applied sciences

MDPI

Article

Boosting Deep Reinforcement Learning Agents with Generative Data Augmentation

Tasos Papagiannis, Georgios Alexandridis and Andreas Stafylopatis *

School of Electrical & Computer Engineering, National Technical University of Athens, Zografou Campus, 15780 Athens, Greece; tasos@islab.ntua.gr (T.P.); gealexandri@islab.ntua.gr (G.A.)
* Correspondence: andreas@cs.ntua.gr

Abstract: Data augmentation is a promising technique in improving exploration and convergence speed in deep reinforcement learning methodologies. In this work, we propose a data augmentation framework based on generative models for creating completely novel states and increasing diversity. For this purpose, a diffusion model is used to generate artificial states (learning the distribution of original, collected states), while an additional model is trained to predict the action executed between two consecutive states. These models are combined to create synthetic data for cases of high and low immediate rewards, which are encountered less frequently during the agent's interaction with the environment. During the training process, the synthetic samples are mixed with the actually observed data in order to speed up agent learning. The proposed methodology is tested on the Atari 2600 framework, producing realistic and diverse synthetic data which improve training in most cases. Specifically, the agent is evaluated on three heterogeneous games, achieving a reward increase of up to 31%, although the results indicate performance variance among the different environments. The augmentation models are independent of the learning process and can be integrated to different algorithms, as well as different environments, with slight adaptations.

Keywords: data augmentation; deep reinforcement learning; generative models; Arcade Learning Environment; diffusion models

Citation: Papagiannis, T.; Alexandridis, G.; Stafylopatis, A. Boosting Deep Reinforcement Learning Agents with Generative Data Augmentation. *Appl. Sci.* **2024**, *14*, 330. https://doi.org/10.3390/app14010330

Academic Editor: Andrea Prati

Received: 30 November 2023
Revised: 22 December 2023
Accepted: 26 December 2023
Published: 29 December 2023

1. Introduction

Recently, deep learning models have demonstrated a significant performance boost when data augmentation techniques are employed, mainly in tasks where it is difficult to obtain high-quality actual data [1,2]. In such cases, synthetic samples are produced out of real ones, in order to tackle dataset imbalance and enlarge small datasets, as the efficiency of deep learning algorithms is highly dependent on the data used during training. Under this scope, several methodologies have been proposed, mainly in the computer vision domain, ranging from geometric transformations [3,4] to more advanced deep learning approaches [5–7]. Tasks such as image classification and object detection are shown to benefit significantly from data augmentation, highlighting the importance of diverse, high-quality training samples.

In the context of reinforcement learning (RL), the generalization and convergence speed of the agent is one of the main subjects of research, as many approaches struggle to efficiently utilize the available data [8,9], especially in environments with sparse rewards. Thus, several attempts have been made to augment the datasets used for the agent's training, achieving quite satisfactory results and indicating that data augmentation is a promising direction in RL [10–13]. Specifically, in RL environments with visual observations (where the state representation is an image), most existing approaches aim to augment data by processing the original images with traditional techniques such as cropping, flipping, etc. [12,14]. Although, these methods increase the model's generalization ability, they lack diversity in the sense that the new data arise directly from existing samples.

In this work, a framework is presented for creating novel synthetic samples by learning the distribution of the original images (states) rather than processing existing observations. The main idea is to generate observations as similar as possible to states the agent could encounter in the future and use them at earlier stages of the agent's training. Depending on the examined environment, states that provide useful information (basically in terms of obtained rewards) may be observed after many timesteps, delaying the progress of training. On top of that, the learned policy determines, to a high degree, the observed states. This interdependence between the policy and the training samples may lead to local optima and prevent the agent from improving its policy. In order to overcome this issue, a generative process is proposed to enhance the diversity of the samples and speed up training by mixing synthetic and real samples in each batch.

Another environment-specific issue that may arise in RL tasks is the imbalance of the training set (with respect to the rewards of the collected samples). Similarly to class-imbalanced supervised tasks, in environments with sparse rewards the agent is trained on a dataset in which the majority of samples are neutral (i.e., the immediate reward is zero), making the distinction among the qualities of different actions quite complex. Under this scope, generative augmentation can be used to enrich the original set (consisting of samples collected by the agent during its interaction with the environment) with samples of a specific category (e.g., states with high reward). Based on the technique introduced in this paper, samples can be produced independently of the type of observed data and augment the dataset towards a certain direction, limiting the effect of class imbalance in the agent's performance.

The proposed methodology is split into two phases. At first, a diffusion model is used to generate images that represent the current state and the next state of a sample. Subsequently, a custom model for predicting the agent's action that was executed in between is employed, while the reward is determined based on the data that were used for the training of the diffusion model. The produced samples are stored in a separate buffer and are used along with real ones during the agent's training to indirectly boost exploration by representing completely novel (or less frequently encountered) states. The presented augmentation models are designed for the Atari 2600 environment, though they can be straightforwardly adapted to any environment with slight modifications. Specifically, in this work, training with augmented data is performed on three different games provided by the aforementioned environment. The proposed methodology improved (or performed similarly to) the classic model-free deep Q-network (DQN) agent (which is used as the base learning algorithm), as well as augmentation-based variants, in terms of the maximum obtained reward and the timesteps needed to achieve it.

The contributions of this work can be summed up as follows;

1. Firstly, data augmentation is performed on visual state representations of RL environments in a generative manner, leading to an increase in the samples' diversity.
2. Additionally, data synthesis can be guided by selecting the training data of the diffusion models in order to generate novel, yet realistic, samples of a specific behavior (in our context, samples of either high or low immediate reward). In this respect, the training data may be more balanced, as, in most cases, state–action pairs with an immediate impact on the obtained reward are quite rare.
3. Finally, the proposed augmentation technique is independent of the core learning algorithm, making it applicable to different agent architectures, as well as training environments.

The remainder of this paper is structured as follows: Section 2 briefly presents fundamental definitions and terminology on agent-based systems and diffusion models, while Section 3 overviews related approaches in both data augmentation and dynamics models for reinforcement learning environments. In Section 4, the proposed augmentation methodology is described along with details on the implementation of the models and the agent's training process. The experimental setup is presented in Section 5 and the obtained

results are analyzed in Section 6. Finally, Section 7 concludes and discusses potential future extensions of the proposed framework.

2. Background

2.1. RL as Markov Decision Process

A reinforcement learning task where, considering a sequence of discrete timesteps t, an intelligent agent makes decisions based on its interaction with the environment can be described as a Markov decision process (MDP) [15]. Formally, an MDP is represented by a tuple $\langle S, A, P, R, \gamma \rangle$ where:

- $\mathbf{S}$ is the state space, i.e., the set of all possible states;
- $\mathbf{A}$ is the action space, i.e., the set of all possible actions;
- $\mathbf{P_a}(\mathbf{s}, \mathbf{s'}) : S \times A \times S \rightarrow [0, 1]$ is the transition function which determines the probability that given a current state s and an action a, the next state will be s'. Thus, $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$;
- $\mathbf{R_a}(\mathbf{s}) : S \times A \rightarrow \mathbb{R}$ is the reward function which determines the immediate reward obtained by executing action a while being in state s;
- $\gamma \in [0, 1]$ is a discount factor used to calculate the expected return $\mathbb{E}[G_t]$.

In this context, the Markov property is satisfied, as the transition from a state s to a state s', as well as the immediate reward r after executing an action a, depend solely on this action and the state s (Equation (1)). In this respect, the state representation at each timestep t describes completely the properties of the state and it is independent of the path that led to it.

$$P(s_{t+1} = s', r_{t+1} = r | s_t, a_t) = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, \ldots, r_1, s_0, a_0) \quad (1)$$

At each discrete timestep t, the agent selects an action $a_t \in A$, under a state representation $s_t \in S$, and transits to a new state $s_{t+1} \in S$ according to the environment's dynamics, as determined by P. During this transition, the agent obtains an immediate reward $r_t \in R$ based on the properties of the specific environment. The goal of the agent is to determine a policy for selecting actions that maximizes the expected return value (Equation (2)):

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[G_t | s_t, a_t] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t, a_t] \quad (2)$$

for each state–action pair, that is a policy π such that $Q_\pi(s, a) \geq Q_{\pi'}(s, a)$ for all $s \in S$, $a \in A$ and any possible policy π'. The role of γ is to progressively reduce the effect of future rewards, yet take them into account in the expected reward. Normally γ takes values in the range $[0, 1]$; in the extreme case of $\gamma = 0$ only the immediate reward is considered, while in the case of $\gamma = 1$ all rewards are treated equally, independently of when they are obtained. The optimal Q-value function $Q^*(s, a) = \max_\pi Q_\pi(s, a)$ satisfies the Bellman optimality equation (Equation (3)):

$$Q^*(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})] \quad (3)$$

which indicates that, given the Q-value is known for all actions in the next state, the best action in the current state is the one maximizing the quantity $r + \gamma Q^*(s', a')$. In order to determine the Q-value function in a real reinforcement learning task, a neural network is used as a function approximator such that $Q(s, a; \theta) \approx Q^*(s, a)$, where $s \in S$ is a state representation of the environment, $a \in A$ is a possible action, and θ are the weights of the model. The loss function (which is derived from the Bellman optimality equation) used for the neural network's training in the deep Q-learning algorithm is as follows (Equation 4)):

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta^-))^2] \quad (4)$$

The samples (s, a, r, s') are drawn uniformly from a replay buffer D, where they are stored during the agent's interaction with the environment. As shown in Equation (4), the target depends on the network being trained. In order to make the training process more stable, the target network's weights θ^- are held fixed for a specific number of iterations and are updated periodically.

2.2. Denoising Diffusion Probabilistic Models

Denoising diffusion probabilistic models (DDPMs) [16] are a class of generative models used to produce novel samples by learning the distribution of the training data points. The main idea is to diffuse the data by progressively adding noise to the samples and then train the model to correctly predict the probability density function of the noise in order to remove it and reproduce the original data. The training process is split in two phases. In the forward diffusion process, noise sampled from a Gaussian distribution is iteratively diffused into the data until they are completely mapped to another, more simple distribution. Subsequently, a neural network models the reverse process that makes it capable of generating new samples, by starting from random noisy data and progressively mapping them to the distribution of the training data.

Formally, let $q(x)$ be the underlying distribution of the training data, and x_0 a sample drawn from it ($x_0 \sim q(x)$). The distribution of the forward diffusion process is determined in Equations (5) and (6). During the forward process, at each step t, a new latent variable x_t is produced by adding Gaussian noise of variance β_t to the previous variable x_{t-1}, as described in Equation (7), where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$.

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \mu_t = x_{t-1}\sqrt{1-\beta_t}, \Sigma_t = \beta_t \mathbf{I}) \quad (5)$$

$$q(x_{1:T}) = q(x_1|x_0)q(x_2|x_1)...q(x_T|x_{T-1}) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \quad (6)$$

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_t \quad (7)$$

By setting $a_t = 1 - \beta_t$, $\bar{a}_t = \prod_{i=1}^{t} a_i$ and using the reparameterization trick [17], x_t can be sampled directly, without calculating all intermediate latent variables. Given the starting data point x_0, the distribution q takes the form of Equation (8) and the latent variable after T timesteps can be calculated as shown in Equation (9).

$$q(x_t|x_0) = \mathcal{N}(x_t; \mu_t = x_0\sqrt{\bar{a}_t}, \Sigma_t = (1-\bar{a}_t)\mathbf{I}) \quad (8)$$

$$x_t = \sqrt{\bar{a}_t}x_0 + \sqrt{1-\bar{a}_t}\epsilon_0 \quad (9)$$

For the reverse process, a neural network is employed to model the distribution p (i.e., the parameters of another Gaussian) that progressively removes the noise and maps any random sample to the original distribution. The goal is to minimize the distance (Kullback–Leibler divergence) between p_θ and the forward posterior distribution $q(x_{t-1}|x_t, x_0)$, which is defined in Equation (10).

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t \mathbf{I}) \quad (10)$$

where $\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{a}_{t-1}}\beta_t}{1-\bar{a}_t}x_0 + \frac{\sqrt{a_t}(1-\bar{a}_{t-1})}{1-\bar{a}_t}x_t$ and $\tilde{\beta}_t = \frac{1-\bar{a}_{t-1}}{1-\bar{a}_t}\beta_t$. By setting the variance to a fixed constant β_t and using the reparameterizaton trick to express x_0 in terms of x_t, $\tilde{\mu}_t$ is defined as follows:

$$\tilde{\mu}_t(x_t) = \frac{1}{\sqrt{\bar{x}_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{a}_t}})\epsilon_t \quad (11)$$

As a result, the neural network only needs to predict the noise $\epsilon_{\theta(x_t,t)}$ at each timestep in order to minimize the following loss function:

$$L(\theta) = \mathbb{E}_{t,x_0,\epsilon}[\|\epsilon - \epsilon_\theta(x_t, t)\|^2] \quad (12)$$

Concerning the model's architecture, while the only restriction is that the input and output of the network are of the same dimensionality, the most common approach is a U-net-based neural network with residual and self-attention blocks. The model takes as input the timestep and the respective sample (image) and predicts the noise at that point. This way, after training is completed, any random noise sample (of the same dimensionality as the original data) can be fed to the model and be mapped step by step to the distribution of the training data, resulting in a completely new sample.

3. Related Work

Several approaches have been presented considering data augmentation for improving the performance of RL agents. In [12], a set of image augmentation techniques (such as translation, rotation, etc.) were applied to batches of original images before using them for the model's training. This technique was incorporated into the SAC [18] and PPO [19] algorithms, achieving state-of-the-art performance on the DeepMind Control Suite [20] and the OpenAI Gym benchmarks. A similar course has been followed in [21], showing that the aforementioned methodology can be effectively combined with asynchronous actor–critic algorithms as well [22]. Another relevant methodology is encountered in [14]. Here, the authors apply simple cropping combined with padding in order to randomly shift the original images in the first place and then average the Q function over a set of similar samples (the ones produced by transformations of the same original image) to reduce variance. The pipeline is integrated to the SAC algorithm and tested on the DeepMind and Atari benchmarks, outperforming state-of-the-art results in several cases. An improved, more time-efficient, version of the algorithm is presented in [13].

In the aforementioned approaches, the augmentation techniques are manually selected and extended experimentation is required in order to determine the optimal method per case. That is because the effect of each method depends on the features of the original image, meaning an image transformation may lead to different behavior in different environments. In order to overcome this issue, several attempts to automate the process of selecting the most suitable augmentation technique have been made. In [7], a search algorithm employing an RNN-based controller is used for automatic augmentation, improving the model's accuracy on image classification tasks. In the same context, the authors in [23] present a differentiable selection policy that significantly reduces the computational cost of the search and makes the algorithm more time efficient. In the field of RL, the concept of automating augmentation selection has been investigated mainly through meta-learning and upper confidence bounds (UCB) [24]. These strategies have been used as selection formulas for specifying the most appropriate approach on RL tasks, achieving notable performance [25,26].

In [27], the agent is trained on artificial trajectories imagined through a learned latent space of the world model, while an extension of the idea is proposed in [11]. The concept of mapping the representation to a latent space is also presented in [28]. In this case, the authors decouple the augmentation process from policy learning in order to benefit from data augmentation without introducing further complexity. A methodology for generating adversarial trajectories, which are combined with the original ones during training, to enhance generalization is described in [8]. In a similar manner, augmented and non-augmented data are used concurrently to jointly optimize the state–action value function based on a redefined objective [29]. Emphasizing the pixels' correlation with the objective, the authors in [30] propose a methodology for transforming only the less-task-relevant pixels, in order to prevent training instability, and in [31] they employ an auxiliary image reconstruction loss to improve sample efficiency.

Concerning the concept of predicting actions based on sequences of states (image frames), this has been studied under the more general scope of computer vision, as well as in reinforcement learning tasks, specifically through inverse dynamics models. In [32,33], the authors proposed a two-stream CNN-based network to predict the transformation between two images, as a methodology for feature learning. In a similar manner, a methodology

has been presented in [34] for predicting the action between states on an RL task, where the respective representations are images. The network is split into two different streams (each for one of the two input images), which are then concatenated and followed by dense layers in order to predict the final output. Each stream contains several convolutional filters which follow the AlexNet [35] architecture and the weights are shared along the two streams. The inverse dynamics model is then used and evaluated in conjunction with imitation learning for a specific robotics tasks, achieving promising performance.

The idea of using an action-predicting model was further developed in [36], where an inverse dynamics model was combined with a forward model, in order to produce an intrinsic reward (independent of the reward provided by the environment) for boosting exploration. Both the current and next states were firstly passed through an encoder for extracting important features in a latent space before predicting the action. A similar architecture is proposed in [37], where ResNet-based encoders are followed by LSTMs for predicting sequences of consecutive actions, conditioned directly on carrying out a specific task. In [38], a neural network is also employed for predicting actions, aiming to transfer knowledge from simulation environments to real-world problems, in cases where this is feasible in the physical properties of the real environment. The OpenAI's MuJoCo [39] environments were used as a testbed, where each state was represented by a feature vector. Thus, in this case, the model's input is a concatenation of several vectors of states and actions (in order to represent a sequence of states and actions) and the model's architecture consists of several fully connected layers before the output layer.

The implementation of forward models producing the next state directly, given the current state and an action, has also been examined. In this context, different architectures have been proposed, based on convolutional neural networks as well, since the input state representation is an image. A dynamics predictive model with deterministic and stochastic transition components is introduced in [40] for online planning in the latent space for model-based RL agents. A dynamics model is also trained in [41] to predict future states at first and then a backwards model is used to generate pairs of previous states–actions, forming a trajectory cycle. Inspired by the concept of curiosity, a forward model is designed in [42] aiming to assist the image encoder in capturing temporal information while providing intrinsic rewards to boost the agent's exploration. A reverse approach concerning the dynamics models and data augmentation is presented in [43]. In this case, data augmentation is used in the initial phase of the algorithm to generate views of the collected states, which are then used to train a consistent dynamics model in a latent space.

The authors in [44] designed an encoder–decoder-based architecture for predicting the next state in Atari Gym environments. Two different variants were tested for the intermediate transformation layer, a feedforward one and a recurrent one, producing realistic state frames. An extended variant of the previous architecture is proposed in [45] for predicting the immediate reward obtained. In this case, a second branch, consisting of a fully connected layer, is connected to the transformation layer output and takes as input the combined high-level information. The final model is able to jointly predict the next frame and the reward very efficiently. A different approach, consisting of several blocks (each containing convolutional layers) and a newly proposed block for pooling, is presented in [46]. Here, the authors propose transforming the action in a one-hot encoded form of stacked frames in the same dimensions as the state frame before feeding it to the network, rather than processing it in a feedforward layer. The dynamics model is then used to produce sequences of future states in a repetitive way.

4. Augmentation Framework

In the RL concept, the training samples are (s, a, r, s') tuples, consisting of a state s, an action a, an immediate reward r, and the resulting state s'. The process of generating synthetic data is split into two phases; initially, a DDPM is used to generate the state s of each sample, as well as the next state s', by learning the distribution of states already seen by the agent. Then, each newly generated pair (s, s') is fed to a model trained to predict

the action that led to a specific transition between the two states. In this way, triplets of the form (s, a, s') are produced by the two networks. An alternative would be to generate only the state s using the DDPM and then train the second network to generate the next state s', given the current state and an action. However, we opted for the former solution, as it is more efficient to produce both states using the DDPM and then predict the action in between (as a classification task) rather than generating a state and training a model to predict the next state (image) based on a specific action.

Concerning the reward, two different datasets have been created, based on states and actions that led to either positive or negative rewards. In most environments, the majority of actions do not provide immediate rewards, slowing down the training process. Under this realm, the synthetic samples can provide more diversity in cases where the immediate reward is non-zero and improve the model's generalization capability. For this purpose, two separate DDPMs have been trained (in the respective datasets) to produce states with high and low rewards. In this way, for each synthetic sample, the reward is known a priori depending on the model used to generate the states, and a complete sample of the form (s, a, r, s') is produced.

The proposed framework is tested on the Atari Learning Environment (ALE) [47]. The preprocessing steps described in [48] are executed before training the models (Section 5.1), similarly to most recent works in the literature. Thus, each state is represented by four grayscale consecutive frames, which are cropped to 84×84 pixels. Consecutive states are overlapped, as the first three frames of a state are essentially the last three frames of the previous one (e.g., if $s_1 = [f0, f1, f2, f3]$, then the next state would be $s_2 = [f1, f2, f3, f4]$). An example of real states obtained from the ALE boxing environment is depicted in Figure 1.

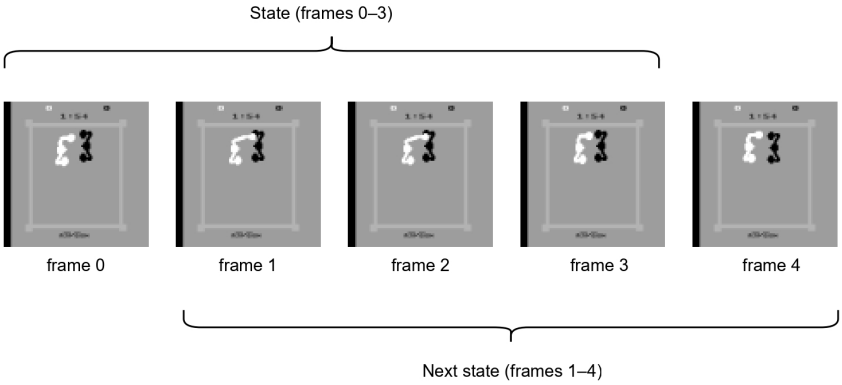


Figure 1. Preprocessed frames of the ALE boxing game.

4.1. Generative Model

For the generation of the states s and s' of each sample, a denoising diffusion probabilistic model is trained on states collected from the ALE environment. The model follows the original implementation presented in [16]. As mentioned above, each state in our case consists of four grayscale images. As the three last frames of the initial state overlap with the first three frames of the next state, once the initial state is created only the last frame of the next state is needed and the rest of them can be directly obtained from the previous state. In this respect, the input (and the output) of the network is an 84×84 image with five channels, which are then used to represent the two states, as shown in Figure 1. In order to adapt to the dimensions of the specific images, the network's architecture is slightly modified and consists of two encoder and two decoder blocks (not including the middle block), so that the image is downsampled two times (up to the shape of $21 \times 21 \times$ no_convolution_filters), in order to be feasible to reconstruct it to the original size.

As already discussed, two different DDPMs have been trained to produce samples with high and low rewards. In the ALE environment used in this work, all rewards are clipped to $\{-1, 0, 1\}$ after the preprocessing steps. Two datasets, each consisting of 2000 samples (with immediate rewards of 1 and -1) obtained from the agent's interaction

with the environment are created and used for training the models. Initially, all images are scaled to the $[-1, 1]$ range. The networks are trained for 140 epochs with the Adam optimizer and a learning rate of 2×10^{-4}.

4.2. Action-Predicting Model

For the action prediction, a CNN-based model with several residual connections is employed. The input of the model is an $84 \times 84 \times 5$ image representing two consecutive states of a game from the Atari 2600 framework, as described above. Initially, the image is passed through a convolutional layer, which is followed by a batch normalization layer and the ReLU activation function. Three residual blocks are employed, each consisting of two mini-blocks of a separable convolutional layer and a batch normalization layer. After the second batch normalization, max pooling is performed. The original input is also passed through a convolutional layer before being added to the residual block output. Another separable convolutional layer followed by batch normalization, ReLU, global average pooling, and dropout is used after the residual blocks. The final output layer employs softmax activation, depending on the specified action space of the game. A detailed description of the layers and hyperparameters of the network is provided in Appendix A (Table A1).

The model is compiled with the Adam optimizer using a learning rate of 10^{-5} and categorical cross-entropy loss. Concerning the training process, the model was trained on a dataset consisting of 100,000 samples of the form ([state, next_state], action) for each game, which were obtained from a random agent's interaction with the corresponding environment.

4.3. Data Generation

The process of generating complete synthetic samples of the form (s, a, r, s') is depicted in Figure 2. Two DDPMs are trained on samples of two consecutive game states in order to produce a state s and the next state s'. The goal is to generate samples with either very high or very low rewards since the original dataset created by the agent's observations is very imbalanced in the sense that the immediate reward in most cases is zero. The training dataset for the first model consists of pairs (s, s') with a high reward, while the second one contains pairs with a low reward. As each model is trained on a specific type of data, we can assume that the states generated by the first model correspond to high rewards while the states generated by the second model correspond to low rewards. In this way, the reward for each synthetic sample is considered as known by the time of its generation.

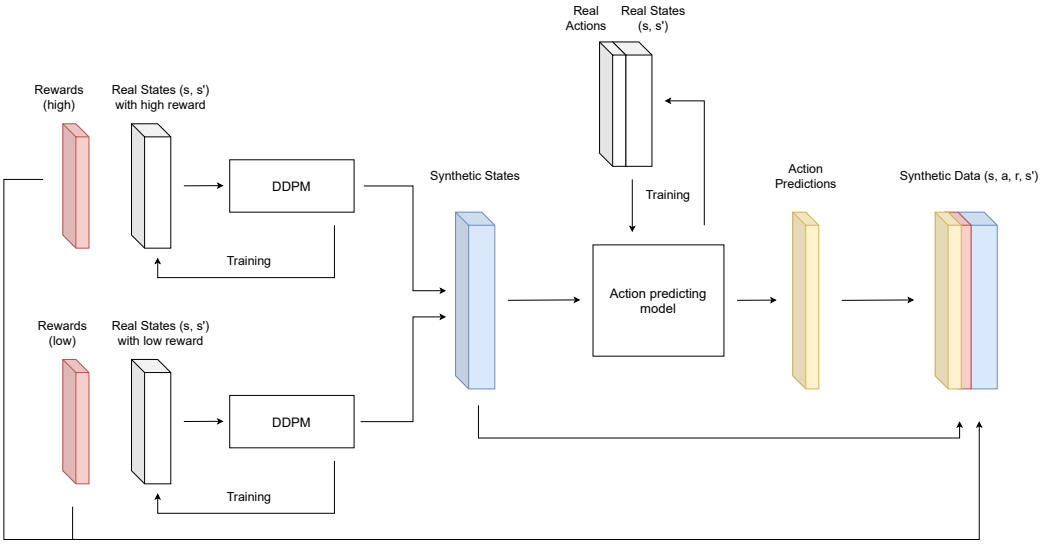


Figure 2. Synthetic data generation.

Since the executed action is not taken into consideration during DDPM training, each generated pair (s, s') could be the result of a different action. In order to produce functional samples that can be used to train the agent, a separate model has been employed for action prediction given two consecutive states. Thus, for each pair (s, s') of synthetic states, the action is predicted by this model and the reward is immediately set to either 1 or -1 depending on the DDPM used for the generation, resulting in a complete training sample. The described process is presented step by step in Algorithm 1.

Algorithm 1: Synthetic data generation

```
Function GenerateSyntheticData():
    action_predictor_buffer, ddpm_buffer_high, ddpm_buffer_low ← [], [], []
    while action_predictor_buffer.size() < max_size do
        a ← agent.select_action(s)
        s', r ← env.execute(a)
        action_predictor_buffer.append(s, a, r, s')
        if r > 0 then
            ddpm_buffer_high.append(s, a, r, s')
        if r < 0 then
            ddpm_buffer_low.append(s, a, r, s')
        s ← s'
    action_predictor.train()
    ddpm_high.train()
    ddpm_low.train()
    {s_h, s'_h} ← ddpm_high.generate_states()
    {s_l, s'_l} ← ddpm_low.generate_states()
    {a_pred} ← action_predictor.predict({s_h, s'_h}, {s_l, s'_l})
    synthetic_data.append({s_h, a_pred, r, s'_h})
    synthetic_data.append({s_l, a_pred, r, s'_l})
    return synthetic_data
```

Figure 3 shows samples generated with the methodology described above for the ALE boxing environment. Specifically, frames h_0– h_4 are generated with the diffusion model trained on states that lead to high rewards, while frames l_0–l_4 are produced by the second DDPM and represent two consecutive states with a low immediate reward. The synthetic images are then fed to the action-predicting model in order to be paired with the most suitable action.

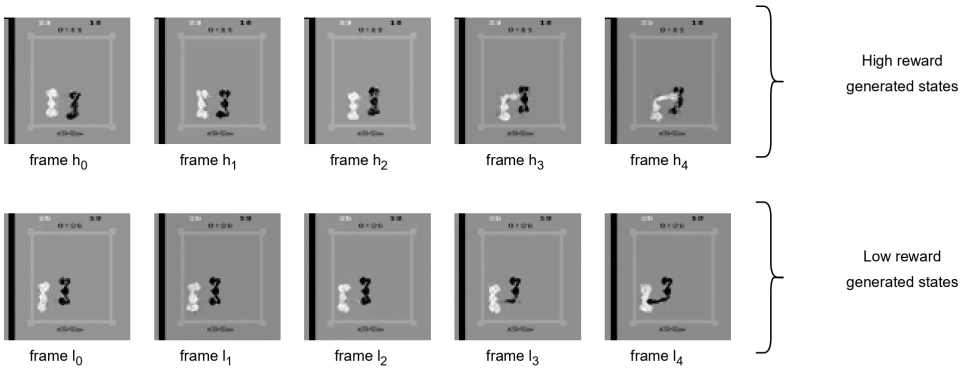


Figure 3. Synthetically generated samples for the ALE boxing game.

It is evident that the generated images are quite similar to the real ones in terms of quality. The generative models successfully produce consecutive frames while maintaining the more general features of the image. As depicted, the two models are able to generate

meaningful states in the sense that the transition between two consecutive frames is smooth, but also capture the most crucial patterns that lead to either positive or negative rewards. In this manner, the produced states are functional and there is no risk of misguiding the training procedure by augmenting the dataset with irrelevant images.

4.4. RL Training with Synthetic Data

In order to improve the agent's training and speed up the learning process, the synthetic data are mixed with real samples at the batch level. Specifically, the generated data are stored in a separate buffer and then samples are drawn from both the real and the synthetic buffer to ensure that every batch contains samples from both categories. In this way, there is full control over the training process, as the exact percentage of each category in every batch can be determined explicitly, as opposed to randomly selecting samples from a shared buffer. Therefore, it is possible to further optimize training by selecting the most suitable ratio, as discussed in Section 5.

Another important part of the proposed framework is the use of the action-predicting model. It is clear that the model's accuracy can greatly affect the overall performance of the agent, since predicting incorrect actions on sequences of states will misguide the agent's network. As the task of predicting the action executed between two states can be quite complex and depends on the training environment and the action space, synthetic samples may be used, based on the confidence of the predicted actions. For this purpose, a threshold is set for the probability produced by the model on the predicted action. Samples containing predicted actions with a probability higher than the aforementioned confidence threshold are stored in the synthetic data buffer, while the rest are discarded. In this way, the number of incorrect generated samples is reduced and the quality of the training data is improved.

The training process of the introduced agent, Deceiver, is depicted in Figure 4 and explained in Algorithm 2. Note that both the confidence threshold and the weight (ratio) of synthetic samples in the batch (synthetic weight (*sw*)) are treated as hyperparameters of the system (lines 1, 6). As shown, the data generation process is independent of the training algorithm, meaning it can be applied to different agent network architectures. In the current work, the proposed methodology is embedded into the DQN algorithm, combining synthetic samples with real observations obtained from the environments of the games.

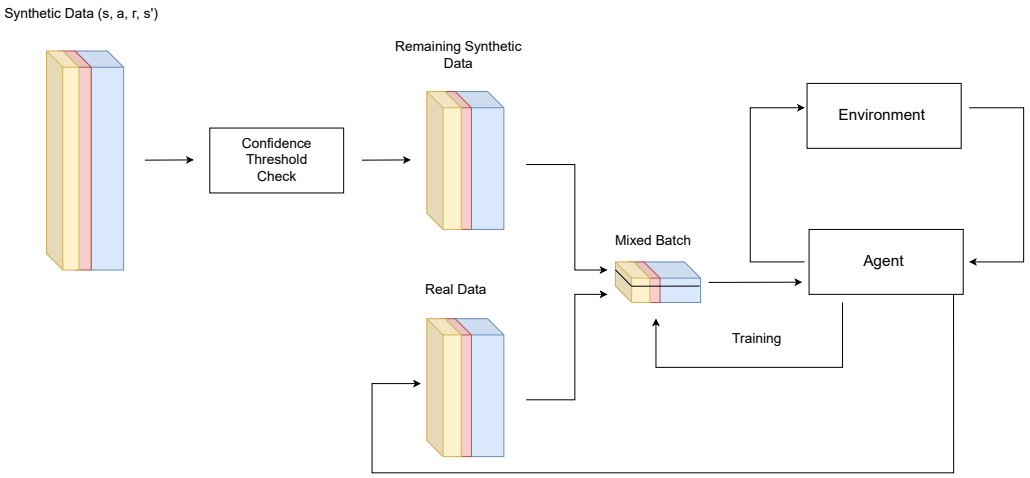


Figure 4. Training on synthetic and real samples.

Algorithm 2: Training on augmented data

```
Function PreprocessData(confidence_thres):
    for (s_i, a_i, r_i, s'_i) in synthetic_buffer do
        if prob(a_i) < confidence_thres then
            synthetic_buffer.remove(s_i, a_i, r_i, s'_i)
    return synthetic_buffer
Function TrainAgent(sw):
    real_samples = real_buffer.sample_batch(batch_size × (1 − sw))
    synthetic_samples = synthetic_buffer.sample_batch(batch_size × sw)
    batch = concatenate([real_samples, synthetic_samples]).shuffle()
    agent.train_on_batch()
Function ActAndTrain():
    PreprocessData()
    while timesteps < termination_thres do
        a ← agent.select_action(s)
        s', r ← env.execute(a)
        real_buffer.append(s, a, r, s')
        if timesteps%train_freq = 0 then
            TrainAgent()
```

5. Experiments

5.1. Setup and Hyperparameter Tuning

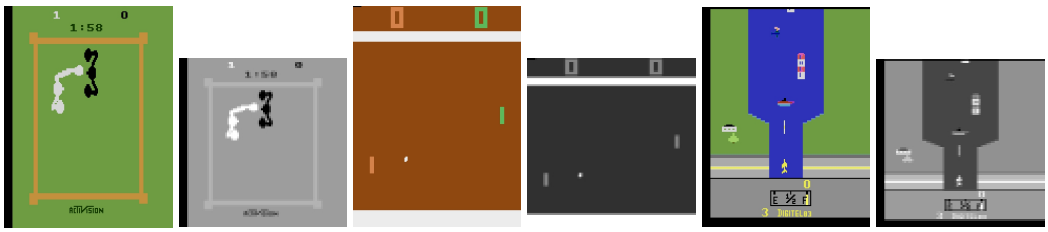
The proposed algorithm is implemented in the Python programming language. Specifically, the TensorFlow [49] (v2.11) library is employed for the development and training of the models and the Gym [50] environment is used for the RL interaction process. The data augmentation and the agent's training are treated as independent tasks and are then combined to form the complete agent. The experiments are carried out on three heterogeneous games of the Gym's Atari 2600 environment:

- **Boxing:** The agent controls a boxer in a fighting ring and is rewarded for hitting its opponent while a negative reward is obtained for getting hit. There are 18 possible actions in this game.
- **Pong:** The classic pong game, where the agent controls a paddle and tries to deflect the ball away from its own goal and send it into its opponents goal. Points are scored accordingly every time the ball ends up in one of the goals. In this game, the action space size is six.
- **Riverraid:** the agent controls a jet over a river, aiming to avoid or destroy enemy objects. The action space size is 18. Only positive rewards are available in this game.

The states fed to the augmentation and agent networks are subjected to several preprocessing steps. Initially, every image is downsampled to 84×84 pixels and converted to grayscale. Frame skipping is also performed in a way that only every fourth frame is taken into account, as the difference between two actually consecutive frames is very slight. Afterwards, four consecutive frames (that remained after skipping) are concatenated to form a state representation, so that the required information concerning the motion of objects is provided. Finally, all rewards are clipped to $\{-1, 0, 1\}$. An example state of each game before and after applying the preprocessing steps is depicted in Figure 5.

Several values are tested for the confidence threshold of the predicted actions, as well as the weight (percentage) of each data category on the mixed batches. Regarding the agent network, it consists of three convolutional layers, followed by a dense layer before the output layer. The replay buffer's size is 100,000 samples, while the synthetic buffer

contains 20,000 generated samples. Complete implementation details of the agent model are provided in Table A2.



(**a**) (**b**) (**c**) (**d**) (**e**) (**f**)

Figure 5. Game states before and after preprocessing. (**a**) Boxing original. (**b**) Boxing preprocessed. (**c**) Pong original. (**d**) Pong preprocessed. (**e**) Riverraid original. (**f**) Riverraid preprocessed.

Figure 6 depicts the accuracy of the action-predicting model depending on the confidence threshold for each game. As the selecting condition becomes more strict, the number of samples that can be used for the agent's training is reduced. This explains, to a certain degree, the decrease in accuracy after a certain threshold (which is expected to increase along with the confidence value), since in this case it is based on a small number of samples. In this respect, there is a tradeoff between the accuracy of the model and the amount of samples to be rejected, as a very small number of synthetic samples may cause overfitting and negatively affect the agent's training. Therefore, for each game several confidence thresholds have been tested for the respective action-predicting model in order to determine the best configuration.

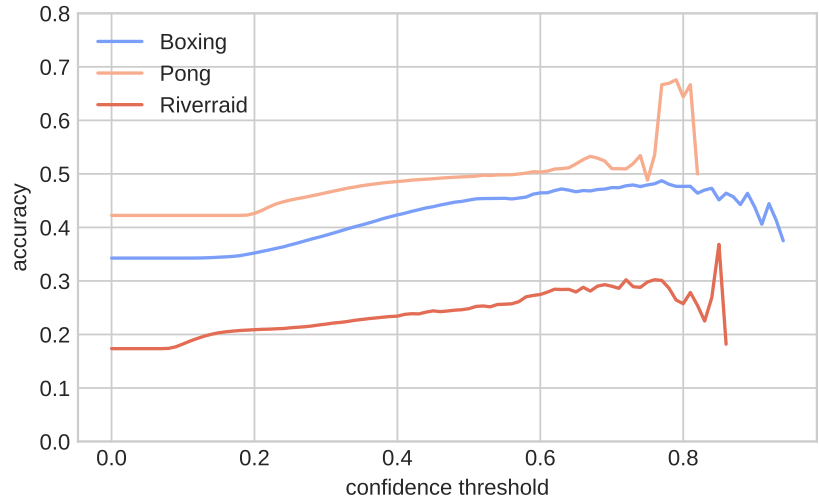
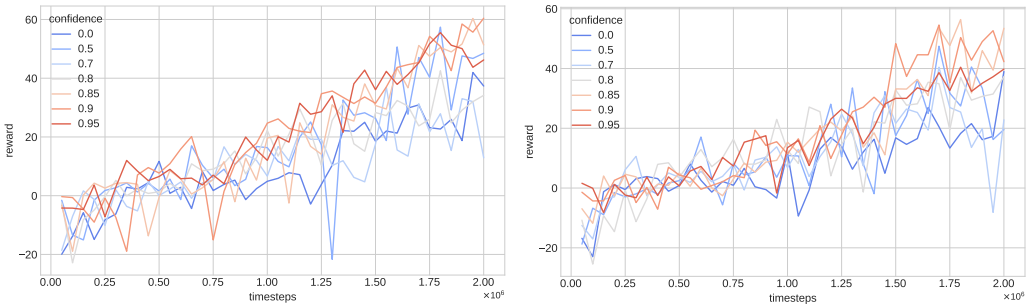


Figure 6. Action-predicting model's accuracy for different confidence thresholds.

Figure 7 shows the average reward achieved by different configurations of the proposed agent on the Boxing game. Specifically, Figure 7a summarizes the performance of the agent for several confidence thresholds and synthetic data weight equal to 0.1, while Figure 7b outlines the respective results for synthetic data weight value 0.2. The reward is calculated every 50,000 timesteps and is averaged over 50 runs.

As expected, the confidence value greatly affects the agent's learning. Higher thresholds lead to a higher accuracy of the prediction model and a lower percentage of incorrect synthetic samples. In this respect, the performance increase along with the increase in the confidence, as highlighted in Figure 7a, seems quite natural. Nevertheless, as discussed above, very high thresholds exclude a large number of samples from the synthetic buffer that could result in the network being trained on the same samples with high frequency. This phenomenon is depicted in Figure 7b, where the synthetic weight is larger, meaning

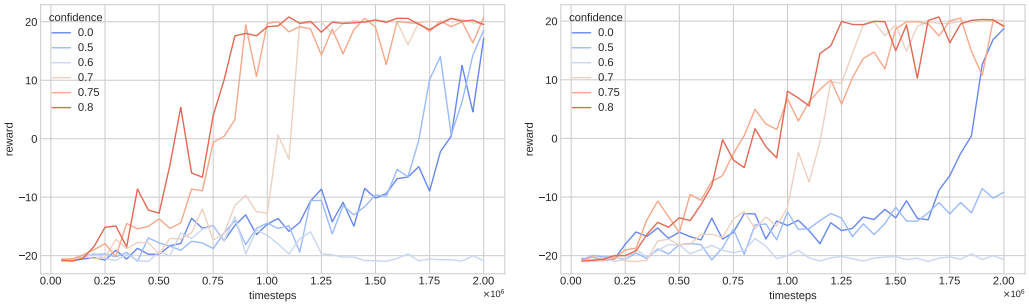
that more synthetic samples are introduced in every batch. In this case, a very high confidence value (specifically the 0.95 configuration) leads to slower learning, while lower values (0.9 and 0.85) result in the agent's top performance in terms of total reward and timesteps needed to achieve it. Thus, the importance of a balance between the action predictor's accuracy and the size of the synthetic buffer is highlighted.



(**a**) (**b**)

Figure 7. Deceiver average reward (Boxing). (**a**) Synthetic data weight of 0.1. (**b**) Synthetic data weight of 0.2.

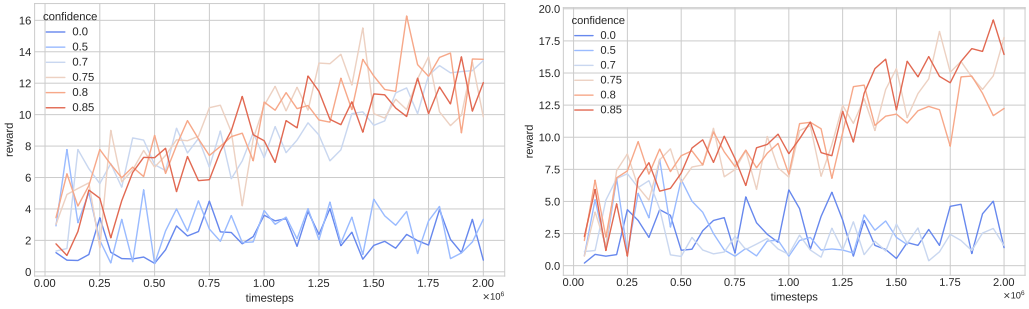
In the case of Pong (Figure 8), the effect of the confidence parameter is more clear. For very low values (up to 0.6), the agent learns very slowly or, in some cases, the agent is not even able to learn, due to the low quality of the generated samples, as a high percentage of the synthetic samples contain incorrect actions (in the sense that the actions determined by the action predictor cannot actually produce the transitions between the states generated by the DDPMs). Concerning the action-predicting model, even though it achieves higher accuracy in this game overall (probably because of the smaller action space size), after a certain confidence value, it cuts off the whole synthetic dataset (e.g., there is no synthetic sample for which the model's probability on the selected action is higher than 0.9, thus setting the confidence threshold to 0.9 would result in an empty synthetic buffer). Therefore, the highest confidence value taken into account for the Pong experiments is 0.8; for higher values there are no remaining synthetic samples (as shown in Figure 6).



(**a**) (**b**)

Figure 8. Deceiver average reward (Pong). (**a**) Synthetic data weight of 0.1. (**b**) Synthetic data weight of 0.2.

Similar conclusions can be drawn from the performance of the agent on the third game, Riverraid (Figure 9). In this case as well, it is not capable of reaching high scores in the setups with low confidence values. The highest threshold allowed by the model's accuracy for this game is 0.85. Regarding the synthetic data weight, the agents with the higher value (0.2) seem to benefit from the larger percentage of synthetic samples per batch and perform better than the others, in contrast with the previous environments, where the most effective agents were the ones in the low synthetic weight (0.1) setup.



(**a**) (**b**)

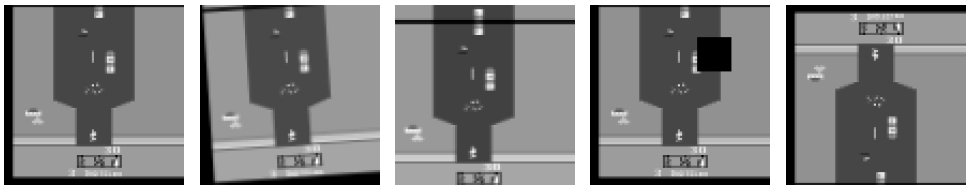
Figure 9. Deceiver average reward (Riverraid). (**a**) Synthetic data weight of 0.1. (**b**) Synthetic data weight of 0.2.

5.2. Results

For the evaluation of the proposed algorithm, the best configuration of Deceiver for each game (as determined in Section 5.1) is presented, along with several agents boosted with traditional augmentation techniques. The following techniques are implemented and tested on the three Atari environments showcased above:

- **Rotation:** Applies random rotation to the image either clockwise or anti-clockwise. In this work each state image is rotated randomly in the $[-7°, 7°]$ degree range.
- **Translation:** The image is randomly shifted horizontally and vertically by w and h pixels, respectively. In our case both values are in the range $[-0.1\times$ *img_height*, $0.1\times$ img_height]. Pixels outside the image area are filled by mirroring the pixels on the edge.
- **Cutout:** A small box area of the image is randomly selected and all its pixels are set to 0. The width and height of the box are in the $[10, 20]$ pixel range.
- **Flip:** The image is randomly flipped vertically, horizontally, or over both axes.

An example of a game state before and after applying each augmentation technique on the Riveraid game is presented in Figure 10.



(**a**) (**b**) (**c**) (**d**) (**e**)

Figure 10. Game states before and after augmentation. (**a**) No augmentation. (**b**) Rotation. (**c**) Translation. (**d**) Cutout. (**e**) Flip.

Figures 11–13 depict the performance of the agents on the Boxing, Pong, and Riverraid environments, respectively. The tested agents are the original DQN agent without augmentation, the four agents enhanced with random rotation, translation, cutout, and flip, as described above, and the proposed agent, Deceiver (the best hyperparameters' values are set as determined in Figures 7–9). As shown, the presented agent outperforms the rest of the agents or achieves comparable results in all cases. Specifically, in the Boxing game (Figure 11) Deceiver achieves a significantly higher reward than the original algorithm, as well as all four augmentation variants. Especially in the second half of the training process (after $1M$ timesteps), the agent seems to benefit in a high degree from the combination of synthetic and real samples and demonstrates a faster learning curve.

Pong is a simpler game, which is confirmed by the fact that three agents achieve the highest possible reward (the rewards shown in Figure 12 are averaged over 50 runs, meaning an amount of variance is introduced, but after a certain point the scores are

consistently very close to 21, which is the maximum achievable score for this game). Under this scope, the performance comparison can be in terms of timesteps needed to reach this score. In this respect, Deceiver first achieves a reward close to the maximum about $100k$ timesteps earlier and seems to have a faster learning curve, in general, during training (up to the point that all agents finally reach the high score).

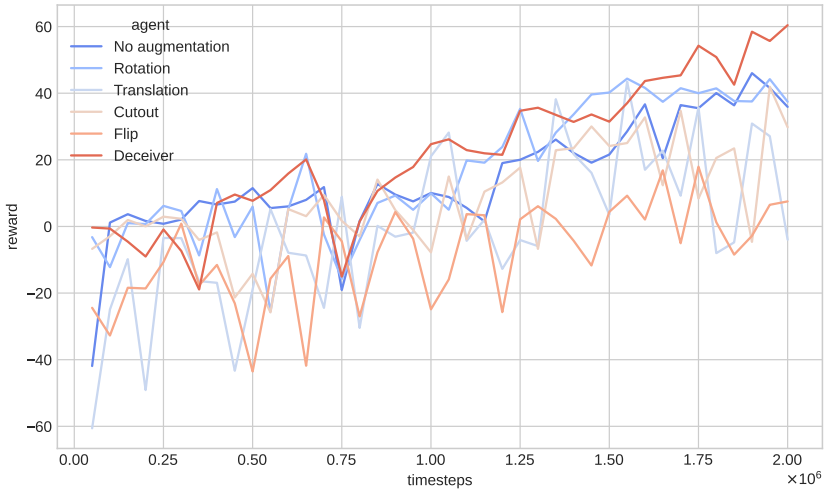


Figure 11. Augmentation methods' average reward (Boxing).

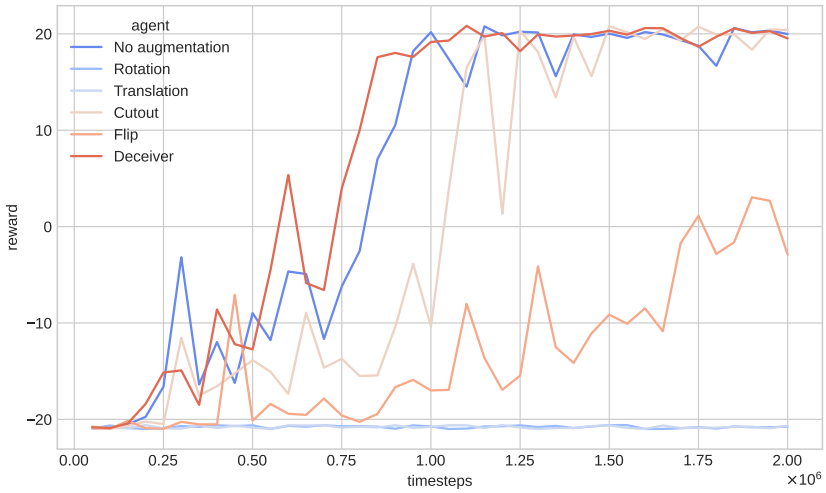


Figure 12. Augmentation methods' average reward (Pong).

In the third game (Figure 13), the proposed algorithm does not seem to have a clear advantage over the other techniques; however, it exhibits similar performance to the best tested variants. This is most favorably assigned to the accuracy of the action predictor which is lower in the case of Riverraid. As highlighted in Figures 7–9 the confidence value and consequently the action predictor's accuracy has a great impact on the learning process. Additionally, a peculiarity of this game that probably led to this behavior is that there are no negative rewards. As a result, in this case only one diffusion model was used for generation of synthetic states with high reward, which could explain the lack of a significant increase in Deceiver's performance on this environment.

Table 1 presents the overall results (max reward, timesteps, action predictor's accuracy) of the implemented agents per game. The best variant of Deceiver for each synthetic weight setup (0.1 and 0.2) is exhibited. As already discussed, the best confidence value is not the same for the different synthetic weight in all cases, which explains the difference in the accuracies of the action predictors in the first and third environments.

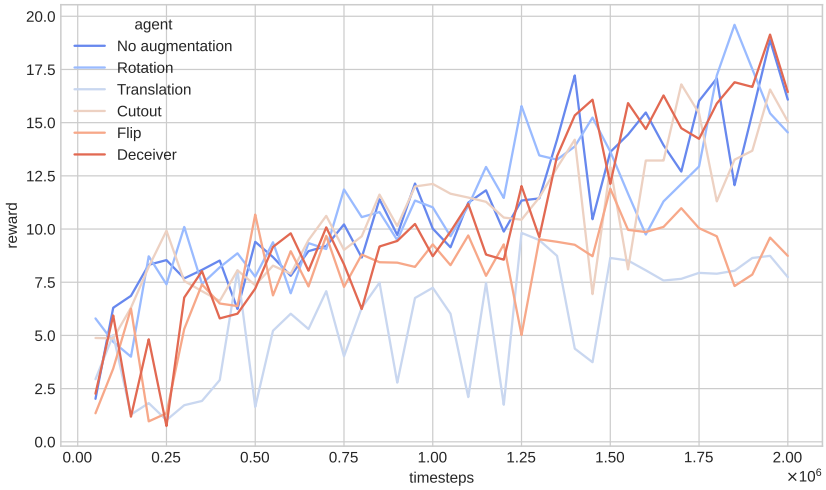


Figure 13. Augmentation methods' average reward (Riverraid).

Table 1. Different augmentation-based agents' performance (the highest reward per game is indicated in bold).

	Max Reward	Timesteps	Predictor's Accuracy	Game
No augmentation	46.02	1.9 M	-	**Boxing**
Rotation	44.38	1.55 M	-	
Translation	43.46	1.55 M	-	
Cutout	41.92	1.95 M	-	
Flip	17.94	1.75 M	-	
Deceiver (0.1)	**60.42**	2 M	0.438	
Deceiver (0.2)	56.38	1.8 M	0.452	
No augmentation	20.78	1.15 M	-	**Pong**
Rotation	−20.60	1.5 M	-	
Translation	−20.60	1.2 M	-	
Cutout	20.80	1.5 M	-	
Flip	3.04	1.9 M	-	
Deceiver (0.1)	**20.83**	1.1 M	0.644	
Deceiver (0.2)	20.72	1.7 M	0.644	
No augmentation	18.88	1.95 M	-	**Riverraid**
Rotation	**19.60**	1.85 M	-	
Translation	9.82	1.25 M	-	
Cutout	16.80	1.7 M	-	
Flip	11.90	1.5 M	-	
Deceiver (0.1)	16.28	1.65 M	0.257	
Deceiver (0.2)	19.14	1.95 M	0.368	

6. Discussion

The key observations of the aforementioned process are reviewed in this section. At first, the experimental results confirm the instability of traditional image augmentation methods in the RL domain. Specifically, random rotation, translation, cutout, and vertical and horizontal flips have been implemented to augment states in the tested environment, similarly to [7,12,21]. Random convolutions, color jitter, brightness contrast, and cutout-color have also been examined in the literature but are not applicable to the current environment, since the input images are grayscale (Section 5.1). As outlined in [25,26], different augmentation types may have a different effect, depending on the environment, and thus, lack generalization. In the examined Atari games, it is evident that their effect on the agent's learning depends highly on the characteristics of the environment. Rotation and cutout are in general the most beneficial ones; however, none of them outperforms the rest in all cases. For example, in the Pong game (Figure 12), the random rotation and

translation methods have a negative impact on the performance of the agent, as it is unable to achieve positive scores, which is quite reasonable since the paddles (and occasionally the ball), which are the most informative parts of the state, are on the edges of the image, meaning that after applying translations or rotations they could be cropped out of the image. On the contrary, in the Riverraid game (Figure 13), where the most informative part lies in the center of the image, rotation seems to provide more diversity and improve the agent's performance.

Concerning the proposed augmentation methodology, it is among the first to consider generative augmentation directly on the original state space, to the best of our knowledge. Recent works have emphasized projecting the states to a latent space and then processing and augmenting the latter representations [11,28,43]. The presented algorithm is completely independent of embedding transformation models. Instead, a generative model is trained on raw observations collected by the agent and produces states which can be used directly for training. In this way, it overcomes the aforementioned generalization issues of traditional augmentation techniques without the need to encode the initial information. The boost in the learning process is not the same across all tested environments in our case as well; however, the agent demonstrates a quite consistent performance. In the worst case, it is comparable to the no-augmentation algorithm (Table 1), indicating that it is not highly affected by the attributes of the images. This behavior can be ascribed to the generational process followed, which leads to completely novel samples, instead of transforming real observations (which may result in pivotal information loss).

A major aspect of our framework is the model predicting the action between consecutive states. Most approaches in the literature employ forward dynamics models, which are able to predict the next state given the current state and a specific action [40–42]. Although the reported results are very promising, our attempts to implement a forward model on the tested games showed that it is a very demanding task as, especially in cases of large action space sizes, it is difficult to predict the exact state changes caused by a specific action. Instead, an inverse dynamics model is used that, given two states, predicts the executed action, as in [34,36,37]. The inverse problem proved to be significantly more simple, yet suitable for the synthetic data generation, since both current and next states are produced by DDPMs.

Regarding the parameters used, the confidence threshold of the action-predicting model and the ratio of real and synthetic samples in every batch proved to be the most important ones. Notably, an increase in the confidence threshold (and consequently the accuracy of the model) led to a significant improvement in the agent's training, even though the accuracy of the model is lower than 50% in two out of the three examined cases. This result highlights the benefits of the proposed augmentation as, despite the fact that many samples do not contain the correct action, the overall effect in performance is favorable. This probably indicates that the actions predicted in such cases are similar to the ones that would actually lead to the specified state transition, and thus, training is not misguided. Nevertheless, the contribution of the action prediction network to the agent is crucial and its improvement could be a key factor for the proposed framework. Concerning the samples ratio per batch, the experiments summed up that a percentage of 10–20% of synthetic samples is optimal for adding diversity and speeding up training, whereas a further increase starts to affect it in a negative way.

7. Conclusions

In this work, a framework for data augmentation in the context of reinforcement learning is proposed. Particularly, two denoising diffusion probabilistic models and a custom CNN-based inverse dynamics model have been employed for synthetic sample generation: the first one is used to learn the distribution of real states and produce new realistic states, while the latter is responsible for predicting the action taken between two consecutive states. Depending on the dataset used for training, each DDPM produces pairs of current and next states of either high or low reward. As a result, complete samples

consisting of a state, an action, the obtained reward, and the next state are generated. In this way, the samples are more diverse, as both the initial and the resulting states are generated from scratch, rather than being variants of existing real samples as in traditional augmentation techniques. The proposed augmentation method is integrated into the DQN algorithm in order to enhance the agent with synthetic samples and boost its performance. The augmentation models are trained and used to generate fake samples, which are stored in a separate buffer. Then, during each training step, samples from both the real and the fake buffer are selected to form batches in order to train the agent.

The enhanced agent, Deceiver, is implemented and tested on the ALE environment, outperforming traditional augmentation-boosted agents in terms of accumulated reward as well as timesteps of interaction with the environment, in most cases. Most of these techniques seem to be highly affected by the environment they are applied to and the properties of the state space. The proposed methodology, even though it does not boost the agent in all cases, is more stable and performs at least comparably to the no-augmentation algorithm on the tested games. Specifically, the maximum reward was reached by Deceiver $200k$ timesteps earlier and was increased up to 31.2% in the best case. The carried out experiments also highlighted the effect of the action-predicting model on the learning curve of the agent and consequently its overall performance. The quality of the synthetic samples depends on the action prediction and greatly affects the agent's behavior. In this respect, a promising direction for further improving the current technique would be to experiment with different architectures of the inverse dynamics model in order to achieve higher accuracy and indirectly boost the agent.

On top of the aforementioned idea, we aim to integrate the augmentation framework into different types of RL algorithms and test the performance of the enhanced agents in terms of convergence speed, as well as the exploration of novel states, in order to confirm their applicability to different setups. Additionally, the augmentation models could be trained in parallel with the agent, in order to investigate the interaction between them while interacting with the environment and a possible improvement on either the augmentation models or the agent, directly. Expanding upon this concept, as the agent collects new samples, a periodical retraining may also improve the precision and diversity of the augmentation networks.

Author Contributions: Conceptualization, T.P. and G.A.; methodology, T.P. and A.S.; software, T.P.; validation, T.P. and G.A.; formal analysis, T.P.; investigation, T.P.; resources, T.P. and G.A.; data curation, T.P. and G.A.; writing—original draft preparation, T.P.; writing—review and editing, G.A. and A.S.; visualization, T.P.; supervision, A.S.; project administration, A.S.; funding acquisition, A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data for this study were generated using the publicly available environment that can be found here: https://github.com/openai/gym (accessed on 29 November 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RL	Reinforcement learning
DQN	Deep Q-learning
MDP	Markov decision process
DDPM	Denoising diffusion probabilistic model
UCB	Upper confidence bound
RNN	Recurrent neural network
CNN	Convolutional neural network

LSTM	Long short-term memory
ALE	Atari Learning Environment
ReLU	Rectified linear unit

Appendix A

Appendix A.1

The hyperparameters and the overall architecture schema of the model used for predicting the actions between synthetic states generated by the DDPMs is presented in Table A1 below.

Table A1. Action predictor model architecture.

	Layer	Output Size	Input Layer	Kernel/Pool Size	Activation
	#1 Input	$84 \times 84 \times 5$	-	-	-
	#2 Conv2D	$42 \times 42 \times 128$	#1	3×3	-
	#3 BatchNorm	$42 \times 42 \times 128$	#2	-	ReLU
ResBlock 1	#4 SepConv2D	$42 \times 42 \times 256$	#3	3×3	-
	#5 BatchNorm	$42 \times 42 \times 256$	#4	-	ReLU
	#6 SepConv2D	$42 \times 42 \times 256$	#5	3×3	-
	#7 BatchNorm	$42 \times 42 \times 256$	#6	-	-
	#8 MaxPooling2D	$21 \times 21 \times 256$	#7	3×3	-
	#9 Conv2D	$21 \times 21 \times 256$	#3	1×1	-
	#10 Add	$21 \times 21 \times 256$	#8, #9	1×1	ReLU
ResBlock 2	#11 SepConv2D	$21 \times 21 \times 512$	#10	3×3	-
	#12 BatchNorm	$21 \times 21 \times 512$	#11	-	ReLU
	#13 SepConv2D	$21 \times 21 \times 512$	#12	3×3	-
	#14 BatchNorm	$21 \times 21 \times 512$	#13	-	-
	#15 MaxPooling2D	$11 \times 11 \times 512$	#14	3×3	-
	#16 Conv2D	$11 \times 11 \times 512$	#10	1×1	-
	#17 Add	$11 \times 11 \times 512$	#10, #16	1×1	ReLU
ResBlock 3	#18 SepConv2D	$11 \times 11 \times 728$	#17	3×3	-
	#19 BatchNorm	$11 \times 11 \times 728$	#18	-	ReLU
	#20 SepConv2D	$11 \times 11 \times 728$	#19	3×3	-
	#21 BatchNorm	$11 \times 11 \times 728$	#20	-	-
	#22 MaxPooling2D	$6 \times 6 \times 728$	#21	3×3	-
	#23 Conv2D	$6 \times 6 \times 728$	#22	1×1	-
	#24 Add	$6 \times 6 \times 728$	#17, #22	1×1	ReLU
	#25 SepConv2D	$6 \times 6 \times 1024$	#24	3×3	-
	#26 BatchNorm	$6 \times 6 \times 1024$	#25	-	ReLU
	#27 GlobalAvPooling2D	1024	#26	-	-
	#28 Dropout (0.5)	1024	#27	-	-
	#29 Dense	actions	#28	-	Softmax

Appendix A.2

The architecture of the main agent's network used in the experiments is described in Table A2. The hyperparameters employed concerning the training process of the model are presented in Table A3.

Table A2. Agent model architecture.

Layer	Output Size	Input Layer	Kernel/Pool Size	Activation
#1 Input	$84 \times 84 \times 4$	-	-	-
#2 Conv2D	$20 \times 20 \times 32$	#1	8×8	ReLU
#3 Conv2D	$9 \times 9 \times 64$	#2	4×4	ReLU
#4 Conv2D	$7 \times 7 \times 64$	#3	3×3	ReLU
#5 Dense	512	#4	-	ReLU
#6 Dense	actions	#5	-	Softmax

Table A3. Agent model hyperparameters.

Hyperparameter	Value
Learning rate	1×10^{-4}
Optimizer	Adam
Batch size	32
Discount factor (γ)	0.99
Real buffer size	100,000
Synthetic buffer size	20,000
$epsilon_{max}$	1.0
$epsilon_{min}$	0.1
Initial random timesteps	30,000
Evaluation episodes	50
Train frequency (timesteps)	4
Update target network frequency (timesteps)	5000

References

1. Shorten, C.; Khoshgoftaar, T.M. A survey on image data augmentation for deep learning. *J. Big Data* **2019**, *6*, 60. [CrossRef]
2. Taylor, L.; Nitschke, G. Improving Deep Learning with Generic Data Augmentation. In Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Bangalore, India, 18–21 November 2018; pp. 1542–1547. [CrossRef]
3. Kim, E.K.; Lee, H.; Kim, J.Y.; Kim, S. Data augmentation method by applying color perturbation of inverse PSNR and geometric transformations for object recognition based on deep learning. *Appl. Sci.* **2020**, *10*, 3755. [CrossRef]
4. de la Rosa, F.L.; Gómez-Sirvent, J.L.; Sánchez-Reolid, R.; Morales, R.; Fernández-Caballero, A. Geometric transformation-based data augmentation on defect classification of segmented images of semiconductor materials using a ResNet50 convolutional neural network. *Expert Syst. Appl.* **2022**, *206*, 117731. [CrossRef]
5. Bowles, C.; Chen, L.; Guerrero, R.; Bentley, P.; Gunn, R.; Hammers, A.; Dickie, D.A.; Hernández, M.V.; Wardlaw, J.; Rueckert, D. Gan augmentation: Augmenting training data using generative adversarial networks. *arXiv* **2018**, arXiv:1810.10863.
6. Frid-Adar, M.; Diamant, I.; Klang, E.; Amitai, M.; Goldberger, J.; Greenspan, H. GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification. *Neurocomputing* **2018**, *321*, 321–331. [CrossRef]
7. Cubuk, E.D.; Zoph, B.; Mane, D.; Vasudevan, V.; Le, Q.V. Autoaugment: Learning augmentation strategies from data. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; pp. 113–123.
8. Zhang, H.; Guo, Y. Generalization of reinforcement learning with policy-aware adversarial data augmentation. *arXiv* **2021**, arXiv:2106.15587.
9. Yuan, Z.; Xue, Z.; Yuan, B.; Wang, X.; Wu, Y.; Gao, Y.; Xu, H. Pre-trained image encoder for generalizable visual reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 13022–13037.
10. Ma, G.; Wang, Z.; Yuan, Z.; Wang, X.; Yuan, B.; Tao, D. A Comprehensive Survey of Data Augmentation in Visual Reinforcement Learning. *arXiv* **2022**, arXiv:2210.04561.
11. Hafner, D.; Lillicrap, T.; Norouzi, M.; Ba, J. Mastering Atari with Discrete World Models. *arXiv* **2022**, arXiv:2010.02193.
12. Laskin, M.; Lee, K.; Stooke, A.; Pinto, L.; Abbeel, P.; Srinivas, A. Reinforcement learning with augmented data. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 19884–19895.
13. Yarats, D.; Fergus, R.; Lazaric, A.; Pinto, L. Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv* **2021**, arXiv:2107.09645.
14. Kostrikov, I.; Yarats, D.; Fergus, R. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv* **2020**, arXiv:2004.13649.
15. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
16. Ho, J.; Jain, A.; Abbeel, P. Denoising diffusion probabilistic models. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 6840–6851.
17. Kingma, D.P.; Welling, M. Auto-encoding variational bayes. *arXiv* **2013**, arXiv:1312.6114.
18. Haarnoja, T.; Zhou, A.; Abbeel, P.; Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Proceedings of the International Conference on Machine Learning, Stockholm Sweden, 10–15 July 2018; pp. 1861–1870.

19. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv* **2017**, arXiv:1707.06347.
20. Tassa, Y.; Doron, Y.; Muldal, A.; Erez, T.; Li, Y.; de Las Casas, D.; Budden, D.; Abdolmaleki, A.; Merel, J.; Lefrancq, A.; et al. DeepMind Control Suite. *arXiv* **2018**, arXiv:1801.00690.
21. Varma, S.; Sinha, V.; Data augmented Approach to Optimizing Asynchronous Actor-Critic Methods. In Proceedings of the 2022 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), Ballari, India, 23–24 April 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1–5.
22. Mnih, V.; Badia, A.P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In Proceedings of the International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016; pp. 1928–1937.
23. Li, Y.; Hu, G.; Wang, Y.; Hospedales, T.; Robertson, N.M.; Yang, Y. Differentiable automatic data augmentation. In Proceedings of the Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, 23–28 August 2020; Springer: Cham, Switzerland, 2020; pp. 580–595.
24. Auer, P. Using Confidence Bounds for Exploitation-Exploration Trade-Offs. *J. Mach. Learn. Res.* **2003**, *3*, 397–422.
25. Gil, Y.; Baek, J.; Park, J.; Han, S. Automatic Data Augmentation by Upper Confidence Bounds for Deep Reinforcement Learning. In Proceedings of the 2021 21st International Conference on Control, Automation and Systems (ICCAS), Jeju, Republic of Korea, 12–15 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1199–1203.
26. Raileanu, R.; Goldstein, M.; Yarats, D.; Kostrikov, I.; Fergus, R. Automatic data augmentation for generalization in reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 5402–5415.
27. Hafner, D.; Lillicrap, T.; Ba, J.; Norouzi, M. Dream to control: Learning behaviors by latent imagination. *arXiv* **2019**, arXiv:1912.01603.
28. Hansen, N.; Wang, X. Generalization in reinforcement learning by soft data augmentation. In Proceedings of the 2021 IEEE International Conference on Robotics and Automation (ICRA), Xi'an, China, 30 May–5 June 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 13611–13617.
29. Hansen, N.; Su, H.; Wang, X. Stabilizing deep q-learning with convnets and vision transformers under data augmentation. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 3680–3693.
30. Yuan, Z.; Ma, G.; Mu, Y.; Xia, B.; Yuan, B.; Wang, X.; Luo, P.; Xu, H. Don't Touch What Matters: Task-Aware Lipschitz Data Augmentation for Visual Reinforcement Learning. *arXiv* **2022**, arXiv:2202.09982.
31. Yarats, D.; Zhang, A.; Kostrikov, I.; Amos, B.; Pineau, J.; Fergus, R. Improving sample efficiency in model-free reinforcement learning from images. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual, 2–9 February 2021; Volume 35, pp. 10674–10681.
32. Agrawal, P.; Carreira, J.; Malik, J. Learning to See by Moving. In Proceedings of the IEEE International Conference on Computer Vision (ICCV), Santiago, Chile, 7–13 December 2015.
33. Jayaraman, D.; Grauman, K. Learning image representations tied to ego-motion. *arXiv* **2016**, arXiv:1505.02206.
34. Nair, A.; Chen, D.; Agrawal, P.; Isola, P.; Abbeel, P.; Malik, J.; Levine, S. Combining self-supervised learning and imitation for vision-based rope manipulation. In Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 29 May–3 June2017; pp. 2146–2153. [CrossRef]
35. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **2012**, *25*, 1097–1105. [CrossRef]
36. Pathak, D.; Agrawal, P.; Efros, A.A.; Darrell, T. Curiosity-driven Exploration by Self-supervised Prediction. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017; Precup, D., Teh, Y.W., Eds.; 2017; Volume 70, pp. 2778–2787.
37. Paster, K.; McIlraith, S.A.; Ba, J. Planning from pixels using inverse dynamics models. *arXiv* **2020**, arXiv:2012.02419.
38. Christiano, P.; Shah, Z.; Mordatch, I.; Schneider, J.; Blackwell, T.; Tobin, J.; Abbeel, P.; Zaremba, W. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv* **2016**, arXiv:1610.03518.
39. Todorov, E.; Erez, T.; Tassa, Y. MuJoCo: A physics engine for model-based control. In Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vilamoura-Algarve, Portugal, 7–12 October 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 5026–5033. [CrossRef]
40. Hafner, D.; Lillicrap, T.; Fischer, I.; Villegas, R.; Ha, D.; Lee, H.; Davidson, J. Learning Latent Dynamics for Planning from Pixels. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; Chaudhuri, K., Salakhutdinov, R., Eds.; 2019; Volume 97, pp. 2555–2565.
41. Yu, T.; Lan, C.; Zeng, W.; Feng, M.; Zhang, Z.; Chen, Z. Playvirtual: Augmenting cycle-consistent virtual trajectories for reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 5276–5289.
42. Nguyen, T.; Luu, T.M.; Vu, T.; Yoo, C.D. Sample-efficient reinforcement learning representation learning with curiosity contrastive forward dynamics model. In Proceedings of the 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Prague, Czech Republic, 27 September–1 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 3471–3477.
43. Huang, T.; Wang, J.; Chen, X. Accelerating representation learning with view-consistent dynamics in data-efficient reinforcement learning. *arXiv* **2022**, arXiv:2201.07016.
44. Oh, J.; Guo, X.; Lee, H.; Lewis, R.L.; Singh, S. Action-conditional video prediction using deep networks in atari games. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 2863–2871 .

45. Leibfried, F.; Kushman, N.; Hofmann, K. A deep learning approach for joint video frame and reward prediction in atari games. *arXiv* **2016**, arXiv:1611.07078.
46. Racanière, S.; Weber, T.; Reichert, D.; Buesing, L.; Guez, A.; Jimenez Rezende, D.; Puigdomènech Badia, A.; Vinyals, O.; Heess, N.; Li, Y.; et al. Imagination-augmented agents for deep reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5694–5705.
47. Bellemare, M.G.; Naddaf, Y.; Veness, J.; Bowling, M. The Arcade Learning Environment: An Evaluation Platform for General Agents. *J. Artif. Intell. Res.* **2013**, *47*, 253–279. [CrossRef]
48. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing Atari with Deep Reinforcement Learning. *arXiv* **2013**, arXiv:1312.5602.
49. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://www.tensorflow.org/ (accessed on 29 November 2023).
50. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. Openai gym. *arXiv* **2016**, arXiv:1606.01540.