

Veeti Hartikainen

SELF-ONNS IN DEEP REINFORCEMENT LEARNING

Faculty of Information Technology and Communication Sciences
Master's thesis
January 2025

ABSTRACT

Veeti Hartikainen: Self-ONNs in deep reinforcement learning

Masters' thesis

Tampere University

Master of electrical engineering

January 2025

Self-organized Operational Neural Networks (Self-ONNs) have been recently introduced as an alternative to Convolutional Neural Networks (CNNs), as a way to introduce non-linearity to the network. Self-ONNs use generative neurons which use some higher degree functions to estimate the data distributions or patterns in the input data. Comparably CNNs can only estimate these patterns with linear or first-degree functions which hinders the networks' ability to learn more complex patterns. Self-ONNs have been used in multiple different use cases and they have been shown to perform better than equivalent CNN networks in tasks, such as image denoising and image transformation.

In this paper I study the possibility of integrating Self-ONNs into deep reinforcement learning networks to see whether they increase the performance of said networks. An equal sized CNN and self-ONN network is placed as a feature extraction network into a deep reinforcement learning network, which uses advantage actor-critic (A2C) algorithm, which is then taught to play ten different Atari 2600 games. The performances of these two networks are then compared to each other.

The test results seem to indicate that there is no benefit in using the self-ONNs in this case, though something that needs to be pointed out is the obviously too small number of frames used to teach the networks these games, as out of ten games only in two of them did the network both see substantial amount of learning as well as achieve average performance which went clearly over a random actor, which is the baseline model in this case.

The originality of this thesis has been checked using the Turnitin Originality Check service.

TIIVISTELMÄ

Veeti Hartikainen: Self-ONNs in deep reinforcement learning
Diplomityö
Tampereen yliopisto
Sähkötekniikan diplomi-insinööri tutkinto
Tammikuu 2025

Itsenäisesti organisoituvat operationaaliset neuroverkot (Self-organized Operational Neural Networks / Self-ONNs) on hiljattain esitelty vaihtoehtona konvoluutioneuroverkoille (Convolutional Neural Networks / CNNs), tuomaan epälineaarisuutta neuroverkkoihin. Self-ONN-verkot käyttävät generatiivisia neuroneita, jotka hyödyntävät korkeamman asteen funktioita arvioidakseen datan jakautumista tai kuvioita syöttö datassa. Vastaavasti CNN-verkot voivat arvioida näitä kuvioita vain lineaarisilla tai ensimmäisen asteen funktioilla, mikä rajoittaa verkkojen kykyä oppia monimutkaisempia kuvioita. Self-ONN-verkkoja on käytetty monissa eri käyttötapauksissa, ja niiden on osoitettu suoriutuvan paremmin kuin vastaavat CNN-verkot tehtävissä, kuten kuvan kohinan poistossa ja kuvan muunnoksissa.

Tässä tutkimuksessa tutkin mahdollisuutta integroida ne syvävahvistusoppimisverkkoihin, jotta selviäisi, parantaako ne näiden verkkojen suorituskykyä. Samankokoiset CNN- ja Self-ONN-verkot asetetaan ominaisuuksien poimintaverkoksi syvävahvistusoppimisverkkoon, joka käyttää advantage actor-critic (A2C) -algoritmi, joka sitten opetetaan pelaamaan kymmentä eri Atari 2600 -peliä. Näiden kahden verkon suorituskykyjä verrataan sitten keskenään.

Testien tulokset näyttävät viittaavan siihen, ettei Self-ONN-verkkojen käytöstä ole hyötyä tässä tapauksessa. On kuitenkin huomattava, että pelien opettamiseen käytetyn syöttödatan määrä oli selvästi liian pieni, sillä kymmenestä pelistä vain kahdessa, saavutti syvävahvistusoppimisverkko merkittävää oppimista sekä keskimääräisen suorituskyvyn, joka ylitti selvästi satunnaisen toimijan, mikä tässä tapauksessa määrittää perussuorituskyvyn.

USE OF AI IN THESIS

I have utilised AI tools in my thesis:

- No
- Yes

The AI tools utilised in my thesis and their purposes are described below:

Names and versions of AI tools: None

Purpose of using AI tools: None

Sections where AI tools were used: None

I acknowledge that I am fully responsible for the entire content of my thesis, including the parts generated by AI, and accept accountability for any violations of ethical standards in publications.

PREFACE

Even though the results achieved were contradictory to what I had thought they would be. As well as the time required to perform the simulations were much longer than expected, and therefore not being able to do as extensive simulations as I would have hoped. Finding out how to utilize the python libraries to perform deep reinforcement learning as well as how to integrate self-ONNs into pre-existing networks was interesting. So, all in all it was a good learning experience even though the results were surprising. This work was written during the time from May till January.

In Tampere, 29.1.2025

Veeti Hartikainen

CONTENTS

1. INTRODUCTION	1
2. PREVIOUS WORK	3
2.1 Deep reinforcement learning.....	3
2.2 Self-organized Neural Networks	8
3. SELF-ONNS FOR DEEP REINFORCEMENT LEARNING	10
4. EXPERIMENTS	12
5. RESULTS	14
5.1 Breakout.....	14
5.2 Ms. Pacman.....	22
5.3 Other Atari games	27
6. CONCLUSIONS.....	44
7. FUTURE WORK	46
REFERENCES.....	47
A PYTHON CODE TEMPLATE FOR TRAINING THE NETWORKS	49
B PYTHON CODE TEMPLATE FOR VISUALIZING THE RESULTS	52
C PYTHON CODE TEMPLATE TO EVALUATE RANDOM ACTOR FOR A GAME	57

LIST OF SYMBOLS AND ABBREVIATIONS

A2C	Advantage actor-critic
A3C	Asynchronous A2C
CNN	Convolutional Neural Network
DL	Deep Learning
DQN	Deep Q-network
DRL	Deep reinforcement learning
ONN	Operational neural network
ReLU	Rectified linear unit
RL	Reinforcement learning
SAC	Soft actor-critic
Self-ONN	Self-organized Operational Neural Network
TD	Temporal Difference

1. INTRODUCTION

As a successor to *reinforcement learning* (RL), *deep reinforcement learning* (DRL) seeks to integrate the advances in *deep learning* (DL) into RL to enable the agents to take actions in environments with high dimensional action- and observation spaces. This is especially useful when using high dimensional input data such as raw pixel data to train the agent some policy according to which it should act. This is preferable as image pixel data is often easy to access and available without any further implementation.

Most of the advances done in DRL focus on making the algorithms better by introducing things like prioritized experience replay [1], noisy networks [2] and distributional RL [3]. Examples of recent new algorithms in the field of DRL also include A3C (asynchronous advantage actor-critic) [4] and SAC (soft actor-critic) [5]. These algorithms seek to implement different methods to optimize the learning process. Prioritized experience replay does this through prioritizing experience to replay important transitions more frequently to add their significance. SAC instead aims to maximize expected value as well as entropy so that the agent is more equipped to act in environments that contain large amounts of randomness such as many real-world robotics cases.

As the tasks, DRL is used to tackle, get more complicated, the more it will matter that the networks used can take higher resolution images in, as some features are lost when resolution is reduced. The increased resolution can bring up smaller features which in some cases can be very useful to capture. As the resolution increases so will the visual complexity in the input images. The standard *Convolutional Neural Network* (CNN) models that are used to produce feature vectors from the input images might not be the best for this task as they are limited by the inherent linearity that the neurons in these models exhibit.

Self-organized Operational Neural Networks (Self-ONNs) have recently been introduced as a way to capture non-linear patterns in the data. The problem of neurons in CNNs only being able to capture linear patterns can now be solved by using generative neurons that are able to learn high degree functions instead of just linear functions. Their concept has been demonstrated to work for various tasks such as real-world blind image

denoising [6], handwritten text recognition [7], real-time glaucoma detection from digital fundus images [8], and severe image restoration problems [9].

Although self-ONNs have been very successful in these various tasks where they have been used in, being demonstrated to be better than CNNs and other state of the art models, the possibility of integrating them into DRL models has not yet been studied. In this paper I propose replacing the standard CNN layers used in DRL models with self-ONNs to enable the DRL models to learn better features from the pixel data given.

Comparisons in this paper will be done using the same Advantage Actor-Critic (A2C) algorithm by replacing the feature extraction network with custom networks of the same size one using CNN layers and one using self-ONN layers. The performances of these two different networks are then compared to each other on ten different games out of which the results for two are used in a more in-depth analysis. It is shown in the results that using self-ONN in this case does not provide any benefit over using CNN. On the contrary, self-ONN based models seem to generally perform worse and take substantially longer to train.

The structure of rest of the paper is as follows: First in the Section 2 previously done work on the fields of DRL and self-ONNs is briefly introduced. In Section 3 the basic idea behind the integration of self-ONNs into DRL, and why performing such integration could be useful, is explained. In Section 4 general information on the experiments done is presented, such as libraries used, and the hardware specifications of the PC used for simulations. Section 5 consists of all the results gotten from all the different simulations. It consists of four sub sections two of which are dedicated to closer observation of the performance difference in two different games, while the last two subsection presents all the results achieved in every game tested. Section 6 presents the conclusions that can be drawn from the results and in Section 7 possible future work that could be done on the topic is discussed.

2. PREVIOUS WORK

2.1 Deep reinforcement learning

DRL combines the principles of RL with DL, enabling agents to learn from high dimensional inputs such as images. The usage of neural networks in RL was originally explored in 1990s in papers like Tesauro's paper (Temporal Difference Learning and TD-Gammon), where the neural network was used to learn the evaluation function for the game of backgammon by playing against itself. The paper was successful in this and showed great promise by learning to play backgammon using neural network as its evaluation function. Though computation power at the time was not great which hindered the results. [10]

Next big advancement in the field of DRL happened 2015 when paper from Mnih, Kavukcuoglu and Silve (Human-level control through DRL) came out. In this paper a *deep Q-network* (DQN) was introduced. DQNs use deep convolutional network to learn action-value function or Q function which tells the reward given by every action. The DQN was demonstrated to be able to learn Atari 2600 games well enough to produce better results than previous methods in 42 of the 49 games tested, as well as being able to perform above human pro level in 29 games. What made this paper truly revolutionary however was the fact that the network learned purely from sensory input i.e. the pixel values on the screen. Instead of having to manually choose and hard code the features that should be tracked the network would learn to track these features on its own. [11]

Following the introduction of DQN, Google DeepMind published another paper on double Q-learning which further improved on its predecessor DQN by reducing the overestimation of action values, which DQN is subject to, by decoupling the action selection from Q-value estimation. [12]. Another paper which successfully was able to improve on DQN also published by Google DeepMind used prioritized experience replay. Key idea in this paper was to show that some transitions are more effective in the RL agents learning than others and that prioritizing the transitions which are more effective for learning produce better results than uniform prioritization of all transitions. This

method was able to improve the results on 41 of the 49 games tested when compared to the original DQN results. [1]

Apart from algorithmic improvements, new methods for DRL have also been introduced. Instead of providing faster learning these different methods are usually chosen based on case-to-case basis.

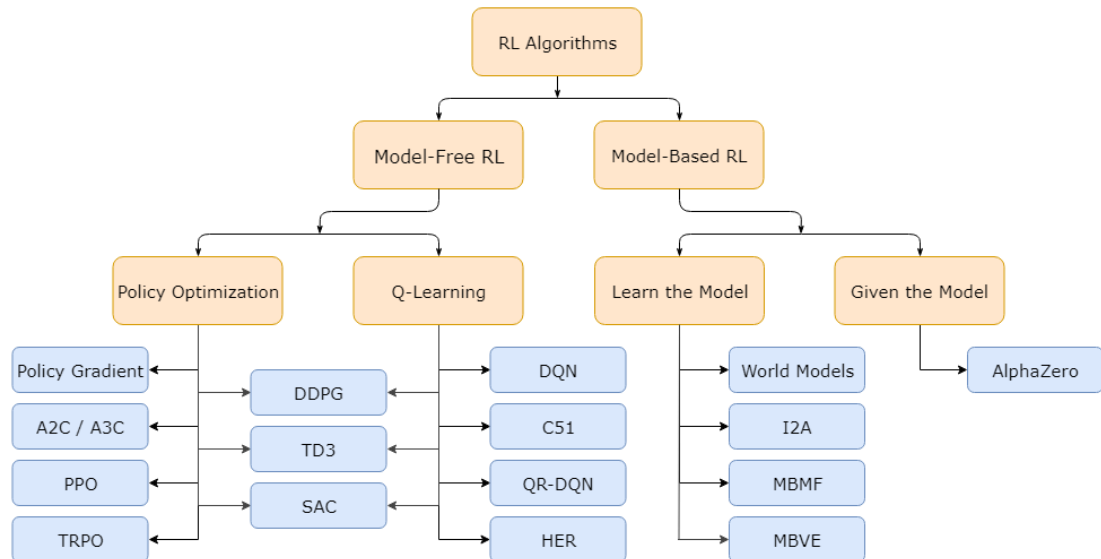


Figure 1. RL algorithms and methods [13]

Figure 1 shows how the RL algorithms are divided into model-free- and model-based RL. Model-free RL has no built-in model and only learns through direct interactions with the environment while model-based RL is either given a model or it learns the model through interacting with the environment. Model-based RL has the advantage of being able to predict the outcome of the moves it makes by simulating the continuation. This decreases the number of interactions that the model needs to take in order to learn how to act well in the environment however it increases the computational requirements as the simulations can be quite computationally heavy. A good use case for model-based RL is to use it in robotics and real-world physical environments where actions can be slow due to physical environments and the agent will not get enough usable training data without the use of simulations. In these cases, model-based RL methods can be utilized to learn, how the physical robot would move given a certain action and so the agent can also learn from the simulations the network creates internally.

There are three types of model-free RL methods, policy based (policy optimization in Figure 1), value based (Q-learning in Figure 1), and actor-critic (intersection between policy optimization and Q-learning in Figure 1). The already introduced DQN is value-based which means that the objective of the network is to learn the action-value function,

sometimes also referred to as value-utility function, which estimates the future rewards for the state-action pairs. The estimated action-value function can be expressed as:

$$Q(s, a). \quad (1)$$

Bellman optimality equation describes the constraint equation that must be true when the Q values are correct. Bellman optimality equation for Q-learning can be expressed as:

$$Q_*(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a'). \quad (2)$$

$R(s)$ is the immediate reward of the state s . Sometimes though $R(s)$ is replaced with $R(s, a)$ which expresses the immediate reward of the state s after taking the action a . Which expression is used usually depends on the problem or game that the function is used on as sometimes the reward may only dependant on the state the actor is in. γ is the discount factor which can be a value between $[0, 1]$ describing how much each step taken should reduce the subsequent reward. If discount factor value is set to 1 then any reward achieved at any point of the game are prioritized the same. If a policy that finds the fastest route to a goal in a labyrinth is to be sought after and the reward is only given in the end of the labyrinth and not in every state then discount factor needs to be set to a value lower than 1, as finding the fastest route might not get prioritized and the function would find any policy that finds the goal no matter how long it will take. $P(s'|s, a)$ describes the probability of transitioning to the next state s' after taking the action a . The fact that the transition is not guaranteed can be the consequence of various things. One such example can be noisy inputs, meaning an input can be read as a different input as intended due to noise. $\max_{a'} Q(s', a')$ is the maximum Q-value for the next state s' . Using

the bellman optimality equation outlined in Function (2) can the update function for Temporal Difference (TD) Q-learning be retrieved. TD learning is a method that updates the value estimates based on predicted and observed values after each action. TD

learning is designed to learn the optimal Q-function without requiring a model of the environment. The update function can be expressed as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)). \quad (3)$$

In this equation α describes learning rate, which controls how much new information is used to override the old information. The loss function of Q-learning is:

$$Q_{loss} = (r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))^2. \quad (4)$$

This function comes from mean square error loss. The mean square error is calculated between the target Q value ($r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')$) and the current estimate of the Q value $Q(s_t, a_t)$. There are four main steps involved in teaching a neural network, which are forward pass, loss computation, backpropagation, and parameter update. In reinforcement learning forward pass involves feeding the network the current state as input. After this the network will produce some output that depends on the input (current state) and the parameter values of the network. During loss calculation is when the loss function is needed. Different problems and applications of neural networks might require different loss functions in Q-learning the loss function used is the one presented in Equation (4). In backpropagation the loss function is used to calculate the gradients with respect to the weights and biases of the neurons in the network. In the parameter update step these calculated gradients are then used to update each neurons weight and bias. This way the network improves its estimation of the Q-function and make more well-informed actions in the future. The update function can then be expressed as [14]:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} Q_{loss}. \quad (5)$$

Alternatively, value-based RL methods can be used to learn the state-value function:

$$V(s). \quad (6)$$

While the state-value function gives the estimated return from state according to the policy the action value function estimates the return an action taken from a state following a policy gives. Similarly, to action-value based method state-value function also has

Bellman optimality function which should be at equilibrium when the optimal values are used. For state-value function the Bellman optimality function can be expressed as:

$$V_*(s) = \max_{a \in A} Q_*(s, a). \quad (7)$$

State-value function also has update functions such as action-value function did. There are multiple different methods of which two quite well known are, Monte Carlo method and the other is TD method. The update function can be expressed as:

$$V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t)), \quad (8)$$

when using the Monte Carlo method. G_t here refers to the accumulated reward at time step t . When using TD method state-value function can be expressed as:

$$V(s_t) \leftarrow V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t)). \quad (9)$$

The loss function can similarly be derived from mean square error for state-value function as it was in the case of action-value function.

$$V_{loss} = \frac{1}{2}(\hat{V}(s) - V(s))^2. \quad (10)$$

$\hat{V}(s)$ is the target value for state s and $V(s)$ is the current estimate of the value of state s . Depending on what method is used $\hat{V}(s)$ can be replaced. Using Monte Carlo method for example $\hat{V}(s)$ would be replaced with G_t and using TD method it would be replaced with $R_{t+1} + \gamma V(s_{t+1})$. [15]

Second method is policy-based which means that instead of estimating the return values of actions the policy is estimated directly. Policy-based methods estimate the policy function:

$$\pi(a|s). \quad (11)$$

This function defines the agent's behaviour, specifying the probability of taking the action a given the state s . Value based methods are generally simpler and well-established, they are effective in problems where optimal policy can be derived from the value functions. As a downside they struggle with large or continuous action spaces and require large amounts of exploration to converge. On the other hand, policy-based methods handle continuous and high-dimensional action spaces more efficiently and they also tend to converge faster to a sufficient policy in complex environments. As a

downside policy-based methods are more complex and computationally intensive compared to the value-based methods. The update function for policy function can be expressed as:

$$\theta_t \leftarrow \theta_t + \alpha \nabla \ln \pi(A_t | S_t, \theta_t) G_t, \quad (12)$$

where $G_t = \sum_{k=t+1}^T \gamma^{k-t-1} R_k$ and θ_t is a set of parameters gotten from parametrising the policy $\pi(a|s)$. [16]

The last method is actor-critic method which combines both value- and policy-based methods trying to get the benefits of both methods while mitigating some of the downsides. Actor-critic models consist of two separate models, actor which learns the policy-function, which dictates what actions to take and critic that learns the value function which estimates the reward for the actions taken by the actor.

One example of actor-critic method being used is A3C algorithm first introduced in the paper from Google DeepMind in 2016. Although it is marked as policy based in Figure 1 this is a mistake since it is stated to be actor-critic based method in the paper it is introduced in. In the paper this algorithm is compared to the previously mentioned DQN, and double Q-learning algorithms and it was able to produce better results in faster time compared to the previously mentioned algorithms. [4]

DRL has seen large number of different use cases. It has been used to master classical full information games like chess, shogi, and go [17], as well as more complex non-full information videogames such as Dota 2 and StarCraft II [18] [19]. Some real-world applications of DRL that have been researched include autonomous driving and robotics [20] [21] [22].

2.2 Self-organized Neural Networks

Self-ONNs have recently been introduced and demonstrated to perform better than equivalent CNNs in various tasks such as image denoising, image transformation, fault diagnosis of rotating machinery, and even ECG analysis [9] [23] [24] [25].

Self-ONNs use generative neurons instead of the used linear neurons commonly used in CNNs. These generative neurons can change and optimize the nodal operator for each connection during the training. The optimal nodal operator for each connection is estimated using Q:th order truncated Taylor approximation. This simplifies to Maclaurin

series when the input is bounded to the range of $[-1, 1]$. This leads to the function below which describes one connection between two neurons:

$$\Psi(\mathbf{w}, y) = w_0 + w_1y + w_2y^2 + \dots + w_Qy^Q. \quad (13)$$

In Function (13) y is a single input to the neuron while \mathbf{w} values are the weights. Also w_0 can be omitted since it will be compensated by each neuron's bias. Compare this to Function (14) which represents the connection in plain CNN structure:

$$y = wx. \quad (14)$$

In Function (14) x represents the input to a single neuron while w is weight. Function (13) will be the same as Function (14) for Q order of 1. This means that self-ONNs are a superset of CNNs. Even self-ONNs with higher Q order can form linear connections by setting the weights of any higher order input values to zero. [9]

The nonlinearity of the connections and the heterogeneity of the network, which results from the possibility of having different power functions, lead to the network being able to generalize better and having the ability to adapt to more complex data. Higher Q order functions do however require more weights or trainable parameters per neuron, which leads to higher computation time. Still however when factoring the number of parameters, by increasing the number of neurons per layer for CNNs, self-ONNs are still demonstrated to outperform CNNs with as many parameters in the various tasks they have been implemented on.

3. SELF-ONNS FOR DEEP REINFORCEMENT LEARNING

Advancements in DRL have mainly focused on coming up with new algorithms, then refining and combining them, as well as developing new methods that better fit specific tasks which were described more in-depth in Section 2.1 i.e. policy-, value-, and actor-critic methods.

Some examples of algorithmic advancements are the double DQN also introduced in Section 2.1 which addresses the overestimation of bias in Q-learning [12]. Another such example is Distributional Q-learning [3] which learns value distribution. Learning the value distribution is demonstrated to matter in this paper because it among other things, reduces chattering which means that it helps the network to converge to a policy, provides richer set of predictions and has well-behaved optimization. In a 2017 paper from Google DeepMind bunch of these algorithmic advancements were combined to develop Rainbow algorithm which successfully beat every individual algorithm, that were combined to create Rainbow algorithm, by a large margin [26].

There has not yet been much exploration on the possibility of integrating self-ONNs into DRL however, very likely since they are a very new discovery in the field of neural networks. Since DRL uses CNNs to capture information from image and pixel data, it is subject to the same problem of homogeneity that CNNs used for image manipulation tasks are subject to. Self-ONNs have been demonstrated to increase the heterogeneity of the network by introducing generative neurons which are able to learn higher order functions instead of the usual linear functions that regular neurons in CNNs learn. The only non-linearity that CNNs have comes from the non-linear activation layers that use ReLU, tanh, sigmoid, or any other non-linear activation function.

As DRL is being used on more and more challenging tasks the visual information that is used in the models get more and more complex. The heterogeneity introduced by self-ONNs could prove to be quite useful in learning more complex patterns from the visual information it is given, possibly having the ability to capture better and more useful features that the basic CNNs cannot, due to the limitations of having only linear operations to use for its neurons. Self-ONNs are also said to generalize and avoid overfitting better than CNNs, both of which could improve the performance and the rate of learning of DRL models. Better generalization would be especially useful as in some cases the input images can vary a lot. For example, the videogame Dota 2 has 124

different heroes with distinct abilities with different effects and 208 different items that provide different improvements to the player. For the DRL model to learn all the important features from visual information only would be quite the task that requires a very sophisticated neural network the simplistic neurons of CNNs might not be able to learn all the features as well as self-ONNs.

In the OpenAI Five paper the features were predetermined by the researchers and not learned [18], which leads to extra work, that needs to be done by the researchers, as all the important features need to be defined. If the network could determine all the important features on its own, then it could just be simulated based on the pixel values similarly to how it was done for the Atari games in the DQN introductory paper [11].

The basic idea on how to integrate self-ONNs into DRL is very simple. DRL models that use image information as input data consist of two main parts, the feature extraction network, which as the name implies extracts the features the network deems important from the image data, and the action or policy network depending on which method is being used, which chooses the control inputs from the extracted features. The feature extraction network used is usually CNN while the action or policy network is usually fully connected neural network. In code the CNN of the feature extraction network can be replaced by a self-ONN easily by replacing the convolutional layers with self-ONN layers. The code for self-ONN layers has already been implemented and can be found in from GitHub [27] and more detailed explanation on the workings of this code as well as how to use it has also been provided [28].

In theory integrating the self-ONN in this way should provide the same benefits for the feature extraction network as it does for other tasks it has been used for. Examples of such benefits include enrichment of solution space as well as network heterogeneity. [9]

4. EXPERIMENTS

DRL model's efficacy is usually measured as average reward over some number of evaluation episodes as there is inherent randomness with respect to the performance of the model over a single episode due to the randomness of the starting position as well as some other possibly random events that occur in games. Therefore, in the experiments the models are evaluated over ten different evaluation episodes. Average reward or the overall performance of the model is then calculated using these scores.

There is also inherent randomness in the generation of the DRL models, which is caused by the randomness that occurs while teaching the model including but not limited to what data the model sees during the training and whether the model converges to another solution from the other models trained. This is why it is not enough to only generate one model with ONN and one with CNN as the randomness is inherent to the model generation might produce positive or negative results all on its own and would not be a proper proof of the concept. This is why every DRL model generated in the experiments are generated ten times and the average, mean, min, and max results of these ten different iterations of the same architecture are compared to the equivalent statistics generated by the other architecture's different iterations. This should eliminate some of the randomness and give more robust results.

On top of the average reward calculated over the episodes, the time required to teach each of the models are also kept track of. This is done because neurons in self-ONNs require more parameters than those of CNNs. The number of parameters that self-ONNs require is linearly dependent on the Q value which determines the maximum order of the functions used. I.e. network using self-ONN layers with Q value of 3 has three times more parameters than CNN with the same structure. The whole DRL network however does not have that many times more parameters as the action network is not replaced with generative neurons.

All codes are run on the same PC setup with 12th Gen Intel(R) Core(TM) i7-12700F Processor and NVIDIA GeForce RTX 3080 GPU. The median of the training times is used to compare the required time to train the models as there might be some inconsistency in the CPU usage which might lead to the code sometimes taking more or

possible less than normally. However, having to produce ten different iterations for each model and taking the median time should eliminate any unseemly large or small times.

Libraries and their versions used include:

- `gym 0.26.2`, which is used to gain access to `shapes.Box` to define the shape of the input to the neural network.
- `stable_baselines3 2.3.2`, where the ready-made functions, `A2C` (used for the ready given A2C algorithm with CNN policy), `VecFrameStack` (used to stack consecutive frames so that movement information can be used), `evaluate_policy` (for estimating the performance of the network), and `make_atari_env` (used to create a readily provided Atari 2600 game environments), are imported.
- `PyTorch 2.2.1` version is used to create the network layers of the custom CNN for the A2C algorithm.
- `Numpy 1.24.3` is used to save the rewards and training times for each of the models into numpy arrays. The saved numpy arrays are then later accessed by another code which creates a visual representation of the data.

Also, `FastONN` code implementation used is taken from the github page of Junaid Malik with GitHub tag of `junaikmalik09` [27]. This code provides the implementation for `SelfONN2d` which is the self-ONN layer used for the self-ONN networks that are used to replace the CNN network in the A2C algorithm. A2C algorithm using the CNN network and A2C algorithm using the Self-ONN network are compared in the results.

Each of the models are trained over 500 000 timesteps total the used environment uses frame skip value of 4 [29] which means that the game is taught on two million frames for comparison the rainbow model was taught on 200 million frames [26]. While this is way less data used for training due to the limited computation power and time it is a necessary evil.

In the end of this paper there are three appendices which include three different code templates which can be used to produce the results from this paper. Appendix A contains the code template for training the networks, appendix B contains the code template for visualization of the results, and appendix C contains the code template for random actor evaluations.

5. RESULTS

5.1 Breakout

The first game the networks are trained on is the classic Atari Breakout. In this game the player moves a small platform to bounce a moving ball to hit tiles on the screen. Each tile broken is one point for the player. Figure 2 demonstrates a possible state of the game Breakout. As can be seen from the image the score is 2 while two tiles are missing, and the player has three lives left.



Figure 2. Breakout

The feature extraction network is a CNN network consisting of three Convolutional 2D layers each followed by one ReLU activation layer. These are followed by a flattening layer which is followed by one linear layer followed by another ReLU activation layer to extract 512 different features from the input images. This linear layer and ReLU activation layer form the action network. The convolutional layers have kernel sizes of 11, 7, and 3 in that order with stride of 5,3, and 1 in that order. The first layer has input size of 4 which comes from the number of stacked layers other than that every layer has input and output size of 64. Learning rate is set to 0.0001 while discount factor is set to

1 since time spent does not matter in Breakout as much as the number of tiles broken. Self-ONN network is otherwise the same but SelfONN2d layers from the `fastonn` library are used instead of Conv2d layers from `torch.nn`. Two different models were trained for the self-ONN cases one with Q set to 3 and the other with Q set to 5. Q is the degree of the polynomial function generated for each neuron. Base model has 376 325 parameters, self-ONN with degree 3 has 913 413 parameters and self-ONN with degree of 5 has 1 450 501 parameters. The self-ONN models do not have exactly 3 and 5 times more parameters respectively because of the linear layers which all use the same basic type of neurons instead of the generative neurons which are used in the self-ONN layers. While average and mean can be used interchangeably and are usually used to mean the same thing as in arithmetic mean in the figures, they are used to describe different things in the following graphs. Mean is used to describe the arithmetic mean value of the rewards of the same model estimated over ten testing periods, while average is calculated using these mean values from multiple different models. Meaning average here means the arithmetic mean of mean rewards of all the trained models.

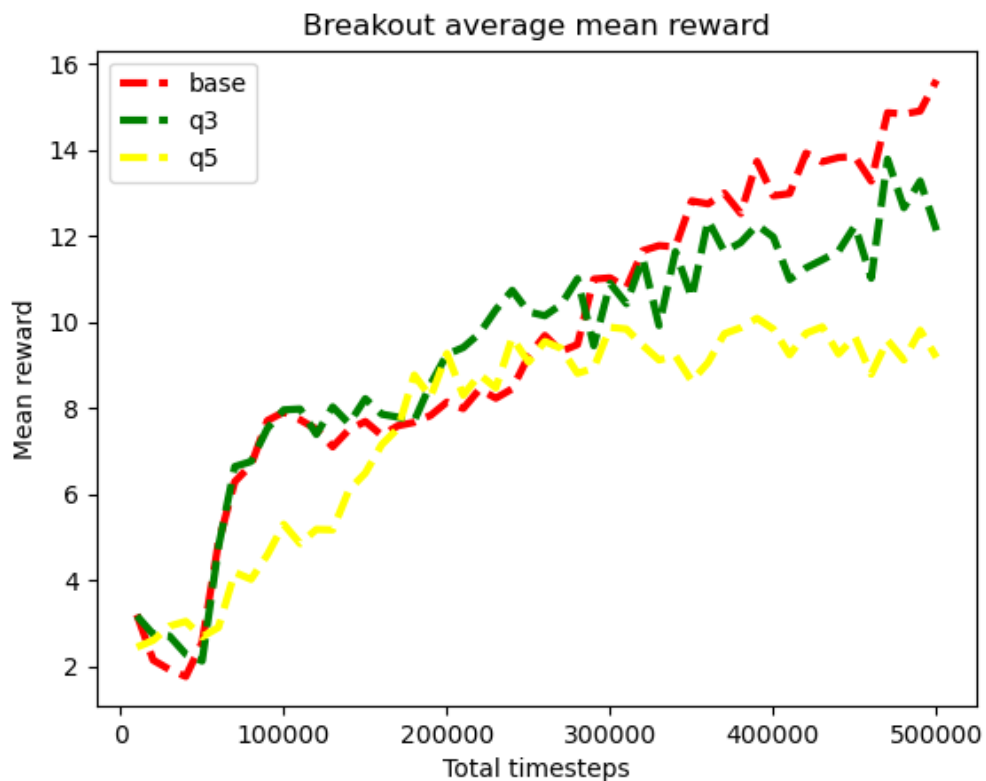


Figure 3. Breakout average mean reward of all the ten models trained for each base, q3 and q5 cases

In Figure 3 the average performance of all the trained models for each case (CNN based base, Self-ONN based q3 and q5) is shown as a function of total timesteps used to train the models. Unlike the initial belief of the Self-ONN providing an advantage over the base CNN in terms of added performance gained from the ability to better handle visual information. It seems that the opposite is true. The models generated by using self-ONN layers performed generally worse on average than the equivalent models using convolutional layers. To get a better understanding of why let us observe other graphs generated using the same models.

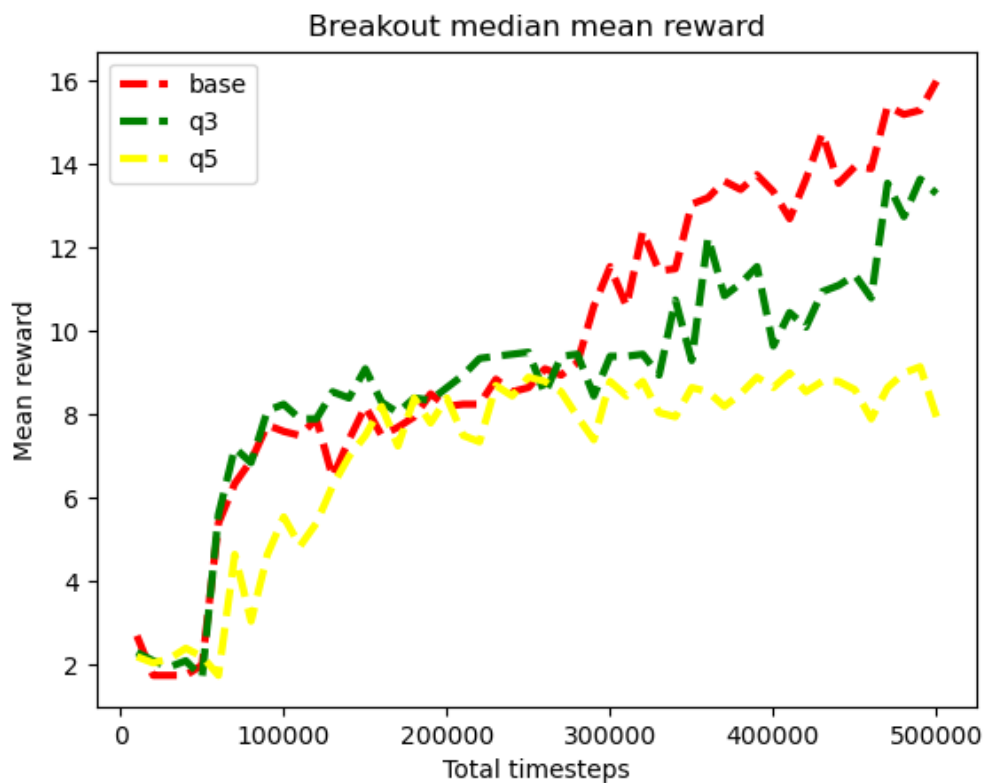


Figure 4. Median of all the mean rewards of all the ten cases

Figure 4 shows the median mean reward of all the generated models during each timestep. This graph closely resembles Figure 3 which means that the different models' performances are evenly distributed around the median performance of all the models. Another thing that can be seen from Figure 3 and Figure 4 is the fact that the higher degree self-ONN performs even worse than the lower degree self-ONN. This would suggest that the added complexity of the generative neurons produce do more harm than good in the learning process of the DRL network, at least in the case of learning Breakout. Some reasons why this might happen include possibly lower rate of convergence due to having higher parameter count and getting more easily stuck in local minima due to having more difficult to travel optimization landscape.

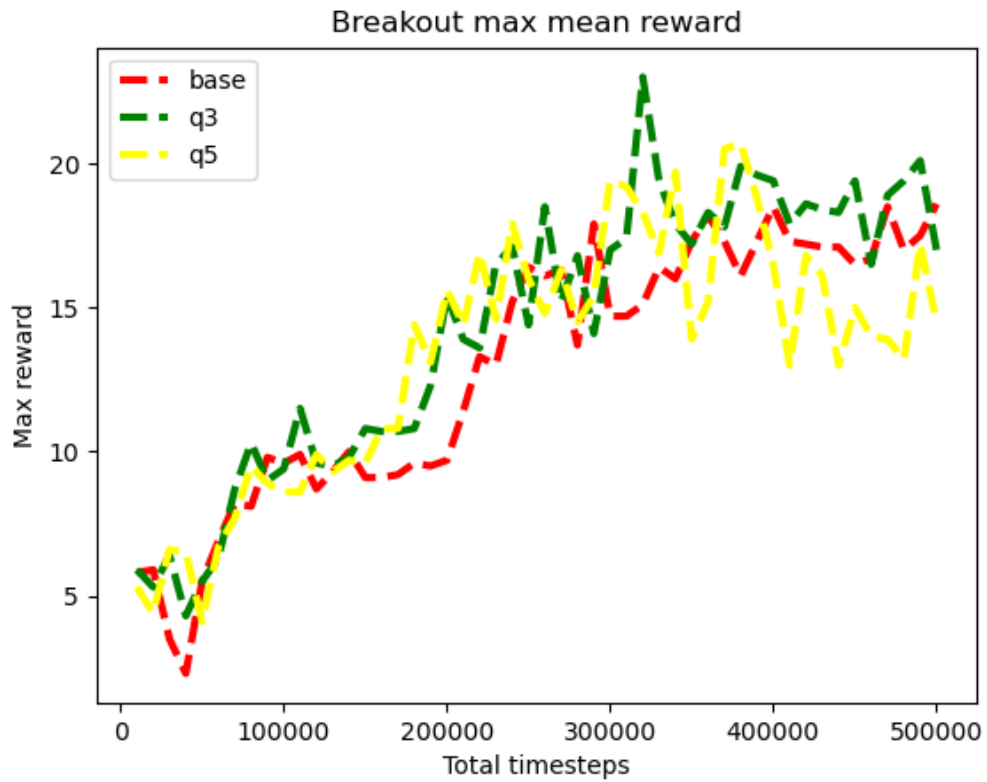


Figure 5. Maximum mean reward for each case generated by any model



Figure 6. Minimum mean reward for each case generated by any model

Figure 5 and Figure 6 show the maximum and minimum mean for each timestep among all the generated models. While the minimum mean rewards are worse for both models using self-ONNs the maximum mean rewards are pretty much equivalent to those of the base CNN architecture. While some variance is expected in rewards during different timesteps in training too high variance can be a sign of the model not converging to a solution and instead beginning to oscillate. This effect can be seen more clearly in Figure 7 and Figure 8 which show the performance curves for all the CNN and self-ONN based models. Not only is there visibly way more variance in the performance over consecutive training steps in the self-ONN based models, but they also seem to be more likely to stagnate to some level of performance and not improve much after. Both reasons lead to the fact that the self-ONN based models end up having more scattered end performance scores compared to the CNN based models.

Figure 9 and Figure 10 highlight the models that ended up having the best and worst final performance out of all the generated models. As can be seen from Figure 9 the best models generated for CNN and both third and fifth degree self-ONN based models all end up with similar performance and the slight differences can be appointed to margin of error or luck as the changes between two consecutive time steps are bigger than the performance difference between the different models. However, in the case of the worst performing models the self-ONN based models performed considerably worse than the CNN based model. In Figure 10 the self-ONN based architecture with the degree of three also sees no clear improvement between the starting and ending values and instead shows clear signs of oscillation as the performance score jumps around the value 6.

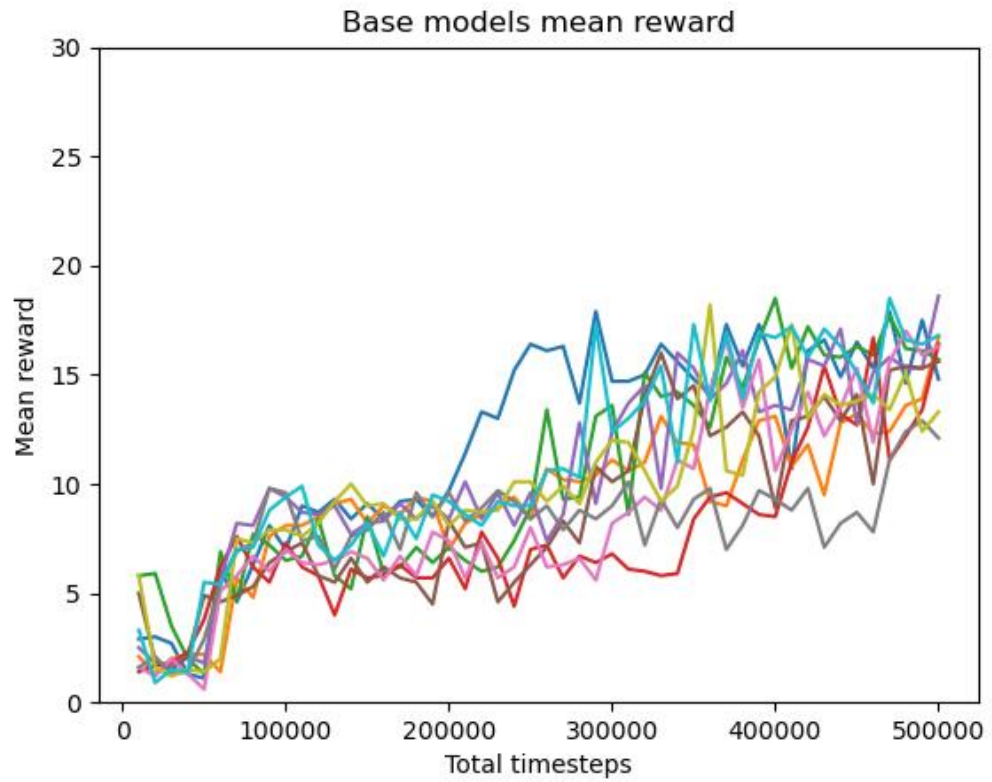


Figure 7. Performance curves for all the base models

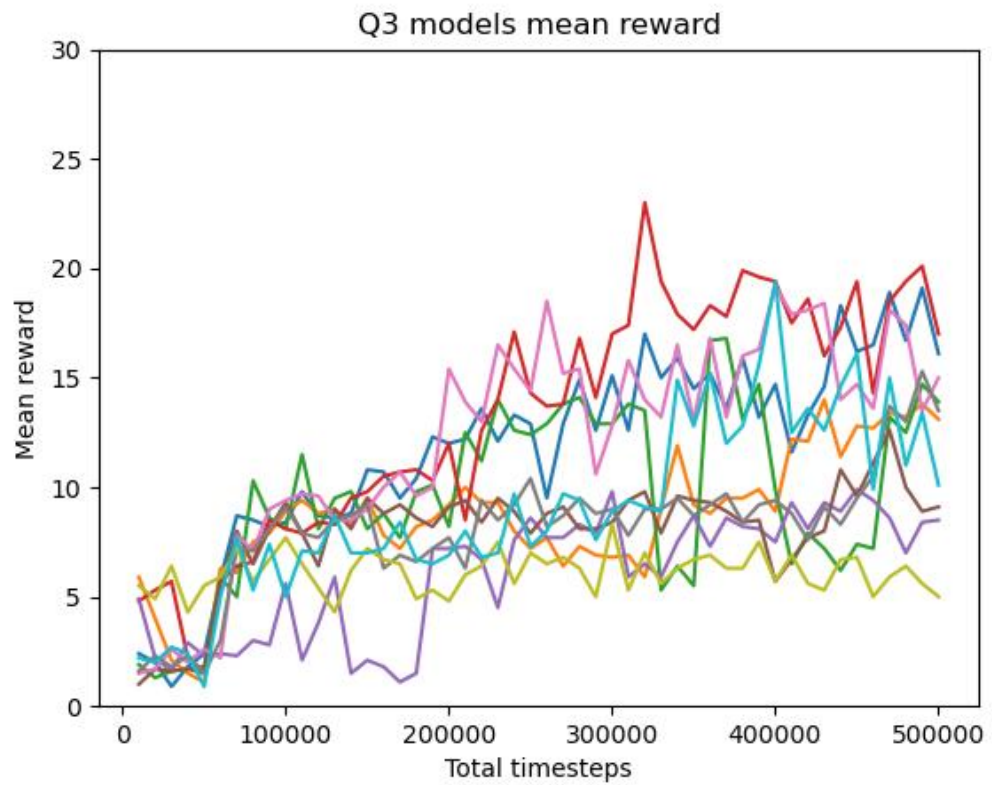


Figure 8. Performance curves for all the third-degree self-ONN models

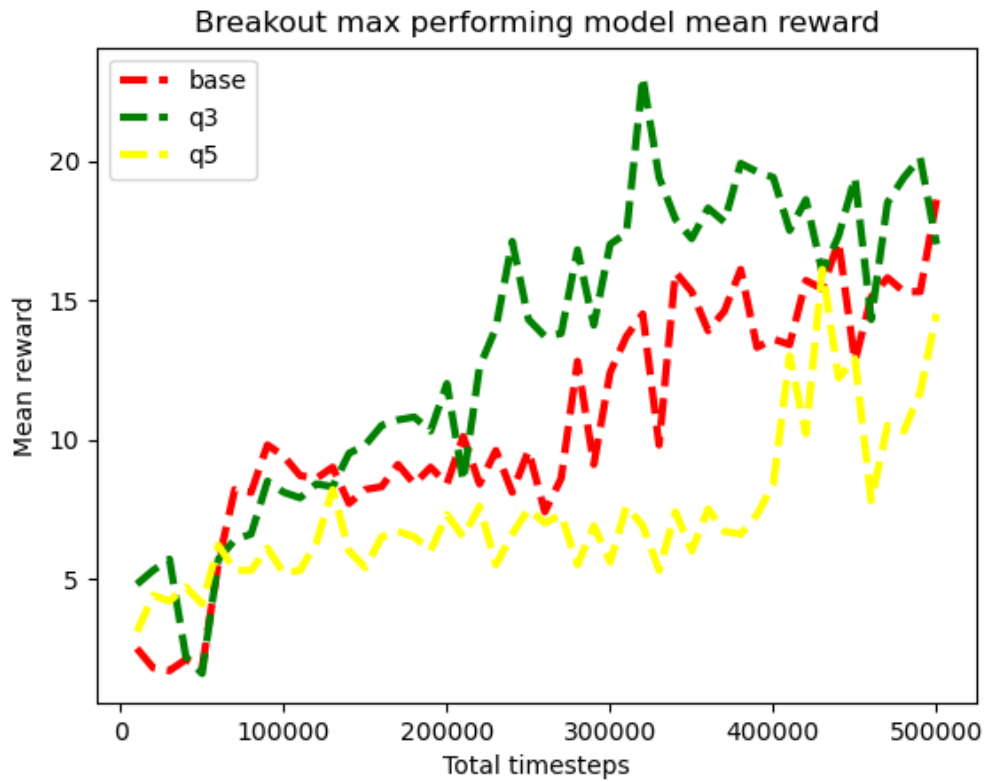


Figure 9. Models with best final score out of the ten generated models for each case

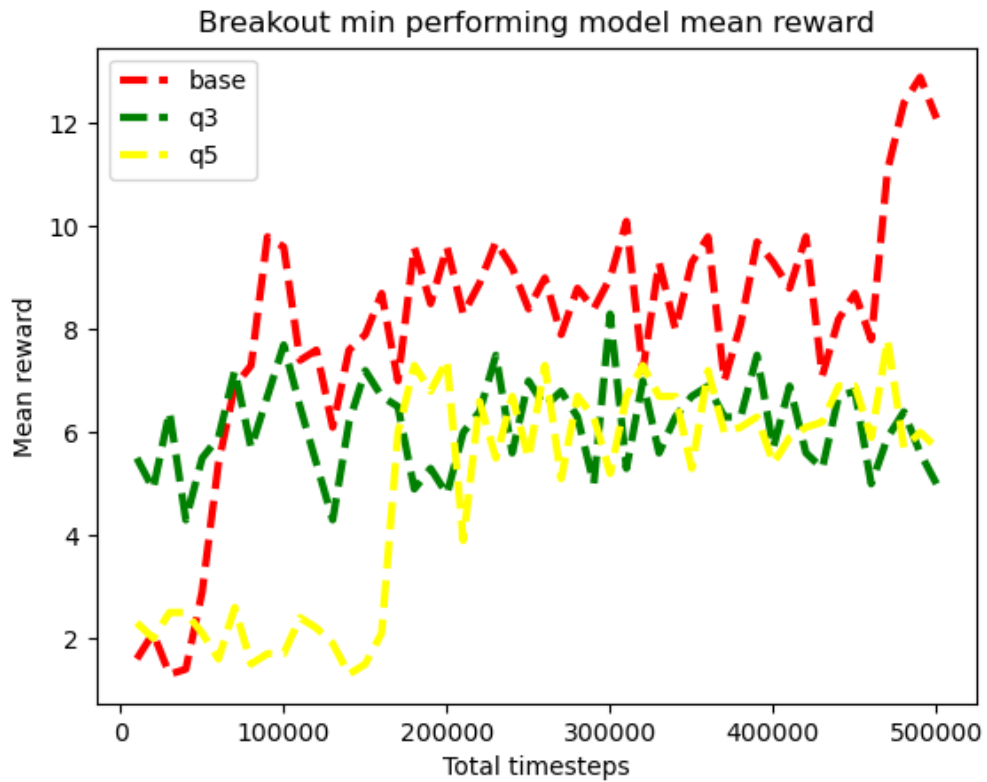


Figure 10. Models with worst final scores

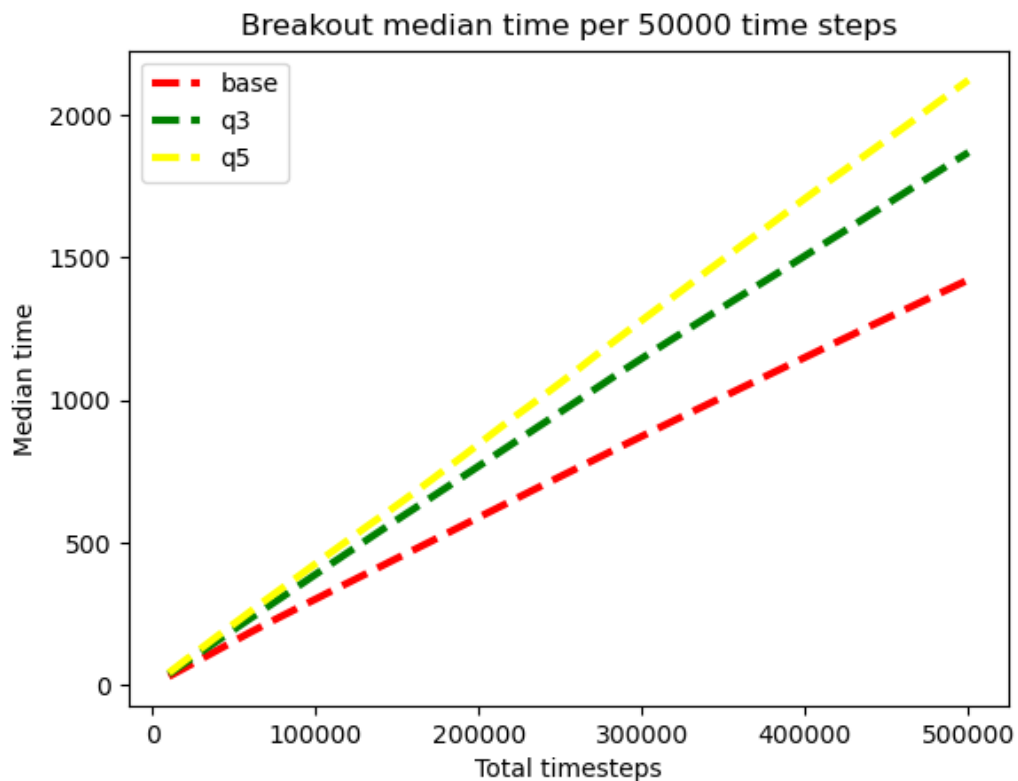


Figure 11. Median time used for training each model

Figure 11 shows how the degree of the functions used for the neurons affect the time required for the learning. Time needed to train one CNN based models take about 24 minutes while self-ONN based models using third degree functions take about 33 minutes to train and self-ONN based models take approximately 35 minutes. While there are also other drawbacks to having models with larger parameter counts such as requiring higher memory and having slower inference, if the self-ONN models cannot compete with CNN models consistently with the same structure it can be reasoned that smaller self-ONN based DL models with same number of parameters as CNN based DL models provide no advantage and so should not be used.

These results have one more very apparent flaw, which is the game that is being tested. Atari Breakout is a simple game with not a lot of visual complexity meaning there is not a lot of different colours or complex shapes. The ball and the platform are red while the disks that are meant to be hit are colourful other than that the game just has black background and grey walls. There also are not that many different features that matter in a game of Breakout. Obvious features that the model should be able to extract from the pixel values are ball location, paddle location, ball direction and ball speed. It is easy to imagine that these features can be extracted with a simple CNN structure especially so since the game consists of clear colours and simple shapes and so using self-ONNs

only adds unnecessary complexity to the models without providing any reasonable improvement.

5.2 Ms. Pacman

Similarly to Section 5.1, results comparing the DRL networks using CNN and self-ONN based models are presented here in this case for the game of Ms. Pacman the same structure networks are used for both games only the output size is changed automatically as Ms. Pacman has larger action space than Breakout.

Ms. Pacman is another classic Atari game where the goal of the game is to eat all the pellets while avoiding the ghosts that are trying to eat the player. Compared to Breakout Ms. Pacman has a lot more complexity in it and the information that should be extracted from the pixel presentation of the screen should be harder to get, such as the colour of the ghosts (as that defines how the ghost acts), position of the ghosts, position of the fruits and pellets, and position of the player (Ms. Pacman). Figure 12 represents a possible state of the game Ms. Pacman. In the image Ms. Pacman (in the bottom right corner) has collected 14 pellets so the score is 140 as each pellet gives 10 points.



Figure 12. Ms. Pacman

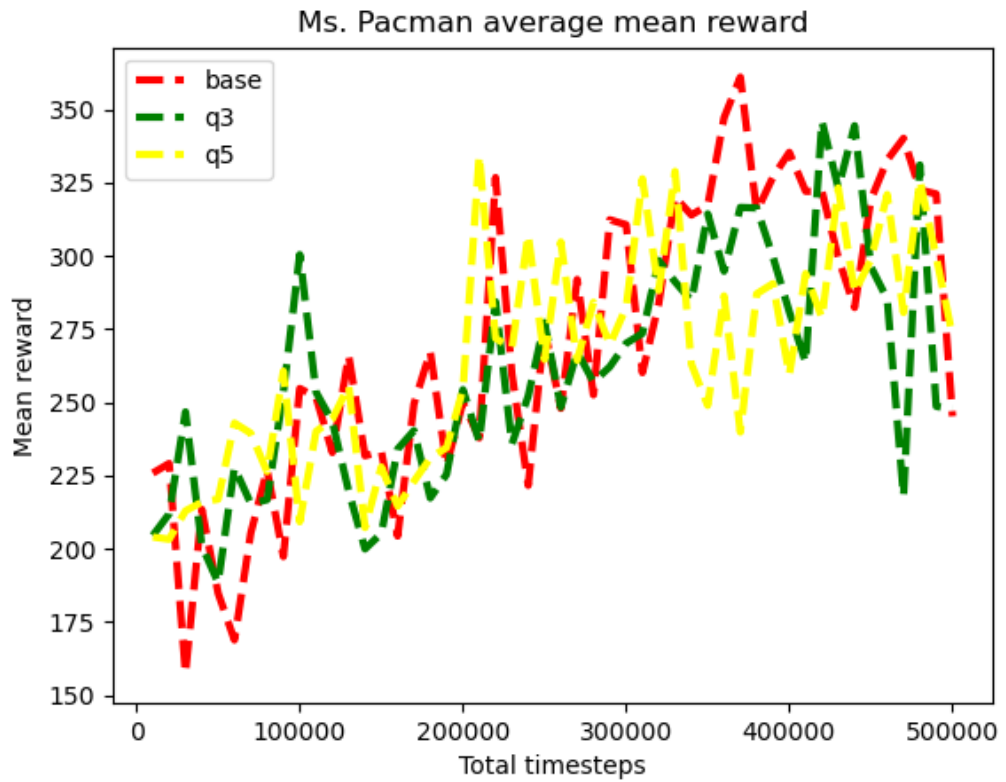


Figure 13. Average mean reward of the models produced for Ms. Pacman

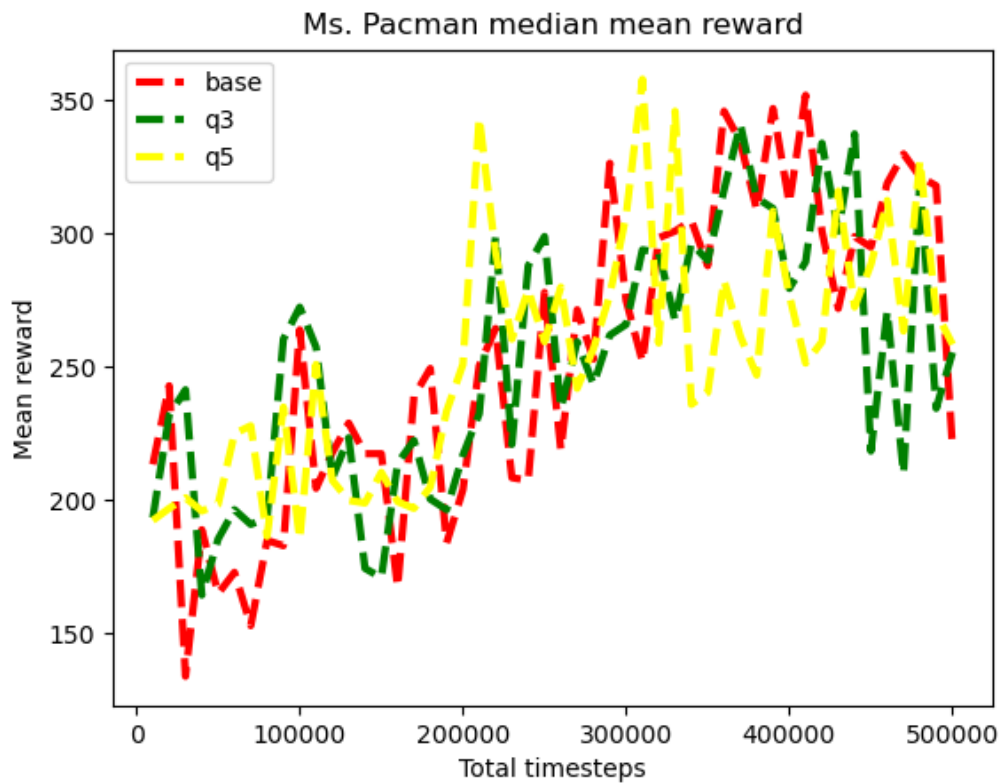


Figure 14. Median mean reward of the models produced for Ms. Pacman

Figure 13 and Figure 14 show the average and median performance scores for the different networks in case of Ms. Pacman. Similarly to Breakout using self-ONNs seem to not provide any improvement when compared to using CNNs in the DRL network, though in the case of Ms. Pacman they also do not seem to perform worse on average like they did for Breakout. The increased variability in scores between each time step can be explained by the increased complexity of the game. Slight changes in the actor's behaviour as well as slightly different initial state of the game can lead to very different scores overall.

The expected reward for a random agent playing Ms. Pacman is around 257 (result given by averaging 1 000 random agent games of Ms. Pacman). Neither self-ONN nor the CNN based models' average performance go significantly over this limit though looking at the average performance graph some form of upward trend can be seen so the networks do still exhibit signs of learning to play the game. Not going over the random actor score though can be seen as a clear sign that the networks have not been taught on enough data to learn the game properly. As stated previously though two million frames of learning material is not that much if compared to the 200 million used in the rainbow paper [12] or the 104 million used in the paper introducing DQN [11]. Another interesting thing about the results is that when comparing to the results from DQN introductory paper Breakout is one of the games that performed best when compared to humans while Ms. Pacman is one of the games that performed the worst. When comparing the results between Ms. Pacman and Breakout, the self-ONN based models performed seemingly as well as CNN based models in learning Ms. Pacman even though they failed to reach the same level of learning in the case of Breakout. Although, the performance difference when compared to the DQN paper could potentially be the result of the change in the algorithm used as the A2C model is used in this paper instead of DQN algorithm, it could also be the result of the inserted self-ONNs being able to better learn patterns that are harder for regular CNNs to learn being able to counter act the negative effects caused by the increased parameter count and the more difficult to navigate optimization landscape that might hinder the performance of the models that use self-ONNs.

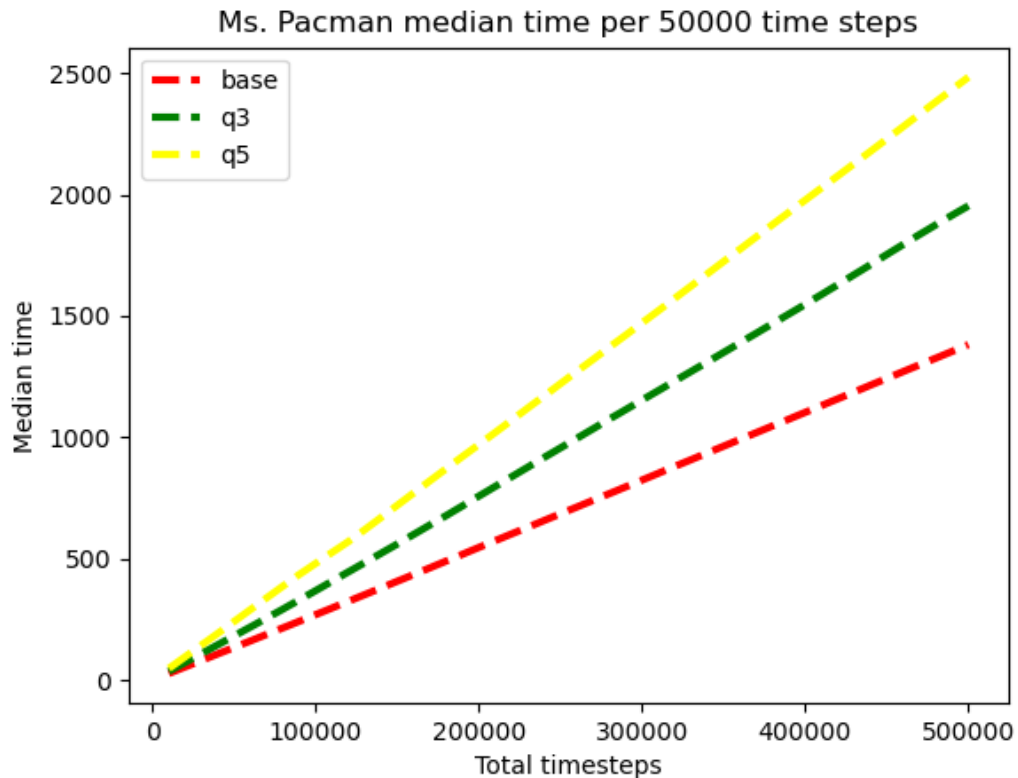


Figure 15. Training time for same size models

Figure 15 shows the median time taken to train each model. As expected, the self-ONN based models require longer time to train as it did in case of Breakout due to having increased parameter count. Since the self-ONN based models did not outperform the CNN based models, using them in this case could not be justified due to the increased time it takes to train them. The increase in performance ought to be as many times bigger as the increase in time consumption, as training time is one big limiting factor in DRL.

Figure 16 and Figure 17 show the models that ended up with the worst and best final scores. Since the variance in scores between two consecutive training epochs are so large the difference in the final scores can be blamed on chance. Since Ms. Pacman can be somewhat chaotic game and small changes in policy can lead to very different final rewards the comparisons of single models are not effective way to compare the models. Which is why comparisons over multiple models and their average and median performance is preferred especially when the models cannot be trained over hundreds of millions of frames. In short, the positive learning curve as well as the difference in performance of the different networks are very hard to see properly from Figure 16 and Figure 17.

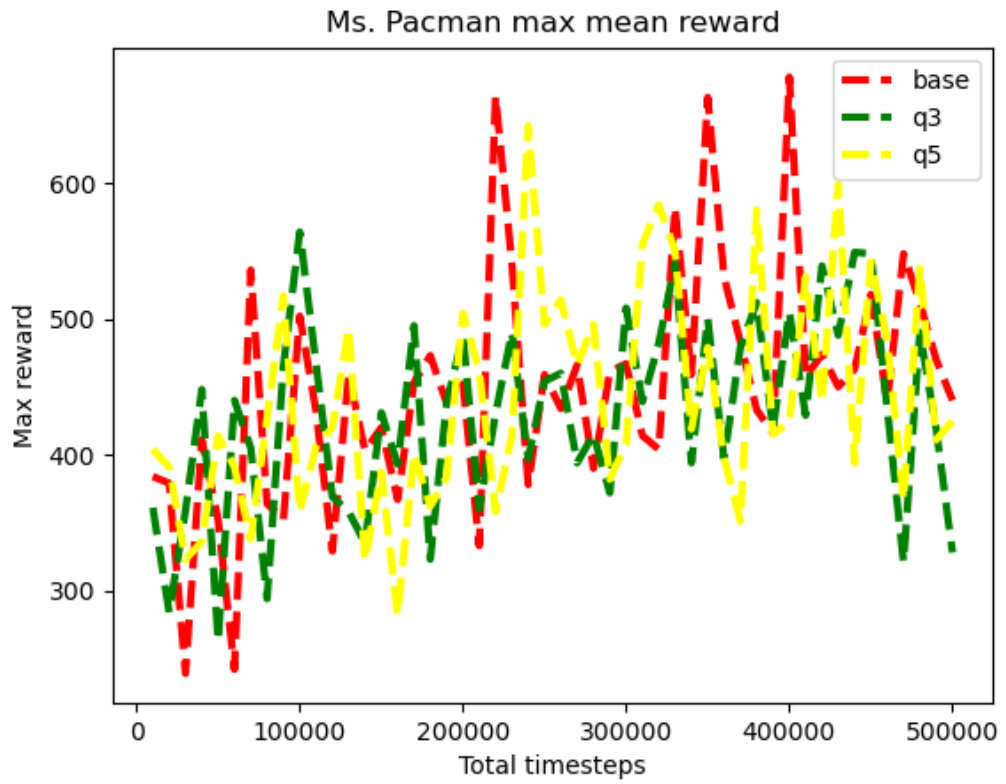


Figure 16. Models with best final scores

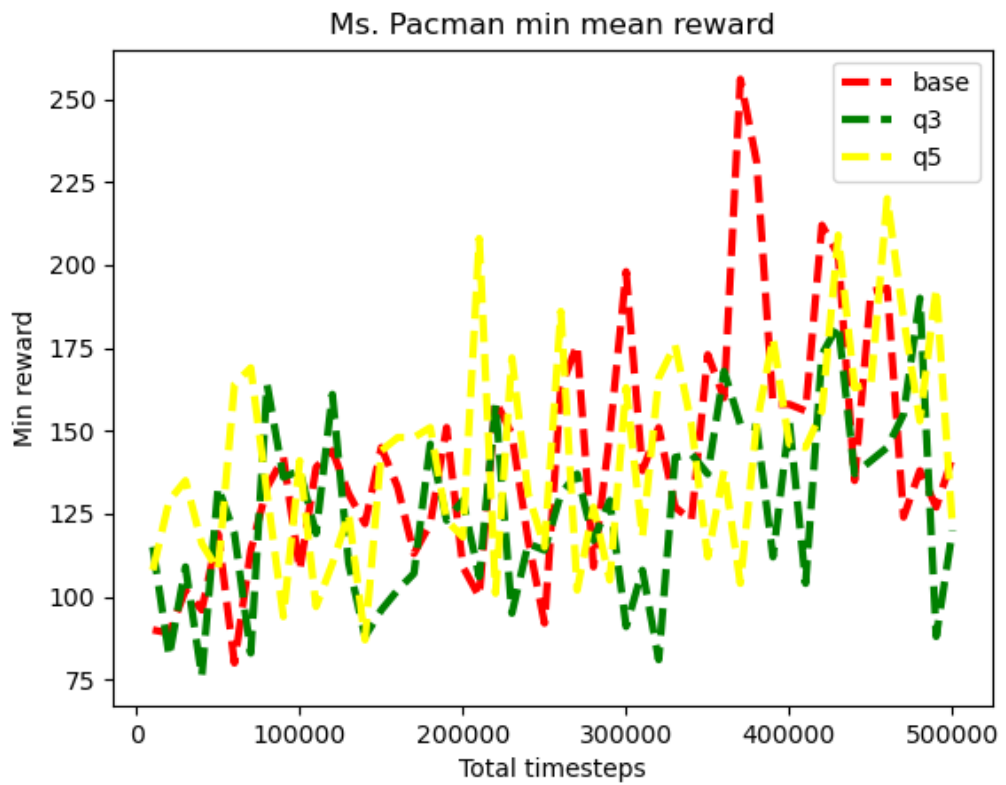


Figure 17. Models with worst final scores

5.3 Other Atari games

In this section ten different the networks performance on ten different Atari networks are compared. The models are generated in the same way as in Section 5.1 and Section 5.2 though the statistics are in lesser detail. Also, no fifth degree self-ONN models are generated in this section. Four different scores are presented for each game. Average first means the average performance achieved after the first epoch or 10 000 total timesteps. Average last is the performance achieved after the last epoch or 500 000 total timesteps which corresponds to two million frames. The difference of these two scores tells whether the network is learning to play the game or not and how much learning happens between these timesteps. The other two scores min and max tell the minimum and maximum final scores achieved by any of the ten models generated for each different network. These scores indicate the spread and consistency of the final performance achieved by each network. Expected random actor score is also given for all the different games tested. These scores are achieved by having a random actor play the game over thousand iterations of the game and taking the mean from these scores. Expected random actor score is the baseline score for the game.

Table 1. Alien (expected random actor score \approx 203)

Model	average first	average last	min	max
CNN	142	113	52	221
Self-ONN q3	98	128	60	208

Table 2. Asteroids (expected random actor score \approx 905)

Model	average first	average last	min	max
CNN	249	280	202	348
Self-ONN q3	256	268	176	343

Table 3. Boxing (expected random actor score \approx 0.7)

Model	average first	average last	min	max
CNN	-28.4	-28.1	-34.1	-23.7
Self-ONN q3	-29.1	-29.7	-34.7	-23.6

Table 4. Breakout (expected random actor score ≈ 1.3)

Model	average first	average last	min	max
CNN	3.2	15.6	12.1	18.6
Self-ONN q3	3.2	12.1	5.0	17.0

Table 5. Gravitar (expected random actor score ≈ 190)

Model	average first	average last	min	max
CNN	52.5	183.5	35.0	250.0
Self-ONN q3	62.5	173.0	35.0	260.0

Table 6. JamesBond (expected random actor score ≈ 28)

Model	average first	average last	min	max
CNN	24	187	120	230
Self-ONN q3	13	138	0	270

Table 7. KungFu Masters (expected random actor score ≈ 514)

Model	average first	average last	min	max
CNN	19	158	20	1 020
Self-ONN q3	7	65	0	160

Table 8. Montezuma's Revenge (expected random actor score ≈ 0)

Model	average first	average last	min	max
CNN	0	0	0	0
Self-ONN q3	0	0	0	0

Table 9. Ms.PacMan (expected random actor score ≈ 257)

Model	average first	average last	min	max
CNN	226	245	142	440
Self-ONN q3	205	248	120	328

Table 10. NameThisGame (expected random actor score \approx 2357)

Model	average first	average last	min	max
CNN	2 080	2 394	1 546	2 893
Self-ONN q3	2 241	2 525	1 691	3 107

As seen from Tables 1-10 only in three of the ten games tested were on average able to learn a better policy compared to random actor. In six games the best iteration was able to learn better policy than random actor. Such poor results are a testament to the clearly too low amount of training data used. The network might learn a strategy that works for a specific scenario well, but when it generalizes that action for other states it ends up being suboptimal. This can lead to the network having a policy worse than random actor, which is the reason that the network should get a much larger amount of training data so that it would be able to generalize the correct actions for each state much more efficiently. Out of all the ten games only in three, did using self-ONNs increase the final average performance although very marginally. So marginally in fact that such a small change could be the result of chance, caused by the variation in model generation as well as model testing, instead of self-ONNs providing some benefit over CNNs.

In games Kungfu Masters, James Bond and Breakout the CNN based models performed substantially better. Although in the case of Kungfu Masters the significantly higher average performance can mostly be attributed to a single outlier model generated that was produced the maximum score of 1 020 if this very fortunate result is omitted from the average final performance score then it would be 62 instead of the 158 it is when this max score is also used in the calculation.

Out of all the games tested only in two cases did the networks both see significant learning and get average performance exceeding the random actor score. These two cases being Breakout and James Bond. In both cases CNN based models outperformed the self-ONN based ones in every aspect other than the maximum performance score which in the case of James Bond game self-ONN based model was able to outperform CNN based model.

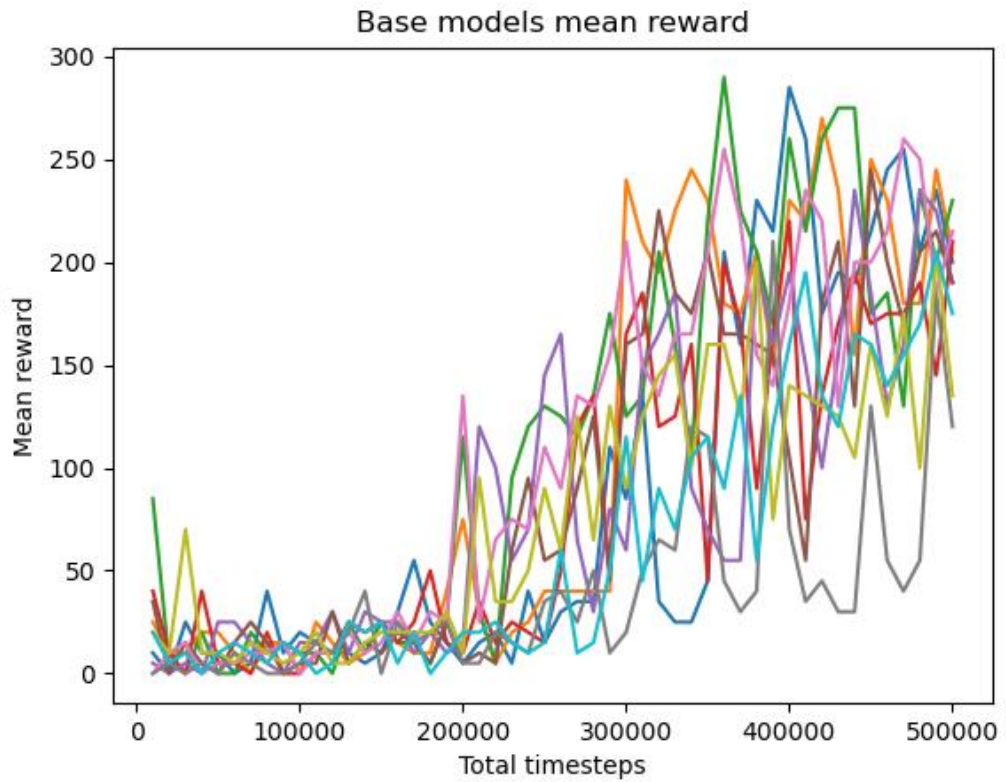


Figure 18. James Bond CNN based models' performances

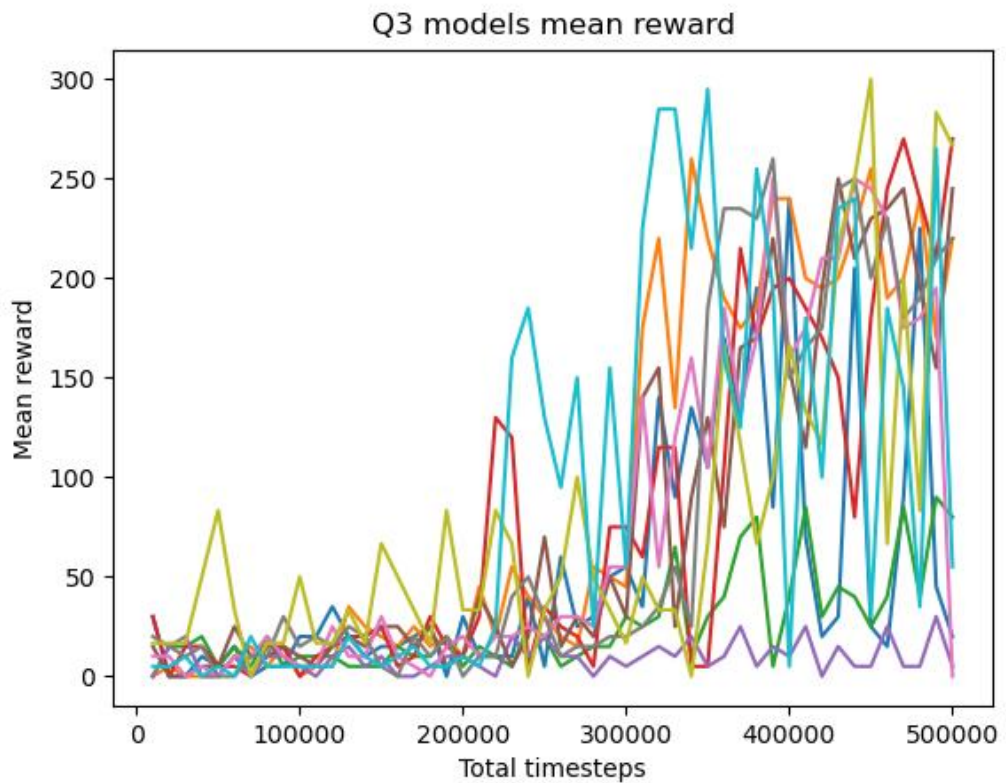


Figure 19. James Bond self-ONN based models' performances

Figure 18 and Figure 19 represent the learning curves for all the models generated for James Bond. The self-ONN based models seem much more volatile than the CNN based models and end up having more spread-out final performance scores, while CNN based models get more consistent results.

Montezuma's revenge is an example of a game ill fitted for DRL models in general. This is due to how points are calculated in the game. To get any points in Montezuma's revenge the player needs to find items that are scarcely scattered around. In case of DRL this would mean that the initial state of the network would need to be very specific to even reach the first reward or when using random actions to explore the space the random actions would need to be very specific in a very specific order. Both cases are so rare that it is very likely that the agent never reaches a reward and so the network will not get the necessary feedback to learn anything. A score higher than 0 was not achieved even with 104 million frames in the DQN paper [11], leaving it to have the worst performance out of all the games tested in that paper compared to humans.

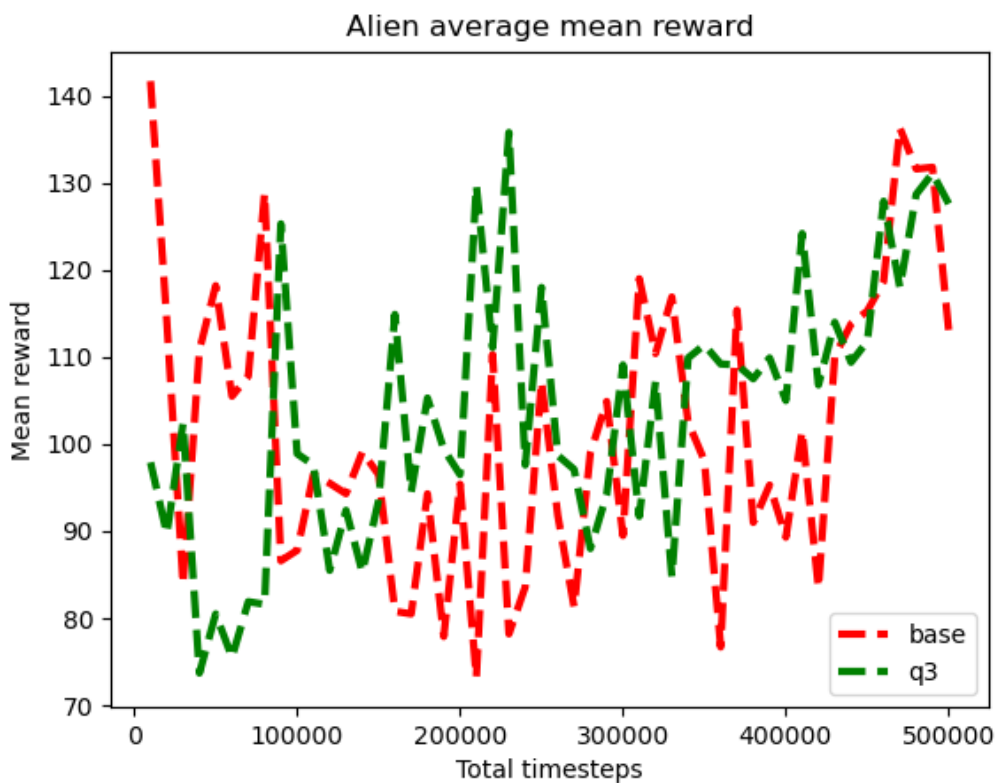


Figure 20. Alien average performance

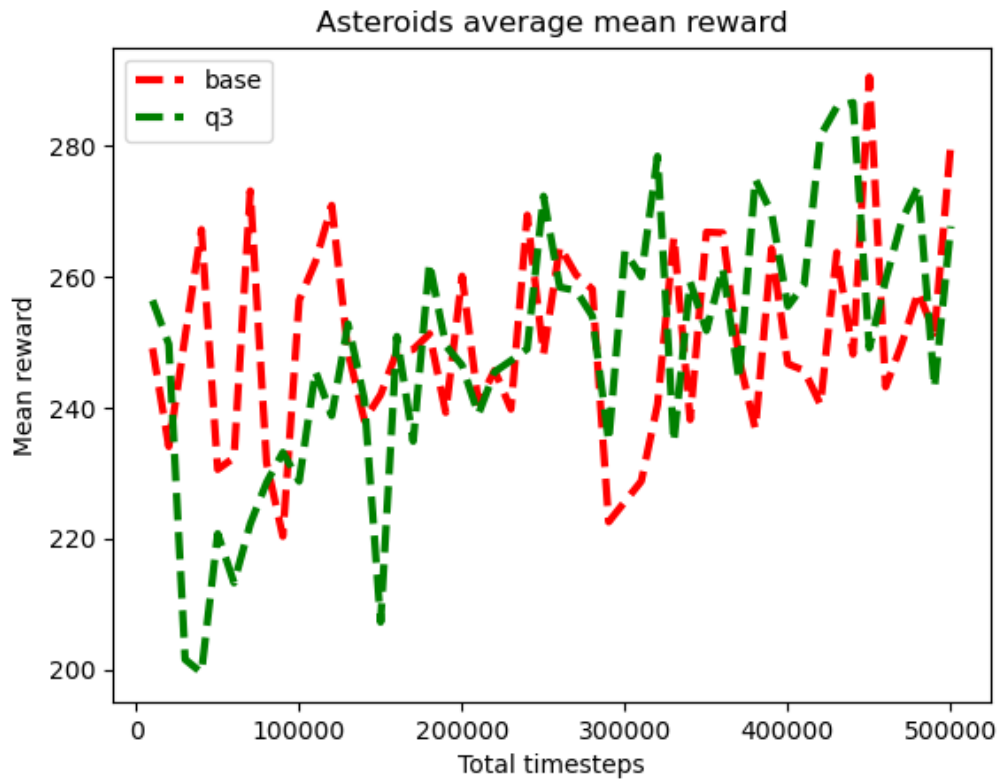


Figure 21. Asteroids average performance

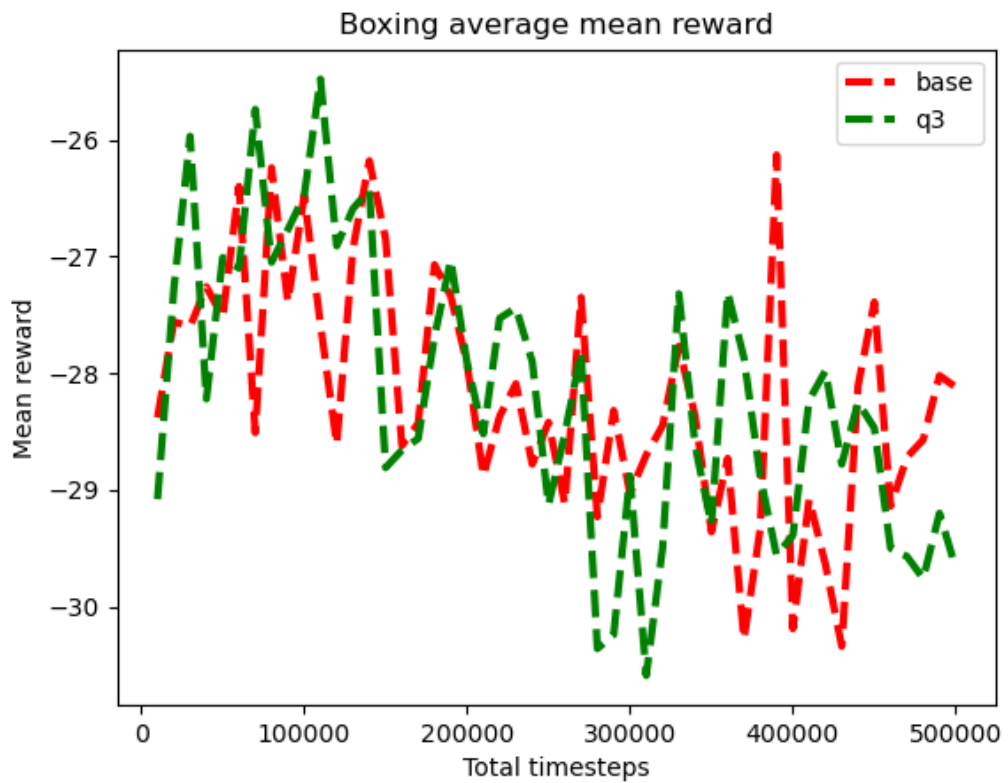


Figure 22. boxing average performance

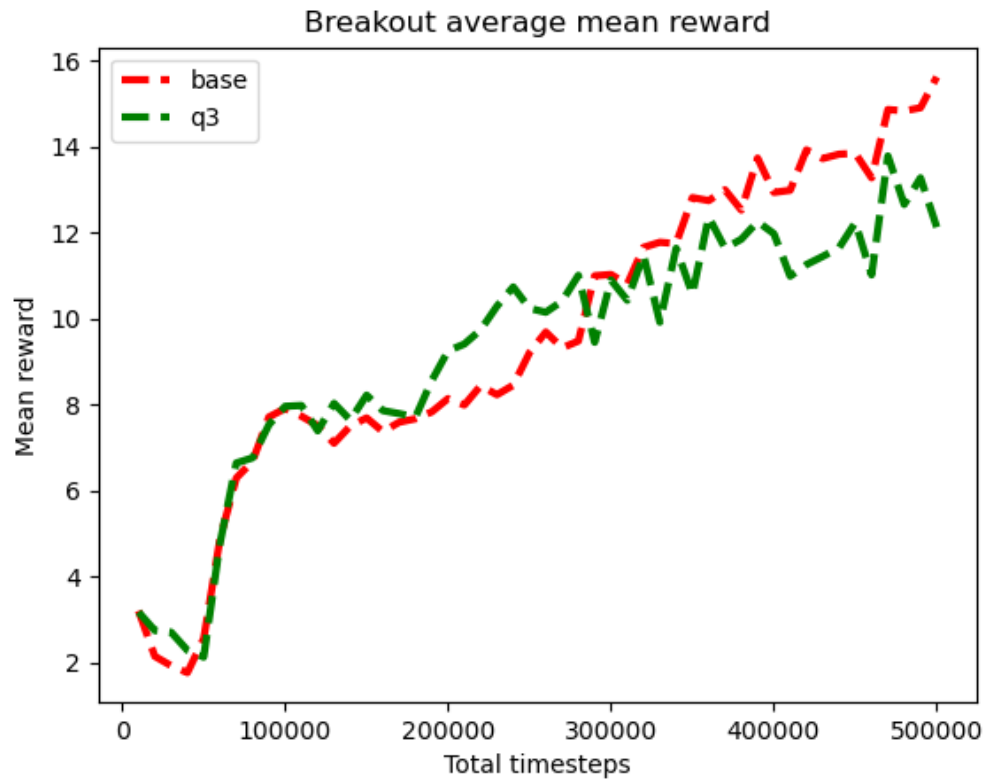


Figure 23. Breakout average performance

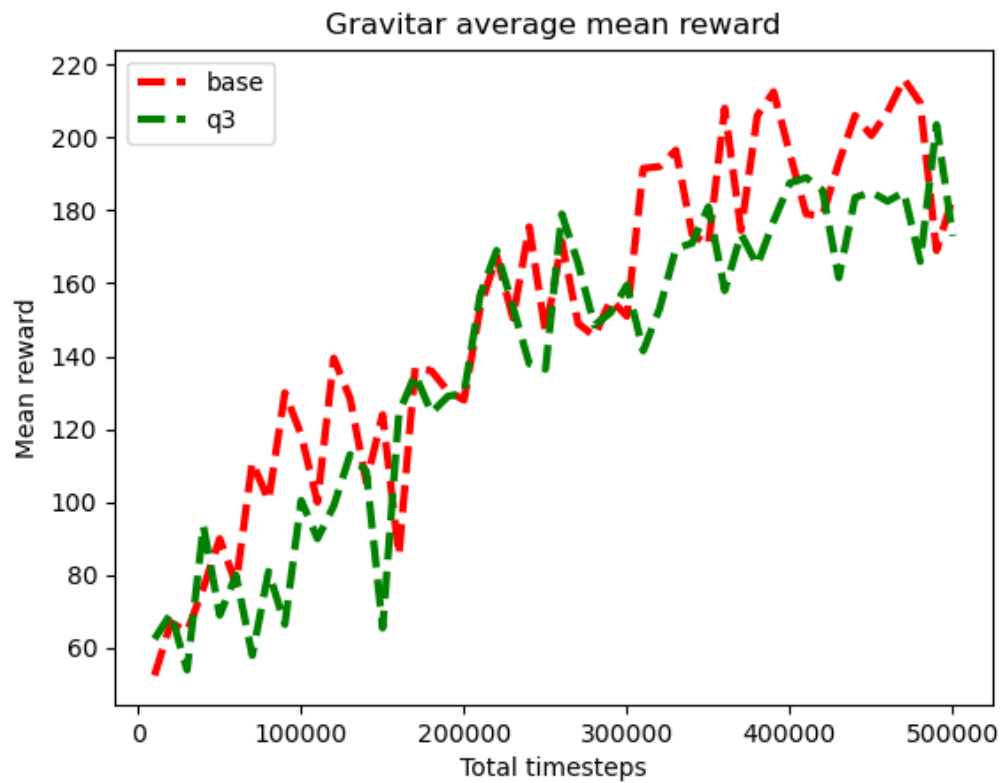


Figure 24. Gravitar average performance

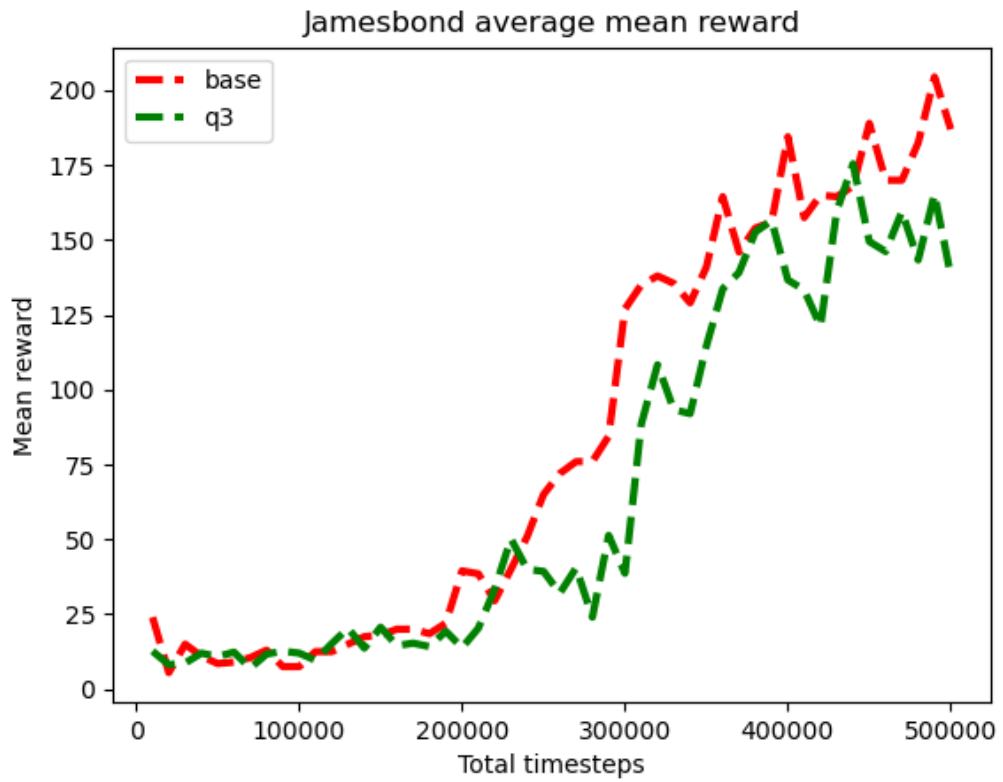


Figure 25. JamesBond average performance

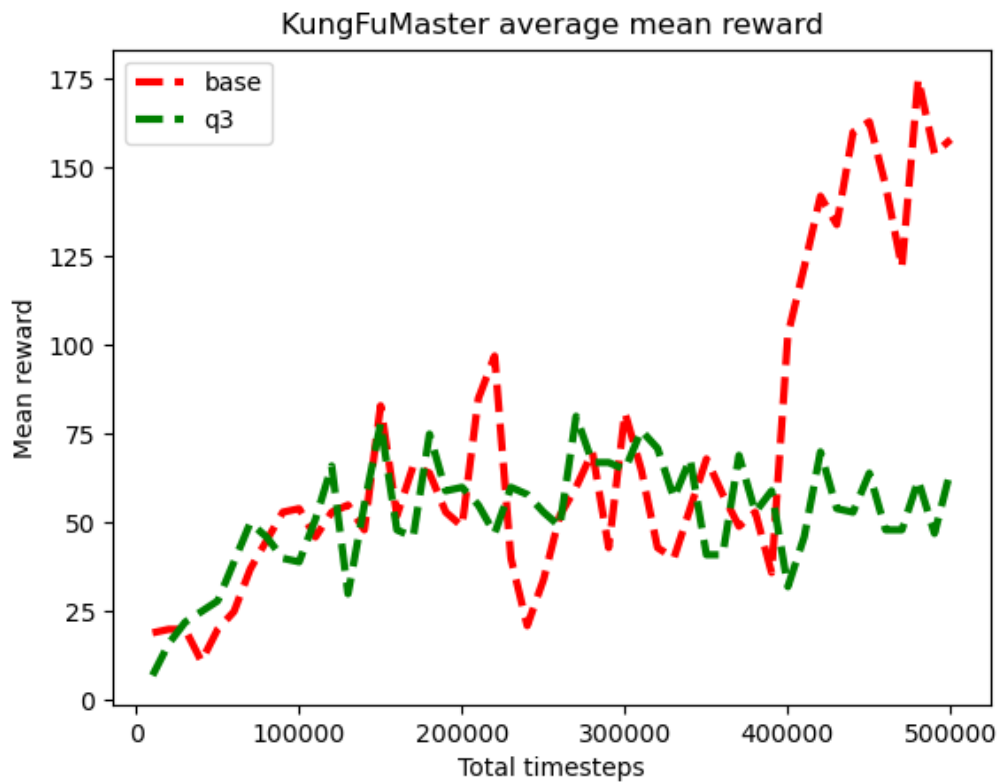


Figure 26. KungFuMasters average performance

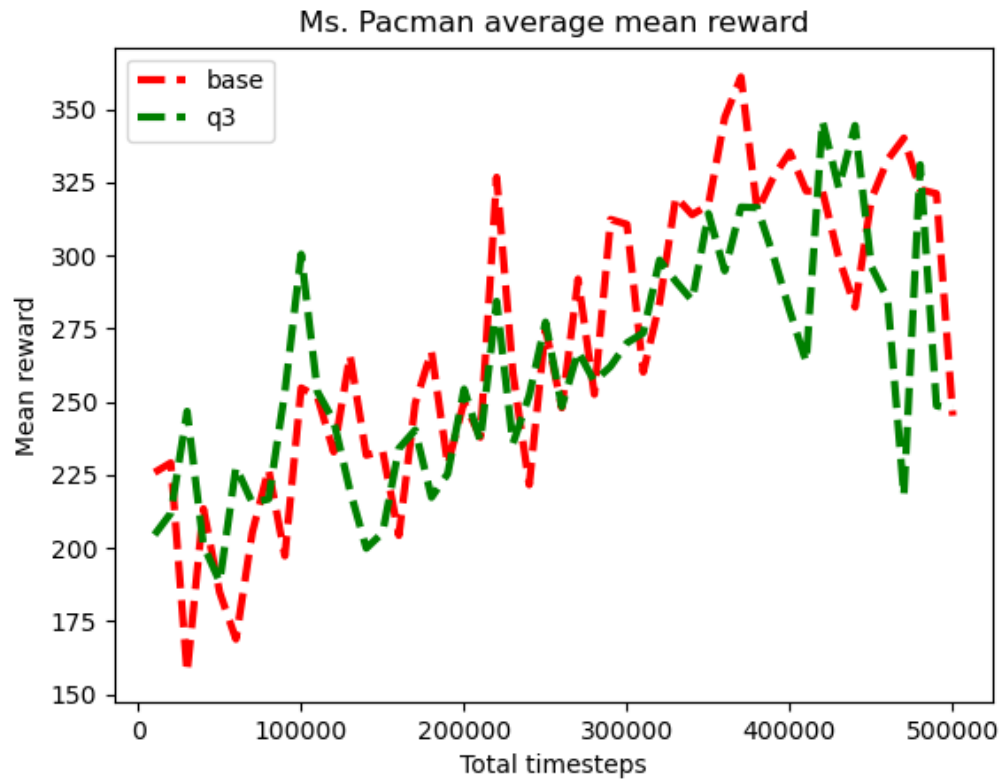


Figure 27. Ms.Pacman average performance

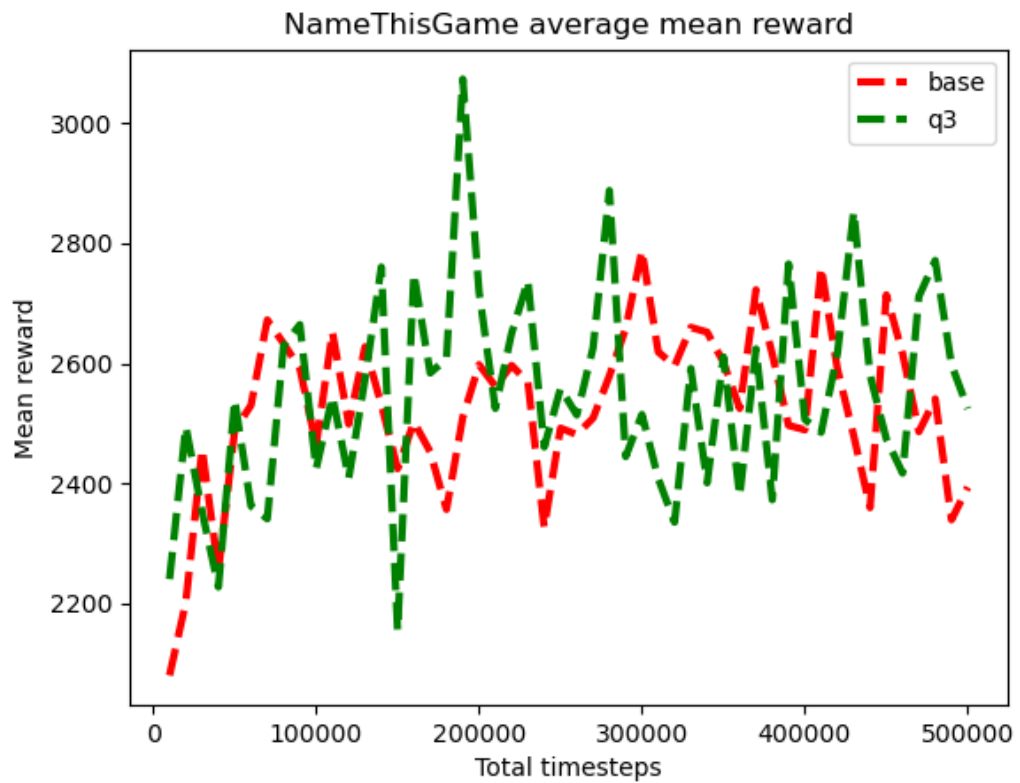


Figure 28. NameThisGame average performance

Figure 20, Figure 21, Figure 22, Figure 23, Figure 24, Figure 25, Figure 26, Figure 27 and Figure 28 show the learning curves of the each respective game show cased in the tables before. These Figures underlie in greater detail the problems with the tests. Only the learning curves for the games Breakout and James Bond behave well. In all the other cases the learning curves have a lot of sudden changes. Since the curves do not show a consistent learning rate, the poor performance of the models might not only be a problem of lack of training data but also a problem of unoptimized hyperparameters.

5.4 More Atari games with better code

After testing different kinds of hyperparameter combinations, trying different learning rates, entropy coefficients and value function coefficients, the apparent reason why the networks learning speed diminished was due to what kind of environment was chosen. The environments used are gymnasium Atari environments and gymnasium is a library developed by OpenAI and such there are multiple different versions of the environments. The previously used environment was -v5 version and after switching the version to NoFrameskip-v4 the networks learned faster and did not plateau to a low score as fast. This can be due to the fact that using no frame skip allows the network to make actions every frame as well as learning from every frame allowing it to make more fast-paced decisions as well as observe more subtle and faster movements.

On top of the environment version change some other minor changes were also made to the network though when tested they did not seem to affect the performance that much. The kernel sizes were changed from (11, 7, 3) to (8,4,3) with strides half the kernel size. Also, an additional linear layer was added to the end with additional rectified linear unit to match it. Entropy coefficient was set to 0.01 to enable some exploration. Value function coefficient was changed to 0.25 from default (0.5) to lessen the possibility of overfitting to long-term reward approximations. These changes were based on an

existing code made for learning Breakout which was tested to perform with the average score of 349.50 +/- 89.74 [30].

Games tested in this section were chosen based on a figure shown on a GitHub page where the average score of these games were shown [31]. This figure is shown below.

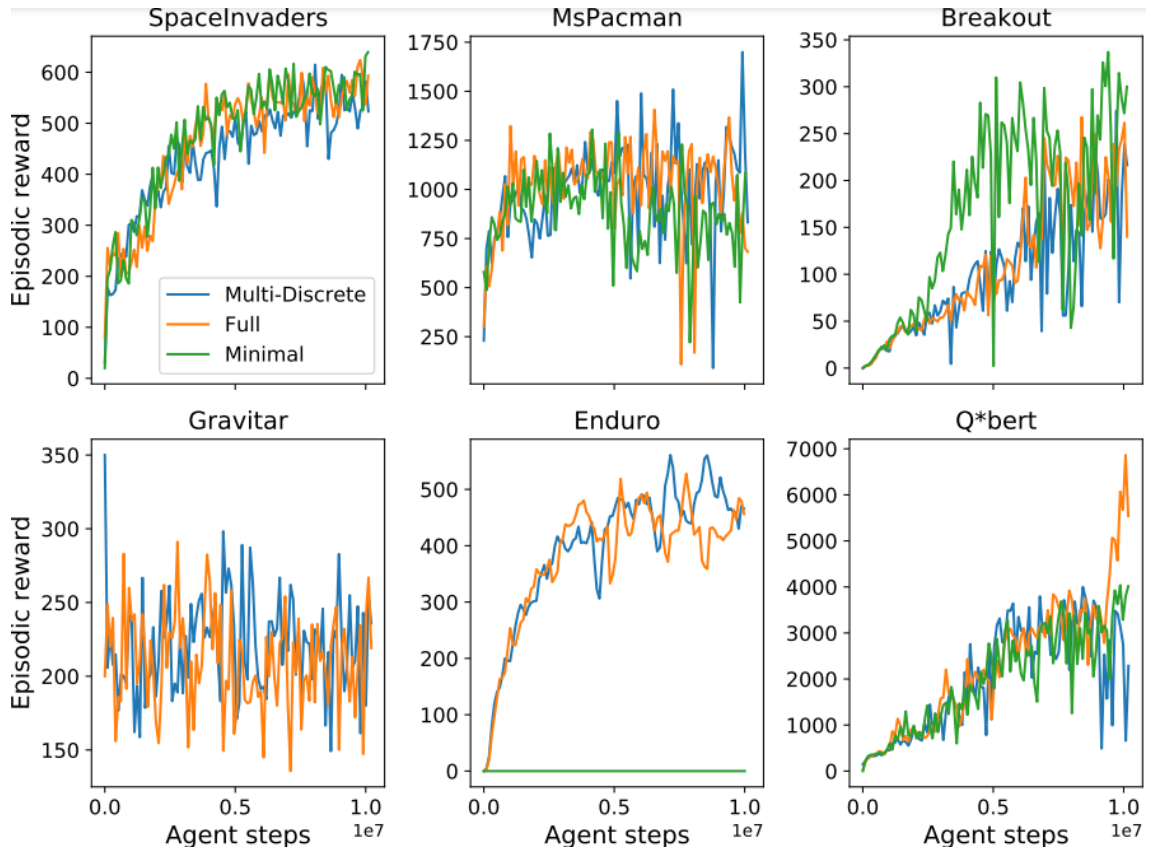


Figure 29. Test results on different games found in GitHub [31]

Figure 29 shows the test results for different games using A2C network and stablebaselines3 that were provided in GitHub page [31]. All these games were chosen for testing the CNN and self-ONN based networks apart from Gravitar, since it is shown to not have a well behaving learning curve, and Enduro since when tested the network

seemed to work similarly to the minimal solution shown in the Figure 29, meaning its average scores stayed near zero the entire training duration.

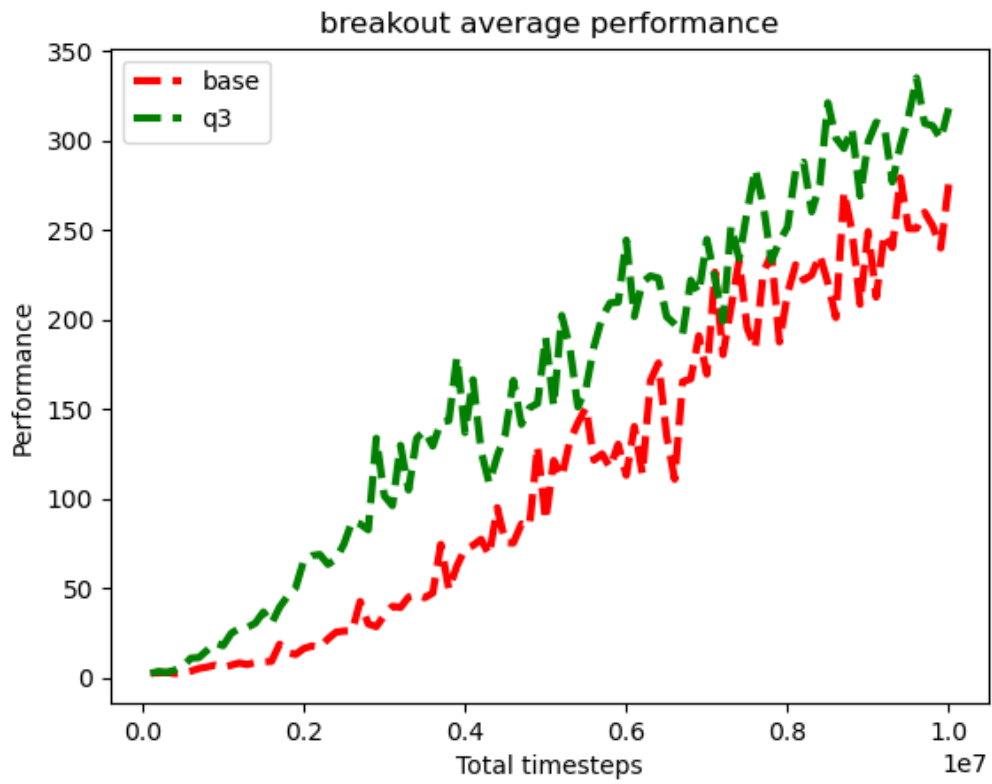


Figure 30. average performance for the game Breakout (random actor ≈ 1.21)



Figure 31. each different model's performance for the game Breakout

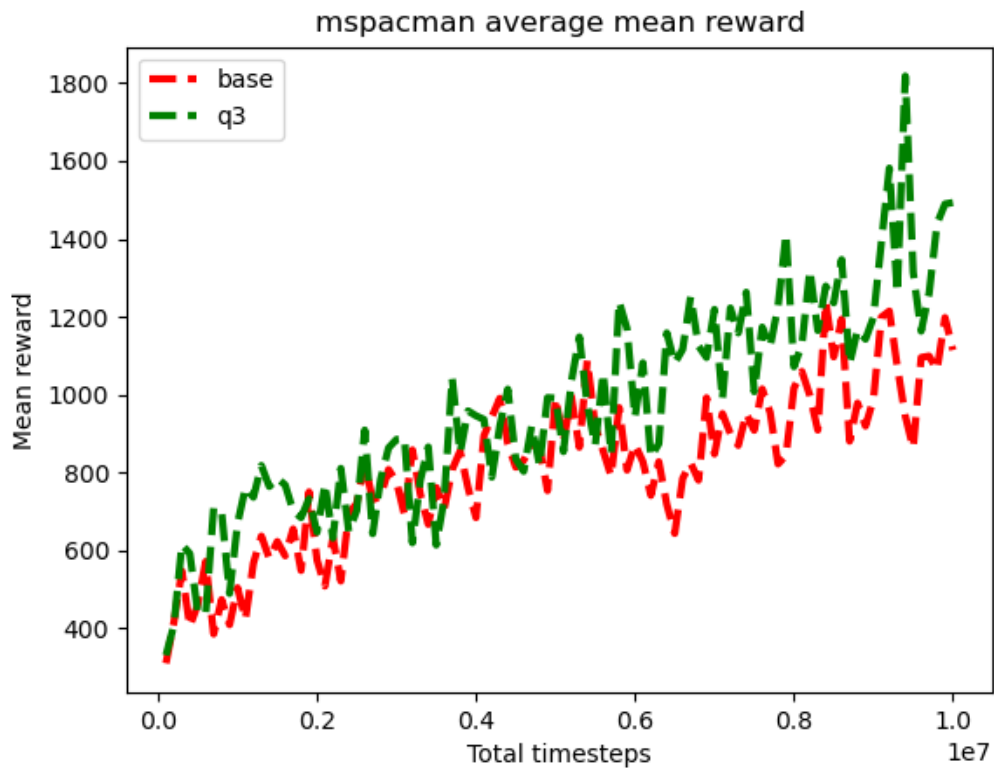


Figure 32. average performance learning curve for the game Ms. Pacman (random actor ≈ 170)

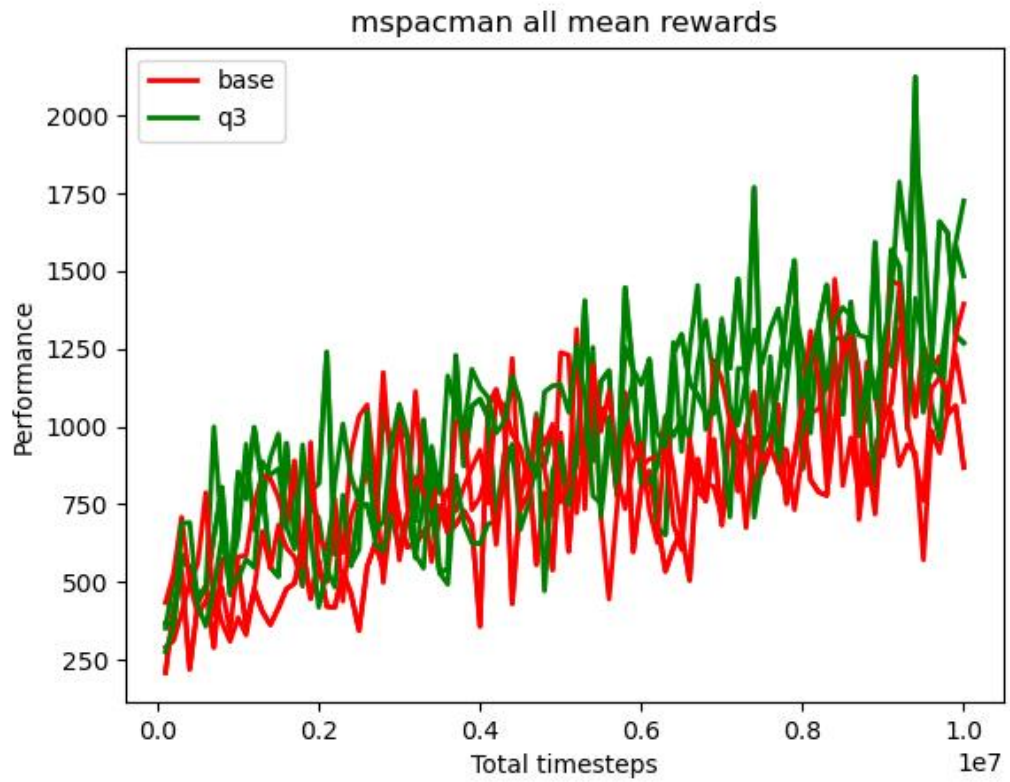


Figure 33. each different model's performance for the game Ms. Pacman

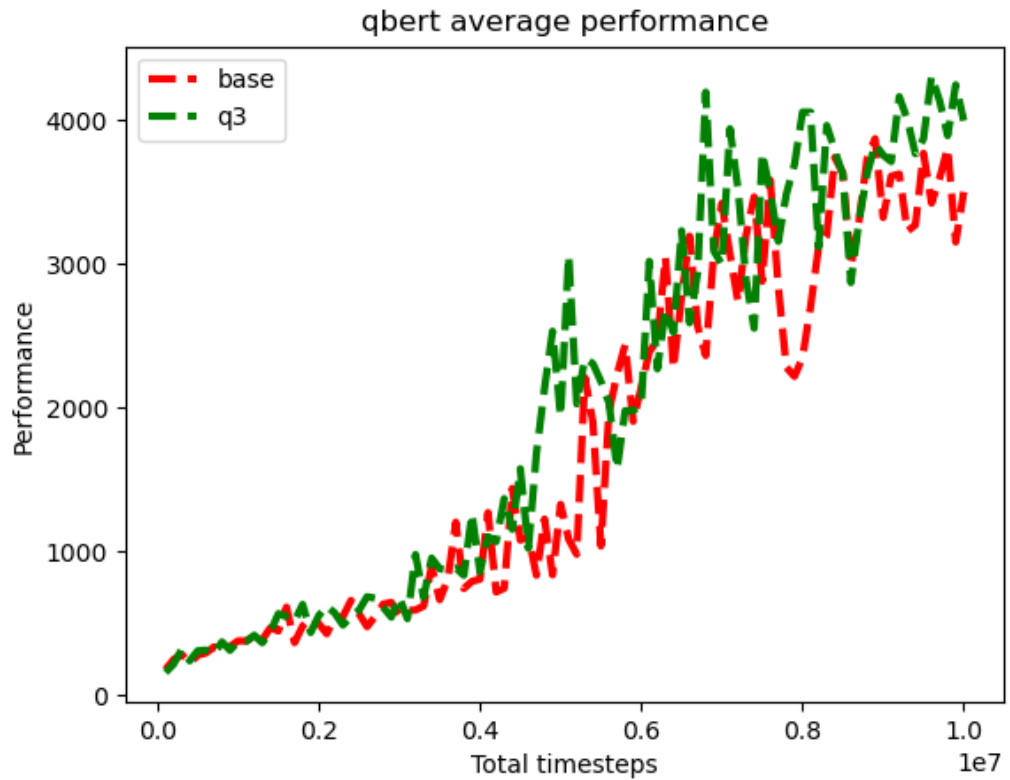


Figure 34. average performance for the game Q*bert (random actor ≈ 164)

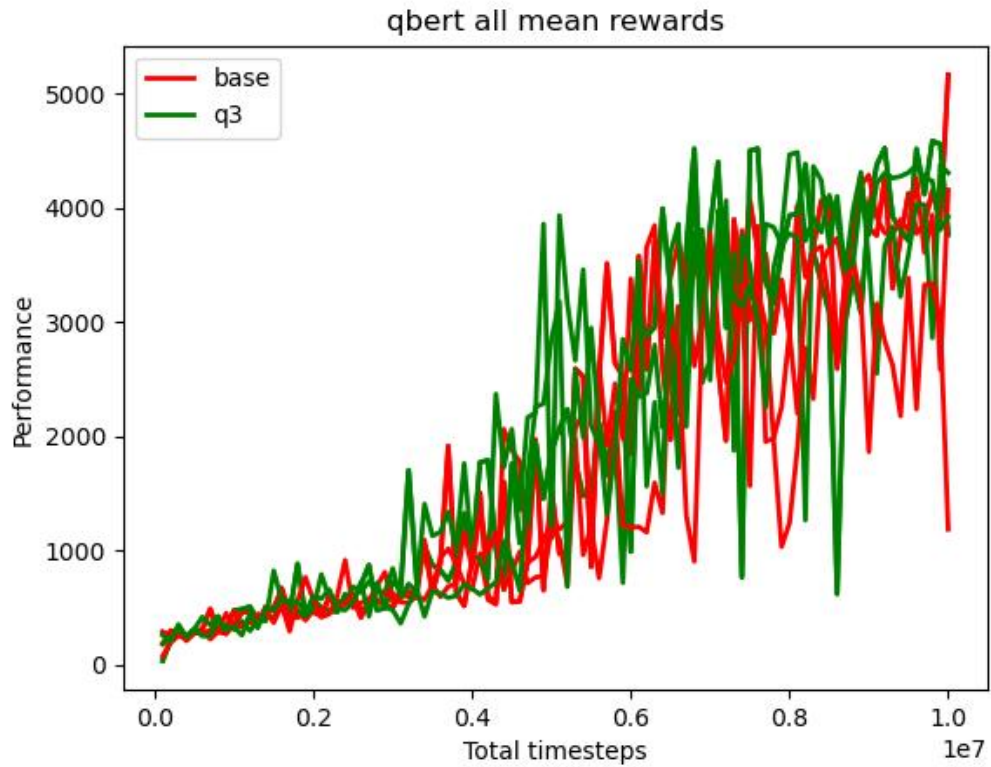


Figure 35. Each different model's performance for the game Q*bert

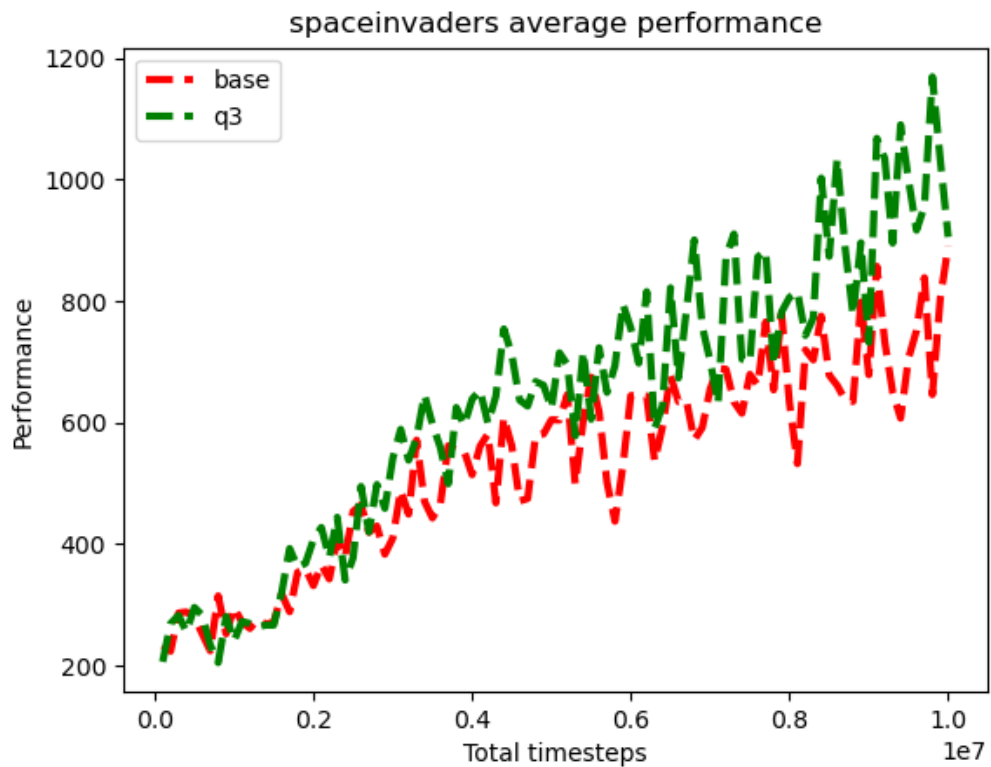


Figure 36. Average performance for the game Space Invaders (random actor \approx 160)

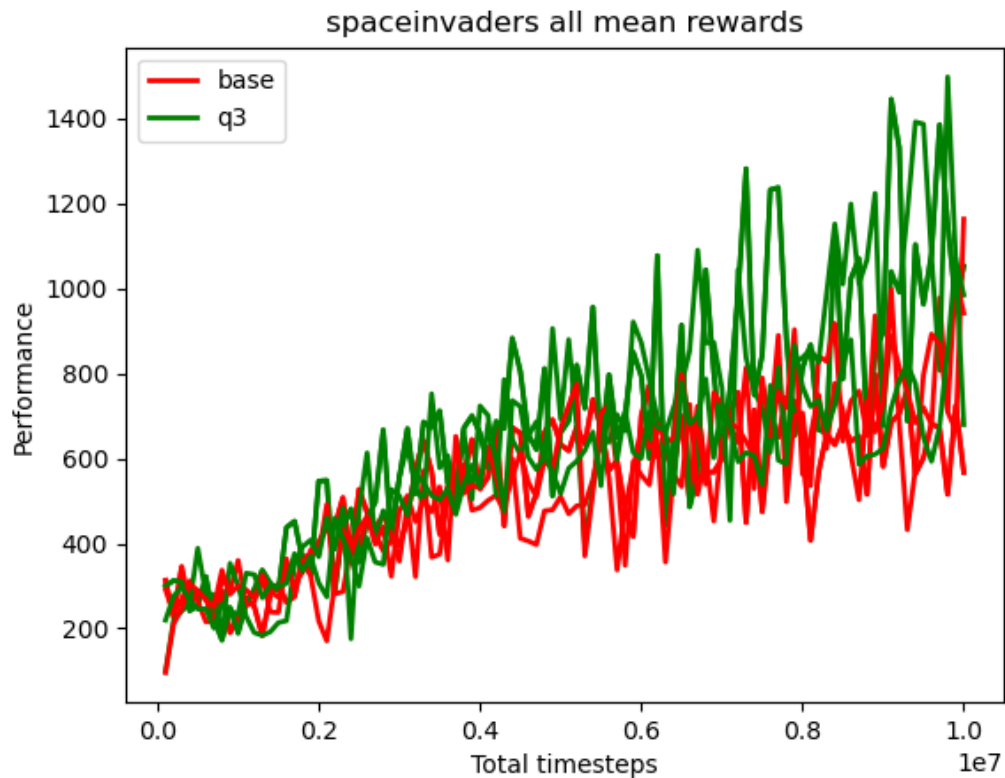


Figure 37. Each different model's performance for the game Space Invaders

Figure 30, Figure 31, Figure 32, Figure 33, Figure 34, Figure 35, Figure 36 and Figure 37 show each models average performance over 3 different models and each models performance. Here in each the Self-ONN based model did better than the CNN based model after ten million timesteps. Each of the trained models also reached a higher score than a random actor by a significant margin. The difference between the ending performance of the average self-ONN based model and the average CNN based model is also quite noticeable.

Table 11. Breakout (expected random actor score ≈ 1.21)

Model	average first	average at million	average last
CNN	2.87	6.07	276
Self-ONN q3	2.53	18.1	318

Table 12. Ms. Pacman (expected random actor score ≈ 170)

Model	average first	average at million	average last
CNN	310	504	1114
Self-ONN q3	331	669	1493

Table 13. Q*bert (expected random actor score \approx 164)

Model	average first	average at million	average last
CNN	178	377	3501
Self-ONN q3	157	373	3997

Table 14. Space Invaders (expected random actor score \approx 160)

Model	average first	average at million	average last
CNN	233	296	891
Self-ONN q3	206	241	906

Tables 11-14 show the average performance of the models at first iteration which equals to 100 000 timesteps at 1 000 000 timesteps and at 10 000 000 timesteps. The CNN based model seems to perform better at earlier timesteps but gets quickly overtaken by the self-ONN based model later. Since the initial tests were done with models trained only with two million frames which equals to 500 000 timesteps, this could explain partly why the CNN performed slightly better in the earlier tests. As to why the self-ONN based models might perform worse in earlier stages of training might come down to them having more parameters to optimize. This can lead to the network not generalizing as well. But the faster generalization will yield worse results in the long run if the network cannot learn to operate in certain specific or niche situations.

Since the results in this section are the averages of only three different models produced for each case, random variation in the training does have a higher impact on the results and thus they are slightly less reliable than if ten different models were produced for each case. As seen from the Figure 31, Figure 33, Figure 35 and Figure 37 which show each different models performance there are CNN based models which did end up having better performance than some of the self-ONN based models. Still however the average performance in all the games were so much higher that it would be quite unlikely that this difference cannot be attributed to a high degree to the use of self-ONNs in the networks.

6. CONCLUSIONS

The results of the section 5.3 would not seem to indicate that using self-ONNs in the feature extraction layer would be beneficial but this is more likely due to poor testing than anything else. The environment used in these tests caused the networks to converge to low scores having additional learning time not improve the results. On top of this the number of frames used for the learning was also wildly insufficient, leading to the networks in many cases to not perform better than a random agent. In the two test cases that did successfully beat a random agent while showing significant learning from the initial value, these cases being James Bond game and Breakout, the results were quite inconclusive. In both of these cases the CNN based models seemed to perform better on average. In the results shown in the section 5.4 however the self-ONN based models outperformed the CNN based models in every game tested, including Breakout where it had in earlier tests shown to perform worse. When comparing the scores that these two different network structures had at different points in training it seems that the CNN based model performed slightly better at 100 000 timesteps while the self-ONN based model performed much better at 10 000 000 timesteps. This might indicate that the CNN based models generalize fast but quite quickly get outpaced by the self-ONN models' added complexity and better fine tuning.

The reason self-ONNs were created is to combat the ever-increasing depth of CNNs by introducing neurons that can capture more complex patterns with less layers. In the case of using DRL to teach AI to play Atari games, the depth of the network is not as big of a concern as it is in many other vision-based tasks where a much deeper network is needed. In tasks like object detection depth of the networks can often reach upwards of hundred layers some of the largest networks currently can even reach around thousand layers such as in the case of ResNet-1001 [32]. In comparison the depth of the DRL network used in the simulations in this paper consists of the CNN which had three convolutional layers and the linear layer which should learn the correct actions to be taken from the features extracted. The depth of the network in this case was not the bottleneck, instead the actual bottleneck ended up being the training time, which greatly limited the number of frames that could be used to train the models. Still considering this the additional performance boost provided by using self-ONNs proved to be quite significant. However, one must also keep in mind that the training time for the self-ONN

based models were also quite a bit longer since the same added complexity that made them perform better in the long run, made the training time of the network to increase.

7. FUTURE WORK

Future work on this subject should focus on producing larger number of models with longer training times to have even more robust results and to see how the learning curves continue to develop onwards of hundreds of millions of timesteps. The different models should also be tested on a larger variety of games to see whether there is additional benefit of using self-ONNs in some games compared to others. In addition, to larger training set and wider range of games tested, fine tuning the hyperparameters for each case would also give more robust and accurate results.

Another way to utilize newer neuron models in DRL would be to replace the action or policy networks' layers with ones using non-linear neurons. Instead of replacing the CNN of feature extraction layer with self-ONN, the action or policy network of the DRL network could be replaced by one using newer neuron models. Action and policy networks are often *Multi-Layer Perceptrons* (MLPs) consisting of linear layers and activation layers. Instead of using MLPs the policy or action network could be replaced with *Generalized Operational Perceptron* (GOP) which consists of operational neurons [33]. While Atari games are not visually very complex and using DRL to play them does not seem to benefit from the usage of newer neuron models. There is a chance that replacing action or policy network, with operational neurons, would improve the performance of the models created.

REFERENCES

- [1] T. Schaul, J. Quan, I. Antonoglou and D. Silver, "Prioritized Experience Replay," *ArXiv*, vol. 1511.05952, 2015.
- [2] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundel and S. Legg, "Noisy Networks For Exploration," *ArXiv*, vol. 1706.10295v3, 2019.
- [3] M. G. Bellemare, W. Dabney and R. Munos, "A Distributional Perspective on Reinforcement Learning," *ArXiv*, vol. 1707.06887v1, 2017.
- [4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *ArXiv*, vol. 1602.01783, 2016.
- [5] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel and S. Levine, "Soft Actor-Critic Algorithms and Applications," *ArXiv*, vol. 1812.05905v2, 2019.
- [6] J. Malik, S. Kiranyaz, M. Yamac, E. Guldogan and M. Gabbouj, "Convolutional versus Self-Organized Operational Neural Networks for Real-World Blind Image Denoising," *ArVix*, vol. 2103.03070, 2021.
- [7] H. H. Mohammed, J. Malik, S. Al-Maadeed and S. Kiranyaz, "2D Self-organized ONN model for Handwritten Text Recognition," *Elsevier*, vol. 127, 2022.
- [8] O. C. Devecioglu, J. Malik, T. Ince, S. Kiranyaz, E. Atalay and M. Gabbouj, "Real-Time Glaucoma Detection From Digital Fundus Images Using Self-ONNs," *IEEE Access*, vol. 9, pp. 140031-140041, 2021.
- [9] J. Malik, S. Kiranyaz and M. Gabbouj, "Self-organized operational neural networks for severe image restoration problems," *Elsevier*, vol. 135, pp. 201-211, 2021.
- [10] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, p. 58–68, 1995.
- [11] V. Mnih, K. Kavukcuoglu and D. Silver, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, p. 529–533, 2015.
- [12] H. v. Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *AAAI Conference on Artificial Intelligence*, 2015.
- [13] "OpenAI Spinning Up," OpenAI, 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below. [Accessed 18 6 2024].
- [14] S. J. Russell ja P. Norvig, *Artificial Intelligence: A Modern Approach*, Third Edition, Prentice Hall, 2011.
- [15] R. S. Sutton ja A. G. Barto, *Reinforcement Learning: An Introduction*, Second edition, London: Bradford Books, 2014.
- [16] K. Sofeikov, "REINFORCE algorithm — Reinforcement Learning from scratch in PyTorch," *Medium*, 4 May 2023. [Online]. Available: <https://medium.com/@sofeikov/reinforce-algorithm-reinforcement-learning-from-scratch-in-pytorch-41fccafa107>. [Haettu 27 August 2024].
- [17] D. Silver, T. Hubert and J. Schrittwieser, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *ArXiv*, vol. 1712.01815, 2017.
- [18] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, . H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," *ArXiv*, vol. 1912.06680v1, 2019.
- [19] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A.

- Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, p. 350–354, 2019.
- [20] Y. Deng, X. Guo, Y. Wei, K. Lu, B. Fang, D. Guo, H. Liu and F. Sun, "Deep Reinforcement Learning for Robotic Pushing and Picking in Cluttered Environment," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 619-626, 2019.
- [21] J. Chen, . S. E. Li and . M. Tomizuka, "Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning," *ArXiv*, vol. 2001.08726v3, 2020.
- [22] M. Bojarski, P. Yeres, A. Choromanaska, K. Choromanski, B. Firner, L. Jackel and U. Muller, "Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car," *ArXiv*, vol. 1704.07911v1, 2017.
- [23] M. Gabbouj, S. Kiranyaz, J. Malik, M. U. Zahid, T. Ince, M. E. H. Chowdhury, A. Khandakar and A. Tahir, "Robust Peak Detection for Holter ECGs by Self-Organized Operational Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 11, pp. 9363-9374, 2022.
- [24] T. Ince, J. Malik, O. C. Devecioglu, S. Kiranyaz, O. Avci, L. Eren and M. Gabbouj, "Early Bearing Fault Diagnosis of Rotating Machinery by 1D Self-Organized Operational Neural Networks," *IEEE Access*, vol. 9, pp. 139260-139270, 2021.
- [25] S. Kiranyaz, J. Malik, H. B. Abdallah, T. Ince, A. Iosifidis and M. Gabbouj, "Self-organized Operational Neural Networks with Generative Neurons," *Elsevier*, vol. 140, pp. 294-308, 2021.
- [26] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar ja D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *ArXiv*, osa/vuosik. 1710.02298v1, 2017.
- [27] J. Malik, "github," GitHub inc, [Online]. Available: <https://github.com/junaidmalik09/fastonn>. [Accessed 9 6 2024].
- [28] J. Malik, S. Kiranyaz ja M. Gabbouj, "FastONN – Python based open-source GPU implementation for Operational Neural Networks," *ArXiv*, osa/vuosik. 2006.02267, 2020.
- [29] "Gymnasium Documentation - Breakout," Farama Foundation, 2023. [Online]. Available: <https://gymnasium.farama.org/environments/atari/breakout/>. [Accessed 7 6 2024].
- [30] araffin, "huggingface.co," huggingface, [Online]. Available: <https://huggingface.co/sb3/a2c-BreakoutNoFrameskip-v4>. [Haettu 12 12 2024].
- [31] Miffyli, "github.com," github, 16 July 2020. [Online]. Available: <https://github.com/DLR-RM/stable-baselines3/pull/110>. [Haettu 12 December 2024].
- [32] K. He, X. Zhang, . S. Ren and J. Sun, "Identity Mappings in Deep Residual Networks," *ArXiv*, vol. 1603.05027v3, 2016.
- [33] S. Kiranyaz, T. Ince, A. Iosifidis and M. Gabbouj, "Generalized Model of Biological Neural Networks: Progressice Operational Perceptrons," *2017 Interantional Joint Conference on Neural Networks*, pp. 2477-2485, 2017.

A PYTHON CODE TEMPLATE FOR TRAINING THE NETWORKS

```

import gym
from stable_baselines3 import A2C
from stable_baselines3.common.vec_env import VecFrameStack
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.env_util import make_atari_env
from fastonn import SelfONN2d
import os
import time
import warnings
import numpy as np
import torch as th
import torch.nn as nn
from gymnasium import spaces
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
from gym.spaces import Box

warnings.simplefilter("ignore", category=DeprecationWarning)
def main(res):
    class CustomCNN(BaseFeaturesExtractor):
        def __init__(self, observation_space: spaces.Box, features_dim: int
= 512):
            super().__init__(observation_space, features_dim)
            n_input_channels = observation_space.shape[0]
            self.cnn = nn.Sequential(
                nn.Conv2d(n_input_channels, 64, kernel_size=11, stride=5,
padding=0),
                # Uncomment SelfONN layers for SelfONN results change q=x
to desired degree
                # SelfONN2d(n_input_channels, 64, 11, q=3, stride=5,
padding=0),
                nn.ReLU(),
                nn.Conv2d(64, 64, kernel_size=7, stride=3, padding=0),
                # SelfONN2d(64, 64, 7, q=3, stride=3, padding=0),
                nn.ReLU(),
                nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0),
                # SelfONN2d(64, 64, 3, q=3, stride=1, padding=0),
                nn.ReLU(),
                nn.Flatten(),
            )
            with th.no_grad():
                n_flatten =
self.cnn(th.as_tensor(observation_space.sample()[None]).float()).shape[1]
                self.linear = nn.Sequential(
                    nn.Linear(n_flatten, features_dim),
                    nn.ReLU()
                )

            def forward(self, observations: th.Tensor) -> th.Tensor:
                return self.linear(self.cnn(observations))

    # Create and wrap the environment
    # replace environment name with the desired environment examples:
'ALE/Breakout-v5' 'ALE/DoubleDunk-v5' 'ALE/MsPacman-v5'
    environment_name = 'ALE/Breakout-v5'

```

```

env = make_atari_env(environment_name, n_envs=4)
env = VecFrameStack(env, n_stack=4)

policy_kwargs = dict(
    features_extractor_class=CustomCNN,
    # can experiment with different features_dim higher is slower in
training but should produce better results
    features_extractor_kwargs=dict(features_dim=512),
)

model = A2C(
    "CnnPolicy",
    env,
    policy_kwargs=policy_kwargs,
    verbose=0,
    # Can experiment with different learning rates and gamma (discount)
values
    learning_rate=1e-4,
    gamma=1.0,
)
policy = model.policy

# Count and print the number of parameters
total_params = sum(p.numel() for p in policy.parameters())
print(f"Total number of parameters in the A2C model: {total_params}")

# Can change iterations and total_timesteps to increase or decrease the
amount of training done model is taught over all the images provided during
all total timesteps iterations are the same as epochs
iterations = 50
total_timesteps = 10000
mean_rewards = np.zeros(iterations)
std_rewards = np.zeros(iterations)
training_times = np.zeros(iterations)
for i in range(iterations):
    start_time = time.time()
    model.learn(total_timesteps=total_timesteps)
    end_time = time.time()
    # Evaluate the policy and save the results to numpy arrays
    mean_reward, std_reward = evaluate_policy(model, env,
n_eval_episodes=10, render=False)
    mean_rewards[i] = mean_reward
    std_rewards[i] = std_reward
    training_times[i] = end_time-start_time
    # print the current epoch and the results
    print(i+1)
    print('mean reward: ', mean_reward)
    print('std reward', std_reward)
    print('time', end_time-start_time)
# directory for the saved results
game = 'Breakout'
model_name = 'A2C'
q = 'base'
result = '_' + str(res)
directorya = 'arrays'
directorym = 'models'

# If the directory does not exist, create it
if not os.path.exists(os.path.join(game, model_name, q, directorya)):

```

```
os.makedirs(os.path.join(game, model_name, q, directorya))

# If the directory does not exist, create it
if not os.path.exists(os.path.join(game, model_name, q, directorym)):
    os.makedirs(os.path.join(game, model_name, q, directorym))

    training_times_path = os.path.join(game,model_name,q,directorya,
'training_times_'+str(total_timesteps*iterations)+result)
    mean_rewards_path = os.path.join(game,model_name,q,directorya,
'mean_rewards_'+str(total_timesteps*iterations)+result)
    std_rewards_path = os.path.join(game,model_name,q,directorya,
'std_rewards_'+str(total_timesteps*iterations)+result)
    models_path = os.path.join(game,model_name,q,directorym,result)

    np.save(training_times_path, training_times)
    np.save(mean_rewards_path, mean_rewards)
    np.save(std_rewards_path, std_rewards)
    model.save(models_path)

# loop the main 10 times to produce 10 different models
for i in range(1):
    main(i+10)
```

B PYTHON CODE TEMPLATE FOR VISUALIZING THE RESULTS

```

import os
import numpy as np
import matplotlib.pyplot as plt

def load_numpy_arrays_from_folder(folder_path):
    # List all files in the folder
    files = os.listdir(folder_path)

    # Filter for .npy files
    npy_files = [f for f in files if f.endswith('.npy')]

    # Initialize a dictionary to hold the loaded arrays
    numpy_arrays = {}

    # Load each .npy file
    for npy_file in npy_files:
        file_path = os.path.join(folder_path, npy_file)
        array_name = os.path.splitext(npy_file)[0]
        numpy_arrays[array_name] = np.load(file_path)

    return numpy_arrays

def filter_and_arrays(numpy_dict):
    # Depending on the number of timesteps taken in the simulations change
    the number below to find the correct directory
    total_timesteps = '500000'
    mean_arrays = {name: array for name, array in numpy_dict.items() if
name.startswith('mean_rewards_'+total_timesteps+'_')}
    std_arrays = {name: array for name, array in numpy_dict.items() if
name.startswith('std_rewards_'+total_timesteps+'_')}
    times_arrays = {name: array for name, array in numpy_dict.items() if
name.startswith('training_times_'+total_timesteps+'_')}

    return mean_arrays, std_arrays, times_arrays

# Change to correct values used in the simulations
total_timesteps = 500000
iterations = 50
timesteps_per_iteration = 10000
x_values = np.linspace(timesteps_per_iteration, total_timesteps,
iterations)

# Change to the correct game directory
game = 'Breakout'
base_array_path = game+"/A2C/base/arrays"
q3_array_path = game+"/A2C_self/3/arrays"
q5_array_path = game+"/A2C_self/5/arrays"
numpy_dict_base = load_numpy_arrays_from_folder(base_array_path)
numpy_dict_q3 = load_numpy_arrays_from_folder(q3_array_path)
numpy_dict_q5 = load_numpy_arrays_from_folder(q5_array_path)
mean_arrays_base, std_arrays_base, times_arrays_base =
filter_and_arrays(numpy_dict_base)
mean_arrays_q3, std_arrays_q3, times_arrays_q3 =
filter_and_arrays(numpy_dict_q3)

```

```

mean_arrays_q5,          std_arrays_q5,          times_arrays_q5          =
filter_and_arrays(numpy_dict_q5)

mean_arrays_base_stack = np.stack(list(mean_arrays_base.values()))
std_arrays_base_stack = np.stack(list(std_arrays_base.values()))
times_arrays_base_stack = np.stack(list(times_arrays_base.values()))

mean_arrays_q3_stack = np.stack(list(mean_arrays_q3.values()))
std_arrays_q3_stack = np.stack(list(std_arrays_q3.values()))
times_arrays_q3_stack = np.stack(list(times_arrays_q3.values()))

mean_arrays_q5_stack = np.stack(list(mean_arrays_q5.values()))
std_arrays_q5_stack = np.stack(list(std_arrays_q5.values()))
times_arrays_q5_stack = np.stack(list(times_arrays_q5.values()))

# Figure for average performance of all the models created
plt.figure(1)
plt.plot(x_values, np.mean(mean_arrays_base_stack, axis=0), label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.mean(mean_arrays_q3_stack, axis=0), label='q3',
linewidth=3, linestyle='--', color = 'green')
plt.plot(x_values, np.mean(mean_arrays_q5_stack, axis=0), label='q5',
linewidth=3, linestyle='--', color = 'yellow')
plt.title(game+' average mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.legend()
plt.show()

# Figure for the median performance of all the models created
plt.figure(2)
plt.plot(x_values, np.median(mean_arrays_base_stack, axis=0), label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.median(mean_arrays_q3_stack, axis=0), label='q3',
linewidth=3, linestyle='--', color = 'green')
plt.plot(x_values, np.median(mean_arrays_q5_stack, axis=0), label='q5',
linewidth=3, linestyle='--', color = 'yellow')
plt.title(game+' median mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.legend()
plt.show()

# Figure for the standard deviation of the rewards of the models created
plt.figure(3)
plt.plot(x_values, np.mean(std_arrays_base_stack, axis=0), label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.mean(std_arrays_q3_stack, axis=0), label='q3',
linewidth=3, linestyle='--', color = 'green')
plt.plot(x_values, np.mean(std_arrays_q5_stack, axis=0), label='q5',
linewidth=3, linestyle='--', color = 'yellow')
plt.title(game+' average std of reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Std of reward')
plt.legend()
plt.show()

print(np.cumsum(np.median(times_arrays_base_stack, axis=0))[-1])
print(np.cumsum(np.median(times_arrays_q3_stack, axis=0))[-1])

```

```

print(np.cumsum(np.median(times_arrays_q5_stack, axis=0))[-1])

# Figure to show the median time to train a model
plt.figure(4)
plt.plot(x_values, np.cumsum(np.median(times_arrays_base_stack, axis=0)),
label='base', linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.cumsum(np.median(times_arrays_q3_stack, axis=0)),
label='q3', linewidth=3, linestyle='--',color = 'green')
plt.plot(x_values, np.cumsum(np.median(times_arrays_q5_stack, axis=0)),
label='q5', linewidth=3, linestyle='--',color = 'yellow')
plt.title(game+' median time per 50000 time steps')
plt.xlabel('Totale timesteps')
plt.ylabel('Median time')
plt.legend()
plt.show()

# Figure to show the minimum rewards across all the models during each
timestep
plt.figure(5)
plt.plot(x_values, np.min(mean_arrays_base_stack, axis=0), label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.min(mean_arrays_q3_stack, axis=0), label='q3',
linewidth=3, linestyle='--',color = 'green')
plt.plot(x_values, np.min(mean_arrays_q5_stack, axis=0), label='q5',
linewidth=3, linestyle='--',color = 'yellow')
plt.title(game+' min mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Min reward')
plt.legend()
plt.show()

# Figure to show the maximum rewards across all the models during each
timestep
plt.figure(6)
plt.plot(x_values, np.max(mean_arrays_base_stack, axis=0), label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, np.max(mean_arrays_q3_stack, axis=0), label='q3',
linewidth=3, linestyle='--',color = 'green')
plt.plot(x_values, np.max(mean_arrays_q5_stack, axis=0), label='q5',
linewidth=3, linestyle='--',color = 'yellow')
plt.title(game+' max mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Max reward')
plt.legend()
plt.show()

# Figure that shows all the base models performances overlapped
plt.figure(7)
for arr in mean_arrays_base_stack:
    plt.plot(x_values, arr)
plt.title('Base models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.ylim(0,30)
plt.show()

# Figure that shows all the third degree self-ONN models performances
overlapped
plt.figure(8)

```

```

for arr in mean_arrays_q3_stack:
    plt.plot(x_values, arr)
plt.title('Q3 models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.ylim(0,30)
plt.show()

# Figure that shows all the fifth degree self-ONN models performances
overlapped
plt.figure(9)
for arr in mean_arrays_q5_stack:
    plt.plot(x_values, arr)
plt.title('Q5 models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.ylim(0,30)
plt.show()

# Figure that shows all the base models standard deviation of the rewards
overlapped
plt.figure(10)
for arr in std_arrays_base_stack:
    plt.plot(x_values, arr)
plt.title('Base models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('STD reward')
plt.show()

# Figure that shows all the third degree self-ONN models standard deviation
of the rewards overlapped
plt.figure(11)
for arr in std_arrays_q3_stack:
    plt.plot(x_values, arr)
plt.title('Q3 models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('STD reward')
plt.show()

# Figure that shows all the fifth degree self-ONN models standard deviation
of the rewards overlapped
plt.figure(12)
for arr in std_arrays_q5_stack:
    plt.plot(x_values, arr)
plt.title('Q5 models mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('STD reward')
plt.show()

last_values_base = mean_arrays_base_stack[:, -1]
max_index_base = np.argmax(last_values_base)
last_values_q3 = mean_arrays_q3_stack[:, -1]
max_index_q3 = np.argmax(last_values_q3)
last_values_q5 = mean_arrays_q5_stack[:, -1]
max_index_q5 = np.argmax(last_values_q5)

# Figure to compare the best model based on the final evaluation for each
case
plt.figure(13)

```

```

plt.plot(x_values, mean_arrays_base_stack[max_index_base], label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, mean_arrays_q3_stack[max_index_q3], label='q3',
linewidth=3, linestyle='--', color = 'green')
plt.plot(x_values, mean_arrays_q5_stack[max_index_q5], label='q5',
linewidth=3, linestyle='--', color = 'yellow')
plt.title(game+' max performing model mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.legend()
plt.show()

min_index_base = np.argmin(last_values_base)
min_index_q3 = np.argmin(last_values_q3)
min_index_q5 = np.argmin(last_values_q5)

# Figure to compare the worst model based on the final evaluation for each
case
plt.figure(14)
plt.plot(x_values, mean_arrays_base_stack[min_index_base], label='base',
linewidth=3, linestyle='--', color='red')
plt.plot(x_values, mean_arrays_q3_stack[min_index_q3], label='q3',
linewidth=3, linestyle='--', color = 'green')
plt.plot(x_values, mean_arrays_q5_stack[min_index_q5], label='q5',
linewidth=3, linestyle='--', color = 'yellow')
plt.title(game+' min performing model mean reward')
plt.xlabel('Totale timesteps')
plt.ylabel('Mean reward')
plt.legend()
plt.show()

```

C PYTHON CODE TEMPLATE TO EVALUATE RANDOM ACTOR FOR A GAME

```

import gym
import numpy as np

# Environment name # Change to the game wanted to evaluate
environment_name = "ALE/MsPacman-v5"

# Create the environment
env = gym.make(environment_name)

# Number of episodes to evaluate
num_episodes = 1000

# Function to evaluate the random action policy
def evaluate_random_policy(env, num_episodes):
    total_rewards = []
    for episode in range(num_episodes):
        state = env.reset()
        done = False
        episode_reward = 0
        while not done:
            action = env.action_space.sample() # Take a random action
            state, reward, done, truncated, info = env.step(action)
            episode_reward += reward
            total_rewards.append(episode_reward)
            print(f"Episode {episode + 1}: Total Reward = {episode_reward}")
    return total_rewards

# Evaluate the random action policy
rewards = evaluate_random_policy(env, num_episodes)

# Print summary statistics
print(f"Average Reward over {num_episodes} episodes: {np.mean(rewards)}")
print(f"Standard Deviation of Reward over {num_episodes} episodes:
{np.std(rewards)}")
print(f"Maximum Reward over {num_episodes} episodes: {np.max(rewards)}")
print(f"Minimum Reward over {num_episodes} episodes: {np.min(rewards)}")

```