DRLND – P2 Continuous Control

Learning Algorithm 1 – Deep Deterministic Policy Gradient (DDPG)

Policy gradient methods are a type of reinforcement learning technique that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. They do not suffer from many of the problems that have been marring traditional reinforcement learning approaches such as the lack of guarantees of a value function, the intractability problem resulting from uncertain state information and the complexity arising from continuous states & actions.

DDPG falls into the group of policy gradient algorithms which relies on an actor-critic architecture with two elements, an actor and a critic. An actor is used to tune the parameter $\boldsymbol{\theta}$ for the policy function, i.e. decide the best action for a specific state:

$$\pi_\theta(s, a) = \mathbb{P}[a \mid s, \theta]$$

A critic is used for evaluating the policy function estimated by the actor according to the temporal difference (TD) error:

$$r_{t+1} + \gamma V^v(S_{t+1}) - V^v(S_t)$$

Where the lower-case *v* denotes the policy that the actor has decided. In this way the critic acts like a Q-Learning agent to learn the optimal action to take in a given state and the equation closely resembles the temporal difference (TD) learning equation from Q-Learning.

DDPG also borrows the ideas of experience replay and separate target network from Deep Q Network (DQN) learning, so there exists both 'local' and 'target' networks for both the actor and critic**.** The local networks are updated after each timestep whilst the target networks are 'soft' updated according to this equation:

$$\theta_{target} = \tau * \theta_{local} + (1 - \tau) * \theta_{target}$$

Where τ is a hyperparameter which can be set prior to training.

An issue for DDPG is that it seldom performs exploration for actions which can be mitigated by adding noise on the parameter space or the action space. One commonly used noise process is Ornstein-Uhlenbeck Random Process which is what was implemented here, along with a decaying epsilon process to reduce the amount of exploration in latter episodes.

See the next page for DDPG pseudocode and hyperparameters used during training.

DDPG Algorithm Pseudocode:

The pseudo code and hyperparameters used for training the full DPPG algorithm are as follows:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Hyperparameters

```
BUFFER_SIZE = int(1e6)      # replay buffer size
BATCH_SIZE = 256            # minibatch size
GAMMA = 0.99                # discount factor
TAU = 0.001                 # for soft update of target parameters
LR_ACTOR = 1e-4             # learning rate of the actor
LR_CRITIC = 3e-4            # learning rate of the critic
WEIGHT_DECAY = 0.0001       # L2 weight decay
UPDATE_EVERY = 20           # how often to update the network
EPSILON = 1.0
```

I chose to follow the approach indicated in the project instructions with regards to updating the networks every 20 time steps (see update_every parameter) rather than every time steps which they mentioned allowed learning to be more stable.

Neural Network Architectures

The Actor neural network was comprised of two fully connected layers containing 128 hidden units each. A ReLU non-linear activation was used between each layer with batch normalization after each activation. The output layer had 4 nodes (one for each action) and a Tanh non-linearity to squash the outputs between the range [-1, 1] which was the valid range of an action for the arm across its 4 inputs.

The Critic neural network was also comprised of two fully connected layers each with 128 hidden units. The output from the first layer was passed through a ReLU non-linear activation, then through a batch normalization regularization layer and finally as input to the second layer. Here it was concatenated with the 4 actions to give 132 hidden units and passed to a single output node via a ReLU activation.

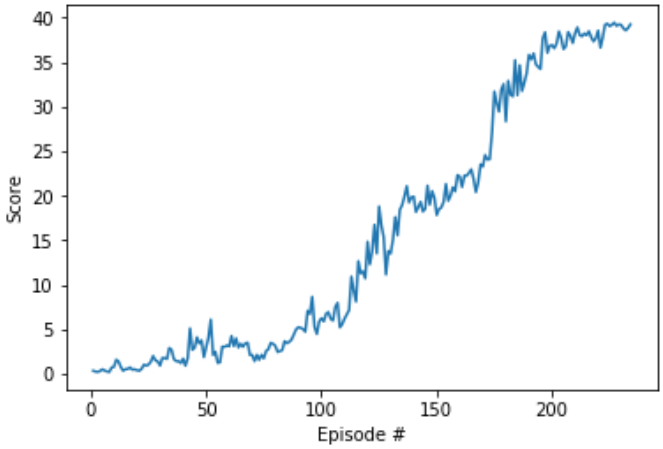
Please see appendix for a full visualisation of each network.

Results

I used the 20 agent version of the environment for training which using the hyperparameters mentioned above lead to the environment being solved in 134 episodes (i.e. the average score across all 20 agents was >= 30.0 for 100 episodes).

Here is a summary of the results from training the agent:

```
Episode 100     Average Score Last 100 Episodes: 2.54   Avg. Score (All Agents) Last Episode: 6.26
Episode 200     Average Score Last 100 Episodes: 20.91  Avg. Score (All Agents) Last Episode: 36.91
Episode 234     Average Score Last 100 Episodes: 30.10  Avg. Score (All Agents) Last Episode: 39.24
Environment solved in 134 episodes!     Average Score: 30.10
```



Future Ideas for Increasing the Agent's Performance

The limiting factor on increasing performance is the amount of time it takes to train an/a agent(s). If time was allowable a grid search methodology could be run allowing a range of hyperparameters (learning rates, discounting, soft update ratios etc.) to be tested so that nearer optimal parameters might be found.

A clear observation is that the agents don't really make substantial increases in their score until around episode 100, which might indicate that there is too much exploration occurring in the early episodes relative to exploitation of a greedy strategy.

Additionally, the structures of the neural networks could be experimented with to see if adding additional layers or hidden units would allow the agent to train faster, different non-linear activations might be experimented with, or further regularizations applied.

Another recommendation might be to have a "warm up" period where the agent acts randomly to collect experiences (s, a, r, s') for a number of episodes in order to fill the experience replay memory before training begins. This would be particularly useful if prioritized experience replay were implemented also as priority would be given to sampling experiences which are likely to give the biggest improvements in the agents learning.

Acknowledgments

I'd like to thank fellow students Gregorio Mezquita, Daniel Barbosa and Hemanta Gupta for their guidance and assistance with this project, in particular their sharing via Slack for what was working for them in training the agents.

The DRLND Slack community is great!

Learning Algorithm 2 – Proximal Policy Optimization (PPO)

Out of curiosity I additionally implemented PPO from Shangtong Zhang's GitHub repo which can be found here: https://github.com/ShangtongZhang/DeepRL

Some modifications were required to the training and monitoring processes to make the code work with the Unity environment rather than OpenAI Gym and to make the results comparable with the DDPG algorithm. In particular I modified the run_steps and ppo_continuous functions and the PPOAgent class. Please refer to the Continuous_Control_PPO jupyter notebook for details.

In summary the PPO Algorithm works as follows:

1. First, collect some trajectories based on some policy $\pi\theta$, and initialize theta prime $\theta'=\theta$
2. Next, compute the gradient of the clipped surrogate function using the trajectories
3. Update θ' using gradient ascent $\theta' \leftarrow \theta' + \alpha \nabla_{\theta'} L_{\text{surclip}}(\theta', \theta)$
4. Then we repeat step 2-3 without generating new trajectories. Typically, step 2-3 are only repeated a few times
5. Set $\theta=\theta'$, go back to step 1, repeat.

Where a trajectory is a collection of rewards accumulated across multiple timesteps and the surrogate function and its gradient are:

$$g = \nabla_{\theta'} L_{sur}(\theta', \theta)$$

$$L_{sur}(\theta', \theta) = \sum_t \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t^{future}$$

More information can be found about PPO here:

https://blog.openai.com/openai-baselines-ppo/

https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Deep-Reinforcement-Learning-Through-Policy-Optimization

Some useful tips regarding hyperparameter ranges from Unity can be found here:

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md

Hyperparameters

The following hyperparameters were used for training:

```
config.discount = 0.99
config.use_gae = True
config.gae_tau = 0.95
config.gradient_clip = 5
config.rollout_length = 4096
config.optimization_epochs = 10
config.num_mini_batches = 256
config.ppo_ratio_clip = 0.2
config.log_interval = 2048
config.max_steps = 2e7
```

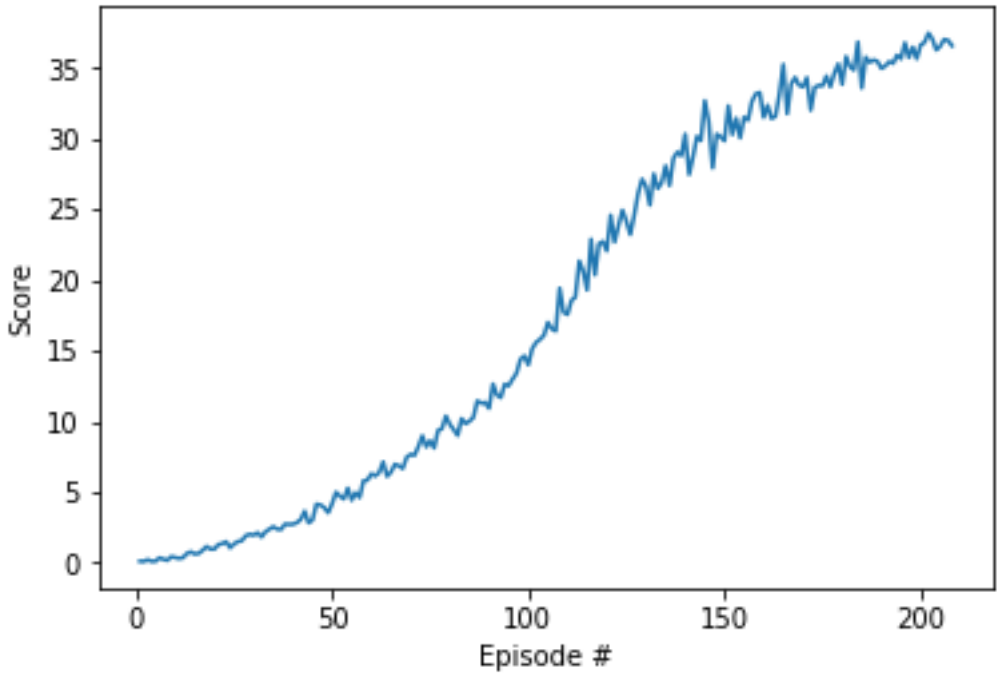
Neural Network Architecture

The neural network was set up as Gaussian Actor Critic structure. The Actor & Critic network contained 2 layers, each of 128 hidden units with ReLU activations. The Actor network had 4 outputs (one for each action) and the Critic had one as we used for the DDPG networks above.

Results

Here is a summary of the results obtained from using the PPO algorithm and hyperparameters mentioned above:

```
Episode 200     Average Score Last 100 Episodes: 28.99  Avg. Score (All Agents) Last Episode: 36.59
Episode 208     Average Score Last 100 Episodes: 30.61  Avg. Score (All Agents) Last Episode: 36.55
Environment solved in 108 episodes!     Average Score: 30.61
```

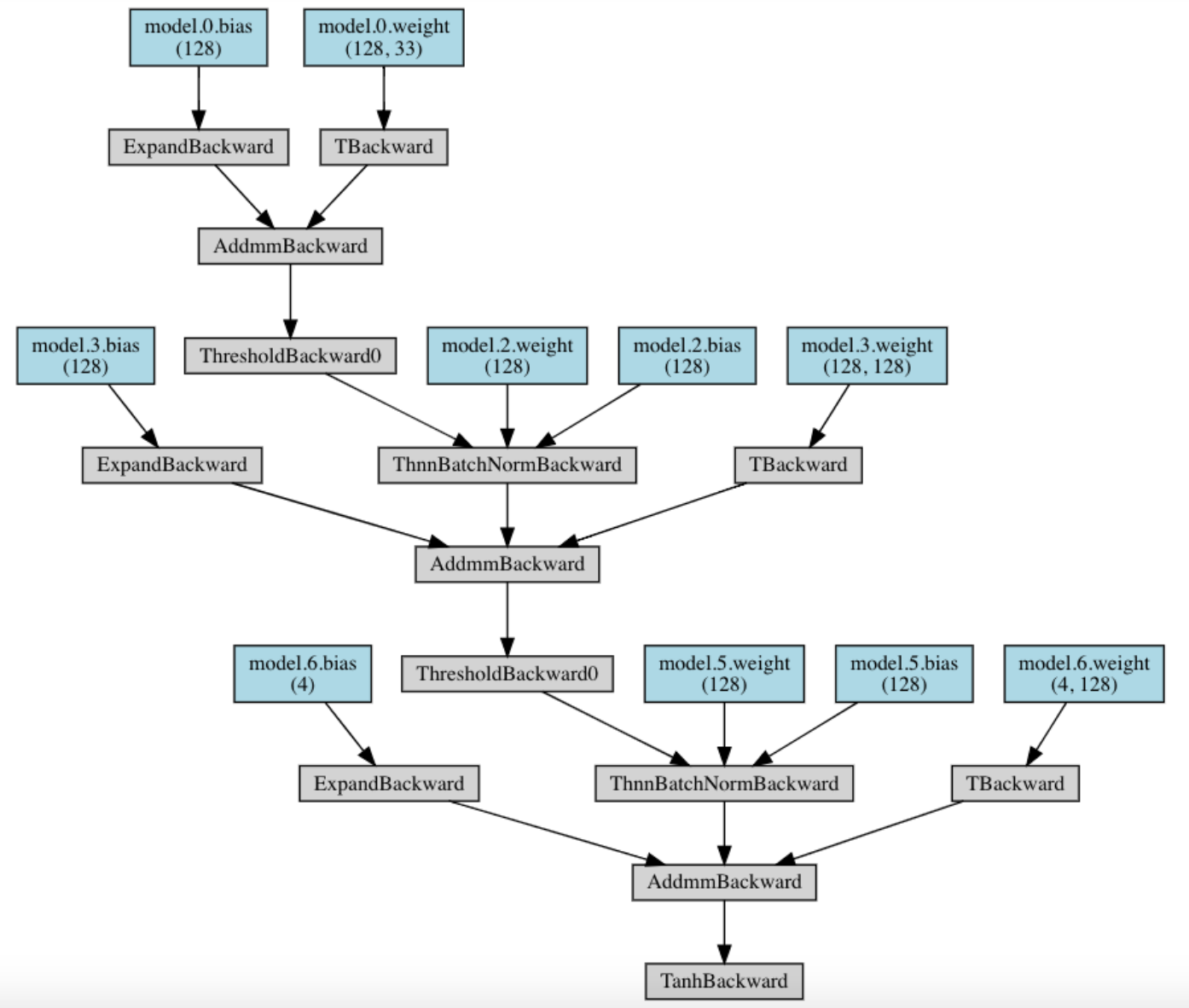


The graph of learning shows that the training for this algorithm follows a much smoother learning curve than was observed for the DDPG algo, it also managed to solve the environment in 26 fewer episodes.

I'd like to spend more time experimenting with hyperparameters for PPO as it seems from this initial experiment to be a very powerful algorithm for solving continuous control tasks.

Appendix

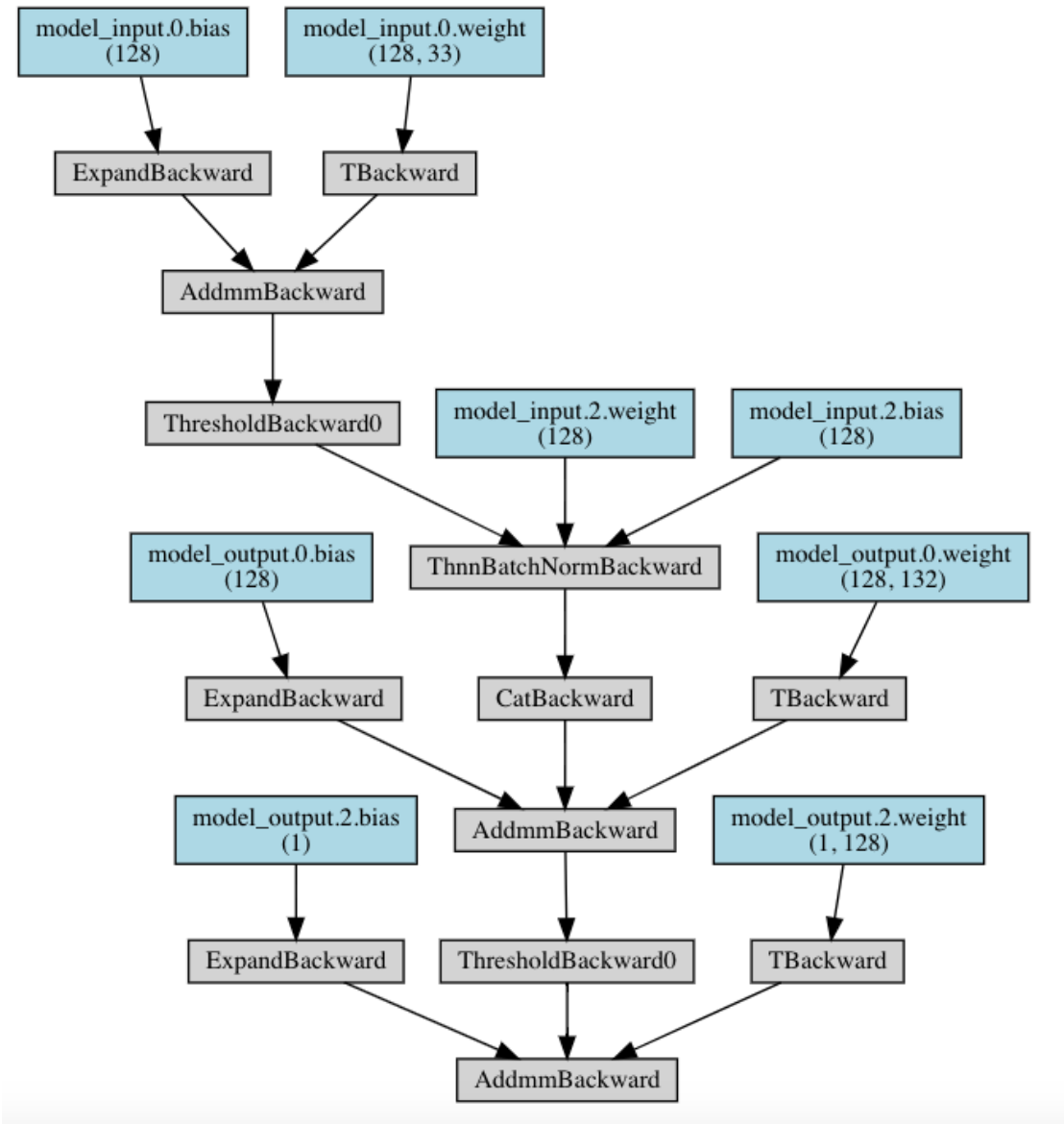
DDPG Actor Network



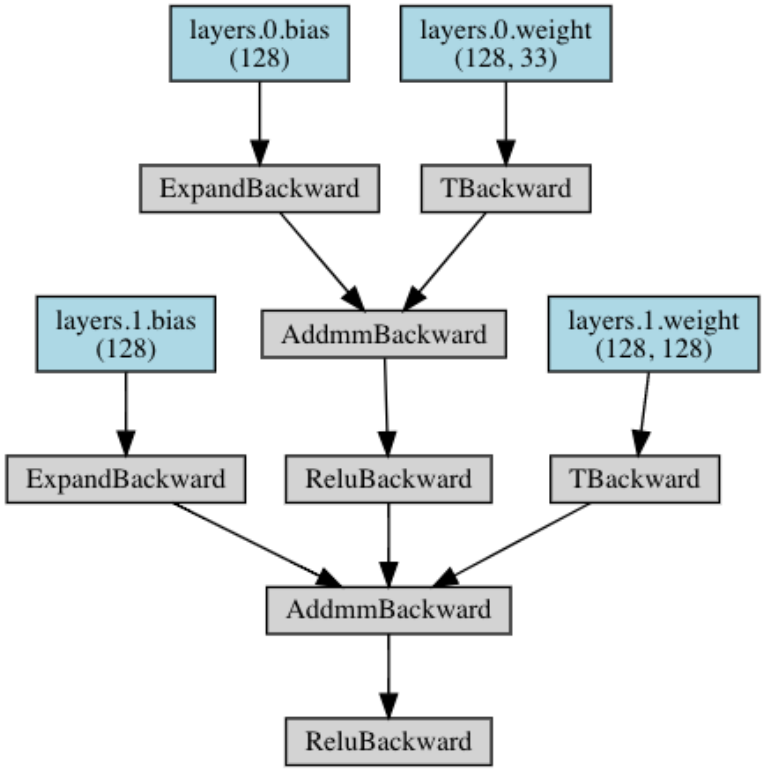
DDPG Critic Network

Please see the following page.

DDPG Critic Network



PPO Actor & Critic (Same structure but different number of outputs)



References:

http://www.scholarpedia.org/article/Reinforcement_learning

http://www.scholarpedia.org/article/Policy_gradient_methods

https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

http://proceedings.mlr.press/v32/silver14.pdf

https://github.com/ShangtongZhang/DeepRL

https://blog.openai.com/openai-baselines-ppo/

https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Deep-Reinforcement-Learning-Through-Policy-Optimization

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md