Pre-loaded Deep-Q Learning

Tristan Falck, Elizabeth Ehlers

▶ **To cite this version:**

Tristan Falck, Elizabeth Ehlers. Pre-loaded Deep-Q Learning. 12th International Conference on Intelligent Information Processing (IIP), May 2022, Qingdao, China. pp.159-172, 10.1007/978-3-031-03948-5_14 . hal-04178733

HAL Id: hal-04178733

https://inria.hal.science/hal-04178733v1

Submitted on 8 Aug 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

Pre-loaded Deep-Q Learning

Tristan Falck[1] and Elize Ehlers[2]

[1] Kingsway Campus, University of Johannesburg, Auckland Park, Johannesburg, 2092, South Africa
Phone: +27 71 361 8133
[2] Kingsway Campus, University of Johannesburg, Auckland Park, Johannesburg, 2092, South Africa emehlers@uj.ac.za
Phone: +27 11 559 2841

Abstract. This paper explores the potentiality of pre-loading deep-Q learning agents' replay memory buffers with experiences generated by preceding agents, so as to bolster their initial performance. The research illustrates that this pre-loading of previously generated experience replays does indeed improve the initial performance of new agents, provided that an appropriate degree of ostensibly undesirable activity was expressed in the preceding agent's behaviour.

Keywords: Reinforcement learning · Q-learning · Deep-Q learning · Experience replay · Neural networks.

1 Introduction

Deep-Q learning, first implemented by Deepmind Technologies in 2013 [5], served as the first major reconciliation between reinforcement learning and deep learning. Broadly speaking, deep-Q learning entails that an agent receives reward or penalty signals according to its performance within an environment - neural network technologies which control the agent use these signals to learn the behaviour which maximises the expected rewards [5].

Since its introduction, deep-Q learning has come to include techniques aimed at stabilizing the training of the constituent neural network. Among the first was allowing the agent in question an experience replay memory buffer [5, 9]; this circular buffer allows the agent to sample from previous experiences within its environment when learning, which diversifies and decorrelates its training data [5, 9].

This paper explores another potential use of the experience replay memory, specifically how it can be used to bolster the initial performance of a totally untrained agent through sharing the memories of a preceding agent. Section 2 of the paper explores the provenance of Q-learning, neural networks and deep-Q learning, respectively. Section 3 explores the agent- and environment-designs used for the experiments, as well as the nature of the experiments themselves. Section 4 illustrates the results of pre-loading the memory buffers of deep-Q learning agents, and section 5 provides an interpretation thereof.

2 Background

2.1 Q-learning

Foundations Provided that an agent's environment is stochastic and constituted by Markovian transitions[3] and additive rewards, the environment is considered a *Markov Decision Process* (MDP) problem [7]. Among the most fundamental tools in solving MDPs, which could also be considered the most popular equation in the field of reinforcement learning, is the *Bellman equation* [7] described by equation 1.

$$U(s) = max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')] \tag{1}$$

The Bellman equation defines a harmony between the state-utilities of an MDP. Specifically, for a state s the utility $U(s)$ is defined as the expected value of the reward $R(s,a,s')$ for the transition between s and the next state s' summed with the product between the utility of the next state $U(s')$ and a discount factor γ, provided that the agent chooses optimal actions. In this way, the Bellman equation recursively defines the utility of a state in terms of the utility of the best following state.

The Bellman equation was intended for MDPs, which are inherently stochastic [7]. The equation may, however, also model deterministic environments. In deterministic contexts, the transition to a state s' from s given an action a is fixed, thus removing the need to account for expected values. Hence, the equation simplifies to the representation shown by equation 2. Given that this research deals in deterministic problems, the deterministic representation of the Bellman equation is used moving forward.

$$U(s) = max_a[R(s,a,s') + \gamma U(s')] \tag{2}$$

Despite its usefulness, the Bellman equation defines the utility of a state in terms of the corresponding optimal action available to the agent. The equation does not, therefore, account for ostensibly suboptimal actions. We can recover more specific information through a *Q-function* $Q(s,a)$ which returns the expected utility of taking an action a within state s [7]. We can thus redefine the deterministic Bellman equation so as to replace the utility function with the Q-function as follows.

$$Q(s,a) = R(s,a,s') + \gamma max_{a'}[Q(s',a')] \tag{3}$$

Equation 3's representation of the Q-function is either recovered (if possible) or approximated in the process of *Q-learning* [7], wherein an agent uses the maximal Q-values across encountered states to define an optimal policy.

[3] A transition between a state s and s' is *Markovian* provided that it is dependent only on s, and not preceding states [7].

Temporal Difference Q-learning Temporal difference Q-learning entails recovering the Q-function through value iteration; this could be thought of as the quintessential implementation of Q-learning. For each transition, the *temporal difference* (TD) described by equation 4 is used to update the preceding Q-value as described by equation 5 [7]. In this context, α is known as the *learning rate* [7].

$$TD = R(s, a, s') + \gamma max_{a'}[Q(s', a')] - Q(s, a) \tag{4}$$

$$Q(s, a) \leftarrow Q(s, a) - \alpha \cdot TD \tag{5}$$

Assuming that ϵ-greedy exploration (see algorithm 2) is used, we may describe the full temporal difference Q-learning algorithm as follows. We assume that an epsilon value of $\epsilon \in [0, 1]$ is chosen and that $A(s)$ represents the set of actions available to the agent for the state s.

Algorithm 1 Temporal Difference Q-learning

1: **while** s is not terminal **do**
2: $a \leftarrow \epsilon$-greedy(s)
3: $Q(s, a) \leftarrow Q(s, a) - \alpha \cdot [R(s, a, s') + \gamma max_{a'}[Q(s', a')] - Q(s, a)]$
4: **end while**

Algorithm 2 ϵ-greedy Exploration Strategy

1: Randomly generate $r \in [0, 1]$
2: **if** $r > \epsilon$ **then**
3: $a \leftarrow argmax_a Q(s, a)$
4: **else**
5: $a \leftarrow$ random value from $A(s)$
6: **end if**

Temporal difference Q-learning recovers a tabular approximation of the Q-function known as a Q-table, which is to say that a table mapping a state s to the corresponding Q-values $\{Q(s, a) \mid a \in A(s)\}$ for each state in the environment must be maintained. In this regard, temporal difference Q-learning can be thought of as approximating the agent function rather than the agent program. Herein lies one of the key weaknesses of temporal difference Q-learning: the space-complexity of the Q-table grows linearly with the number of states in the environment. For state-spaces consisting of more than 10^6 states [7], this approach converges far too slowly assuming it does not become entirely intractable. Additionally, temporal difference Q-learning does not allow for generalization across similar states [5]. These shortcomings of temporal difference Q-learning serve as the impetus for more sophisticated means of approximating the Q-function, such as deep-Q learning.

2.2 Neural Networks

Foundations The history of neural networks can be traced back to linear regression, one of the simplest machine learning algorithms. The first step came in evolving the concept of linear regression into perceptron learning - this was achieved through passing the output of the linear regression model through a threshold function, which would output one if the threshold was surpassed and zero otherwise [7]. Perceptron learning was capable of solving problems of binary classification, provided that the data were linearly separable [7].

The next evolution came in composing perceptrons into layered architectures, where adjacent layers of perceptrons were connected through weighted edges. This structure, known as a multilayer perceptron, allowed the inputs of later perceptrons to be the weighted summations - with the addition of a bias variable - of previous perceptrons' outputs [6]. A multilayer perceptron is a type of neural network; a neural network describes a network of units or neurons each equipped with some activation function through which the unit's input is passed, and so multilayer perceptrons can be viewed as neural networks possessing threshold activation functions. Despite the apparent simplicity of a perceptron unit, multilayer perceptrons are capable of realizing *NAND* gates [6], and can therefore perform any realizable computation if made large enough.

Traditional perceptron learning came to evolve into logistic regression, which saw the perceptron's hard threshold function replaced with the smoother sigmoid function. Unlike perceptron learning, logistic regression could facilitate the gradient descent optimization algorithm since the sigmoid function is differentiable [7]. In the same manner, and with the same impetus, multilayer perceptrons evolved into more contemporary neural networks. The threshold activation functions within the network's units were replaced with sigmoid activation functions, allowing the neural network to exhibit the complex computational power of the multilayer perceptron while also facilitating gradient descent. Key to implementing gradient descent within neural networks, however, was the back-propagation algorithm developed by Hinton et al. in 1986 [8], which yields the requisite partial derivatives for gradient descent.

Today, the study of neural networks continues to elucidate their potential: techniques on regularization and new activation functions such as ReLu allow for the creation of deeper neural networks [6], and more specific architectures are illustrating their capabilities, such as with convolutional neural networks within the field of computer vision [7].

Functionality and Algorithms Provided that a neural network has at least two layers, with the first of which possessing a nonlinear activation function, it will conform to the universal approximation theorem [7], which is to say that it may approximate any continuous function to an arbitrary degree of accuracy. This property lends neural networks the flexibility to solve both classification- and regression-based problems of numerous kinds.

The inference and training of a feedforward[4] neural network are performed by forward-propagation and back-propagation, respectively. For a layer l in the network, we define σ^l as its activation function, w^l as its weight matrix, b^l as its bias vector and a^l as its output vector. Forward-propagation, through which the network process input data into a prediction, is described by algorithm 3.

Algorithm 3 Forward-propagation for input x

1: $a^{l_{input}} \leftarrow x$
2: **for** $l \leftarrow 1, l_{output}$ **do**
3: $\quad a^l \leftarrow \sigma^l(a^{l-1} \cdot w^l + b^l)$
4: **end for**
5: **return** $a^{l_{output}}$

Neural networks train through optimization algorithms such as stochastic gradient descent or the more recent Adam optimization. Requisite to the more popular optimization techniques is the back-propagation algorithm; this provides the partial derivatives of the loss function with respect to all the weight- and bias-variables within the network [8]. We define L as the loss function and δ^l as the error of a layer[5] l. We further define w^l_δ and b^l_δ as summing the partial derivatives of the loss w.r.t the weights and biases - respectively - for layer l across multiple training examples. The back-propagation algorithm is detailed by algorithm 4, assuming that an input x has been forward-propagated [6].

Algorithm 4 Back-propagation for input x

1: $\delta^{l_{output}} \leftarrow \nabla_a L \odot a'$
2: **for** $l \leftarrow l_{output} - 1, 1$ **do**
3: $\quad \delta^l \leftarrow ((w^{l+1})^T \cdot \delta^{l+1}) \odot a'$
4: **end for**
5: $w^l_\delta \leftarrow w^l_\delta + \delta^l \cdot a^{l-1}$
6: $b^l_\delta \leftarrow b^l_\delta + \delta^l$

As mentioned, back-propagation is requisite to many of the optimization algorithms used in training neural networks. This project, however, will make use of stochastic gradient descent (SGD), specifically. Lastly, we define the learning rate of the neural network as η [6] in order for the training of a neural network to be described by algorithm 5.

[4] A neural network containing only forward-connections [7].

[5] This error is technically defined as a vector of the derivatives of the loss function w.r.t the layer's weighted summation which is passed through the activation [6].

Algorithm 5 Neural network training with SGD

```
1:  for batch ← batches do
2:      for x ← batch do
3:          Forward-propagate x                         ▷ See algorithm 3
4:          Back-propagate                              ▷ See algorithm 4
5:      end for
6:      for l ← 1, layers do
7:          w^l ← w^l − (η / length(batch)) w^l_δ
8:          b^l ← b^l − (η / length(batch)) b^l_δ
9:      end for
10: end for
```

2.3 Deep-Q Learning

Foundations Given its intractibility for large state-spaces and inability to generalize, temporal difference Q-learning is very seldomly implemented for realistic applications; other means of approximating the Q-function are necessary. Any number of machine learning algorithms may be used to approximate the Q-function (such as linear models, which have been used for this purpose before [5]). Neural networks, however, are especially good candidates given their conformation to the universal function approximation theorem. That is, the complexity and dimensionality of the Q-function do not serve as obstacles to the neural network approximating it.

Mnih et al., 2013 patented the so-called *deep-Q learning* algorithm in their paper *Playing Atari with Deep Reinforcement Learning* [5], which entailed that the Q-function approximation was facilitated by a convolutional neural network. The algorithm was trained to play seven distinct Atari games given only raw pixel data from the games as input. At the time, this deep-Q learning system outperformed all previous machine learning approaches for six of the Atari games, and surpassed expert human-level play for three of them [5].

Experience Replay Memory As originally highlighted in the paper which introduced deep-Q learning, a key obstacle faced by the learning algorithm is the stability of the contained neural network's training. Most deep learning architectures require that their training data is *independent and identically distributed* (i.i.d.) [5, 7, 9]. This property certainly does not hold for unaltered Q-learning data; not only is $Q(s, a)$ dependent on $Q(s', a')$ for transitionally adjacent states s and s', the former is defined explicitly in terms of the latter (see equation 3). This strong correlation throughout the Q-values necessitates mechanisms through which the data can be made to be i.i.d..

The first mechanism used to combat dependency within Q-learning data was the inclusion of an *experience replay memory buffer*, the notion of which was first introduced by Long-ji Lin in 1992 [4]. The experience replay is a circular *LIFO*[6] data buffer which tracks (s, a, r, s') tuples, the entries of which correspond

[6] Last-in first-out

to a state, action taken, reward received and resultant state, respectively [4]. For each of the deep-Q learning agent's training steps, the agent will randomly sample multiple 'experience' tuples from its replay memory and use those as its training data. This inclusion of an experience replay memory redistributes and thus diversifies the data recovered from the agent's environment, making it an invaluable tool for stabilizing deep-Q learning.

The ability of an agent to encode past experiences into its training also entails a less obvious utility: the experience replay makes the agent more robust against catastrophic forgetting [7].

Target Network Unlike with supervised applications of neural networks, deep-Q learning is not provided with a stationary ground-truth for its training; rather, the deep-Q system backpropagates an error it itself generates through the Bellman equation [5]. The target of a deep-Q network - therefore - is intrinsically non-stationary. While this variation within the target cannot be circumvented, it can be mitigated through the use of a *target network*. The target network within a deep-Q learning system is a copy of the primary network which is only synchronised therewith at certain intervals - this more stable network is delegated the task of generating the truth-values to be used in for training.

Contemporary Deep-Q Learning Following its advent, deep-Q learning has come to include numerous techniques aimed at improving its stability and policy generation. Shaul et al., 2016 improved the foundational experience replay through prioritized sampling [9,10]; that is, the likelihood of an experience being sampled therefrom was made proportional to that experience's significance. It was demonstrated that deep-Q learning agents which made use of prioritized experience replays outperformed those without in 41 of the 49 Atari games used for assessment [10].

Another advance within the field was that of double deep-Q learning, which adjusted the loss calculation of the constituent neural network so as to combat overestimation bias [3,9]. Dueling deep-Q learning - on the other hand - cleaved the agent's neural network into two streams following the initial layers, and each stream was delegated the task of regressing segments of a decomposed Q-function [9,12]. This allowed for the lose inference of a state's value without the need for coupling it with some specific action [9,12].

3 Implementation

3.1 Environment: The *Snake* Game

This research aims to assess the potentiality of pre-training deep-Q learning agents with the experience replay memories of preceding agents, thus allowing the new agents some degree of competence prior to any interfacing with their environments. Consequent experiments therefore require an environment into which deep-Q learning agents may be deployed for training and assessment.

Candidate environments for such reinforcement-learning systems vary greatly in nature, ranging from turn-based board games to the Atari games used by Deepmind's original implementation of the deep-Q learning algorithm [5,7]. Generally speaking, the most popular environments used are simple in nature - such as the pole-balancing problem [7] or the multi-armed bandit problem.

This research makes use of the *Snake* game as the agents' environment. The game was chosen as it is simplistic and its rules easily understood, but - given that the enponymous snake's tail increases in length as it performs well within the game - the difficulty of the game scales roughly with the agent's performance. For an illustration of the game, please see figure 1.

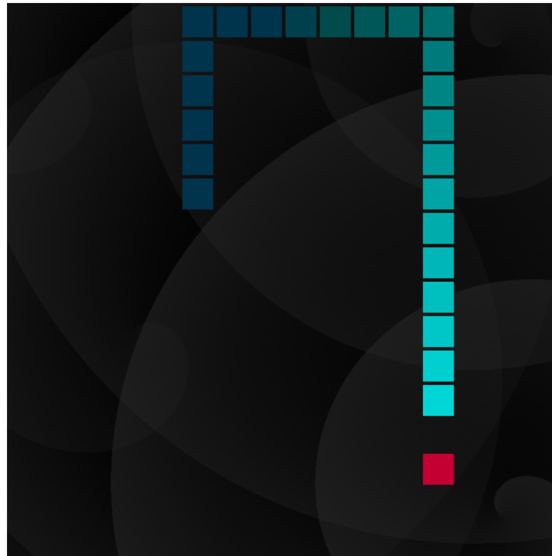

Fig. 1. A cropped screenshot from the program illustrating the *Snake* game being played by a well-trained agent.

Rules of the *Snake* Game The rules of *Snake* vary across implementations, however this research makes use of the following rule-set:

- The snake (agent) may move up, down, left or right at any given frame (that is, the snake may move backwards into its tail at any moment, differentiating the game from its regular human-playable implementation).
- The snake may not collide with a wall or its tail, as doing so will reset the game and the current score.
- Upon finding the 'food' object within the game, the score will increment, the snake's tail will grow by a single unit and a new 'food' object will be initialized randomly within the map.

Reward Structure In order to facilitate reinforcement learning within the system's agents, reward- and punishment-signals must be recovered from the environment according to their actions [5,7]. These rewards - as is the case with most reinforcement learning implementations - are scalar in nature, and are for this research expressed as integer values. Numerous potential reward structures exist, however this research made use of an adapted version of that proposed by Hennie de Harder, 2020 [2] in her own application of deep reinforcement learning to the game of Snake. Specifically, the actions and corresponding rewards are detailed in table 1.

Table 1. Table detailing the rewards and punishments (right) corresponding to given actions (left).

Snake moves closer to food	$R = 1$
Snake moves further from food	$R = -2$
Snake retrieves food	$R = 10$
Snake collides with wall or tail	$R = -100$

3.2 Agents

Neural Networks and Q-networks The system's agents themselves are deep-Q learning agents, which is to say that a given agent achieves function-approximation Q-learning through neural network technologies and an experience replay memory buffer [5,7]. Unlike the original deep-Q learning algorithm - which made use of convolutional neural networks to facilitate the learning - this research saw the implementation of feedforward artificial neural networks. A neural network framework was created and assessed in a general context prior to its deployment within the deep-Q learning system; specifically, a neural network was created using the framework and assessed on the MNIST dataset of handwritten digits, upon which it achieved an accuracy of roughly 95% accuracy. The framework was also successfully evaluated on a series of regression-based tasks, given that deep-Q learning ultimately entails that the networks regress Q-values.

For the implementation used within this research, neural networks used by the system were wrapped into new Q-network objects, simplifying communication between the neural network and its containing agent.

State Structure and the Experience Replay Buffer At each frame of the game's operation, the agent recovers a binary string constituting the state-structure is recovered from the environment. Each bit within the state-structure binary string - which was proposed by Hennie de Harder, 2020 [2] - represents some component of the game's current properties relative to the agent, and the input layers of neural networks are designed so as to directly receive the string

as input. The exact nature of the state-structure binary string - along with the resultant actions of the agent - is illustrated by figure 2.

At any given frame, the immediate state-structure information is unlikely to be passed directly into the deep-Q learning agent's neural network, as it is initially saved into the experience-replay memory buffer. Given that this implementation sees the experience replay implemented as a circular buffer, the new binary string will replace the oldest in the replay. At each frame of the game's operation, a random sample is drawn from the replay to generate a training dataset for the main neural network controlling the agent. For this implementation, the experience replay buffer was made to hold 25000 experience entries, and each random training sample would consist of 100 experiences.

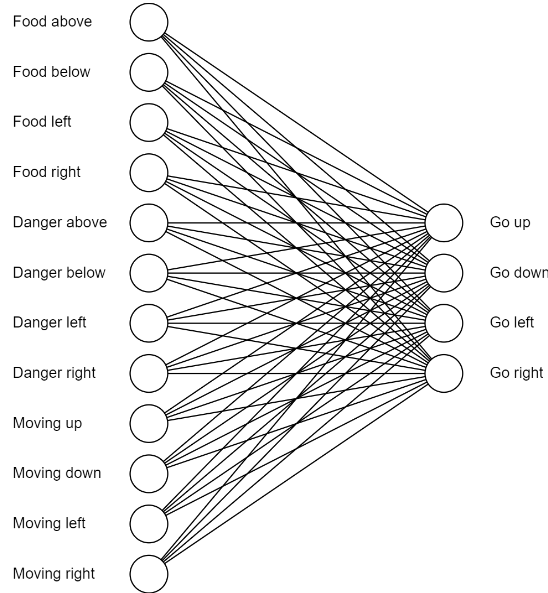

Fig. 2. The state-structure binary string proposed by Hennie de Harder, 2020 [2], along with the actions to which the regressed Q-values of the illustrated neural network map (for simplicity, hidden layers have been omitted the diagram).

Exploration Strategy: The Epsilon-greedy Algorithm As can often be the case with reinforcement learning problems [7], the exploration/exploitation dilemma arises in the context of deep-Q learning. Modern strategies aimed at handling the dilemma can entail the tempering of exploration according to time such as with simulated annealing [7], or the dilemma could be handled probabilistically through - for example - the *softmax* function [11]. This probabilistic approach by way of *softmax* is especially desirable for Q-learning, since the mag-

nitude of regressed Q-values is generally proportional to the agent's confidence within that action.

The attractiveness of more sophisticated exploration strategies notwithstanding, this research makes use of the ϵ-greedy strategy (see algorithm 2) - which entails the hard-coding of an ϵ-value to weight exploration and exploitation. This decision was made as the degree of exploration had a significant effect on the results of the research, and was thus subjected to manual engineering.

3.3 Pre-loading Experience Replays

In order to evaluate the potentiality of pre-loading new agents with preceding agents' replay memory buffers, the buffers themselves were implemented as serializable objects. Upon the instantiation of a new agent, the program allows for it to be fitted with a preceding agent's replay memory buffer which had been written to disk. For each of the experiments performed, a pre-trained agent was made to act within the environment until its replay memory buffer had been fully populated, after which the replay was written to disk.

The new agents, in possession of the pre-loaded replays, were made to sample from the replays and train for each of the entries within the replay (that is, 25000 training cycles) prior to their true deployment into the game. Upon their true deployment, the agents' neural networks were given deterministic control over the snake ($\epsilon = 0$). Three primary experiments were performed within this research, each of which adjusted the agent's behaviour through tuning the ϵ-value (and - in turn - also adjusted its experience replay buffer). Each of the neural networks within the tested agents were constituted by a 12-neuron input layer, two 8-neuron hidden layers and a 4-neuron output layer.

Experiments

- $\epsilon = 0.0$: The preceding agent's primary neural network was given deterministic control of the snake without any exploration.
- $\epsilon = 0.4$: 40% of the agent's actions were random, with the remaining 60% of the actions being controlled by the primary neural network within the system.
- $\epsilon = 1.0$: The snake behaved entirely randomly with no input from the primary neural network.

3.4 Technical Details of Implementation

The neural network and Q-network frameworks used within the system were created using the Java programming language, inclusive of the linear-algebra- and calculus-based functionality constituent to their operation. The *Snake* game and illustrations of both the neural network and the agent's performance were created through the JavaFX platform. In order to record results, the Python scripting language along with the *Matplotlib* library were used.

4 Results

The research successfully proved that pre-loading a deep-Q learning agent's replay memory buffer does indeed bolster its initial performance upon deployment. Figure 3 compares the initial performances of two agents, with one in possession of a pre-loaded replay; the average reward is observably higher across the activity period for the agent with the pre-loaded replay.

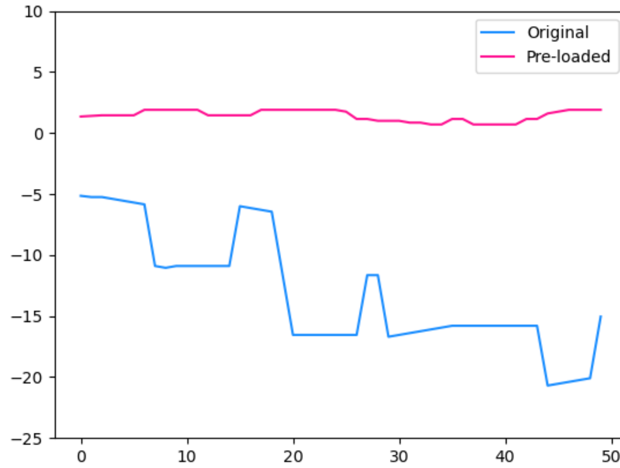

Fig. 3. A comparison between the performance of a regular deep-Q learning agent over its initial 10,000 frames of activity and that of one equipped with a pre-loaded replay memory buffer ($\epsilon = 0.4$), with the ϵ-values of each agent having been set to 0.0.

Unexpectedly, the success of pre-loading an agent's replay memory buffer was heavily dependent on the value of ϵ in the ϵ-greedy exploration strategy. Notably, the new agent's initial performance seemed to scale with the degree of randomness present within the activity of the preceding agent. Figure 4 compares the initial performances of three agents possessing pre-loaded memory buffers, and illustrates that the more stochasticity was in the preceding agent's activity (and thus its replay), the better the new agent performed.

5 Analysis and Critique

Interpretation of Effect of ϵ on New Agent Performance The effect of ϵ on the performances of agents possessing pre-loaded replay memory buffers was unexpected, though not inexplicable: an initial observation within the field of

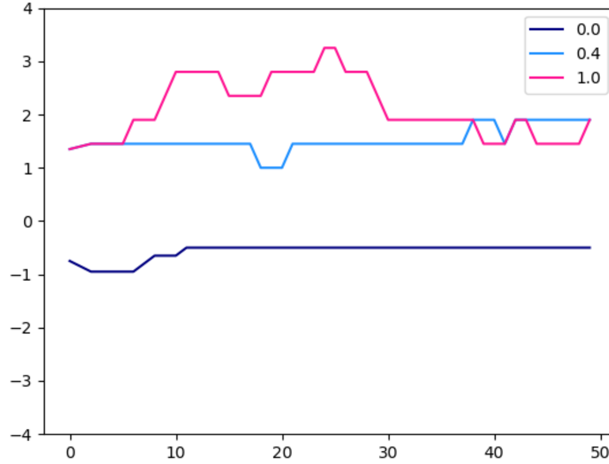

Fig. 4. The initial performances of three agents each in possession of pre-loaded replay memories. The experience-replay memories were generated by agents with $\epsilon = 0.0$, $\epsilon = 0.4$ and $\epsilon = 1.0$, respectively. The performance of each agent was measured over its initial 10,000 frames, with the ϵ-values of each agent having been set to 0.0.

reinforcement learning is that pure master-generated data is not enough, and agents need to know what the 'wrong' answers are as well as what the 'right' answers are [7].

Ultimately, the new agent performed best when its preceding agent behaved entirely randomly ($\epsilon = 1.0$). On the other hand, the new agent performed poorly if the preceding, well-trained agent was given deterministic control of the snake ($\epsilon = 0.0$). When considering how the deep-Q agents regress Q-values, this makes some intuitive sense. At any given time, the agent should regress at most two positive Q-values and at least two negative values for the current state (there are always at least two undesirable actions within the game). The deterministic agent's replay buffer includes very few mistakes, and so the new agent is less likely to be capable of regressing the negative Q-values. When the preceding agent behaves randomly, however, the new agent is transitively exposed to negative reward signals, thus learning to regress negative Q-values for undesirable actions.

Despite its success, it is hypothesised that pre-loading an entirely random replay memory buffer is better due to the nature of the *Snake* game - at any given moment, at least a quarter of the actions available to the agent are desirable. In a more complex context where such desirable actions are less common, lower ϵ-values, and thus lower degrees of randomness from the preceding agents, is expected to outperform entirely random replay memory buffers.

Future Work A sensible next step in exploring pre-loaded replay memory buffers is to do so in more complex environments - those where the proportion of desirable actions is far exceeded by that of undesirable actions. A replay memory buffer encodes no assumptions as to its own generation, and so the exact nature of the replays to be pre-loaded could be augmented in two ways. Firstly, multiple replay buffers from different agents could be concatenated into a single, massively pre-aggregated replay memory buffer, upon which a new agent could train. Secondly, some of the experiences within the resultant massive replay could be generated by humans; provided that an adequate number of negative experiences (those with negative rewards) are present within the massive replay, the experiences aimed at teaching the agents the 'right' answers could be human-generated.

6 Conclusion

This research explored deep-Q learning and its provenance, and assessed the potentiality of pre-loading a new deep-Q learning agent's experience replay memory buffer with that of a previous agent. Neural network and Q-network frameworks were created, along with the toy environment of the *Snake* game within which agents could be assessed. The research illustrated that pre-loading experience replay memory buffers within new agents does bolster their initial performance upon their deployment. Given that such agents generally require adequate exposure to negative reward signals, the pre-loading of memory replays only illustrated desirable results provided that there was an adequate degree of randomness within the preceding agent's behaviour.

References

1. Dittrich, M. and Fohlmeister, S., (2020). A deep q-learning-based optimization of the inventory control in a linear process chain. Production Engineering,.
2. de Harder, H. (2020). Snake Played by a Deep Reinforcement Learning Agent [Internet]. Towards Data Science. [cited 2 November 2021]. Available from: `https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36`
3. van Hassalt, H., Guez, A. and Silver, D., (2015). Deep Reinforcement Learning with Double Q-learning.
4. Lin, L., (1992) Reinforcement Learning for Robots Using Neural Networks. Carnegie Mellon University, USA.
5. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., (2013). Playing Atari with Deep Reinforcement Learning.
6. Nielsen, M., (2015). Neural Networks and Deep Learning. Determination Press.
7. Russell, S. and Norvig, P., (2020). Artificial Intelligence: A Modern Approach. 4th ed. Upper Saddle River: Pearson.
8. Rumelhart, D., Hinton, G. and Williams, R., (1986). Learning Representations by Back-propagating Errors. Nature, 323(6088), pp.533-536.
9. Sanghi, N., (2020). Deep reinforcement learning with Python. 1st ed. Bangalore, India: Apress.

10. Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritized Experience Replay. ICLR 2016,.
11. Tokic, M. and Palm, G., (2011). Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax. KI 2011: Advances in Artificial Intelligence, pp.335-346.
12. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N., (2016). Dueling Network Architectures for Deep Reinforcement Learning.