TECHNICAL UNIVERSITY OF CRETE
School of Electrical and Computer Engineering

ADAPTATION OF ACTION SPACE FOR REINFORCEMENT LEARNING

Diploma Thesis

by Dimitrios Kontzedakis

Chania, October 2018

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΠΡΟΣΑΡΜΟΓΗ ΤΟΥ ΧΩΡΟΥ ΕΝΕΡΓΕΙΩΝ ΓΙΑ ΕΝΙΣΧΥΤΙΚΗ ΜΑΘΗΣΗ

Διπλωματική Εργασία

του Δημητρίου Κοντζεδάκη

Χανιά, Οκτώβριος 2018

Abstract

Reinforcement Learning is a Machine Learning technique, where a decision making algorithm, also known as autonomous agent, interacts with an (unknown) environment by making observations and taking actions, while it is receiving positive or negative rewards at each step based on its performance. During this process, the agent tries to learn an optimal decision making policy, namely which action selections at each state will help to maximize the expected total reward in the long term. This technique is ideal for optimal control problems, games and many other domains. Many RL architectures use a discrete set of actions to represent a continuous Cartesian action space and the agent is called to select one of these discrete actions at each time step. Usually, this discretization of a continuous action space reduces the ability of the agent in taking actions that perform best, since the agent is forced to choose among the discrete actions. There are two alternative solutions to this problem: either increase the density of discrete points, which affects the responsiveness of the agent, or adopt a discretization of variable resolution which adapts to the needs of the problem. In this thesis we present a method for creating discretizations able to adapt dynamically according to the use of the action space. The proposed adaptive discretization can match automatically a big variety of different patterns in a few adaptation steps, while maintaining a constant number of discrete points. We embed this adaptive discretization method into the action space of a particular Deep RL agent performing in specific environments that require precision. Our adaptive discretizations take advantage of the selective use the agent makes over the action space and adjusts the density of the discrete points in the space, giving increased number of discrete actions and thus higher resolution to regions where it is needed. As a result, the agent's precision and learning performance is increased, without significant increase in computational resources.

Περίληψη

Η Ενισχυτική Μάθηση είναι μια τεχνική Μηχανικής Μάθησης, όπου ένας αλγόριθμος λήψης αποφάσεων, γνωστός και ως αυτόνομος πράκτορας, αλληλεπιδρά με ένα (άγνωστο) περιβάλλον κάνοντας παρατηρήσεις και ενέργειες σε αυτό, ενώ ταυτόχρονα παίρνει θετική ή αρνητική επιβράβευση σε κάθε βήμα με βάση την απόδοσή του. Μέσα από αυτή τη διαδικασία, ο πράκτορας προσπαθεί να μάθει τη βέλτιστη πολιτική λήψης αποφάσεων, πιο συγκεκριμένα να βρει επιλογές ενεργειών σε κάθε κατάσταση που θα βοηθήσουν να μεγιστοποιηθεί η αναμενόμενη συνολική επιβράβευση μακροπρόθεσμα. Η τεχνική αυτή είναι ιδανική για προβλήματα βέλτιστου ελέγχου, για παιχνίδια και πολλά άλλα πεδία. Πολλές αρχιτεκτονικές πρακτόρων Ενισχυτικής Μάθησης χρησιμοποιούν ένα σύνολο διακριτών ενεργειών που αναπαριστούν έναν συνεχή Καρτεσιανό χώρο ενεργειών και ο πράκτορας καλείται να επιλέξει μία από αυτές τις διακριτές ενέργειες σε κάθε χρονικό βήμα. Συχνά, αυτή η διακριτοποίηση του συνεχή χώρου ενεργειών μειώνει την ικανότητα επιλογής ενεργειών που αποδίδουν καλύτερα, ενώ ο πράκτορας είναι αναγκασμένος να επιλέξει μόνο μεταξύ των διακριτών ενεργειών. Υπάρχουν δύο εναλλακτικές λύσεις σε αυτό το πρόβλημα: είτε να αυξηθεί η πυκνότητα των διακριτών σημείων, το οποίο θα επηρεάσει την ταχύτητα αντίδρασης του πράκτορα, είτε να υιοθετηθεί διακριτοποίηση με μεταβλητή ανάλυση προσαρμοσμένη στις ανάγκες του προβλήματος. Σε αυτήν την εργασία παρουσιάζουμε μια μέθοδο δημιουργίας διακριτοποιήσεων που έχουν τη δυνατότητα να προσαρμόζονται δυναμικά ανάλογα με τη χρήση του χώρου ενεργειών. Η προτεινόμενη μέθοδος προσαρμοσμένης διακριτοποίησης μπορεί να χειριστεί αυτόματα μια μεγάλη ποικιλία μοτίβων μέσα σε λίγα βήματα προσαρμογής, ενώ διατηρεί τον αριθμό των διακριτών σημείων σταθερό. Ενσωματώσαμε αυτή τη μέθοδο στον χώρο ενεργειών ενός συγκεκριμένου πράκτορα Βαθιάς Ενισχυτικής Μάθησης που ενεργεί σε περιβάλλοντα που χρήζουν αυξημένης ακρίβειας. Οι προσαρμοσμένες διακριτοποιήσεις μπορούν να εκμεταλλευτούν την επιλεκτική χρήση που κάνει ο πράκτορας στο χώρο ενεργειών και να αυξομειώσουν την πυκνότητα των διακριτών σημείων ανά περιοχή, δίνοντας αυξημένο αριθμό ενεργειών και συνεπώς υψηλότερη ανάλυση σε περιοχές όπου υπάρχει ανάγκη. Αυτό είχε σαν αποτέλεσμα να αυξηθεί η ακρίβεια και τελικά και η απόδοση του πράκτορα, χωρίς σημαντική αύξηση στις απαιτήσεις υπολογιστικών πόρων.

Contents

1. Introduction **7**
1.1 Thesis Introduction 7
1.2 Thesis Contribution 7
1.3 Thesis Overview 7
2. Background **9**
2.1 Machine Learning 9
2.1.1 Reinforcement Learning 9
2.1.2 Neural Networks 10
2.2 Discretization 11
2.2.1 Annotations and symbols 11
2.2.2 Uniform PDF 13
2.2.3 Non-uniform PDF 15
3. Problem description **17**
3.1 Introducing the agent 17
3.2 Related work 19
4. Proposed Approach **20**
4.1 Adaptive discretization 21
4.1.1 Structure 21
4.1.2 Evaluation 24
4.1.3 Modification 25
4.1.4 Adaptation 26
4.1.5 Error function 28
4.1.6 Performance 29
4.1.7 Efficiency 30
4.2 Adaptive action space 31
4.2.1 Adaptability on training 32
4.2.2 Square error function 32
4.2.3 Normalization of action space 33
5. Results **35**
5.1 Adaptive Discretization 35
5.1.1 Adaptation 35
5.1.2 Efficiency 38
5.1.3 Adaptability 39
5.2 Adaptive action space in RL 42
5.2.1 Inverted Pendulum 43
5.2.2 Cart-Pole 47
6. Conclusion **49**
6.1 Future work 49
References **50**

1. Introduction

1.1 Thesis Introduction

Reinforcement Learning (RL) architectures (Sutton and Barto, 1998) are used to solve a big variety of problems, such as strategic decision making, optimal control, games and many others. RL agents repeatedly compute their next actions based on their previous observations until they will get to their target, while they are continuously receiving rewards at each step. By processing the information collected through the interaction with their environment, they have the ability to adopt complex action decision policies, which maximize the total expected reward in the long run. Their actions are usually multi-dimensional continuous control signals within a specified range. Some RL agents are designed to work with discrete control signals, also known as discrete action spaces. In these cases, the agent is forced to select one of the available discrete actions at each time step. This setting restricts their ability to be flexible and perform smoothly or, in other words, to make use of the action space in any way they want. As a result, agents that use discrete action spaces are forced to use discretizations with increasingly higher resolution, which reduces their performance, making them incompetent to address problems that require real-time responses.

1.2 Thesis Contribution

Most times, agents are using extensively some parts of the action space, while others may not get used at all. By taking advantage of this behavior, it is possible to create discretizations that use lower resolution on the parts that are used less, to increase the resolution on the other parts that are used more. This thesis presents a method that creates adaptive discretizations. These discretizations are adapting according to the frequency of the use of each part of the space, while the number of discrete points remain constant. If a part is used more, the resolution on this part will increase and vice versa. The goal is to embed this method to RL agents with discrete action space, to provide them with the ability to be smoother and faster at the same time. This method gives them also the ability to be competitive in high-dimensional environments where a uniform discretization that would cover the whole space would be prohibitively large. This method was implemented in Python 3, with extensive use of many well-known libraries and tools like Tensorflow and openai-gym environments. All the plots in this thesis were made with using the **`maplotlib`** library.

1.3 Thesis Overview

Chapter 2 contains a basic theoretical background on the Machine Learning algorithms that are used later and a brief mathematical background on discretizations. Chapter 3 contains the statement of the problem we study and an introduction to the Deep Reinforcement Learning agent we used. In Chapter 4, a detailed description of the structure and functionality of the proposed Adaptive Discretization method is given, as well as its embedding in the RL agent. Finally, the results and the conclusion of the proposed method can be found in Chapter 5.

2. Background

The complexity of the problems assigned to computers is increasing with enormous rate. Due to this complexity, the ability to solve problems with hard-coded programs is very limited. Thus, Artificial Intelligence (AI) was created (Russell and Norvig, 1995), a field of computer science that attempted to give to computers the ability to solve complex problems and find strategies that bring the best results. Reverse engineering of the behavior of human thinking and many other intelligent behaviors found in nature created autonomous agents, able to solve problems without any knowledge of the rules and limitations that govern them.

2.1 Machine Learning

Machine Learning (ML) is a subfield of AI that focuses on making computers discover the best strategies by using learning as the main tool. Learning is a procedure of understanding the goals of a given task within a certain domain and the ways to reach these goals, by collecting and generalizing over experience collected from the target domain. Every ML algorithm has an architecture able to learn things using a training procedure. This training procedure is where the system is trying to generalize over the collected experience and reach as close as possible to an optimal solution in rational time. ML splits into three main categories that are differentiated by the training techniques used:

- Supervised learning: A procedure of repeatedly feeding the system with inputs and the desirable output trying to discover the hidden mapping function. This learned function will be used afterwards to output results for any given input, even unseen ones.
- Reinforcement learning: A technique used frequently in environment-interaction problems, where an agent must adapt its actions in order to reach certain goals. Through behaviorism, the agent is trying to find the best action in each state it encounters and finally discover the best action strategy to follow in the environment.
- Unsupervised learning: Like supervised learning, the agent is given a set of inputs and tries to discover patterns behind them, but this time without any output labels or any kind of help. The algorithm by itself has to generalize over the input and understand the hidden structures.

2.1.1 Reinforcement Learning

In reinforcement learning (RL), the environment is modeled as an Markov Decision Process (MDP) (Bellman, 1957), where a decision maker, known as the agent, interacts with a stochastic environment in sequential steps and collects rewards at each step depending on its ability to fulfill the goal. The agent can be at any given time t in any state s of the state space $S \in \mathbb{R}^m$, and has to select one action a from the action space $A \in \mathbb{R}^n$ to apply to the environment. Each state or action is a n-dimensional and m-dimensional vector respectively inside a continuous or discrete space. Applying the selected action the agent, stochastically transits to the next state s_{t+1} with probability $P(s_{t+1}|s_t, a_t)$. After each step, it is rewarded by a value drawn from the reward function $r(s_t, a_t)$. The rewards are discounted by a factor $\gamma \in [0, 1]$ over time, so the total reward shrinks in the long run and becomes a finite number even for an infinite horizon of steps. After a full episode *E* of interaction that lasted *T* steps, the agent collects a total reward R_E as follows:

$$R_E = \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$$

Using samples of state, action, next state and reward (s_t, a_t, s_{t+1}, r) at each step, and no other information, the agent has to adopt a behavior, called policy, $\pi(s)=a$, under which it will select its action *a* for any state *s*. Most RL algorithms make use of the recursive state-action value function *Q(s, a)*, which can be computed using Bellman's equation, to calculate the expected reward and thus evaluate the current policy π.

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \gamma \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t)\, Q^\pi(s_{t+1}, \pi(s_{t+1}))$$

The goal is to learn an optimal policy $\pi(s)$ that maximizes the expected total reward over all possible episodes. Because the transition model *P* is part of the environment, the reward system is the only thing that forces the agent to improve itself. Thus, setting the goals of the agent is done by customizing the reward function to reward the desirable behaviors and punish the unwanted ones.

In real world problems, the agent can be the brain of a robot, with the measurements of the sensors as input states and all the possible combinations of control signals to the actuators as actions. By setting the appropriate reward function, the agent can learn to perform any task in a way that delivers the maximum expected total reward, such as controlling a vehicle, playing a game or performing competitively in any other decision making domain.

2.1.2 Neural Networks

Neural networks (NN) are supervised learning models inspired by the biological neurons found in the brains of all animals, including human. They process and transmit signals in parallel and form complex circuits that are able to implement all kinds of intelligent behavior.

A NN consists of a set of neurons connected in a network topology formation. Neurons have input and output connections. Each input connection of a neuron holds a weight that acts as a multiplicative factor to the carried incoming signal. The neuron itself sums up all these weighted input signals and transfers the result to its output connections. If x is the input signal vector and $\boldsymbol{W}$ the weight matrix, then the output vector is $y=\boldsymbol{W}\times x+b$ where b is the bias value preventing the neuron to give always zero output in case of zero weight. Finally, a *sigmoid* function is applied to y to constrain the output value into the desirable range. By adjusting the $\boldsymbol{W}$ and b parameters, it is able to modify the output to a great variety of functions, while stacking many neurons in a layer with the same inputs, the output can implement more complex functions. Also, stacking multiple layers in a row, creating a multi-layer NN, or Deep Neural Network (DNN), the functionality is extended to even more complex function approximations.

A procedure for training is essential for the network to adjust its weights and biases in order to produce the desired outputs. This process of adjustment is called backpropagation. A comparison between the current output of the network and the desirable one, gives enough information to trace back into the network to refine any weight and bias, so that it will produce the right result. Repeating this procedure for many inputs customizes all the parameters as needed. So, training requires a large number of labeled samples, that is a set of inputs with their corresponding outputs, to feed into the network repeatedly followed by back propagations. The samples must be carefully selected, so that they match most of the possible inputs. NN's and DNN's ability to generalize over the input samples make them a powerful tool for classifying inputs or for approximating functions.

2.2 Discretization

Discretization is the procedure of associating a continuous set to a discrete one. It splits the continuous set into smaller subsets and assigns a discrete value to each one of them to represent them. These sets can be of any type, but most of the times they are continuous spaces that are approximated by discrete points. If the space is restricted, then the number of the discrete points is always finite which makes the discretization function a mechanism that associates elements of an infinite set with a finite one. This is really useful in computer science, where information is always discrete. In order for the representation to be accurate, points of the continuous set have to be "close" to points of the discrete set. It is possible for discretizations of the same continuous space to provide different accuracy. This brings out the term of efficiency of a discrete set.

2.2.1 Annotations and symbols

For this thesis, it is assumed that the continuous and the discrete spaces that are used are multi-dimensional with specified limited ranges along each axis. Discrete points sit inside the continuous space at specified locations. Continuous points are assigned to their closest neighbor discrete point, meaning that for each discrete point, there is a region around it, where all continuous points are nearest to that point. Where one regions ends, another one starts, so they all collectively cover to the complete continuous space.

Before digging into the details, it is necessary to introduce some terminology and a list of symbols that will be used.

- **Symbols**
  - ➝ n : number of dimensions, $\{n \in \mathbb{Z}^+\}$
  - ➝ R : continuous space
  - ➝ a, b: the lower and the upper limit of R, $\{a,b \in \mathbb{R}^n\}$
  - ➝ $x = [x_1, x_2, \ldots, x_j, \ldots, x_{n-1}]$: a point inside the space, $\{x \in \mathbb{R}^n\}$ and $\{j \in \mathbb{Z}: 0 \leq j \leq n-1\}$

→ k : the total number of discrete points, $k \in \mathbb{Z}^+$

→ $x_i = [x_{i,1}, x_{i,2}, ..., x_{i,j}, ..., x_{i,n-1}]$: i-th discrete point of the discretization,

$\{i \in \mathbb{Z}: 0 \leq i \leq k-1\}$ *and* $\{j \in \mathbb{Z}: 0 \leq j \leq n-1\}$

→ R_i : region around the discrete point x_i, whose points x are assigned to x_i

→ $a_{i,j}, b_{i,j}$: the lower and the upper limit of R_i along axis j, $\{a_{i,j}, b_{i,j} \in \mathbb{R}^n : a_{i,j} < b_{i,j}\}$ and $\{j \in \mathbb{Z}: 0 \leq j \leq n-1\}$

- **Formulas**

→ $V_i = \prod_{j=0}^{n}(b_{i,j} - a_{i,j})$: The volume of R_i. In case of $n=1$, V_i is just the length of this range, and in case of $n=2$, V_i is the surface area of the corresponding plane.

→ $d(x_i, x) = \sqrt{\sum_{j=0}^{n-1}(x_{i,j} - x_j)^2}$: The well known Euclidean distance of a discrete point to a point in the continuous space.

Also, a discretization is rated by its accuracy and its error, which is associated with the distance of a point x to its neighbor x_i. Distances and errors are correlated through an Error Function (EF), *error = EF(distance)*. For the purpose of this introduction and in order to make it simpler, we will eliminate the EF and assign directly the distance to the error, (*error* = *distance*). Error functions will be covered in detail later. The average distance, or Mean Error (*ME*), will be used as evaluation metric for each discrete point x_i or for the whole set of discrete points. $ME(x_i)$ will be the annotation for the error produced by the point x_i and ME will be the total average error produced by the whole set of discrete points of the current discretization.

$$ME(x_i) = \int_{\forall x \in R_i} P(x)\, d(x_i, x)\, dx$$

$$= \int_{a_{i,0}}^{b_{i,0}} \int_{a_{i,1}}^{b_{i,1}} \cdots \int_{a_{i,n-1}}^{b_{i,n-1}} P(x) \sqrt{(x_{i,0} - x_0)^2 + (x_{i,1} - x_1)^2 + \ \ldots\ + (x_{i,n-1} - x_{n-1})^2} dx_1 dx_2 \ldots dx_{n-1}$$

P(x) is the probability of the appearance of x. Total ME is the sum of all the errors produced by the set of the discrete points:

$$ME = \sum_{i=0}^{k-1} ME(x_i)$$

2.2.2 Uniform PDF

Most of the times, discretization of a space is done by simply setting the k discrete points at a constant distance to each other, creating a uniform grid. Behind this discretization, there is a hidden assumption, that each x in that space is equally possible to appear. In other words, the Probability Density Function (PDF) of x is uniform, with value:

$$P(x) = \frac{1}{V_R} = p$$

For a specific x_i, $ME(x_i)$ becomes:

$$ME(x_i) = \int_{x \in R_i} P(x)\, d(x_i, x) = p \int_{x \in R_i} d(x_i, x)$$

For example, let's set $n=1$:

$$p = \frac{1}{V_R} = \frac{1}{b-a}$$

and

$$ME(x_i) = p \int_{x \in R_i} d(x_i, x) = \frac{1}{b-a} \int_{a_i}^{b_i} \sqrt{(x_i - x)^2} dx = \frac{x_i^2 - (a_i + b_i)x_i + \frac{a_i^2 + b_i^2}{2}}{b-a}$$

The minimum of this function is located at the middle of $[a_i, b_i]$

$$argmin(ME(x_i)) = \frac{a_i + b_i}{2}$$

with value

$$ME(\frac{a_i + b_i}{2}) = \frac{(b_i - a_i)^2}{4(b-a)}$$

It is clear that ME depends only on the length of the range. Applying this to the total ME formula gives:

$$ME = \sum_{i=0}^{k-1} ME(x_i) = \sum_{i=0}^{k-1} \frac{(b_i - a_i)^2}{4(b-a)} = \frac{1}{4(b-a)} \sum_{i=0}^{k-1} (b_i - a_i)^2$$

The placement of the discrete points inside the space affects the lengths of the ranges and finally the total *ME*. The minimum of ME function for the set of x_i's in this case can be found through the Cauchy–Schwarz inequality.

$$(\sum_{i=0}^{k-1} x_i y_i)^2 \leq \sum_{i=0}^{k-1} x_i^{\ 2} * \sum_{i=0}^{k-1} y_i^{\ 2}$$

Setting $x_i = b_i - a_i$ and $y_i = 1$:

$$(\sum_{i=0}^{k-1} (b_i - a_i))^2 \leq \sum_{i=0}^{k-1} (b_i - a_i)^{\ 2} * \sum_{i=0}^{k-1} 1^{\ 2}$$

On the left side is the sum of all the sub-regions of the space that add up to *b-a*. Also the sum of *k* ones is *k*.

$$(b-a)^2 \le k * \sum_{i=0}^{k-1} (b_i - a_i)^2 \quad \Leftrightarrow$$

$$\frac{(b-a)^2}{k} \le \sum_{i=0}^{k-1} (b_i - a_i)^2$$

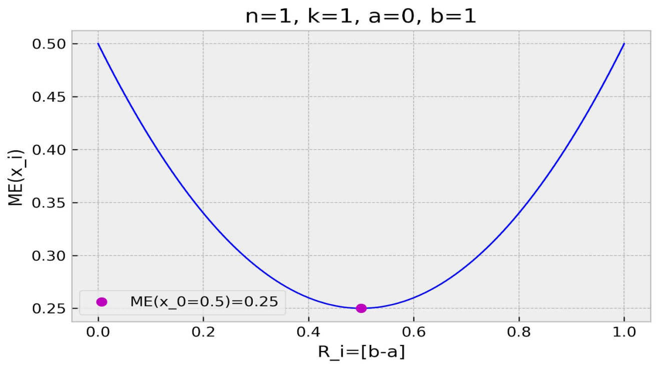
The minimum is at the point where the equality is satisfied. Then apply to the total ME to calculate the minimum value:

$$ME = \frac{\sum_{i=0}^{k-1} (b_i - a_i)^2}{4(b-a)} = \frac{(b-a)^2}{4k(b-a)} = \frac{b-a}{4k}$$

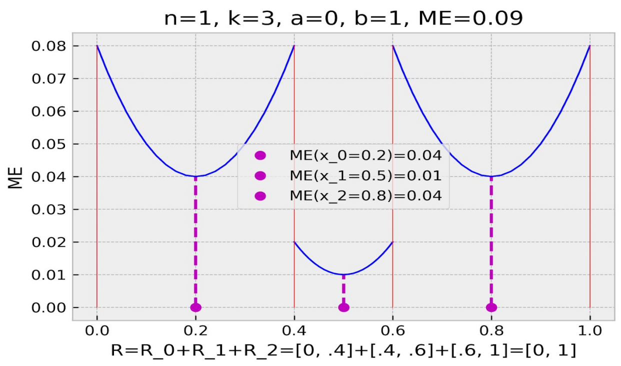
This proves that the optimal discretization for a uniform PDF is the discretization where the ranges are the same for each point and where the points are exactly at the middle of their ranges. The same concept is true for any uniform PDF with *n≥1*, but with slightly different values. That's why the uniform grid is used most of the times for discretizing a space.

Example:

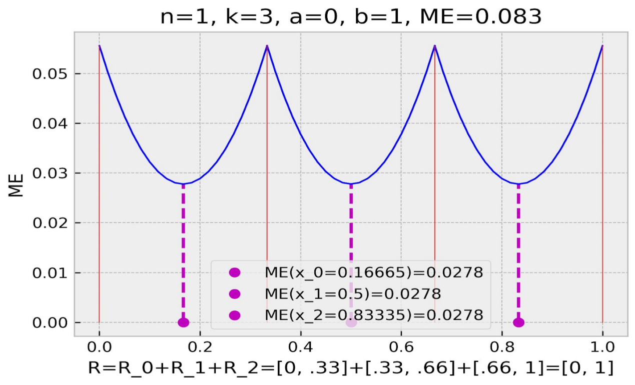
Assume n=1, *a=0, b=1, k=1*



Making *k*=3 and arranging the points arbitrarily in locations $x_0=0.2$, $x_1=0.5$, $x_2=0.8$ with $a_0=0$, $b_0=0.4$, $a_1=0.4$, $b_1=0.6$, $a_2=0.6$, $b_2=1$ gives:



Spreading uniformly the points, in locations x_0*=0.16665,* x_1*=0.5,* x_2*=0.83335* with a_0*=0,* b_0*=0.33,* a_1*=0.33,* b_1*=0.66,* a_2*=0.66,* b_2*=1* gives the optimal discretization:



2.2.3 Non-uniform PDF

Uniform PDF of *x* is one possible PDF among infinite others, so chances are that most of the time PDF won't be a uniform one. In case of a non-uniform PDF, the above principles no longer apply. To achieve optimal discretization, discrete points have to be more concentrated near the regions, where the PDF is high. In other words, the resolution of points has to adapt to the PDF. But, solving the equations with the new PDF is complex, especially for more than one dimensions. Also, most of the times it is not known, and/or is not even stable, as it can change slowly over time. In those cases, a uniform discretization is just a safe solution, with sub-optimal results. One possible solution is an adaptation procedure of repeatedly estimating the PDF, evaluating the current set of discrete points, and adjusting their positions in such a way that decreases the total ME. Assuming that each of these steps is done correctly, this procedure will converge to a solution close to the optimal.

Estimation of the PDF

This step is the easiest, as the only thing it requires is a number of samples big enough to make an good estimation. Of course, this number has to be the smallest possible, so in case of a non-stable PDF, the buffered samples would match the current shape of this PDF.

Evaluating the current discretization

The total *ME* can be used as evaluation metric, as it describes the error that has to be minimized. If *N* is the total number of samples, and x_{Ri} the sampled continuous points inside R_i, $ME(x_i)$ becomes:

$$ME(x_i) = \sum_{\forall x \in x_{R_i}} \frac{d(x_i,x)}{N}$$

The constant factor $\frac{1}{N}$ can stay outside the sum, and also outside of the sum for the total *ME.*

$$ME = \frac{1}{N} \sum_{i=0}^{k-1} \Big(\sum_{\forall x \in x_{R_i}} d(x_i, x) \Big)$$

The above formula of *ME* can be read as the average sum of the distance between each sampled continuous point x and its nearest discrete point x_i.

Adaptation of discrete points

The last and most important step is to use the information from the two previous steps to move the discrete points, such that the total *ME* will decreased. This part is the most difficult, because there is no straightforward way to be done. Changing the location of a x_i, changes its corresponding a_i, b_i as new continuous points will be closer to this new location. Those new edges produce another *ME* for that x_i. This procedure for each point can recursively solve the equations, but its complexity rises very quickly, which makes it untrackable.

3. Problem Description

Environments in RL usually have continuous action spaces that represent control signals, such as forces, electric pulses, etc. A great variety of RL architectures use discrete action spaces with a fixed number of possible actions. Often, this number is limited by the ability of the agent to process them at every step and maintain a real time response. Most of the times, if the action space is bounded, agents use a grid of actions evenly spread over the whole space. This makes any part of it accessible with a constant resolution. The problem with this approach is that the resolution is constrained to a constant "actions per unit of space" ratio. If we try to increase this ratio to achieve better precision and sensitivity, we have to increase the number of discrete points. This may work well in single dimensional spaces, despite the fact that this ratio is increased in the whole space, which is suboptimal. In case of a space with higher dimensionality, the increase in the number of points is exponential and becomes intractable very quickly. So, in domains that require both real-time response and sensitive controls, agents with discrete action space are doomed.

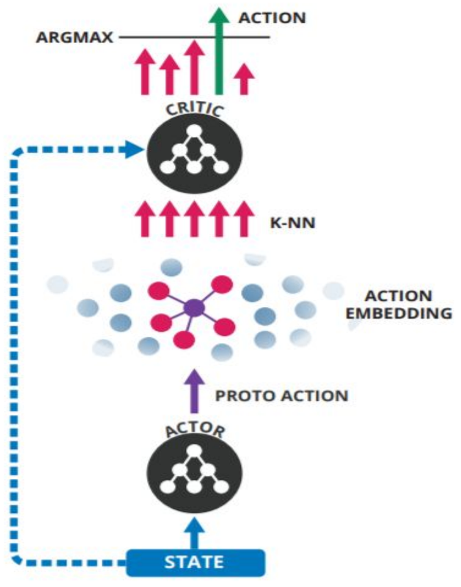
A technique to increase the resolution locally, without exceeding the fixed number of discrete actions and without making parts of the actions space unavailable, is what would solve this problem. In almost all the classic-control domains, the agent has to use mostly a few specified regions of the action space. For example, balancing an inverted pendulum by using force on its bottom part, requires strong forces only at the beginning to bring it in an upright position, but after that only slight corrections have to be made for the rest of the episode in order to keep it that way. In terms of action spaces, this results to a frequent use of actions near zero, and to a rare use of all the other actions. This is the case in most control problems, where most of the times few of the actions are used consecutively, while other may never get used at all. By taking advantage of this situation it is possible to gain an increased resolution, while maintaining the number of actions.

3.1 Introducing the agent

The goal of this thesis is to help agents with discrete action space to handle tasks that require gentle controls. Of course, this ability doesn't come only from the discretization, but also from the potential of the agent itself. To do so, it requires plenty of skills that depend on its architecture. Skills, like the ability to generalize over previously unseen actions and dealing with a great number of actions, are essential.

A very suitable agent for this job, is the Wolpertinger architecture (Gabriel Dulac-Arnold, Richard Evans et al., 2016) presented by the Google DeepMind team in the paper *Deep Reinforcement Learning in Large Discrete Action Spaces*. As shown in the figure below, this agent uses the actor-critic (Sutton and Barto, 1998) framework with multi-layer NNs to implement both actor and critic that are trained according to the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2015). The Fast Library for Approximate Nearest

Neighbors (FLANN) (Muja and Lowe, 2014) is also used to search in logarithmic time among the actions for the nearest neighbors.



Wolpertinger architecture.
[from Gabriel Dulac-Arnold, Richard Evans et al., 2016]

Briefly, the current state of the environment is feed into the Actor NN, where a continuous action, called proto-action, is produced. This proto-action is used to search its K nearest discrete actions, where $1 \leq K \leq$ total actions. Those K discrete actions are then evaluated from the Critic NN, which assigns a Q-value to each one of them. Finally, the action with the highest rate is selected to be applied. After the agent applies the action to the environment, it observes the next state and the reward it got. Each (state, action, reward, next state) sample is stored into a replay buffer. At each step, a batch of samples is drained from the buffer to be used for training the Actor and Critic NNs. Critic is trained directly from the samples, while for the Actor, a Q-Learning evaluation of the current policy must be done in the beginning in order to apply Policy Gradient to change the policy. More information about the exact procedure can be found in the corresponding papers.

The creators of this agent, break this procedure into two stages: the generation of the action and its refinement. Generation is the stage where K discrete actions are created from an input state and refinement is the stage where a single discrete action is selected among those K. It is like the Actor recommends some actions and the Critic makes the final decision. By changing the value of K, it is able to choose which one will dominate the final selection. The two extreme cases are where K equals to a single action and where K equals to the total number of discrete actions. For K=1, the Actor dominates, because it recommends a single action and leaves the Critic with no alternative actions to choose from. For K=total actions, the Actor is forced to recommend all the possible actions and the Critic is the one that will make the actual choice.

A recommended value for the K, depending the environment, is around 10% of the total number of actions, as the authors suggest. With this value, it is able to achieve a performance close enough to the performance of 100%, but with ten times less computational load. Also, as the experiments during the progress of this thesis revealed, for most domains used here, ratios bigger than that were confusing the agent. This happens, because this agent is actually an extension of another agent introduced in the *Deep Deterministic Policy Gradient* paper. All its structure is based on the DDPG agent, except that action selection and refinement process. For the DDPG agent, the Critic's role is only to help the Actor get trained by helping on the policy evaluation and policy gradient step. As a result, the Critic is not actually made for taking actions, and when it is called to do so, its results are not optimal. We will see more about that in the final results.

What makes this agent perfect for this kind of problems is that it is generalizing over these environments. A classic control environment is ruled by deterministic laws that are described by physics and mathematical equations. Actor and Critic NNs are able to learn those formulas, if trained well. This actually means that they understand the environment and are able to predict results for states and actions that were never seen before. Of course, states and actions have to be the control variables of theses formulas with enough information to recreate the missing parts. For example, if the representation of the action space is good, actions near the proto action will be similar, and have similar effects, if applied. Thus, the K discrete actions will be representative of the proto action, so the Critic can help to find the best one. Discretization then, will affect only the resolution of the space and the sensitivity of the agent, but not its behavior.

3.2 Related work

Although there is a lot of work in the literature on variable-resolution discretizations of state spaces, there is not much on agents with discrete actions spaces that were using anything else than a uniform discrete action space. Binary Action Search (Pazis and Lagoudakis, 2009) is a method that is able to search entire continuous action spaces with a binary search procedure, similar to the expansions of our proposed method. In follow-up paper, the same authors show how this binary search procedure can be extended to multi-dimensional action spaces (Pazis and Lagoudakis, 2011).

There are also some techniques that use continuous spaces with dynamic discrete resolution. Some of them were also used as an inspiration for the proposed method of this thesis. Adaptive Mesh Refinement (Berger and Colella, 1989), or AMR, is one of them, a method of adapting the resolution on a grid in order to improve the accuracy of a solution within certain sensitivity. Also, quadtrees and octrees, acted as inspiring architectures for the placement of the discrete points on a multidimensional spaces.

4. Proposed Approach

Actor and Critic are actually function approximators that generalize over continuous spaces, which makes the discretization of the action space detached from the agent's action selection and it only affects the quantity of available actions on each region of that space. In the training process, the agent tries to explore the environment and discover how actions affect its observations and rewards. This results in a wide exploratory use of the action space. After gaining some experience, the agent starts understanding the physics behind the environment and what the goals are. Thereafter, the action space is used more selectively, as the agent tries to optimize its policy. When it is finally trained, it only uses the exact required actions, which most of the time are a small subset of the whole discrete action set.

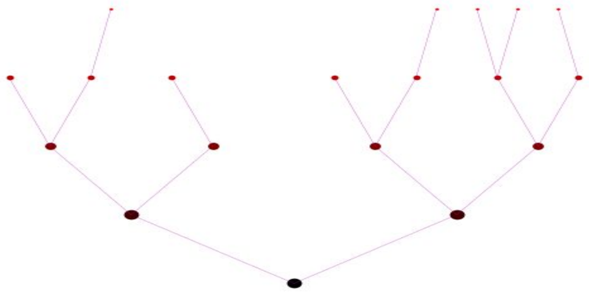
The goal of this thesis is to find a way to take advantage of this fact in order to provide a more flexible action space that will adapt to the needs of the agent. This means that is has to have increased resolution of available actions on the regions that the agent is more interested in, and decreased resolution on regions that the agent is less interested in, so that the total number of actions will remain constant to a number set by the designer. However, the agent must always have the ability to choose all kinds of actions, which means that there must always be enough coverage of discrete actions in the whole action space. Also, it is very important to make all this changes without disturbing the real-time action response of the agent. Changes also have to be made with a small computational cost. This is almost impossible, because changes in a particular area trigger changes over the whole action space, which means that all actions have to be processed during this process. Another option is to make heavier changes at times that agent is not busy and with a very small frequency. Also, another thing that we have to consider is that RL agents are model-free architectures and are made to work in any environment. This means we have no prior information about the final "shape" of the action space for each case. So, the action space has to be uniform at the beginning, and adapt over time through a fully automated way. Last, but not least, this architecture has to work for action spaces with any number of dimensions and any number of discrete points.

The available input data for this method will be:

- ➢ The number of dimensions, n.
- ➢ The limits of the action space, $\{a,b \in \mathbb{R}^n\}$.
- ➢ The total number of discrete points, k.
- ➢ The continuous points produced by the Actor, $\{x \in \mathbb{R}^n\}$.

4.1 Adaptive discretization

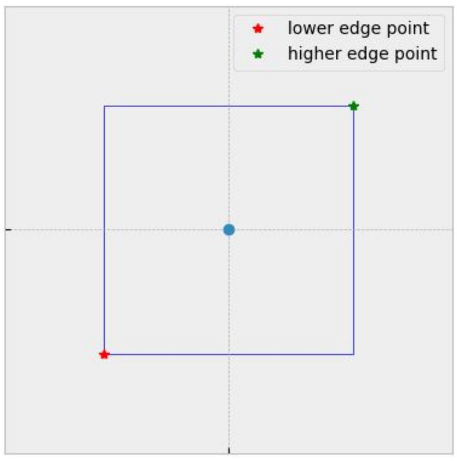
As mentioned before, creating a discretization that fits perfectly on a PDF is a procedure that can be done by repeatedly improving a current one. This forces a pass through all the current points in order to adjust their positions. The presented method works a bit differently, as it doesn't actually moves the points, rather it chooses between fixed sets and combination of points. It is inspired by tree structures that explore the action space by growing branches across all dimensions. Regions where branches are taller have higher resolution and vice versa. An example of such a tree in one dimension is shown below.



Note: Before digging into the details of the functionality and the structure of this methods, we will make an assumption that will be explained later. Each axis of the multidimensional space will have a range of [0,1]. So, finally the space will be an n-dimensional unit cube. We will explain later why and how this is happening. For now, let's just assume this is always true.

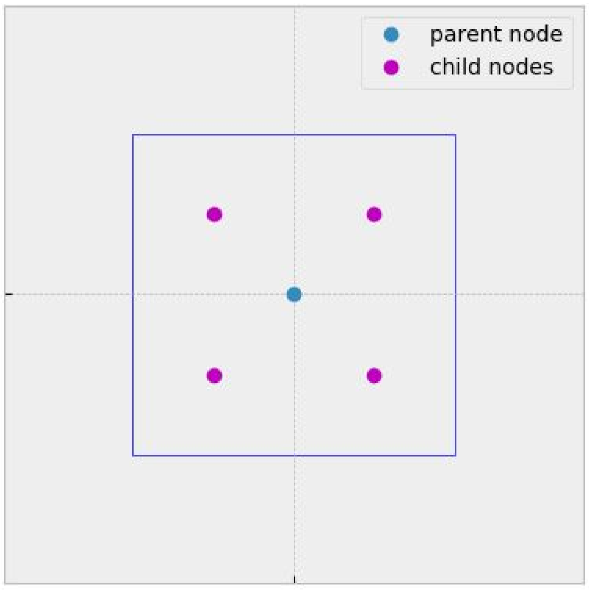
4.1.1 Structure

In detail, the base structure of this architecture is the n-dimensional tree, or n-tree. Each node holds a discrete action-point and the two other points that define the edges of the space that is associated with that point, which we will call as the action-point's region. These lower and higher edge points are the a and b we talked about on the discretization background. The inside space is an n-dimensional cube and the discrete point is located right on the middle of this cube. If we assume that this point sits at the center of the axis system, it will subdivide its cube into 2^n smaller equal cubes.



2-dimensional point in the middle of its associated space

These cubes will be the regions of each one of the child nodes in case of expansion. So, when a node expands, 2^n child nodes will be created at the middle of each smaller cube.



Expansion in 2 dimensions

Child nodes share equally their parent's region, which means that their region is a subset of their parent's region, and these regions together will cover exactly their parent's region. Also, it is important that no-overlapping occurs between nodes at the same level. Because no node will end up with the same low and high edge point, no discrete point will end up being the same with any other from the whole tree. This mean that any continuous point of the space belongs to exactly one region per level. If k is the total number nodes of a fully-grown tree, each continuous point belongs to *log(k)* nodes, one for each level. One of these nodes will contain the nearest discrete point to that continuous one, and if we want to get closer to it, we have to expand the node in the highest level among those *log*(*k*) nodes. Of course, root node's region is the whole actions space, which in this case is a n-dimensional unit cube, and is located on [*0.5, 0.5, ..., 0.5*]. Initially, the tree is fully grown to a certain level, so its size will approach as close as it can to the maximum number of discrete points user set. This produces an almost uniform discretization consisted of multiple layers of nodes. The size ***s*** of a fully grown ***n***-dimensional tree at a certain height ***h*** is given by the formula:

$$s = \sum_{l=0}^{h} 2^{n*l}$$

If, for example, the user sets the maximum number of discrete points to 1000 in a 3-dimensional space, the initial tree will grow up to 3 levels with size $s=2^{3*0}+2^{3*1}+2^{3*2}+2^{3*3}=1+8+64+512=585$ discrete points. Changes can be made to the tree, by expanding or cutting nodes, according to the evaluation of each node. Expansion will increase the resolution of discrete points in the area of the expanded node, and cuts will remove leaf nodes to save space for expansions. After all the changes, a new set of points is returned, which is actually the new discretization, or the new x_i's.

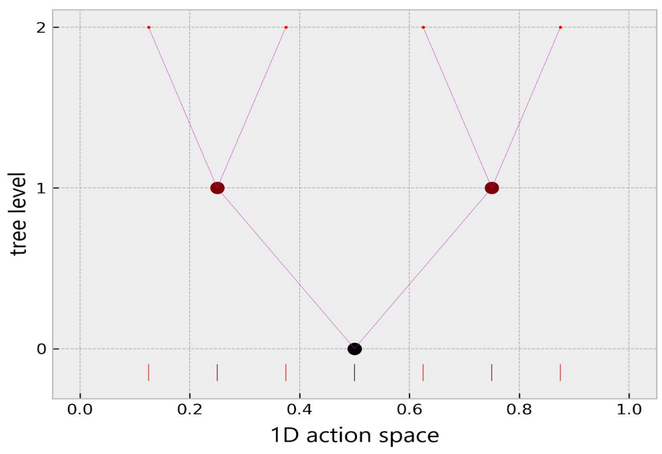
To make it more clear, let's take an example with *n=1*. The root, the node in level zero, will have the point [*0.5*] in the middle of the associated range [*0, 1*] (1-dimensional unit cube). Its $2^n=2$ children, or branches, will split this area in 2 equal 1-dimensional cubes and the tree will end up with the following structure:

- Root node at level 0 with range [*0, 1*] and point [*0.5*]
- First child node at level 1 with range [*0, 0.5*] and point [*0.25*]
- Second child node at level 1 with range [*0.5, 1*] and point [*0.75*]

Because each point is unique, nodes can be referred to with the location of their point from now on (node[0.5], for example).

Creating one more layer to this tree will add four more nodes:

- Node with range [*0, 0.25*], point [*0.125*] and parent node[0.25]
- Node with range [*0.25, 0.5*], point [*0.375*] and parent node[0.25]
- Node with range [*0.5, 0.75*], point [*0.625*] and parent node[0.75]
- Node with range [*0.75, 1*], point [*0.875*] and parent node[0.75]



Example 1D-tree of height=2 and size=7.

4.1.2 Evaluation

The most important part of creating a good discretization, is the evaluation. While there are many different methods for producing new discretizations, the optimality of the result depends on the ability to select the best between all of these sets. As said earlier, ME is a very good factor for evaluation, as it provides the ability to evaluate each discrete point separately. If every x_i has its own value, then the selection between all points becomes trivial. Because, the ME is correlated to the average distance between a continuous point x and its nearest discrete point x_i, another useful information is the ME of its parent in case of removing that x_i. We will call the difference between these MEs the pruning error from now on. Thus, nodes must keep some useful information that helps on their evaluation. Each node stores the total error produced on it, and information about the pruning error of each of its branches.

For each continuous point x searched in the space, all the associated nodes are recursively traversed from the root to the leaves. Between those nodes, there are some that play an important role. One is the node with the nearest discrete point to that x. This node and its parent, will store a record of the error produced by the distance between their point and x. The distance of the nearest point and x is used to calculate the error produced by this point. The distance of its parent point and x is used to calculate the estimated pruning error. Also, another important node from this traverse, is the leaf, which is the node that should be expanded in order to get closer to this x and minimize the error. After searching a representative sample of x's that follow the PDF, all tree nodes have collected the desired information. Each node has collected the error of its discrete point x_i, which is the ME(x_i). The sum of ME(x_i) for each x_i is the total ME. Also, all nodes have collected the pruning errors of their branches.

Using the previous example, for x=[0.3], the traverse will pass through the root node[0.5], its branch node[0.25] and will finish on node[0.375]. Node[0.25] is the nearest one and the distance *dist*([*0.25*], [*0.3*])=*0.05* will be stored as the error. Node[0.5] will keep a record of the distance *dist*([*0.5*], [*0.3*])=*0.2* as information for the pruning error of this particular child. Also, node[0.375] is the leaf node, which is the only one among these three that has the ability to be expanded, but this is going to be explained later.

4.1.3 Modification

At this point, we have available the ME(x_i)'s and the pruning error for each node of the tree. This is enough information to start making expansions and cuts and change the shape of the tree. There are many ways to choose which nodes to expand and which to cut. The method used in this thesis brings very nice results, as each adaptation always decreases the total ME until it reaches close to the convergence point. It doesn't ensure that the solution will be the closest possible to the optimal, but it is fast and stable.

Because each discrete point has a fixed location that must not change, the goal is to select the points that perform the best. So the adaptation procedure is actually a careful selection using the information collected through the sampling. Each adaptation step has two sub-steps. First is pruning, that cuts unused nodes to save space, and second is the expansion which increases resolution in the regions of interest to decrease ME. In order for the adaptation to be stable, pruning and expansion is applied only to the outer layer of the tree.

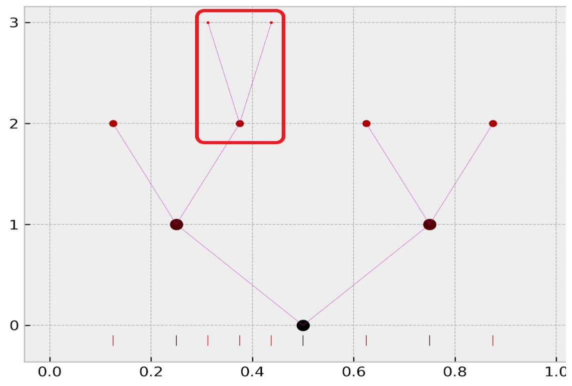
- **Decrease resolution**

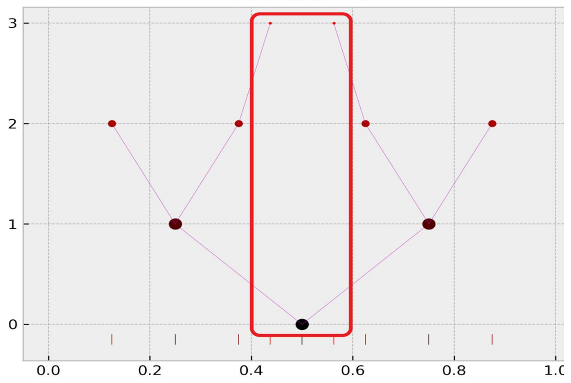
  Decreasing resolution almost any time increases the ME, because almost every node has smaller ME than pruning error. But, it is necessary to maintain the right number of points, so we have to cut as many as we can to make more space for other nodes to be expanded. Pruning is done by simply deleting/cutting a leaf node from its parent. Because leaf nodes have no sub-branches, the total number of points is decreased by 1 for each cut.

- **Increase resolution**

  In order to decrease the ME, the resolution of points in regions that is needed must increase. Obviously, the way to increase resolution on the desired regions is to expand the corresponding nodes. There is a limited number of nodes that can be expanded depending on the current size of the tree, and the maximum size set by user. Nodes with the biggest error are prioritized for expansion, regardless of their position in the tree. Their position on the tree categorize them into two sets. The expandable ones, that have unexpanded sub-branches, and the not expandable that are already fully expanded and are located in lower levels.
  Expanding an expandable node is done by creating all its sub-branches, if this doesn't exceed the maximum acceptable size of the tree. New points are spread in all directions trying to fill up uniformly the space around the parent point. The goal is to actually search this region for a place of interest. Points that fall into this place will be marked as useful by the evaluation process and will "survive". The other points will be deleted in the next update.
  Expanding a non-expandable node is trickier than expanding an expandable one. The

evaluation system marked them for expansion, but they are already fully expanded. Fully expanding its closest expandable children would be a solution, but it will create too many useless new nodes. The other option is to expand some of its descendants from the outer layer; only the 2^n nodes that are closer to the initial non-expandable node.
Finally, all the nodes that will approach the desired point are created. This means that distant and also direct (if there are any) sub-branches will be added to the tree. So, the expansion always creates 2^n new nodes, but not necessarily in the same levels.


Expansion of an expandable node.


Expansion of a non expandable node.

4.1.4 Adaptation

Evaluation and modification are two powerful tools that can be used to achieve a great discretization. Evaluation gives enough information for the utility of each node, and modification makes possible to make any changes on the set of the nodes. Adaptation is the mechanism that combines these two tools. It uses the available information and decides where to apply modifications. Also, this is where decisions about quality and speed trade-offs are made. An extensive search on which nodes should be cut and which should be expanded, would lead to a deeper improvement that throughout time will create an optimal solution. But, the computational cost may grow enough to make it intractable. On the other hand, a fast sketchy selection may provide pure results and finally produce an unconverged solution.

As described in the theory chapter, the optimal goal is to end up with the *k* nodes that produce the lowest error. Some nodes collect small total error because they are out of region of interest and the recursive search rarely traverses them. Other because they reached very close to a point of interest. Cutting nodes of the first case will increase the ME by a small amount and will free one slot for another node to be created and decrease it again. Though, cutting nodes of the second case will lead to a big increase of the ME, as the resolution of points will shrink in a region of interest and the pruning error is very big. But, as it is described on evaluation, each node's parent holds the information that is needed to predict that increase. The difference between this value, and the actual error produced in it, is an estimation for the pruning error. Also, only leaf nodes are available for pruning. Cutting a non-leaf node will result cutting its branches too, which is not always what we want. For expansions, nodes with the highest error are the first one to expand, so they get expanded in descending error order. Note that in expansion, it doesn't matter if the nodes are located in the leaves or anywhere else. There is always the option to add new leaves in order to decrease the ME of any node.

With those two factors available, selection of which nodes to cut and which to expand is trivial. But there are some other things that have to be taken into account, like the stage of the adaptation that the tree is already in. If the solution is found, then there is no reason to keep making changes. Ignoring the adaptation factor, and keep making changes to the tree after that will lead to instabilities on convergence. There must be a criterion that stops the modifications once adaptation is achieved. The error of the nodes outside the regions of interest is the sum of a few big errors, and the error of the nodes inside the regions of interest is a sum of many tiny errors. When these errors are almost equal, we can assume that adaptation has achieved. There are no longer nodes with very high or very low total errors. But this similarity is subjective and cannot be measured or normalized into a standard range. The only factor that provides some information is the average error of all the nodes. Before pruning process, a pass through all the nodes is made to calculate the total ME. Then a second pass is made to cut all the nodes whose pruning error is less than this value. This ensures that, if the predictions of the pruning errors are correct, then there will be no cuts that will increase the total ME very much. Also, when adaptation achieved, no node will satisfy this condition and no cuts will be made. After the pruning, the size of the tree will be less than *k*. The rest of the nodes will be iterated in descending value order. One by one, they will suggest the 2^n nodes that are closer to them for expansion. These suggested nodes are selected in order to decrease the error of this particular node. Expansions will be made that way until the tree reaches again the maximum size. Sometimes, because the number of prunes was way too big, there are not enough nodes for expansion and the tree is left with less nodes than *k*. This of course is not optimal, but it will be fixed on the next update. This happens usually at the first adaptation step where the shape of the tree is usually at its maximum distance from the optimal one and there are many nodes that need to be cut. Another alternative is to expand some nodes more than once, but this will lead to instabilities. So, at each update, the outer layer of the tree can lose and gain one level of nodes at each region. Finally if no changes are made to the tree after an adaptation update, we can assume that the tree is adapted to the current PDF and we can stop making updates. Sometimes though, because of some small changes, this detection malfunctions. This of course doesn't mean that the tree is not adapted. It is just unable to automatically detect it. A threshold value of differences or a maximum number of iterations can solve this problem.

Algorithm: Adaptation Update

```
points_before = tree.get_points()
ME = tree.get_mean_error()
pruned_nodes = 0
for all node in tree.get_leaf_nodes() do:
    if node.pruning_error < ME then:
        node.delete()
        pruned_nodes += 1
nodes_to_expand = max_size_of_tree - tree.current_size
nodes_in_descending_error_order =
                        sort(tree.nodes, key=node.error)
suggestions = []
for all node in nodes_in_descending_error_order do:
    suggestions.extend(node.suggest_for_expansion())
while nodes_to_expand>0 do:
    new_nodes = node.expand(limit=nodes_to_expand)
    nodes_to_expand -= new_nodes
tree.refresh_nodes()
points_after = tree.get_points()
if points_before equals points_after then:
    return True      # adapted
else:
    Return False     # not adapted
```

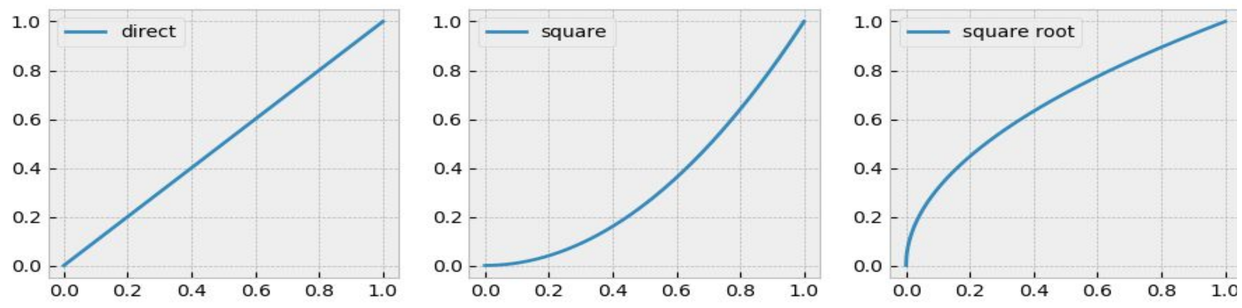
4.1.5 Error function

Now that we know how the errors affects the shape of the tree, it is time to talk about its exact correlation with the distance between two points. As we said before, EF converts distance to error. This conversion aims to give user the ability to focus on some distance values and affect slightly the shape of the adapted tree.

To understand the way it affects the shape of the tree we must first analyze the adapting procedure. When it is time to make an update, nodes will get expanded depending on the total error they collected, from highest to lowest. So nodes with the highest total error will be expanded first. There are two cases for these nodes. In the first case, there are nodes on regions of high interest whose error is a sum of very large number of tiny errors. In the second case, there are nodes on regions with very low interest. Because the resolution there is very low, they cover big regions, and their error is a sum of a small number of very big errors.

Now, with that in mind, let’s assume we have an EF that gives an error equal to the squared distance. Big distances will be affected more than the smaller ones and will produce greater errors. Nodes from the second case will end up with a slightly increased total error and nodes from the first case will end up with decreased total error. This will change their order of expansion on the updating procedure, as some nodes from the second case will climb the ordered list a little bit. Finally, the tree will end up more wide and less tall than before.

Of course, the opposite will happen if we use an EF that returns an error equal to the square root of the distance. Depending on the what user prefers, small distances might be more important than larger ones gain some sensitivity, or in reverse, user might want to spread the points more to avoid saturation of points. There are many options for error functions, but for this thesis, we will use only square, square root and direct EFs.



Error functions used in this thesis.

4.1.6 Performance

One key feature this architecture should have, was its ability to take care about the adaptation without disturbing the real time response of the agent. This means that it should perform in sub-linear time at each step. This method requires two procedures in order to work properly. The feeding procedure where samples of *x*s that follow the current PDF are fed into the tree to calculate the errors and estimate the PDF, and the update procedure where the modifications and the adaptations occur according to these errors.

Starting with feeding procedure, it is done by making the corresponding searches for each *x*. A traverse from root to a particular leaf is made for each search. Depending on the shape of the tree, this ranges from $O(log\ k)$ for a balanced tree, to $O(k)$ for a tree that grow up mostly on its depth. Also, the *x* samples have to be enough in number to give a representative view of the PDF. Assuming that *N* is the number of sampled *x*s, the whole feedback procedure will cost between $O(\ N\ log\ k)$ and $O(N\ k)$. It is important to note that the probability for the tree to end up being a “list” and the search to cost $O(k)$ is so small that there is almost no need to take into account.

The cost of the update is bigger but is paid much more rarely. Even the fastest update requires a couple of passes through the whole set of nodes, which means that in the best case, is scales linearly with *k*. For an extensive update, using dynamic programming techniques that are able to predict very accurately all the possible results in order to select the optimal one, the complexity becomes exponential. Adaptation is done once per many steps, which makes a linear or even slightly bigger complexity affordable. Though, exponential complexity may sometimes be a problem even for a very rare use. Actually, it depends on the problem whether it is possible to pay that cost or not. The method suggested here provides stable results that are converging on a decent solution the vast majority of times, with an acceptable computational cost of $O(k \log k)$. Each pass through the whole set of nodes costs $O(k)$, while the sorting that needs to be made in order to expand the nodes with the highest errors costs $O(k \log k)$.

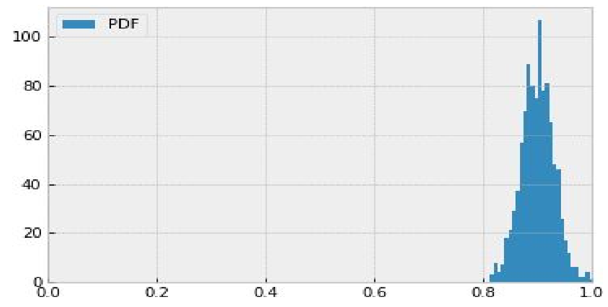
These costs are not too high, and there are many options on when to pay them. It is not mandatory to make the search right when a new x is sampled, as it can be stored to a buffer and feed them all together before the update. This will leave the real time response unaffected. Additionally, updates are essential for the adaptation, but their frequency can be reduced as we get closer to the final solution. It is all about what user needs, and what the problem requires. A PDF that is constantly changing would require frequent updates whereas a stable PDF will need a few updates in the beginning until adaptation achieved, and never again. This is a general purpose tool that can be used in most cases, but can work more optimal in special cases.

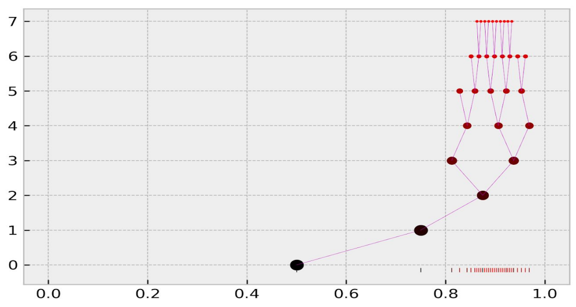
4.1.7 Efficiency

Efficiency here describes the utility of each point in a discretization. The fact that points have fixed locations, constrains the overall efficiency of this architecture. Some points contribute more to the decrease of ME while others are necessary and act as base branches that support other branches on top of them. Other approaches of spatial division that use trees, delete parent after expansion. When a reduction is required, all the child nodes merge together to create again the parent. This forces all the child nodes to remain and reduces the ability of deleting single leaf nodes which often is a waste of space and count as overhead. Keeping the parent after the expansion makes possible to delete any of the children with only a little overhead.

All the locations of the discrete points are predetermined by the geometry of the tree. For example, root will always be at the center of the unit cube, its children will split the space into predetermined equal sub-spaces and so on. The final distribution will be result of selection of the right subset of possible discrete points. Child nodes can’t exists without their parents, thus reaching a particular region of the space requires a number of ancestor nodes to predate. These base nodes may not be necessarily useful on the reduction of the ME while they take up space. The nodes that actually do decrease ME, are called optimizer nodes and most of them are on the outer layers. Some can be both base nodes and optimizers, but the count only as optimizer when efficiency is calculated. Also, some nodes are not used as base, but don’t contribute on the decrease of ME either. They have no meaning of existence and they will be deleted on the next few adaptations. Finally the efficiency is described by the ratio of the optimizers to the total number of nodes.

Base nodes are usually on the lowest layers of the tree. Few levels higher, the tree is more flexible and can spread on the whole space to reach very easily any region of interest. Because of the expansion rate, each level adds exponentially more nodes that count as optimizers. Base nodes quickly become a very small portion of the total nodes and the efficiency rises. Of course PDF affects the efficiency, but in general the smaller the tree, the bigger the chances of being more inefficient. In conclusion, the geometry of the tree has a size threshold, under which the tree has not enough nodes to adapt properly to the PDF. On the example below, we can see how a tree with 31 nodes adapted to this particular PDF. Nodes at level 0 and 1 are base nodes, as they are actually outside of the range of the PDF. Nodes at level 2 and 3 are both base nodes and optimizers, as they are inside PDF's range, but they mostly support the layers of optimizers above them.





An example PDF. The corresponding adapted tree.

Speaking about efficiency, there is one more that worths to be mentioned. As described before, the initial size of the tree may differ from the maximum size that user set as parameter. This difference comes from the fact that the maximum number of points must never be exceeded so the level of initialization is calculated as the maximum possible level on which the tree doesn't exceed this limit. This means that the difference on the size may be big in some cases, but it is actually only one level away. Of course this is will produce a bigger ME on the use of the tree before the first update. But this difference will be replenished immediately on the first update.

4.2 Adaptive action space

Now it is time to plug the adaptive discretization tool on the agent's action space and configure it to work properly. This tool needs some information to work in the first place. Firstly, it needs the number of dimensions of the space and the maximum number of discrete points. Both numbers are available to the agent on the initialization. The limits of the space are also known and the only information left is the continuous points x sampled from the PDF.

Because action space has to be adapted to the needs of the agent, PDF must be a distribution that describes this information. There are two options here. Actor's proto-action and Critic's result action. As we said earlier, Critic's main job is to train the Actor, so its ability to select actions is not that reliable, while on the other hand, Actor is the heart of the action selection process. Also, using Critic's action may result to an action space that constrains the Actor, as less discrete actions may be near to proto-action. Finally, Critic's actions are discrete and cannot provide the desirable resolution for PDF. So Actor's continuous actions will be used as continuous points *x* to be fed on the tree.

Another issue is the frequency of the adaptation updates. The number of sampled *x*s must be big enough to be representative to the PDF, but small enough to take most advantage of the adaptation and to be always updated in case of small changes. The updating procedure starts when agent requests an adaptation. After waiting for the adaptation to finish, agent can use the resulted new discrete points as discrete action space. Of course nearest neighbor search module (FLANN) have to be reset after that so it can return the new nearest neighbors of each point. This whole process might take enough time to disturb the real time response so it has to be done when the agent is not very busy, for example at the end of an episode. After it is done, agent can continue with a brand new action space to use.

4.2.1 Adaptability on training

As the RL agent interacts with the environment, it passes through a number of different stages. In the beginning, it has no idea of what the goal is, what the state vector means and how its actions influence those states and the environment. After collecting some experiences, it starts understanding the mechanics of the environment and starts exploring the state and action space. Finally it understands its goal and tries to reach it and bring optimal results. All these stages have a different imprint on physiology and the behavior of the agent. The use of the action space in all this stages is different. In the beginning agent makes almost random actions, and while it starts understanding how things work, it starts making more specific use of the some portions of the space.

But environment is pretty much random, meaning that there is no certainty on where all the different scenarios will appear. The same is true for the training of the neural networks. We can't predict the time that the agent will be trained or even the time that it will start providing some correct results. Before that, both Actor and critic are spitting out random results as proto-actions and state-action evaluations. The adaptable action space will adapt to these random actions and will constrain the agent from searching any other regions of the actions space, as the resolution on these region will drop. After the point that agent starts gaining some understanding about the environment and starts seeking for optimizations, it searches the state and action space passing through many different regions. Again, it won't be profitable to make any changes to the action space while agents searching it. Due to the above facts, adaptation of the action space is better to be disabled while training, or at least at the first stages.

4.2.2 Square error function

Agent uses some actions more times than others, but this is not necessarily means that the ones that uses less are not important. Adaptive discretization tries to minimize the total ME and in this process it deletes points that are used less if needed. If some points are used the vast majority of times, it will focus on them because they produce way more ME that the others. This may eliminate the other points, leaving the agent without the option to choose actions from some regions. In some cases, this can lead to a decreased performance, making the uniform discretization a better pick. To reduce or even eliminate this effect, we can use an error function that will produce a wider tree, such as a square function. That way, errors produced by small distances on the regions of high resolution will decrease, and errors produced by greater distances on regions out of interest will attract more attention.

Another issue comes from the fact that this particular agent makes an approximate nearest neighbor search in the space, which returns a constant number of points K, as it is described on the introduction. Increasing the resolution on a region means that this search will return neighbor points from a smaller range of the space. These actions will be very similar to each other which leads to saturation. This will make the agent unable to pick the best action on the refinement as all alternative actions are now almost the same. Action refinement can take advantage of a bigger variety of different actions, not bigger on quantity. Again, a square error function can provide a solution. Because it produces a wider tree, and the number of total actions is constant, the tree will end up shorter too. Shorter tree means slightly lower resolution on regions of interest, which decrease the saturation effect. Because square error function reduces both effects, we use it as the default option in this thesis. For other cases or other agents, where the maximum resolution is poor, a square root function will fix it.

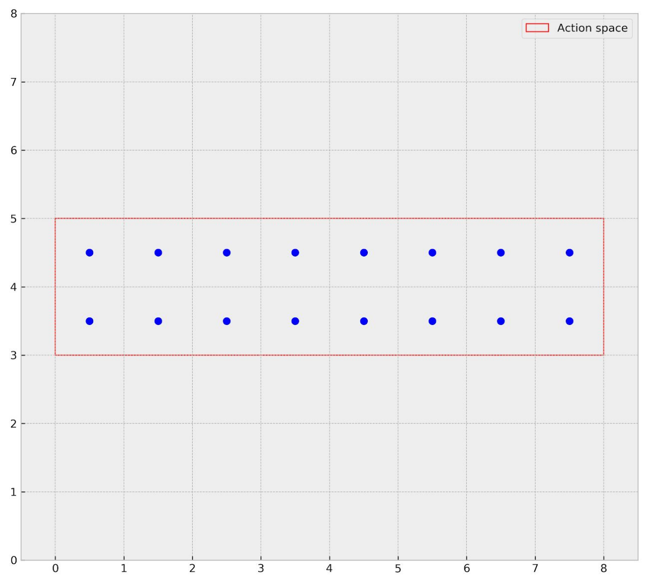
In conclusion, the agent needs a supportive tool to improve its freedom of choosing actions and counteract the restrictions made by the discrete action space. The actual selection have to remain on its hands and the adaptive action space has to play a secondary role by just following the agent. There must always be discrete actions close to any proto-action and a rich variety of different actions among the *K* neighbors.

4.2.3 Normalization of action space

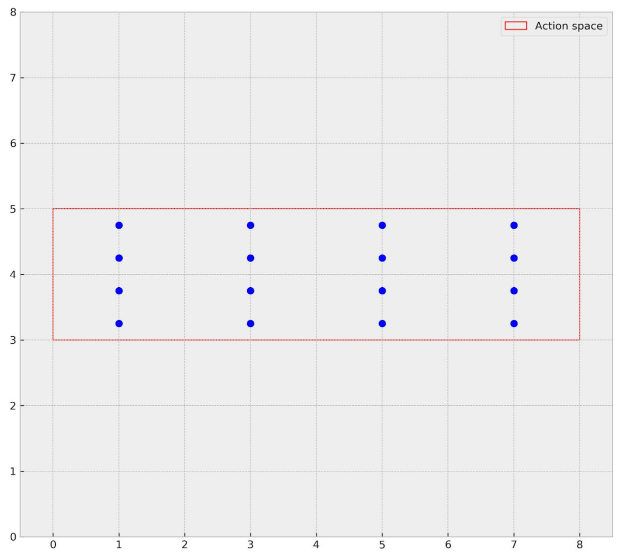
We previously assumed that the range on each axis of the space is [0, 1]. The reason for that is that we normalize the action space. From now on, *a*=[*0, 0, ..., 0*] and *b*=[*1, 1, ..., 1*] for any space *R* which makes this n-dimension rectangle a n-dimensional unit cube. Each *x* will be linearly transferred into this space, where all the discrete points sit. After the nearest neighbor search, all the *K* actions will be transferred back to the original action space, to be evaluated by the Critic. There are two reasons for that.

One reason is to simplify things, as from now on there will be no need to care about limits and ranges. The only overhead of this procedure is the two transformations, the first to import the proto action into the unit cube, and the second to export the K points to the action space. These transformations are just a couple of linear operations, a minor overhead that doesn’t affect the real time response.

The second and more important reason is to make nearest neighbor search work properly in any case. When the action space is more stretched in some of its directions, discretizing becomes tricky as the sensitivity on each axes and the distance between the neighbor points can’t be the same. For example, assume there is an environment with 2-dimensional action space, where x-axis is ranged between [0, 8] and y-axis is ranged between [3, 5]. A uniform grid of 16 points will be like:



This discretization gives 4 times greater resolution in x-axis that y, meaning that the agent's sensitivity in x will be 4 times more. The other option it to make the discretization uniform in a relative way, with relative distances, to get a 4x4 grid.



The problem now is even bigger because actions in y axis are closer to each other relative to actions in x. Searching for the nearest neighbors of point will return more points from y-axis than x-axis which is suboptimal. Normalizing the action space will detach the sensitivity and the distance of the points from the geometry of the the space, and will fix both issues with almost no trade-off.

5. Results

The first part of this thesis was to find an architecture that implements the adaptive discretization, and the second part to embed this architecture on a discrete RL agent. Because these two parts are independent and have different evaluation factors, we have to test them separately to conclude for each one.

5.1 Adaptive Discretization

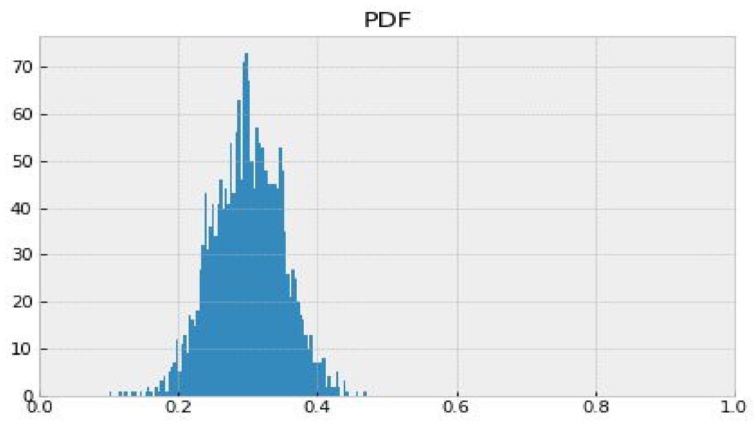
As we said in the beginning, discretizations are evaluated through their accuracy. In other words, a discretization is good when it matches the PDF of the space. For adaptive discretizations, two more evaluation factors are added. The adaptability, or in other words the ability of the methods to adapt to all kinds of PDFs, and the time needed to do this, or the adaptation time.

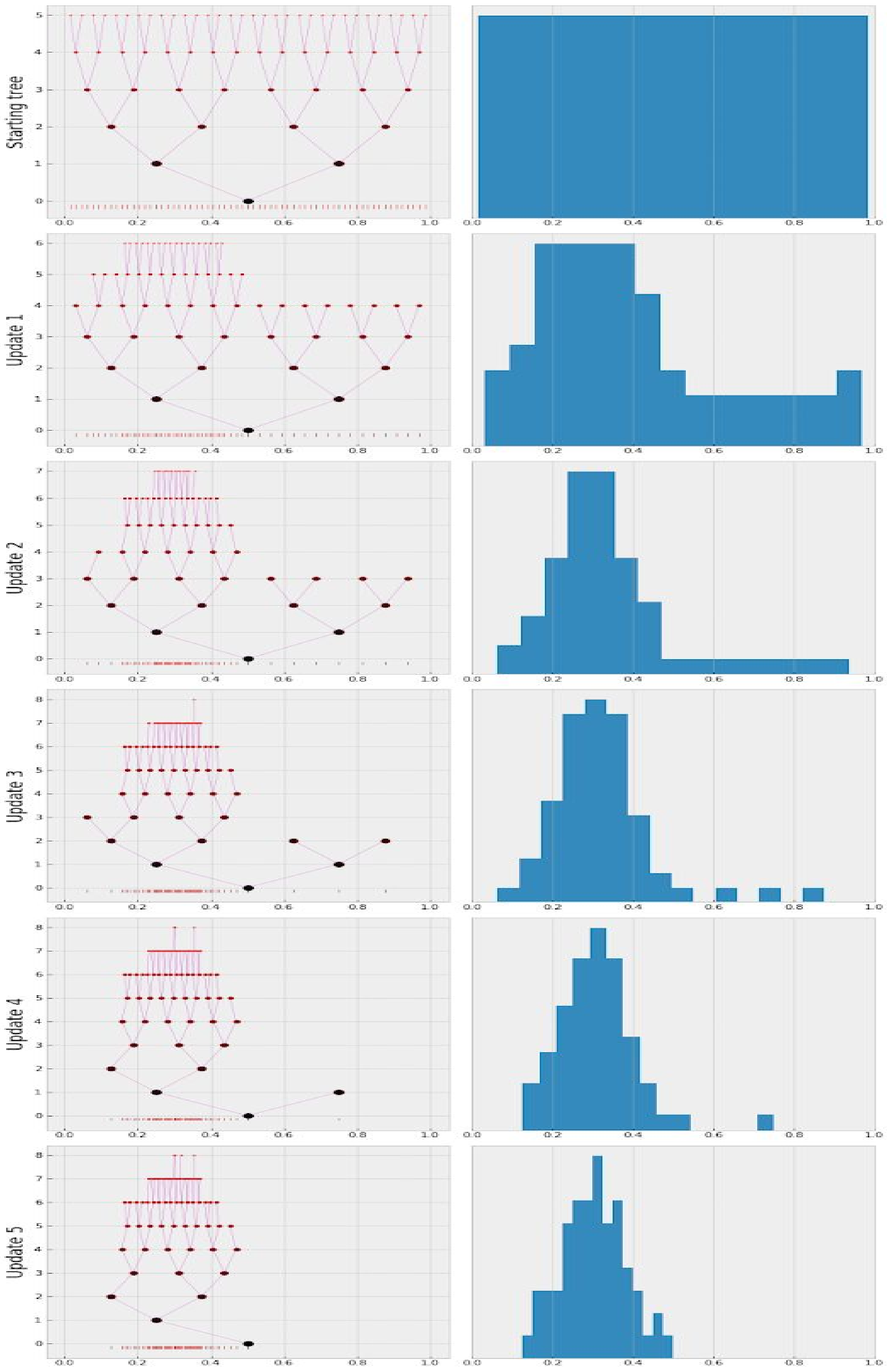
The accuracy of a discretization on a certain space is measured as the average distance from the continuous points that appear in this space according to the PDF, to the nearest discrete points of the discretization. This accuracy must be compared then to the accuracy of a uniform discretization. Also the number of updates needed to achieve this adaptation must be considered too. Lastly, adaptability and efficiency can be evaluated through the accuracy achieved for a variety of different PDFs, and through a comparison of the resulted shape of the tree to the actual PDF.

Before going to the results, we must note that evaluation on low dimensional spaces is easier as it is possible to visualize them. So we will use them more here. There are infinite combinations and extreme cases that will not perform as the experiments below. For this thesis we must refer to the behavior that describes the most usual cases. So the results will be representative to the common case.

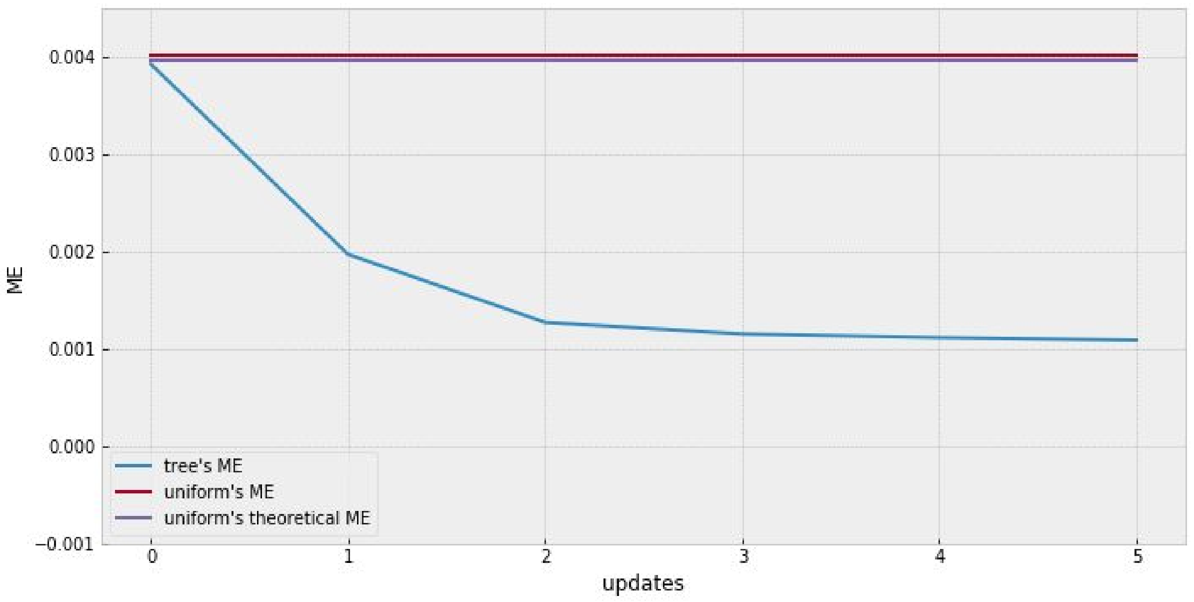
5.1.1 Adaptation

At the beginning, just for the demonstration, we will present an adaptation of a single dimensional tree to a simple PDF, step by step.



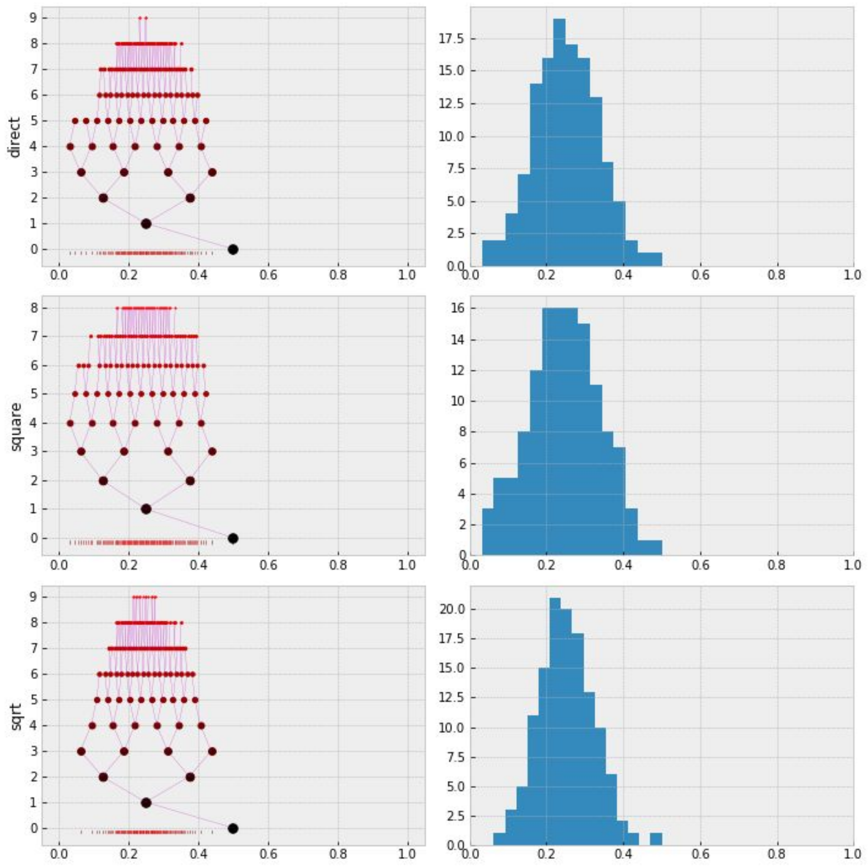

Starting tree
Update 1
Update 2
Update 3
Update 4
Update 5

We can see the tree on the left side, and the distribution of its points on the right side. We can also see that at each step, the tree can lose or gain only one layer. On the area around 0.3 the tree gains one layer at each step, and on the area at the left of 0.5 the tree loses one layer a each step. We can notice that the adaptation time depends on the levels of the tree and this restriction. If a certain number of layers had to be removed or added, then the adaptation can't take less steps than this number. Also, the final distribution of the points matches the most with the given PDF. This match is getting better with bigger trees. Finally, the following diagram shows clearly what happened with ME.



The ME of the adaptive discretization started at the value of the ME a uniform discretization would produced, and ended up at almost a quarter of this value. At the first update it decreased to half of its initial value, and very soon it adapted and reached the minimum value. Note that ME never increased in this process. Also, note that in this particular example, the ME of a the uniform discretization happened to be slightly above the theoretical value. This happens because the theoretical value assumes that the PDF is uniform and errors greater than this value are eliminated by the corresponding errors that are lower than it. If more points happen to be in the maximum possible distance of their nearest discrete ones, then the ME will be greater than the theoretical one, and in reverse. Something that won't happen with an adaptive discretization.

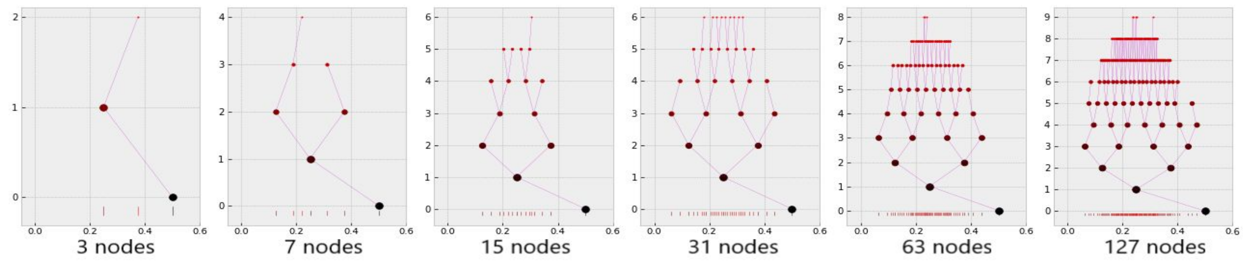
Next step is to demonstrate the effect of the error functions. We will use a similar PDF as before and trees with the exact same sizes.

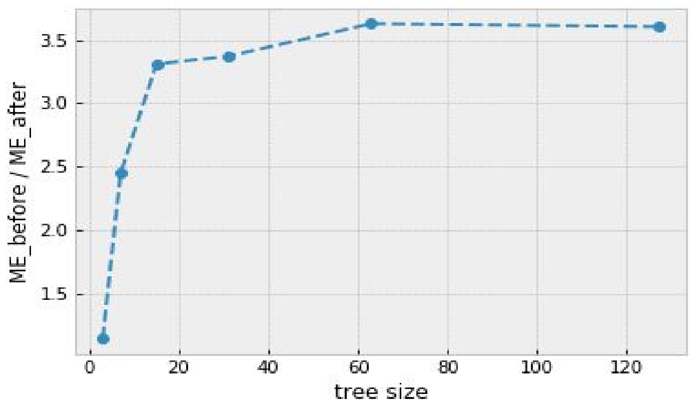


As expected, square error function produce a tree wider and shorter than direct, while square root error function produces a taller and more narrow one.

5.1.2 Efficiency

Measuring the efficiency of tree would be trivial if we knew the number of base and optimizer nodes. Yet, there is no simple method of classifying the nodes of a tree into base nodes and optimazires. Factors like the distance to the nearest high resolution region and the error of a node comparing to its height can be taken into account. However, in most of the cases these factors won't produce any reasonable result. So, as an alternative, we can use the ratio of the ME before any update, and the ME of the adapted tree. Once again, the same number of dimensions and the same PDF will be used. Additionally, to eliminate the initialization size error of the tree and its imprint on the ME before the first update, sizes will be selected to much the sizes of full grown trees.



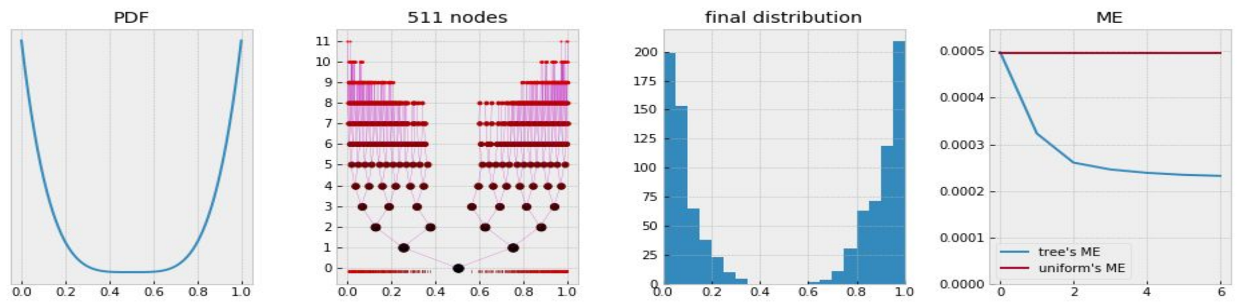


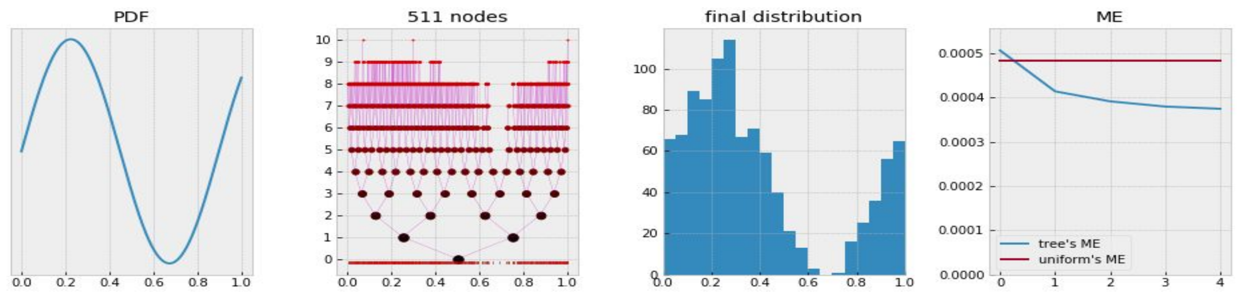
As expected, below a certain size the tree is unable to work properly, but above that point, the tree reaches close to its maximum efficiency, like a threshold value. It is very important to mention that this result are not the same for any tree and for any PDF. Some regions can be reached easier than other, which means that the same trees with a different PDF would perform slightly different. Complexity of the PDF plays an important role as well. If there are many regions of interest, optimizers have to split to those regions and require more base nodes too. But most of the times the behavior of the adaptation is similar to the above. In conclusion, as we said earlier the results are representative and describe an overall behavior.

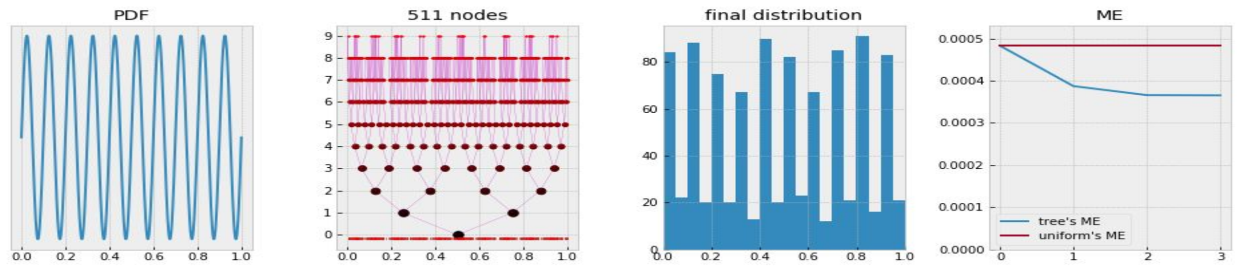
5.1.3 Adaptability

To test adaptability, few complex shapes of PDFs will be tested for the visual demonstration. For each row, the first plot shows the PDF we used for the samples, second plot shows the adapted tree, the third plot shows the distribution of the points of the tree, in other words the final distribution of the adaptive discretization, and last plot shows the decrease of the ME comparing to the ME produced by a uniform discretization of similar size.

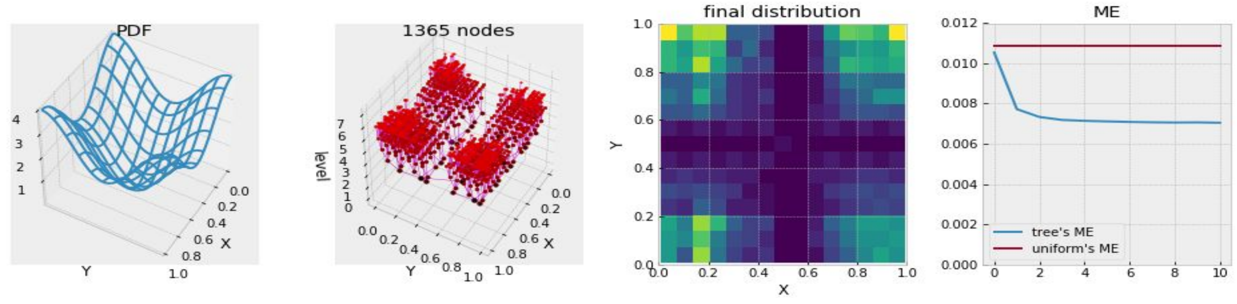
1 dimension

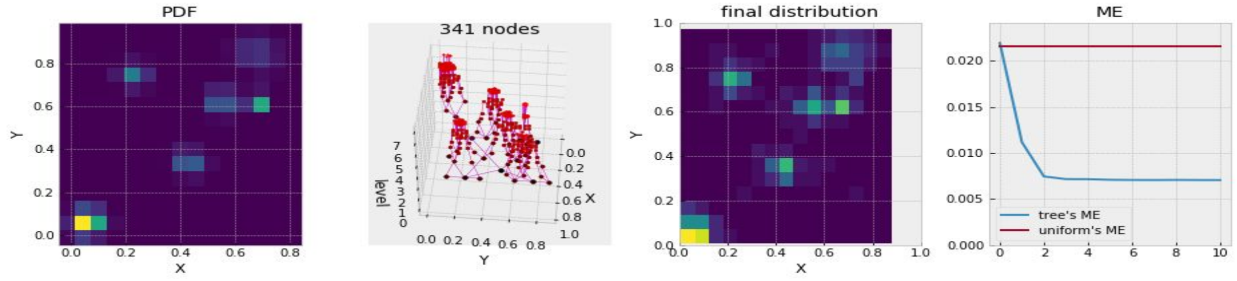




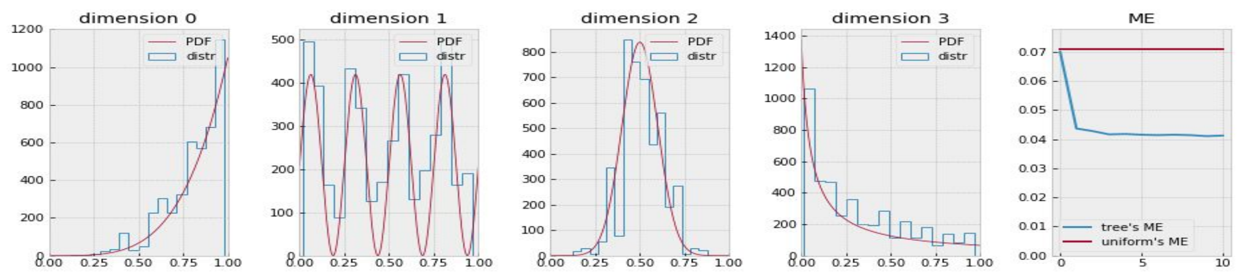


2 dimensions

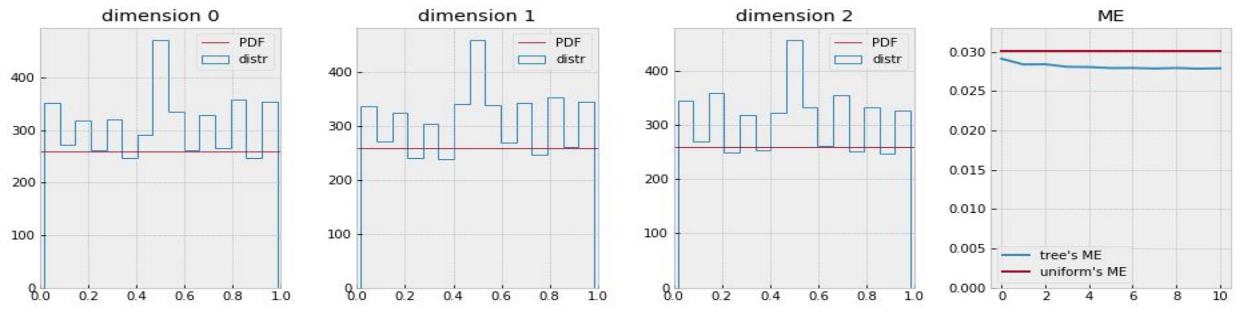




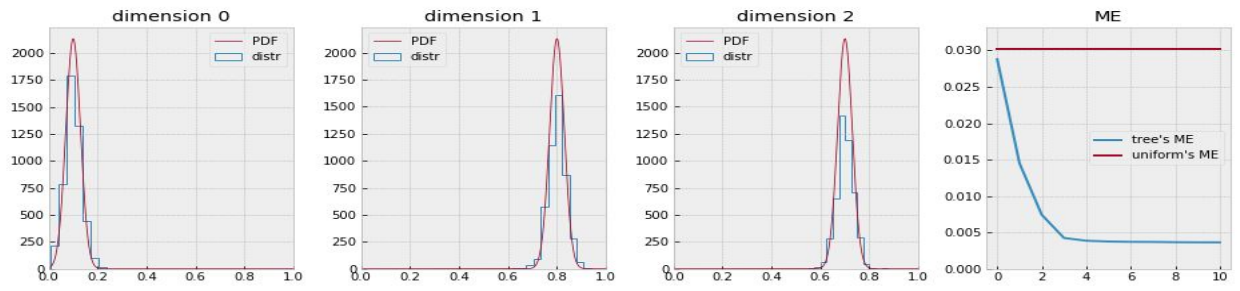
More dimensions



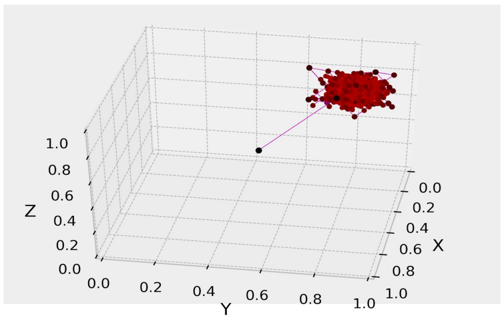
One the last experiment, we sampled 4 different PDFs, and used these samples as coordinates of 4-dimensional points, on the first coordinate samples from the first PDF, one the second from the second PDF and so on. So this function has a different shape for each different axis. We then fed this samples to a tree of size 4369 nodes and update until it adapts. After, we did the reverse procedure. We take the points of the adapted tree and plot the distribution of each coordinate-dimension of these points to see if it matches the corresponding PDF. There was no other way to visualize this experiment or the shape of the PDF or the point distribution. Of course, PDF graphs are not to scale.



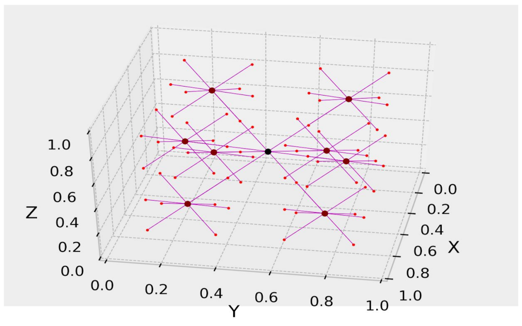
Also, a uniform PDF should be tested as well. This 3-dimensional PDF was tested with tree of size 4681. This method is designed to work well on non uniform PDFs so as expected it does not perform well. It is still slightly better than uniform discretization, because of samples are not perfectly sampled and don't match exactly a uniform PDF, and adaptive discretization can take advantage of this. Also, the size of the uniform discretization is 4096 (16^3), which is about 13% less that the size of the adaptive. So in fact, adaptive performs slightly worse than uniform. One last thing we notice in this situation is that there are tiny bumps on ME on the first updates which means that it is not always decreasing, due to imperfections on the adaptation update.



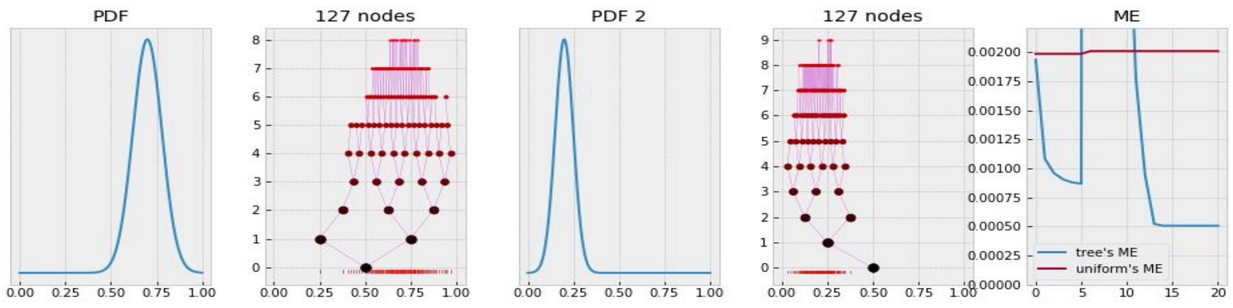
A PDF with a single lobe, that takes a very small portion of the space. We can see here that the tree manages to reach that location with the least amount of base nodes, and then it expands all its available nodes there. This is the best case scenario for this method as it manages to maximize its efficiency.



Tree adapted to single-lobe PDF



Tree adapted to uniform PDF



Finally, this example demonstrates multiple adaptations. We let the tree adapt fully to the first PDF and then we swap to a completely different one. This is an extreme case, as most of the times changes on PDF are occurring slowly. This example aims to show the ability of these methods to adapt to a changing pdf. On this process, ME rises as new base nodes have to created before new optimizers start decreasing it again.

Similar behavior as the examples above, is extended for any number of dimensions. If the size is above the lower efficiency limit, the tree will adapt. Bigger size will result a better adaptation. Though, as the dimensionality rises, exponentially more points are needed.

5.2 Adaptive action space in RL

Applying Adaptive Discretization to this particular discrete RL agent was a straightforward process, thanks to the fact that it is completely detached from the actual discrete action space. Evaluating this application is a bit trickier though, because it is difficult to isolate the results of the adaptation from all the other factors. Wolperdinger agent is a bit unstable at the beginning by itself as its actor and critic neural nets are initialized randomly. The environments (or domains) that were used are also initialized randomly at each episode, so different agents architectures, or the same agent but trained differently might end up using different policies that may benefit more from an adaptation of the action space. While it’s very easy to examine if this method actually improved agent’s performance or not, all this noise make a precise evaluation of how much it is improved impossible. So we will focus more on showing the quality of the results.

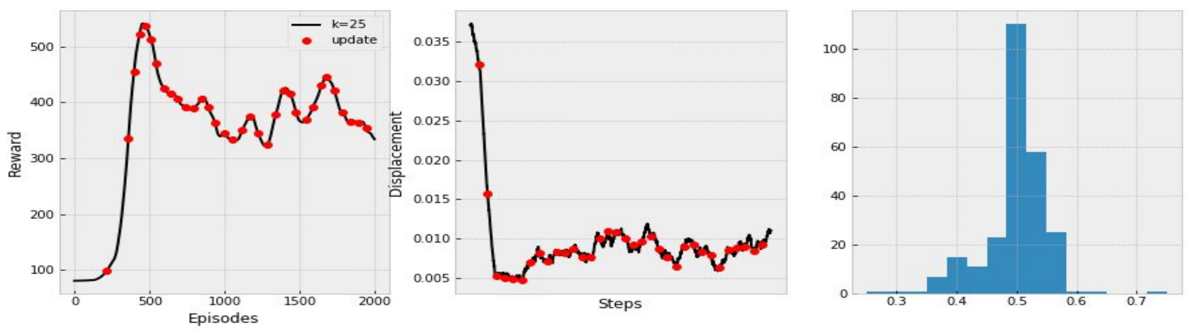
Candidate domains for this architecture are domains with continuous state and action spaces. A very important feature of each one is to implement a deterministic like behavior so the agent will be able to generalize over the states and the actions. Applicable problems with these features are optimal control problems and real world games. Another feature is for the domain to reward sensitivity of the actions of the agent. For example, a domain where the goal is to just balance an inverted pendulum between a certain range of angles does not require that much sensitivity as the agent can keep it in this range with jerky actions. If the goal is to keep it as close as it can to zero angle, then more gentle actions have to be taken, something that will take advantage of the adaptive action space.

5.2.1 Inverted Pendulum

The goal in this domain is to keep a stick as upright as possible by applying forces to its bottommost part. Balancing occurs only in one axis. Positive reward is given depending on the distance of the perfect upright position. This problem has a 4-dimensional state space where the position and the velocity of the bottommost and uppermost points of the stick are described, and a single-dimensional action space that refers to the force applied on the end of the stick. The environment resets when the episode ends. This happens after a maximum number of steps, or when the angle of the stick becomes greater than a threshold value (where we assume that it fell) or when the whole stick goes out of the space limits (on the left or the right edge) as it cannot move further to this direction.

This environment is one of the implemented openai-gym mujoco environments. Its reward function is giving a small positive reward for each step until the episode ends. This reward doesn’t motivate the agent to keep it upright, as it can swing it inside the acceptable range of angles and take the same reward. To make it keep the pendulum closer to zero angle, and check its ability to be precise, we changed this reward to be a positive number proportional to the angle.

Let’s begin with a typical performance plot of the agent on evaluation run, and the corresponding final distribution of the adapted actions space. The size of the action space in this example is 255 discrete points and the k nearest neighbors are 25 (10%).



The first two plots are showing the performance of the agent. The first plot shows the reward per episode. Red dots are indicating the episodes that an adaptation update is made. Notice that their frequency is not constant because the x axis is counting episodes which have not constant length. There are 3 distinct parts on this plot. The episodes before the first red dot, where is the performance of the uniform discretization where no adaptation updates have been made. The second part is the maximum peak. This comes after a few updates and is the maximum performance that the agent achieved with only the help of adaptive discretization. Last part is the decrease in performance that is caused by the saturation of the action space. Sometimes, there is no saturation and this part is an extension of the previous one.

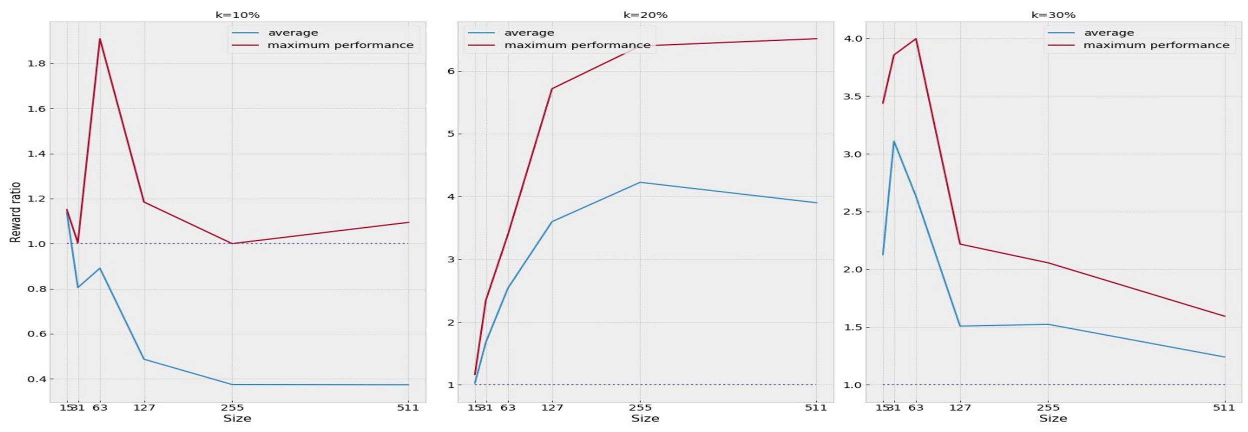
The second plot shows running average on the timeline of the displacement. Displacement is the distance of the upper part of the stick to the lower in x axis, which we use as an angle. This is a more accurate evaluation factor as is shows how adaptation affect the ability of the agent to keep the stick upright. It is more accurate than the reward plot because the total reward of an episode is affected by other things too. On this plot, updates are shown to have a constant frequency because x axis is in steps. At the points that this frequency drops is due to an adaptation that didn't actually change anything.

Finally, the distribution of the discrete points at the end is the expected one. Agent uses big forces for the first steps to bring the pendulum upright and little forces for the rest of the episode to stabilize it. We can see that the middle spike contains more than 100 of the 255 actions, which is about 40%. Considering that the histogram uses 15 discrete bins, this density would require a uniform discretization of more than 1500 points to provide the same resolution. This means that this method gave us the ability to use a discretization 83% smaller, with almost the same results. Distribution plot also gives much information about the policy that the agent adopted.

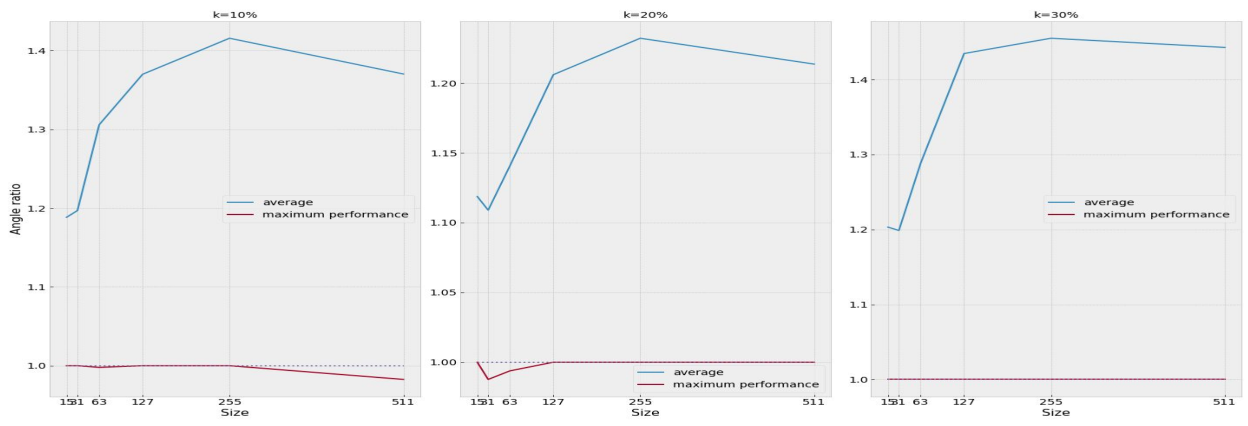
We run many different test configurations to get a clear view of the performance of this method relative to some factors. Also we tried two agent trained differently. The one was trained extensively, while the training of the second stopped before it managed to optimize its policy. Because there are many different test configurations, we can't show all the plots separately, so we categorize them in 2 plots for each agent, one for the rewards and one for the angles/displacement. We tried sizes of 15, 31, 63, 127, 255 and 511, for 10%, 20% and 30% nearest neighbors (k).

Less trained agent

Reward:

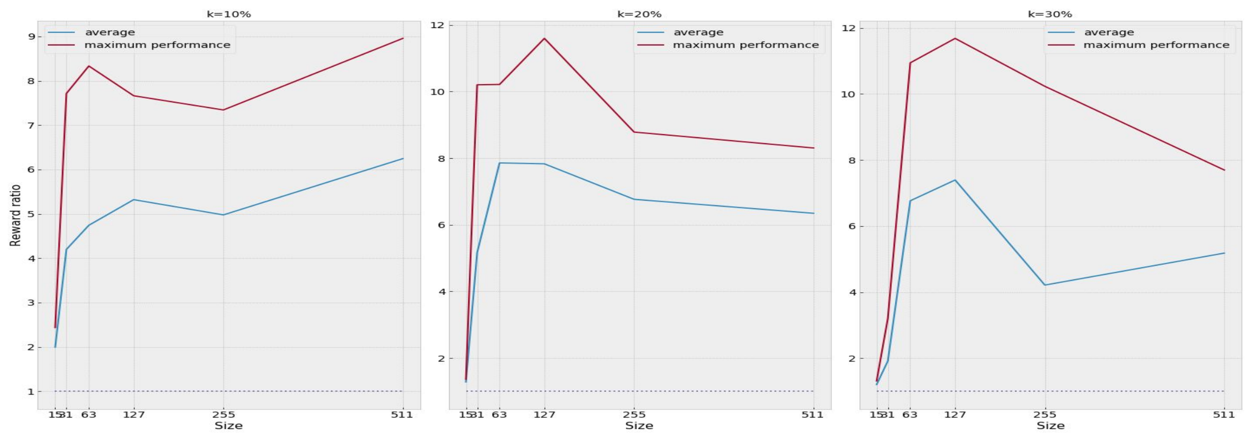


Angle:

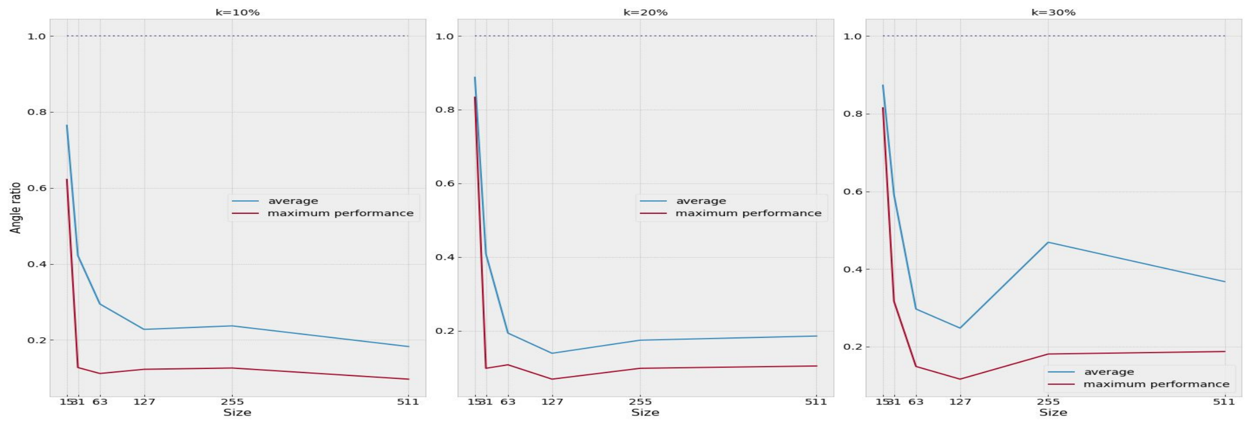


Extensively trained agent

Reward



Angle



Before analyze the data, let's describe what we actually see on these plots. Each plot contain the performance of the agent for each number of k-nearest neighbors. Reward and angle performance is plotted separately. On x-axis there is always the size of the action space. On y-axis, there is a ratio between the average(blue line) or maximum(red line) performance of the agent with adaptive action space, and its performance without it. A ratio above one means better performance on rewards. Because the goal is to reduce the angles of the pendulum, the ratio in angle plots is better when it is below one. Average performance is the average value of the rewards or the angles after the first adaptation, and maximum performance is the average value of the rewards or angles between the adaptation that brought the best results and its next. If on an experiment, the agent reaches its maximum performance and saturation doesn't affect it, then this performance will remain. This will keep the average and the maximum performance almost equal. In reverse, great difference between these two lines means big saturation most of the times. So we have almost all the data on these four graphs.

Starting with the well trained agent, it manages to increase its performance at least 5 times on all configurations for action spaces with size greater or equal to 63. For smaller ones, there is still an increase but a smaller one. This is similar to the efficiency graph we saw on the architecture of adaptive discretization. We can assume that the efficiency limit for this PDF is around this size. For k=10%, performance rises as the size increases, while for the other two ks it drops, especially for the k=30%. As more decision making is shifted to Critic, saturation effects get bigger and bigger. If Critic was able to make correct decisions, the increase of the resolution combined with a high k value would be very beneficial for the agent. The best performance comes for sizes of 63 and 127 and ks of 20% and 30%. At this point, there is a great variety of discrete actions to select while saturation effect is still low.

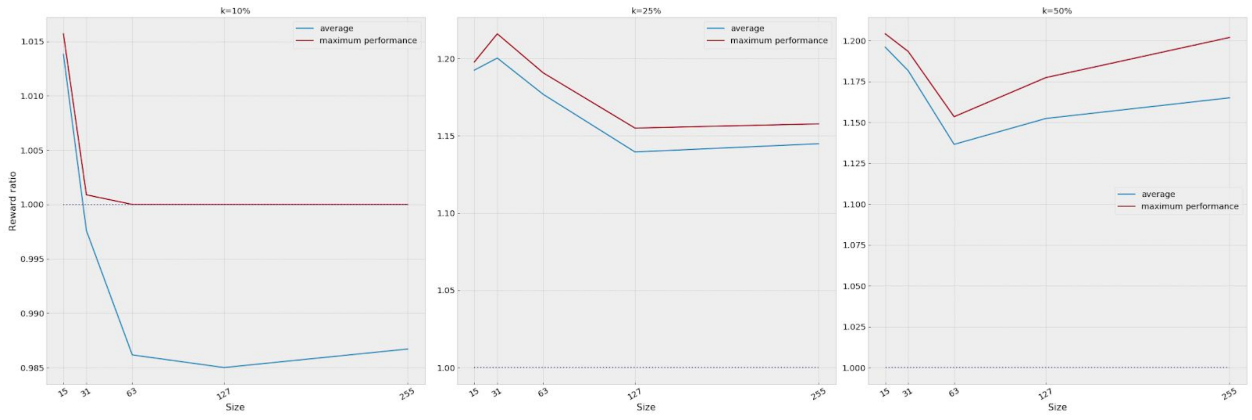
Proceeding to the less trained agent, things change. For k=10% its rewards drops, while at its best, it barely manages to produce results higher that the results of a uniform discretization. Of course at this time Actor is not trained enough to produce good proto-actions. As the action space adapts to these proto-actions, situation gets even worse as the density of points increases in regions of space that is not needed, and decrease on region that is needed. For k=20%, reward performance is pretty good but it quickly drops again when increasing k to 30% due to Critic saturation. Though, it still performs better than uniform. Angles on the other side are constantly worse than uniform's. This seems as a disadvantage of the adaptive action space at the first glance, but if we combine the results with the performance on the rewards we can clearly see that it is not. What is happening actually is that the agent at this point hasn't yet figured what the goal is and has adopted a slightly wrong policy. Its policy at this time makes it to oscillate left and right around zero angle. This oscillation will shrink after some more training time when it corrects its policy. The reason that it manages to increase its rewards with this angles, is that adaptive action space actually helps it to follow this policy. It gives the agent the ability to be more precise on this oscillation, which keeps the pendulum from falling. So it is actually an advantage for adaptive discretization and for the agent too, as it increases its performance even before training and with a wrong policy.

5.2.2 Cart-Pole

This domain is just like Inverted-Pendulum, but it is open-source. This gives us the ability to experiment further with the reward function and the dimensionality. First we removed the spatial restriction for the position of the cart, giving the agent the freedom to use as much space as it needs to balance the pole. This will result a smaller variation on performance, and a more precise inprint of the angles on the rewards. So for this experiment, there is no need to use angle graphs, as the information on the rewards graphs contains all the essential information. We also added the ability to extend the problem to one more dimension, so that the agent will have use forces on both axes to keep the pole upright. Additionally to make the adaptation more complex, forces on one axis have twice the range of the forces on the other. This will make the adaptive discretization to adapt differently to match the desired sensitivity.

1 dimension

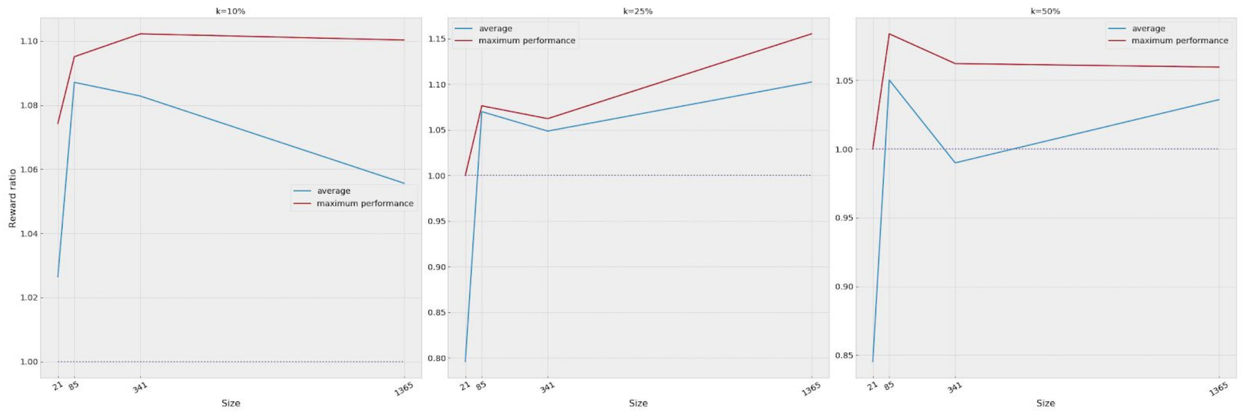
First we quote the results of the 1d version for reference, because cart-pole and inverted pendulum may use different physics parameters. Of course, we will need two different agents for 1d and 2d, which may end up with different policies. This difference in policies will affect the performance for each setup and the magnitude of the increase. But as we said earlier, because of all these differences on the agent policies and on the environmental noise, we can't provide any precise information for the magnitude of the increase.



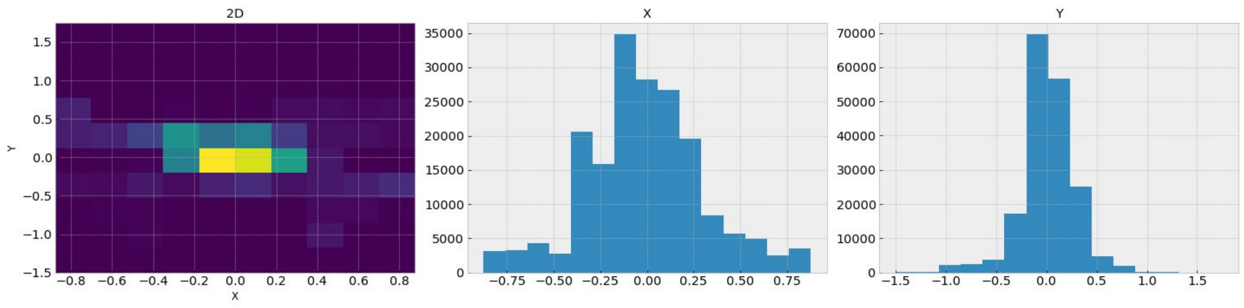
As we can see, due to this freedom of movement, agent is producing much more stable results, and the improvement is smaller as the reward is affected only by the angle.

2 dimensions

Because action space is now spread into two dimensions, there are more nearest neighbors in a close distance. For this reason, we have to use bigger values of k, to represent a similar "range" of the space.



The use of space is similar to the previous experiments, but extended in two dimensions. This will produce a tree wider on x axis and more narrow on y to produce the same sensitivity.



The quality of the results for one and two dimensions is similar to the inverted pendulum. On most of the cases adaptive action space increase the performance of the agent, especially for medium size discrete action sets and k~=20%.

6. Conclusion

The presented method adapts decently to all the possible PDFs, regardless the shape, the complexity or the dimensionality. This adaptability, means that it is able to produce a very wide range of different resolutions across the space. Also, it performs better than a uniform discretization in almost any case, as it produces an almost optimal ME with low computation cost overhead. From the very first adaptation steps it almost reaches close to its maximum performance. This allows adaptation on PDFs that change over time, with low ME increase. The worst case scenario is PDFs similar to uniform, with flat regions, where it performs slightly worse. Still, if there are any imperfections to that uniformity, adaptive discretization will take advantage of them.

Nevertheless, the proposed method has some disadvantages too. Firstly, it doesn't work for any number of discrete points, as it has a lower limit. Depending on the dimensions and the shape of the PDF, there is a certain number of base nodes that have to be created. Sizes smaller than this number, make this method work inefficiently, as there are not enough optimizations to actually decrease the mean error. Also, there is a limitation that comes from the adaptation update complexity. Because of this complexity, there are cases of very large discretizations, where performance may be not acceptable.

As for the adaptive action space, the results show that this action embedding was beneficial to the agent's behavior. The increased density of discrete actions gives the agent the precision it needs to follow better any policy and produce better results. Also, due to the more effective use of the action space, a smaller set of discrete actions is required. This can be used to speed up the response of the agent, or to extend the abilities and the problems we can address to it.

Limitations, such as the Critic's saturation effect and the inability to use adaptations on early training stages, have an impact on agent's behavior too. Both make the agent get confused and decrease its performance. Also, results show that the efficiency of the presented method in very small discretizations acts as a threshold value, below which the performance drops as adaptations for these discretization leave large parts of the space empty of discrete actions.

6.1 Future work

For this thesis, we focus more on providing a method for discretizing spaces adaptively, and to show how it can affect agents with discrete actions spaces. Most of the disadvantages of this method have to do with special features of the agent or the domains and are easy to fix. But, we focused more on provide generalized results about this method. Each application of these method will require a set of settings that depend on this exact case.

Future work on a method to decide automatically when to make updates, would be able to detect and avoid any saturation, so the agent can use adaptive action space at its maximum advantages. Also, a more exploratory method to extend the current tree's adaptation procedure would be able to discover expansions and prunnings that our current method misses and would reach closer to the optimal solution with no computational overhead.

References

- ➢ Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. The MIT Press, Cambridge, Massachusetts, 1998.
- ➢ Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 2009.
- ➢ Richard E. Bellman. Dynamic Programming. Princeton University Press, Princeton, New Jersey, 1957.
- ➢ Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. Published in ArXiv: 1512.07679v2, 2015.
- ➢ Lillicrap, Timothy P, Hunt, Jonathan J, Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous Control with Deep Reinforcement Learning. Published in ArXiv:1509.02971, 2015.
- ➢ Muja, Marius and Lowe, David G. Scalable Nearest Neighbor Algorithms for High Dimensional Data. IEEE Transactions on Pattern Analysis and Machine Intelligence, **36**, 2014.
- ➢ Berger, M. J.; Colella, P. Local Adaptive Mesh Refinement for Shock Hydrodynamics. Journal of Computational Physics, **82**: 64–84, 1989.
- ➢ Pazis J., Lagoudakis M.: Binary Action Search for Learning Continuous-Action Control Policies, Proceedings of the 26th International Conference on Machine Learning (ICML), Montreal, Quebec, Canada, pp. 793–800, June 2009.
- ➢ Pazis J., Lagoudakis M.: Reinforcement Learning in Multidimensional Continuous Action Spaces, Proceedings of the 2011 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011), Paris, France, April 2011, pp. 97–104.