Final Project: Cart Pole using Function Approximation

Sherwyn Braganza

December 12, 2022

Summary of Contents

1 The Background and Introduntion

The aim of this project was to implement a Reinforcement Learning Algorithm that was able to solve the Cart-Pole Problem. The Cart-Pole problem involves balancing a pole attached to a cart that can be moved linearly along one dimension (Sutton, Barto (2019) - Reinforcement Learning: An Introduction). Figure 1 provides a representation of what the environment looks like. The aim of this game is to keep the pole up indefinitely and ends if the pole drops or the cart goes out of bounds.

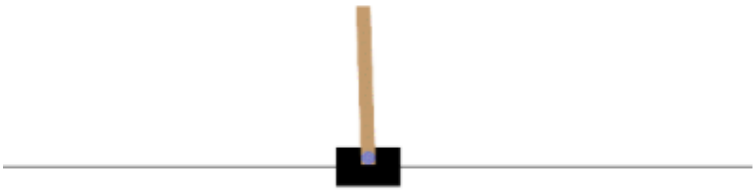


Figure 1: Cart Pole Environment

2 The Simulator

I used OpenAI Gym's Classic Control - Cart Pole v1 module as my simulator. This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson in "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem". A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart (Figure 1).

2.1 The Action Space

The action is a numpy.ndarray with shape (1,) which can take values 0, 1 indicating the direction of the fixed force the cart is pushed with (Table 1).

Num	Action
0	Push cart to the left
1	Push cart to the right

2.2 The State Space

The observation is a numpy.ndarray with shape (4,) with the values corresponding to the following positions and velocities:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	- 5 units/s	5 units/s
2	Pole Angle	- 0.42 rad	0.42 rad
3	Pole Angular Velocity	-5 rad/sec	5 rad/sec

1. The cart x-position (index 0) can be take values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range.
2. The pole angle can be observed between (-.418, .418) radians (or ±24°), but the episode terminates if the pole angle is not in the range (-.2095, .2095) (or ±12°)

2.3 Reward Space

The original reward space was: **1** for every step taken and **0** if the state was terminal (the cart was out of bounds or the pole fell down).

With testing and trying out different reward parameters, I found that **a reward of 0** for every step taken and **a reward of -1** for terminating worked best for training the model.

3 The Learner

For my learner, I used **SARSA Lambda with Linear Function Approximation**. I chose the **Fourier Function Approximator** to apporximate my state-value function.

3.1 SARSA Lambda

The rough algorithm for SARSA(λ) that I followed is shown below in Figure 2.

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose $A' \sim \pi(\cdot|S')$ or ε-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
    $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
    $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
    $\delta \leftarrow R + \gamma Q' - Q$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\right)\mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    $Q_{old} \leftarrow Q'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    $A \leftarrow A'$
  until S' is terminal

Figure 2: Online SARSA (λ)

w is your weight vector and **x** is the function approximation (in this case, a Fourier function approximation) for a given state and action.
α is the learning rate, λ is the trace decay rate and ϵ is a control factor that tells the algorithm whether it should follow the policy or take a exploratory action.

There were however a few changes made to this algorithm so that it was tailored to the requirements of this project. A new status - stable state - was achieved when the algorithm reached 3000 steps. The end of an episode was achieved when either:

1. The state of the cart was outside the state bounds.
2. The system reached a stable state.

3.2 The Function Approximator

I used the method of Fourier Function Approximator described in Value Function Approximation in Reinforcement Learning using the Fourier Basis by Konidaris et. al to predict the Q values for a given state-action pair. The function took in two instantiation variables - the number of states and order that corresponded to the order of the Fourier basis function approximation. Using this it created a $(M+1)^2$ x N matrix (**C**), where N is the number of states and M is the order of the Fourier approximation. The **getFourierBasisApprox** function in my code used Equation 1 to get ϕ or w or the Fourier basis which was then dotted with the weight vector in the SARSA algorithm to get the $\boldsymbol{Q}$ for a given state-action.

$$\phi_i(x) = \cos(\pi \mathbf{c}^i \cdot \mathbf{x}) \tag{1}$$

In my implementation of the Fourier Function Approximator, I chose 4 variations of the function approximator - and order 3, order 5, order 7 and order 9 function approximators. Function approximators above order 9 get computationally intensive and are difficult to train.

4 Analysis and Results

My implementaion had two running modes:

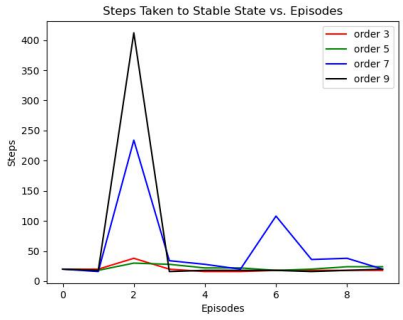
1. **Visual Mode** - Runs the implementation and brings up a visual of of the cart-pole balancing experiment with order 5 Fourier Function Approximation. Run instructions: *python3 agent.py visual*
2. **Analysis Mode** - Runs the implementation without a GUI and performs tests for different order function approximators, learning rates, epsilon values. Run instructions *python3 agent.py*

As mentioned above, the analysis mode performed experiments using different hyperparameters and the results obtained are shown below. In our case here, a learning curve represents the number of time steps achieved by the learner during an episode. The plots below show the steps per episode for different hyper-parameters.

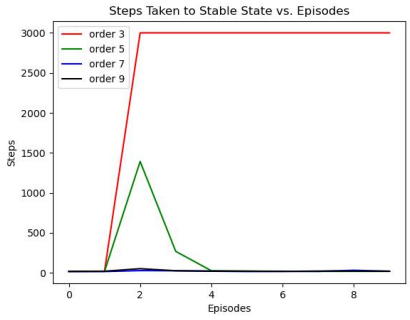
The algorithm seems to learn very fast once (within 3 episodes) for the restricted state space once configured with the best learning rate and epsilon values.

4.1 Variable α, ϵ = 0.0 (Purely On-Policy)

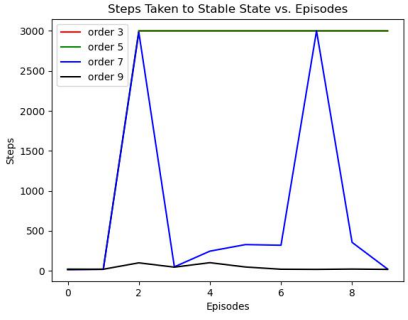
The results obtained below are shown for a variable learning rate, with other hyperparameters held constant ($\epsilon = 0.0$, $\gamma = 0.99$, $\lambda = 0.95$).



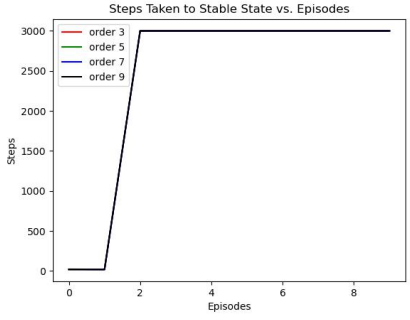
(a) Learning Curve, alpha = 0.1



(b) Learning Curve, alpha = 0.01



(c) Learning Curve, alpha = 0.003

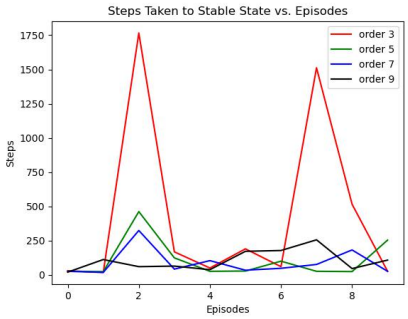


(d) Learning Curve, alpha = 0.001

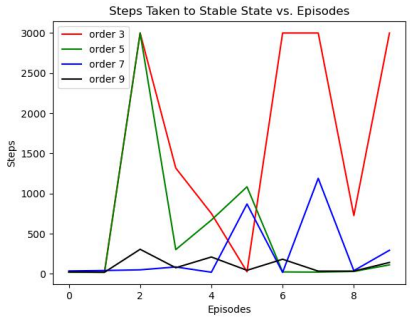
The higher order models seem to learn really well with a smaller learning rate. This kind of seems to make sense the number of coefficients in play are 4 x $(state\_space\_size)^{order}$. The increase in coefficients in the Fourier approximation is tied to an increase in precision but also a higher chance of overshooting the best approximation.

4.2 Variable ϵ, α = 0.001

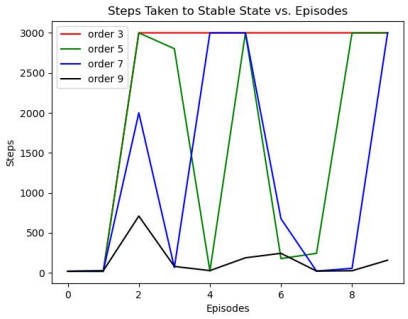
ϵ is the policy control variable. When ϵ is 0, the performance is purely on-policy in which the policy followed is ϵ-Greedy. When ϵ is 1, the performance is purely off-policy in which every action chosen is a random action. The graphs below show the changes in the learning curve with different values of ϵ.



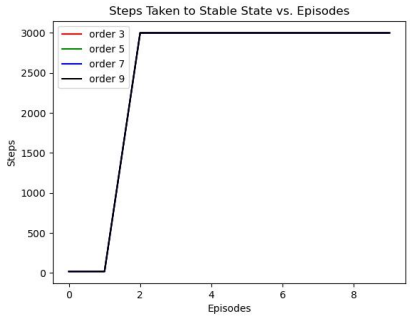
(a) Learning Curve, epsilon = 0.5



(b) Learning Curve, epsilon = 0.3



(c) Learning Curve, epsilon = 0.15



(d) Learning Curve, epsilon = 0.0

What I learnt from changing ϵ values is that the cart-pole experiment is very sensitive to non-policy actions. The learner is unable to recover if too many detrimental random actions are performed. An ϵ = 0.5 means that every second action is off policy and hence could be a detrimental action.

4.3 Changing γ and λ

Changes made to lambda and gamma do not really affect the learning curve by a lot for fixed α = 0.001 and ϵ = 0.0. The main reasoning behind this is that any step (barring the terminal state landing step) results in a reward of 0; this results in a null trace update. The only time it makes a difference is at the terminal state, where the reward is -1. The backpropagation of this reward, although not much in magnitude has a huge effect mainly because of the sign change.

5 Comments and Future Work

This entire project is available to clone here.

The future improvements to be implemented to the project are as follows:

1. GUI for choosing modes
2. Object Oriented Programming Implementations
3. State Space bound expansion
4. Radial Basis Function Approximation
5. Polynomial Basis Function Approximation
6. Deep Learning Function Approximation
7. Performance comparisons between different Function Approximators
8. Expansion to Q-Learning based Learners

6 References

1. Sutton, R., and Barto, A. 1998. Reinforcement Learning: An In- troduction. Cambridge, MA: MIT Press.

2. Aleida, L.; Langlois, T.; Amaral, J.; and A.Plakhov. 1998. Param- eter adaptation in stochastic optimization. In Saad, D., ed., On- line Learning in Neural Networks. Cambridge, MA: Cambridge University Press. 111–134.

3. Open AI Gym Classic Control Documentation, *'https://www.gymlibrary.dev'*

4. Konidaris, G., and Osentoski, S. 2008. Value function approxima- tion in reinforcement learning using the Fourier basis. Techni- cal Report UM-CS-2008-19, Department of Computer Science, University of Massachusetts, Amherst.