Project 2: Continuous Control

Featuring the Unity Reacher environment

Learning Algorithm

Deep Deterministic Policy Gradient (DDPG)

Deep-Deterministic-Policy-Gradient is an Actor-Critic algorithm which is model-free as well as off-policy. The method consists of two main components. The Actor and the Critic. Both are represented by Neural Networks. It uses concepts from both the Deep-Q-Network (DQN) as well as policy methods. By combining elements of both, DDPQ (or actor-critic models in general) are able to apply the concepts of DQNs on continuous action spaces through application of policy evaluation and noise, while also improving the stability and robustness of policy methods.

The Actor is represented by a neural network (NN). It is used to evaluate the best action to take considering the current state of the environment, as calculated by the current policy.

The Critic is also represented by a NN and is used to improve the learning of the actor. To accomplish this, the critic is also trained on the same pool of data as the actor but is then used to calculate the loss of the current actor model. This improves the trajectory of the learning.

Parameters:

The parameters were set as following:

BUFFER_SIZE:	The size of the experience buffer (deletes from bottom if overflow). Seemed fairly robust to changes.
BATCH_SIZE	The size of the experience sample the networks are updated with. Best Value was 128. 256 reduced training velocity significantly.
random_seed	A random seed to create reproducible results. This greatly had effect on the performance. Selected seed was 1.
GAMMA	The cost of future rewards. Did not seem to impact greatly except for fairly high/low values. Was left to 0.99.

TAU	The gravity of change to the target network during update. This was set to 0.001. Higher values led to bad performance.
LR_ACTOR	The Learning Rate of the critic.
LR_CRITIC	The learning rate of the critic. Positive Effect if it was lower than the LR of the Actor
WEIGHT_DECAY	Decay of the network weights. Was not used in this sample.
UPDATE_EVERY	Update Cycle of multi-agents. Was not used in this sample.
UPDATE_NETWORK	Update Number of multi-agents. Was not used in this sample.
max_t	The number of steps taken every episode. Performance dropped if under 600 or over 1200.

Components:

Environment:
This is the environment the agent interacts with. In this case the reacher environment from the UnityAgents. The environment is reset at every episode and the actions calculated by the agent are passed in order to make a step in time.

Memory:
This is the shared memory buffer. If the environment only uses one agent, this is identical to an agent owned memory. If the environment uses multiple agents, this memory serves as shared experience pool for all agents.

Agent:
This is theinstance of the Agent class tha interacts with the environment. At the beginning of every episode the agent is reset. It then calculates the preferred action with the .act function, using its local actor network. After interaction with the environment, the interaction is passed to the agents .step() function which saves the experience and eventually updates the local and target networks.

Scores, scores_dequ, scores_list:
These variables are used to store the performance of the agent over time.

Training

Performance:

The Agent shows very slow improvement at the beginning with the first 100 episodes leading to scores around 5 – 8. At around 130 episodes, the agent reaches scores of about 10 and significantly increases velocity. After about 200 episodes, the agent reaches the goal of a reward of 30 with occasional dips under 30. Afterwards the agent stays consistently over 30 on average. This training cycle had a surprising dip around episode 300, but was able to recover afterwards. Going further, the Agent shows a very consistent performance with very few dips. At max, the agent reaches average scores of about 34.

Challenges:

I must admit that challenges were faced with the project as the Udacity workspace would not work with GPU acceleration. I therefore trained the Agent on CPU and was not able to solve the multi-agent environment due to this, which makes me a tad bit disappointed. The code implemented works well though and I was able to get through 200 episodes at 1000 steps in around 45 minutes. Due to these constraints I refrained from all too much parameter optimization.

Plot of rewards:

Here we can see the plot of rewards in raw and aggregated over five episodes for readability.

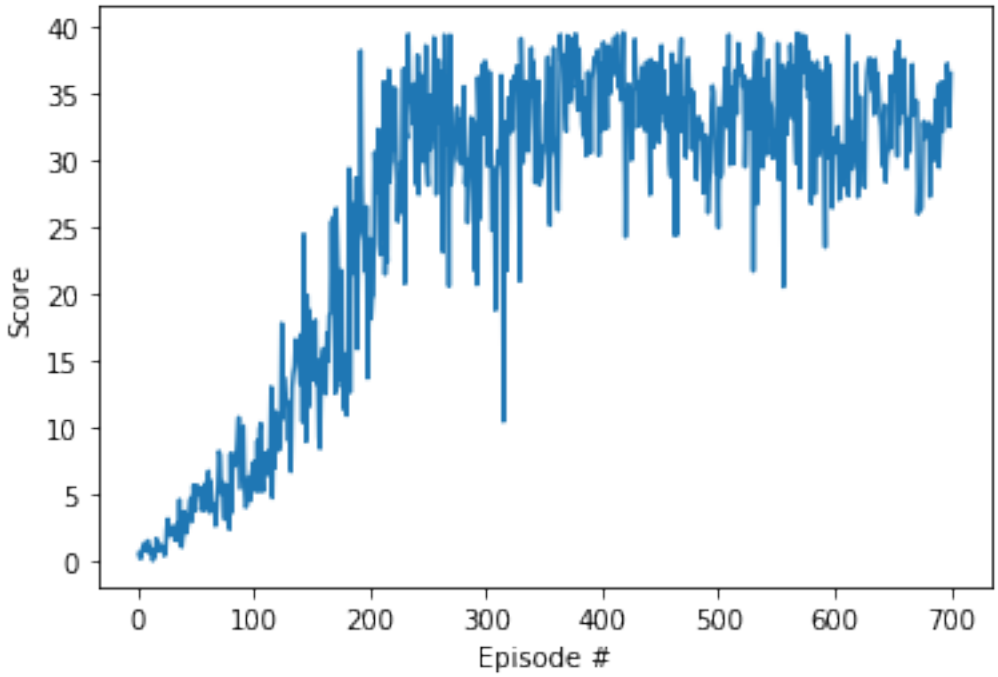

Figure 1: Scores over 700 Episodes raw

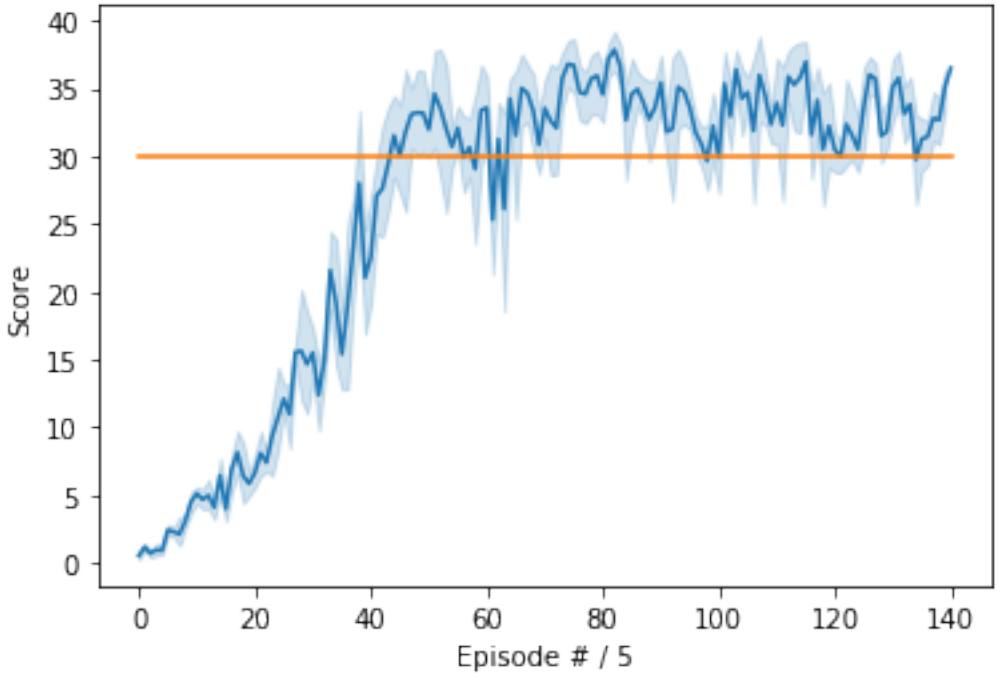

Figure 2: Scores averaged over 5 with goal threshold

Ideas for Future Work

As soon I have the chance I want to revisit the model and apply MADDPG (Multi-Agent-DDPG) as suggested by Rowe et al.(2017).
Also, it would be interesting to experiment with A3C, D4PG and centralizing other components than the memory. A centralized critic makes for slower training but could prove to be very stable in comparison to an individual critic for every agent.

References:

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, O. P., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems* (pp. 6379-6390).