Ready, Set, Go Humanoid!

Jasleen Kaur Dhanoa, Swati Gupta

***Abstract*—This paper discusses various Reinforcement Learning algorithms applied to the problem of making a Humanoid walk without falling over, in a continuous state and action space environment. In this report, we discuss numerous RL algorithms like Tabular Q-Learning, Q-Learning with basic function approximation, NFQ, Deep Q Learning, Vanilla Policy Gradient, DDPG and PPO for this problem and report the performance as well as the observations made in each of the approaches. We show that after training, the Humanoid is able to successfully stand and walk in the simulated environment.**

***Index Terms*—Reinforcement Learning, Deep Q-Networks (DQN), Deep Deterministic Policy Gradient (DDPG), Vanilla Policy Gradient (VPG), Proximal Policy Optimisation (PPO), Multi-Layer Perceptron (MLP), Humanoid-v2**

I. Motivation

The problem statement we tackled for this project is to learn how to make a 3-dimensional humanoid (bipedal robot) walk without falling over. Since this is a continuous control task, it has been best solved by approaches using the Reinforcement Learning paradigm. [3]

Our motivation to work on this project stemmed from the NeurIPS 2018: AI for Prosthetics Challenge (shorthand: Prosthetics Challenge) that was conducted at Stanford. We were inspired to build something that would lead to a positive contribution to the society through advancement in the domain of Rehabilitation Research and have useful applications in Robotics.

II. Related Work

Sutton and Barto must be included in all Reinforcement Learning works because it is a timeless classic of reinforcement learning theory, and contains references to the earlier work which led to Q learning based methods as well as modern policy gradients. [1] The Riedmiller M. (2005) paper [9] discusses NFQ (Neural Fitted Q-Learning) algorithm which produces a decent performance improvement over tabular Q Learning and basic function approximators using MLP. A breakthrough work that really made a huge impact in RL for Q Learning based algorithms came with Mnih, Volodymyr, et al. (2013) [8]. This paper introduces optimisations that stabilise the neural network training process and give great performance in discrete action tasks like Atari games. Finally, Deep Deterministic Policy Gradient Paper by Lillicrap et al. 2016 [10] builds on top of DQN concepts and adapts them for continuous action spaces using policy network approach, so is an important work that must be stated here.

Schulman 2016 [14] provides an introduction to Policy Gradient Algorithms and Schulman 2016 [2] provides an introduction to Generalized Advantage Estimation Function which are used in our implementation of PPO and VPG to get a stable estimate of returns. We also referred to Duan 2016 [3] as it provides a benchmark of RL algorithms for high dimensional continuous control tasks which is exactly what we are tackling in this project.

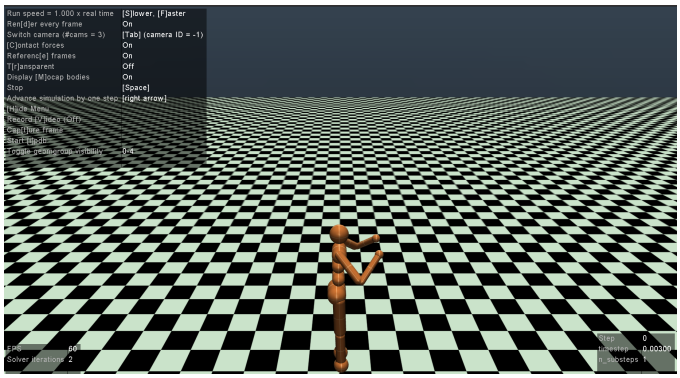


Fig. 1: Humanoid-v2 GYM environment

III. Dataset

Since this is a reinforcement learning problem, there is no ”dataset” in the traditional sense. We used the openAI GYM environment - Humanoid-v2 for this task.(Fig. 1)

After considering our available options, we found that the openSym simulator used in the original AI for Prosthetics challenge (i.e. the Stanford Physics simulator) is much slower than the GYM environment. Also, the complexity of the environment is significantly higher due to obstacles on the route etc. Hence, for this project, we have decided to use the Humanoid-v2 GYM environment.

The Humanoid-v2 GYM environment has 376 Observation dimensions and 17 action dimensions (i.e. it is a high dimensional observation and action spaces). Each dimension value in the observation space can vary from $-\infty$ to ∞ and in the action space can vary from -0.4 to 0.4. A scalar valued reward is emitted when the robot takes an action(step) in the environment. Our goal is to maximise the total expected reward and find the optimal policy that emits this reward.

IV. Problem Formulation

The problem falls under the reinforcement learning paradigm and is a continuous control task (as. opposed to tasks in discrete spaces which are comparatively easier to tackle). This formulation can be written as follows:

How can the robot learn to walk for itself to achieve best performance from a limited reward signal it gets from the simulated environment?

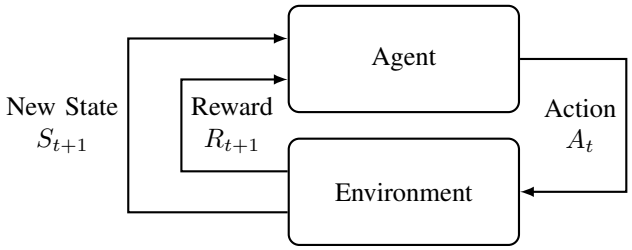


Fig. 2: Reinforcement Learning Feedback Loop

In short, we need to learn an optimal policy in this continuous state and action space.
Mathematically, we consider a Markov decision process (MDP), defined by the tuple (S, A, p, R, γ, H), where the state space S and the action space A are continuous, and the unknown state transition probability p : S × S × A $\rightarrow$ [0, ∞) represents the probability density of the next state $s_{t+1} \in S$ given the current state $s_t \in S$ and action $a_t \in A$. The environment (simulated GYM environment) emits a bounded reward R : S × A $\rightarrow$ [rmin, rmax] on each state transition. γ is the discount factor, and H is the horizon. Our goal is to then learn the optimal policy $\pi^*(a_t|s_t)$ that maximises the overall expected reward (discounted). Ref Fig-2 for pictorial representation of RL feedback loop.

We have made extensive use of Bellman Equation as the basis of our loss functions with additional bells and whistles that are specific to each algorithm implementation. For Q Learning based methods/q-networks, Bellman updates are used directly while in Policy Gradient and Actor-Critic algorithms, the policy network weights are updated using Policy Gradients. In mathematical terms, The Bellman equations can be stated as follows:
Optimal Value function :

$$V^*(s). = \max_{\pi} E[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1}|\pi, s_0 = s)]$$

Here, $V^*(s)$ = sum of discounted rewards when starting from state s and acting optimally (using policy π) Optimal Q function

$$Q^*(s, a) = \max_{\pi} E[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1}|\pi, s_0 = s, a_0 = a)]$$

Here $Q^*(s, a)$ = expected utility starting in s, taking action a, and (thereafter) acting optimally (using policy π).
Both these equations are basically equivalent. They can be re-written as an update method as follows and will converge (in an EM like fashion) to determine the optimal value/Q function as well as the optimal policy as $H \rightarrow \infty$.
These function values can be approximated using neural networks as well (i.e. non-linear function approximators), which is especially useful in our case is we are operating in high dimensional, continuos state and action spaces (where maintaining these values in a table is not very feasible unless we use discretisation or dimensionality reduction techniques). The Q function based update equation (used as loss function for Neural Network based methods) is as follows:

$$Q(s_t, a_t) =$$

$$Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a^*}(Q(s_{t+1}, a^*) - Q(s_t, a_t)))$$

where α is the learning rate. The first component: $r_t + \gamma \arg\max_{a^*} Q(s_{t+1}, a^*)$ is often referred to as target value (expected reward from the next state onwards under the policy), whereas the latter component $Q(s_t, a_t)$ is the current estimate of how good the state-action pair is.

For Policy Gradient methods, the objective is to maximise the expected reward under a parameterised policy:

$$J(\theta) = E_\pi[r(\tau)]$$

$$= \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s, a)$$

The gradient of this objective function is then used for the gradient descent (ascent) update and is computed as follows:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s, a)$$

$$\propto \sum_{s \in S} d^\pi(s) \sum_{a \in A} Q^\pi(s, a) \nabla_\theta \pi_\theta(a|s)$$

Multiplying and dividing by $\pi_\theta(a|s)$ inside the inner summation term, we get:

$$= \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s, a) \nabla_\theta \pi_\theta(a|s) / \pi_\theta(a|s)$$

$$= E_\pi[Q^\pi(s, a) \nabla_\theta \ln \pi_\theta(a|s)]$$

This can now be represented as an update equation for the policy (i.e. loss update to train the policy neural network) as follows:

$$\theta_{t+1} = \theta_t + \alpha(Q^\pi(s, a) \nabla_\theta \ln \pi_\theta(a|s))$$

V. Methods

The Baseline we chose for making the task of making the bipedal robot in Humanoid-v2 move without falling over is to take randomly sampled actions from the action space and collect the rewards using this completely "random" policy. This is a very standard approach in Machine Learning to generate baselines as any other algorithm we will write is expected to perform better than this "dumb" baseline approach. It is expected that the various models utilising different policies and characterization of observation and action space will improve upon the performance of the baseline model.

A. Tabular Q Learning (ϵ -greedy)

Since our state and action space are high dimensional continuous spaces we need to discretize and reduce the dimensionality in order to implement Tabular Q-Learning. In order to discretize the state and action spaces we used K-Means algorithm. We initially ran the random baseline algorithm and obtained a list of valid state action pairs. Then, we performed clustering on the state space and action space using the K-Means algorithm. Now, for any given input state it is assigned to one of the clusters and the Q-Learning algorithm is run with the center of that cluster as the current state in place of the input state. The Q table is initialised as zero and it gets updated with each state-action pair that is utilised. For ϵ - greedy Tabular Q Learning, it chooses a random action ϵ times and follows the learnt policy which leads to choosing an action that maximizes the reward for the given state the rest of the times. After taking the action, we observe the reward and the next state. Then we update the Q table as:

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S',a) - Q(S,A)]$$

where S is the current state, A is the Action taken from state S using ϵ greedy policy, R is the reward observed on taking the Action and S' is the new state. There are two hyper-parameters involved α which is the Learning rate and γ which is the exploration factor.

B. Q Learning with Function Approximation using Multi Layer Perceptrons (MLP)

Q learning using neural network functions instead of tables (as we did in the first approach) is a logical extension to this problem of physical control task, given that we are dealing with continuous state space.

In this setting, the Q values are now estimated using neural networks instead of querying a table every time. But, this technique cannot be straightforwardly applied here as we have to rely on finding actions that maximise the Q value function, which in continuous action space is a challenging task.

In order to adapt Q Learning to this task, we thus applied discretisation on the action space, dividing the possible actions in each dimension (i.e. the available degrees of freedom) into 21 bins and acted in this discrete action space instead of the full continuous space (which would have been an intractable problem).

We then tried out three different variants of Q learning (with function approximation using MLPs), given this discrete space preprocessing:

1) Basic Function Approximation using MLP (No Optimisation): First, we replaced the tables with an MLP. The state is given as the input to the MLP and the Q-value of all possible actions is generated as the output. At train time, the agent follows an epsilon-greedy policy to pick actions at each step, and the next state observed is used to update the weights in the network using the bellman equation.
   Mathematically, the back-propagated loss in the Q_network (MLP) is the L1 loss computed between target value ($r + \gamma \max_a Q^*(s', a)$) and $Q(s, a)$. Ref Algorithm-1.

Algorithm 1 Basic Function Approximator

for iteration $= 1, 2, \ldots$ **do**
  for episode $= 1, 2, \ldots, N$ **do**
    while not terminal_state **do**
      Select an action a based on ϵ-greedy policy π
      Obtain (s, a, s', r, done) tuple using action a
      Forward pass in q_net to obtain Q(s, a) batch.
      Forward pass in q_net to get batch of

$$\max_{a^*} Q(s', a^*)$$

      Back propagate the batch loss using:

$$||r + \gamma \max_{a^*} Q(s', a^*) - Q(s, a)||$$

      through the q_network.
    end while
    ϵ = ϵ $\times$ decay_rate
  end for
end for

2) Function Approximation using NFQ Agent Optimisation: In this approach, using Riedmiller M. et al. [9] as reference, we implemented a replay buffer to store (s, a, $s^{'}$, r, done) tuples for episode rollouts, which when sampled randomly, are uncorrelated. These are then used after said random sampling, in a sort of supervised fashion for batchwise Bellman update, which improves stability of the training compared to the unoptimised version stated before. Ref Algorithm-2.
3) Deep Q Learning (DQN) using replay buffer and target network optimisation: Using Mnih, Volodymyr, et al. [8] as reference, we implemented a target network in addition to the replay buffer. This target network has the same architecture as the q-value network, but is not independently trained. Instead, it gets updated with the weights of the q-value network periodically. Since the target is fixed for each batch of rollouts being used, the algorithm training is much more stable and performs better than NFQ. Ref Algorithm-3.

C. Vanilla Policy Gradient

Vanilla Policy Gradient is an on-policy Policy Gradient algorithm. The main idea behind Vanilla Policy Gradients is to maximize the likelihood of actions that will lead to high reward return and minimize the likelihood of actions that will lead to low return. At each iteration, the Vanilla Policy Gradient method samples N trajectories under the current policy. As the policy π in Policy Gradient Methods is parameterized (with parameters θ), the effect of changing the parameters θ on the reward obtained is captured by infinite-horizon cumulative return where the future returns are discounted. Then, the benefit of taking an action is estimated

Algorithm 2 NFQ [9]

for iteration $= 1, 2, \ldots$ **do**
  for episode $= 1, 2, \ldots, N$ **do**
    while not terminal_state **do**
      Select an action a based on ϵ-greedy policy π
      Obtain (s, a, s', r, done) tuple using action a
      Store the tuple in R (experience replay buffer)

      if buffer_size >= batch_size **then**
        Sample batch_size randomly from the buffer.
        Forward pass in q_net to obtain batch of Q(s, a) .
        Forward pass in q_net to get batch of

$$\max_{a^*} Q(s', a^*)$$

        Back propagate the batch loss using:

$$||r + \gamma \max_{a^*} Q(s', a^*) - Q(s, a)||$$

through the q_network.
      end if
    end while
    ϵ = ϵ $\times$ decay_rate
  end for
end for

Algorithm 3 DQN [8]

for iteration $= 1, 2, \ldots$ **do**
  for episode $= 1, 2, \ldots, N$ **do**
    while not terminal_state **do**
      Select an action a based on ϵ-greedy policy π
      Obtain (s, a, s', r, done) tuple using action a
      Store the tuple in R (experience replay buffer)

      if buffer_size >= batch_size **then**
        Sample batch_size randomly from the buffer.
        Forward pass in q_net to obtain Q(s, a) batch.
        Forward pass in target_q_net to get batch of

$$\max_{a^*} Q_{\text{target}}(s', a^*)$$

        Back propagate the batch loss using:

$$||r + \gamma \max_{a^*} Q_{\text{target}}(s', a) - Q(s, a)||$$

through the q_network.
      end if
    end while
    ϵ = ϵ $\times$ decay_rate
  end for
  Copy over the weights from q_network to target_q_network
end for

using advantage functions i.e. how good or bad an action is compared to a default action under the policy. Advantage Functions are used in place of reward return estimates as they help to reduce the variance in estimates. Since the current literature suggests that for high dimensional continuous control tasks Generalized Advantage Estimation(GAEs) work well, we implemented them in our implementation of Vanilla Policy Gradient [2]. Then the policy gradient is estimated and the policy is updated using gradient ascent. As we are updating the policy in the direction of the policy gradient, actions sampled under this updated policy are likely to lead to higher rewards. After updating the policy, the value function is fitted on the the expected future rewards using gradient descent algorithm. Ref Algorithm-4. [12]

Algorithm 4 Vanilla Policy Gradient Algorithm [3]

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for k = 0,1,2,... **do**
  Collect set of trajectories $\mathcal{D}_k$ = $\{\tau_i\}$ by running policy π_k = $\pi(\theta_k)$ in the environment
  Compute rewards-to-go $\hat{R}_t$
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}
  Estimate Policy Gradient as

$$\hat{g_k} = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$$

  Compute Policy update using standard gradient ascent

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g_k}$$

  Fit value function by regression on mean -squared error

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{H} (V_\phi(s_t) - \hat{R}_t)^2$$

  using gradient descent algorithm
end for

D. Proximal Policy Optimisation(PPO)

PPO (Proximal Policy Optimisation) is a Policy Gradient, on-policy algorithm that directly optimises the policy an agent should follow to maximise the expected reward. While performing the policy update, we want to take the largest step that leads to improvement in policy without straying too far from the current policy which could lead to a future decrease in performance. While TRPO (Trust Region Policy Optimization) [13] algorithm also tackle the same problem use trust regions instead of line searches, they are relatively harder to implement as compared to PPO algorithm. There are two classes of PPO implementations which use penalty term to perform a constrained update and clipping of the objective

function to prevent the new policies from diverging too far from the old policy. PPO algorithm operate in an actor-critic style, alternating between sampling from the environment and optimising a "clipped" objective function. In order to further prevent the new policy from diverging too far from the old policy early stopping is performed. If the KL divergence of the new policy from the old goes beyond a certain threshold, we perform early stopping. The rest of the algorithm is similar to Vanilla Policy Gradient and here as well we used Generalised Advantage Estimation (GAE) function while calculating the advantage estimate. Ref Algorithm-5. [12]

Algorithm 5 PPO-Clipping with early stopping based on approximate KL Divergence [7]

Input: Initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectories $\mathcal{D}_k$ = τ_i by running policy π_k = $\pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi k}$
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1}=\arg\max_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta k}(a_t|s_t)} A^{\pi_{a_k}}(s_t,a_t), g(\epsilon, A^{\pi_{a_k}}(s_t,a_t))\right)$$

  typically via stochastioc gradient ascent with Adam
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2$$

  typically via some gradient descent algorithm
end for

Algorithm 6 DDPG [10]

Input: initial policy parameters θ_0, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
while Not Converged **do**
  Observe state s and select action a = clip($\mu_\theta(s) + \epsilon, a_{\text{Low}}, a_{\text{High}}$), where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  if it's time to update **then**
    for however many updates **do** Randomly sample a batch of transitions, B = (s,a,r,s',d) from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r = \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta \text{targ}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s,a) - y(r,s',d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s) \in B} (Q_\phi(s, \mu_0(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$$

$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

    end for
  end if
end while

E. Deep Deterministic Policy Gradients (DDPG)

Using DQN with discretized action space needlessly throws away useful information about the action domain, which leads to subpar results. DDPG [10] is an actor-crtic algorithm that adapts the ideas of DQN to high-dimensional, continuous action spaces. Two neural networks - Policy Network and Q-value network are maintained along the lines of actor crtic methods. But here, further optimisation is done to ensure better results. Drawing heavily from the DQN paper [8], a replay buffer of transitions sampled from the environment is maintained and at each training timestep, a minibatch is sampled to update the actor (policy network) and critic (Q value network). The concept of target networks is also used (a extra network for each actor and critic), but with "soft" target updates instead of hard replacement of weights (polyak update). Ref Algorithm-6.

VI. Implementation Approach

We implemented the Random Baseline, Tabular Q-Learning, Basic Function Approximator using Multi-Layer Perceptron (without optimisation), Function Approximation with NFQ Agents and Deep Q Learning from scratch in pyTorch. For performing dimensionality reduction in Tabular Q-Learning using K-means algorithm we used Scipy's K-Means implementation. However, for the Policy Gradient algorithms (VPG, PPO and DDPG) which are harder to implement independently, we built upon the open source code (i.e. OpenAI Baseline implementation [12] and the Deep RL implementation [12] repositories). We have converted them into jupyter notebook form, modified wherever appropriate and tuned hyperparameters that gave best results, according to our problem statement.

A. Hyperparameter Tuning and effects

Random Baseline had no hyperparameters to tune as it is just taking random actions.

For Tabular Q-Learning we first had to select the right number of clusters for dimensionality reduction. Our input dataset of state-action pairs generated by following Random Baseline model was clustered in both state space and action space consisted of 10,000 observations. Our current implementation contains 1000 clusters on state space and 100 clusters on action space. What was interesting to note while we were training the hyperparameters was that choosing a cluster size of 100 on state space and 10 on action space also gave similar results. We have still kept a cluster size of 1000, 100 as we think that it captures more granularity of state action space for clustering.

We also explored different learning rates(α), discount factor(γ) and exploration proportion(ϵ) but the model still performed equally good/bad as Random Baseline model nevertheless. We decided to go ahead with the following values α= 0.01, γ= 0.99, ϵ = 0.99 so as to provide a level playing ground for comparing the various algorithms.

In VPG we kept the value of discount factor(γ) =0.99 and GAE factor smoothing factor (λ)=0.97 as suggested by the papers [2] [15]. The policy update learning rate(policy_lr) was set at 1e-3 and the value function update learning rate(vf_lr) was also set at 1e-3.

In PPO we kept the value of discount factor(γ) =0.99 and GAE factor smoothing factor (λ)=0.97 the same as for VPG. The policy update learning rate(policy_lr) was reduced to 3e-4 as we are clipping the objective function and don't want the new policy to deviate too much from the current policy. The value function update learning rate(vf_lr) was set at 1e-3, the clipping parameter(clip_param) was set at 0.2 and the approximate kl divergence between the target policy and current policy(target_kl) was set as 0.01 [16]

In Table-1 we have summarised the final Hyperparameters used in the various algorithms we tried out in this project.

TABLE I: Summary of Hyperparameters used for each algorithm

RL Algorithm	Hyperparameter values
Random Baseline	-
Tabular Q-Learning	No. of clusters on state space= **1000**, 100, No. of clusters on action space =**100**,10 $\alpha = 0.01, \gamma = 0.99, \epsilon = 0.99$
Q-Learning Function Approximator using MLP (no optimisation)	Bin size for action space discretisation = 21, γ=0.99, ϵ = 0.9, ϵ_decay_rate = 0.9965, max steps per episode = 1000, learning rate = 7.5e-4
Q-Learning with NFQ	Bin size for action space discretisation = 21, sampled batchsize from replay buffer = 64, γ=0.99, ϵ = 0.9, ϵ_decay_rate = 0.9975, max steps per episode = 1000, learning rate = 1e-5, replay buffer max size = 1e5
DQN	Bin size for action space discretisation = 21, sampled batchsize from replay buffer = 64, γ = 0.99, ϵ = 0.9, ϵ_decay_rate = 0.9975, max steps per episode = 1000, learning rate = 9e-6, replay buffer max size = 1e5, target network update frequency = 100
VPG	γ=0.99,λ=0.97,policy_lr=1e-3, vf_lr=1e-3, max steps per episode = 1000
PPO	γ=0.99, λ=0.97,policy_lr=3e-4, vf_lr=1e-3,clip_param=0.2,target_kl=0.01, max steps per episode= 1000
DDPG	γ = 0.99, action noise=0.1, buffer_size = 1e4 batch_size = 64, policy network learning rate = 3e-4, qf network learning rate = 3e-4, exploration_before constant = 2000, train_after constant = 1000,

VII. Experiments and Results

We trained all our models for about 3000 episodes due to constraints on training time/resources and to obtain a general trend on which algorithm performs better. In order to obtain state of the art results we would need to train the algorithms for around 10,00,000 episodes (according to openAI documentations) but this was not feasible in the time-frame we had. So we focused on getting the general trends for training and evaluation. We believe that the trends in performance will continue to be the same as training time and number of episodes are increased.

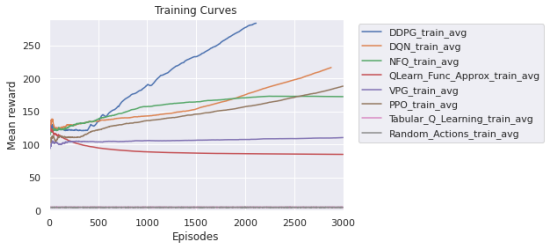
For generating the plots, we have used the below scheme:

For every iteration, we run a "training" phase and an "evaluation" phase. For training, we allow the agent to execute epsiodes until it has taken a max steps_threshold of 4000 steps (i.e actions) overall in the environment. Hence, the actual number of episodes that the agent executes during this phase actually depends on how successful it was during the said episodes (if it was more successful in an episode, it took more steps in the environment before falling off, so subsequently, it will get lesser episodes to run afterwards). Since there are 30 iterations of train-eval loop, we get roughly 3000 episodes to plot on the x-axis. Interestingly, the graph of DDPG gets cut off before even 2000 episodes are reached (Ref Fig 2), this is because it performed so well that the steps_threshold got reached beforehand. We maintain a running average of the rewards we get per episode during training and store them for plotting purpose.

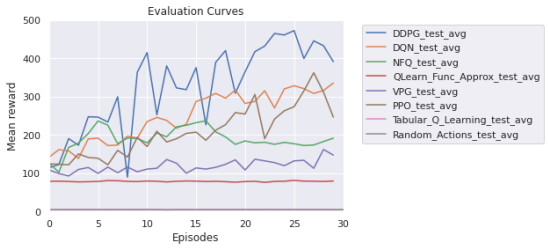
In each iteration, after training phase completes, i.e. the agent has taken close to steps_threshold of 4000 steps in the environment, we execute a evaluation phase, in which the agent executes 10 episodes and collects reward under the currently learned policy. The average reward for these 10 episodes/epoch is then stored for plotting purpose. So, for 30 iterations, we get 30 plot points on the x-axis, with each point being the average reward obtained in 10 episodes executed with that instance of learned policy. So this plot successfully shows how well the agent has learned after each iteration of training.

VIII. Conclusions and Discussions

The episode vs reward plots for training and evaluation phases have been presented in a consolidated form (for all the

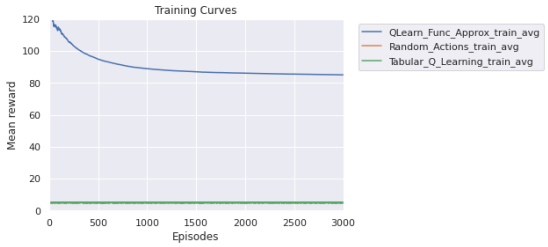


(a) Training Curves

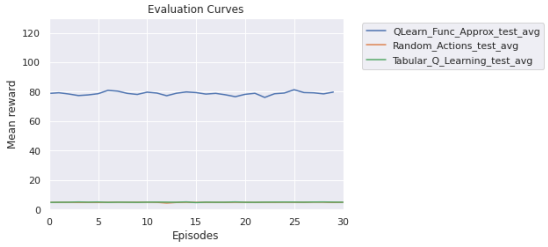


(b) Evaluation Curves

Fig. 3: Comparison of all RL algorithms

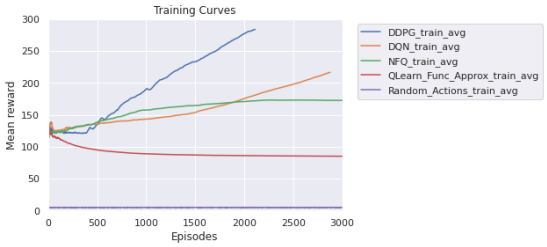


(a) Training Curves

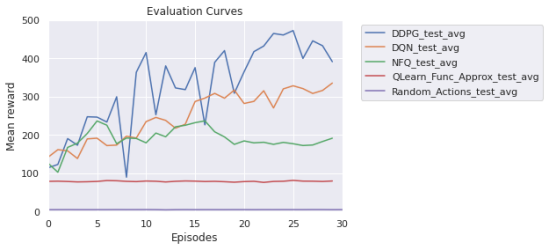


(b) Evaluation Curves

Fig. 4: Comparison of Random Baseline, Tabular Q-Learning and Function Approximation Algorithm (Basic Q-Learners)

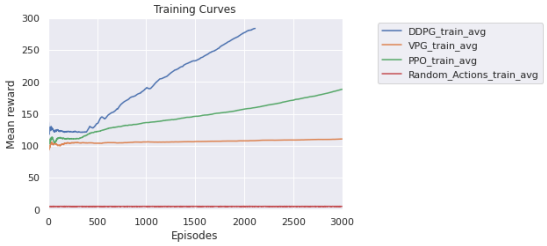


(a) Training Curves

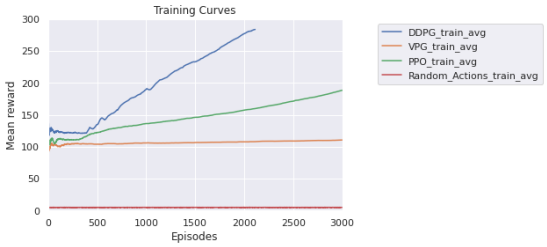


(b) Evaluation Curves

Fig. 5: Comparison of Random Baseline, DQN, NFQ, DQN and DDPG (Deep Q Learning methods)



(a) Training Curves



(b) Evaluation Curves

Fig. 6: Comparison of Random Baseline, VPG, PPO and DDPG (Policy Methods)

algorithms) in Fig 3 a and b. We have also separately plotted three different categories - Simple Q-Learning methods (Fig-4 a,b), MLP based function approximator methods (Fig-5 a,b) and policy based methods (Fig-6 a,b) to show trends among similar classes of algorithms. Baseline has been included in all the plots to get an idea of individual algorithm performances when compared with the baseline.

Lastly, we report a table summarising the final performance (rewards) of all the algorithms after training completion in Table-2.

Our results made qualitative sense as we found expected trends among the various algorithms that we tried out for this continuous control task of making a humanoid walk.

As expected, all our neural network based models performed better than the random-action baseline. Tabular Q-learning strongly assumes discretised state and action spaces, which in this environment threw away too much valuable information even after we tried out tricks like K-Means clustering (to discretise better), so it could not outperform the baseline.

TABLE II: Summary of RL Algorithms and their performance on Humanoid-v2 env

RL Algorithm	Reward[a]
Random Baseline	4.974
Tabular Q-Learning	5.119
Q-Learning Function Approximator using MLP (no optimisation)	79.789
Q-Learning with NFQ	191.585
DQN	335.623
VPG	147.348
PPO	246.502
DDPG	458.497

[a]Reward obtained after training for 30 epochs of $\sim$ 4000 env steps (3000 episodes)

Among the neural network models, we again saw how optimisation led to better results among the Q learning function approximator algorithms, with DQN doing the best and unoptimised algorithm doing the worst. Interestingly, basic Q-function approximator using MLP (with no optimisation) settled at a reward value lower than what it initially achieved (i.e diverged). But this is actually not outside of expectation as there is high instability in directly using Neural Nets as q-value approximators without any added optimisation like replay buffer or target networks, so there is actually a very high chance of said divergence.

Among Policy based methods, DDPG and PPO outerperformed Vanilla Policy Gradients. DDPG is actually the best performer in the group of all the algorithms we tried out. This is consistent with the state of the art results for this problem, as actor-crtic methods outerperform others in OpenAI baseline documentations. We did expect PPO to perform at par with DDPG which it did not, but if we look at the rising trend in the plot and extrapolate, PPO's performance is actually expected to improve with enough training episodes, which is a good sign.

LEARNINGS

From this project, we realised that Reinforcement Learning is a very rapidly-evolving field, and every algorithm we came across builds heavily on prior algorithms, with major/minor modifications to improve the performance. These modifications keep building on top of each other, and when trying out a new problem, it's hard to untangle the web and figure out which approach/modification would work best for the current problem. We tried our best to take this into account, and actually tried out three different modifications for Q Learning function approximator algorithms, and three for policy based methods to see if results made sense, which they did, as explained in the previous section. This helped build the intuition behind the different family of algorithms, as well as the math that made it all work.

But due to time constraints, we were unable to explore and exploit this fully, especially for policy gradient methods, where there is a myriad of modifications over prior algorithms in every other paper. This requires an even more extensive literature review, along with lots of attention to the math behind, so it was out of scope for this project.

FUTURE WORK

For function approximator algorithms using MLP that we implemented, we discretized the action space to make the problem tractable (i.e. to compute argmax over action space). But this resulted in unnecessary loss of information. One way to tackle this issue of computing argmax on continuous action space is to allow actions as input to the neural network as well, and then gradient descent over the action space at each step of training to determine the argmax value. This is a little hard to implement, which is why we did not tackle it as part of this project, but this is an interesting future work to try out. We would also like to explore and try out the various actor-crtic methods in future like the soft actor critic etc, as these are the current state of the art in continuous control tasks.

TABLE III: Summary of Symbols and Notations used in the paper

Symbol/Notation	Meaning
s, S	Current State/Observation
a, A	Action
s', S'	Next State/Observation
r, R	Reward
$done$	Terminal state flag
π	Policy
π_θ	Parametrised Policy
$Q(s,a)$	Q Value function
$V(s)$	V value function
$A(s,a)$	Advantage Estimate
θ	Policy Parameters
ϕ	Value Function Parameters
α	Learning Rate
γ	Discount Factor
ϵ	Exploration Proportion
H	Horizon
$\mathcal{D}_k$	Set of trajectories under policy π and policy parameter $_k$

ACKNOWLEDGMENT

We would like to thank Dr. Lyle Ungar for making this course so fun and for giving us the chance to work on this project. His expert opinions on some of the things we were trying out during the project gave us direction and motivation. We would also like to thank our TAs - Zhangkaiwen Chu and Kevin Li for their support and guidance throughout the course of this project.

REFERENCES

[1] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[2] Schulman, John, et al. "High-dimensional continuous control using generalized advantage estimation." arXiv preprint arXiv:1506.02438 (2015).

[3] Duan, Yan, et al. "Benchmarking deep reinforcement learning for continuous control." International conference on machine learning. PMLR, 2016.

[4] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." NIPs. Vol. 99. 1999.

[5] Kidziński, Łukasz, et al. "Artificial intelligence for prosthetics: Challenge solutions." The NeurIPS'18 Competition. Springer, Cham, 2020. 69-128.
[6] Peters, Jan, and Stefan Schaal. "Policy gradient methods for robotics." 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2006.
[7] Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).
[8] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).
[9] Riedmiller M. (2005) Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In: Gama J., Camacho R., Brazdil P.B., Jorge A.M., Torgo L. (eds) Machine Learning: ECML 2005. ECML 2005. Lecture Notes in Computer Science, vol 3720. Springer, Berlin, Heidelberg.
[10] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).
[11] https://github.com/dongminlee94/deep_rl
[12] https://spinningup.openai.com/en/latest/
[13] Schulman, John, et al. "Trust region policy optimization." International conference on machine learning. PMLR, 2015.
[14] Schulman, John. Optimizing expectations: From deep reinforcement learning to stochastic computation graphs. Diss. UC Berkeley, 2016.
[15] Gu, Shixiang, et al. "Q-prop: Sample-efficient policy gradient with an off-policy critic." arXiv preprint arXiv:1611.02247 (2016).
[16] https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe