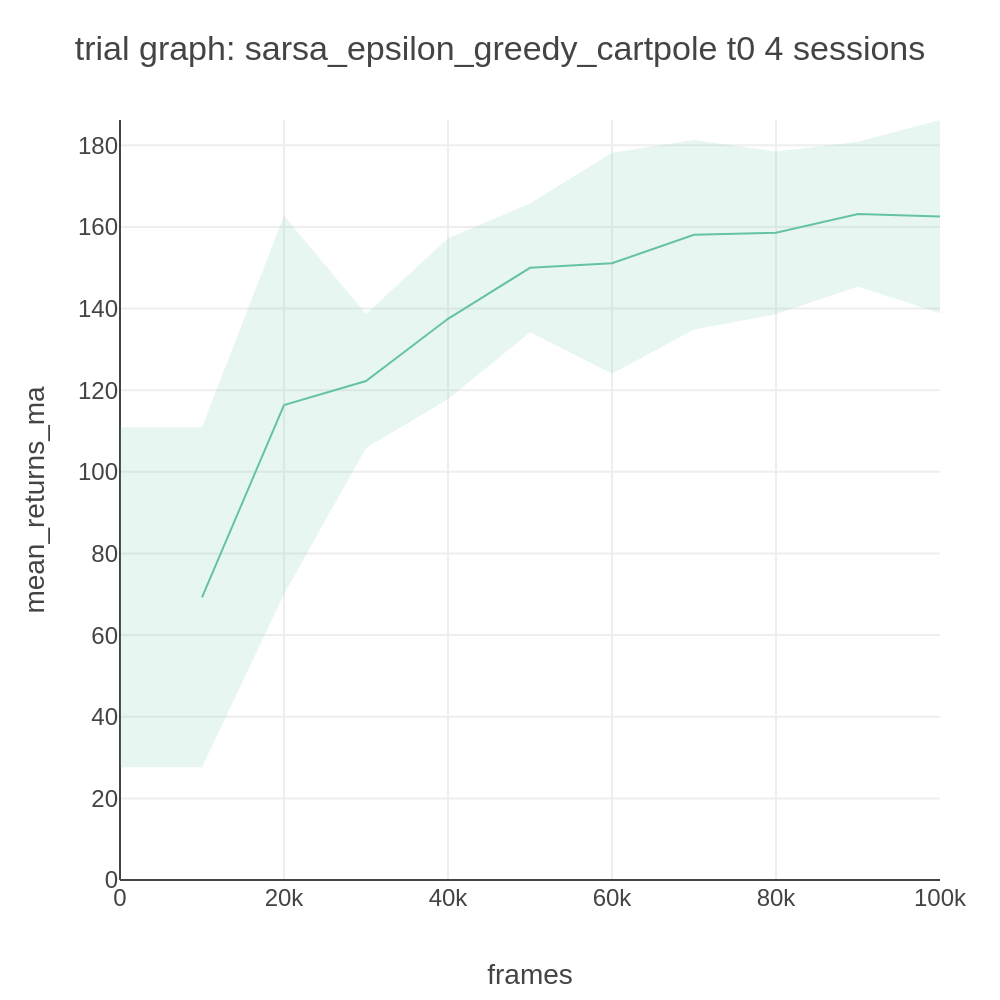
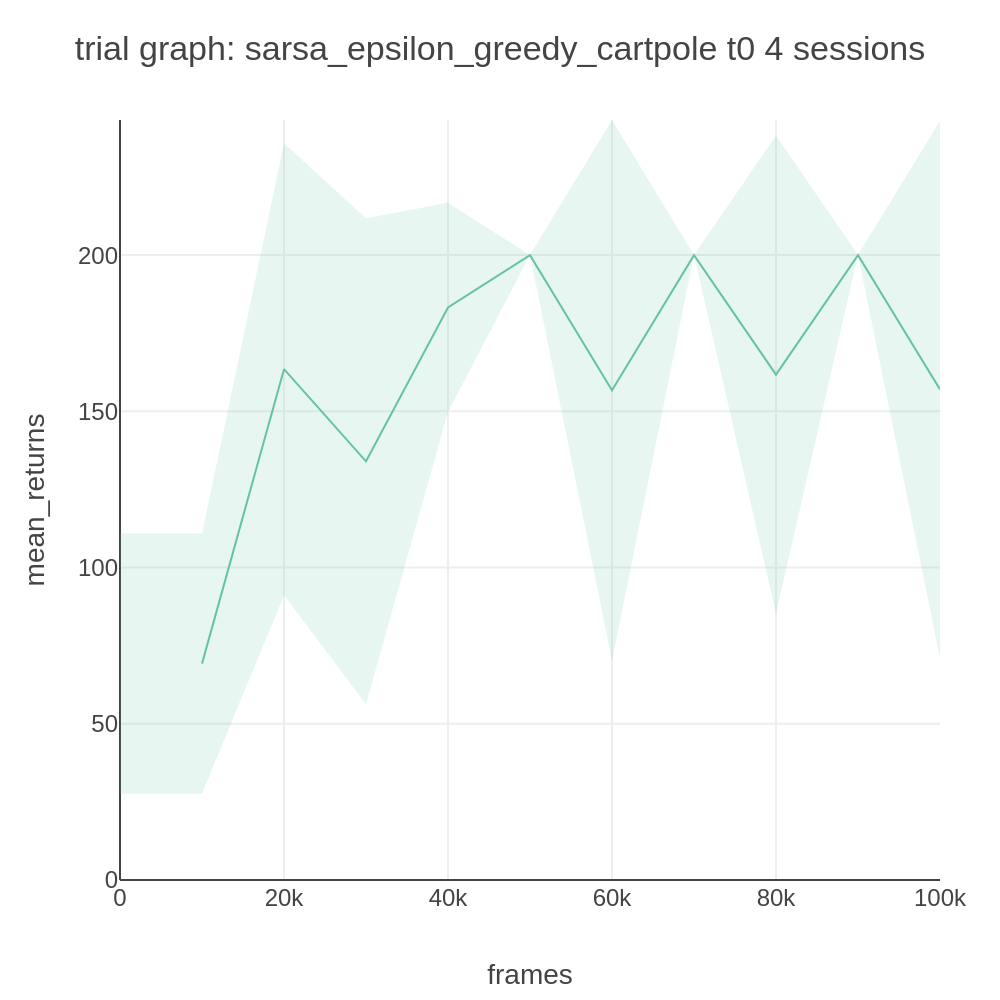
Search are the hyperparameters to search over, and the methods used to sample them. Here we are searching through the learning rate value.

We can run our first experiment with:

conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/sarsa/sarsa\_cartpole.json sarsa\_epsilon\_greedy\_cartpole train

First, we will activate our virtual environment that already has all the packages we will need. Next, we will ask python to train based on our settings. In the next pages, we will discuss the results.

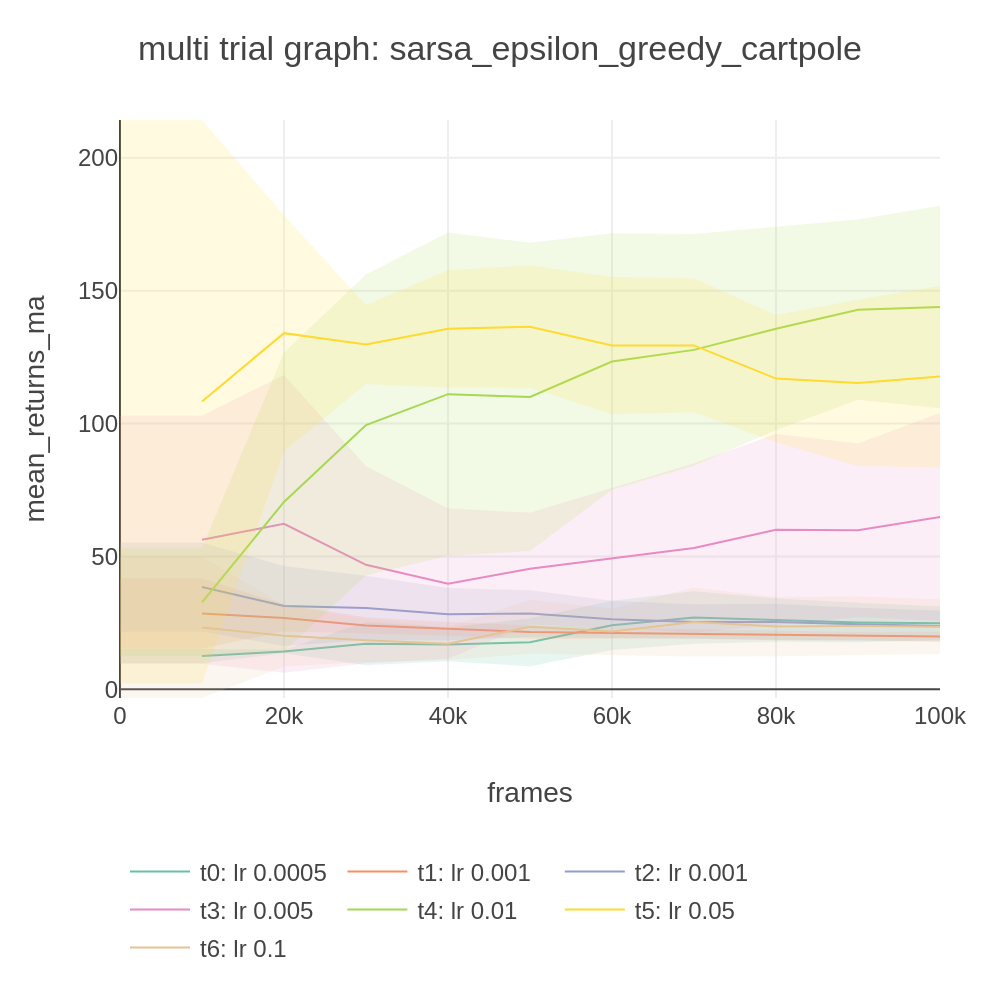
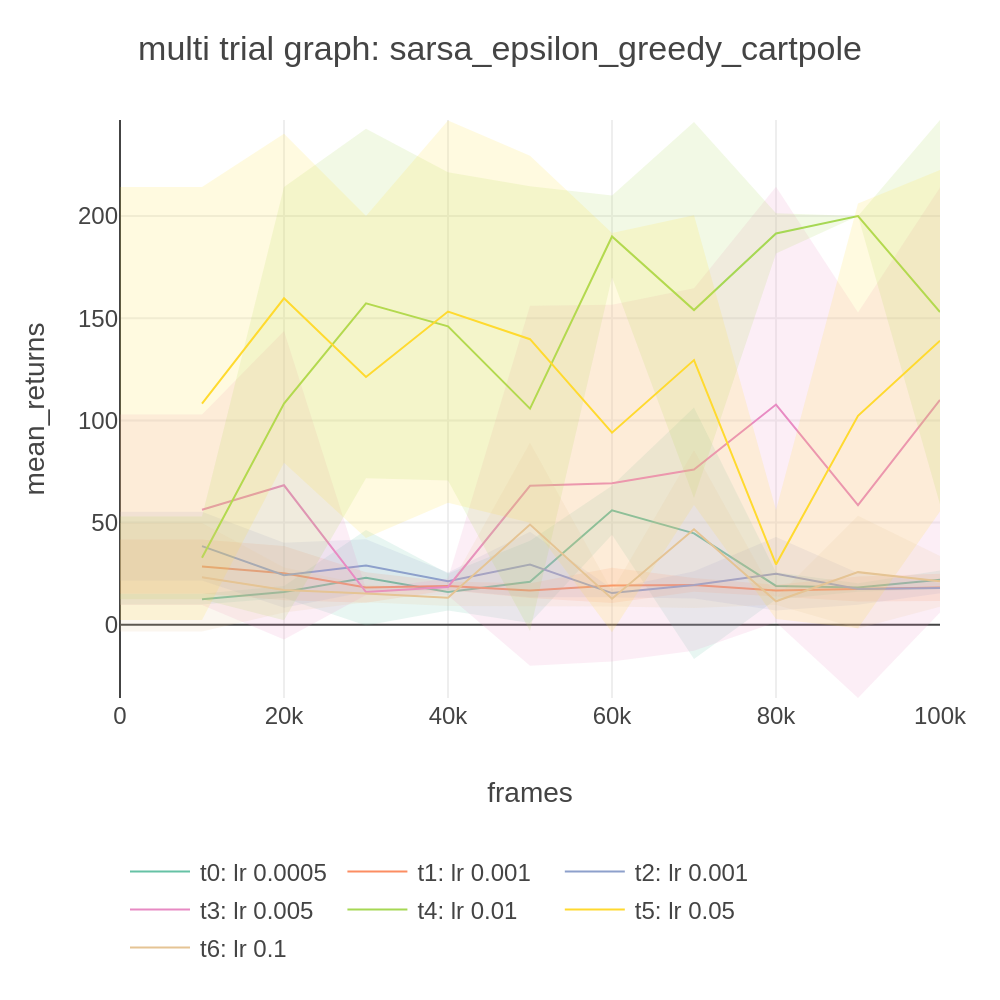


These are the graphs averaged over four sessions with a batch size of 32. These trial graphs are averaged over four sessions. The vertical axis shows the total rewards, while the horizontal shows the total training frames. The graph on the right is a moving average with a window of 100 evaluation checkpoints. We can see in the left graph how it reaches a maximum return from timestep 50k and then spends the following timesteps going from around 150 returns to the max. In the mean graph, we can see a better-smoothed version, but the peak is not achieved until around 100k timesteps.

By using the search function, we obtain the following graphs by looking for the optimal learning rate for the carpole use case. To use this search, we will just modify a little bit our first execution code.

conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/sarsa/sarsa\_cartpole.json sarsa\_epsilon\_greedy\_cartpole search



We can see that a higher learning rate makes the returns go higher in less time, but also it is possible that our parameter update will become too large and change for the worst as the training cycle continues, so it is not constant and might not be the best option in all cases. But in the cases where the learning rate is too small, the training process is very slow. For this scenario, we can see how the best learning rate was 0.01, as it reached the maximum value faster (60k timesteps) and didn’t deteriorate as much as other learning rates.

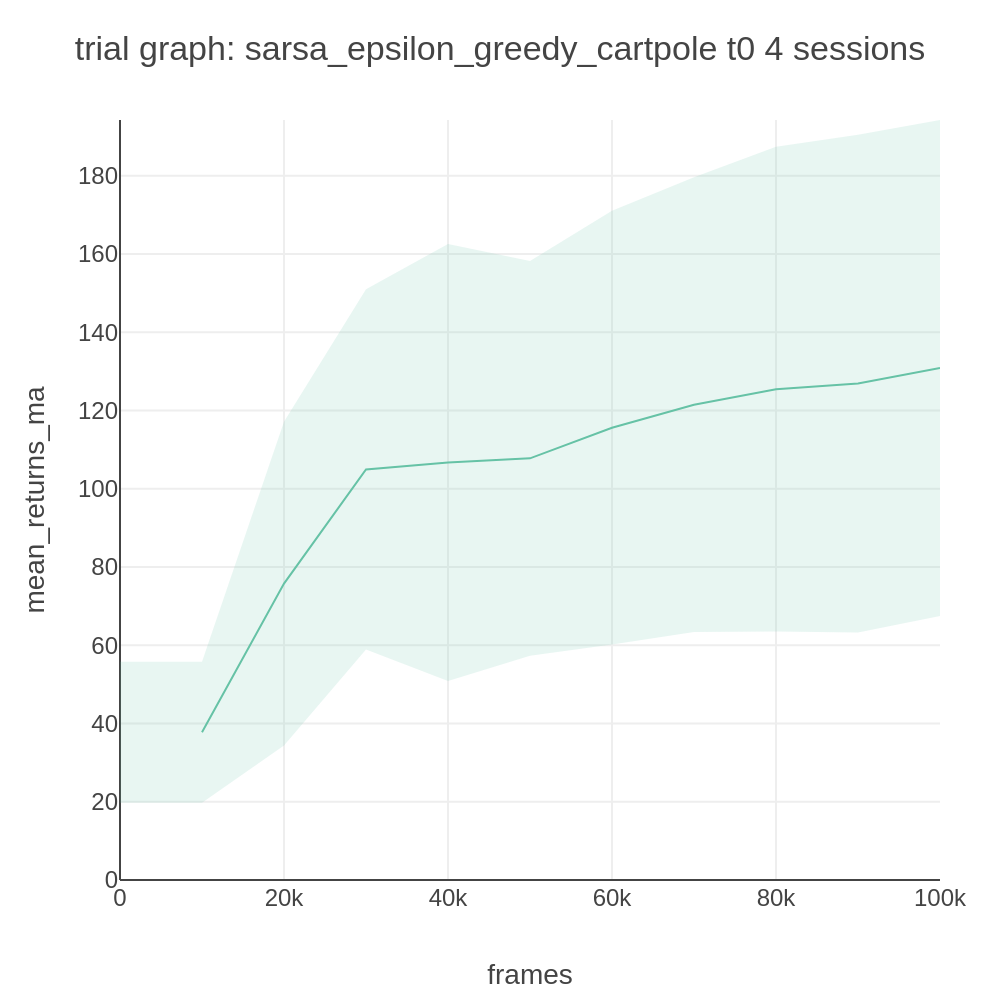
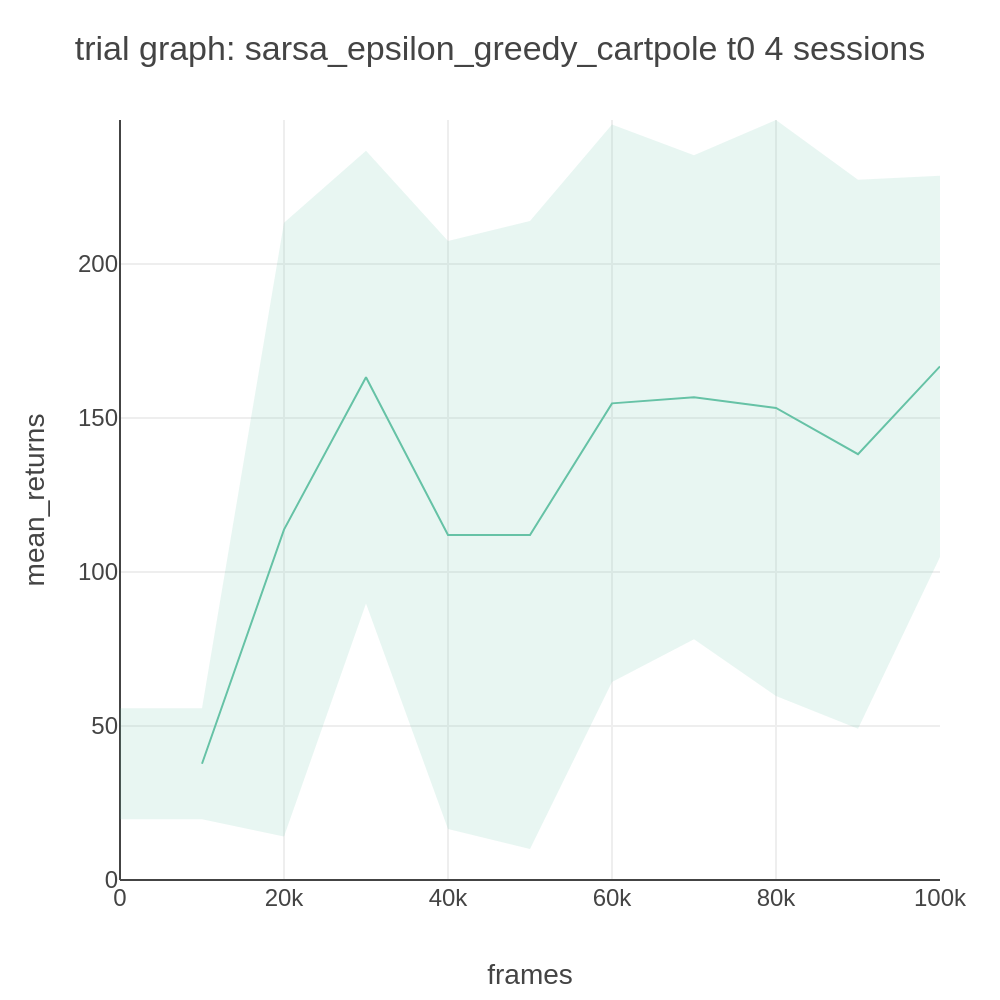
Next, we show the loss and total return of using our agent with batch size 5. To change the batch size, we must remember a line we worked with in our spec.

"training\_frequency": 5

After changing the batch to 5, we obtain the following graphs when we run our experiment using again:

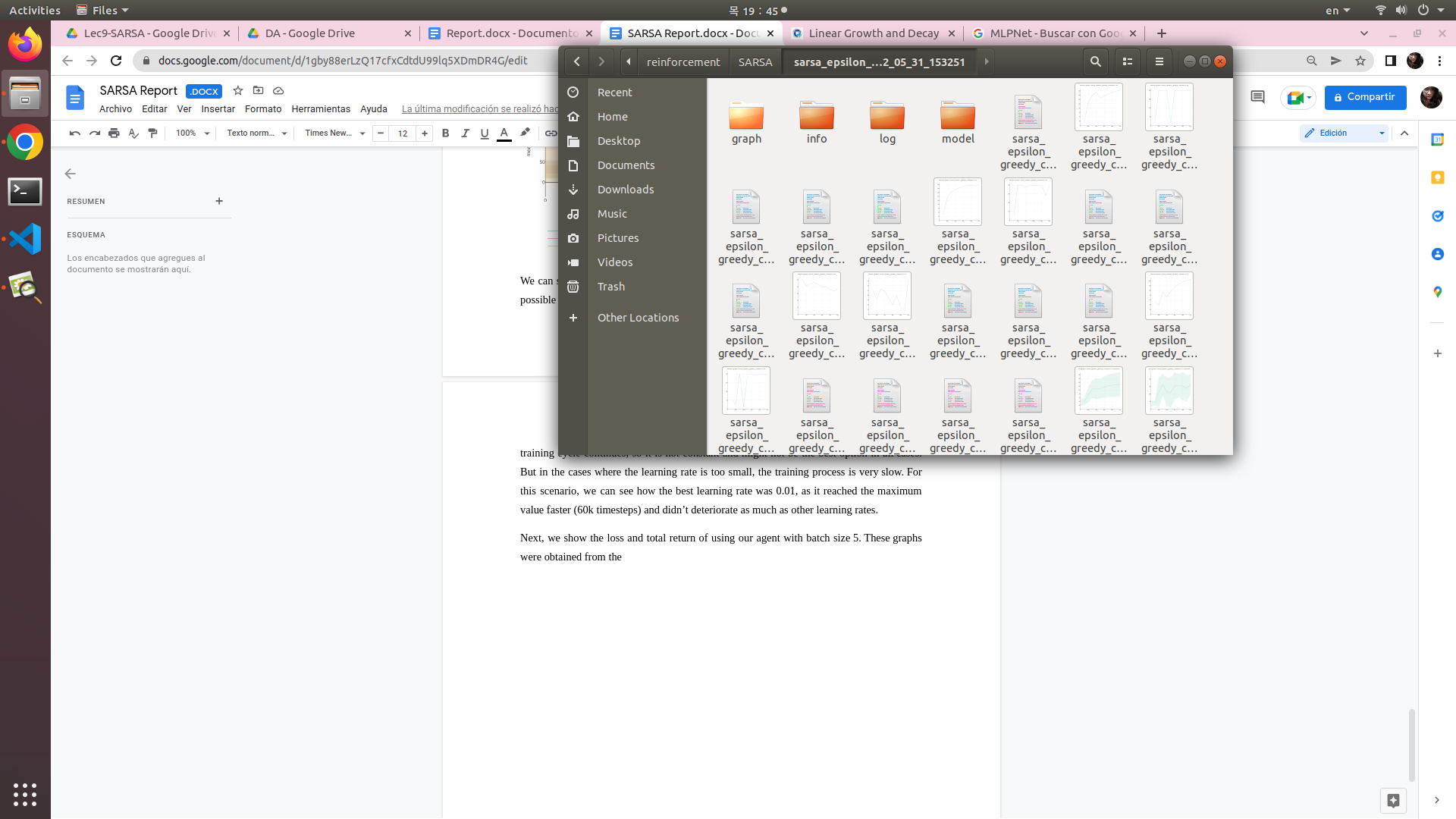
conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/sarsa/sarsa\_cartpole.json sarsa\_epsilon\_greedy\_cartpole train

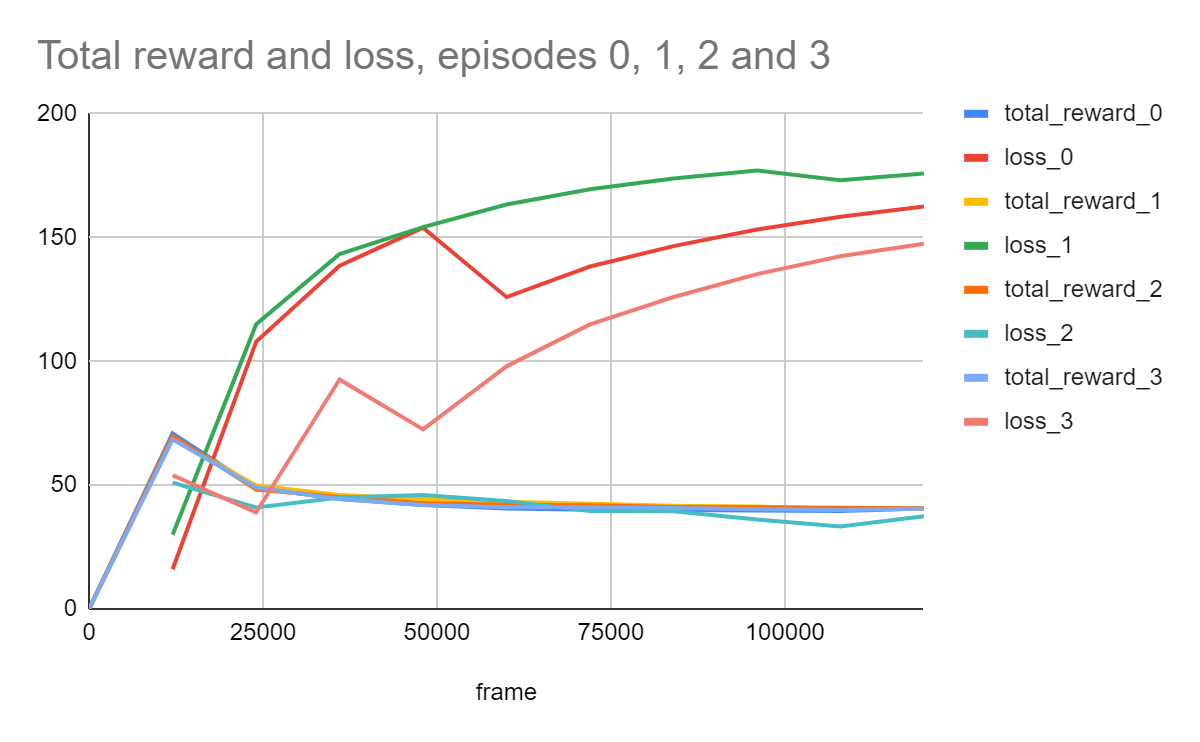


We can see that the returns are less with smaller batch size, and the peak returns are not achieved until 10k steps later (30k instead of 20k).

These graphs were obtained from the results of the lab experiment. We can obtain them from the graph, info or log folders. For now, we will use the csv created in the info folder and create a graph for each episode, using the training information.



This is the graph of the results. As we can see, the behaviour is similar throughout the sessions.



We can conclude that a smaller batch size did not help our agent. The loss keeps growing as the timesteps continue, white the total reward is maximized too quickly. This might be because smaller batch sizes produce more variance.

The code of this report can be found in:

<https://github.com/ErickaBermudez/reinforcement22/tree/master/SARSA>