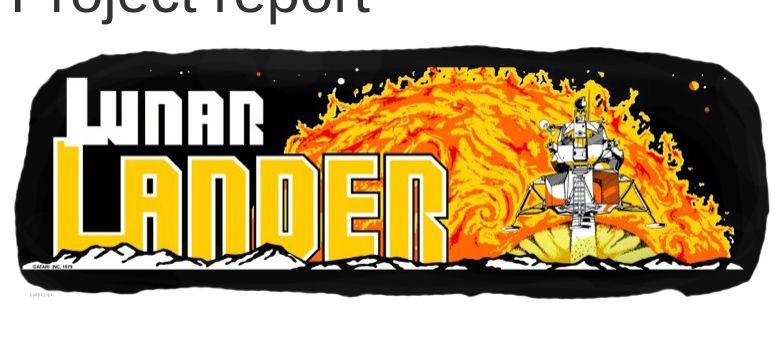
Project report



General information

For the course:

Implementing Artificial Neural Networks (ANNs) with Tensorflow (winter term 2019/20)

Topic:

A2C for continuous action spaces applied on the LunarLanderContinuous environment from OpenAI Gym

Participants:

Jonas Otten
Alexander Prochnow
Paul Jänsch

Outline

1. Introduction/Motivation

As a final project of this course one possible task was to identify an interesting research paper in the field of ANNs and reproduce the content of the paper with the knowledge gained during the course. In our case we first decided on what we wanted to implement and then looked for suitable papers on which we can base our work.
Inspired by the lecture about Deep Reinforcement Learning (DLR) held by Leon Schmid we wanted to take the final project as an opportunity to gather some hands-on experience in this fascinating area. So Reinforcement Learning it is. But where to start? We were looking for something which offers a nice tradeoff between accessibility, challenge and chance of success. The Gym environments provided by OpenAI seemed to do just that. Most of them are set up and ready to run within a minute and with them there is no need to worry about how to access observations or perform actions. One can simply focus on implementing the wanted algorithm.
Speaking of which, from all the classic DRL techniques we knew so far, the Synchronous Advantage Actor-Critic algorithm (short A2C) seemed most appropriate for the extent of the project. Challenging but doable in a few weeks. This left us with two remaining questions to answer before we could start our project.
First, which environment exactly should our learning algorithm try to master using A2C? Since there are better solutions than A2C for environments with discrete action spaces, Leon recommended us to go with the LunarLanderContinuous environment.
And second, which A2C related papers provide us with the neccessary theoretical background and also practical inspiration on how to tackle the implementation? The answer to this question we want to give in the next section about background knowledge.

2. Theoretical Background

In RL an agent is interacting with an environment by observing a state s_t of a state space S and taking an action a_t of an action space A at each discrete timestep t. Furthermore the agent receives a reward r_t at particular timesteps after executing an action. The agents takes the actions accoring to a policy π. In the LunarLanderContinuous environment the agent receives a reward after each action taken.

We assume that the environment is modelled by a Markov decision process (MDP), which consists of a state transition function $\mathcal{P}$ giving the probability of transitioning from state s_t to state s_{t+1} after taking action a_t and a reward function $\mathcal{R}$ determining the reward received by taking action a_t in state s_t. The *Markov property* is an important element of a MDP, that is the state transition only depends on the current state and action and not on the precending ones.
In RL the goal is to maximize the cumulative discounted return at each timestep t:

$$G_t = \sum_t^{\infty} \gamma^t r_t$$

with $\gamma \in (0, 1]$ at each timestep t. There are two estimates of the return, either the state value function $V^{\pi}(s_t)$ giving the estimated return at state s_t following policy π or the state-action value function $Q(s_t, a_t)$ giving the estimated return at state s_t when taking action a_t and following policy π afterwards. In classical RL this problem is approached by algorithms which consider each possible state and action in order to find an optimal solution for the policy π. In continuous state and/or action spaces this approch is computionally too hard.

In order to overcome this problem function approximation has been used to find a good solution for policy π, that maximizes the return G_t. Common function approximators are deep neural networks (DNNs), which gain raising success in RL as a way to find a good policy π in large state and action spaces.

A big problem in the usage of DNNs for RL is the difficulty of computing the gradient in methods, which estimate the policy π_θ with parameters θ directly. The reward function, which depends on the policy π_θ, being maximized is defined by:

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) V^{\pi} = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_\theta(a|s) Q^{\pi}(s, a))$$

$d^{\pi}(s)$ is the stationary distribution, that gives the probability of ending up in state s when starting from state s_0 and following policy π_θ. To compute the gradient $\nabla_\theta J(\theta)$ it is necessary to compute the gradient of the stationary distribution which depends on the policy and the transition function $d^{\pi}(s) = \lim_{t \to \infty} \mathcal{P}(s|s_0, \pi_\theta)$, since the environment is unknown this is not possible.
A reformulation of the gradient of the reward function called the policy gradient theorem (proof: Sutton & Barto, 2017) avoids the calculation of the derivative of the stationary distribution:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} Q^{\pi}(s, a)) \pi_\theta(a|s) \\ &\propto \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} Q^{\pi}(s, a)) \nabla_\theta \pi_\theta(a|s) \\ &= \mathbb{E}_\pi [Q^{\pi}(s, a) \nabla_\theta \ln \pi_\theta(a|s)] \end{aligned}$$

This formula holds under the assumptions that the state distribution $s \sim d_{\pi_\theta}$ and the action distribution $a \sim \pi_\theta$ follow the policy π_θ (on-policy learning). The action-state value function acts as an incentive for the direction of the policy update and can be replaced by various terms, e.g. the advantage function A_t.

An algorithm that makes use of the policy gradient theorem is the actor-critic method. The critic updates the parameters w of a value function and the actor updates the parameters θ of a policy according to the incentive of the critic. An extension of it is the synchronous advantage actor-critic method (A2C). Here multiple actors are running in parallel. A coordinator waits until each agent is finished with acting in an environment in a specified number of discrete timesteps (synchronous). The received rewards are used to compute the cumulative discounted return G_t for each agent at each timestep. No we can get an estimate of the advantage A_t, that is used as incentive for the update of the policy: $A_t^w = G_t - V_t^w$. The gradients get accumulated w.r.t. the parameters w of the value function and θ of the policy:

$$\begin{aligned} d\theta &= d\theta + A_w \nabla_\theta \ln \pi_\theta \\ dw &= dw + \nabla_w (G - V_w)^2 \end{aligned}$$

These gradients are used to update the parameters of the value function and the policy. After that all actors start with the same parameters. This algorithm is a variation of the original asynchronous actor-critic method (A3C), where each actor and critic updates the global parameters independently, which leads to actors and critics with different parameters.

Sources used:
* the A3C paper
* Lilian Weng's blogpost about Policy Gradient Algorithms
* A2C Code provided by OpenAI

Project development log

Here we desribe how we approached the given problem, name the steps we have taken and lay out the motivation for the decisions we made in the process of this project. (Readers only interested in the final result with explanations to the important code segments can skip this part and can continue with the paragraph "The model and the experiment")

Instead of directly heading into the complex case of an environment with continuous action space, we decided to first starting with a simpler version of A2C. Namely, A2C for a discrete action space and without parallelization. For this we took the CartPole gym environment. Mastering this environment was the objective of phase 1, which also can be seen as a prephase to phase 2 (the main phase)

Phase 1:

- getting the gym environment to run
- setting up two simple networks for the actor and the critic
- using the actor network to run one agent for arbitrarily many episodes and save the observations made
- using the saved observations to train both actor and critic based on the estimated return

Even with our simple network architecure we were able to observe a considerable learning effect, finally leading to our agent mastering this simple environment. Although the training result was not stable enough (after several succesful episodes the agent started to get worse again) we decided to not optimize our setup on the CartPole environment, but instead switching to an environment with continous action space and optimizing our learning there. Which leads us to phase 2.

Phase 2:

- changing to the LunarLanderContinuous gym environment
- deviding the current jupyter notebook into seperate python files(main.py, coordinator.py, agent.py, actor.py and critic.py)
- the agent now contains the
  - creation of the environment,
  - running an episode and saving the observations
  - computing the gradients for both networks and returning them to the coordinator
- the coordinator
  - creates the agent
  - tells the agent to run an episode based on the current actor
  - and uses the returned gradients to update the networks
- modifying the network architecture of the actor to match the new action space: it now has to return two pairs of mean and variance values, each pair describing one normal distribution from which we sample the action for the main and the side engine
- at this point we decided to implement parallel computing of episodes with multiple agents to speed up the learning (because up to this point we were not able to see any useful learning):
- we looked at different parallelization packages and after some testing we decided to go with Ray
- Ray allowed us to run multiple agents on our CPUs/GPUs in parallel and with this significantly boosting our learning

With the speed-up provided by the parallelization and further fixes of minor but sometimes critical issues we were finally able to observe our agents learning useful behaviour in the LunarLander environment up to the point where the Lander actually stoped chrashing down on the moon every single time and made its first successful landings. That's one small step for the RL research, one giant leap for our team.
But we were not quite satisfied with the result yet. The learning process was still very slow and so we decided to add one more ingredient: Long short-term memory or LSTM for short. Adding LSTM to the actor network is said to greatly improve its performance. Further it might enable our agents to solve other environments, like the BipedalWalker, which require some kind of longer lasting memory.
We advanced into the last phase of our project, which mainly deals with improvements like the implementation of LSTM but also with cleaning, restructuring and polishing the code to achieve its final form.

Phase 3:

- LSTM implementation:
- adding the pre-build LSTM-Layers by Keras to the Actor network
- expanding the parameter list of the actor's constructor such that one can choose whether the network should use the newly added LSTM layers or the previously used Dense layers
- **(describe problems of LSTM here and write that we will not remove the LSTM code beacuse it is a nice approach and the default learning can still be done with the Dense Layers)**
- have code infer parameters from environment
- adding an ArgumentParser to the main.py to allow for different settings to be used when calling the main.py (test/training run, type of actor network, number of agents used, type of environment)
- cleaning the code:
- removing old unused code parts
- adding necessary comments to the code

The model and the experiment

This section makes up the main part of our report. Here we will highlight and explain the important parts of our project's implementation. We are trying to present the code in the most semantic logical and intuitive order to facilitate the comprehension. The code itself is already structured into several classes and we will always indicate which class we are currently talking about.
We are starting with the coordinator class because, as its name suggests, it organizes the use of every other class and also the whole procedure of the learning process. From there we will go step by step and jump into the other classes as they are coming up.

```
28      if args.train:
29          # start training run with given hyperparameters
30          coord = Coordinator(
31              num_agents=args.num_agents,
32              network=args.network_type,
33              env_name=args.environment,
34              num_steps=args.num_steps)
35          coord.train()
```

Figure 1: main.py

The instantiation of the coordinator happens in the main.py **(Figure 1)** and the execution of its `__init()__` method initializes everything needed for successful learning. The most crucial point in this part is probably the instantiation of the two Neural Networks which build the core of the A2C method, namely the Actor and the Critic.

```
35          # Initialize model, loss and optimizer
36          self.actor = Actor(temp_env, network)
37          self.critic = Critic()
38          self.actor_optimizer = tf.keras.optimizers.Adam(learning_rate=0.00005)
39          self.critic_optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
40          self.mse = tf.keras.losses.MeanSquaredError()
41          self.actor_loss = None
42          self.critic_loss = None
```

Figure 2: coordinator.py init

As one can see here, network related things like the loss function and the optimizers are also created at this point. But let's take the chance to go into both classes and look at the architectures of the networks **(Figure 3 & 4)**.

The Critic:

```
5   # State value fuction estimator used to compute the advantage
6   class Critic(Layer):
7
8       def __init__(self):
9           super(Critic, self).__init__()
10          self.fc1 = keras_layers.Dense(units=128, input_shape=[8,], activation='relu', kernel_regularizer="l2")
11          self.fc2 = keras_layers.Dense(units=64, activation='relu')
12          self.out = keras_layers.Dense(units=1, activation=None)
13
14      def call(self, x, training=False):
15          x = self.fc1(x)
16          x = self.fc2(x)
17          x = self.out(x)
18          return x
```

Figure 3: critic.py

The Actor:

```
5   # Policy/actor network
6   # Estimates the parameters mu and sigma of the normal distribution
7   # used to sample the actions for the agent
8   class Actor(Layer):
9
10      def __init__(self, env, network):
11          super(Actor, self).__init__()
12
13          self.action_space_size = env.action_space.shape[0]
14          self.type = network
15
16          if self.type == "lstm":
17              self.lstm1 = kl.LSTM(32, return_sequences=True, return_state=True)
18              self.lstm2 = kl.LSTM(32, return_sequences=True, return_state=True)
19
20          if self.type == "mlp":
21              self.fc1 = kl.Dense(units=128, activation='relu', kernel_regularizer="l2")
22              self.fc2 = kl.Dense(units=64, activation='relu')
23              self.fc3 = kl.Dense(units=32, activation='relu')
24
25          self.batch_norm = kl.BatchNormalization()
26
27          self.mu_out = kl.Dense(units=self.action_space_size, activation='tanh')
28          self.sigma_out = kl.Dense(units=self.action_space_size, activation='softplus')
29
30      def call(self, x, initial_state=[None, None]):
31          state = None
32
33          if self.type == "lstm":
34              x, s1_h, s1_c = self.lstm1(x, initial_state=initial_state[0])
35              x = self.batch_norm(x)
36              x, s2_h, s2_c = self.lstm2(x, initial_state=initial_state[1])
37
38              state_1 = [s1_h, s1_c]
39              state_2 = [s2_h, s2_c]
40              state = [state_1, state_2]
41
42          if self.type == "mlp":
43              x = self.fc1(x)
44              x = self.fc2(x)
45              x = self.fc3(x)
46
47          mu = self.mu_out(x)
48          sigma = self.sigma_out(x)
49
50          return tf.reshape(mu, [-1, 2]), tf.reshape(sigma, [-1, 2]), state
```

Figure 4: actor.py

This is just to gain a quick overview of the networks for now, as we will explain our choice of e.g. activation functions as they become more apparent.
Back in the `__init()__` method of the coordinator, there is one more important step to talk about. The creation of the agents which will run on the environment in a parallel manner.

```
55          # instantiate multiple agents (ray actors) and set first one as chief
56          self.agent_list = [A2CAgent.remote(self.num_steps, env_name) for _ in range(num_agents)]
57          self.agent_list[0].set_chief.remote()
```

Figure 5: coordinator.py agent instantiations

The instantiation of the agents exhibits an anomaly: the keyword `.remote`. This is necessary, because the agent class is declared as a Ray remote class, which has the following implications when instantiated:

- Instantiation must be done with `Agent.remote()` instead of `Agent()` as seen in the screenshot
- A worker process is started on a single thread of the CPU
- An Agent object is instantiated on that worker
- Methods of the Agent class called on multiple Agents are executed on their respective worker and can therefore execute in parallel, but must be called with `agent_instance.function.remote()`
- Returns of a remote function call now return the task ID, the actual results can be obtained later when needed by calling `ray.get(task_ID)`

After instantiation we assign the first agent to be the "chief" **(Figure 5)**. His environment will be rendered during training, while the environments of the other agents will run in the background. This adds a fun way to watch the performance of our AI, other than graphs and numbers (not that those are not fun, too).

Besides the Ray specific specialties, the agent class still has a normal `__init()__` method on which we want to have a short glance now:

```
9       def __init__(self, num_steps, env):
10          self.chief = False
11          self.env = gym.make(env)
12          self.num_steps = num_steps
13          self.finished = False
14
15          # environment parameters
16          self.obs_space_size = self.env.observation_space.shape[0]
17          self.action_space_size =  self.env.action_space.shape[0]
18          self.action_space_bounds = [self.env.action_space.low[0], self.env.action_space.high[0]]
19
20          # get initial state and initialize memory
21          self.state = self.env.reset()
22          self.memory = Memory(self.num_steps, self.obs_space_size, self.action_space_size)
```

Figure 6: agent.py init

Noteworthy here is the creation of the OpenAI Gym environment in which the agent will act **(Figure 6 line 11)** and the instantiation of the agent's memory **(Figure 6 line 22)**. The memory is represented as an object of our Memory class. As expected an object of this class is responsible for storing the observations an agent makes temporally. This includes states visited, actions taken, rewards received, information whether a terminal state is reached and, not being an observation in particular, a return estimate. We will have a look at important methods of the Memory class when we are dealing with the agents executing actions and making observations.

The rest of the coordinator's `__init()__` handles the preparation of the tensorboard in order to be able to inspect the training progress.
Now that our coordinator is fully operational we can start the training by calling its `train()` method in the main.py **(Figure 1 line 35)**.
This method is the heart of the coordinator and will be assisted by quite a lot of helper methods and also some other classes we did not talked about in detail yet. We will go through all of them and explain their use in the order they are needed in the `train()` method.

```
67      def train(self, num_updates=6000):
68          # called from main
69          cum_return = 0
70          num_epsisodes = 0
71          for i_update in range(num_updates):
72              # Collect num_agents * num_steps observations
73              for t in range(self.num_steps):
74                  memories = self.step_parallel(t)
```

Figure 7: coordinator.py train

First, we advance the environments a number of timesteps equal to our hyperparameter `num_steps` by calling `step_parallel(t)` accordingly **(Figure 7 line 73-74)**. Then, in later parts of the `train()` method, we use the collected observations to update the networks. This way we update the network parameters only every `num_steps` (e.g. 32) timesteps. Before we can get into how we update the networks though, we must first have our agents act in the environment and return observations to us. This is the purpose of the `step_parallel(t)` method. It advances all environments from timestep t to t+1 by observing the current state and then computing an action to perform **(Figure 8)**.

```
129         # Compute one step on all envs in parallel with mlp as policy network
130         if self.network == "mlp":
131             # Observe state to compute an action for the next time step
132             states, dones = zip(*(ray.get([agent.observe.remote(t) for agent in self.agent_list])))
133             action_dist, _ = self.get_action_distribution(np.array(states))
134             # Sample action from the normal distribution given by the policy
135             actions = np.array(action_dist.sample())
136
137         # Execute action and obtain memory after num_steps
138         memories = ray.get([agent.execute.remote(actions[i], t) for i, agent in enumerate(self.agent_list)])
139         return memories
```

Figure 8: coordinator.py step_parallel

Observing the current states of the environments of multiple agents can be done in parallel by calling the `agent.observe()` function on all agents **(Figure 8 line 132)**. Being a remote function, our list comprehension will return a list of task IDs and not the actual states, therefore we must call `ray.get()` to obtain them. Taking a look at the `observe()` function **(Figure 9)** we notice that if we are at the start of a new update, it will reset the agents memory, since we only want to take observations made in the current update cycle into account for the current network update **(Figure 9 line 26-27)**. We will elaborate on the memory class in the coming section. For now, all we want is the current state, which is stored in the `self.state` attribute of the agent. If the previous episode was finished the environment will be reset and the attribute will instead contain the initial state of the new episode **(Figure 9 line 30-31)**.

```
24      def observe(self, t):
25          # reset memory for new network update
26          if t == 0:
27              self.memory.reset(self.num_steps, self.obs_space_size, self.action_space_size)
28
29          # reset environment at the end of an episode
30          if self.finished:
31              self.state = self.env.reset()
32
33          # render chief
34          if self.chief:
35              self.env.render()
36
37          self.state = np.reshape(self.state, [1,self.obs_space_size])
38
39          return self.state, self.finished
```

Figure 9: agent.py observe

Returning the current state of every agent to the coordinator, we are now ready to compute our next action for each agent. As described previously: In Lunar Lander, our agent's action consists of two values, one controlling the main engine, the other the side engines. The values can take on virtually any real number within [-1, 1]. We sample these values from two normal distributions per agent **(Figure 8 line 133-135)**, each with parameters mu and sigma, denoting the location and scale of the distribution. These parameters (one mu and sigma for the main engine distribution and one mu and sigma for the side engines distribution per agent) are the output of our Actor neural network. To compute them, we call the `get_action_distribution()` function **(Figure 10)**, which passes the current states of all environments to the actor network **(Figure 4)**. It returns the mentioned mus and sigmas, which we use to create normal distribution objects **(Figure 10 line 145)** that will now be sampled from **(Figure 8 line 135)**.

```
142     def get_action_distribution(self, state, recurrent_state=None, update=False):
143         # Get the normal distribution over the action space, determined by mu and sigma
144         if self.network == "mlp":
145             mu, sigma, _ = self.actor(state.squeeze())
146             return Normal(loc=mu, scale=sigma), None
```

Figure 10: coordinator.py get_action_distribution

At this point the reasons for our architectural choices for the actor network become apparent: The tanh of the mu output layer **(Figure 4 line 27)** keeps the center of our normal distributions, i.e. the average of our sampled values within [-1, 1], which is useful since this is exactly the action space. Similarly for the sigma output layer **(Figure 4 line 28)**, a softplus activation ensures that the mathematical restrictions of the standard deviation are upheld, namely $\sigma \geq 0$.

Lastly, to complete our step in the environment, we execute the computed actions **(Figure 8 line 138)**. A deep dive into the agent's execute function is required before we return the agent's memories to the coordinator. That is, because we have to form the agent's memories first. Let us take a look how this is done.

```
41      def execute(self, action, t):
42
43          if self.finished:
44              self.finished = False
45
46          # clip action value if necessary to be within action space
47          action = np.clip(action, self.action_space_bounds[0], self.action_space_bounds[1])
48
49          # perform action and store the resulting state and reward
50          next_state, reward, done, _ = self.env.step(action)
51          self.memory.store(self.state, action, reward, done, t)
52          self.state = next_state
53          self.finished = done
54
55
56          if t == (self.num_steps - 1):
57              return self.memory
58          else:
59              return None
```

Figure 11: agent.py execute

The agent performs the action given on the environment and stores the resulting state, reward and done flag returned by the environment, then updates the internal state `self.state` and the finished flag **(Figure 11 line 50-53)**.
His observations are stored by the memory object instantiated from our Memory class (memory.py). It is initialized in the agents `__init__` as seen before **(Figure 6 line 22)** and posseses numpy arrays to store states, actions, rewards, estimated returns and terminal booleans denoting whether a terminal state is reached. The agents memory starts of empty **(Figure 12 line 10-15)**. Observations can be stored in the arrays via the index representing the timesteps **(Figure 12 line 17-22)**

```
5   class Memory:
6       def __init__(self, num_steps, obs_space_size, action_space_size):
7           self.obs_space_size = obs_space_size
8           self.action_space_size = action_space_size
9
10          # use environment parameters to initialize observation arrays
11          self.states = np.empty(shape=(num_steps, obs_space_size))
12          self.actions = np.empty(shape=(num_steps, action_space_size))
13          self.rewards = np.zeros(shape=(num_steps, 1))
14          self.estimated_return = np.empty(shape=(num_steps, 1))
15          self.terminals = []
16
17      def store(self, state, action, reward, done, t):
18          # store observations from timestep t
19          self.states[t] = state
20          self.actions[t] = action
21          self.rewards[t] = reward
22          self.terminals.append(done)
23
24      def reset(self, num_steps, obs_space_size, action_space_size):
25          # clears observation arrays
26          self.__init__(num_steps, obs_space_size, action_space_size)
```

Figure 12: memory.py store

Now that the memory of the agents are filled with the exciting experiences of one timestep, they are eager to return them to the coordinator. But we have taught them well, so that they will only return them to the coordinator when the required `num_steps` is reached **(Figure 11 line 56-59)**. Until then they repeat the observe and execute routine and only afterwards collectively return a list containing every agent's memory object to the coordinator. Now it is time for the coordinator to utilize these memories to make the agents better.

```
67      def train(self, num_updates=6000):
68          # called from main
69          cum_return = 0
70          num_epsisodes = 0
71          for i_update in range(num_updates):
72              # Collect num_agents * num_steps observations
73              for t in range(self.num_steps):
74                  memories = self.step_parallel(t)
75              # Compute discounted return and concatenate memories from all agents
76              [m.compute_discounted_cum_return(self.critic) for m in memories]
77              self.memory = sum(memories)
78
79              # calculate mean gradient over all agents and apply gradients to update models.
80              mean_policy_gradients, mean_critic_gradients = self._get_mean_gradients()
81              self.actor_optimizer.apply_gradients(zip(mean_policy_gradients, self.actor.trainable_variables))
82              self.critic_optimizer.apply_gradients(zip(mean_critic_gradients, self.critic.trainable_variables))
83
84              self.step += self.num_steps
```

Figure 13: coordinator.py train

We do this by first computing the discounted cumulative return **(Figure 13 line 76)**. As described in the theoretical background, our goal is to maximize the cumulative discounted return at each timestep. We had defined it as $G_t = \sum_t^{\infty} \gamma^t r_t$. For each memory object, we store G_t in the attribute `self.estimated_return`. This is calculated by iterating over the reversed list of rewards **(Figure 14 line 54)** and summing up all rewards, but discounting future rewards more heavily **(Figure 14 line 57)**, e.g. $G_1 = R_{t=1} + \gamma \cdot R_{t=2} + \gamma^2 \cdot R_{t=3} + \dots$. Our function does this in an unintuitive manner, but it becomes apparent if one would go through this with an example: Looking at a reward list with only 3 entries at the end of an episode: Our `cumulative_return` gets initialized with 0. Our `estimated_return[3]` for timestep 3 is R_3 (`self.rewards[3]`). Our `cumulative_return` is now R_3. For timestep 2 the estimated return is $R_2 + \gamma \cdot R_3$. Now finally for timestep 1 the discounted cumulative return G_1 (our `estimated_return[1]`) is $R_1 + \gamma \cdot (R_2 + \gamma \cdot R_3) = R_1 + \gamma \cdot R_2 + \gamma^2 \cdot R_3$ if we multiply the γ into the brackets. This is exactly what we wanted and we hope that the functionality of this method is now more clear. It is important to notice that this estimated return only depends on rewards of following states and not of the ones before.

Back in the coordinator, concatenating all the made observations accross all agents can then be done using the sum function **(Figure 13 line 77)**, as we have adjusted the memory class's `__add__` behavior method, i.e. what happens when adding two memory objects together, namely that their observations are concatenated.

```
44      def compute_discounted_cum_return(self, critic):
45          # compute the discounted cumulative return after observing num_steps observations
46          self.estimated_return.setflags(write=1)
47          idx = (len(self.rewards) - 1)
48          # initialize the estimated return for the last observation
49          if self.terminals[idx]:
50              cumulative_return = 0
51          else:
52              cumulative_return = critic(np.reshape(self.states[idx], [1,self.obs_space_size]))[0,0]
53          # reverse the observations and compute the gamma discounted return for each timestep
54          for i in range(idx, -1, -1):
55              if self.terminals[i]:
56                  cumulative_return = 0
57              self.estimated_return[i][0] = self.rewards[i][0] + GAMMA * cumulative_return
58              cumulative_return = self.estimated_return[i][0]
```

Figure 14: memory.py compute_discounted_cum_return

The memory object of the coordinator now contains the collective memory of all agents and their discounted returns. These are needed to compute the actor loss and critic loss, which we want to minimize, so we compute their gradients. This is coordinated by the `_get_mean_gradients()` function **(Figure 13 line 80, Figure 15)**. Since we have two networks, two gradients are computed: The policy gradients minimize the actor loss, therefore maximizing the estimated return **(Figure 15 line 161)** and the critic gradients minimize the critic loss, which will minimize the Mean Squared Error for the state value function **(Figure 15 line 163)**.

```
159     def _get_mean_gradients(self):
160         # Compute gradients for the actor (policy gradient), Maximize the estimated return
161         self.actor_loss, policy_gradients = self._compute_gradients('actor')
162         # Compute gradients for the critic, minimize MSE for the state value function
163         self.critic_loss, critic_gradients = self._compute_gradients('critic')
164         return policy_gradients, critic_gradients
```

Figure 15: coordinator.py get_mean_gradients

Let's look at how we calculate the two losses. Firstly, we see that the final actor loss **(Figure 16 line 170)** is adjusted by an entropy term. Adding the entropy term to the actor loss has been found to improve exploration, which minimizes the risk of convergence to an only locally optimal policy (A3C paper page 4). This adds a new hyperparameter, the entropy coefficient (`ENTROPY_COEF`), which balances the amount of exploration.

```
166     def _compute_gradients(self, type):
167         with tf.GradientTape() as tape:
168             if type == 'actor':
169                 # Compute the actor loss:
170                 loss = self._actor_loss() - self.action_dist.entropy() * ENTROPY_COEF
171             else:
172                 # Compute the state value
173                 state_v = self.critic(self.memory.states, training=True)
174                 # Compute the critic loss
175                 loss = self.mse(self.memory.estimated_return, state_v, sample_weight=0.5)
176             # Compute the gradients
177             return loss, tape.gradient(loss, self.actor.trainable_variables if type == 'actor' else
    self.critic.trainable_variables)
```

Figure 16: coordinator.py compute_gradients

The unmodified actor loss is returned from our `_actor_loss` method, which first estimates the state value by passing all states to the critic **(Figure 3)**. In the Lunar Lander environment, a state is a vector of 8 values, denoting different aspects within the environment, e.g. the coordinates of the vessel. So our critic takes this state vector as an input and outputs the state value. Applying L2-regularization has improved our critic loss during training **(Figure 3 line 10)**.
The state values are now used to get an estimate of the advantage **(Figure 17 line 183)**. But our actor loss also consists of a second part. As described in the theoretical background, our actor gradients are updated via $d\theta = d\theta + A_w \nabla_\theta \ln \pi_\theta$. We have the advantage A_w, now we need to compute the log policy probability $\ln \pi_\theta$. Here's how:
Using our previously mentioned `get_action_distribution` function (**Figure 10**), we recompute the normal distributions that we sampled our performed actions in each respective state from **(Figure 17 line 186)**. Inputting the recorded action back into the normal distribution's log probability density function returns us the relative log probability $\ln \pi_\theta$ of sampling that action **(Figure 17 line 187)**.

```
179     def _actor_loss(self):
180         # Compute state value
181         state_v = self.critic(self.memory.states)
182         # Get advantage estimate
183         advantages = self.memory.estimated_return - state_v
184         advantages = tf.cast(advantages, tf.float32)
185         # Get log probability of the taken action
186         self.action_dist, self.update_recurrent_state = self.get_action_distribution(self.memory.states, update=True)
187         logprob = self.action_dist.log_prob(self.memory.actions)
188         # Advantage as baseline
189         return -logprob * advantages
```

Figure 17: coordinator.py actor_loss

Our actor loss therefore consists of the log probability and the advantage, which is used as a baseline for the log probability here to reduce the variance of the policy gradients (A3C paper page 3). These gradients must still be computed from the actor loss, which we do using the tensorflow gradient tape **(Figure 16 line 177)**.
Having finished computing the policy gradients, we now move on to the critic gradients **(Figure 15 line 163)**. For those we calculate the Mean Squared Error between the discounted cumulative returns **(Figure 14)** and the state values, which are again obtained from pushing the observed states through the critic network **(Figure 16 line 172-175)**. As we did with the actor loss, gradient of the critic loss is now computed with the help of the gradient tape **(Figure 16 line 177)** and the policy and critic gradients are now returned to the coordinator **(see below: Figure 13 line 80)**.

```
67      def train(self, num_updates=6000):
68          # called from main
69          cum_return = 0
70          num_epsisodes = 0
71          for i_update in range(num_updates):
72              # Collect num_agents * num_steps observations
73              for t in range(self.num_steps):
74                  memories = self.step_parallel(t)
75              # Compute discounted return and concatenate memories from all agents
76              [m.compute_discounted_cum_return(self.critic) for m in memories]
77              self.memory = sum(memories)
78
79              # calculate mean gradient over all agents and apply gradients to update models.
80              mean_policy_gradients, mean_critic_gradients = self._get_mean_gradients()
81              self.actor_optimizer.apply_gradients(zip(mean_policy_gradients, self.actor.trainable_variables))
82              self.critic_optimizer.apply_gradients(zip(mean_critic_gradients, self.critic.trainable_variables))
83
84              self.step += self.num_steps
```

Figure 13: coordinator.py train

Finally, we can let the Adam optimizers travel through the loss spaces roughly oriented towards the direction of greatest descent given by our gradients, i.e. applying the gradients to update our models **(Figure 13 line 81-82)**.

Our A2C algorithm has now run through one iteration and performed a single coordinated update of our two networks. All agents will now start their next set of steps in the environment with the same network parameters, which is the before mentioned specialty of the A2C algorithm compared to A3C.

How to run training and testing:
Having read our implementation, we hope you are now eager to try it out!
We have included our trained models, which were saved using the `tf.train.Checkpoint` function. These can be tested by calling `python3 main.py --test`, which will automatically load the trained models and render the environment. We have also added further arguments to the command line parser, which can be viewed using `python3 main.py --help`. Most notably the policy network type can be changed here (`--network_type "mlp"` or `--network_type "gru"`). The number of agents for training can be changed as well, e.g. `--num_agents 12` . To train the model, use `--train` .

Visualization and results

...