Learning to Walk: A Reinforcement Learning Approach

Johann Meyer

***Abstract*—This is paper aims to implement the twin delayed deep deterministic (TD3) policy gradients algorithm using only Numpy and a few standard Python libraries. The algorithm will be used to train a bipedal walker to move as far right as possible in a continuous state-space world. The bipedal walker takes as input 4 continuous-valued actions. The agent managed to solve the environment (as stipulated by the OpenAI gym) by achieving a 100-episode reward average of more than 300.**

***Index Terms*—Deep Reinforcement Learning, Deep Deterministic Policy Gradients, TD3, Bipedal Walker, Neural Network, Function Approximation, Actor-Critic**

I. Introduction

Reinforcement learning has in recent years gained interest as major hurdles that arise from the deadly triad[20] have been alleviated by new approaches in the field meaning that the technique can now be applied to a much larger variety of real-world problems. The benefit of reinforcement learning over traditional control system approaches is that the system has the ability to adapt to a changing environment and the emergence of interesting solutions to problems. In the first case, this is interesting for fault-tolerant flight control where the aircraft model may change during the flight and thereby require an alternative flight control strategy to handle the change in dynamics. In the second case, some problems may not present an obvious solution and the ability of reinforcement learning agents to exploit the reward process leads to interesting emergent behaviour[13]. Of course, exploiting the reward process is a double-edged sword as a poorly defined reward process can lead to issues such as 'reward hacking' and reward function 'gaming'.[1] Other issues such as safe exploration are still on-going problems in the field and the gap between simulation and hardware-in-the-loop is still problematic. Moreover, certification and safety of black box systems continues to be an issue.

The aim of this paper is to develop a reinforcement learning controller with continuous actions. The goal is that the controller can learn to walk in a continuous state-space environment with no model of the environment and purely through interaction with the environment.

II. Background

Reinforcement learning is an approach to teach an agent how to act optimally within its environment by interacting with its environment and learning from these interactions to maximise a reward signal that the environment gives it (see Fig. 1). It is distinct from supervised learning, where a correct action a in each state s would need to be known a priori and the agent would simply fit its models to those predetermined actions. However, since these actions are not known a priori, the credit assignment problem arises.

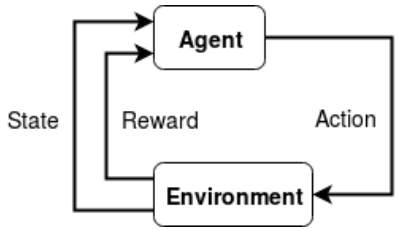


Fig. 1. An agent determines what action to take given its current state and determines which actions lead to better rewards by exploring its environment and learning which states and actions led to higher rewards.

The environment of the agent can in general be described as a Markov Decision Process (MDP). An MDP is a specification of a problem that describes the probabilities of receiving rewards and transitioning to certain states given a current state and an action in that state. In dynamic programming, the MDP specification is assumed to be known. In contrast, reinforcement learning assumes no knowledge and may either be model-free or it may learn a model from interaction with the environment.

The credit assignment problem arises due to sparse rewards and rewards that arise due to a series of coordinated actions. A simple example of this problem in practice is any score-based game, for example Pong or football. Players do not receive any feedback about how they are playing until a point is scored. The actions that led to scoring the point should be taken more often than those that did not; however, which actions directly led to scoring the goal may not be obvious. For example, actions taken 20 minutes before scoring the point is most likely irrelevant, but perhaps the action the player took 10 seconds before receiving the ball is more relevant. However, if the player was distracted and the point was more luck than skill then it will still be reinforced. This is why reinforcement learning often has the problem of high variance during training. Bias can be introduced by making assumptions, such as, only the most recent actions are important and that is where reward discounting comes into play. This will make sure the actions taken 20 minutes ago have little effect when learning what to do in the current state.

The correctness of the assumption depends on the problem being solved. Reward discounting is given by eq. 1[20], where γ is the discounting factor in the range 0 to 1 and R_{t+1} is the reward obtained at time t. The discounted reward G_t can be succinctly described as the discounted sum of future rewards.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

Another important consideration is the exploration vs exploitation trade-off. It is a very difficult trade-off, especially in actual systems where damage to the system may be expensive or dangerous. The learning algorithm used may also impact the approach taken to this trade-off. For example, SARSA adapts its policy to include the effects of exploration, which will be discussed in more detail later on. Alternatively, stochastic policy methods will implicitly explore the environment due to the stochastic nature of the policy. Too much exploration means the agent will not be able to use its acquired knowledge effectively and too little means the agent will not discover new and possibly better approaches to solving the problem leading to a sub-optimal agent.

There are two main approaches to determining the optimal policy: action-value methods and policy gradient methods. Action-value methods determine the value functions for each state after taking a specific action. By determining the optimal action-value function, the actions in each state can be determined by selecting the action that has the highest expected value. The policy is thus implicit in the action-value function. In contrast, policy gradient methods directly parameterise the action to be taken given a state. This has several advantages and disadvantages. Policy methods usually have higher variance leading to slower or less stable training; however, in some complex environments learning the action-value function may be more difficult[20]. Moreover, by directly parameterising the action to be taken, the algorithm is no longer bounded by low-dimensional action spaces[20], [9]. This problem is referred to as the curse of dimensionality and is also the reason that methods like tile-coding and tabular methods are avoided in very high-dimensional state-spaces and instead nonlinear function approximators are used. In addition, in contrast to value function methods, policy gradient approaches can learn stochastic policies, which may be necessary to solve some types of problem (e.g. poker or problems with aliased states) but be unfavourable in others (e.g. robotics) where deterministic actions makes it, for example, easier to certify.

Finally, there is the issue of the stability of reinforcement learning algorithms. Instabilities in the algorithms usually arise when the all three elements of the deadly triad are present. The deadly triad consists of: function approximation, bootstrapping, and off-policy training.

A. Action-value Methods

Value functions solve eq. 2, which is commonly referred to as the Bellman equation. V_π is the value function for the given policy π, R_{t+1} and s_{t+1} are the sampled reward and state transition from the environment, and γ is the aforementioned discount factor. Value functions can be sufficient to determine what actions need to be taken provided there is a model of the environment. However, often the model is not present or very accurate and repeated sampling from the model for an accurate estimate of the best action makes it computationally slower to an action-value method. Action-value methods require more memory than a value functions but do not require a model, which in turn may require significant additional memory if implemented by, for example, a table-lookup. Action-value functions are denoted by Q. Note that a_{t+1} is not sampled from the environment and thus a choice must be made about the next action.

$$V_\pi(s_t) = \mathbb{E}_\pi \left[R_{t+1} + \gamma V_\pi(s_{t+1}) \right] \tag{2}$$

$$Q_\pi(s_t) = \mathbb{E}_\pi \left[R_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \right] \tag{3}$$

The two primary forms of action-value learning are SARSA and Q-learning. SARSA is an on-policy method, meaning that it updates its action-value function to reflect its current policy (i.e. a_{t+1} is the next action taken). Q-learning, on the other hand, is an off-policy method and directly approximates the optimal action-value function (i.e. a_{t+1} is the action that leads to the maximum Q-value regardless of whether or not it is taken in the next time-step). Both methods have their own merits, for example, SARSA learns the optimal policy given the exploration policy (i.e. the action-value function is conservative), which may not be the optimal policy for the environment, if exploration is not reduced asymptotically to zero. Q-learning, on the other hand, does not factor the exploration policy into the value estimates, which means potentially hazardous moves may be taken. This is aptly described by the cliff-walking task in [20].

Off-policy methods have further disadvantages when combined with nonlinear function approximation, which leads to an unstable learning algorithm; however, recent research has enabled the two to be combined[11]. The main techniques that enabled DQN to work was decorrelating the updates of the value function by sampling random batches of data from a replay memory and utilising a target network to avoid the moving target problem. For action-value methods with nonlinear function approximation, a common optimisation is to make the function approximator accept the state as input and output the value for all actions in that state. The benefit of this approach is that the network can perform a single forward pass and compute all the values for all actions instead of a single pass per action, which would significantly slow down the algorithm for larger action spaces. One issue with optimising a Q network is that it suffers from maximisation

bias. Double DQN[22] solves this by using a different network to select the maximising action in the target for the action-value update step.

B. Policy Gradient Methods

Traditional policy gradient methods, like REINFORCE[23] are based on Monte-Carlo returns from episodes. They work on the principle that you should increase the probability of actions that led to good rewards and decrease those that led to worse rewards. This idea is captured by the policy gradient theorem given by eq. 4 in the discrete case or with the summations replaced with integrals in the continuous case. The policy gradient theorem is an analytical expression for the gradient of the performance $\nabla J(\theta)$ of the agent with respect to the parameters θ of the stochastic policy π_θ given the normalised on-policy state distribution $\mu(s)$. Note that $\mu(s)$ can be seen as a probability of being in state s given the current policy π_θ. It should then be clear that eq. 5 is actually the expectation of the policy, as in eq. 6. The term $\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}$ is called the eligibility vector in [20] and can be transformed using the likelihood ratio trick (eq. 8) to an often mathematically simpler form (eq. 7). The form in eq. 6 is useful from an intuition perspective as it describes a update that is proportional to the state-action value function and inversely proportional to the probability of selecting that action when following the policy. The reason the inverse of the probability is used is so that less probable actions can receive larger updates as they are less likely to be visited and updated in the future.[20] Most importantly, the expression is not dependent on the derivative of the state distribution meaning the theorem can be applied in a model-free approach.

$$\nabla_\theta J(\theta) \propto \sum_{s_i} \mu(s_i) \sum_a Q_\pi(s_i, a) \nabla_\theta \pi_\theta(a|s_i) \tag{4}$$

$$\propto \sum_{s_i} \mu(s_i) \sum_a \pi_\theta(a|s_i) Q_\pi(s_i, a) \frac{\nabla_\theta \pi_\theta(a|s_i)}{\pi_\theta(a|s_i)} \tag{5}$$

$$\propto \mathbb{E}_\pi \left[Q_\pi(s, a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \right] \tag{6}$$

$$\propto \mathbb{E}_\pi \left[Q_\pi(s, a) \nabla_\theta \ln \pi_\theta(a|s) \right] \tag{7}$$

$$\nabla_\theta \ln \pi_\theta(a|s) = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{8}$$

As a consequence of being a Monte-Carlo-based algorithm, the variance will be high. This is as a result of there being many different actions that may lead to the same rewards but the idea is that overtime the better actions will become more likely than the worse ones. To reduce the variance of REINFORCE, a baseline may be introduced. The baseline does not introduce bias but it does reduce variance. The baseline is often chosen to be the state-value function and can be interpreted as saying how much better was the selected action compared to the expectation of the agent. Bias can be introduced by converting REINFORCE with a baseline to an actor-critic algorithm. This algorithm can be implemented as an online algorithm and no longer requires the entire episode to be completed before updates can be made to the actor and critic. The difference between the critic and the baseline is that the critic attempts to directly estimate the improvement a selected action provides by utilising bootstrapping. Bootstrapping is where an estimate is used to improve another estimate and the estimate will be biased by its initial estimates. Generalised Advantage Estimation[17] (see eq. 10 and eq. 11) is a method to smoothly interpolate (λ) the estimated advantage function $\tilde{A}$ between a biased and an unbiased estimate. The value function is typically learned using the Monte-Carlo returns as it reduces the bias in the final policy. An alternative to this approach is using n-step truncated returns as in, for example [10]. It is also often beneficial for stability to utilise what is known as trust regions, which was introduced in [16], as it results in nearly monotonic improvement in the policy. Proximal Policy Optimsation (PPO)[18] is an efficient approximation of TRPO that tends to perform almost as well but with significantly less wall time. Finally, it should be noted that most discounted actor-critic algorithms introduce bias when the policy update is not discounted with a γ^t term, where t is the time-step number; however, this leads to worse data efficiency and should thus be avoid, if possible.[21]

$$A_\pi(s_t, a_t) := Q_\pi(s_t, a_t) - V_\pi(s_t) \tag{9}$$

$$\tilde{A}_t^{\mathrm{GAE}(\gamma,\lambda)} := \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{10}$$

$$\delta_t^V = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \tag{11}$$

A recent variant of policy gradients does not require a stochastic policy and is thus appropriately termed deterministic policy gradients (DPG)[19]. Learning in DPG is faster since integration only occurs over the state-space and not in addition to the action-space.[19] However, exploration is now handled separately using a stochastic behaviour policy to learn about a deterministic target policy. The resulting algorithm is therefore an off-policy algorithm. DPG works by updating the policy in the direction of increasing Q. By doing this, the maximisation step in Q-learning can be avoided. Equation (12) is called the DPG theorem and states that the gradient of the performance function is given by chaining the gradient of the action-value function (parameterised by θ') with respect to the input action, which is the output of the deterministic policy μ (parameterised by θ). Deep DPG is an extension of DPG that enables the use of nonlinear function approximators by utilising some of the techniques from [11].

$$\nabla_\theta J(\theta) = \mathbb{E}_\mu \left[\nabla_\theta \mu_\theta(s_t) \nabla_{a_t} Q_{\theta'}(s_t, a_t) \right] \tag{12}$$

III. Problem

The goal is to get a 2D bipedal walker to navigate through a terrain with an uneven surface, see Fig. 2. The environment[1] is provided by the OpenAI Gym[2][3]. It is important to note that this is not the same as the Bipedal Walker in the Mujoco gym environment that is often used for benchmarking. The agent can operate the bipedal walker by applying torques to its 4 joints: 1 at each knee and 1 at each hip. Each action is defined in the range $[-1, 1]$. The agent's observation of its environment is composed of 24 states. The reward function is -100 if the hull touches the ground, a small negative reward $(-0.00035 \cdot |\text{torque}|)$ for applying torques, and a positive reward $(4.33 \cdot \Delta x)$ for moving the agent towards the right of the environment. A penalty is also applied based on the orientation of the hull $(-5 \cdot |\text{hull angle}|)$. This is to ensure that the hull remains approximately horizontal. The environment is episodic and has a limit of 1600 steps and terminal states at $x = 88$ and $x < 0$ and when the hull touches the ground. The environment is considered solved when the agent achieves a 100-episode average reward of at least 300.

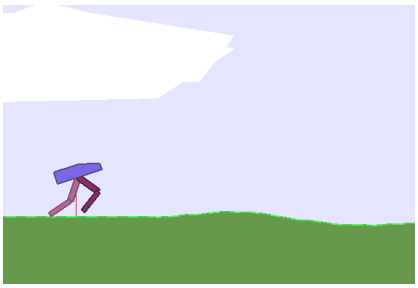


Fig. 2. A screenshot of the simulation of the OpenAI Gym bipedal walker environment.

IV. Algorithm

This section will explain the Twin Delayed Deep Deterministic (TD3) policy gradients algorithm[4]. However, first, the Deep Deterministic Policy Gradient (DDPG) algorithm[9] will be explained as it forms a basis for the TD3 algorithm.

A. Deep Deterministic Policy Gradients (DDPG)

Deep Deterministic Policy Gradients is a variation of Deterministic Policy Gradients which has been improved with the modifications found in DQN[11].

Key modifications:

1) Experience Replay
2) Target Networks

[1]https://github.com/openai/gym/wiki/BipedalWalker-v2
[2]https://gym.openai.com/

Experience Replay is a buffer of (s, a, r, s', D), where D is whether or not the state being transitioned to is a terminal state and is necessary for determining whether or not the target Q value should include the bootstrapped estimate.

In contrast to DQN, DDPG updates the target network with soft updates opposed to hard updates in the DQN algorithm. Soft updates can be seen as a low-pass filter that blends the weights of the target network with the weights in the main network. Hard updates are where the main network's weights are copied into the target network at regular intervals which are slower than the main network's update rate. The idea behind the target network is to avoid the moving target problem and that the main network should be able to converge to the estimates provided by the target network before it changes. This alleviates the problem described by Baird[2] which is, in summary, due to the fact that the update to one state leads to updates to all the similar states as the weights are shared. The alternative solution described by Baird is to use residual algorithms which account for this effect; however, in practice, the estimates do not necessarily converge to the true values of the states and learning can be significantly slower.[20]

Algorithm 1 DDPG Algorithm

Initialise critic network Q_θ and actor μ_ϕ with random parameters θ and ϕ
Initialise target networks $\theta' \leftarrow \theta$ and $\phi' \leftarrow \phi$
Initialise replay buffer $\mathcal{B}$
for $t = 1 \rightarrow T$ **do**
  Select action with exploration noise $a \sim \mu_\phi(s) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s.
  Store transition tuple (s,a,r,s',D) in $\mathcal{B}$
  Sample mini-batch of N transitions from $\mathcal{B}$
  $a'_i \leftarrow \mu_{\phi'}(s'_i)$
  $y_i \leftarrow r_i + \gamma \cdot (D_i - 1) \cdot Q_{\theta'}(s'_i, a'_i)$
  Update Critic θ by minimising $N^{-1} \sum_i (y_i - Q_\theta(s_i, a_i))^2$
  Update Actor ϕ by maximising $N^{-1} \sum_i \nabla_a Q_\theta(s_i, a_i) \cdot \nabla_\phi \mu_\phi(s_i)$
  Update target networks
  $\theta' \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta'$
  $\phi' \leftarrow \tau \cdot \phi + (1 - \tau) \cdot \phi'$
end for

1) Action-value Function Approximation: DQN cannot be applied to continuous action spaces since taking the action that maximises the Q function requires the Q function to be computed for every possible action combination, which quickly becomes intractable even if discretised (curse of dimensionality). The solution in DDPG is that the Q function is not used to select the action but rather informs the actor of the local direction that would gain more discounted future reward. This means that only one forward pass of the Q function is required. In order to be able to determine the local gradient, the action must now be an input to the network and no longer, as in DQN, one output per action.

2) Policy Network: The policy network takes as input the state and outputs the deterministic action (the mean action) for each

actuator. The output activation is a tanh layer to ensure the outputs remain within the range of operation of the actuator. The tanh activation function outputs a value in the range [-1, 1]; however, some actuators may have a range that differs from this. The solution is to scale the outputs to the valid action range. This scaling should be applied to the gradients as well when applying backpropagation.

B. Twin Delayed Deep Deterministic Policy Gradients (TD3)

Twin Delayed Deep Deterministic policy gradients[4] provides several improvements to the DDPG algorithm.

Key modifications:

1) Two critics
2) Policy updated less frequently
3) Action noise when training critics to smooth the action-value estimates.

The role of the additional critic is analogous to what Double DQN aims to improve over DQN: the issue of maximisation bias. The issue arises because the agent's action selection and action evaluation are coupled and leads to the agent improving actions that are already preferred. By decoupling them, this issue is avoided and the expected action-values are then unbiased[22]. This leads to ultimately better and more stable policies[22] and faster training[4], [20]. The update rule used in the final TD3 algorithm may result in underestimation bias and does not result in an unbiased estimate but is preferable as underestimation does not impact the learning as negatively.[4]

The second modification is the addition of the hyperparameter d which determines the number of updates to the critic before the actor is updated. The hyperparameter d is set to 2 in the original paper. The reasoning behind this modification is that if the actor is being updated on high-error states the critic will not provide the correct gradient. Moreover, it limits the number of updates the policy takes with respect to an effectively unchanged critic.[4]

The third modification is the addition of clipped Gaussian noise with a standard deviation $\sigma_{\text{TPS-reg}}$ to the action estimate of the next state. In the original paper, this is termed target policy smoothing regularisation. The consequence of this is a smoother Q function and avoids the potential for spikes in the Q function causing overly optimistic Q estimates. This makes the following the gradient of the critic much better behaved. This does make the assumption that similar actions should lead to similar discounted rewards. If small deviations from the optimal action can be catastrophic or lead to very different rewards as in precision or safety-critical environments then this modification should not be used. Moreover, care should be taken that the clipping of the Gaussian noise that is added to the action is not clipped such that the boundaries become more probable and ultimately limit the ability of modification to smooth the Q function.

Algorithm 2 TD3 Algorithm

Initialise critic networks Q_{θ_1}, Q_{θ_2} and actor μ_ϕ with random parameters θ_1, θ_2 and ϕ
Initialise target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$ and $\phi' \leftarrow \phi$
Initialise replay buffer $\mathcal{B}$
for $t = 1 \rightarrow T$ **do**
  Select action with exploration noise $a \sim \mu_\phi(s) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s.
  Store transition tuple (s,a,r,s',D) in $\mathcal{B}$
  Sample mini-batch of N transitions from $\mathcal{B}$
  $a'_i \leftarrow \mu_{\phi'}(s'_i) + \epsilon$, where $\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma_{\text{TPS-reg}}), -c, c)$
  $y_i \leftarrow r_i + \gamma \cdot (D_i - 1) \cdot \min_{j=1,2} Q_{\theta'_j}(s'_i, a'_i)$
  Update Critic θ_j by minimising $N^{-1} \sum_i (y_i - Q_{\theta_j}(s_i, a_i))^2$
  if (t mod d) == 0 **then**
    Update Actor ϕ by maximising $N^{-1} \sum_i \nabla_a Q_{\theta_1}(s_i, a_i) \cdot \nabla_\phi \mu_\phi(s_i)$
    Update target networks
    $\theta'_i \leftarrow \tau \cdot \theta_i + (1 - \tau) \cdot \theta'_i$
    $\phi' \leftarrow \tau \cdot \phi + (1 - \tau) \cdot \phi'$
  end if
end for

C. Reward Clipping

Reward clipping is the process by which the range of the rewards that can be received is reduced to a smaller range. This may change the optimal policy if not performed carefully but it can significantly improve training stability and the rate of convergence. This improved rate of convergence arises by not marking potentially good states as very bad during exploration leading to the state not being visited before all other states have been deemed worse. It is also not necessary for the state to be much worse than another action for it not to be selected as long as it is still worse than the optimal action. In the specific case of the bipedal walker, there are small negative rewards for applying torques on the joints and a large negative reward of -100 for failure. A very large negative reward can have impacts on states much prior to the failure state depending on the discount factor. This has the implication of being able to negatively impact good states on long trajectories. In other words, the discount factor determines the temporal importance of prior states and by having rewards of varying scales this temporal importance is much harder to select to ensure lower variance without losing important reward information for the optimal policy. Furthermore, neural networks are not typically well-suited for large variances in their outputs. By clipping the reward to -1, for example, the discount factor can no longer negatively impact these prior states too severely but still enough to ensure that the trajectory is ultimately discouraged. In summary, having rewards on very different scales can lead to unstable and slow training and has empirically very little added benefit to solving the problem optimally, for example [11] achieved human-level control using clipped rewards.

V. Preliminary Studies

Since the problem involves a large continuous state- and action-space, it is useful to perform some preliminary studies on a simpler task to assess the impact of the various parameters on the reinforcement learning problem in general. Although the parameters from the simpler tasks cannot be transferred to the more complex problem it will be beneficial to understand how the parameters impact learning so that an educated process may be followed when applying it the considerably more time-consuming larger problem where a thorough hyper-parameter exploration is less feasible. For this, the classical cart pole problem will be used. The cart pole problem involves a cart that can be manoeuvred discretely, either left or right, and the goal is to ensure the inverted pendulum does not fall over or exceed a certain angle while remaining within the bounds of the arena. The reward is +1 for each time step for a total reward of 500.

For the cart pole problem, REINFORCE, and actor-critic algorithms will be studied. A comparison of the actor-critic and REINFORCE algorithms will illustrate the benefit of the bootstrapping process over a Monte-Carlo process and the ability of a baseline/critic to reduce the variance during training. Generalised Advantage Estimation will be used to control the amount of variance and to correctly assess the impact of bootstrapping. Only for the cart pole problem is an adaptive learning rate utilised. This is not referring to the learning rate used internally in the optimisers but the baseline learning rate that is used. It is adaptive as the learning rate is divided by the size of the batch as longer episodes will lead to more updates and as a result instability. This adaptive learning rate works for cart pole problem since the episode length determines success and, usually, when the episodes are longer, smaller updates should be used. It also allows fast learning in the initial phase. This modification does not lead to better training on environments where this assumption no longer holds, for example the lunar lander environment.

VI. Results & Discussion

In this section, the results from the preliminary study and the main problem will be presented and analysed.

A. Preliminary Results

In this subsection, a few tests are performed using the cart pole problem. First, the impact of discounting will be looked. Thereafter, the impact of using a baseline to reduce the variance will be investigated. Third, the impact of bootstrapping on learning performance will be assessed using generalised advantage estimation as a method to control the amount of bootstrapping. Finally, the impact of batch size on learning performance will be assessed. All figures in this section are produced with three runs and the shaded region is the max-min bounds over the three runs.

Discount factors are used to trade-off between bias and variance, as mentioned in section §II. Here, an empirical analysis of this trade-off will be performed. In Fig. 3, it is clear that using a smaller discount factor can reduce the variance in the learning stability; however, the bias introduced by the smaller discount factor can sometimes lead to worse asymptotic performance. In this case, a discount factor of 0.95 is a good balance between bias and variance and leads to a steady growth in performance. In Fig. 4, the reduction of variance is still visible; however, the impact of the bias seems to drastically reduce the final performance of the algorithm.

Another method to reduce variance in policy gradient methods is to introduce a baseline. In Fig. 3, the higher variance results in oscillatory learning and large run-to-run variation in learning. In contrast, Fig. 4 illustrates very little run-to-run variation even for differing discount factors and the performance improvement is practically ideal with an almost monotonic increase in performance. Moreover, in all cases, REINFORCE with a baseline outperforms vanilla REINFORCE in final achieved score.

The last method to reduce variance is by utilising bootstrapping. In Fig. 5a, increasing the level of bootstrapping can increase the final performance of the algorithm but also increases the run-to-run variance. In contrast in Fig. 5b, bootstrapping, if not too large, significantly stabilises the learning and does not reduce the final performance of the agent.

Stochastic gradient descent performs one update per sample; however, the results in Fig. 6, show that small batch sizes can lead to unstable training. It is also apparent from the figure that larger batch sizes do result in more stable learning; however, also significantly slower learning. The reason is that larger batch sizes means fewer updates per batch of rollouts.

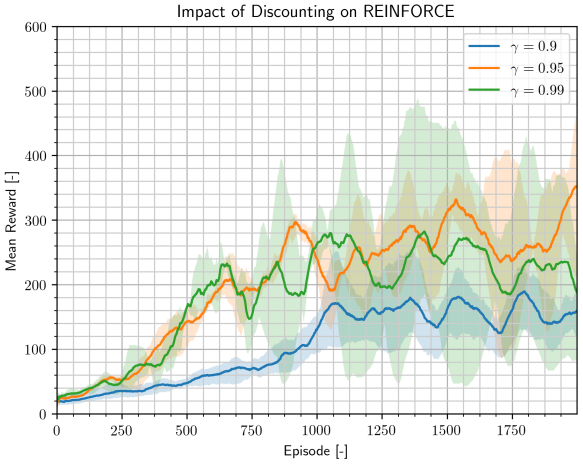


Fig. 3. REINFORCE on the cart pole problem.

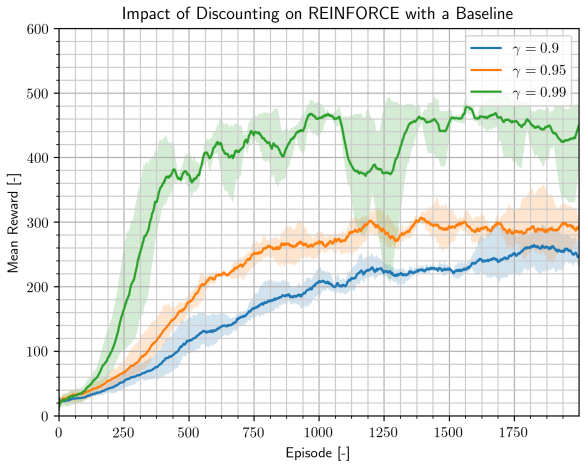
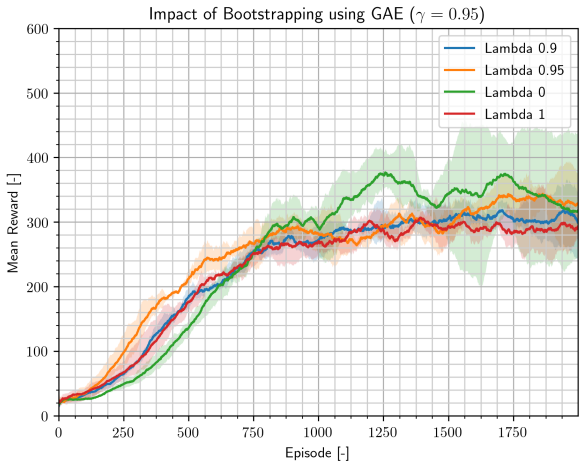
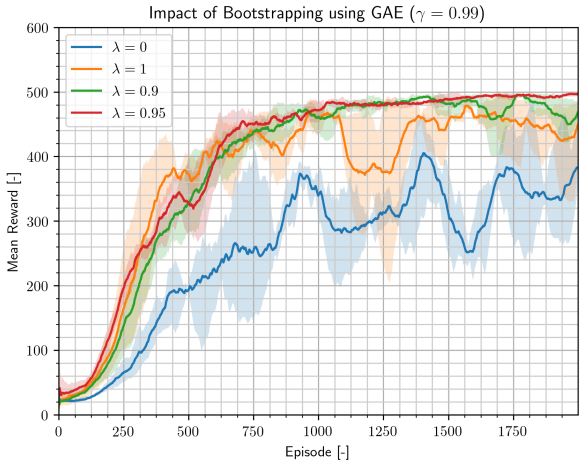


Fig. 4. REINFORCE with baseline on the cart pole problem.



(a)



(b)

Fig. 5. The impact of varying the level of bootstrapping on the learning performance in the case of the cart pole problem. (Algorithm: Generalised Advantage Estimation)

B. Bipedal Walker Results

In this section, the results for DDPG and TD3 will be discussed. In contrast to the previous section, only one run will be performed per configuration as the training is much slower than for the cart pole problem as the networks need to be significantly larger. This of course does limit the ability of statistically significant conclusions to be drawn as even averaging over batches of 5 runs per configuration can lead to significantly varying conclusions[5]. First, the impact of the batch size on training stability will be checked then the impact of the size of the buffer used for the replay memory will be evaluated.

Modifying the batch size, as in Fig. 7, does not have the same stabilising effect as it did for the cart pole problem and this can mean either one of two things, even larger batch sizes are needed or that the batch size is already large enough and that the quality of the gradient direction is not the cause for instability. The maximum 1-episode reward achieved for batch sizes of 32, 64, and 128 are 305, 309, and 303, respectively. This is not significantly different for a single run of each configuration. In Fig. 8, it is clear that this instability is not as problematic for the same parameter values as DDPG. This leads to the hypothesis that the instability is being caused by maximisation bias. Furthermore, different batch sizes do not lead to different maximum 100-episode averages leading to the conclusion that possible smaller batch sizes may be plausible; however, the maximum 1-epsiode reward achieved may be impacted for much smaller batch sizes. The 1-episode reward maxes are now 318, 316, and 322 which is larger than the what was empirically achieved using DDPG. It was also found that the 1-episode rewards can be increased by switching to a deterministic behaviour policy where the rewards are then around 334. The reward variance in the bipedal environment is also partly responsible for the large sudden losses in performance as a failure is a reward of -100 and if the algorithm temporarily overestimates the value of bad state, the 100-episode average will quickly suffer.

The size of the replay buffer does have a significant impact on the learning process but not on the maximum 1-epsiode reward. For the 3 memory replay sizes from smallest to largest, the 1-epsiode maximum reward achieved was 318, 314, and 316. The replay buffer size did not significantly impact this; however, it smaller buffers do lead to loss of performance when the old experience, particularly the initial exploratory phase, is replaced with newer experience. This is illustrated in Fig. 9 where it can be seen that the duration of stable performance is related to the amount of experience that can be stored. The amounts are approximately 3125-5000 episodes for the largest buffer size and 183-315 episodes for the smallest. It is thus recommended to use a buffer size that is as large as possible. The initial learning phase does appear to improve due to the larger and more diverse buffer; however, more tests would need to confirm that this is not an artefact of the network initialisation.

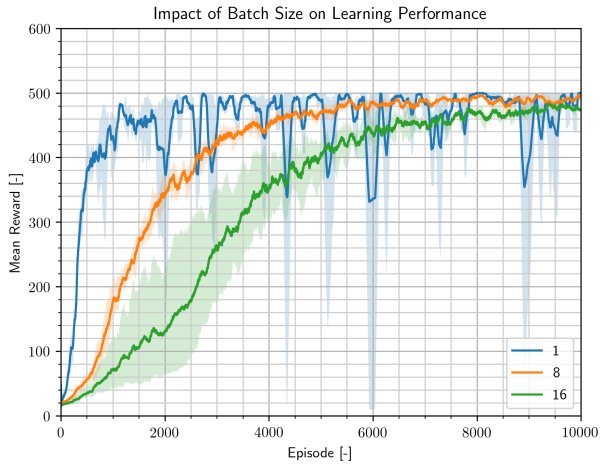


Fig. 6. REINFORCE with baseline on the cart pole problem with varying batch sizes.

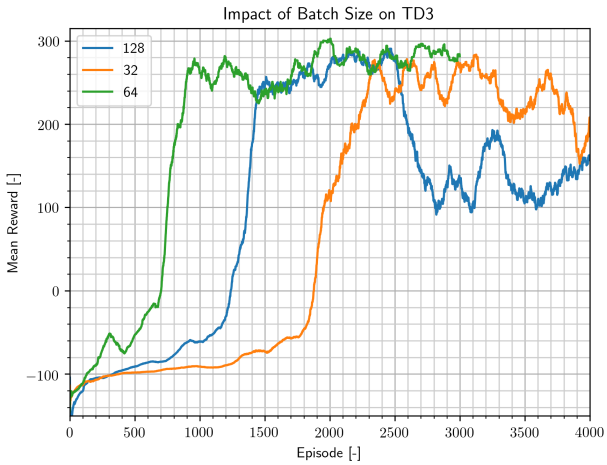


Fig. 8. TD3 with varying mini-batch sizes.

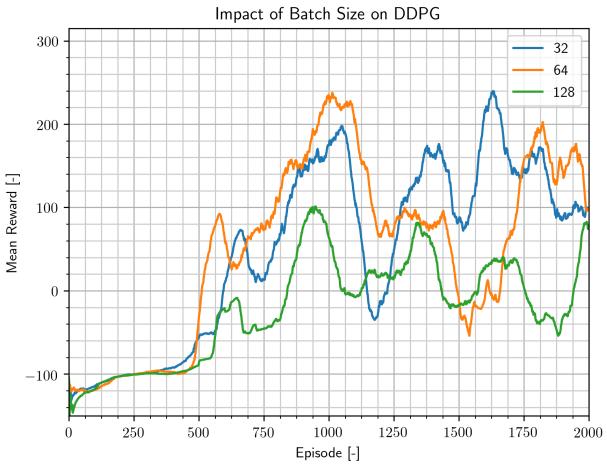


Fig. 7. DDPG with varying mini-batch sizes.

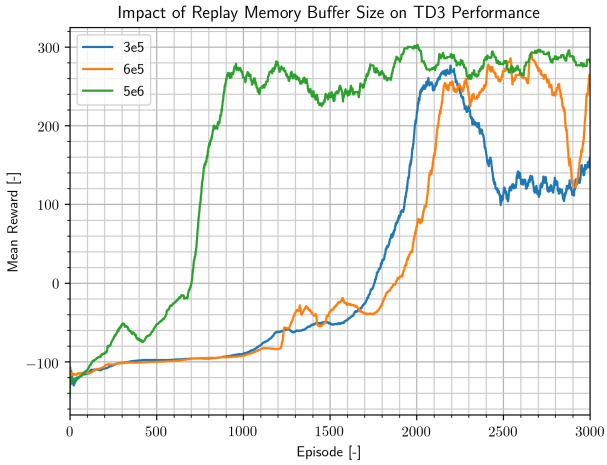


Fig. 9. TD3 with varying sizes for the replay memory buffer.

VII. Conclusions & Recommendations

A complete neural network library was developed using only numpy and the standard python library. Several reinforcement learning algorithms were developed and tested using the neural network library to gain an understanding of the impact of the many parameters involved in the algorithms. It was learned that deep reinforcement learning is a much harder problem than typical deep supervised learning due to the non-stationarity of the learning objective and that bootstrapping exacerbates this problem. Finally, the improvement in stability and performance of Twin Delayed Deep Deterministic (TD3) Policy Gradients was shown over the use of vanilla Deep Deterministic Policy Gradients (DDPG). DDPG never managed to be stable enough to achieve a 100-episode average reward of 300 whereas this was more consistently achieved by TD3. In general, value based methods can be very unstable as small changes in the value function can cause large changes in the policy. In practice, it was found that using a learning rate that is an order of magnitude lower for the actor than the critic leads to much more stable learning. This allows the critic to more accurately approximate the gradient for the policy.

Problems that still need to be addressed is that the policies do degrade after a while of very good performance. Thus it would be interesting to explore concepts similar to trust regions for on-policy algorithms in the off-policy situation to mitigate these issues. Moreover, difficulties with stochasticity in the environment and initialisation can lead to completely different results and conclusions[5], which means a more extensive Monte-Carlo study should be performed than was possible with the utilised hardware. In addition, the current formulation of the environment results in a terminal state when the time runs out; however, it was shown in [14] that handling time limits in this manner leads to worse performance. It is thus recommended for this specific problem that if the time limit is reached that the target should use the bootstrapped value from that state and not 0. Finally, it would be interesting to compare the results achieved with a model-based reinforcement learning

approach.

REFERENCES

[1] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mane, "Concrete problems in ai safety," 2016.

[2] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," 1995.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[4] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[5] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," *Association for the Advancement of Artificial Intelligence*, 2017.

[6] A. Karpathy, "Cs231n convolutional neural networks for visual recognition." [Online]. Available: http://cs231n.github.io/

[7] ——, "Deep reinforcement learning: Pong from pixels," May 2016. [Online]. Available: http://karpathy.github.io/2016/05/31/rl/

[8] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations*, 2016.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[11] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, February 2015.

[12] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[13] S. Nolfi, *Neurocomputing*. Elsevier, 2002, vol. 42, ch. Power and Limits of Reactive Agents, pp. 119–145.

[14] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[15] S. Ruder, "An overview of gradient descent optimization algorithms," 2016.

[16] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 37, 2015.

[17] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *International Conference on Learning Representations*, 2016.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[19] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, no. 31st. Journal of Machine Learning Research, 2014.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[21] P. S. Thomas, "Bias in natural actor-critic algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, 2014.

[22] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Machine Learning*, 1992, pp. 229–256.

APPENDIX A
NEURAL NETWORK IMPLEMENTATION

This section will explain some of the details of the implementation of the neural network. First, it will be explained how predictions are made with the network and then an explanation of how to update the weights in the network to better predict some target value will be given.

A. Inference

Inference or also known as the forward pass of the neural network is the process whereby the inputs to the neural network are transformed into an output. The output will vary depending on the purpose of the network, for example, regression or classification; however, the method for prediction is mostly similar, except for the activation functions that are often used for the output of the network.

A neural network consists of a set of layers, which are comprised of neurons or computation units and an activation function. The computations are often of the from $\vec{z} = W\vec{x} + \vec{b}$, where W is a weighting matrix that weights each input $\vec{x}$ of the layer and $\vec{b}$ is the bias or offset of the inputs. The output $\vec{z}$ is then passed through a nonlinear activation function, which performs an element-wise operation to each value in $\vec{z}$. If the activation function is not nonlinear, then the output will be a linear combination of the inputs, therefore, for the universal approximation theorem to hold, nonlinearities (such as Rectified Linear Units (ReLU), Sigmoids, Softmaxes or Radial Basis Functions (RBF)) are required.

The choice of activation function may depend on the application, or it may depend on the size of the network. For example, deep neural networks suffer from the vanishing gradients, which arises when the gradient of the activation function becomes zero. This occurs in Sigmoids, Softmaxes, RBFs, and tanh nonlinearities. ReLUs cannot output negative values or bounded values, whereas the other nonlinearities can. The key benefits of ReLUs are the sparse activation and the unidirectional saturation of the gradients, which leads to faster training.[6] Furthermore, ReLUs suffer from the dying ReLU problem, which occurs when certain neurons will never activate regardless of the input, which is a variant of the vanishing gradient problem.[6] Linear activation functions and Softmaxes are often only used as output layers. The softmax outputs a probability distribution much like the Sigmoid in the binary classification task. Tanh is in the range -1 and 1 and is effectively a scaled and shifted variant of the Sigmoid activation function. It is zero-centered and has a larger gradient than the Sigmoid. Tanh is often of primary interest in control as it is easily extended to represent a magnitude of the input and is bounded. RBFs are often convenient as their activation is limited to a local region. This is useful in tile-coding[20] and avoiding the learning process from changing the entire shape of the output of the layer after each example instead of locally refining it. The disadvantage of local activation functions is the inability to generalise and may even lead to a greater chance of overfitting.

Implemented Activation Functions:

- ReLU
- Radial Basis Function
- Sigmoid
- Softmax
- Tanh
- Linear

B. Learning

This section is divided into two sections: backpropagation and optimisers. Backpropagation is the technique that enables optimisers to improve the weights of the network to achieve the desired output.

1) Backpropagation: Backpropagation is a technique that analyses the gradient of the output of the network with respect to each of the weights. By utilising an analytical gradient, the gradients can be computed much faster than when using techniques like finite differences. Moreover, utilising analytical gradients is more accurate and stable as the network does not need to determine the step size for the finite difference. The backpropagation code in [12] was used as a foundation for the current code but has been adapted and extended for use with reinforcement learning.

In order to make backpropagation tractable, it is common to think of a neural network as individual operations that are applied sequentially to an input. Taking the derivative of the entire network is much harder than taking the derivative of each operation. For example, the derivative of multiplication is simply the coefficient of the multiplication, see eq. 13, eq. 14, and eq. 15. Of course, when there an input is applied to multiple neurons in a layer the gradient from each neuron should be summed to determine the total gradient of the input to that layer. It is also necessary to determine the derivative of the activation function. Note, however, that it is not necessary for the function to be differentiable but only that a gradient exists for each input. A ReLU, is not differentiable at zero; however, it can be arbitrarily chosen to be either +1 or 0.

Often the gradient will be modulated by a loss function or a scaling function. A scaling term will determine whether the output should be made more or less likely or be changed in a direction of another function. A loss function is a variant of this scaling and determines the direction of the change. In order to apply a loss function, the derivative of the loss function should be known and the output of the network will be fed into the loss function along with a set of target values. The optimiser will then decrease the loss using these gradients.

$$f = xy \tag{13}$$

$$\frac{df}{dx} = y \tag{14}$$

$$\frac{df}{dy} = x \tag{15}$$

2) Optimisers: Optimisers utilise the gradients from the backpropagation algorithm and apply them to the weights of the network. The heuristics behind each optimisation algorithm differs and a brief survey will now be given. The implementations are based primarily, but not exclusively, on the overview provided in [15].

Gradient descent is not one single algorithm but rather a collection of algorithms. Gradient descent works by changing the weights in the negative direction of the gradient, which is given by eq. 16. $\nabla_\theta W$ is the gradient of the output with respect to the weights and α is some positive step size parameter that determines the magnitude of the step in the direction of minimising the loss. All the algorithms have been defined as minimisers; however, to change the problem to a maximisation problem (gradient ascent) the only change necessary would be to negate the sign of the learning rate. Next, the algorithms can operate on randomly sampled or shuffled mini-batches of data. This leads to algorithms termed stochastic gradient descent (SGD). These stochastic algorithms have better guarantees for optimality as the optimisers are less likely to get stuck in local optima.

$$W_{t+1} = W_t - \alpha \nabla_\theta W \tag{16}$$

Building on SGD, there are algorithms that utilise the concept of momentum. The idea behind momentum is to increase the rate of convergence to a local optimum by increasing the magnitudes of updates in the same direction and damping those that are oscillatory. A popular variant of a momentum-based optimiser is Nesterov Acclerated Gradient (NAG). The key difference between it and traditional momentum-based optimisation is that it employs lookahead to the update. In other words, it determines the gradient at the estimate for the next parameters.

The final variant that will be touched upon is Adaptive Momentum Estimation (Adam)[8] and as the name suggests is a variant of momentum-based optimisation; however, the algorithm utilises also the second moment and it uses a correction for the updates such that they are not biased by the initialisation of the momentum terms.

Implemented Optimisers:

- Gradient Descent
- Stochastic Gradient Descent
- Momentum
- Nesterov Accelerated Gradient
- Adam

C. Numerical Stability

While implementing a number of reinforcement learning algorithms using the implemented neural network library, several numerical stability issues arose. The Sigmoid function and the Softmax function both utilise the exponentiation function meaning that the outputs of the functions can quickly approach the limits of the computer. To combat these problems, some simple techniques are applied. First, the Sigmoid will be analysed and then, subsequently, the Softmax.

The Sigmoid's numerical instability comes from the way it is formulated and the aforementioned limits of the computer to represent large numbers. The solution is an equivalent reformulation of the same equation whereby large positive

numbers in the exponentiation are avoided. The formulation in eq. 17 causes overflow at $-\infty$ and eq. 18 has an overflow at $+\infty$. eq. 17 is able to handle the overflow when the math library is written according to IEEE specifications which makes overflow lead to ∞ and $\frac{1}{\infty} = 0$, as is the case with Numpy; however, it is still preferable to avoid the problem altogether for portability and future proofing due to specification changes.

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{17}$$

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)} \tag{18}$$

Next, the Softmax (given by eq. 19) is unstable again for the same reasons; however, numerical stability is introduced by subtracting the maximum x value from all the x values, making the largest exponentiation input 0 and thus the largest output 1. This is tersely described by eq. 20.

$$\text{softmax}(x) = \frac{\exp(x)}{\sum_i \exp(x_i)} \tag{19}$$

$$\text{softmax}(x) = \frac{\exp(x - \max(x))}{\sum_i \exp(x_i - \max(x))} \tag{20}$$

APPENDIX B
EXECUTING THE CODE

All the code pertaining to this assignment was coded in Python 3.5 and tested on Ubuntu 16.04 and utilising the following dependencies. Note imageio library was only used to generate gif of the agent and can be commented out with no impact on the algorithm.

- numpy >= 1.14.5
- box2d-py >= 2.3.5
- gym >= 0.10.5
- imageio