# Preface

This work was done in collaboration with the Institute for Human and Machine Cognition (IHMC) and is a continuation of the project done by Bart Keulen. His master thesis on this topic can be found here and the source code for his original project can be found here.

In his master thesis, Keulen built the concept of Smart Start around discrete state and action spaces, and in this project, it is extended to continuous state and action spaces. This document is for documenting the main ideas of the algorithms used in this project and the process of creating the project. The section Project Progression acts like a diary of the project, and will link to more detailed portions of the document about the specific algorithms.

Also note that this document won't be sufficient in teaching everything related to the project. It is more of an super rough summary of the related information. If you already know the material hopefully it should serve as a good refresher. If it isn't familiar the document can hopefully show you what keywords to search up and point you in the right direction of where to start learning.

# Project Progression

## 1.1 Smart Start and Reinforcement Learning Basics

Before starting this project, it was necessary to understand the basics of reinforcement learning and the original Smart Start paper that Keulen wrote. Keulen's project is built for discrete state and action spaces, so to begin on this project it was necessary to be familiar with the discrete algorithms as well as the problem of exploitation vs. exploration which Smart Start is meant to address. The problem of exploitation vs. exploration is best demonstrated in environments with sparse rewards or misleading rewards. Sparse rewards are where most of the transitions near the start state give no reward, making the agent wander randomly until it randomly reaches a good reward. Misleading rewards is one where there is a small reward close by, making the agent exploit this easy low reward, where a far greater reward farther away is still present. Keulen tested his smart start framework with a Gridworld Maze with sparse rewards and got significant improvements especially on far more sparse environments.

One of the first things I did was add a smart start visualizer, which allows for visualization of which discrete states were chosen as smart start states.

## 1.2 Continuous Reinforcement learning

To begin with Smart Start Continuous, we needed to formulate similar problems in continuous domains. The environment used in this project was the Mountain Car Continuous v0 environment from OpenAi's `gym` python package. This environment seen in figure 1 has sparse reward (only positive reward is reaching the top of the hill), and the car itself doesn't have enough power to go directly to the top of the hill, and must build up momentum by going back and forth. If the agent does not discover the reward at the top of the hill, it will remain at the bottom because it is slightly penalized for accelerating forward and backward.

For testing Smart Start we need a base continuous algorithm to compare the results

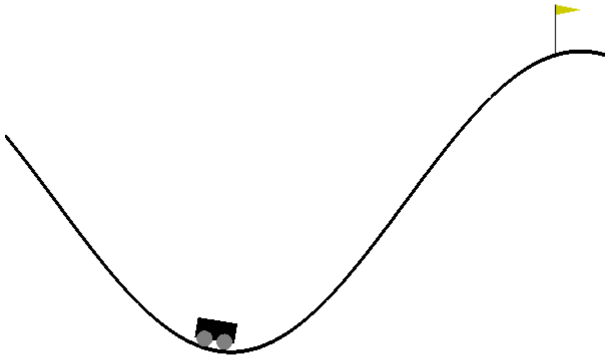

Figure 1: Mountain Car Continuous Environment

before Smart Starts and after Smart Starts. Researching popular continuous reinforcement learning algorithms led me to find some of the earliest algorithms used for continuous state and action spaces. First is Policy Gradient methods where you would initialize a random policy (a policy determines what actions to perform at what state) which relies upon some parameters $\theta$. After using the policy a bunch of times, you would use small perturbations in each dimension of $\theta$ to estimate the $\nabla J(\theta)$ where $J(\theta)$ represents the average total reward in the environment using a policy with parameters $\theta$. With the gradient, we perform hill climbing to change $\theta$ to maximize reward. There exists modifications to calculate the gradient analytically rather than through random sampling. This whole process along with the analytical solution are described well in this video.

After that we have Deep-Q learning which uses a Neural Network that takes in a state, and outputs an approximated Q-value which we can use similarly to how Q-learning works covered in Reinforcement Learning Basics. This Neural network is trained using the bellman updates for Q-learning (instead of temporal difference learning, the difference is converted into loss, which is minimized with gradient descent). These last 2 algorithms saw some results, but had many drawbacks and often weren't able to learn too well/reliably.

The base algorithm that was chosen for the experiments in the project was Deep Deterministic Policy Gradient (DDPG) which is an actor-critic approach that had very promising results in the world of continuous reinforcement learning. Many algorithms would work (Keulen suggest also looking at Natural Advantage Functions) but remember from Smart Start Discrete that our base algorithm must be able to provide a state-value for each given potential smart start (an evaluation of the expected episode reward after visiting a certain state).

### 1.3 Continuous Smart Start Selection

Recall that this is the first part of any Smart Start episode. This uses the Upper Confidence Bound (UCB) algorithm for solving the multi-armed bandit problem where we have $N$ different slot machines and we figure out based on the past, what slot machine to pull next. In terms of Smart Start, all visited states in the past are slot machines, base algorithm's state-value evaluation of that state is the expected reward of that slot machine, and the number of times that state has been visited is the number of times that machine has been pulled. The UCB algorithm takes in the state value and visitation count for each state. Getting the state value is dependant on the base algorithm, but it's more difficult for visitation count.

In a continuous state space, you virtually will never visit the same state twice: even with similar actions instead of being at state (1,1) you might end up at state (1.0001,1.0002). Plus you would have an infinite number of states to keep count for. Instead this project, as recommended by Keulen, uses Kernel Density Estimation to estimate visitation count. To decide the visitation count of one state using Kernel Density Estimation, you must compare that state to all other states previously visited (or in the replay buffer) making the computation time $O(n^2)$, so this project has smart start set a variable $n_{ss}$ which specifies the number of smart start states to consider each time, making the computation time grow linearly.

### 1.4 Continuous Smart Start Navigation

Navigating to the Smart Start State is a reinforcement learning problem in itself, so it was rather difficult to figure out a good way to go about doing this. Keulen recommended a method called Iterative Linear Quadratic Regulator which seems to stem more from control theory rather than reinforcement learning. As a result I was not able to thoroughly understand the details of the algorithms used here and here.

In the end I used a Neural Network Dynamics with Model-Based Predictive Controller to navigate to a smart start state. Full details of how it works are in that section linked above. There are a decent number of ways it can be improved from here.

### 1.5 Continuous Smart Start Results

### 1.6 Future Considerations

Improve the Neural Network Dynamics Model Predictive Controller (implemented in `NND_MB_agent.py`)'s reward function. Currently adds up reward over total *horizon* steps, but should just do reward of **last** step!!!

# Reinforcement Learning Basics

Reinforcement learning is a type of machine learning suited to solve a specific type of problem. The algorithm is supposed to control an agent (think of an agent like a video game character) who is able to observe its current state (think of it as the character's location, level, health, etc.), able to take actions (such as moving left, attacking, using items, etc.), and able to observe its reward (checking your score). The agent finds itself inside its an environment that changes the agent's state based on the action the agent takes, and based on the change in states and the action the agent took, it also produces a reward for the agent. The agent's goal is to maximize the total reward it can get. The transitions from one state to another given a certain action aren't always deterministic.

To formally describe this, the environment is a set of functions where $T$ describes the probability of transitioning from state $s$ to state $s'$ given that the action $a$ is taken. $R$ describes the reward for successfully transitioning from $s$ to state $s'$ given that the action $a$ is taken.

$$T(s, a, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

$$R(s, a, s') = reward$$

This all happens in episodes: at the beginning of each episode the agent starts from a variety of starting states. At each time step the agent chooses an action based on its current state, and the environment moves the agent's state. At specific "terminal states" or a max number of time steps, the episode will end, forcing the agent to restart.

Here are some terms you should know/search up:

- State-Value / Q-Value / Discount Factor: State value is supposed to represent the expected reward received from a given state. Q-Value is the expected reward received from taking a specific action at a specific state. Discount factor ensures that later rewards are not weighted as high, encouraging getting rewards faster.
- Value Iteration: iterative method for leaning the state values where you make the current state value reliant upon the state value of surrounding states. The Bellman equation for the relation between current and future states is as follows:

  $$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

  (Q learning is similar)
- Model-Based Reinforcement Learning: This is a type of reinforcement learning where your objective is to learning an accurate representation of the environment's $T$ and $R$ functions. Once these are known/estimated, it uses model to learn the best actions to choose. Discrete examples include: Value Iteration
- Model-Free Reinforcement Learning: Doesn't learn the model, but just learns a policy as in just figures out what actions to do at what states. Example is temporal difference learning with Q-learning.

# Smart Start Discrete

Reinforcement learning has a problem of exploitation vs. exploration. If you know certain areas have high reward, you would want to navigate to those areas learn how to be more efficient, but if you only search the known areas of high reward you lose the potential of finding unknown reward sources.

Smart Start poses this problem as what is called a **multi-armed bandit problem**. The problem is basically given $N$ slot machines, and for each machine it returns some reward from a unique but unchanging probability distribution. Given some history of pulling some machines and getting some reward from those machines, what is the best strategy for picking which slot machine to pull? There is a strategy called the Upper Confidence Bound Algorithm that performs well on this problem.

Smart Start says that for all states in the past, they will be modelled as a slot machine, and visiting that state is ”pulling” the machine. The average reward in this case would be the State-Value of that state. This chooses states with a good potential of reward, accounting for possible unknown rewards.

This concept creates a smart start framework that can augment any other reinforcement learning algorithm that produces estimates for state-values. The framework operates in episodes. Sometimes the base algorithm will be allowed to run and observe the environment for the whole episode, but some episodes will randomly have a smart start episode begin where the framework will:

- Pick a Smart Start State
- Navigate to the Smart Start State
- Give control back to the base algorithm for the rest of the episode

Picking the smart start state has already been described, but as for navigating to it, that itself is a reinforcement learning algorithm in itself and can be dealt with in a variety of different ways. In Keulen's project, he uses Value iteration. All details of Smart Start can be found in the original master thesis paper here.

# Deep Deterministic Policy Gradient (DDPG)

This continuous reinforcement learning algorithm relies on Neural Networks and uses what is called the actor-critic approach. When having one Neural Network generate Q-Values (like in Deep-Q learning), you have the problem of trying to find the action that maximizes the value outputted by your Neural Network when your action space is continuous. Actor-Critic approaches create a critic neural network that approximates the Q-value of each state, action pair, and also creates an actor neural network that takes in a state and returns the best action to take. The Q-value network learns similarly to Deep-Q learning, and the actor network is ”connected” to the critic network to allow for gradient ascent of the parameters of the actor network. (If this doesn't make sense look at the research paper linked to at the end of this section, also this short video was really helpful)

DDPG takes it a step further and stabalizes the learning process with insights from the Deep-Q Networks/Double Q Network papers which use two networks instead of one. DDPG uses two actor networks and two critic networks. The reason is because DDPG learns at each step of the environment, but the critic network is taught by minimizing the loss determined by the bellman update equation where the target (the $Q(s', a')$ on the right side) changes with each step as well. This makes for very unstable learning. Therefore by having a separate critic network that slowly updates to converge to the main critic network, we can have more stable learning.

$$Q^*(s,a) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$$

The network training requires that the states it is training on are independent of one another, but that isn't the case if they are all in one episode, so the algorithm has a replay buffer (one that stores a limited about of timesteps) and randomly draws samples from the replay buffer to simulate independent samples.

State Value estimation for this works as following: For a given state, feed it into the actor network to get the "best possible action" and then feed the state, action pair into the critic network to get a q-value. This provides a state value.

This project uses OpenAi's implementation as a base, with some slight custom modifications. OpenAi's implementation can be found in the python package `baselines`.

All details of the Deep Deterministic Policy Gradient algorithm can be found in the original paper here.

## Kernel Density Estimation

Kernel Density Estimation is a way of approximating the distribution of a random variable given a bunch of random outputs from that variable. Around each sample of points, you basically create some distribution around that point (commonly a Gaussian distribution is used). By averaging over all the sub-distributions or kernels of each point, you create an approximation for the variable's distribution like in figure 2. The same can be done for multi-dimensional variable distributions as shown in figure 3. The multi-dimensional case is necessary for this project, but luckily there was a `numpy` python function that implemented it for us.

Note that the KDE creates an estimation of the Probability Density Distribution (PDF). This is a function $PDF(s)$ where $s$ is some state ($n$-dimensional vector) such that for a $n$ dimensional region $R$, the probability that the next state is within $R$ is

$$\int \cdots \int_R PDF(s) du_1 \dots du_n$$

For Smart Start Continuous we use the multivariate KDE to estimate visitation count. For the "visitation" of a given state $s$ we define a region $R$ as a hyper-ellipsoid centered at $s$. The radii of the hyper-ellipsoid in each dimension is designated to be "one average step-size along the given dimension". Specifically for the radii in the $i$th dimension, we look at some previous episode, and average the absolute value of change in the $i$th dimension of the state at each time step. To get the visitation count at state $s$ we take $PDF(s)\texttt{volume}(R)(|D|)$. Where $|D|$ is the number of states previously visited (or number of states within the replay buffer/ memory)

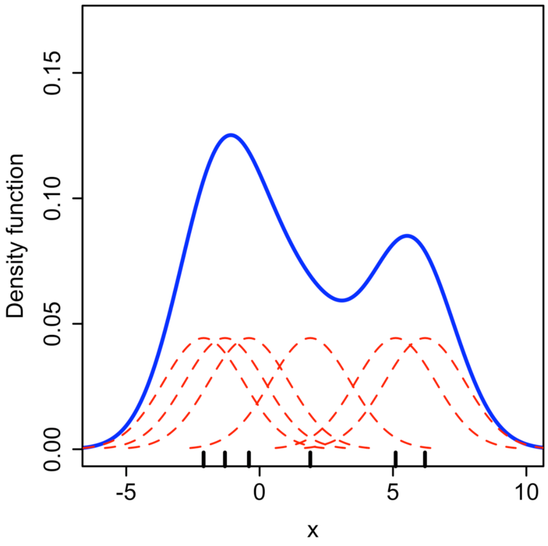

Figure 2: Univariate KDE with Gaussian Kernel

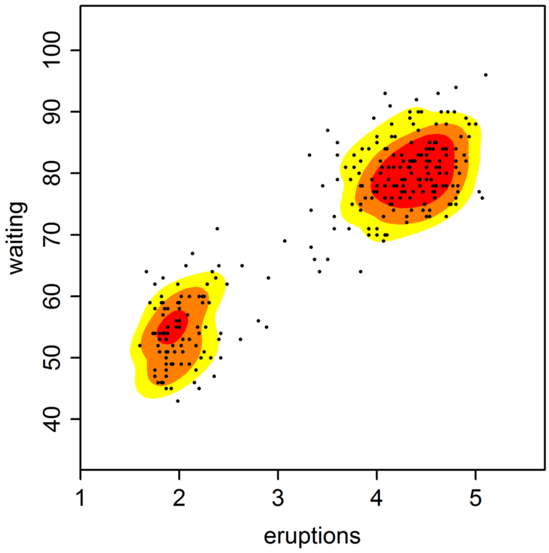

Figure 3: Multivariate KDE

# Neural Network Dynamics for Model-Based Learning

## 1 Research Paper Info

This project used a Neural Network with a particular setup that allowed it to take in a state and action pair and output the predicted new state. That is roughly how it work (glossing over a lot of details) and using this along with a model predictive controller, they were able to get a very sample efficient reinforcement learning algorithm. The paper itself delves into the fact that their model-based controller was unable to find absolutely optimal solutions as model-free approaches did, but using their model-based agent, they initialized a model-free agent with a very similar policy which ended up learning the optimal solution but still ended up being sample efficient in the long run.

## 2 Model Predictive Control

For Smart Start State Navigation, however, the optimal navigation path isn't so necessary, so the most important part is just getting to the end of the path, so the sample efficiency of this algorithm is perfect. The Model Predictive Controller which controls the agent with the Neural Network Dynamics approximater, works as following: for a given state $s$, it generates $N$ random sequences of actions. A horizon variable $h$ controls how long the sequences are. For all $N$ sequences of $h$ actions, it creates generates $N$ sequences of $h$ states based on the initial state $s$ and the Neural Network Dynamics approximater. Through some reward function, it calculates the reward of each action sequence and the chooses the action sequence of the best reward. Finally it executes only the **first** action of that sequence, then it repeats.

## 3 Reward Function

Given a single Smart Start State, it is difficult to navigate directly to in a straight line, so the agent will look in the replay buffer at the episode in which that smart start state was visited (all potential smart start states have been visited at least once). The Model Predictive controller will then take the path previously taken to get to the Smart Start State and try to do some path shortening on it. With a shortened path, the MPC create some waypoints on the path (currently each state on the path becomes a waypoint) and will reward the agent for travelling closer to the waypoints. Once the agent has gotten close enough to the current waypoint **or** has become closer to the next waypoint than the current waypoint, it will start tracking the next waypoint. The agent is also penalized for getting too far away from the current line segment (defined by current waypoint to next).

## 4 Leniency

For any given waypoint, if after $numStepsUntilWaypointGiveup$ steps it hasn't changed waypoints, it will automatically move on to the next one. Finally after it reaches the final waypoint, after a certain number of $finalSteps$ the agent will declare it has "reached the final state" even if it hasn't.

All details of Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning can be found in the original paper here. They have a webpost displaying videos of their work and results, and their code which was used in this project is available here.

# Distance Function

It is necessary to define a distance function since Euclidean distance isn't always the best measure. For this project the main distance function used is basically an "average step-size distance". The goal of this distance function made by `elliptical_euclidean_distance_function_generator` in `smartstart/utilities/numerical.py` is to have the average step-size be set as 1. It takes some episode previously travelled and will average all the absolute value of changes in each dimension. Then the average step size in the $i$th dimension will be a distance of one, and with all these averages in each dimension, it forms a hyper-ellipsoid where each point on the hyper-ellipsoid is a distance of one. Note that sometimes these averages can have one standard deviation added to it.

# Optimal Path Shortening

The Smart Start State Navigation takes a smart start state, and tries to navigate the exact path that was previously taken to reach that smart start state. Sometimes with a episode time-step limit of 1000 steps, the smart start path would be 900+ steps, leaving almost no time for the base algorithm to learn. Upon further inspection, most smart start paths walk around aimlessly repeating locations a lot. This means by recognizing when states are repeated, we can shorten the path that we walk.

To optimally take the shortcuts, we take all states on the path and compare their distances to all other states on the path. This if done with matrix operations will give you a $NxN$ matrix of distances where $N$ is the number of states on the path and the diagonal is all 0's. This shouldn't take too long since there are a limited number of steps. Now decide on a threshold of "close enough to shortcut" (in the project a distance function is used such that each distance of 1 is approximately an average step so the threshold used is just 1) and form pairs of indices of states on the path where all states in between the two states can be removed. This will create a bunch of intervals and not all of the intervals can be removed. This can be reduced to a weighted activities scheduling problem where the weight of each activity is the time lenght of the activity. This can be easily solved in $O(N \log N)$ time.

## References

[1] Fred G. Martin *Robotics Explorations: A Hands-On Introduction to Engineering.* New Jersey: Prentice Hall.

[2] Flueck, Alexander J. 2005. *ECE 100*[online]. Chicago: Illinois Institute of Technology, Electrical and Computer Engineering Department, 2005 [cited 30 August 2005]. Available from World Wide Web: (http://www.ece.iit.edu/ flueck/ece100).