Project “Multi-Agent DRL”: Report

This report describes the techniques used and the ideas behind the chosen methods. It will not repeat the content of the ReadMe file.

Intro

The Reinforcement Learning (RL) problem posed by the Unity multi-agent tennis problem, from a RL-agents perspective is defined by the environment.

Environment

The environment is defined by its interaction with the agent. Therefore it is defined by its state space, its action space and the reward that is returned when acting on the environment, while moving a time step ahead into the future.

Our task is episodic, therefore the environment needs to return whether an episode has passed or not to the agent. If an episode has passed, the environment has to reset to its original state.

Specifications:

State space dimension: 24 continuous features
Action space dimension: 2 continusous actions in range [-1, 1] for each agent (2 agents)
Rewards per timestep and specific to agent who hits the ball: Ball across net: +0.1
Reward if ball hits ground (both agents): -0.01
Episode over: done = 1, else 0

Approach: MADDPG

A multi-agent deep deterministic policy gradient (DDPG) agent is utilized to solve this task. This is an actor-critic approach.

- The “Critic” estimates the value function. This could be the action-value (the Q value)
- The “Actor” updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

In our example the neural network (NN) of the “Actor” learns to choose actions that step into the direction of the optimal policy, proposed by the “Critic” NN. Once the “Actor” has finished training, the “Critic” NN is no longer required, as the “Actor” itself incorporated stepping towards the optimal solution, along the optimal policy, layed out by the “Critic”.

Techniques to enhance learning

DDPG

In this project an "actor-critic" DDPG, as defined in Continuous control with deep reinforcement learning by Timothy P. Lillicrap et al. was implemented.

Improved Exploration

To improve exploration of this agent stochastic Ornstein-Uhlenbeck was added to the selected action, which lead to an improved learning curve.

Stable Deep Reinforcement Learning(DRL)

In order to make the DRL agent more stable in regards to auto-correlation of weights adjustments, the "Fixed Q-Targets" method was utilized combined with a "Replay Buffer". We update the target neural networks (NN) wiht a "soft-update" method after each training step. Thereby the target network (see "Fixed Q-targets") is iteratively updated with the weights of the trained "regular" NN.

Multi-Agent approach

In order to solve this environment, we will train an agent that has 2 different policies for its 2 sub-agents. One sub-agent is controlling one of the tennis players, the other sub-agent is controlling the other. Each sub-agent tries to maximize its own reward.

As they have a common reward of -0.01 in case the ball hits the ground, the agents will learn to collaborate in respect to not let the ball hit the ground. They will compete in respect to the reward of +0.1 to the respective agent who hits the ball cross the net.

Technical realization

To have 2 different policies for taking actions, 2 different “Actor” neural networks must exist, along with their set of functions, such as “act(), step(), learn()”.

The “Critic” assesses the Q-value(state-action value) of the state-action pair in the learning process of the actors.

Important!!

In this case, as the environment is symmetrical, and the sub-agents observations are symmetrical in regard to their position as well, the “Critic” can be shared among them.

For an asymmetric environment, where the states for the sub-agents are not symmetric, a separate “Critic” for each agent is required!!

Model

Instead of a Q-table, a Q-networks are used as knowledge storage. This Q-networks are deep neural network.

Fixed Q-Targets

To implement the above mentioned technique of “Fixed Q-Targets” 2 NNs are required both for “Actor” and “Critic”. A regular network (also referred to as “local”) and one target network (the one that eventually should make the predictions).

Both NNs for the “Actor” are initialized exactly the same, have same layers and same weights. Same for the 2 NNs of the “Critic”.

Actor 1 and Actor 2

Why fully connected layers?

I chose to use a neural network with 3 hidden layers. These 3 hidden layers are fully connected layers. The idea behind choosing fully connected layers is, that the input features are not related. (In contrast to pixels in an image)

In the following I will describe the neural network from input layer to output layer.

Input layer

The input layer is a 1-d array of length of the state size. Each input feature is represented as an element of this array.

Fully connected layer 1

The first hidden layer is a fully connected layer with 512 neurons. We reduce the number of dimensions in order to be able to narrow down on information that lies in the combination of the input features.

RELU activation function

This layer is succeeded by a rectified Linear Unit (RELU) activation function, that is applied on each neuron. This activation function maps the output of the neuron to max(0,output). This is done to make activation more clear. If a neurons values sum up to a negative value, the neuron will not have negative impact on the output, but will have output 0 instead. This makes computation faster.

Fully connected layer 2

We then gradually decrease the number of dimensions to intensify the physical meaning of the dimensions(neurons).

The second hidden layer is a fully connected layer with 256 neurons. This layer is succeeded by another RELU activation function.

RELU activation function

This layer is succeeded by a rectified Linear Unit (RELU) activation function, that is applied on each neuron. This activation function maps the output of the neuron to max(0,output). This is done to make activation more clear. If a neurons values sum up to a negative value, the neuron will not have negative impact on the output, but will have output 0 instead. This makes computation faster.

Fully connected layer 3: Output layer

We further decrease the dimensions to concentrate the meaning in the output neurons.

The third hidden layer is a fully connected layer with 2 neurons. This layer outputs 2 dimensions. They represent the 2 dimensions of the action.

Hyperbolic tanget function

Instead of using a ReLU activation function in the output layer, which would flatten negative output, we chose the hyperbolic tangent function. This function maps the complete range of [-inf, inf] to the range of [-1,1]. This helps at computation of the gradient.

Critic

Input layer

The input layer is a 1-d array of length of the state size. Each input feature is represented as an element of this array.

Fully connected layer 1

The first hidden layer is a fully connected layer with 512 neurons. We reduce the number of dimensions in order to be able to narrow down on information that lies in the combination of the input features.

RELU activation function

This layer is succeeded by a rectified Linear Unit (RELU) activation function, that is applied on each neuron. This activation function maps the output of the neuron to max(0,output). This is done to make activation more clear. If a neurons values sum up to a negative value, the neuron will not have negative impact on the output, but will have output 0 instead. This makes computation faster.

Concatenate Action Dimensions from “Actor”

We concatenate the output of the ReLU function of the first fully connected layer with 256 neurons with the 4 action dimensions, predicted by the “Actor”. These also must play into the predictions the “Critic” will make for the Q-value of choosing the predicted action in the current state.

Fully connected layer 2

We then gradually decrease the number of dimensions to intensify the physical meaning of the dimensions(neurons).

The second hidden layer is a fully connected layer with 256 + 4 neurons, according to the output of the concatenated layer. This layer is succeeded by another RELU activation function.

RELU activation function

This layer is succeeded by a rectified Linear Unit (RELU) activation function, that is applied on each neuron. This activation function maps the output of the neuron to max(0,output). This is done to make activation more clear. If a neurons values sum up to a negative value, the neuron will not have negative impact on the output, but will have output 0 instead. This makes computation faster.

Fully connected layer 3: Output layer

We further decrease the dimensions to concentrate the meaning in the output neurons.

The third hidden layer is a fully connected layer with 1 neuron. This layer outputs the predicted Q-Value for choosing the action (predicted by the actor) in the current environment state.

Code

The model is defined in **model_new.py**

Agent

The agent implements the necessary functions it requires to perform RL with an environment. The agent itself can be used on different environments. Therefore its functions are general to all environments.

An RL agent needs to implement these functions:

- Choose action
- Update the knowledge

As this agent implements the advanced techniques of “Experience Replay” and “Fixed Q-targets” some additional functionality is required:

- Append experience to replay buffer (Experience Replay)
- Update the knowledge of the target network (Fixed Q-targets)

Agent functions

The agent is implemented in mddpg_agent.py. The following explanations are regarding the functions in that file:

- Act(): Chooses an action epsilon greedy, to allow for less greedy action selection in the beginning of learning, therefore high exploration, by random action selection. Increasing greedy action selection over the course of learning and decreasing exploration.
- Step(): Stores a State-Action-Reward-Next_State (SARS) tuple in the Replay buffer, that was created by acting on the environment.
- Learn(): Uses Deep Learning to learn from a batch of experiences from the Replay Buffer and update the knowledge (Q-networks)
- Soft_update(): As the error function for the Deep Learning task is computed using the “Fixed Q-targets” technique, a target Q-network exists. This Q-network is updated, by simply copying the

weights and biases of the original Q-network, that was trained (after the training batch has passed)

Experience Replay: Replay Buffer

The idea behind experience replay is, to keep SARS tuples in memory. This makes sense, because these tuples are not affected training the agent. The same action in the same state will always yield the same reward, no matter how smart the agent is. Therefore keeping these tuples in memory is useful, as they can be used at a later point in time.

Fixed Q-targets

This removes the risk of learning a correlation of weight adjustment to the output. While learning adjusting weights can be interpreted as a correlation to the error. This will lead to an increasing or oscillating error function. It is countered by introducing a second set of weights (second Q-network) similar to the original one. The weights of this Q-network are only used to compute the error, and are not changed in the course of learning.

This second "target" Q-network is only updated once a learning batch from the Replay Buffer has passed the original network.

Learning Algorithm

The agent is defined in **ddpg_agent.py**

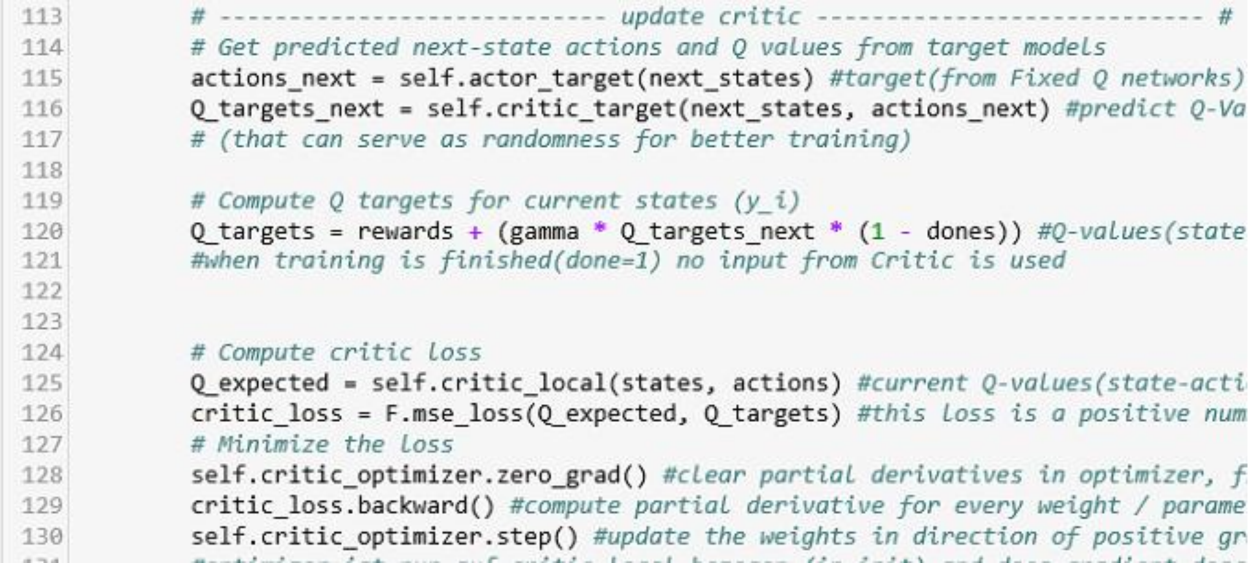
The learn() function implements the DDPG approach, in which the "Critic" and "Actor1" and "Actor2" NNs are trained.

"Critic" Loss Function

The loss function of the "Critic" is the average deviation of the predicted reward (using the local "Critic" NN) -> **Q_expected**, and the (more accurate) reward, predicted by taking the actual reward of the current state and computing the Q-value of the next state by adding the estimated, discounted future rewards to it -> **Q_targets**.

Important to note: When the update is performed the reward of the next state, that the environment returned when acting an action $\in$ *actions* (below code is for multiple agents) on the current state, is known!

Therefore the estimate including the reward, along with predictions for future state-actions **Q_targets** is more accurate than the one based on prediction alone **Q_expected**.

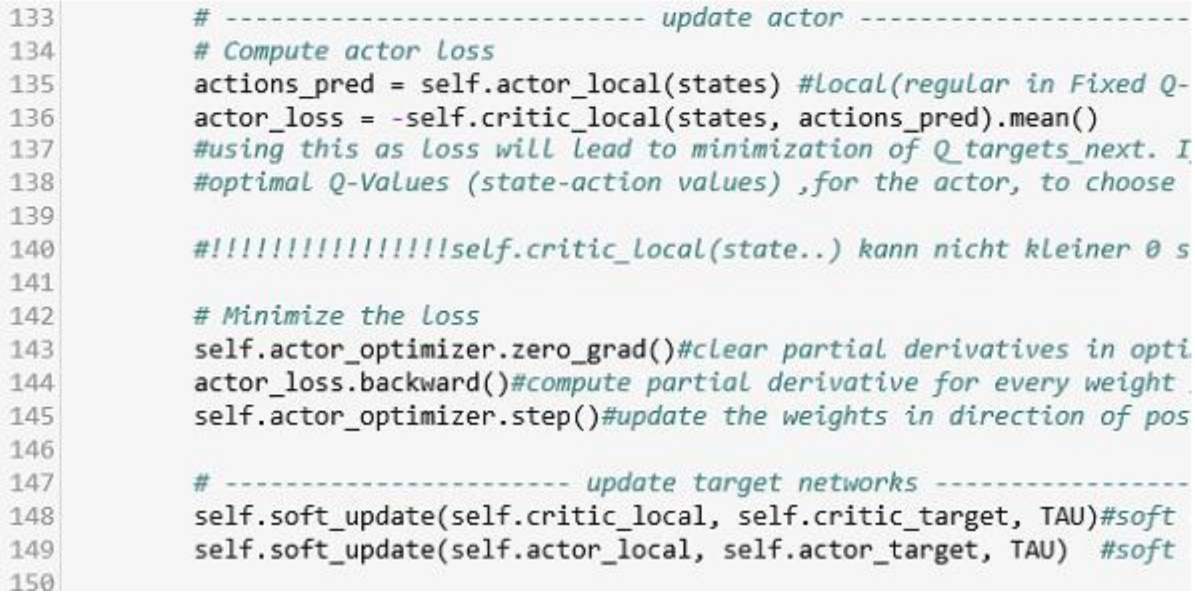


```
        # --------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states) #target(from Fixed Q networks)
        Q_targets_next = self.critic_target(next_states, actions_next) #predict Q-Va
        # (that can serve as randomness for better training)

        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones)) #Q-values(state
        #when training is finished(done=1) no input from Critic is used


        # Compute critic loss
        Q_expected = self.critic_local(states, actions) #current Q-values(state-acti
        critic_loss = F.mse_loss(Q_expected, Q_targets) #this loss is a positive num
        # Minimize the loss
        self.critic_optimizer.zero_grad() #clear partial derivatives in optimizer, f
        critic_loss.backward() #compute partial derivative for every weight / parame
        self.critic_optimizer.step() #update the weights in direction of positive gr
```

This loss function is minimized using gradient descent, by the optimizer.

"Actor" Loss Function

The actor loss function is the negated Q-value function, given by the prediction of the "Critic".



```
        # --------------------------- update actor ----------------------
        # Compute actor loss
        actions_pred = self.actor_local(states) #local(regular in Fixed Q-
        actor_loss = -self.critic_local(states, actions_pred).mean()
        #using this as loss will lead to minimization of Q_targets_next. I
        #optimal Q-Values (state-action values) ,for the actor, to choose

        #!!!!!!!!!!!!!!!self.critic_local(state..) kann nicht kleiner 0 s

        # Minimize the loss
        self.actor_optimizer.zero_grad()#clear partial derivatives in opti
        actor_loss.backward()#compute partial derivative for every weight
        self.actor_optimizer.step()#update the weights in direction of pos

        # ---------------------- update target networks -----------------
        self.soft_update(self.critic_local, self.critic_target, TAU)#soft
        self.soft_update(self.actor_local, self.actor_target, TAU)  #soft

```

By minimizing the negated Q-value (state-action value) function, the Q-value is maximized. This maximizes the expected reward.

Ornstein Uhlenbeck Process

To increase exploration of the agent the actions are summed with noise computed by the Ornstein-Uhlenbeck process. In order not to act forbidden actions, out of the action range, the action dimensions are truncated to the range [-1, 1]:

return np.clip(action, -1, 1)

Train Function

This function is defined in the Jupyter Notebook as **mddpg().**

First loop:

This algorithm iterates over the number of episodes specified. In each episode the environment is reset to an initial state.

Second loop:

In the first loop: While the environment does not indicate that the episode is over, the following is done:

- Agent selects an action: act()
- Action is acted onto environment: env.step()
- The environment passes the new state, along with the reward, and whether the episode is over
    - This SARS tuples are written to the replay buffer for both sub-agents: actor_0.step(), actor_1.step()
- Training occurs in the critic.step(subagent_actor, replay_buffer) function, in which the sub-agent-to-train and the **replay buffer (shared, as environment is symmetric)** are specified.
- Current score (for both sub-agents) is updated with the reward received

The agent internally decides when to learn. It does so, once it has enough experiences collected in the Replay Buffer.

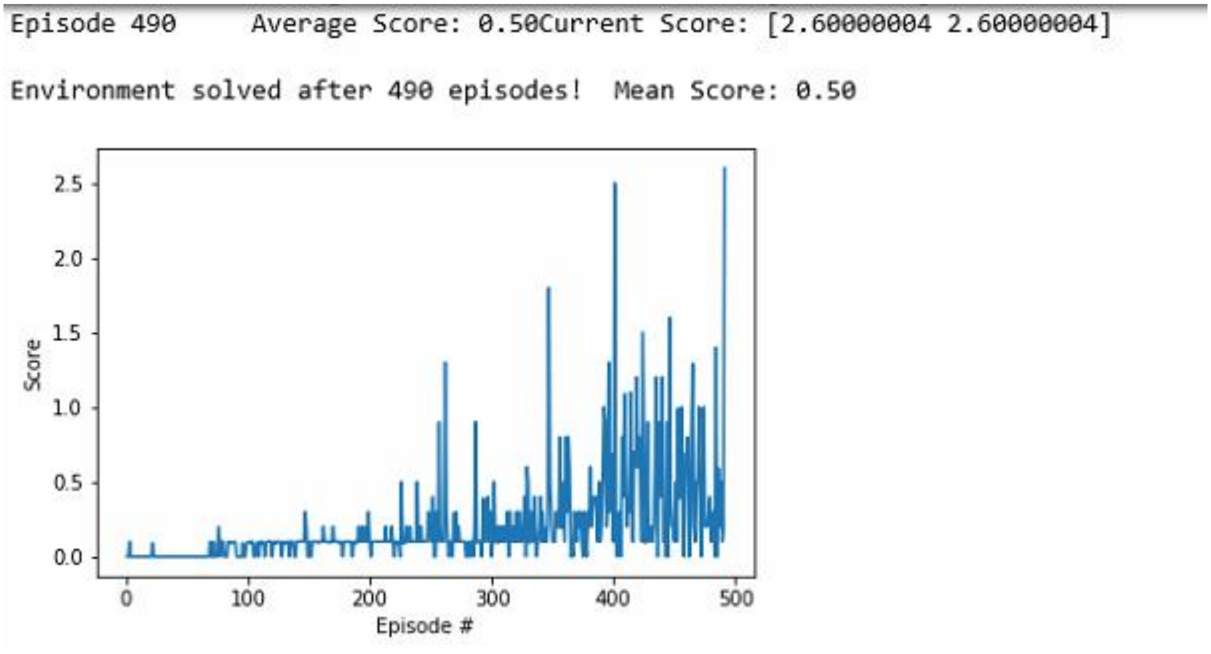
Hyperparameters

The following hyperparameters are used:

- Replay buffer Size: 1000000 (Experience Replay)
- Batch Size for learning: 128 (Experience Replay)
- Gamma: 0.99 (for discounting)
- Tau: 0.1 (Fixed Q-Targets: update parameter for interpolating original Q-network and target Q-network, when writing updated weights to target Q-network)
- Learning Rate for Adam optimizer: 0.0001 (both optimizers: “Critic” and “Actor”)
- WEIGHT_DECAY: 0 (L2 regularization to reduce overfitting)

Result

The rewards per episode evolved in the following way:



```
Episode 490	Average Score: 0.50Current Score: [2.60000004 2.60000004]

Environment solved after 490 episodes!  Mean Score: 0.50
```

After 490 epsiodes, the rolling average (over last 100 episodes) exceeded the target reward value of +0.5.

Enhancements

The result could possibly be enhanced by applying the advanced techniques of

- Adding Ornstein-Uhlenbeck Noise to the parameters of the NNs to enhance learning.
- Add L2 regularization to reduce overfitting
- Add Prioritized experience replay