Universiteit Leiden

Master Computer Science

Applying Rainbow DQN to FlappyBirds and exploring its transfer sensitivty to PCG parameter changes

Name:	V.Rohit Nekkanti
Student ID:	S2561816
Date:	10/09/2021
Specialisation:	Computer Science: Data Science
1st supervisor:	Aske Plaat
2nd supervisor:	Matthias Müller-Brockhausen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In recent years, deep reinforcement learning is gaining traction due to its effectiveness in manifold of applications. Despite its prospects, deep reinforcement agents are infamous for their long training times. In recent years, there has been extensive research aimed at making deep RL agents train faster by transferring knowledge from pre-trained models. In our work, we investigate the influence of PCG parameters towards transfer sensitivity in deep Q-networks on FlappyBird. Furthermore, we introduced an automated curriculum learning algorithm for our DQN agent to improve sample efficiency and have faster learning dynamics. We found that for positive transfer characteristics in continual transfer learning, the target tasks need to be sufficiently related to each other to improve sample efficiency of the transferred agents. Additionally, using our automated curriculum approach designed for DQN's we emperically show faster learning dynamics on much harder tasks.

Contents

1 Introduction 3
  1.1 Research Question 5

2 Related Work 6
  2.1 Deep Q-learning 6
  2.2 Transfer Reinforcement Learning 7
  2.3 Curriculum Learning 7

3 Methodology 9
  3.1 Reinforcement Learning 9
  3.2 Q-Learning 10
  3.3 Deep Q-learning Networks 11
  3.4 DQN-Extensions 12
    3.4.1 N-step DQN 12
    3.4.2 Double DQN 13
    3.4.3 Noisy Network 13
    3.4.4 Prioritized Replay Bu er 13
    3.4.5 Dueling DQN 14
  3.5 ALP-Curriculum 15

4 Design 19
  4.1 Deep Q-Networks 19
    4.1.1 n-step DQN 20
    4.1.2 Double DQN 20
    4.1.3 Noisy DQN 21
    4.1.4 Prioritized Experience Bu er 21
    4.1.5 Dueling DQN 22
    4.1.6 Rainbow DQN 22
  4.2 ALP-curriculum 22
  4.3 Flappy Bird environment 23
    4.3.1 Preprocessing 24

5 Results 26
  5.1 DQN-variants 26
  5.2 Transfer Learning 28
    5.2.1 Type-1 29
    5.2.2 Type-2 30

5.3 ALP-Curriculum: DQN specific 32
5.4 FlappyBird-RGB 34
5.5 FlappyBird-LIDAR 38

6 Conclusions 42
6.1 Future Work 42

7 Appendix 48
7.1 Experiment - 1 48
7.2 Experiment - 2 51
7.3 Experiment - 3 57
7.4 Experiment - 4 61

Chapter 1

Introduction

Since the inception of deep reinforcement learning, we have seen numerous breakthroughs in the fields of robotics, healthcare, autonomous driving, natural language processing, financial trading and agents playing video games[1–6]. Reinforcement learning has become a widely used framework for tasks involving sequential decision-making. Despite its potential, agents trained using deep reinforcement learning su er when the environment is subjected to continual change [7]. Often times, the training procedure is restarted when the environment changes. Although these RL-agents are shown to outperform human players on Atari benchmark[8] they can't generalize gathered experience towards new tasks as well as humans do. Due to the computational overhead involved in training deep reinforcement agents, it can be ine cient to restart training from scratch.

The challenging aspect for current deep reinforcement learning algorithms is learning a decent initial control policy for vastly complex environments. Instead, the learning procedure experiences frequent and long periods of unstable sub-standard policy [9, 10]. Intuitively, the agent needs to learn the dynamics of the environment from ground zero by continually interacting with the environment and exploiting on the knowledge learnt. As environments are complex and the innate requirement of neural networks needing a lot of training samples makes the learning extremely slow[10, 11]. Transfer learning and curriculum learning have shown great potential in alleviating some on these issues.

In our work we investigate transfer sensitivity of DQN agents to PCG parameters on FlappyBird game. We chose FlappyBird as our testing environment due to its small action space and relative simple implementation. FlappyBird is an arcade game where the players objective is to navigate the bird through pipes as shown in Figure without crashing onto the pipe or running out of the screen. The game di culty is controlled by our PCG parameter *pipe gap*, which denotes the distance the upper and lower pipes are separated. The higher the pipe gap, the easier the game is. Our initial work is to benchmark the performance and training characteristics of Deep *Q*-networks[9] and compare them to the

improved variants proposed in [12].

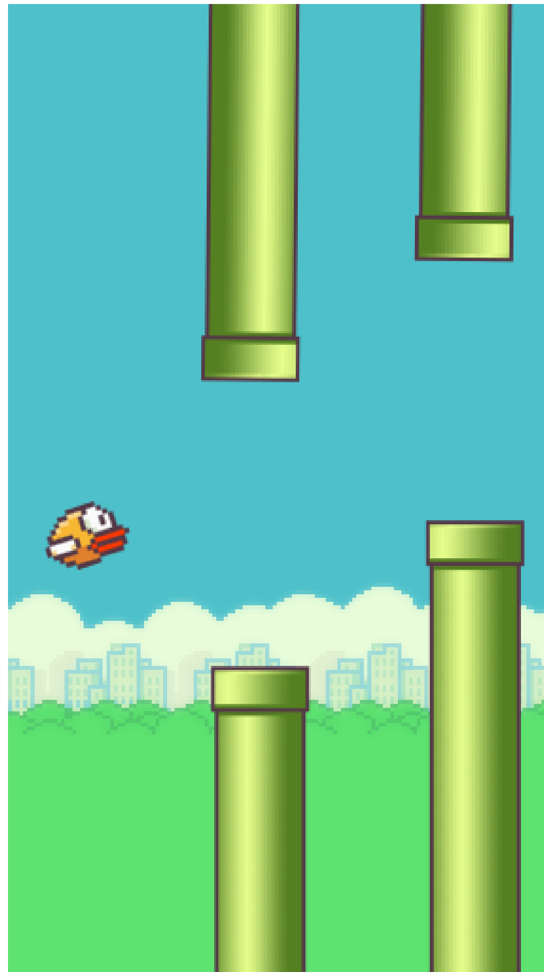

Figure 1.1: An observation from our FlappyBird environment

The novelty our work presents is three folds. First, we present a new FlappyBird environment where the level di culty could be controlled using PCG parameters. Second, we investigate the e ect of transfer learning on DQN agent in two di erent settings. The former case when source task and target task having similar visual distribution and the latter when the source and target task having di ering visual stimuli. Third, we propose a new automated curriculum algorithm on our PCG environment for training DQN agents on hard environments and inspect its learning behaviour.
The report is structured as follows; in Chapter-2 we discuss a brief overview of related work around this thesis. In chapter-3, we introduce common mathematical notion and brief theoretical formulations relevant to our work. In chapter-4 we touch on the design and implementation details of methods discussed in chapter-3. Chapter-5 outlines the results with appropriate discussion and finally in chapter-6 we provide our conclusions explaining our findings while proposing avenues for future work.

1.1 Research Question

Our works aims to investigate whether transfer of pre-trained policy is viable for continual learning in deep-Q networks when the observation space changes while the reward dynamics and action space are unchanged. To formalize, we sought whether

- a positive transfer is achieved on our FlappyBird environment when the observation space of target tasks closely resembles the one of source task.
- a positive transfer is achieved on our FlappyBird environment when the observation space of target task is changed to varying degrees.
- Can the agents performance on hard task be improved by starting training on easier tasks and gradually increasing the task difficulty
- Can we automate the curriculum learning procedure for training DQN agents based on learning progress over a set of tasks with varying difficulty

Chapter 2

Related Work

Our work is centered around Deep *Q*-networks, transfer learning and curriculum learning. This section accounts for current research that influenced our work.

2.1 Deep Q-learning

DeepMind in 2013 introduced a new deep learning model for reinforcement learning and demonstrated its potential to achieve di cult control policies for Atari 2600 video games directly from high-dimensional sensory visual data[9]. [13] discuss why DQN can be overoptimistic for large-scale applications and showed that there model *Double DQN* can reduce the overoptimism problem leading to more stable and reliable learning by finding better policies. [14] introduces a di erent experience replay framework called *Prioritized Experienced Replay* which samples experiences based on computed priorities rather than uniform sampling. Using prioritized experience replay bu er has show to speed up learning two fold on the Atari Benchmark[14]. [15] introduces a novel architecture for DQN that decouples values and advantages all the while sharing same convolutional layers. The dueling architecture has shown significant improvement over in performance compared to the traditional DQN variant. [16] proposes a new exploration scheme in contrast to the -greedy scheme where the network weights are perturbed through an injected noise that is tuned by the algorithm. *NoisyNet* have shown to over-perform over DQN's performance and even achieved super human performance in a select few Atari games. [17] notes on the importance of using value distribution for each action in contrast to selecting actions just based on expected state-action values. Distributional DQN's show that it achieved better learning behaviour using both theoretical and anecdotal evidence by stating the importance of value distribution in better approximating the *Q*-values. [12] integrates six most prominent improvements developed by DeepMind from the time DQN's are first introduced into a single learning algorithm. *Rainbow DQN* achieves better state-of-the-art performance on the Atari benchmark.

2.2 Transfer Reinforcement Learning

Transfer learning in reinforcement learning context has seen major interest lately. The technique in utilizing external expertise from other tasks to benefit learning progress of the target task is studied under di erent transfer frameworks catered towards di erent goals. We discuss the most relevant works and ideas that influenced our work.

Learning from Demonstrations: To speed up the training process, the knowledge transfer is made using external demonstrations. Demonstrations could include hand curated experiences from experts, pre-trained expert policy or a sub-optimal policy. For demonstration based transfer learning the source tasks and targets tasks are the same[18]. [18] notes that providing demonstrations could lead towards better and e cient exploration thus serves as a key method for speeding up the training. For o -policy methods[19, 20] demonstrations are used to train the agents by supervised learning framework on the contrary for on-policy[21] methods, the demonstrations are used in the initial stages to obtain a good starting or a sub-optimal policy that could aid learning through e cient exploration.

Policy Transfer: the knowledge transfer takes the form of pre-trained policies from the source task. Policy distillation is an transfer learning approach that is extended from supervised learning framework[22]. Policy distillation uses a student model that uses learned knowledge from the ensemble of multiple teacher models into a single student model. The student policy is learnt by minimizing the disparity of action distributions between the teacher policy and student policy[23, 24]. Another policy transfer approach denotes the framework that reuses the policy from source tasks. Policy reuse was proposed by [25], that learns expert policies based on the policy learnt using the source tasks.

2.3 Curriculum Learning

We humans learn and master various skills through compounding our experiences from easy to hard tasks designed as curriculum through rigorous practice and evaluations. Recent work in reinforcement learning has leveraged these ideas of developing curricula for deep reinforcement learning agents to improve their performance and accelerate learning on a di cult set of tasks. Most recent works have been studying on generating curriculum automatically to train reinforcement learning agents. For the sake of brevity, we discuss the most influential ideas that inspired our work. For a thorough account on current state-of-the-art developments, we refer the reader towards [26–28].

Improving the sampling e ciency is one of the core mechanisms of automated curriculum learning. [14] has developed a framework that prioritizes experiences to sample more frequently from replay bu er for learn more e ciently. [29] expands on the prioritized replay

framework for actor-critic architectures. [30] discusses on improving sampling e ciency in policy gradient methods by discarding non-informative transitions from the gradient update step.

Automated curriculum learning has shown major strides in solving hard tasks that could not be solved directly due to sparse reward dynamics. [31] proposes a novel method to develop robust policies using ACL to improve task generalization and reduce policy brittleness. The task selection is carried out using what they term as "Reward guided curriculum" that selects tasks that yield highest learning rewards using EXP3.S algorithm developed towards adversarial bandits theory. Our work in creating curriculum learning framework for PCG-controlled environment is influenced by the works in [32, 33]. [32] discuss a method which samples parameters that control stocastic procedural generation of environments for better task generalization and faster learning progress. [33] emphasizes on active task selection strategies to uniform random sampling for achieving agents that could perform better on various environment settings thus achieving a small degree of generalization.

Chapter 3

Methodology

Reinforcement learning has seen significant progress over the last decade which laid the groundwork in developing A.I agents that could defeat the world Go champion [6], crushing human set records on Atari video-games [8] and more impressively winning against seasoned veterans in real time strategy video-games like Dota[7] and StarCraft [34]. This chapter marks on the quintessential components of deep reinforcement learning algorithms used in this paper and discusses them briefly. For this paper, we emphasize on deep Q-learning networks by [8], a handful of improvements proposed in [12] to address the limitations with vanilla DQN's. For a thorough account in the field of reinforcement learning we refer the reader to [10, 35]

3.1 Reinforcement Learning

Reinforcement learning is formalized following the conventions of Markov Decision Processes [10]. The primary constituents of such conventions include a state space $\mathcal{S}$, which represents the various states in the environment; an action space $\mathcal{A}$, which is a set of legal actions that could be performed on each state of the environment; a control policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$; a reward that is obtained when acting upon a state of the environment r; and a transition function $\Pr(S_{t+1}, R_{t+1} \mid S_t, A_t)$ which defines the dynamics of the environment. Reinforcement learning aims at learning an optimal policy π^*, that maps observed states to actions that results towards a maximum reward. To achieve the optimal policy, the agent should act on the environment, collect experiences and tune its control policy based on the encountered experiences. These experiences are represented as a quadruple (s, a, r, s') where s is the observed state, a is the action taken when presented with state s, r is the obtained reward for performing the action a on state s and finally s' is the observed next state.
For finite sequence episodic task, the cumulative reward obtained by the agent is defined as [10],

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T \tag{3.1}$$

However, to assign higher relative importance towards immediate rewards, a discount factor $\gamma \in [0, 1]$ is used in equation 3.1 as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T, \tag{3.2}$$

It is paramount to note that the discounted return shown in equation 3.2 is accumulated by an agent following the policy π. Accounting for stochastic dynamics, we aim towards maximising the expected cumulative reward instead of deterministic reward show in equation 3.2. The expected cumulative reward obtained by the agent following a policy π on the observed state s is denoted by a value function [10]

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{3.3}$$

$$= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_\pi(s')], \tag{3.4}$$

, where $\pi(a|s)$ indicated the probability that action a is chosen under policy π when the observed state is s and $p(s', r|s, a)$ is the dynamic function of the environment. Equation 3.3 is referred to as the the Bellman equation which characterizes the relationship of a state to its successor states.
To convert the optimal policy to actions, a notion of state-action value function $q_\pi(s, a)$ is defined in [10]. In essence, this state-action value function is a translative look-ahead for the value-function v_π in equation 3.3 and defined as

$$q(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{3.5}$$

$$= \sum_{s', r} p(s', r \mid s, a)[r + \gamma \max_{a'} q(s', a')], \tag{3.6}$$

.

3.2 Q-Learning

In the previous section 3.1, both the state and state-action value functions are theoretical notions of defining the optimality of a particular control policy. However, in practive these functions are ofter unknown and should be estimated appropriately. These function estimations are indicated as $V_\pi(s)$ and $Q_\pi(s, a)$ for the state and state-action value functions, respectively.
A naive-algorithm for estimating such *optimal* state-action value functions is called *Q*-learning. The *Q*-learning algorithm employs off-policy TD-learning[10]. The algorithm starts by assigning random values to the state-value function for each observed state-action pair. Now, Based on experience tuples from the environment $e_t = (s_t, a_t, r_t, s_{t+1})$ sampled

using the random policy, it updates the current estimate as follows,

$$Q(s_t; a_t) = Q(s_t; a_t) + \alpha[R_t + \gamma \max_{a_{t+1}} Q(s_{t+1}; a_{t+1}) - Q(s_t; a_t)]. \quad (3.7)$$

Since the experiences obtained are independent of the current policy this approach is categorized as an off-policy algorithm.
The Q-learning algorithm is proved to converge to the optimal state action value function when the amount of iteration grows towards infinity[10]. However, this is not practical especially when the environment's state-space or the action-space is vastly complex. Such limitations call for methods that could use parameterized function approximators which are recently referred to as neural networks.

3.3 Deep Q-learning Networks

Owing to recent breakthroughs in the field of deep-learning [36], deep neural networks have become particularly favourable for reinforcement learning applications. Using neural networks as a nonlinear function that maps state representations to actions, we could modify the Q-learning algorithm discussed in section-3.2. However, its not as straight forward as it seems and suffers learning instabilities. Such limitations of using neural networks are outlined by the term 'deadly-triad', which states that learning process can become unstable or even diverge when bootstrapping, off-policy learning, and function approximation are used[10]. However, DeepMind came up with an algorithm *Deep Q-Learning Network(DQN)*[8], that showed stability in learning despite violating the deadly triad's principles. The DQN algorithm is denoted as follows.

Algorithm 1 Deep Q-learning with Experience Replay[9]

```
Initialize replay memory D to capacity N;
Initialize action-value function Q with random weights
for episode = 1 ... M do
    Initialise sequence s_1 = {x_1} and preprocessed sequenced φ_1 = φ(s_1);
    for t = 1 ... T do
        With probability ε select a random action a_t
        otherwise select a_t = max_a Q'(φ(s_t), a : θ)
        Execute action a_t, get reward r_t and preprocessed next state φ(s_{t+1})
        Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
        Sample random mini-batch of transitions (φ_j, a_j, r_j, φ_{j+1}) from D
        y_i = r_j + γ max_{a'} Q(φ_{j+1}, a' : θ)
        if φ_{j+1} is terminal then
            y_i = r_j
        end if
        perform a gradient step on (y_j − Q(φ_j, a_j; θ))^2
    end for
end for
```

DQN extended the naive Q learning algorithm by addressing three vital issues that affect training stability. The first of which is introducing *epsilon-greedy policy* to interact

with the environment to sample experiences. The epsilon-greedy policy is hypothesized to strike a balance to the exploration-exploitation trade-off towards learning optimal control policies[35]. As Q-learning mimics the abstraction of supervised learning, there exist a fundamental requirement that training data(i.e., experience tuples) obtained should be independent and identically distributed(i.i.d). Since, the sampled tuples are temporally correlated the use of large *replay buffer* to store transitions and sample batches randomly to train the network was the second proposed solution. And, the third issue is when we bootstrapping the state-action values $Q(s_t, a_t)$ and $Q(s_{t+1}, a_{t+1})$ from the bellman equation in 3.7. Since states s_t and s_{t+1} are subsequent to each other, bootstrapping and updating the network would change the networks standing on s_{t+1} making the training unstable. [8] gets around this issue using a *target network*, that is the copy of our original network but updated periodically after a set of iterations.

3.4 DQN-Extensions

The DQN paper was published in 2013, and over the years there were genuine criticisms' regarding the approach that perpetuated towards making improvements. DeepMind have published another paper *Rainbow DQN* [12] incorporating six enhancements to their original algorithm. In this section, we discuss five of them that are relevant to our paper's scope.

3.4.1 N-step DQN

N-step DQN is an improvement that was taken from an old work of *Richard Sutton* in 1998[37]. In section 3.2, equation 3.7 defines the bellman update for our DQN method. Due to its recursive formulation one can express $Q(s_{t+1}, a_{t+1})$ in terms of itself as

$$Q(s_t, a_t) = R_t + \max_{a_{t+1}}[r_{a+1,t+1} + \max_{a_{t+2}} Q(s_{t+2}, a_{t+2})]^1 \tag{3.8}$$

Under the assumption that actions are chosen under optimal policy, equation 3.8 could be phrased as

$$Q(s_t, a_t) = R_t + r_{t+1} + {}^2\max_{a_{t+2}} Q(s_{t+2}, a_{t+2}). \tag{3.9}$$

Following the same idea, we could further unroll the Q-function for preceding states any desired number of times. Unrolling the action-value function has show faster training times, as better estimates for Q-value are propagated n-steps faster in contrast to each step for vanilla DQN[38]. The previous statement holds true only for small values for n, because equation 3.9 is bounded by the assumption that all chosen action are optimal; however, in reality we aim to find optimal policy leading to such actions.

[1] For the sake of simplicity the parameter in equation 3.7 is assumed to be equal to 1.

Furthermore, n-step DQN transforms our traditional vanilla DQN from o -policy learning algorithm to on-policy method as actions are chosen based on the current policy. [38] notes that using large replay bu ers will hinder the training as the likelihood of sampling experiences from bad old policy is high, thus leading to bad updates in our Q-network. For a much theoretical discussion on n-step approaches for Q-learning we refer the reader to [37, 38].

3.4.2 Double DQN

The next improvement to traditional DQN has been proposed by DeepMind in 2015 named Double Q-Learning. DQN overestimates the values for Q in the bellman update in equation 3.7 that could result towards sub-optimal policies [13]. For an elaborate proof for the reason of cause we refer the reader to [13]. To tackle this problem of overestimation, a solution to modify the bellman's update is proposed as

$$Q(s_t, a_t) = R_t + \max_{a_{t+1}} Q'(s_{t+1}, \text{argmax}_{a_{t+1}} Q(s_{t+1}, a_{t+1})). \tag{3.10}$$

Where, the action is selected based on the Q-network rather than the target network Q' discussed previously in section 3.3.

3.4.3 Noisy Network

Traditional variant of DQN discussed in section 3.3 explores the environment choosing the -greedy policy, i.e., the agent takes a random action with the probability otherwise chooses action based on higher action-value based on the network outputs. [9] suggests defining an initial to 1 and slowly annealing over time, thus resulting in extra hyperparameter that has to be tuned for each specific environment. The -greedy policy explores the environment until a period of time and then chooses actions based on current policy once the saturates which might lead to a sub-optimal policy that is stuck in the local minima.

In 2017 DeepMind proposed a new framework *Noisy Networks* that explores the environment during training instead of having a separate exploration schedule[16]. The proposed solution is to perturb the weights of fully connected layers in our network and adjust the level perturbation during training based on the loss function i.e., TD-error. For a complete theoretical discussion we refer the reader to [16].

3.4.4 Prioritized Replay Bu er

The DQN algorithm outlined in Algorithm 1 uses a replay bu er to sample experiences and break correlations between immediate experience transitions. Sampling experiences

uniformly from the replay bu er to train on the network is crucial to adhere to i.i.d. property for training networks using gradient descent approach. However, the uniform sampling approach from the replay bu er had been questioned, as there may be a chance the network fails to see experiences that are important and could be thrown away when the replay bu er updates. To address such shortfall, DeepMind proposed a replay bu er framework that could prioritize samples based on TD-error or training loss[14].

Assigning priorities to experiences based on the TD-error and sampling experiences proportional to the priorities had shown positive learning characteristics in DQN's[14]. However, when we embed priorities to experiences and sample the replay bu er based on the defined priorities we may loose the i.i.d. property and overfit on the sampled subset as we impute bias in our selection. To curb such issues [14] defines the priority of each experience in the bu er as

$$P(i) = \frac{p_i}{\sum_k p_k}; \tag{3.11}$$

where p_i is the priority for the $i^t h$ experience in our bu er and $\in [0; 1]$ is the parameter that controls the emphasis given to priorities. A value of $= 0$, makes the sampling uniform and higher values impose higher emphasis on the computed priorities.
[14] proposes two variants in computing the priorities. The most practiced one is where the relationship between the batch weights and individual experience in the batch is defined as

$$w_i = \left(\frac{1}{N \quad P(i)}\right) \tag{3.12}$$

where $\in [0; 1]$ is a hyperparameter that accounts for bias imputed due to prioritized sampling. For $= 1$, the imputed bias is compensated by making the all the priorities same as in the case of uniform sampling. For better learning characteristics [14] suggests gradually increasing from 0 to 1.

3.4.5 Dueling DQN

The final improvement we discuss over vanilla DQN's is *Dueling Network Architecture* proposed by [15]. The prominent feature of the dueling architecture is that the Q-values our network approximates is divided into two quantities:

- value of the state $V(s)$ and,
- advantage of actions for the observed state, $A(s)$.

Using theoretical notion the division of Q-values could be shown from by modifying equation 3.7 as

$$Q(s_t; a_t) = V(s_t) + A(s_t; a_t) \tag{3.13}$$

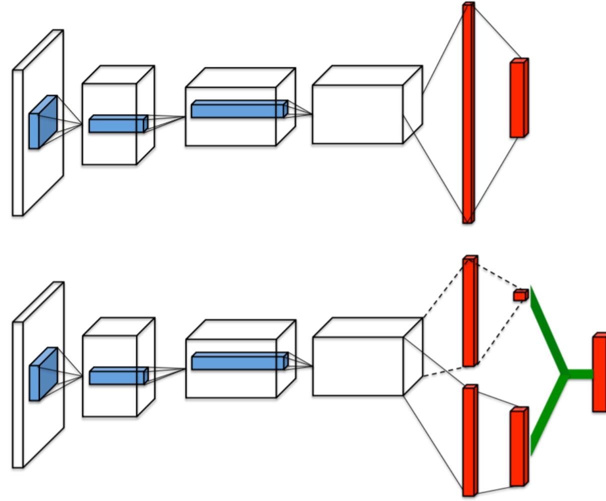

Figure 3.1: A single stream Q-network(top) and dueling Q-network(bottom). The dueling network has two streams to estimate state-value and the advantages for each actions; the green output module implements equation 3.14 to aggregate the outputs. Both networks output Q-values for each action. Figure taken from [15]

where, advantage $A(s_t, a_t)$ can be interpreted as the extra improvement in reward from a particular action on the observed state. Intuitively, using dueling architecture, the agent could differentiate between states without having to learn the effect of each action on the observed state[15]. Fig 3.1 shows the networks architecture for dueling networks that showed better training stability, convergence and higher rewards on the Atari benchmark. The convolution features are shared between the value network and advantage network followed by aggregating the outputs as

$$Q(s, a) = V(s) + [A(s) - \frac{1}{|A|}\sum_{a'} A(s, a')] \tag{3.14}$$

where, the output values from the advantage network are mean centered at 0.

3.5 ALP-Curriculum

Current state-of-the-art reinforcement learning algorithms fail to yield good performance on tasks with sparse rewards. For such tasks the likelihood for an agent in achieving a decent control policy is dependent on the quality of exploration. Curriculum learning has shown improvement in learning characteristics by aiding better exploration [39] for harder tasks by acquiring experience from simpler tasks.

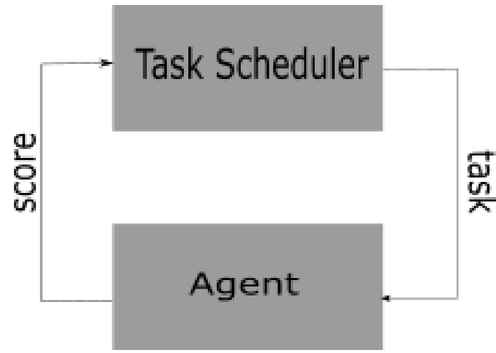

Figure 3.2: A generic representation of our automated curriculum learning framework

To automate the task selection we adhere to the framework shown in Figure 3.2. At each learning step, the task scheduler chooses a task for the agent to train on. The agent trains on the chosen task and returns back a score that is indicative of its learning progress. Based on the agents score, the task scheduler should decide on which task to select, such that the agents learning progress doesn't degrade.
Selecting a task for an agent to train on is dependent on the score obtained after the training interval. The algorithms in the following sub-section 3.5 outline the task selection strategies employed for curriculum learning.

Algorithms

Our approach of task selection is based on the bandit algorithm in [10] where each task is aliased as a bandit arm. We use an exponentially weighted average to update the expected return Q from different tasks as shown in equation 3.15:

$$Q_{t+1}(a_t) = \alpha r_t + (1 - \alpha) Q_t(a_t) \tag{3.15}$$

where α is the learning rate and r_t is the obtained reward. The next task is chosen using an ϵ-greedy exploration. Our algorithms follow the same framework while the major difference lies in how the reward r_t is calculated.

Algorithm 2 Online Curriculum

```
Initialize DQN-Agent
Initialize replay memory D to capacity N;
Initialize expected return Q(a) = 0 for all K tasks
for t = 1 ... T do
    choose task a_t based on |Q| using ε-greedy policy;
    fill the replay buffer D with experiences from task a_t
    Train the DQN agent and observe reward r_t = x_t^(a_t) − x_t'^(a_t)
    Update expected return Q(a_t) = αr_t + (1 − α)Q(a_t)
end for
```

The first steps in our algorithms is to initialize our DQN agent, the replay buffer and assign the expected return $Q(a)$ as 0. We use an ϵ-greedy approach to choose an action a_t that represents the task to train on. Following the task selection, our DQN agent uses its

current policy to sample experiences from the chosen task and stores them in replay buffer for use during the agents learning stage. We note that the replay buffer doesn't discard its previous experience samples from the preceding task making this approach limited to DQN agents. However with slight modification, we outline the current approaches in the Appendix 7.4 to be compatible with other reinforcement learning algorithms.
Our first version of algorithm named *Online* curriculum is outlined in 2.
Since task selection is reliant on the score r_t we discuss three approaches to define and evaluate these scores. Our first approach outlined in 2 computes the difference in episodic rewards before and after training the agent on the chosen task. This approach is simple and intuitive as we want our task scheduler to choose tasks that show greater learning progress. Furthermore, to avoid task forgetting i.e., performance degradation on other task the task scheduler makes the decision based on the absolute expected returns $|Q(a)|$. Choosing the task by taking the absolute expected return forces the task scheduler to pick up tasks that show a greater degree of forgetting.

Algorithm 3 Repeat Curriculum

Initialize DQN-Agent
Initialize replay memory D to capacity N;
Initialize expected return $Q(a) = 0$ for all K tasks
for t = 1 ... T **do**
  choose task a_t based on $|Q|$ using ϵ-greedy policy;
  Reset $W = \emptyset$
  fill the replay buffer D with experiences from task a_t
  for j = 1 ... J **do**
    Train DQN agent and observe score $o_t = x_t^{(a_t)}$
    Store score o_t in list W
  end for
  Apply linear regression on the scores in W and extract slope as r_t
  Update expected return $Q(a_t) = \alpha r_t + (1 - \alpha) Q(a_t)$
end for

The observed score should try to estimate the learning progress of the agent. Our second approach tries to better estimate the learning progress on each task by training on the task multiple times. This can be seen as practicing a task several times before moving on to the next one. 3 describes the algorithm for our second approach where the scores are calculated by computing the slope on all the obtained scores after each training sub-intervals. A higher slope means greater learning progress, a negative slope represents the case of unlearning and a slope of zero denotes no progress made through training.
Running the training sequence multiple times for a specific task could be expensive when the performance on that specific task starts to saturate. To counter this, we employ a *first in first out(FIFO)* buffer that keeps track of the previous K scores along with its time-steps in our final algorithm outlined in 4.

Algorithm 4 Window Curriculum

Initialize DQN-Agent
Initialize replay memory D to capacity N;
Initialize expected return $Q(a) = 0$ for all K tasks
initialize FIFO bu ers $W(a)$ with length L
for t = 1 ... T **do**
  choose task a_t based on jQj using ϵ-greedy policy;
  fill the replay bu er D with experiences from task a_t
  Train the DQN agent and observe score $o_t = x_t^{(a_t)}$
  Store score o_t in $W(a_t)$
  Apply linear regression on the scores in $W(a_t)$ and extract slope as r_t
  Update expected return $Q(a_t) = \alpha r_t + (1 \quad \alpha)Q(a_t)$
end for

Having a stand-alone FIFO bu er for each task doesn't waste extra computational resources on tasks that show no progress. The scores are evaluated by the computing the slopes on each of the bu er assigned to the tasks.

Chapter 4

Design

This section details on our implementation details and design choices for Deep Q-Network, its variants and curriculum learning algorithms discussed in Chapter 3. In addition, we discuss the environment design and detail on the preprocessing methods used in improving the training speed. Our implementation throughout uses deep-learning framework PyTorch[40] for architecture design and uses a high level library PyTorch-Ignite[41] for training and evaluating our networks.

4.1 Deep Q-Networks

Model Architecture: The deep Q-network architecture for our implementation follows closely to what has been proposed in [9]. Our model has three convolution layers followed by fully connected layer separated by rectified linear units(ReLU) for non-linear activations. The first convolution layer with a kernel size of 8 spanning over the observation space with a stride of 4 with 32 convolved channels. Following, the first layer a second convolution layer with 64 convolved channels of kernel size 3 spanning with stride of 2. And, the final layer with 64 convolved filters with kernel size 3 that span with a stride of 1. The output of the final convolution layer is flattened and a hidden layer of 512 units are connected to the 2 units(alias for 2 actions) that output the state-action values.

Replay Buffer: A replay buffer is a storage mechanism that stores experiences in the fixed length buffer and has to omit previous observations when adding new experiences while the buffer is at its capacity. For our implementation we use the *Deque* container of fixed capacity from python's *collections* library that stores experiences. The experiences stored in our *Deque* container are *namedtuple* data-structure from python's collections library that house the information of state, action, reward, done-flags and new states.

Training Hyperparameters: Our training workflow follows the same one described in Algorithm 1 using *Adam* optimizer to update the network parameters after every

Parameter Name	Value	Description
Replay Bu er Size	100,000	Number of experience to keep track to sample and learn from later in
Warm start	10,000	Number of random samples to gather before starting the learning procedure
Exploration Schedule	100,000	Linear annealing schedule for ϵ-greedy action selection
Exploration start	1	Probability of choosing a random action at the start of the schedule
Exploration Min.	0.02	Probability of choosing a random action at the end of the schedule
Batch size	32	Number of experience samples to be sampled from replay bu er to train the network on
Target update frequency	1000	Number of training iteration before copying the network to our target network
Optimizer learning rate	0.0001	Learning rate for Adam optimizer
Discount Factor	0.99	Horizon discounting factor for TD-error

Table 4.1: Hyperparameters for the DQN training procedure

training step. Table 4.1 details on the choice for our selected parameters. In contrast to the parameters specified in [9] we have been judicious in selecting the size of replay bu er owing to hardware constraints available at our disposal.

4.1.1 n-step DQN

Extending our DQN model towards n-step DQN is fairly straight forward. The most important change to our implementation would lie on how experiences are stored. As mentioned in Section 3.4.1 a high value of n is not preferred and based on [37]'s suggestion we chose a value of $n = 4$ and unroll our TD-update equation 3.7 by 4 steps. This results in changing our experience tuple slightly, where, the next state would be chosen after taking n-actions based on current policy and the reward would be the total accumulated reward obtained from n-steps. However, when we encounter an end state we should we stop rolling further and update the experience tuple until the current step. Another implementation feature one might overlook is the discount factor as after n-steps, the discount factor on the n-th state would be ^{n}. For our experiments using n-step DQN we use the same hyperparameters as shown in table 4.1 while keeping number of steps $n = 4$.

4.1.2 Double DQN

The core di erence between Double DQN and vanilla DQN from implementation point of view lies in the loss function passed to PyTorch optimizer. As discussed in section 3.4.2, we calculate the best action on subsequent state using our main network while the Q-values corresponding to these actions are computed through the target network. For our experiments using Double DQN we use the same hyperparameters listed in table 4.1

4.1.3 Noisy DQN

In regards to Noisy network implementation the key changes lie in the network architecture of vanilla DQN where the final fully-connected layers are replaced with *Noisy* layers that are implemented using core PyTorch[40] functionality. To perturb the network weights in our fully-connected layers in vanilla DQN, we sample random values from normal distribution with mean $= 0$ and standard deviation $= 0.017$ as suggested in [16]. Additionally we want to wrap the standard deviation as a trainable parameter that decreases based on the loss function. To make a trainable parameter, we use PyTorch functionality *nn.Parameter* to define it as a trainable parameter that can be updated by the optimizer after every training step. Furthermore, the authors from [16] suggest toward initializing the network weights using a uniform distribution with mean $= 0$ and standard deviation

$$= \sqrt{\frac{3}{N_{inputfeatures}}} \tag{4.1}$$

Finally, during the networks forward pass we sample random noise based on the defined parameters and perform linear transformation of input data as in the case of feed-forward network. For our Noisy network, we stick to the same set of hyperparameters from table 4.1 where the exploration parameter epsilon is redundant.

4.1.4 Prioritized Experience Buffer

Replacing our replay buffer from our DQN's implementation with Prioritized Experience buffer might seem like a straight forward task. However, there exist stark changes to that of our replay buffer functionality. We use the same data structure to that of replay buffer discussed previously with an additional variable "*weights*" assigned to each experience tuple. Sampling from the traditional replay buffer has complexity of $O(1)$ whereas the complexity of sampling of a prioritized replay buffer would be high. The framework for prioritized experience buffer used in [14] is defined to be $O(logN)$. However, due to the lack of any open-source implementations, we used the naive version of prioritized replay buffer that has a complexity of $O(N)$. Our implementation uses the hyperparameter that accounts for the bias imputed due to prioritized sampling as described in [14] experiments on Atari benchmarks. For our experiments we start with $= 0.4$ and incrementally increase it towards 1 over the span of $100,000$ iterations. While populating the replay buffer with a new experience we assign the maximum value of priority of existing experiences in the buffer to the new experience. But, when the buffer is empty, the newly added experience get a priority of 1. While sampling from the buffer the priorities are converted into probabilities and experiences are sampled based on these probabilities using NumPy's random sampling methods. For the task of updating the priorities of each experience in the replay buffer we get the losses from our loss function and perform the update step as shown in equation 3.12.

4.1.5 Dueling DQN

The only change compared to the vanilla DQN's implementation lies in the architecture as discussed in section 3.4.5. The only change to the architecture is having two separated fully connected layer to estimate the state-value and action-advantage which are cascaded as shown in figure 3.1

4.1.6 Rainbow DQN

To mark all the improvements discussed over vanilla-DQN, we combine everything in an orderly fashion. We replace the replay bu er with prioritized replay bu er, the experience tuples are structured based on n-step rolling, we change the loss function to account for overestimation, use the dueling architecture for our model with noisy fully-connected layers. We trained the combined-DQN using the same set of hyperparameters from table 4.1.

4.2 ALP-curriculum

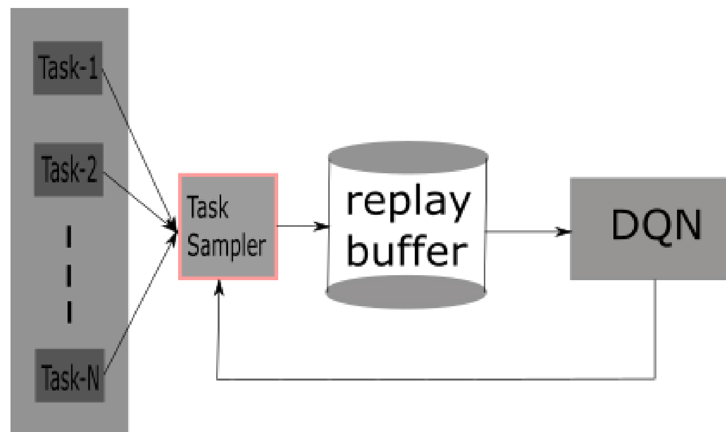

Figure 4.1: Automated curriculum framework for DQN. The task sampler chooses a task from the task pool and add experiences specific to the task into the replay bu er for the agent to sample and train on.

From our discussion of Automated Learning-Progress(ALP) based curriculum in section 3.5 we detail on the design and implementation of the same in the current section. Figure 4.1 expands on our discussed framework of curriculum learning as shown in figure 3.2. We note that the design showed in Figure 3.2 is fused into the DQN architecture by using a common replay bu er to collect experiences from di erent tasks by the task controller. For the model-independent variation of our approach the key change would be to make the replay bu er independent of the task scheduler by re-initializing it every curriculum time-step. We compare our approaches to hand-crafted curriculum we refer to as *oracle* and *uniform* curriculum that samples tasks randomly with equal probability from the task pool regardless the score. Our experiments on curriculum strategies are done on a total of

500,000 training iterations with new task selected after 10,000 iterations. For a detailed view of chosen hyperparameters for curriculum algorithms refer to table 4.2

Parameter Name	Value	Description
Exploration Schedule	20	Linear annealing schedule for ϵ-greedy action selection
Exploration Start	0.6	Probability of choosing a random action at the start of the schedule
Exploration Min.	0.05	Probability of choosing a random action at the end of the schedule
Learning Rate (α)	1	Blending factor
Run iterations	500,000	Total number of training iterations
Task change frequency	10,000	Frequency at with the curriculum task selector evaluates and samples new tasks
sub-task frequency (Naive)	1000	Frequency of stored episodic rewards during the task training period

Table 4.2: Hyperparameter configuration for our curriculum learning task scheduler

4.3 Flappy Bird environment

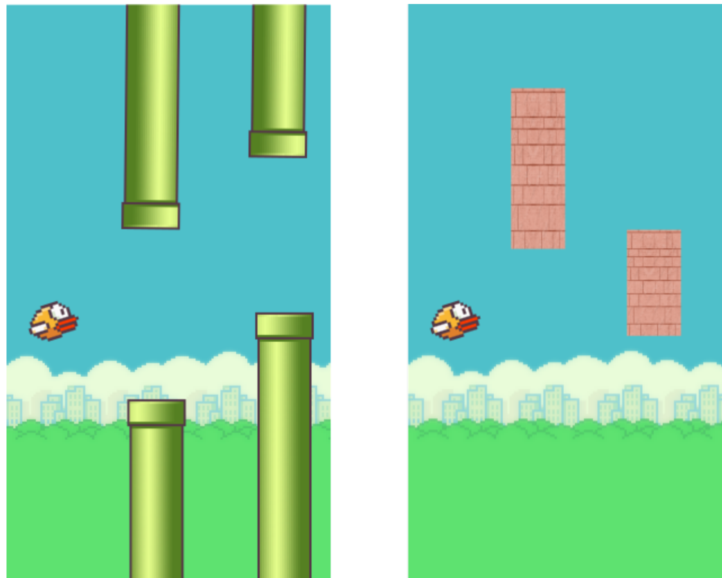

Figure 4.2: (left) denotes the observation of our FlappyBird environment with the pipe-gap of 200 pixels and (right) denotes the observations when the probability of walls is set to 100%

Our implementation in designing the flappy bird environment involves use of PyGame[1] module for writing the game logic and rendering the environment based on player inputs. We closely base our game logic from an existing GitHub repository[2]. The action space in both our implementations is discrete with two action queries namely *flap* and *no flap*. The di erence however lies in the way the actions is registered. In our implementation, we code the action *flap* to *1* and action *no flap* to *0* while the previous uses a mouse click to register the *flap* action. In addition, we wrapped the game logic to the OpenAI gym[42] environment framework for better accessibility to train and test various RL algorithms and methods.

[1] https://www.pygame.org/
[2] https://github.com/sourabhv/FlapPyBird

The game is designed to run at a resolution of 288×512 pixels, with pipes spawning with random placement every 200 pixels. At a given any time frame, the player could see at-most 3 pipes. For every step the player takes without crashing, we reward the player with a reward of 0.1, for every pipe crossed a reward of $+1$ and a reward of -1 when crashed are awarded. We have included a parameter *pipe gap* to our implementation, through which the user has control over the separation between the pipe on top and bottom.

The novelty in our implementation is addition of a different surface we call *wall* into the game to test our agents learning behaviour when the training environment changes. We include the parameter *wall probability* that spawns a walls of varying heights in-place of pipe with the specified probability. Figure 4.2 show the environment observation when we choose the wall probability to be 1. The generated wall and pipes have the same width but differ in vertical placement. The obstacles are separated at a fixed distance for the game to be deemed solvable. The reward structure is kept the same irrespective whether the obstacle is a wall or a pipe. Having the reward structure intact we could inspect the effects of agents learning behaviour while the visual stimuli changes.

FlappyBird-LIDAR:

To further simplify our environment, we change our observation space from a tensor of stacked RGB observations to a numerical vector representing

- Distance to pipe,
- Distance to ground, and
- Velocity of the bird

for further testing our curriculum strategies on environment with different observation characteristics.

4.3.1 Preprocessing

As discussed in the previous section, each observation is an rgb-image of dimensions $(288, 521, 3)$. Using CNN's in our network architecture the agent could look into local spatial features that could aid its decision making[43]. DQN's need a lot of time to train to arrive at an optimal policy. To accelerate learning, we apply several transformations on our observation space. Since we already discussed that our environment is designed to resemble OpenAI-gym's environment functionality we frame our transformations as OpenAI-gym wrappers over both environments observation space and action space. All the transformation we implemented are motivated from [8] on training on Atari benchmarks

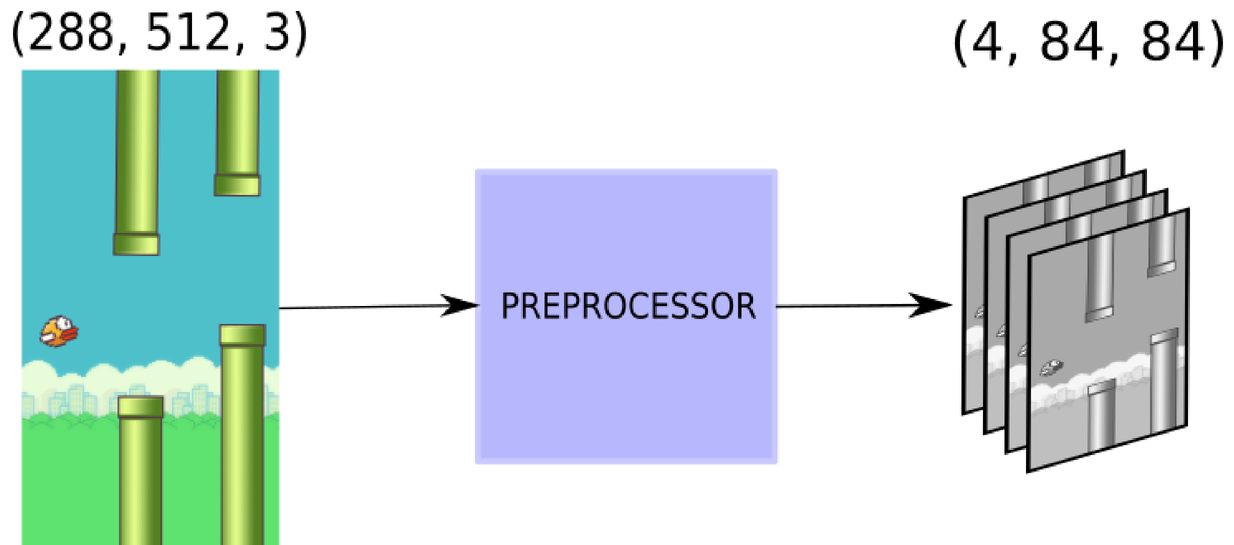

Figure 4.3: Preproccesing function(ϕ) output: The input image is down-scaled, converted to gray-scale, and stacked with 4 preceding observations

To increase the training speed of our DQN agent, the action taken by the agent is repeated for the subsequent K-frames. We choose $k = 4$ as done by the authors in [8]. Repeating the action on k-frames could lead to significant training speeds as the we cut down on processing every frame with our network.

Furthermore, we convert our observation space from RGB-colour space to grayscale followed by scaling our images to a resolution of $(84; 84)$ and normalizing them. As the network can't get the dynamics of the game state, [9] suggests to stack several preceding frames together as the network could draw information about the game dynamics. In our implementation we stack preceding 4 frames to the our observation space. And, finally we clip the rewards using signum function. Figure 4.3 shows the transformation of our observation space as discussed here

Chapter 5

Results

5.1 DQN-variants

Our first experiments entail inspection of various DQN improvements proposed in [12]. All our experiments are repeated three times with di erent seed values to validate that the results we get aren't by mere chance. We compare each of our improvement discussed in chapter 3 to the traditional variant of DQN in [9].

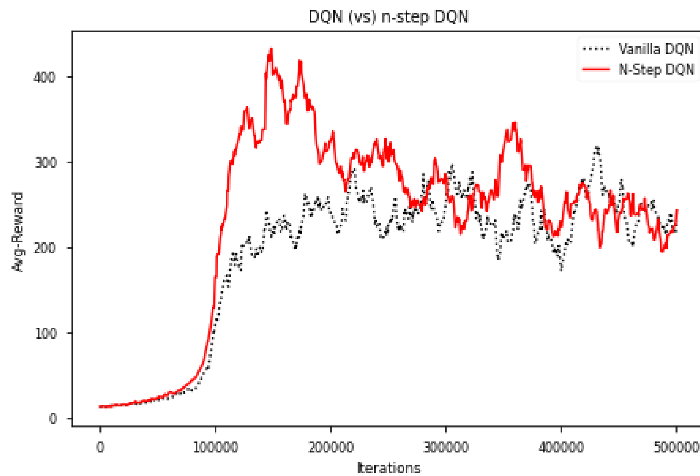

Figure 5.1: n-step DQN learning characteristics compared to vanilla DQN

Our first experiments test the e ectiveness of n-step DQN [38] to the vanilla DQN. Figure 5.1 shows the learning behaviour of both our agents trained using DQN and n-step DQN. We observe a clear faster learning behaviour by unrolling our TD-equation in 3.7 by 4 to have faster convergence to the reward achieved by vanilla DQN.

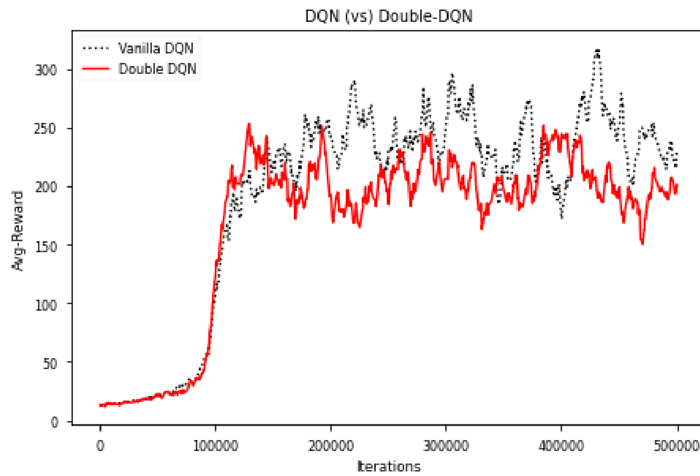

Figure 5.2: Double-DQN learning characteristics compared to vanilla DQN

Next, we compare the learning dynamics of double DQN to vanilla DQN. From figure 3.4.2 we observe the reward dynamics of double DQN is almost as equal compared to vanilla DQN. We haven't observed any significant better reward behaviour in comparison to vanilla DQN as discussed in [13].

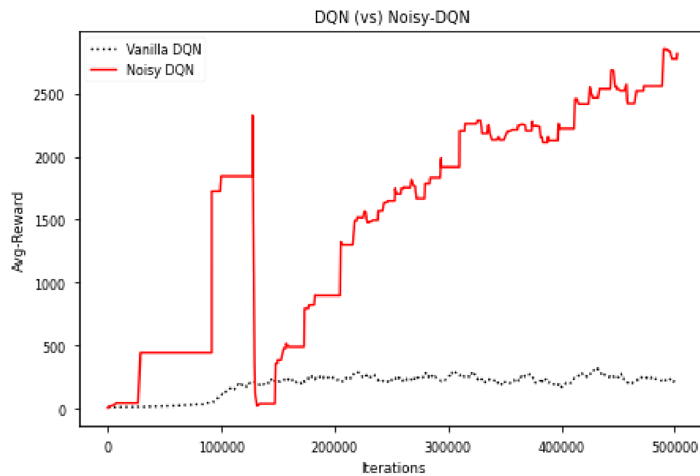

Figure 5.3: Noisy-DQN learning characteristics compared to vanilla DQN

Replacing the vanilla DQN's exploration behaviour with a noisy fully connected layer shows much better training dynamics. NoisyDQN's were able to reach the vanilla DQN's behaviour in less than $100k$ iterations as shown in figure 5.3. Furthermore we can see that the signal to noise ratio in the output layer of our network increases up until $100k$ iterations and decreases gradually after. This explains the agents behaviour as it starts reaching high scores gradually beyond this point where the SNR starts to decrease.

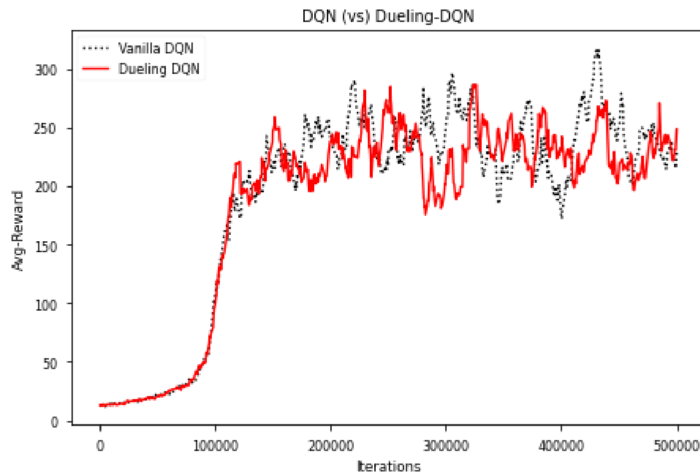

Figure 5.4: Dueling-DQN learning characteristics compared to vanilla DQN

Double DQN architecture separates the state-values and state-dependent-actions(Advantage) into two di erent streams. From the learning dynamics in figure 5.4 of double DQN to vanilla DQN for FlappyBird we observe almost similar behaviour. We expect such behaviour as the agent has to take successive actions to evade obstacles continuously which could be explained by the fluctuating values of the advantage network in

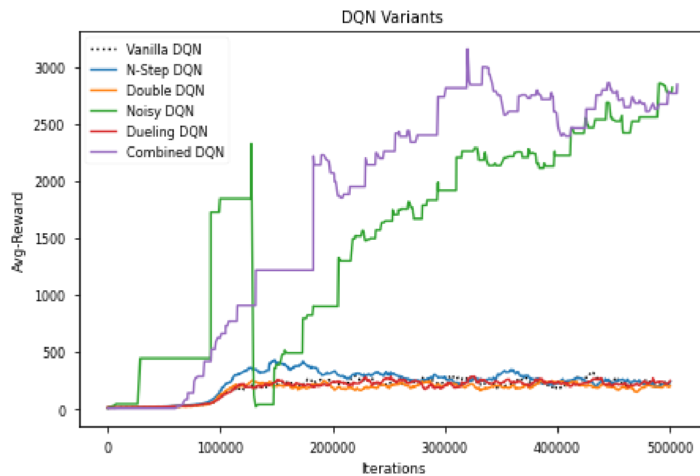

Figure 5.5: Rainbow DQN learning characteristics compared to all the DQN enhancements

Finally, we combine all the improvements and as discussed in section 4 and compare it against all models. Figure 5.5 shows that the combination resulted in a stable and better learning behaviour.

5.2 Transfer Learning

For our transfer learning experiments we use weights of pre-trained network trained on a specific task and initialize the training on next task with the same weights. We categorize

our transfer learning experiments into two categories where section 5.2.1 discuss on transfer learning characteristics when the distribution on observed pixel data on source task and target task have almost similar distribution. Meanwhile, section 5.2.2 discuss the transfer learning characteristics when the distribution on observed pixel data on source task and target task change at user-controlled rate.

5.2.1 Type-1

For our first transfer learning experiment, we inspect the training characteristics when an agent is trained on the FlappyBird environment where the gap between the pipes is 200 pixels and see how the agent learns when the gap is reduced to 150 pixels making the game a bit harder. To compare the transfer characteristics we train the agent on the target task with random initialization and compare it with agent initialized with weights initialized with model trained on source task with $100k; 200k$ and $300k$ iterations.

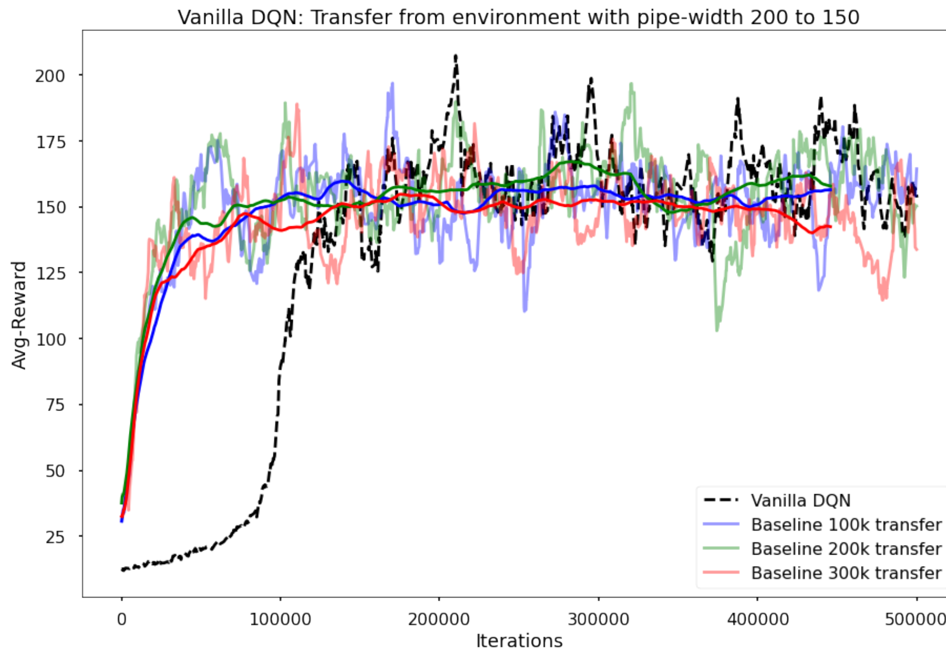

Figure 5.6: Cumulative episodic reward comparison between an *vanilla-DQN* agent that is trained on the environment with PCG parameter 150 to the agent that is pre-trained on the environment with PCG parameter 200 for $100k$, $200k$ and $300k$ iterations

Figure 5.6 details the learning characteristics of vanilla DQN when transferred from source task to target task. The black dotted line in figure 5.6 shows the learning behaviour of vanilla DQN agent with random weight initialization. The dark-solid lines in figure 5.6 represent the rolling average of the agents learning initialized with trained weights on source task to elucidate on training behaviour much clearly. We observe a positive transfer for vanilla DQN agent. However, the e ect of weights transferred from source task trained for $100k; 200k$, and $300k$ show no significant di erence. But, on a closer look we can observe

that the transfer from source task trained for $300k$ performs slightly worse in comparison. Which could be due to the agent trying to unlearn from its experiences gained in source task.

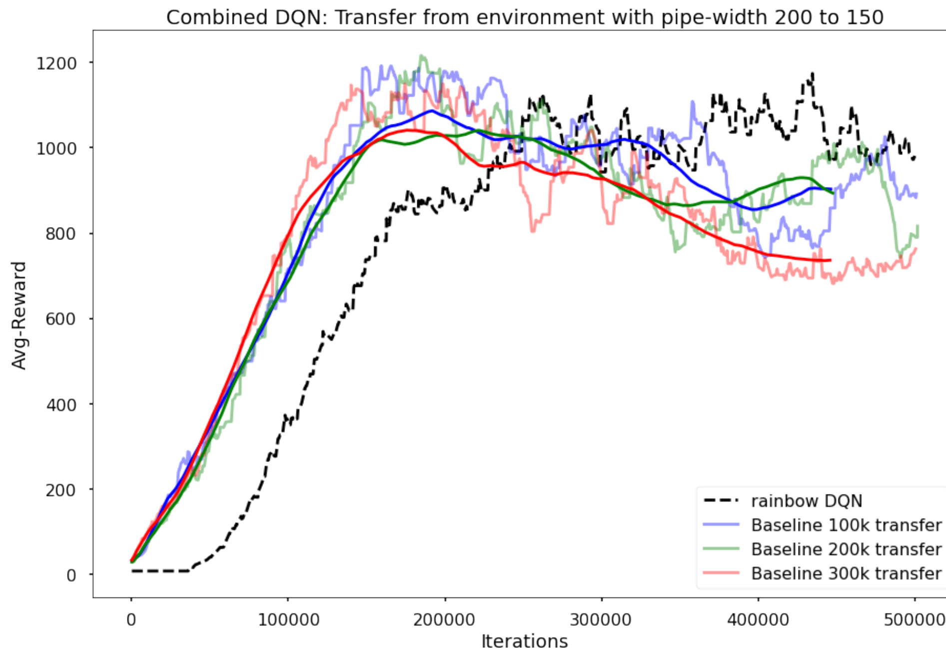

Figure 5.7: Cumulative episodic reward comparison between a *rainbow-DQN* agent that is trained on the environment with PCG parameter 150 to the agent that is pre-trained on the environment with PCG parameter 200 for $100k$, $200k$ and $300k$ iterations

Figure 5.7 shows the learning characteristics of combined DQN with the same source and target task setup. The black dotted line in figure 5.7 is the training dynamics of Combined DQN agent with random initialization. Similar to our results from vanilla DQN, we observe the same positive transfer from source task to target task. Again, we observe that that the transfer from source task trained for $300k$ iterations shows slight performance degradation compared to the ones trained on $200k$ and $100k$ iterations.

5.2.2 Type-2

For our second transfer learning experiment, we inspect the learning characteristics when an agent is trained on the same FlappyBird environment where the gap between the pipes is 200 pixels and see the transfer characteristics of an agent trying to learn slightly more di cult level with where the gap between the pipes is reduced to 150 pixels whiles the environment observation characteristics change with wall appearing in-place of pipes with a probability of $10\%; 25\%; 50\%$ and 100%. To maintain a level field between the agents in our experiment the transfer is initiated when the agent trains on the source task for $500k$ iterations.

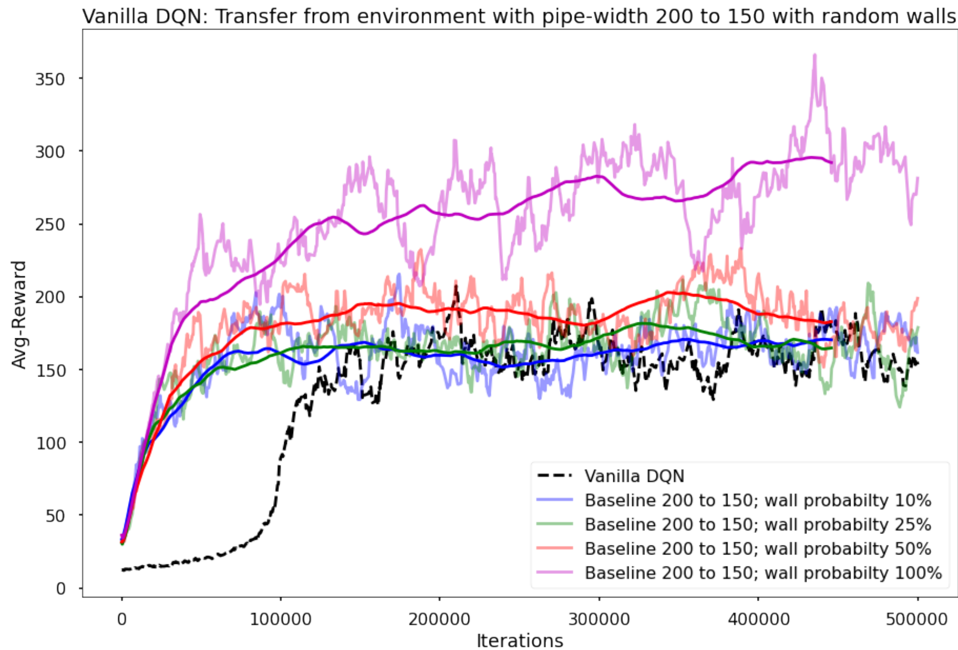

Figure 5.8: Cumulative episodic reward comparison between a *vanilla-DQN* agent that is exclusively trained on the environment with PCG parameter 150 to the agent pre-trained initially on the environment with PCG parameter 200 and resumed on same environment with but varying degrees of stocasticity to replace pipes with walls.

Figure 5.8 visualizes the learning characteristics of vanilla DQN when transferred from source task to target task. The black dotted line in figure 5.8 show the learning characteristics of vanilla DQN on the target task with random weight initialization. We observe a positive transfer from source task to target. From figure 5.8 we see that the transfer works well when the target task constitutes walls in place of pipes. This behaviour is expected as the di culty level of the game is decreased in comparison to the di culty of our target task without walls. What we find interesting is that the more the environment is subject to change, the better the transfer characteristics as shown in figure 5.8 because agent show slightly worse learning characteristics when the walls are occurring with 10% probability. We believe this drop in performance is due to the fact that the replay bu er consists of fewer samples with walls when they generated with a low probability thus, making it di cult for the agent to sample and learn before the experiences are completely lost.

Furthermore, Figure 5.9 shows the learning characteristics of the combined DQN with the same source and target setup. Unlike the previous result with vanilla DQN, we observe a negative transfer in this case.

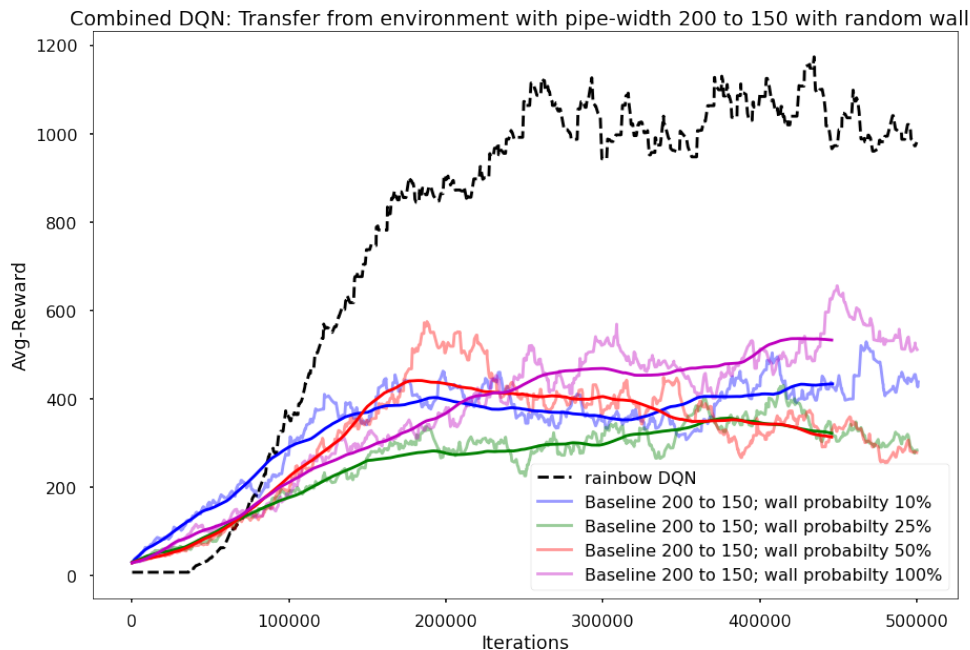

Figure 5.9: Cumulative episodic reward comparison between a *rainbow-DQN* agent that is exclusively trained on the environment with PCG parameter 150 to the agent pre-trained initially on the environment with PCG parameter 200 and resumed on same environment with but varying degrees of stocasticity to replace pipes with walls.

5.3 ALP-Curriculum: DQN specific

To test our curriculum approaches outlined in section 3 we use our self implemented FlappyBird environment that procedurally generates obstacles that are vertically separated by user controlled parameter called *pipe-gap*. In all our experiments, we create a task pool with three di erent *pipe-gap* configurations

- task-0: *pipe-gap* = 225
- task-1: *pipe-gap* = 175
- task-2: *pipe-gap* = 125

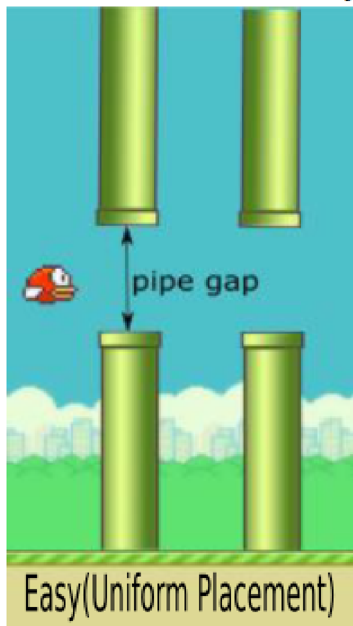

Figure 5.10: Observation space for our Easy and Hard tasks. In the Easy task, the pipes are placed uniformly and in the Hard task,the pipes are added in random locations

Intuitively, we expect the agent to learn a sub-standard policies from easier tasks that could aid in faster and better learning on harder tasks. The standard game of FlappyBird is designed to run with a fixed pipe gap of 175 thus making experiences obtained on environment with pipe gap of 225 the easier task and 125 the harder task in our task pool. Furthermore, we define two versions of the environment, *Easy* and *Hard*. The easy version of the environment generates pipes that are placed uniformly and the hard version introduces stochasticity in placement and arrangement as shown in figure 5.10. We discuss our results in two folds, the first sub-section examines our transfer learning results when the environment observation space is RGB encoded and the later sub-section details on curriculum learning results when the environment observation space is represented as a 1-D vector.

5.4 FlappyBird-RGB

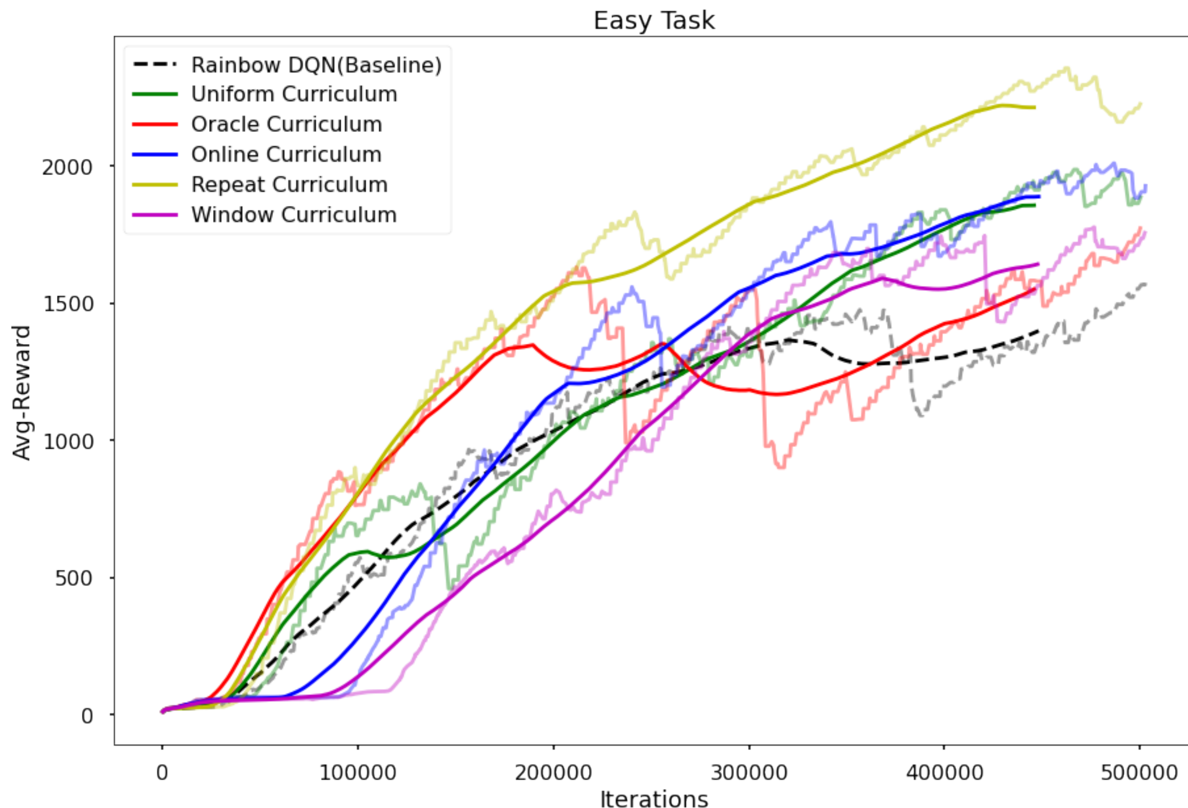

Figure 5.11: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

This section discusses our curriculum-learning results when the provided inputs for agents' learning are the true visual stimuli(RBG values) from the environment. Figure 5.11 shows the training characteristics of our combined DQN agent trained on the *Easy* environment using di erent curriculum strategies discussed in section 3.5. Our curriculum strategies are compared against the baseline trained solely on the hardest task(pipe-gap of 125) and the oracle curriculum strategy designed using human intuitive knowledge. We observe that all the automated curriculum strategies perform better than the baseline. However, the online curriculum strategy and window curriculum strategy show slow learning compared to the baseline in the initial stages of learning but outperforms the baseline after $300;000$ iterations of training. This drop in performance during initial learning iterations could be explained due to the stochastic task selection behaviour of our task scheduler. Figure 7.4 in Appendix shows the task distribution handled by each representative curriculum task scheduler. The slow learning progress for online curriculum could be explained due to its bias towards training on the easiest task(Task-0) than much harder task(Task-2). In addition, training more on the easiest task could have aided the agent learn a robust policy on harder task by honing on its experiences from the easy task. On the contrary,

the higher learning performance using repeat curriculum strategy can be explained by its higher a nity towards choosing the di cult task after training on easier tasks initially.

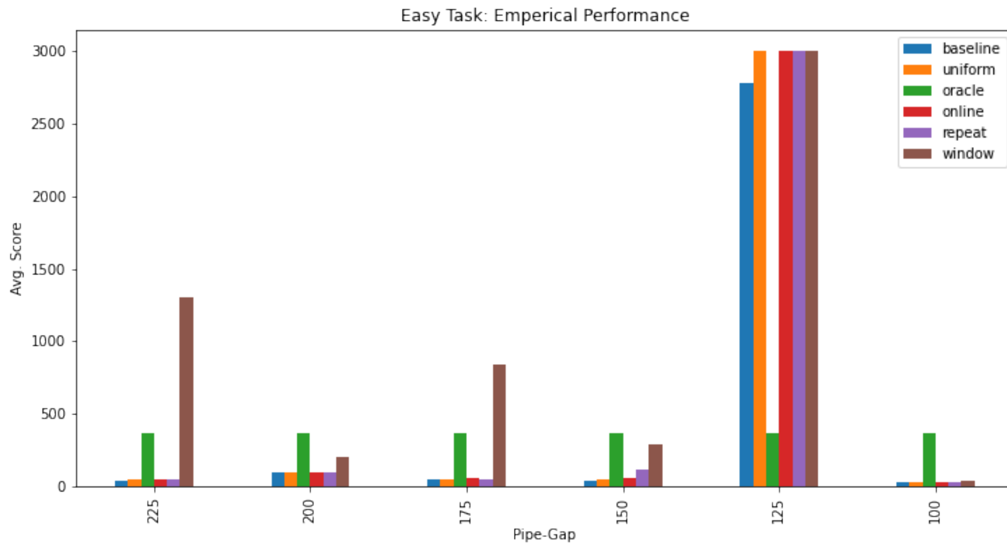

Figure 5.12: Emperical performance of the agents employed with di erent curriculum strategies on environments with uniform pipe placement for di erent PCG parameter(pipe-gap).

Figure 5.12 shows the empirical performance of our trained agent over the range of di erent environment parameters. During the training phase, the agent is shown experiences from tasks with a pipe gap of 225;175 and 125 making environments with pipe-gap parameters 200;150 and 100 unseen. From our results on environment with Easy task, we observe that the baseline agent achieves good performance on the trained task with pipe gap 125 and performs worse on easier tasks with larger pipe gaps. Furthermore, all other employed curriculum strategies except the oracle strategy show a slight improvement in performance compared to the baseline on all other tasks. However, what we find interesting is that the oracle strategy has almost equal performance on all tasks including the unseen hard task. This behaviour of showing equal performance on tasks could be due to the human induced learning heuristics that fail to counter the forgetting on easier tasks while transitioning to much di cult tasks making learning unstable.

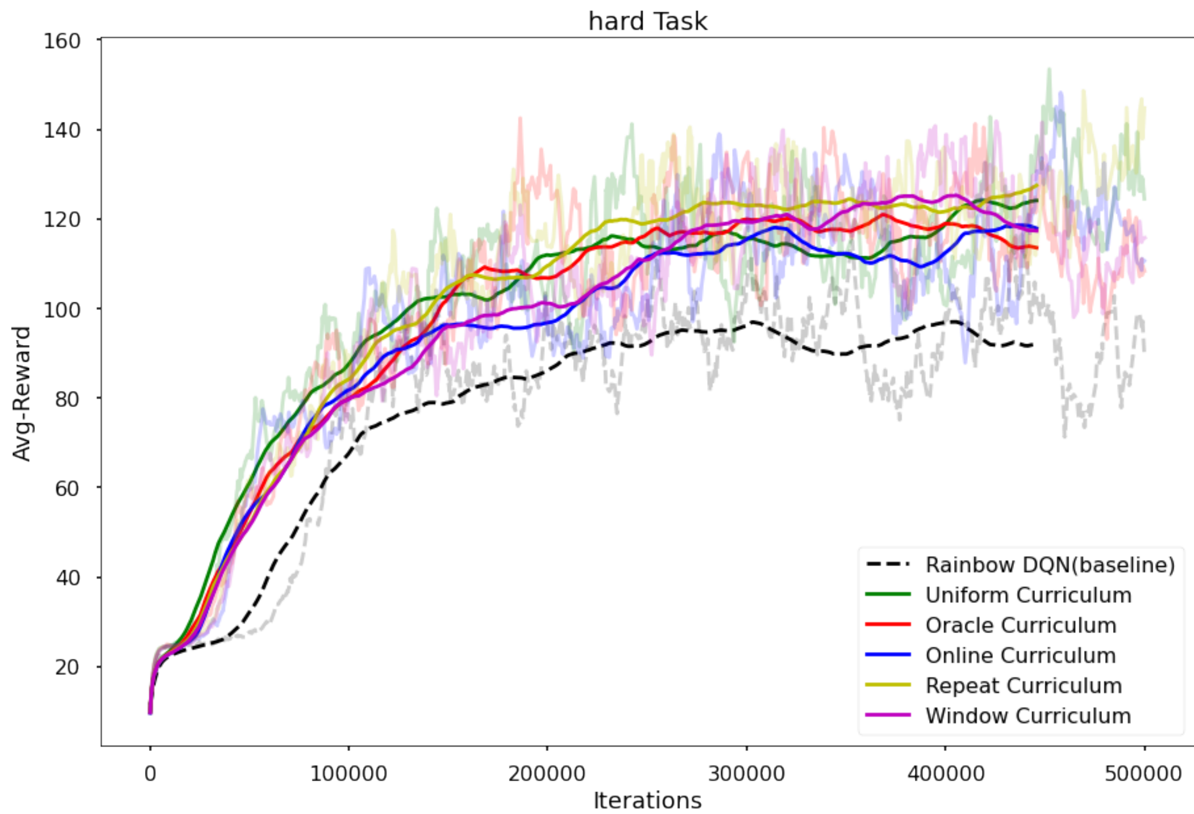

Figure 5.13: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

Figure 5.13 shows the learning characteristics of our curriculum strategies against the combined DQN agent when the environment is subjected to stochastic pipe placement. We clearly observe better learning behaviour from the agents trained using curriculum strategies.

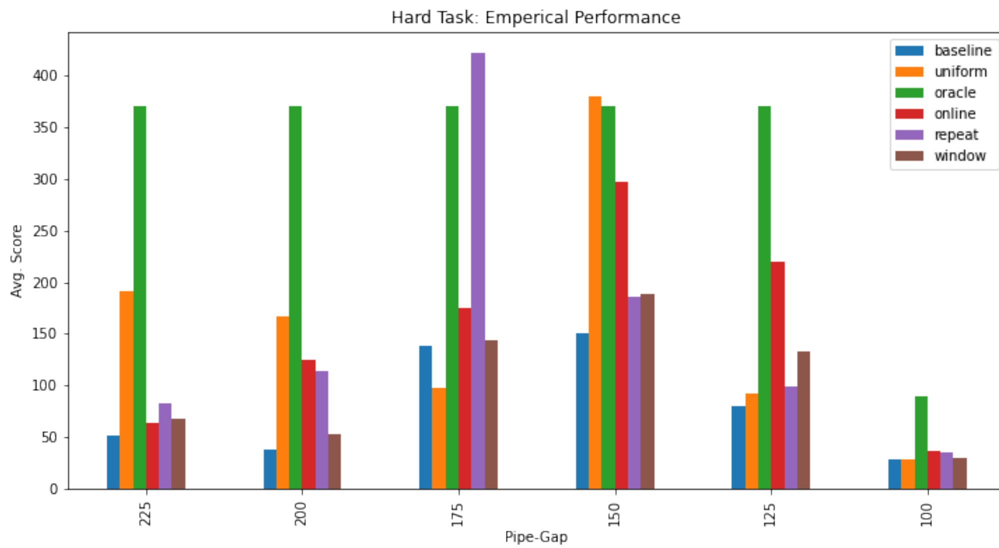

Figure 5.14: Emperical performance of the agents employed with di erent curriculum strategies on environments with random pipe placement for di erent PCG parameter(pipe-gap).

Our previous observation that the baselines performs better on the task it is trained on but performs worse on our pool of testing tasks holds true for harder environment configurations as shown in Figure 5.14. Furthermore, the oracle curriculum strategy performs significantly well unlike its performance when the environment has no degree of stocasticity in the placement of pipes(EASY task). Introducing stocaticity in our environment might be the reason we observe overall relative improvement in performance on tasks that are not the target tasks(i.e., pipe-gap $\neq$ 125) as the gathered experiences are diverse, making the agents ability to handle changes robustly.

5.5 FlappyBird-LIDAR

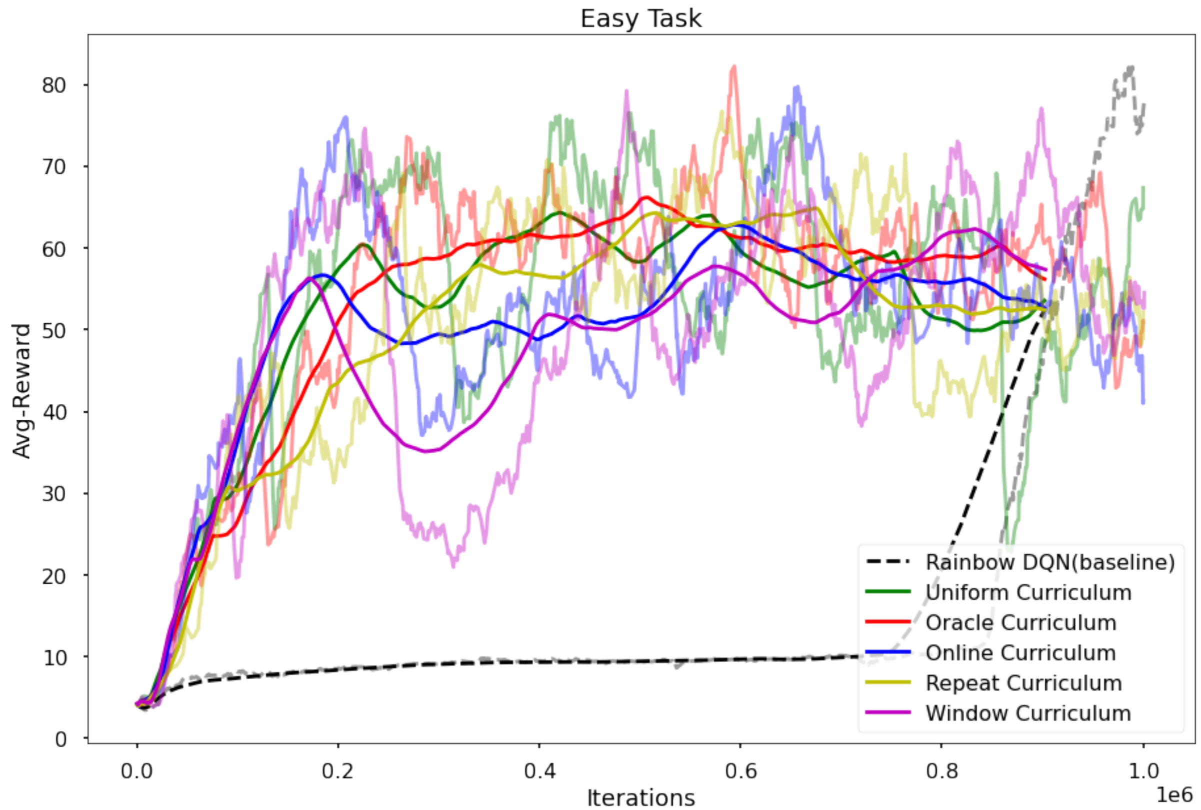

Figure 5.15: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms

This section discusses our curriculum results when the provided inputs for the agents are proximal numerical representations as discussed in section 4. Figure 5.15 shows the learning characteristics of our agents trained using di erent curriculum strategies detailed in section 3. Irrespective of the observation space representation, we observe the similar trend where the learning using curriculum strategies is better than the baseline performance that is trained on the hardest task alone.

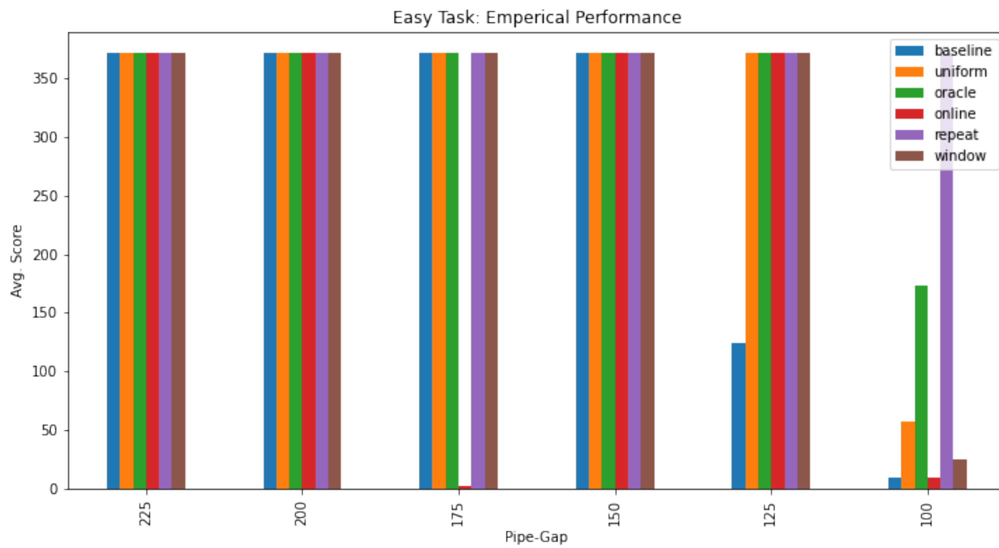

Figure 5.16: Emperical performance of the agents employed with di erent curriculum strategies on environments with uniform pipe placement for di erent PCG parameter(pipe-gap).

Figure 5.16 further attests to our previous observations by showing that the agents empirical performance on wide range of tasks is significantly better than that of the baseline. For a the specific case where the environment is simple and the observation space is a 1-D vector, repeat curriculum shows better performance on all tasks including the unseen yet more di cult task with a pipe gap of 100.

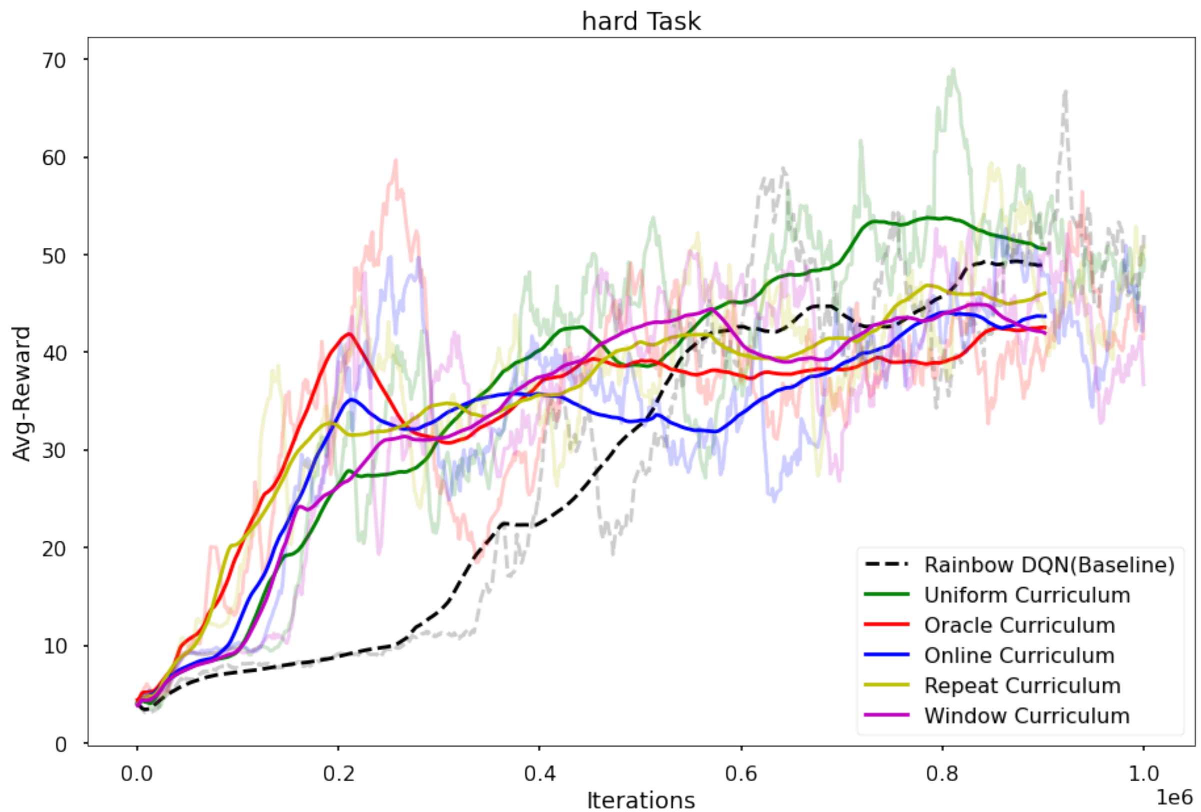

Figure 5.17: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

The learning characteristics of agents trained using di erent curriculum strategies when the environment is subjected to stocastic change is shown in Figure 5.17. We observe that that our curriculum task selection strategies show faster learning behaviour in comparison to the baseline but exhibit similar learning characteristics during the far later iterations of learning.

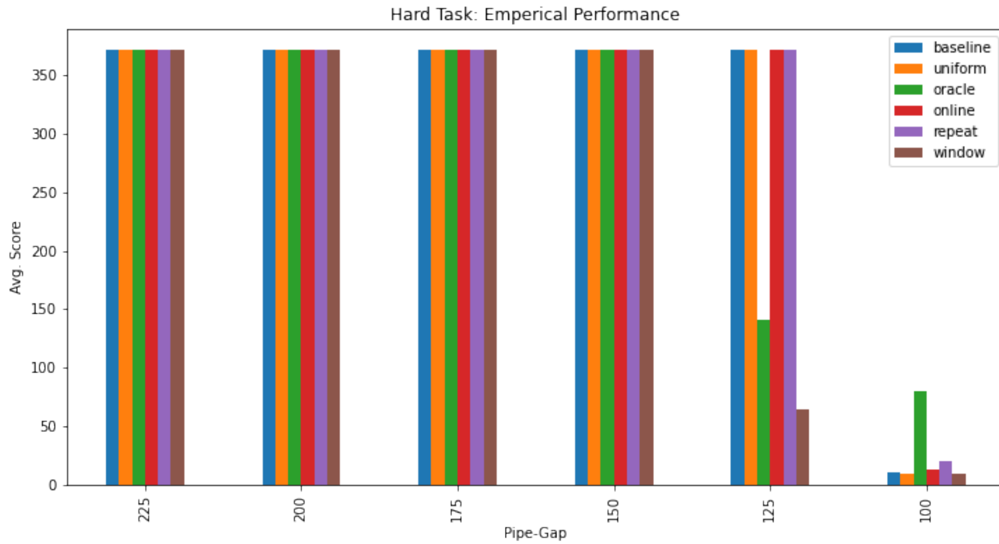

Figure 5.18: Emperical performance of the agents employed with di erent curriculum strategies on environments with random pipe placement for di erent PCG parameter(pipe-gap).

Additionally, the empirical performance of our agents shown in Figure 5.18 correspond to our previous observation that the baseline performs on-par with agents trained using our curriculum approaches. However, window-curriculum task selection and oracle show degraded performance on our target task which is quite unexpected. This drop in performance could be attributed to the fact that the environment is complex but the observation space is quite myopic for the agent to learn and make appropriate decisions.

Chapter 6

Conclusions

In our work we compared di erent Deep Q-network variants and showed that the combination of them resulted in a strong agent with stable learning characteristics.
Furthermore, we demonstrated di erent cases for continual transfer, and empirically showed the success of transfer learning when both the source task and target task share common visual distribution. Our observations show a positive *jumpstart*, *asymptotic learning performance* and higher *total reward* which are three transfer success metrics discussed in [44].
On tasks where the target task is altered we see vanilla DQN showing better transfer characteristics in comparison to the combined DQN. We note the degradation in performance in the combined DQN is due to improper network initialization of our noise parameters.

The significant part of our paper lies in our formulation of automated curriculum learning algorithm that selects tasks and samples experiences based on the agents learning progress on the set of tasks. We discussed three variants named Online, Repeat and Window each estimating the agents learning progress di erently. Based on our experimental results we empirically showed that our implementation of automated curriculum learning strategies perform significantly better to the agent trained individually on the hardest task.
Furthermore, we carried out the same experiments when the observation space of our environment is represented as a vector rather than a tensor of RGB values. We show that our curriculum approaches works well irrespective of the observation space representation.

6.1 Future Work

Our approach of transfer learning when the source task and target task are similar but di er in level di culty has shown good transfer characteristics. Our work on designing the automated curriculum learning framework is currently limited to discrete set of tasks chosen based on pre-selected values. Our future work could focus on developing a curriculum

framework that automatically chooses task by choosing the parameter values from a continuous interval without human supervision.
Moreover, since our work is centered around choosing PCG parameters from a set of discreet values the next plausible extension would be to extend the task selection on a continuous PCG parameter space.
Additionally, since the entirety of our work is focused on value-based methods, we could extend our methods towards policy gradient algorithms in the future.

Bibliography

1. Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W. & Bergstra, J. Benchmarking Reinforcement Learning Algorithms on Real-World Robots. *arXiv:1809.07731 [cs, stat].* arXiv: 1809. 07731 (Sept. 20, 2018).
2. Komorowski, M., Celi, L. A., Badawi, O., Gordon, A. C. & Faisal, A. A. The Artificial Intelligence Clinician learns optimal treatment strategies for sepsis in intensive care. *Nature Medicine* 24, 1716–1720 (Nov. 2018).
3. Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S. & Pérez, P. Deep Reinforcement Learning for Autonomous Driving: A Survey. *arXiv:2002.00444 [cs].* arXiv: 2002. 00444 (Jan. 23, 2021).
4. Paulus, R., Xiong, C. & Socher, R. A DEEP REINFORCED MODEL FOR ABSTRACTIVE SUMMARIZATION, 12.
5. Xiong, Z., Liu, X.-Y., Zhong, S., Yang, H. & Walid, A. Practical Deep Reinforcement Learning Approach for Stock Trading. *arXiv:1811.07522 [cs, q-fin, stat].* arXiv: 1811. 07522 (Dec. 1, 2018).
6. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K. & Hassabis, D. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815. arXiv: 1712. 01815 (2017).
7. OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., D biak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F. & Zhang, S. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv:1912.06680 [cs, stat].* arXiv: 1912.06680 (Dec. 2019).
8. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. Human-level control through deep reinforcement learning. *Nature* 518, 529–533. ISSN: 00280836 (Feb. 2015).
9. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs].* arXiv: 1312.5602 (Dec. 2013).
10. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* Ed. 2 (The MIT Press, 2018).
11. OpenAI, Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L. & Zaremba, W. Learning Dexterous In-Hand Manipulation. arXiv: 1808.00177 (Jan. 2019).

12. Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. & Silver, D. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv:1710.02298 [cs].* arXiv: 1710.02298 (Oct. 2017).
13. Van Hasselt, H., Guez, A. & Silver, D. Deep Reinforcement Learning with Double Q-learning. *arXiv:1509.06461 [cs].* arXiv: 1509. 06461 (Dec. 8, 2015).
14. Schaul, T., Quan, J., Antonoglou, I. & Silver, D. Prioritized Experience Replay. *arXiv:1511.05952 [cs].* arXiv: 1511. 05952 (Feb. 25, 2016).
15. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. & de Freitas, N. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv:1511.06581 [cs].* arXiv: 1511. 06581 (Apr. 5, 2016).
16. Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C. & Legg, S. Noisy Networks for Exploration. *arXiv:1706.10295 [cs, stat].* arXiv: 1706. 10295 (July 9, 2019).
17. Bellemare, M. G., Dabney, W. & Munos, R. A Distributional Perspective on Reinforcement Learning. *arXiv:1707.06887 [cs, stat].* arXiv: 1707. 06887 (July 21, 2017).
18. Argall, B. D., Chernova, S., Veloso, M. & Browning, B. A survey of robot learning from demonstration. *Robotics and Autonomous Systems* 57, 469–483 (May 2009).
19. Schaal, S. *Learning from Demonstration* in *Advances in Neural Information Processing Systems* 9 (MIT Press, 1997).
20. Zhang, X. & Ma, H. Pretraining Deep Actor-Critic Reinforcement Learning Algorithms With Expert Demonstrations. *arXiv:1801.10459 [cs, stat].* arXiv: 1801. 10459 (Feb. 9, 2018).
21. Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W. & Abbeel, P. Overcoming Exploration in Reinforcement Learning with Demonstrations. *arXiv:1709.10089 [cs].* arXiv: 1709. 10089 (Feb. 25, 2018).
22. Bengio, Y. *Deep Learning of Representations: Looking Forward* in *Statistical Language and Speech Processing* (eds Dediu, A.-H., Martín-Vide, C., Mitkov, R. & Truthe, B.) (Springer, Berlin, Heidelberg, 2013), 1–37.
23. Yin, H. & Pan, S. J. Knowledge Transfer for Deep Reinforcement Learning with Hierarchical Experience Replay, 7.
24. Parisotto, E., Ba, J. L. & Salakhutdinov, R. Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. *arXiv:1511.06342 [cs].* arXiv: 1511. 06342 (Feb. 22, 2016).
25. Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. & Silver, D. Successor Features for Transfer in Reinforcement Learning. *arXiv:1606.05312 [cs].* arXiv: 1606. 05312 (Apr. 12, 2018).
26. Soviany, P., Ionescu, R. T., Rota, P. & Sebe, N. Curriculum Learning: A Survey. *arXiv:2101.10382 [cs].* arXiv: 2101.10382 (Jan. 2021).
27. Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E. & Stone, P. Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. *arXiv:2003.04960 [cs, stat].* arXiv: 2003.04960 (Sept. 2020).
28. Portelas, R., Colas, C., Weng, L., Hofmann, K. & Oudeyer, P.-Y. Automatic Curriculum Learning For Deep RL: A Short Survey. *arXiv:2003.04664 [cs, stat].* arXiv: 2003.04664 (May 2020).

29. Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H. & Silver, D. Distributed Prioritized Experience Replay. *arXiv:1803.00933 [cs].* arXiv: 1803.00933 (Mar. 2018).

30. Flet-Berliac, Y. & Preux, P. Only Relevant Information Matters: Filtering Out Noisy Samples to Boost RL. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence,* 2711–2717 (July 2020).

31. Mysore, S., Platt, R. & Saenko, K. Reward-guided Curriculum for Robust Reinforcement Learning, 10.

32. Portelas, R., Colas, C., Hofmann, K. & Oudeyer, P.-Y. Teacher algorithms for curriculum learning of Deep RL in continuously parameterized environments. *arXiv:1910.07224 [cs, stat].* arXiv: 1910. 07224 (Oct. 16, 2019).

33. Mehta, B., Diaz, M., Golemo, F., Pal, C. J. & Paull, L. Active Domain Randomization. *arXiv:1904.04762 [cs].* arXiv: 1904. 04762 (July 10, 2019).

34. Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfa , T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C. & Silver, D. Grandmaster level in StarCraft II using multi-agent reinforcement learning. en. *Nature* 575, 350–354 (Nov. 2019).

35. Plaat, A. *Learning to Play: Reinforcement Learning and Games* en (Springer International Publishing, 2020).

36. Arulkumaran, K., Deisenroth, M. P., Brundage, M. & Bharath, A. A. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine* 34, 26–38. arXiv: 1708. 05866 (Nov. 2017).

37. Sutton, R. S. Learning to predict by the methods of temporal di erences. *Machine Learning* 3, 9–44 (Aug. 1, 1988).

38. Hernandez-Garcia, J. F. & Sutton, R. S. Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target. *arXiv:1901.07510 [cs, stat].* arXiv: 1901. 07510 (Feb. 7, 2019).

39. Wu, Y. & Tian, Y. TRAINING AGENT FOR FIRST-PERSON SHOOTER GAME WITH ACTOR-CRITIC CURRICULUM LEARNING, 10 (2017).

40. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. in *Advances in Neural Information Processing Systems 32* (eds Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. & Garnett, R.) 8024–8035 (Curran Associates, Inc., 2019).

41. Fomin, V., Anmol, J., Desroziers, S., Kriss, J. & Tejani, A. *High-level library to help with training neural networks in PyTorch* https://github.com/pytorch/ignite. 2020.

42. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. OpenAI Gym. *arXiv:1606.01540 [cs].* arXiv: 1606. 01540 (June 5, 2016).

43. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *ImageNet Classification with Deep Convolutional Neural Networks* in *Advances in Neural Information Processing Systems* 25 (Curran Associates, Inc., 2012).

44. Taylor, M. E. & Stone, P. Transfer Learning for Reinforcement Learning Domains: A Survey, 53.

Chapter 7

Appendix

7.1 Experiment - 1

This section entails the supplementary figures from the experiments carried out on the FlappyBird-RGB environment using our automated curriculum learning algorithms.

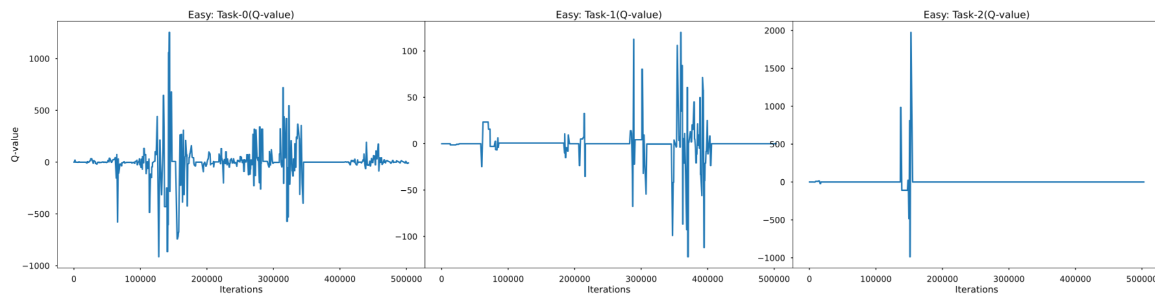

Figure 7.1: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with uniform pipe placement

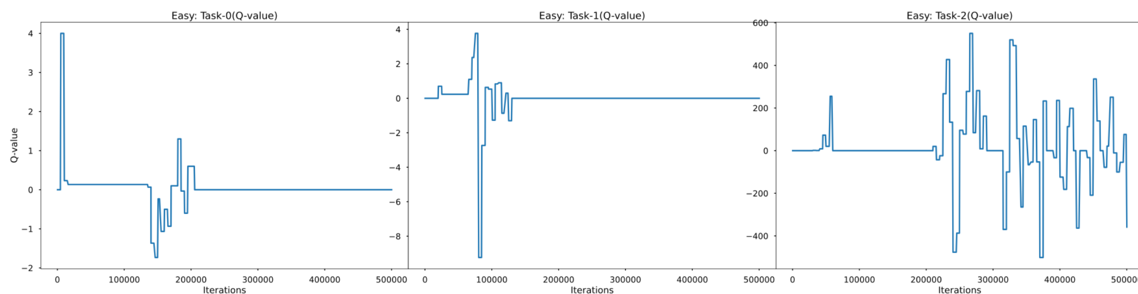

Figure 7.2: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with uniform pipe placement

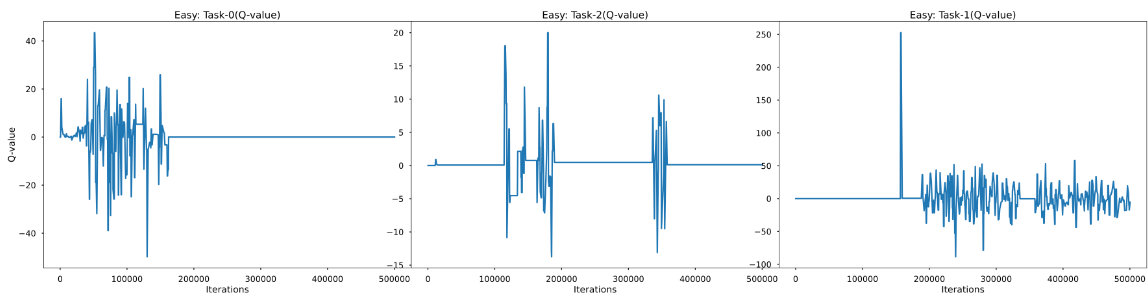

Figure 7.3: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with uniform pipe placement

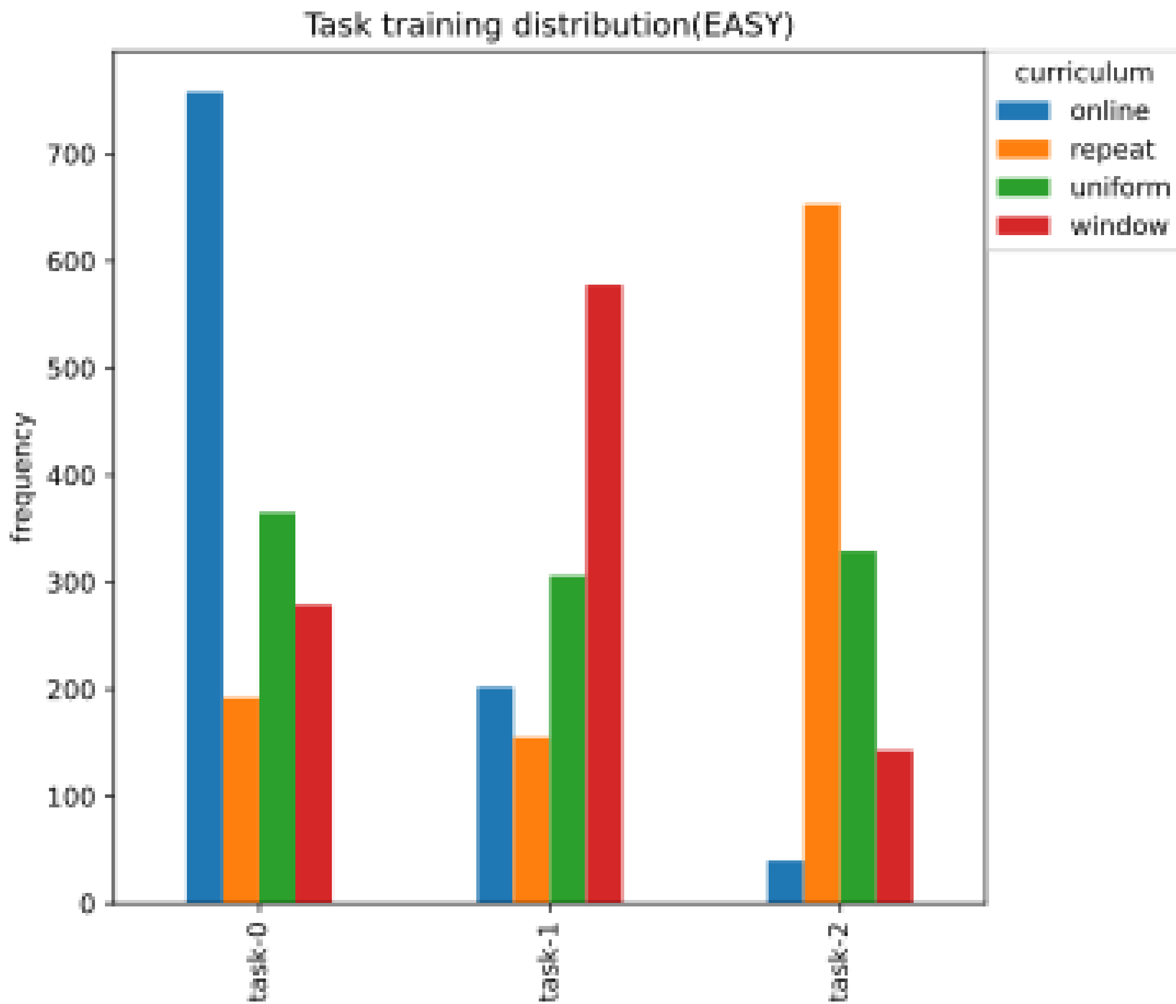

Figure 7.4: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with uniform pipe placement

Figure 7.5: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with random pipe placement

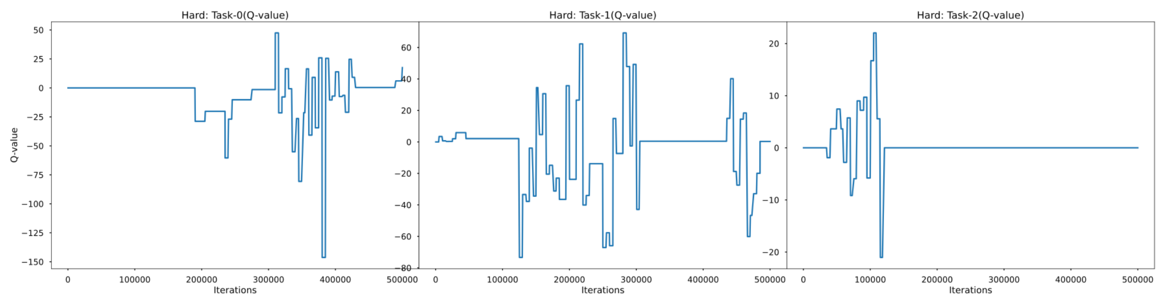

Figure 7.6: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with random pipe placement

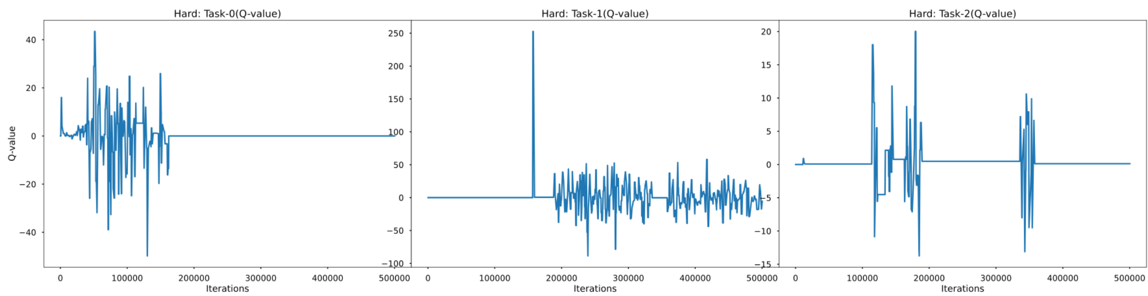

Figure 7.7: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with random pipe placement

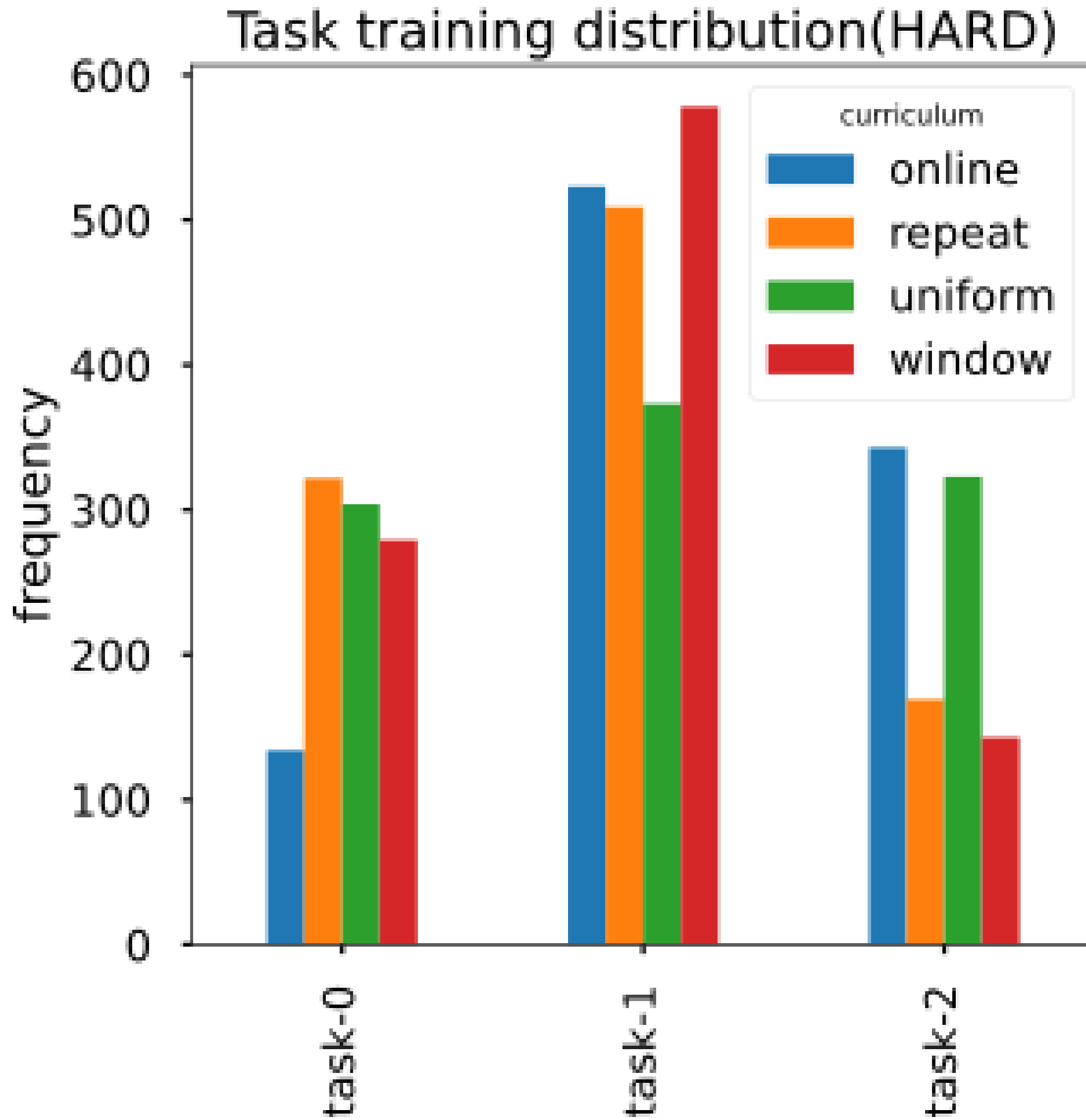

Figure 7.8: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with random pipe placement

7.2 Experiment - 2

This section entails the experiments carried out by decoupling the replay bu er from our automated curriculum learning framework to make the algorithm more general for any RL methods. The following figures are from the experiments run on our FlappyBird-RGB environment.

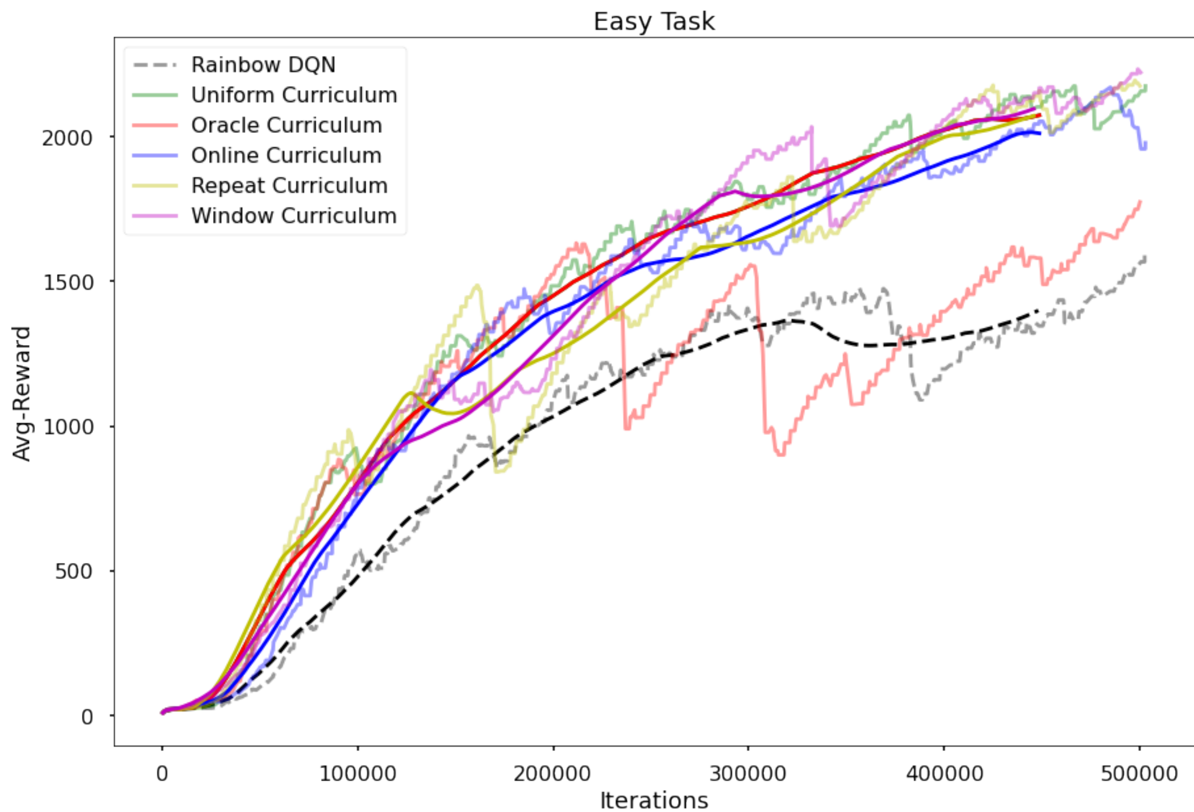

Figure 7.9: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

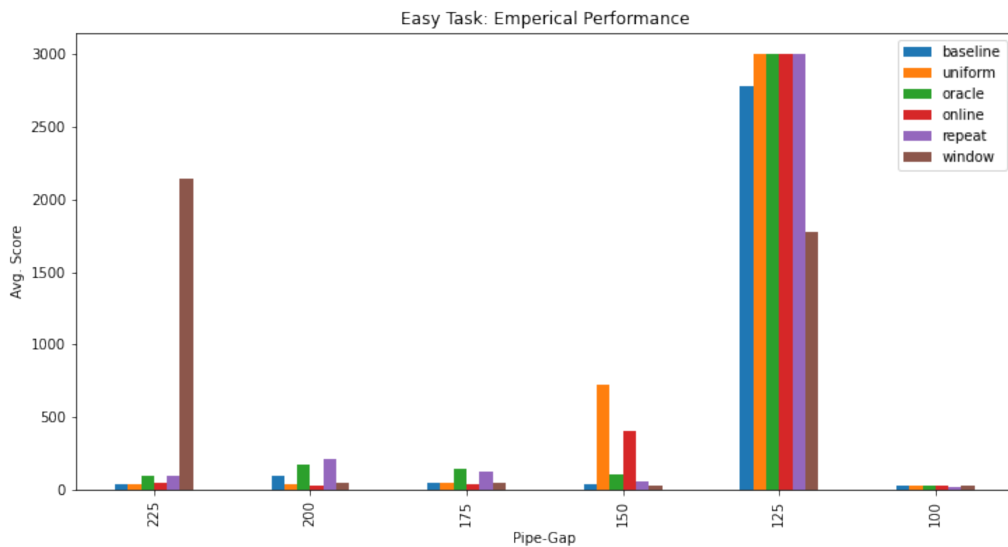

Figure 7.10: Emperical performance of the agents employed with di erent curriculum strategies on environments with uniform pipe placement for di erent PCG parameter(pipe-gap).

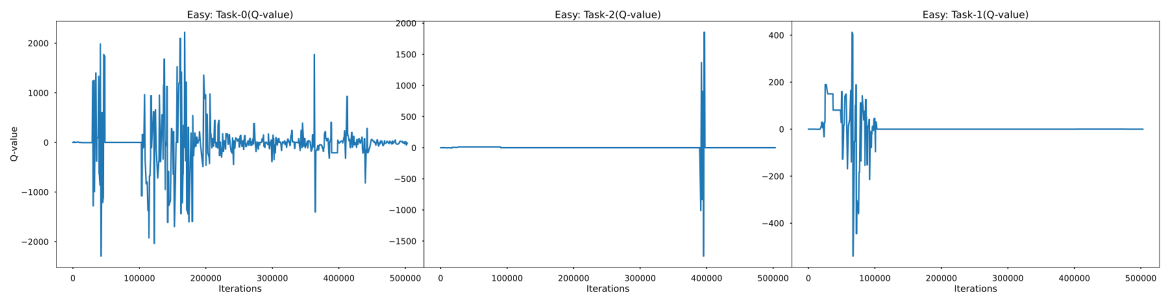

Figure 7.11: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with uniform pipe placement

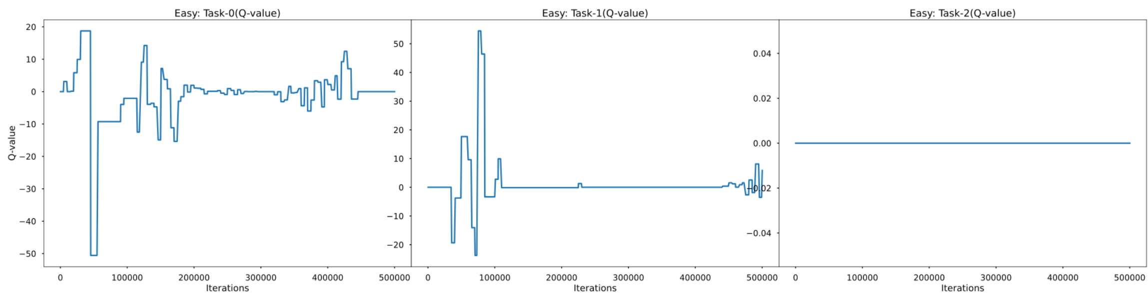

Figure 7.12: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with uniform pipe placement

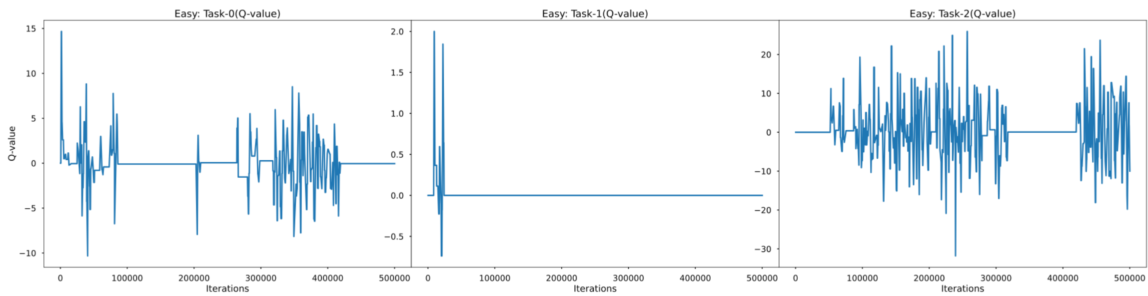

Figure 7.13: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with uniform pipe placement

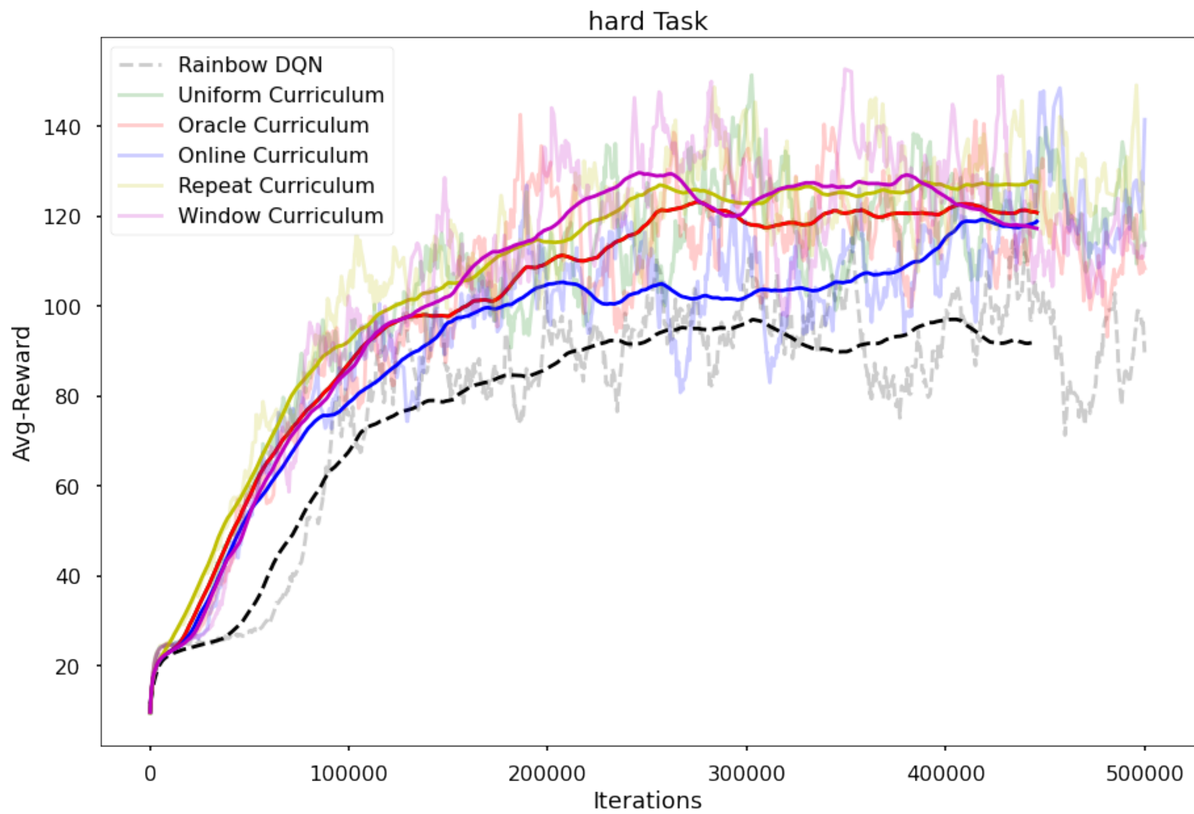

Figure 7.14: Cumulative episodic reward for agents trained on the same environment with random pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

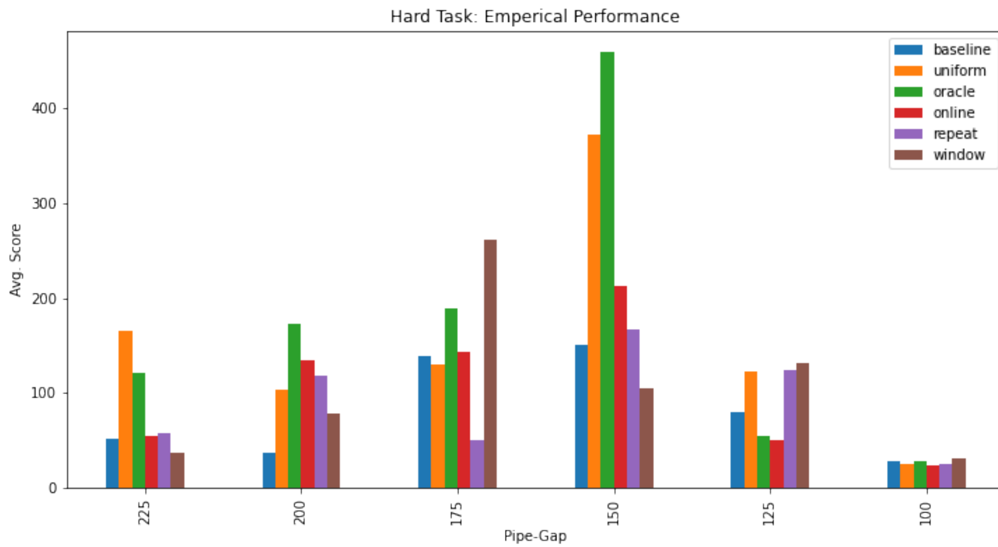

Figure 7.15: Emperical performance of the agents employed with di erent curriculum strategies on environments with random pipe placement for di erent PCG parameter(pipe-gap).

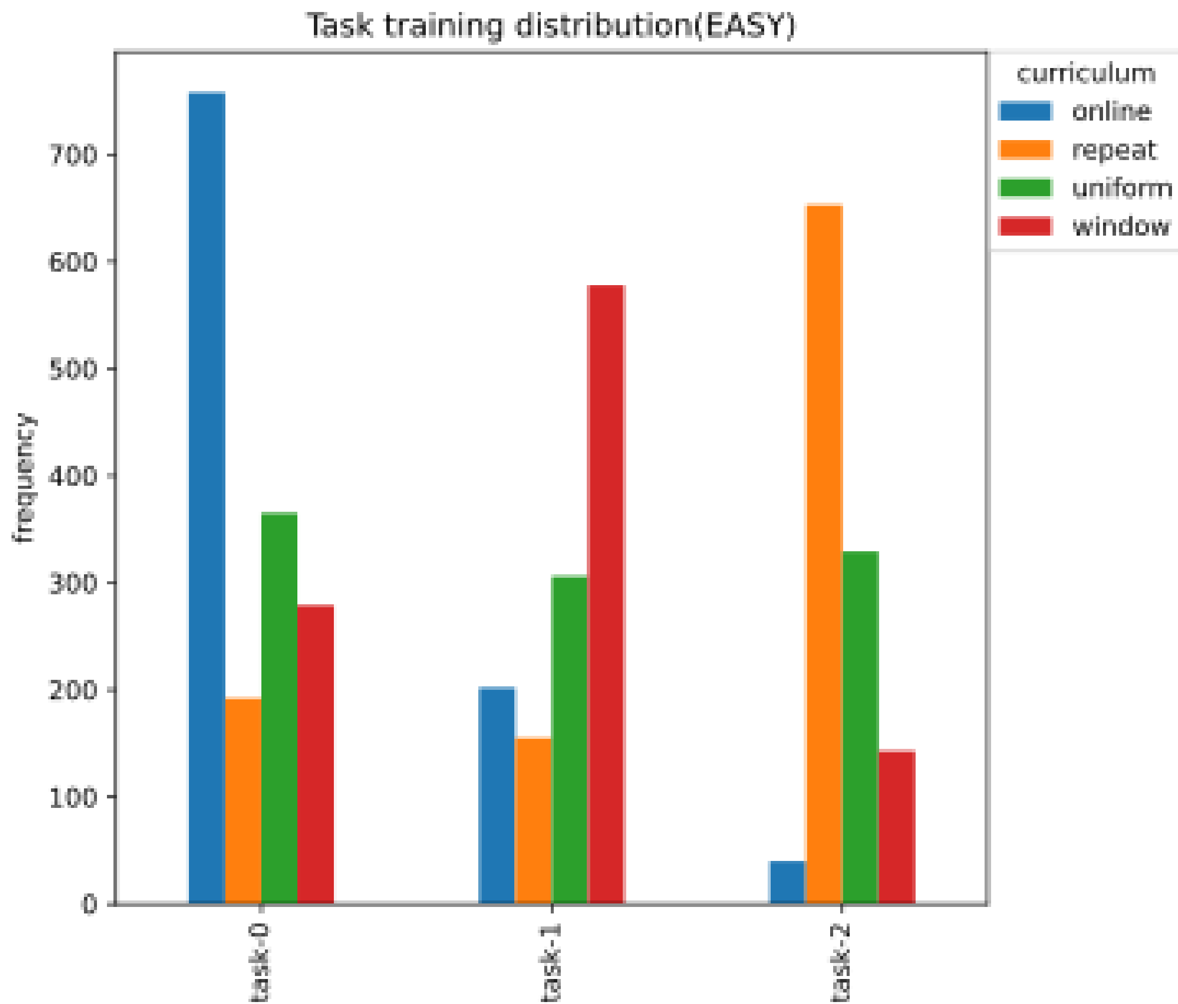

Figure 7.16: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with uniform pipe placement

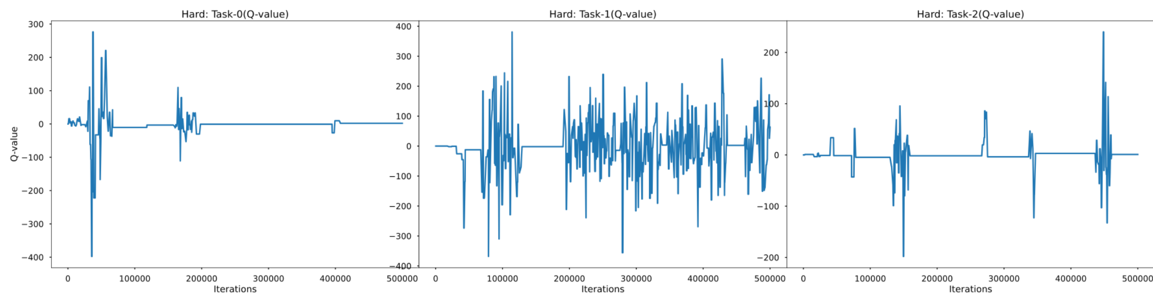

Figure 7.17: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with random pipe placement

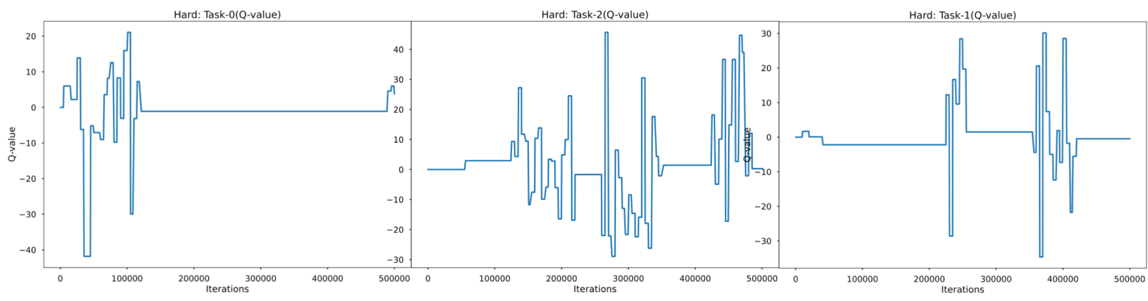

Figure 7.18: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with random pipe placement

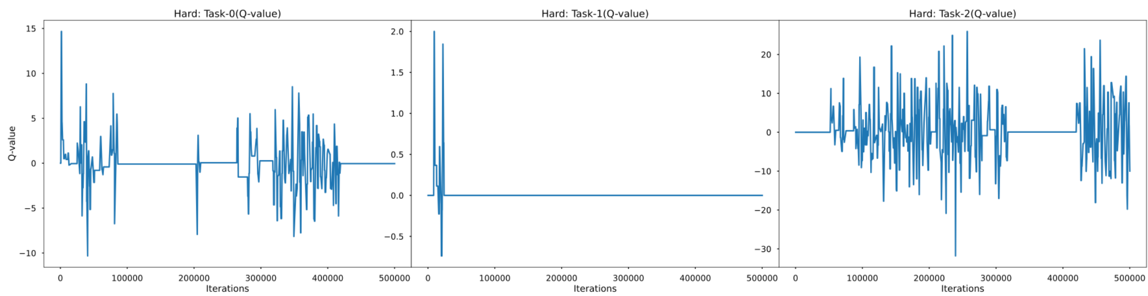

Figure 7.19: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with random pipe placement

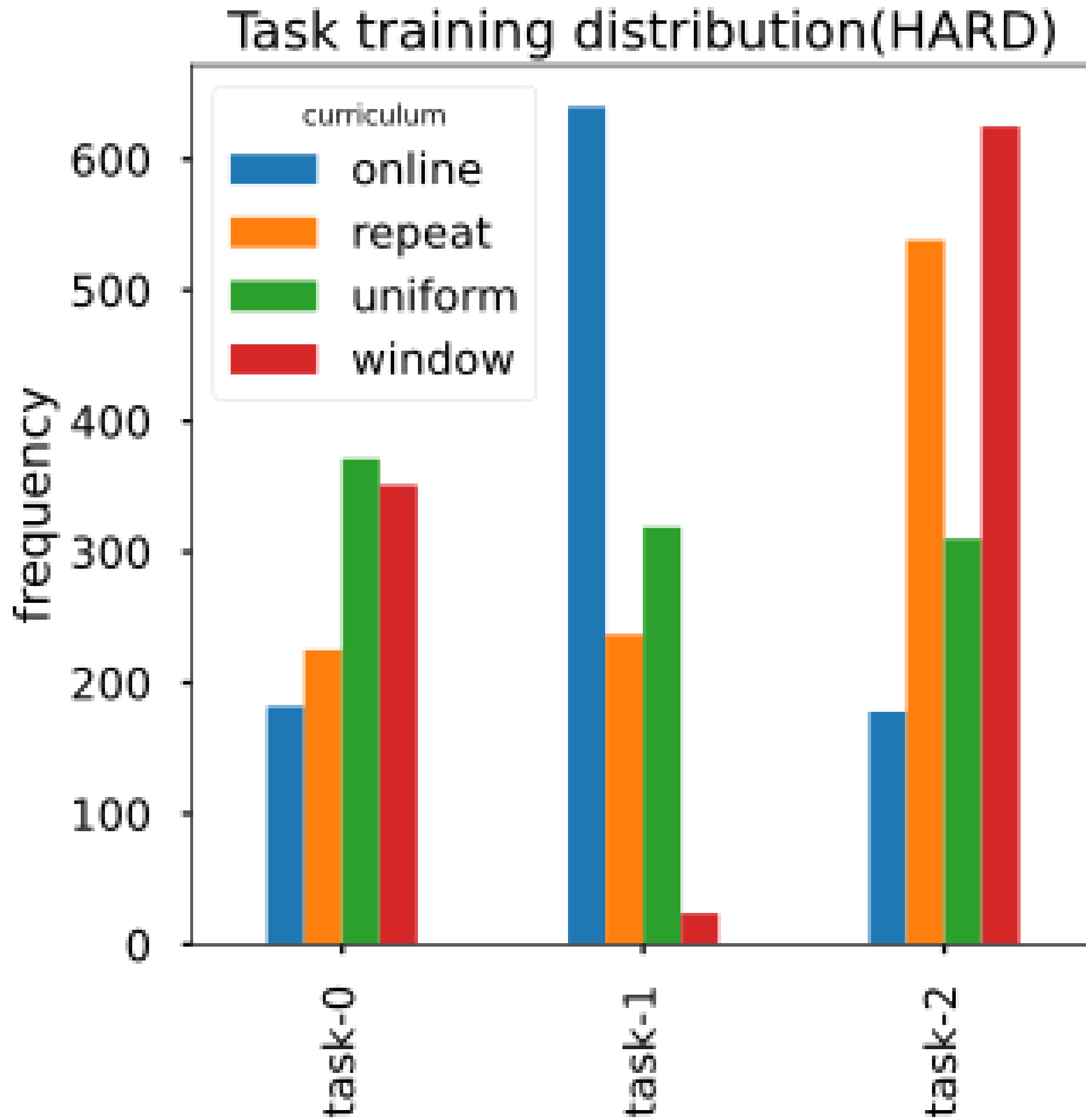

Figure 7.20: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with random pipe placement

7.3 Experiment - 3

This section entails the supplementary figures from the experiments carried out on the FlappyBird-LIDAR environment using our automated curriculum learning algorithms.

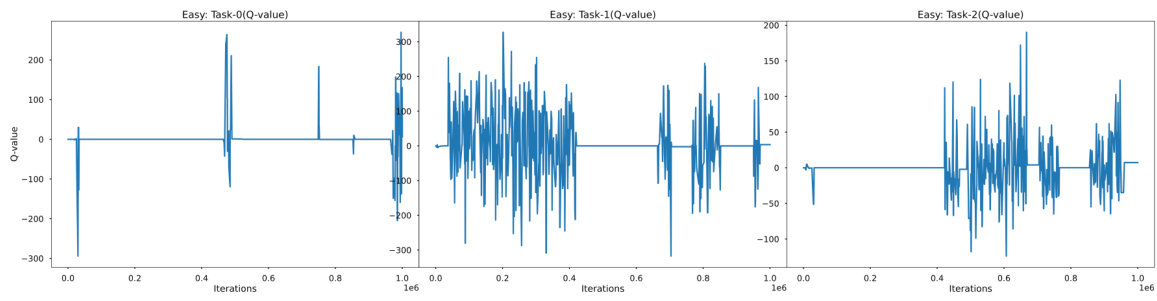

Figure 7.21: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with uniform pipe placement

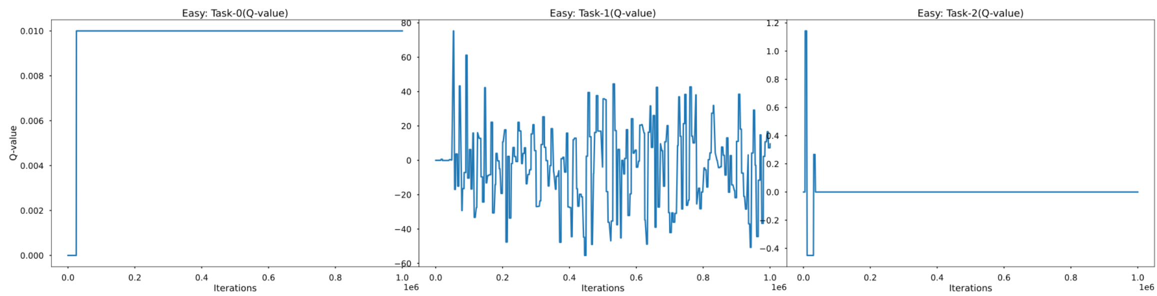

Figure 7.22: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with uniform pipe placement

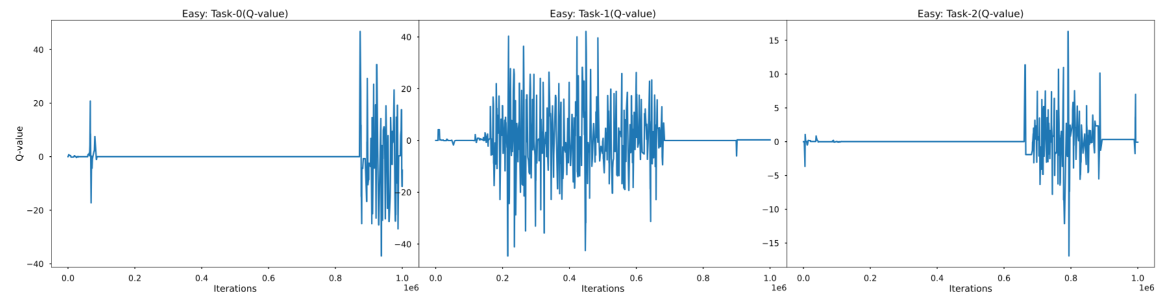

Figure 7.23: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with uniform pipe placement

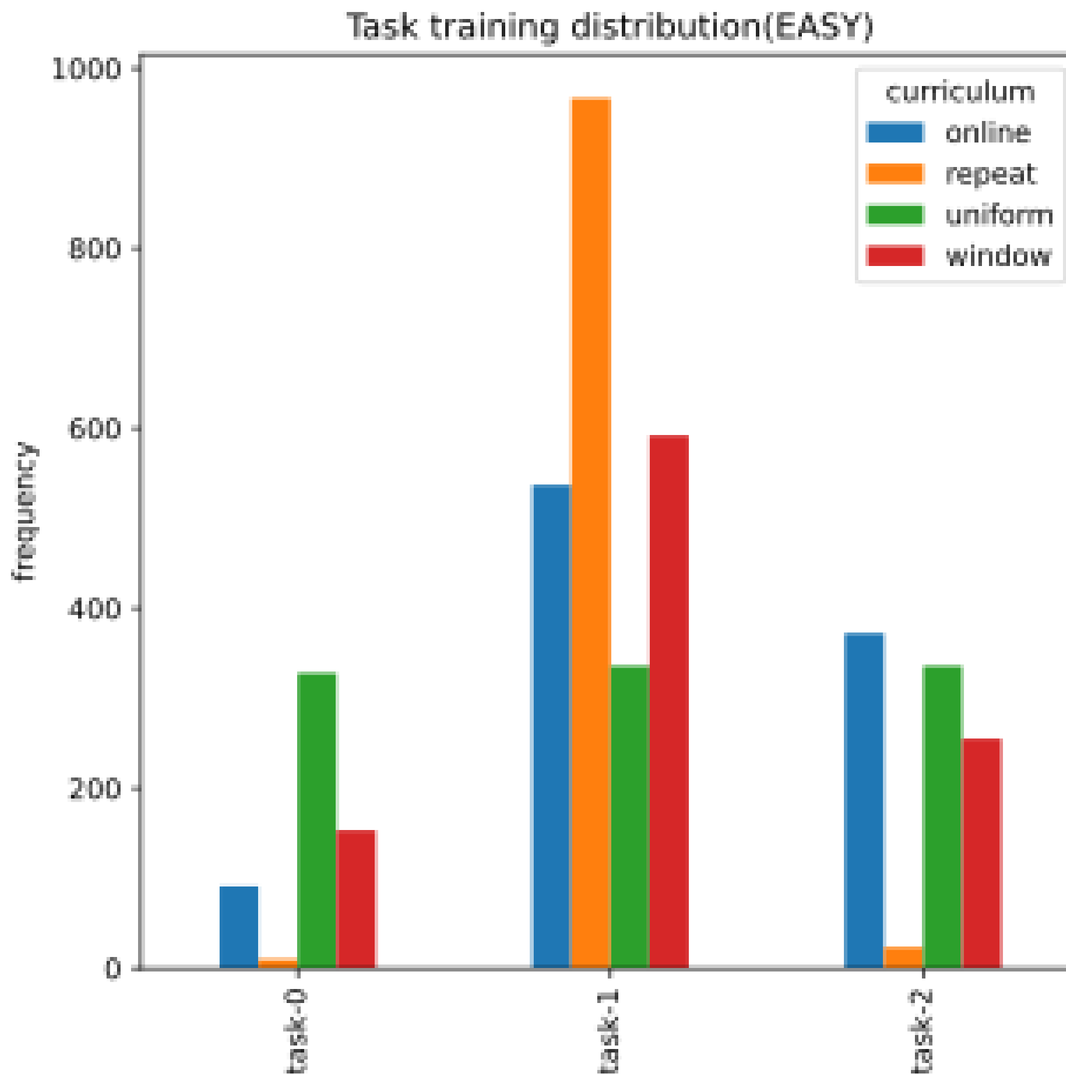

Figure 7.24: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with uniform pipe placement

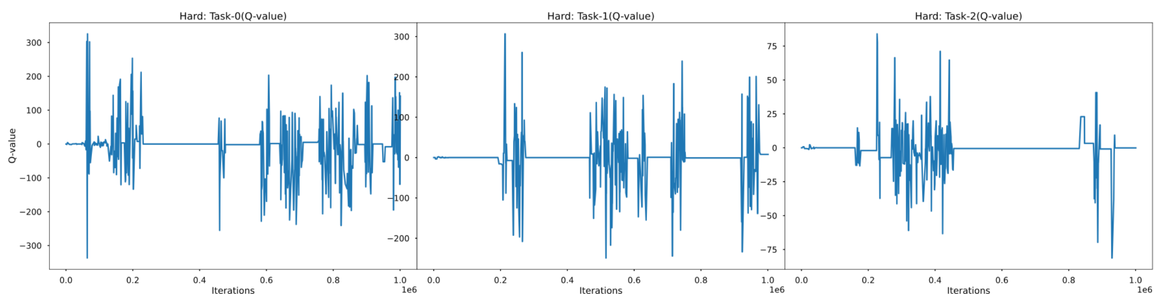

Figure 7.25: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with random pipe placement

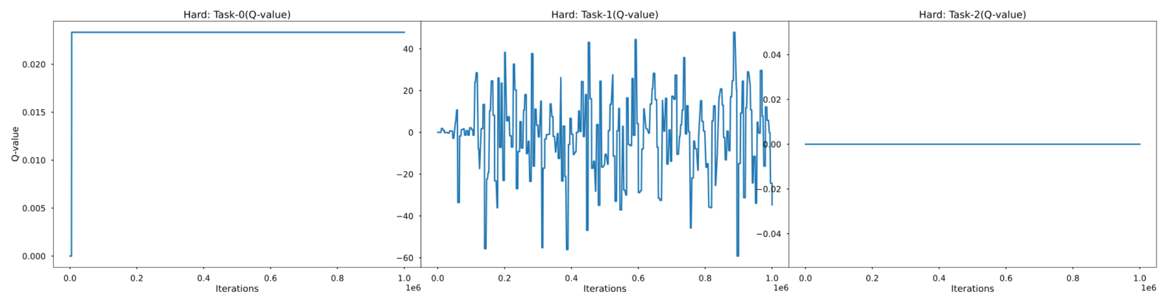

Figure 7.26: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with random pipe placement

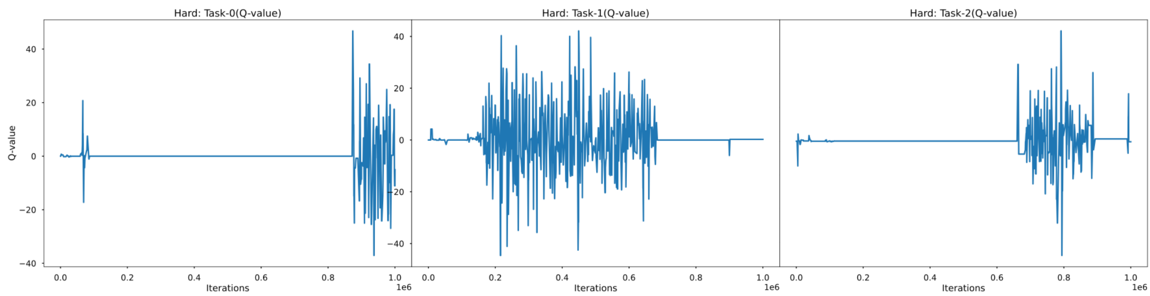

Figure 7.27: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with random pipe placement

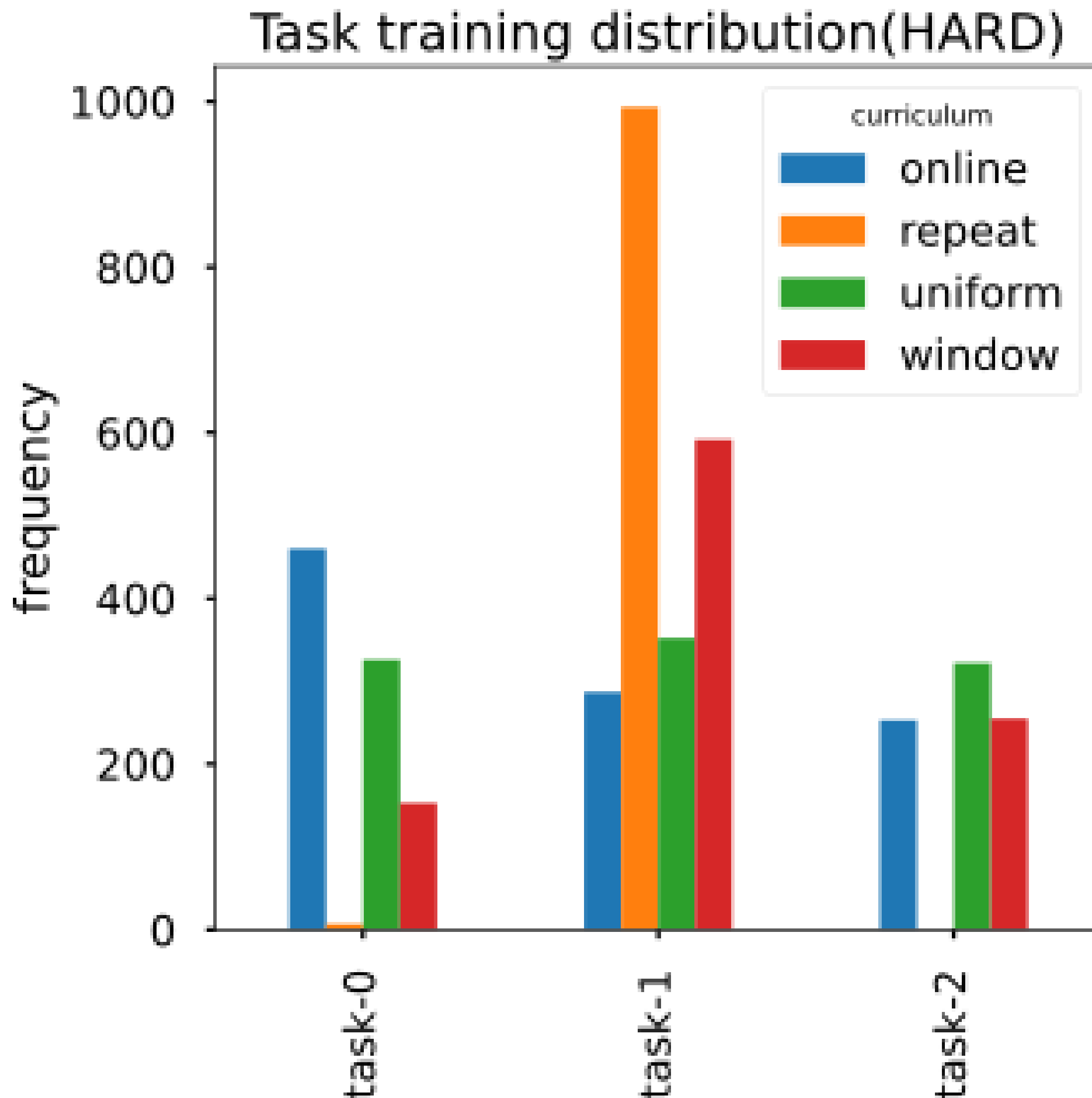

Figure 7.28: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with random pipe placement

7.4 Experiment - 4

This section entails the experiments carried out by decoupling the replay bu er from our automated curriculum learning framework to make the algorithm more general for any RL methods. The following figures are from the experiments run on our FlappyBird-LIDAR environment.

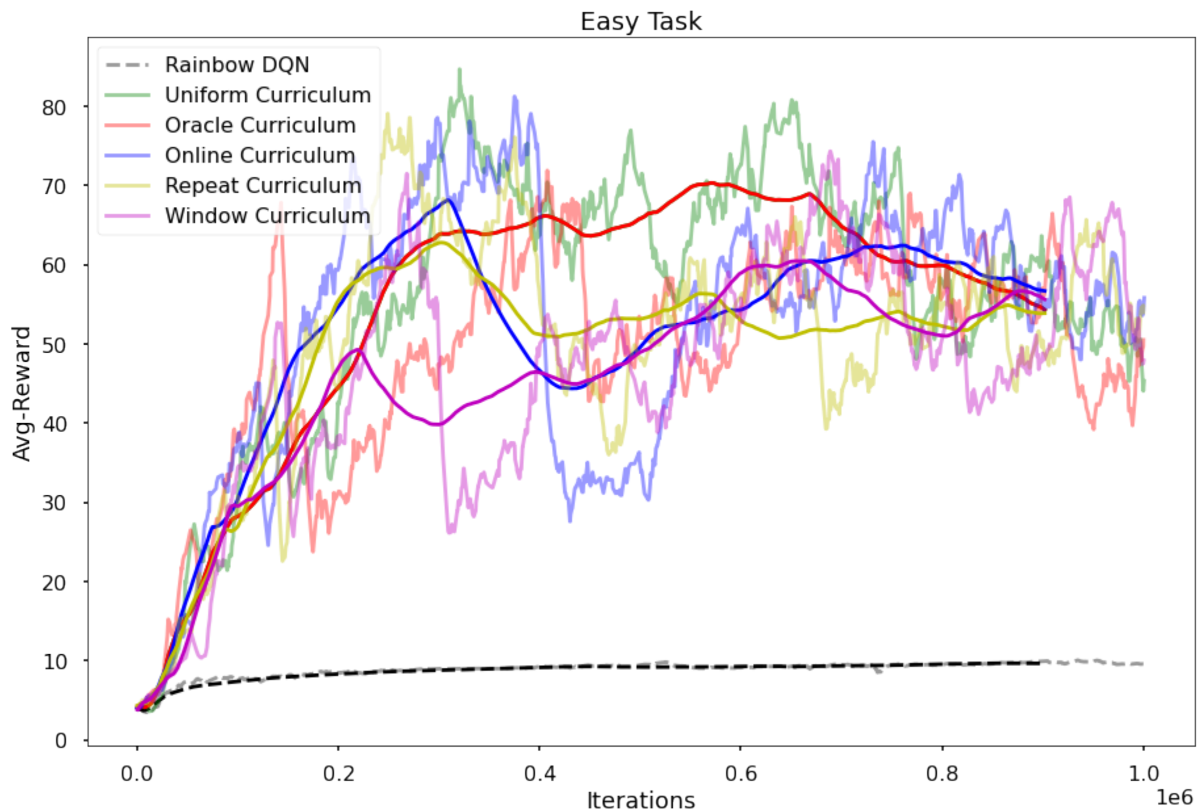

Figure 7.29: Cumulative episodic reward for agents trained on the same environment with uniform pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

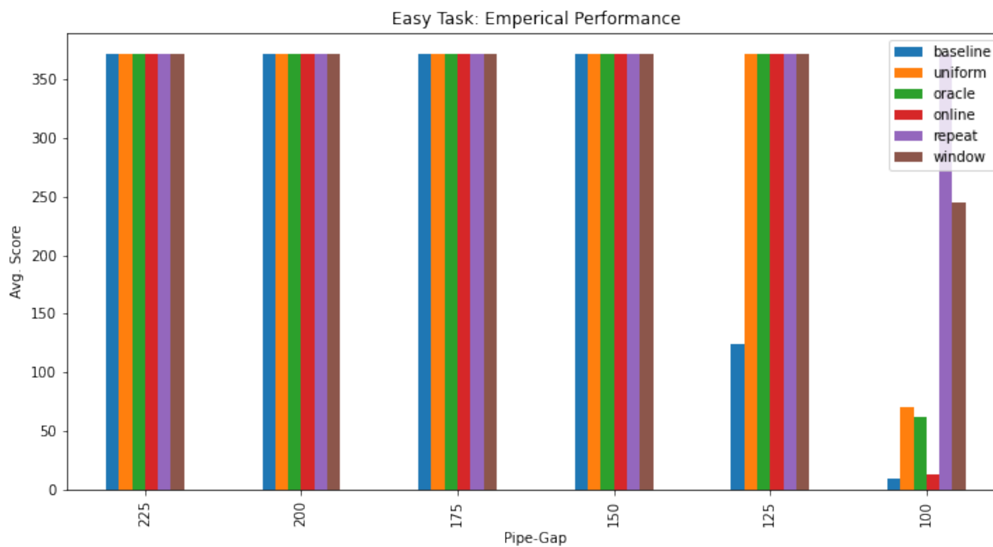

Figure 7.30: Emperical performance of the agents employed with di erent curriculum strategies on environments with uniform pipe placement for di erent PCG parameter(pipe-gap).

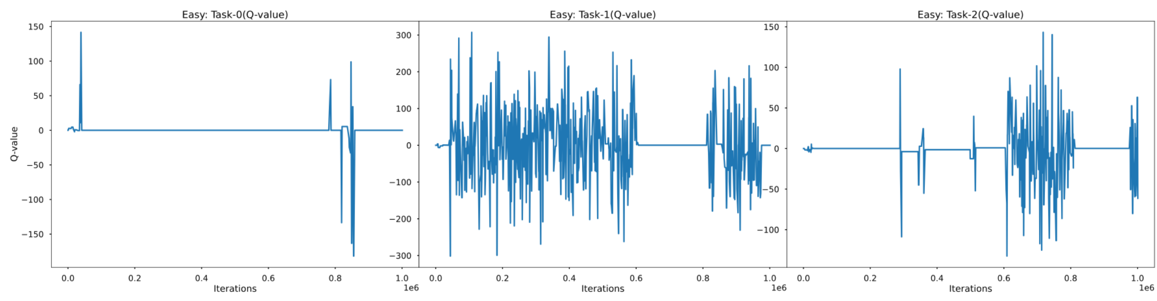

Figure 7.31: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with uniform pipe placement

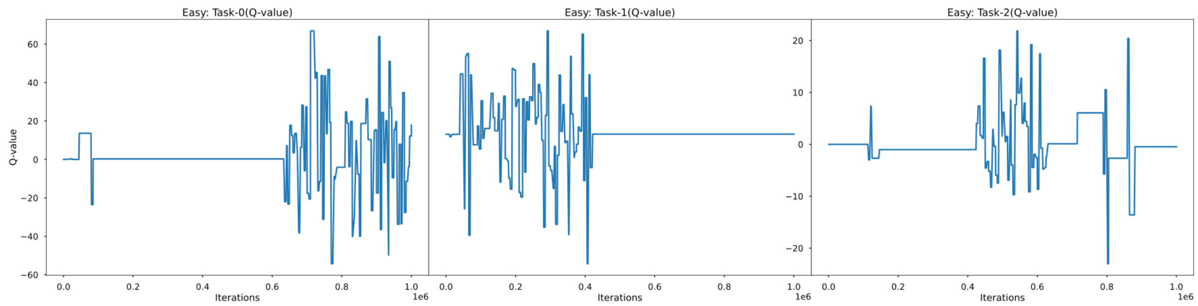

Figure 7.32: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with uniform pipe placement

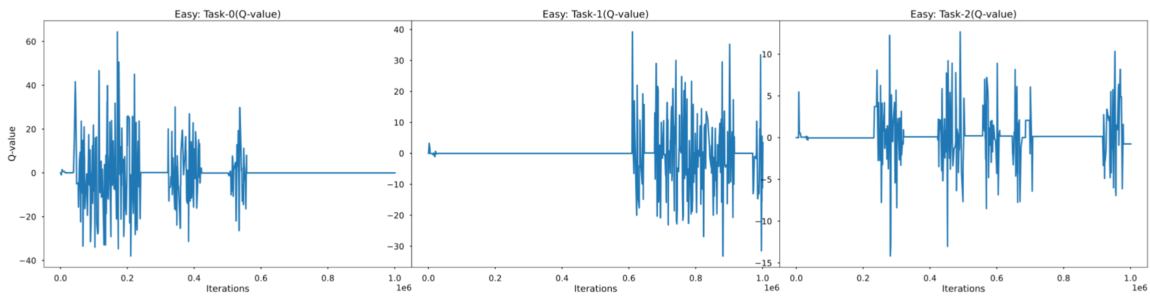

Figure 7.33: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with uniform pipe placement

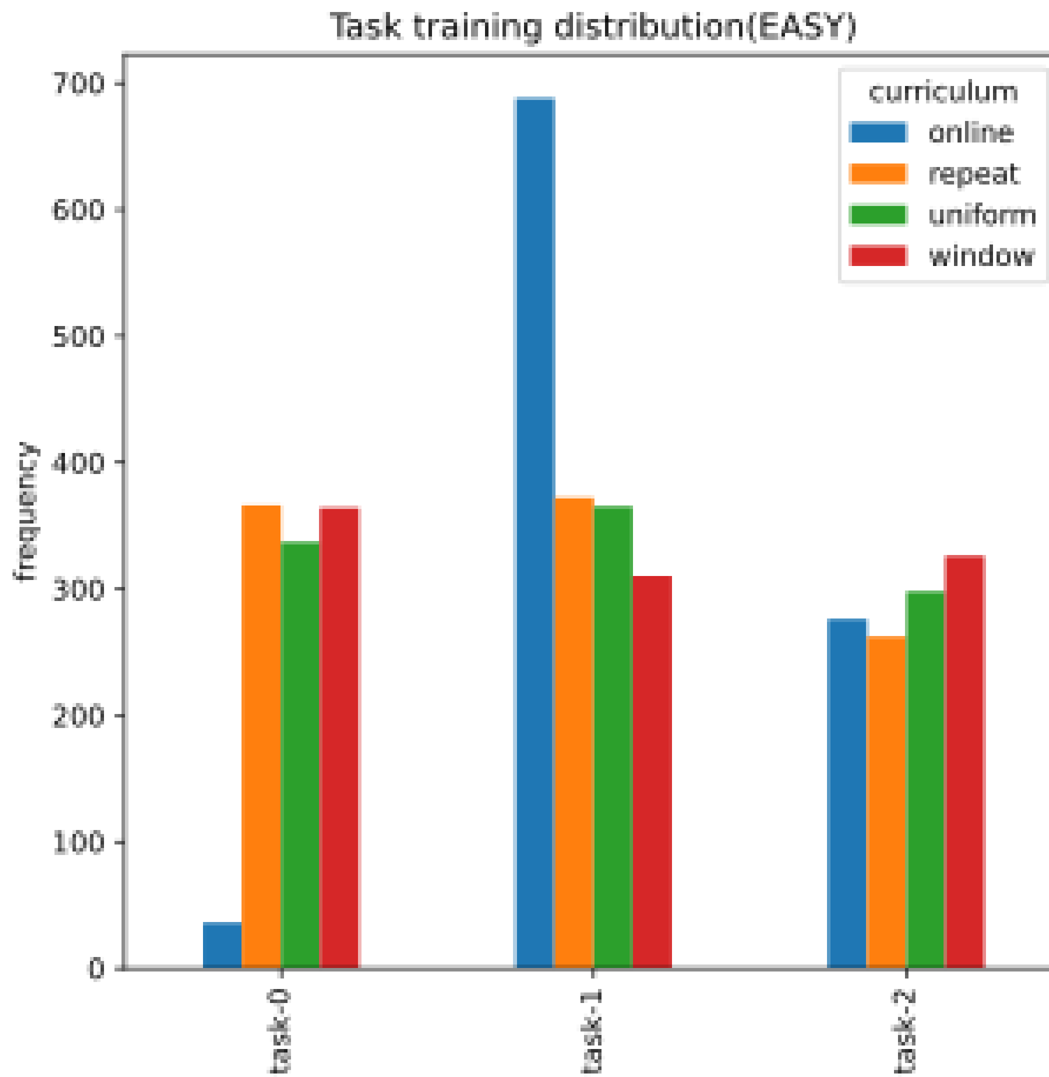

Figure 7.34: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with uniform pipe placement

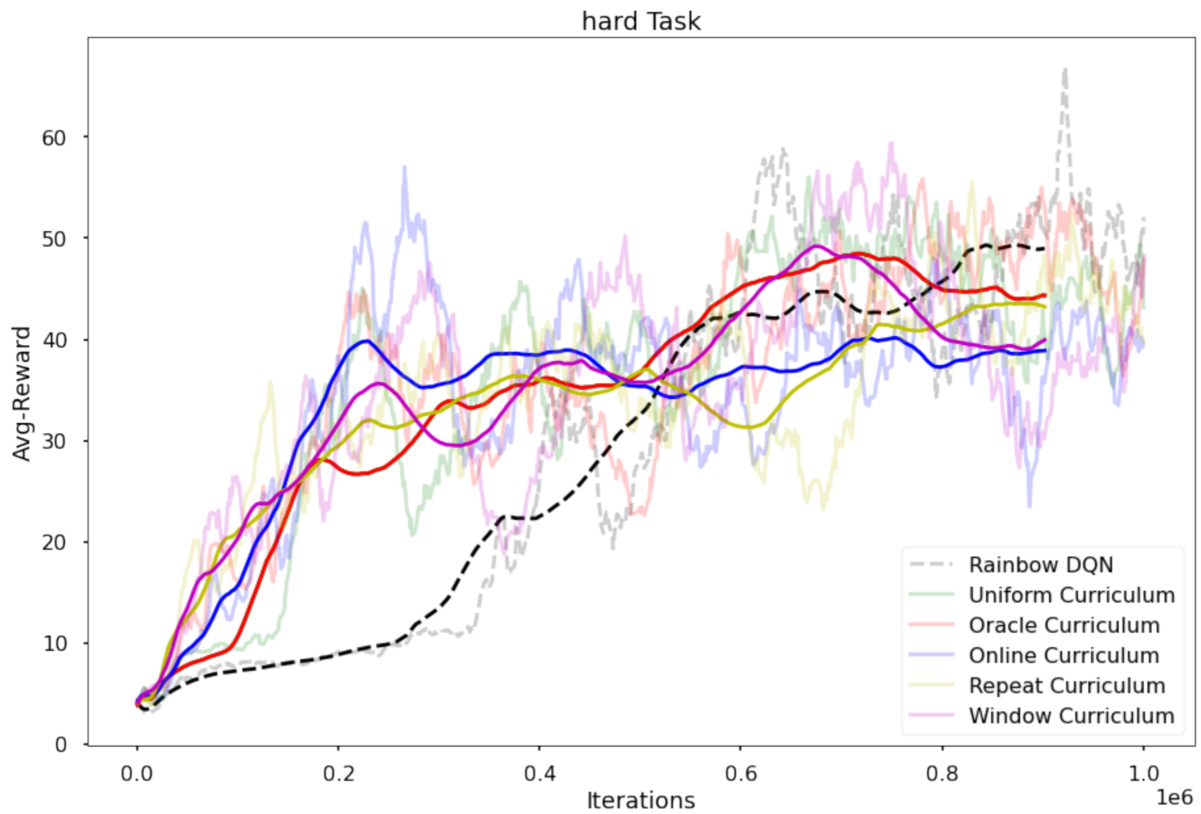

Figure 7.35: Cumulative episodic reward for agents trained on the same environment with random pipe placements with a pipe gap of 125 employed with di erent curriculum learning algorithms.

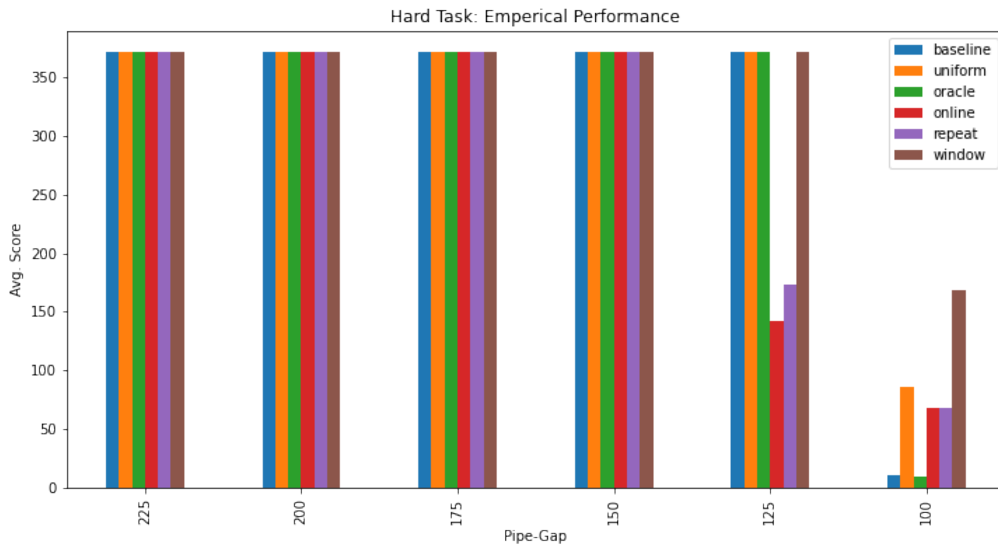

Figure 7.36: Emperical performance of the agents employed with di erent curriculum strategies on environments with random pipe placement for di erent PCG parameter(pipe-gap).

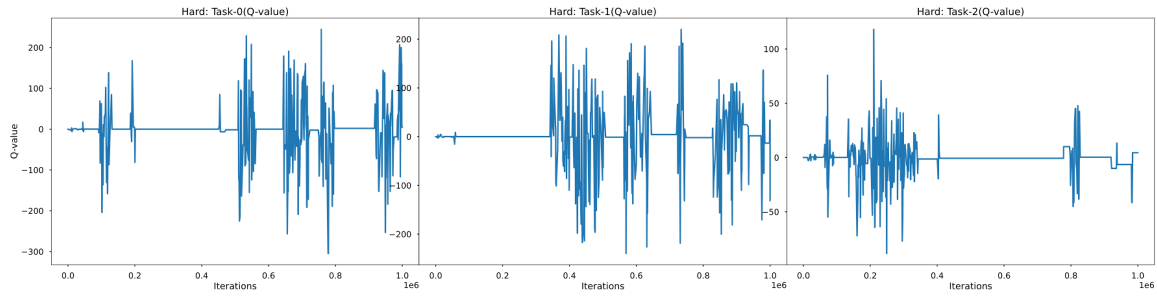

Figure 7.37: Q-values for each task during the task selection cycle for online curriculum scheduler on the environment with random pipe placement

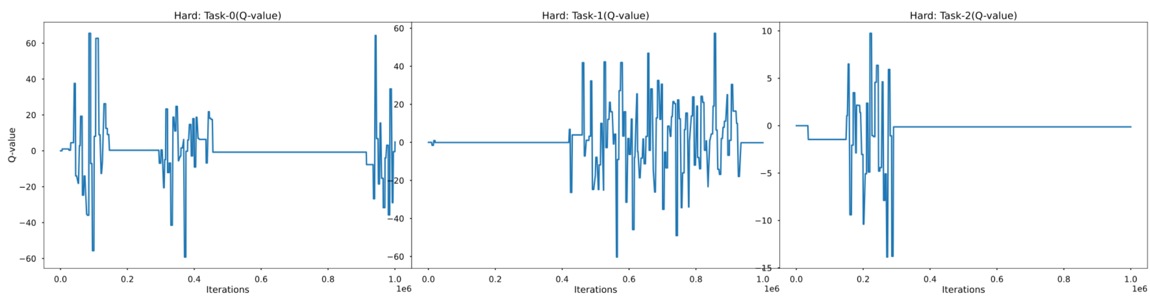

Figure 7.38: Q-values for each task during the task selection cycle for repeat curriculum scheduler on the environment with random pipe placement

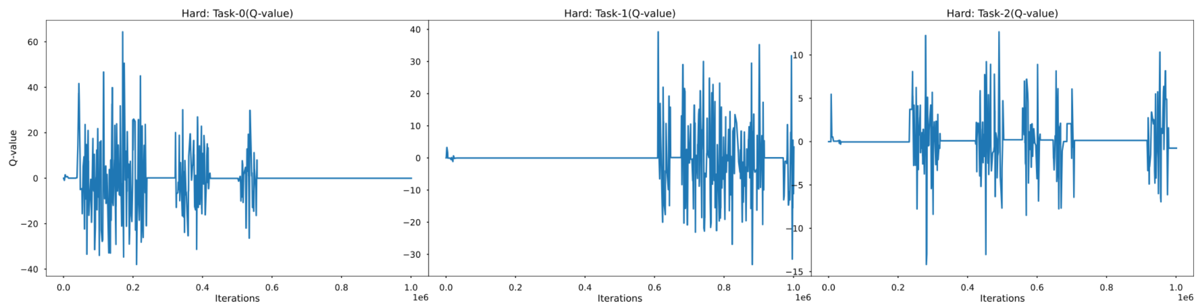

Figure 7.39: Q-values for each task during the task selection cycle for window curriculum scheduler on the environment with random pipe placement

Figure 7.40: Distribution of tasks selected during training phase for automated curriculum algorithms on the environment with random pipe placement