Reinforcement learning project 2: Continuous Control

Reza Doobary

August 8, 2019

Contents

---

1 Introduction

In this short report we provide model implementation details and analysis on the reinforcement learning project entitled 'Continuous control' as part of the Udacity course on deep reinforcement learning.

Whilst full task details of the agent, the environment and success criteria have been outlined in the accompanying README.md, we repeat these for completeness. The agent in this case is a simulated double jointed arm whose essential task is to follow a target within its reach. The state space of the agent is a 33 dimensional real float space and contains the position, rotation, velocity, and angular velocities of the arm. The action space is 4 dimensional float space and corresponds to the torque applied on the two joints. The reward strategy is that for every step that the agent remains in contact

with the target, it obtains a reward of +0.1, 0 otherwise. The task is deemed solved if the agent gets an average score of +30 over 100 consecutive episodes.

NOTE : In this project, significant effort has been placed into reporting on the theoretical understanding of the models implemented. To go straight to the actual implementation of the algorithms, please go to section 3. For extremely quick results, with nothing more than the hyper-parameters given please go to section 4.

2 General concepts

In this section, an overview of some of the main concepts used in this project are provided.

2.1 Policy gradient methodology

This entire project is based around the idea of policy gradient methodologies, by which we mean to directly optimise for a policy rather than an optimal action. One of the main motivations for such methodologies is that it is rather more straight forward to generalise to continuous actions. This will be the main focus and explained shortly.

As an example, consider the double jointed arm in this project. However, rather than having the continuous four-dimensional action space corresponding to the torques of the two joints, we instead fix twenty positions the arm can take. We thus can be define a neural network such as:

$$\underset{\text{input layer}}{\text{33 dim'l}} \Longrightarrow \underset{\text{NETWORK}}{\text{NUERAL}} \Longrightarrow \underset{\text{output layer}}{\text{20 dim'l}}, \tag{1}$$

whereby the twenty dim'l output is in fact a discrete probability distribution, thus a natural output layer is the Softmax function. In effect, whilst Q-learning type algorithms would like to model the state-value or action-value function with a neural network, here we are modelling the conditional probability distribution with a neural network, symbolically $\mathbb{P}(\text{action}|\text{state})$.

Maintaining complete generality, we can devise an algorithm which will optimise the network to give the distribution which maximises the reward. However, in order to deduce the next action we must sample from a probability distribution, thus a natural exploration behaviour is built-in[1].

Let us now consider the case at hand, namely when the action space is truly continuous. In this case, we cannot use the Softmax function as there infinitely many possible action choices. Instead, the neural network will model the continuous probability distribution parameters and the sampling is done on the continuously re-parametrised probability distributions with each iteration of the algorithm. More concretely, let us consider the an agent whose action space is m-dimensional. We propose that the probability distribution associated the action space is a joint normal distribution among the action elements. The parameters of such a distribution are the m means, which we could denote $\mu_i \forall\, i \in [1, m]$, but also the joint symmetric covariance matrix Σ which is $\frac{m(m+1)}{2}$. The model should therefore have an output layer with $m + \frac{m(m+1)}{2}$ variables. This is rather a lot, and a simplification which we will employ in our four dimensional case is to decouple the distribution (thus each element of the output action is independent). Furthermore, we can also set the standard deviation of each action to one.

In the next subsection we outline our very first concrete algorithm which we have employed the use of in this project.

2.2 Vanilla policy gradient (VPG)

2.2.1 The basic algorithm

VPG was studied in [2] and the goal of VPG is rather simple. We would like to maximise the expected return. Fleshing the statement out a little, we can generate many 'trajectories', namely many ways in

[1] This is somewhat of an avatar to the ϵ-learning prescription of Q-learning algorithms.

which the agent reacts to the environment given an action and reward structure. So we can generate the τ-many tuples (where τ is the number of trajectories):

$$(s_0, a_0, s_1, a_1, s_2, a_2, \ldots, s_H, a_H, s_{H+1}), \tag{2}$$

and the reward structure is simply:

$$R(\tau) = \sum_{j=0}^{H+1} r_j, \tag{3}$$

where the r_i are generated from the agent interaction with the environment. We can define a probability measure $\mathbb{P}(\theta)$ which is parametrised by some model, in this case a neural network (but not necessarily so). Then, we are interested in maximising the expected return $\mathbb{E}_\theta[R]$ whose estimator is given by:

$$\sum_\tau \mathbb{P}(\tau, \theta) R(\tau). \tag{4}$$

Using standard optimisation techniques, we could very well produce an algorithm that will maximise the estimator above. However, there are some simplifications and further enhancements that will help the optimisation. Writing, $U(\theta) = \mathbb{E}[R]$ we are interested in the iteration step:

$$\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta). \tag{5}$$

Given trajectories of length H and a sample size of m trajectories, we have

$$\nabla_\theta U(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi(a_t^{(i)} | s_t^{(i)}, \theta) R(\tau^{(i)}), \tag{6}$$

the derivation of which can be found in appendix A.

It is common to refer to the surrogate part of 6 as

$$L_{\text{surrogate}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R(\tau^{(i)}), \tag{7}$$

such that the estimator (often referred to as $\hat{g}(\theta)$) can be written as $\hat{g}(\theta) = \nabla_\theta L_{\text{surrogate}}(\theta)$.

2.2.2 Credit assignment

A modification to the algorithm to ensure efficiency is if we consider expected future rewards only. The current state of the reward structure is $R(\tau) = \sum_{j=0}^{H+1} r_j$, which can be rewritten as $R(\tau)_{\text{past}} + R(\tau)_{\text{future}} = \sum_{j=0}^{t-1} r_j + \sum_{j=t}^{H} r_j$. We can then exclude the past reward and only take the future reward into account, thus equation 6 becomes:

$$\nabla_\theta U(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R_{\text{future}}(\tau^{(i)}). \tag{8}$$

We now have

$$L_{\text{surrogate}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R_{\text{future}}(\tau^{(i)}), \tag{9}$$

is often identified with a surrogate function, such that $\hat{g}(\theta) = \nabla_\theta L_{\text{surrogate}}$.

It is worth noting that since $G_t = \mathbb{E}[R_t | s_t, a_t]$, and R_{future} is the future reward structure subject to a known starting state and action, we can identify R_{future} with G_t. Thus

$$\nabla_\theta U(\theta) = \mathbb{E}\left[\sum_{t=0}^{H} \nabla_\theta \log(\pi_\theta(a_t, s_t)) G_t \right]. \tag{10}$$

The algorithm becomes

Randomly choose θ
for i in range $0 \ldots$ number of trajectory iterations **do**
  Collect trajectories based on π_θ
  $\theta \leftarrow \theta + \alpha \nabla_\theta L_{\text{surrogate}}(\theta)$
end for

2.3 Proximity policy optimisation (PPO)

At the heart of the PPO (studied in [3]) algorithm is importance sampling. Importance sampling is commonly associated with vanilla Monte Carlo and is used to reduce the variance. We give a brief overview of importance sampling.

2.3.1 Importance sampling

Consider a well-defined probability space $\mathbb{P}$. An expectation value of some function $f(X)$ [2] on this space is define to be:

$$\mathbb{E}[f] = \int_{\mathcal{D}} f(X) d\mathbb{P}(X), \tag{11}$$

where $\mathcal{D}$ is the domain space of the probability space.

Suppose we have a different probability space $\tilde{\mathbb{P}}$ which we would like to use instead of the one originally offered. We can do this by employing the use of the Radon Nikodym theorem, and the associate Radon Nikodym derivative $Z(X)$. Essentially, under certain formal conditions we can write

$$\int_{\mathcal{D}} f(X) \frac{d\mathbb{P}(X)}{d\tilde{\mathbb{P}}(X)} d\tilde{\mathbb{P}}(X) = \int_{\tilde{\mathcal{D}}} f(X) Z(X) d\tilde{\mathbb{P}}(X). \tag{12}$$

Therefore the following statement is achieved:

$$\mathbb{E}[f] = \tilde{\mathbb{E}}[fZ]. \tag{13}$$

A common situation one may find themselves is to appropriately choose Z in such a way as to make the expectation value as easy to compute as possible, in some cases reducing a necessarily numerical evaluation into an analytic one.

2.3.2 The basic algorithm

Armed with an understanding of importance sampling, we can apply this to our policy gradient algorithm. The basic idea is to use a policy π_θ to generate a dataset of trajectories, then use the data to generate probabilities and optimise the neural network subject to these. This is all as opposed to taking a trajectory learning and throwing it away. Let us make this concrete.

We begin with the non-estimated derivative of $U(\theta)$ which is equation **??** as a starting point.

$$\nabla_\theta U(\theta) = \sum_\tau \mathbb{P}(\tau, \theta) R(\tau) \sum_t \nabla_\theta \log\left(\pi_\theta\left(a_t^\tau | s_t^\tau\right)\right) \to \sum_\tau \mathbb{P}(\tau, \theta) \sum_t \frac{\nabla_\theta \pi_\theta\left(a_t^\tau | s_t^\tau\right)}{\pi_\theta\left(a_t^\tau | s_t^\tau\right)} R_{\text{future}}(\tau), \tag{14}$$

Now we make use of importance sampling by doing

$$\nabla_\theta U(\theta) = \sum_\tau \mathbb{P}(\tau, \theta') \frac{\mathbb{P}(\tau, \theta)}{\mathbb{P}(\tau, \theta')} \sum_t \frac{\nabla_\theta \pi_\theta\left(a_t^\tau | s_t^\tau\right)}{\pi_\theta\left(a_t^\tau | s_t^\tau\right)} R_{\text{future}}(\tau). \tag{15}$$

Using this together with the definition of the probabilities of trajectories, we get

[2] We define $f(X)$ such that X takes value in the domain of the probability space $\mathbb{P}$

$$\nabla_\theta U(\theta) = \sum_\tau \mathbb{P}(\tau,\theta')\nabla_\theta L_{\text{surrogate}}(\theta',\theta) = \sum_\tau \mathbb{P}(\tau,\theta')\sum_t \frac{\nabla_\theta \pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_{\theta'}\left(a_t^\tau|s_t^\tau\right)} R_{\text{future}}(\tau), \tag{16}$$

from which a derivation of is found in appendix B. The definition of the surrogate function this time is therefore

$$L_{\text{surrogate}}(\theta',\theta) = \sum_t \frac{\pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_{\theta'}\left(a_t^\tau|s_t^\tau\right)} R_{\text{future}}(\tau) \tag{17}$$

By the similar practicality arguments, an estimator for this function is

$$\sum_\tau \mathbb{P}(\tau,\theta')\nabla_\theta L_{\text{surrogate}}(\theta',\theta) = \nabla_\theta \frac{1}{m}\sum_{i=0}^{m}\sum_t \frac{\pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)}{\pi_{\theta'}\left(a_t^{(i)}|s_t^{(i)}\right)} R_{\text{future}}(\tau^{(i)}), \tag{18}$$

Keeping with the idea if importance sampling, the usual problem is suitably choose the second probability space $\mathbb{P}(\tau,\theta')$, which in effect is choosing a different neural network parametrisation θ'. Another aspect to keep in mind is that the point of importance sampling is to choose the second probability space to make the algorithm more efficient[3]. With this in mind, the basic PPO algorithm is as follows:

Initialise $\theta' = \theta$
for i in range $0 \ldots$ number of trajectory iterations **do**
  Collect trajectories based on π_θ
  for j in range $0 \ldots$ number of iterations **do**
    $\theta' \leftarrow \theta' + \alpha\nabla_{\theta'} L_{\text{surrogate}}(\theta',\theta)$
  end for
  Set $\theta = \theta'$
end for

The algorithm thus uses policy π_θ to generate the trajectories, uses π'_θ to train the model based on π_θ before setting $\theta = \theta'$ and regenerating new trajectories based on the recently trained parameters θ.

2.3.3 Clipping

One efficiency improvement on PPO, is clipping. It is important to keep mind that the surrogate function that is used for training, is really only an estimator for the derivative of the true return $\nabla_\theta U(\theta)$ we are trying to maximise. Thus, whilst a good estimator it is not clear that it will not miss the optimal point on the parameters manifold and go well-past the optimal policy.

Recall that the surrogate function for PPO, comes about because we take the conditional probabilities π_θ and π'_θ to be equivalent. The clipping simply imposes that the new θ' isn't too far away from the previous one. This is done by introducing the function clip_ϵ which has the following definition:

$$\begin{aligned}\text{clip}_\epsilon(x) &= 1+\epsilon \text{ if } x > 1+\epsilon,\\ &= x \text{ if } 1-\epsilon \le x \le 1+\epsilon,\\ &= 1-\epsilon \text{ if } x < 1-\epsilon.\end{aligned} \tag{19}$$

We then take

$$L^{\text{clip}}_{\text{surrogate}}(\theta',\theta) = \sum_t \min\left\{\frac{\pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_{\theta'}\left(a_t^\tau|s_t^\tau\right)} R_{\text{future}}(\tau), \text{clip}_\epsilon\left(\frac{\pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_{\theta'}\left(a_t^\tau|s_t^\tau\right)} R_{\text{future}}(\tau)\right)\right\}, \tag{20}$$

[3]Note, that for vanilla Monte Carlo algorithms this is like choose a second probability space which is *close* to the estimator is some way.

as the new surrogate function. Here, we wish to bound the resulting new surrogate function from the top but allow the minimum to be quite small. In this way we have $L^{\text{clip}}_{\text{surrogate}}(\theta', \theta) \leq L_{\text{surrogate}}$, which retains a soft conservation. All this means, is that we small changes remains small and large changes are bounded. REFERENCE!!!

For completeness the algorithm now becomes

Initialise $\theta' = \theta$
for i in range $0 \ldots$ number of trajectory iterations **do**
  Collect trajectories based on π_θ
  for j in range $0 \ldots$ number of iterations **do**
    $\theta' \leftarrow \theta' + \alpha \nabla_{\theta'} L^{\text{clip}}_{\text{surrogate}}(\theta', \theta)$
  end for
  Set $\theta = \theta'$
end for

2.4 Generalised Advantage Estimation (GAE)

A common problem in almost any statistical estimation problem, we would find ourselves concerned with biases and variances. Low bias would imply that the estimator misses the return by some margin, whilst high variance would imply a very large number of iteration required for the algorithm to converge. The best possible outcome would be with high bias and low variance. There are two distinct ways to estimate the returns G_t of an agent. Firstly, we could consider the Monte Carlo approach, namely to consider many trajectories and taking the average of the corresponding returns. This method has low bias, but higher variance depending on the number of trajectories chosen. A second method involves using the temporal difference (TD) estimates, which makes use of the estimated second order returns. This has lower variance but owing to the second order estimate has higher bias. The goal of GAE is to lower this bias variance trade off. This was studied in [1].

In [1], the authors consider the advantage function (as opposed the return) which is known to have lower variance. We therefore consider this also below.

The definition of the advantage function is $A(s_t, a_t) = \mathbb{E}\left[Q(s_t, a_t) - V(s_t)\right]$. However, $r_t = \gamma V(s_{t+1})$ is an estimator for the action-value function $Q(s_t, a_t)$, thus the advantage function can be estimated by the TD estimator:

$$A(s_t, a_t) = \mathbb{E}\left[r_t + \gamma V(S_{t+1}) - V(S_t)\right], \tag{21}$$

which we denote by a hat. Now, whilst r_t is observed from the environments reaction to the agent, the value function are estimated[4]. This mean, we can introduce levels of the estimated advantage function that are observed (akin to monte carlo) or estimated (akin to TD estimation a la bellman equations). We can write the advantage estimators in an order by order way[5] (denoting TD^V_t as the TD estimator)

$$\begin{aligned}
\hat{A}^{(1)}_t &= r_t + \gamma V(S_{t+1}) - V(S_t) = \text{TD}^V_t, \\
\hat{A}^{(2)}_t &= r_t + \gamma r_{t+1} + \gamma^2 V(S_{t+2}) - V(S_t) = \text{TD}^V_t + \gamma \text{TD}^V_{t+1}, \\
\hat{A}^{(3)}_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+3} + \gamma^3 V(S_{t+3}) - V(S_t) = \text{TD}^V + \gamma \text{TD}^V + \gamma^2 \text{TD}^V,
\end{aligned} \tag{22}$$

we can therefore write

$$\hat{A}^{(k)}_t = \sum_{l=0}^{k-1} \gamma^l \text{TD}^V_{t+l}. \tag{23}$$

[4]For example, for our implementation it will come from the deep neural network

[5]Note that for each k, $\hat{A}^{(k)}_t$ is in of itself a valid estimator for the advantage function.

The GAE(γ, λ) is defined to be the exponentially-weighted average:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = (1-\lambda)\sum_{l=1}^{\infty}\lambda^{l-1}\hat{A}_t^{(l)}. \tag{24}$$

After some manipulation as detailed in C, we get

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l\,\mathrm{TD}_{t+l}^{V}. \tag{25}$$

The reason this is so powerful is due to the introduction of the hyper-parameter λ,for which when $\lambda = 0$, we obtain the first order TD which is the 'high variance' estimator. If we take $\lambda = 1$, we retrieve the sum over all discounted TD estimates, which is nothing more than $\sum_{l=0}^{\infty}\gamma^l r_{t+l} - V(s_t)$, which is the 'high variance' estimator. Therefore, λ is a smoothing parameter between these two extremes and attempts to allow us to capture the best of both estimators.

As a point of clarity, we will employ the use of GAE with the PPO model, thus we want to evaluate

$$\mathbb{E}\left[\sum_t \frac{\nabla_\theta \pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_{\theta'}\left(a_t^\tau|s_t^\tau\right)}\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}(\tau)\right] \tag{26}$$

2.5 Actor-Critic methods and asynchronism

The Actor-Critic methodology is a group of model architectures and concepts in its own right. It is essentially when the entire model has two submodels:

1. An actor : The sub-model associated with the action that the agent will actually take. In the Poligy gradient context this is the model responsible for producing the action distribution of the agent.

2. A critic : The sub-model associated with an output that will measure the relative improvement of performance of the agent. For example, a sub-model responsible for producing the value function.

Let's return to VPG. As noted in equation 10, we are interested in the quantity $\sim \mathbb{E}\left[\sum_{t=0}^{H}\nabla_\theta \log\left(\pi_\theta(s_t,a_t)\right)G_t\right]$. However, the expectation can be decomposed into two pieces, that which occurs at time t and before, and that which occurs after. Namely,

$$\mathbb{E}\left[\sum_{t=0}^{H}\nabla_\theta \log\left(\pi_\theta(s_t,a_t)\right)G_t\right] \sim \sum_{t=0}^{H}\mathbb{E}_{s_0,a_0,dots,s_t,a_t}\left[\nabla_\theta \log\left(\pi_\theta(s_t,a_t)\right)\right]\underbrace{\mathbb{E}_{s_{t+1},a_{t+1},r_{t+1},\ldots,s_H,a_H,r_H}\left[G_t\right]}_{Q(s_t,a_t)}. \tag{27}$$

Now, this gives $\mathbb{E}\left[\sum_{t=0}^{H}\nabla_\theta \log\left(\pi_\theta(s_t,a_t)\right)Q(s_t,a_t)\right]$. Which is highly suggestive of introducing a second model.

For instance, we could introduce a new neural network [6], and model the action-value function $Q(s,a)$ from it. We could think of the expectation more as $\mathbb{E}\left[\sum_{t=0}^{H}\nabla_\theta \log\left(\pi_\theta(s_t,a_t)\right)Q_\psi(s_t,a_t)\right]$, where θ parametrises the model associated to the action distribution (which is the actor) whilst ψ parametrises the model associated to the action-value function (which is the critic). The basic update rule associated to the actor is $\theta \leftarrow \theta + \alpha Q_\psi(s,t)\nabla_\theta \log(\pi_\theta(s,a))$, then the update of Q_ψ occurs subject to the standard TD estimate $\psi \leftarrow \psi + \alpha \mathrm{TD}_t \nabla_\psi Q_\psi(s,t)$. We thus see that for a low value choice the critic effectively penalises the corresponding direction ins the action distribution, thus scrutinising choices made.

[6] However, we could choose any model.

2.5.1 A2C

A2C is a term used for the advantage actor-critic model where rather that consider the expected return, we attempt to compute the expected advantage function. The basic idea here is that the function $\mathbb{E}\left[\sum_{t=0}^{H} \nabla_\theta \log\left(\pi_\theta(s_t, a_t)\right) Q_\psi(s_t, a_t)\right]$ has high variance and this can be lowered by introducing a baseline function $\mathbb{E}\left[\sum_{t=0}^{H} \nabla_\theta \log\left(\pi_\theta(s_t, a_t)\right) \left(Q_\psi(s_t, a_t) - b_\psi(s_t, a_t)\right)\right]$.

A legitimate choice is the advantage function, and a complete discussion is given in section 2.4, particularly in the context of VPG, the estimator is

$$\mathbb{E}\left[\sum_{t=0}^{H} \nabla_\theta \log(\pi_\theta(s_t, a_t)) \left(r_t + \gamma V_\psi(s_{t+1}) - V_\psi(s_t)\right)\right]. \tag{28}$$

The algorithm would then take the form (where the surrogate function is the same as 9 but $r_t + \gamma V_\psi(s_{t+1}) - V_\psi(s_t)$ replaces $R(\tau)$):

Randomly choose θ and ψ
for i in range $0 \ldots$ number of trajectory iterations **do**
  Collect trajectories based on π_θ
  for j in range $0 \ldots$ number of iterations **do**
    $\theta' \leftarrow \theta' + \alpha \mathrm{TD}_t \nabla_{\theta'} L_{\text{surrogate}}(\theta)$
    $\psi' \leftarrow \psi' + \alpha \mathrm{TD}_t \nabla_\psi V$
  end for
end for

2.5.2 A3C

The *A3C* algorithm is essentially the same as *A2C* but done with parallel agents. The concepts was introduced in [5] and essentially allows for a larger number of data inputs to be applied to the training algorithm. Put simply, we have N_a-many agents, all of which interacting with the environment is the same way and extracting action decisions and rewards for the same policy. It is not until the very last step before backward propagation, that an average among all the agents is taken.

3 Implementation and Analysis

In this section, we finally discuss key implementation details and the results that follow. In this project, we have have implemented two models.

1. VPG : The standard VPG model with twenty asynchronous agents,
2. PPO : PPO model with GAE, clipping and twenty asynchronous agents.

The actual parallel programming that comes with synchronous is handled by the unity engine and requires not further work from us aside from the basic handling of the data.

In all occasions where a neural network is used, for example in the VPG model we have the actor model and in the PPO model we employ the use of the Actor-Critic model. We always have only a single hidden layer which is 512 dimensional.

3.1 VPG model

In our implementation of the VPG model, we consider the most basic of models. We collect the data using a trajectory class found in `trajectories.py`, namely the class `trajectories_returns` which inherits from `trajectories_basic`. The key lines of `trajectories_basic` are [7] the contents of the

[7] Note that we have omitted parts of the code not pertinent to this discussion.

following for-loop from the _collect_basic method:

```
for _ in range(self.trajectory_length):
    actions, log_probs, _ = self.network(states)

    # given the action we take a step in the environment
    next_states, rewards, is_done = self.env__.step(actions.cpu().detach().numpy())

    terminals = np.array(is_done, dtype=int)
    self.all_rewards += rewards

    for i, terminal in enumerate(terminals):
        if terminals[i]:
            self.episode_rewards.append(self.all_rewards[i])
            self.all_rewards[i] = 0

    # append to trajectory
    trajectory.append([states, actions, log_probs, rewards, 1 - terminals])
    states = next_states
```

The trajectory length chosen for VPG 2048 dimensional. The trajectory array is then passed through to a processing step, found in the class named `trajectories_returns`, where the discounted returns are computed. The key for-loop is found in the `_collect_returns` method.

```
processed_trajectory = [None] * (len(trajectory) - 1)
for i in reversed(range(len(trajectory) - 1)):
    states, actions, log_probs, rewards, terminals = trajectory[i]
    terminals = torch.Tensor(terminals).unsqueeze(1)
    rewards = torch.Tensor(rewards).unsqueeze(1)
    actions = torch.Tensor(actions)
    states = torch.Tensor(states)

    if i == len(trajectory) - 2:
        returns = rewards
    returns = rewards + self.discount_rate * terminals * returns

    processed_trajectory[i] = [states, actions, log_probs, returns]
```

At this point the array `processed_trajectory` is essentially a list of lists. The inner-most list is effectively a single processed trajectory containing, the states observed, the actions taken, the associated logarithmic probabilities as well the returns. The outer-most list is a list of these collected trajectories. We are now ready to pass this data onto training. As the agent steps through the environment it collects trajectories and learns from them via the `VanillaPolicyGradientOptimisation` class in `learner.py`. The key set of lines is given by the `learn` method:

```
# Collect trajectories
processed_trajectories, states, all_rewards, episode_rewards =
    self.trajectory.collect(states)
states_, all_rewards_, episode_rewards_ = states, all_rewards, episode_rewards
states, actions, log_probs_old, returns = map(lambda x: torch.cat(x, dim=0),
    zip(*processed_trajectories))

# write up a surrogate function
surrogate = log_probs_old * returns
```

```
policy_loss = -surrogate.mean()

# optimise
self.optmiser.zero_grad()
policy_loss.backward()
self.optmiser.step()
del policy_loss
```

The model (i.e. the deep neural network) is defined by `PolicyMod` in `model.py`.

3.1.1 Results

The neural network used to model the policy distribution has a single 512 dimensional hidden layer. This means there is an input layer of 33 dimensions, followed by the 512 dimensional layer and finally a 4 dimensional output layer corresponding to each action. The actions themselves are viewed as means of decoupled Gaussian distributions, each with a standard deviation of 1. The output gate of this basic model is the tanh function.

In the 300 episodes, each with 2048 collected trajectories, the task does not complete. In any case, we can provide the plot in figure 1. We notice the general upward trend towards solving the task eventually, however it would simply take too long[8].

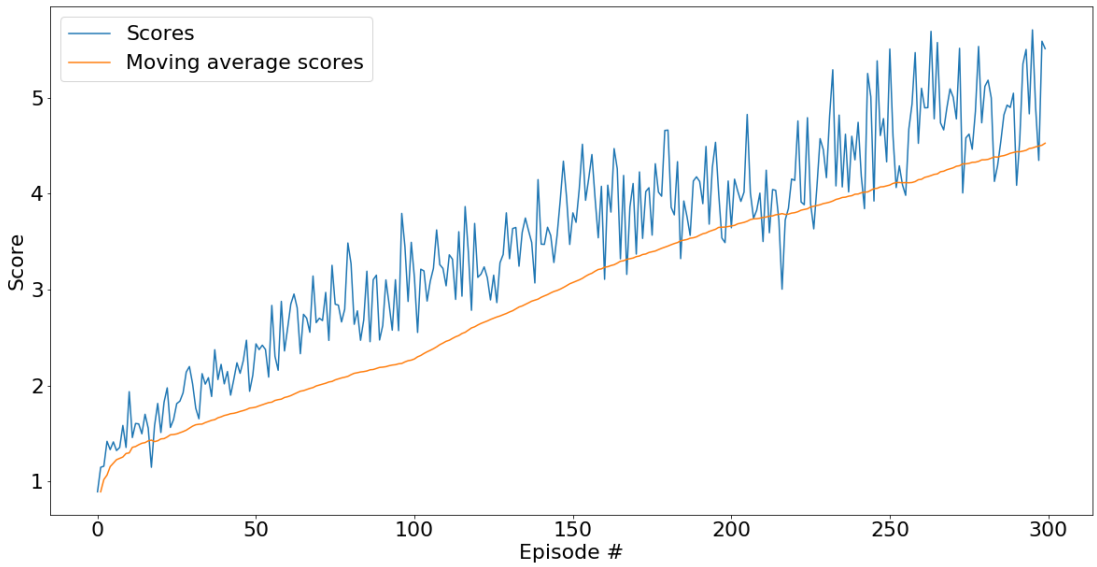

Figure 1: The VPG does not solve the task in 300 episodes.

3.2 PPO model

For our PPO model, we have implemented both GAE and clipping, wof which implementation details will be described below. The data collection for our PPO algorithm is much the same as for VPG,

[8]Employing the use of remote GPUs was not used, but it probably could be solved with a GPU.

only that once we have collected the basic data, it is processed differently. The key lines are found in the _collect_advantages method of the trajectories_advantage class:

```
processed_trajectory = [None] * (len(trajectory) - 1)
advantages = torch.Tensor(np.zeros((self.number_of_agents, 1)))
returns = trajectory[-1][1]
for i in reversed(range(len(trajectory) - 1)):
    states, value, actions, log_probs, rewards, terminals = trajectory[i]
    terminals = torch.Tensor(terminals).unsqueeze(1)
    rewards = torch.Tensor(rewards).unsqueeze(1)
    actions = torch.Tensor(actions)
    states = torch.Tensor(states)
    next_value = trajectory[i + 1][1]
    returns = rewards + self.discount_rate * terminals * returns

    td_error = rewards + self.discount_rate * terminals * next_value - value

    # GAE computation
    advantages = advantages * self.lambda_ * self.discount_rate * terminals + td_error
    processed_trajectory[i] = [states, actions, log_probs, returns, advantages]
```

We see that as with VPG, we have collected the discounted returns. However, in addition we also have the line:

`advantages = advantages× lambda× discount_rate × terminals + td_error`

We can recall that formula $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \mathrm{TD}_{t+l}^V$, is being implemented here by being continuously multiplied up by $\lambda\gamma$ as in the code. In this way,w e have captured the formula for each time step at any point in any episode. Thus, the GAE is implemented here. In particular it is implemented with $\lambda = 0.95$.

Next, the processed data is now entered into the for-loop (which is the `learn` method of the `ProximalPolicyOptimisation` class found in `learner.py`:

```
for _ in range(self.number_of_epochs):
    # set up batcher
    self.batcher.get_iter()
    while not self.batcher.end():
        # sample from space of data
        sampled_states, sampled_actions, sampled_log_probs_old, sampled_returns,
            sampled_advantages = \
            self.batcher.sample(states, actions, log_probs_old, returns, advantages)

        # detach data so we can compute things
        sampled_states, sampled_actions, sampled_log_probs_old, sampled_returns,
            sampled_advantages = \
            sampled_states.detach(), sampled_actions.detach(), sampled_log_probs_old.detach(),
                sampled_returns.detach(), sampled_advantages.detach()

        # re-forward pass through network to find log_probs and values
        #checkpoint 1
        _, log_probs, values = self.network(sampled_states, sampled_actions)
        ratio = (log_probs - sampled_log_probs_old).exp()
        obj = ratio * sampled_advantages


        #checkpoint 2
        if self.ppo_clip:
```

```
        obj_clipped = ratio.clamp(1.0 - self.ppo_clip, 1.0 + self.ppo_clip) *
            sampled_advantages
        policy_loss = -torch.min(obj, obj_clipped).mean(0)
    else:
        policy_loss = -obj.mean(0)

    value_loss = 0.5 * (sampled_returns - values).pow(2).mean()

    self.optmiser.zero_grad()
    # perform optimisation of the sum of policy_loss and value_loss - optimising the same
        parameters.

    #checkpoint 3
    (policy_loss + value_loss).backward()
    if self.gradient_clip: nn.utils.clip_grad_norm_(self.network.parameters(),
        self.gradient_clip)
    self.optmiser.step()
    del policy_loss
```

We have given checkpoints at three noteworthy places in the code. At checkpoint 1, the model is used again to find new logarithmic probabilities and values. This represents the difference between the θ and θ' described earlier in the importance sampling section. The ration of the two probabilities are computed and we move onto checkpoint 2. At checkpoint 2, the clipping occurs. At checkpoint 3, the policy step occurs, as one can see we optimise these together.

3.2.1 Results

In the PPO algorithm, we employed the use of the actor-critic model. Thus, we have two simultaneous models that we optimise together. The actor is the same model as the VPG case, namely with a single hidden layer of 512 dimensions. The critic produces the value function for each state, which in this context has only one hidden layer of 512 dimensions. The critic model has an output layer of 1 dimensions instead of the 4 of the actor.

The task **required** 203 **episodes** to gain an average reward of at least 30. Each episode contained the sampling of 2048 trajectories with 10 epochs, with each epoch re-sampling a batch of data the training the model.

A plot of the scores together with the moving average is plotted in figure 2.

4 Quick Results

The neural network used was the basic actor model.

4.1 VPG

Parameter	**Value**
DNN - input layer	33
DNN - hidden layer	512
DNN - output layer	4
Adam optimiser - learning rate	3e-4
Adam optimiser - epsilon	1e-5
Number of learning epochs	10
Trajectory length	2048
Discount Rate	0.99

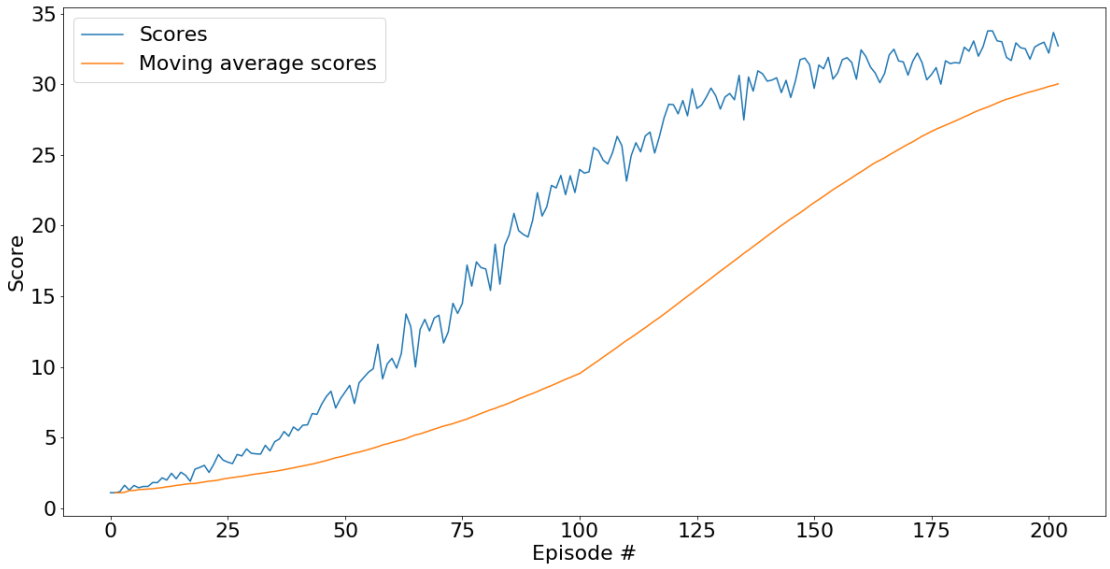

Figure 2: The PPO plot being solved in 203 episodes.

The model was not solved in 300 episodes and was abandoned. The plot of the training can be found in figure 1.

4.2 PPO

The neural network used was the actor-critic model, with the critic given by the value function.

Parameter	**Value**
DNN - input layer	33
DNN - hidden layer	512
DNN - output layer	4
Adam optimiser - learning rate	3e-4
Adam optimiser - epsilon	1e-5
Number of learning epochs	10
Trajectory length	2048
Discount Rate	0.99
Clipping	0.2
Gradient Clipping	5
GAE λ	0.95

The model was solved in 203 episodes. The plot of the training can be found in figure 2.

5 Conclusion and further work

The main conclusion of this project was that the asynchronous 20 agent environment requires 203 episodes to be solved by an agent following an actor-critic model with a PPO update rule, together

with clipping and GAE.

For further work, it would have been interesting to do the following:

1. To consider the DDPG model [4]. This involves applying the DQN model with continuous action policy.
2. To consider more intermediate models in this project. We studied the standard VPG model and then straight to a fairly efficient PO model. It would be interesting to consider PPO without GAE or VPG with the Actor-Critic method, and see at what point significant effeciency occurs.

Appendices

A Derivation of equation 6

We begin with $\nabla_\theta U(\theta) = \nabla_\theta \sum_\tau \mathbb{P}(\tau,\theta)R(\tau)$. Now, we bring the derivative in and multiply by the smart factor of 1:

$$= \sum_\tau \frac{\mathbb{P}(\tau,\theta)}{\mathbb{P}(\tau,\theta)} \nabla_\theta \mathbb{P}(\tau,\theta)R(\tau) = \sum_\tau \mathbb{P}(\tau,\theta)\frac{\nabla_\theta \mathbb{P}(\tau,\theta)}{\mathbb{P}(\tau,\theta)}R(\tau) = \sum_\tau \mathbb{P}(\tau,\theta)\nabla_\theta \log(\mathbb{P}(\tau,\theta))R(\tau). \tag{29}$$

The last expression is not trivial and is in principle equivalent to $\mathbb{E}_\theta\left[\nabla_\theta \log(\mathbb{P}(\theta))R\right]$.

As a result, from a practical point of view it makes sense to consider a sample of trajectories leading to the estimator of the expectation value

$$\frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log(\mathbb{P}(\tau^{(i)},\theta)R(\tau^{(i)}). \tag{30}$$

Now note that

$$\mathbb{P}(\tau^{(i)},\theta) = \prod_{t=0}^{H} \mathbb{P}(s_{t+1}^{(i)}|s_t^{(i)},a_t^{(i)})\pi_\theta(a_t^{(i)}|s_t^{(i)}), \tag{31}$$

from which the usual log identity follows:

$$\log\left(\prod_{t=0}^{H} \mathbb{P}(s_{t+1}^{(i)}|s_t^{(i)},a_t^{(i)})\pi_\theta(a_t^{(i)}|s_t^{(i)})\right) = \sum_{t=0}^{H}\left(\log\left(\mathbb{P}(s_{t+1}^{(i)}|s_t^{(i)},a_t^{(i)}\right) + \log\left(\pi_\theta(a_t^{(i)}|s_t^{(i)})\right)\right), \tag{32}$$

in which it is clear that the first term has no dependence on θ, whilst the second one does. From here the result follows:

$$\nabla_\theta U(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau^{(i)}), \tag{33}$$

B Derivation of PPO surrogate function (equation 16)

Starting with equation 15, namely

$$\nabla_\theta U(\theta) = \sum_\tau \mathbb{P}(\tau,\theta')\frac{\mathbb{P}(\tau,\theta)}{\mathbb{P}(\tau,\theta')}\sum_t \frac{\nabla_\theta \pi_\theta\left(a_t^\tau|s_t^\tau\right)}{\pi_\theta\left(a_t^\tau|s_t^\tau\right)} R_{\text{future}}(\tau). \tag{34}$$

We can employ the use of the relation

$$\mathbb{P}(\tau,\theta) = \prod_{t=0}^{H} \mathbb{P}(s_{t+1}^\tau|s_t^{(i)},a_t^\tau)\pi_\theta(a_t^\tau|s_t^\tau), \tag{35}$$

we find that equation 15 is equivalent to

$$\begin{aligned}\sum_{\tau}\mathbb{P}(\tau,\theta')\frac{\prod_{t=0}^{H}\mathbb{P}(s_{t+1}^{\tau}|s_t^{\tau},a_t^{\tau})\pi_\theta(a_t^{\tau}|s_t^{\tau})}{\prod_{t=0}^{H}\mathbb{P}(s_{t+1}^{\tau}|s_t^{\tau},a_t^{\tau})\pi_\theta(a_t^{\tau}|s_t^{\tau})}\sum_t\frac{\nabla_\theta\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)}{\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)}R_{\text{future}}(\tau)\\ =\sum_{\tau}\mathbb{P}(\tau,\theta')\frac{\prod_{t=0}^{H}\pi_\theta(a_t^{\tau}|s_t^{\tau})}{\prod_{t=0}^{H}\pi_\theta(a_t^{\tau}|s_t^{\tau})}\sum_t\frac{\nabla_\theta\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)}{\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)}R_{\text{future}}(\tau)\end{aligned}. \tag{36}$$

The denominator $\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)$ can be cancelled with the new numerator and so long the a parametrisations θ and θ' are close, we can approximate the remaining terms to unity, with a remaining term $\pi_{\theta'}\left(a_t^{\tau}|s_t^{\tau}\right)$. This leaves us with 16, which is

$$\nabla_\theta U(\theta)=\sum_{\tau}\mathbb{P}(\tau,\theta')L_{\text{surrogate}}(\theta',\theta)=\sum_{\tau}\mathbb{P}(\tau,\theta')\sum_t\frac{\nabla_\theta\pi_\theta\left(a_t^{\tau}|s_t^{\tau}\right)}{\pi_{\theta'}\left(a_t^{\tau}|s_t^{\tau}\right)}R_{\text{future}}(\tau). \tag{37}$$

C Derivation of GAE (equation 25)

Recall that the GAE(γ,λ) is defined to be the exponentially-weighted average:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)}=(1-\lambda)\sum_{l=1}^{\infty}\lambda^{l-1}\hat{A}_t^{(l)}, \tag{38}$$

and we have that

$$\hat{A}_t^{(k)}=\sum_{l=0}^{k-1}\gamma^l\text{TD}_{t+l}^V, \tag{39}$$

So,

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)}=(1-\lambda)\sum_{l=1}^{\infty}\lambda^{l-1}\sum_{m=0}^{l-1}\gamma^m\text{TD}_{t+m}^V, \tag{40}$$

simply expanding this l then m shows that we get

$$\sum_{l=0}^{\infty}\lambda^l\text{TD}_{t+0}^V+\sum_{l=0}^{\infty}\lambda^{l+1}\gamma\text{TD}_{t+1}^V+\sum_{l=0}^{\infty}\lambda^{l+2}\gamma^2\text{TD}_{t+2}^V+\ldots, \tag{41}$$

then employing the use of $\frac{1}{1-\lambda}=\sum_{l=0}^{\infty}\lambda^l$ for $|\lambda|<1$, we get

$$\begin{aligned}&\frac{1}{1-\lambda}\text{TD}_{t+0}^V+\frac{\lambda}{1-\lambda}\gamma\text{TD}_{t+1}^V+\frac{\lambda^2}{1-\lambda}\gamma^2\text{TD}_{t+2}^V+\ldots\\ &=\sum_{l=1}^{\infty}(\gamma\lambda)^l\,\text{TD}_{t+l}^V.\end{aligned} \tag{42}$$

We thus get

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)}=\sum_{l=1}^{\infty}(\gamma\lambda)^l\,\text{TD}_{t+l}^V. \tag{43}$$

References

[1] John Schulman and Philipp Moritz and Sergey Levine and Michael Jordan and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. Proceedings of the International Conference on Learning Representations (ICLR) 2016.

[2] Sutton, R. S. and Mcallester, D. and Singh, S. and Mansour, Y.. Policy gradient methods for reinforcement learning with function approximation. Advances in Neural Information Processing Systems 12. 2380307.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms. abs/1707.06347, 2017.

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra. Continuous control with deep reinforcement learning. abs/1509.02971, 2019.

[5] Volodymyr Mnih, Adrià Puigdomènech Badia , Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu.Asynchronous Methods for Deep Reinforcement Learning. abs.1602.01783, 2018.