Latent Property State Abstraction For Reinforcement learning

John Burden
Centre for the Study Of Existential Risk
University of Cambridge
jjb205@cam.ac.uk

Sajjad Kamali Siahroudi
L3S Research Center
Hanover
kamali@l3s.de

Daniel Kudenko
L3S Research Center
Hanover
kudenko@l3s.de

ABSTRACT

Potential Based Reward Shaping has proven itself to be an effective method for improving the learning rate for Reinforcement Learning algorithms — especially when the potential function is derived from the solution to an Abstract Markov Decision Process (AMDP) encapsulating an abstraction of the desired task. The provenance of the AMDP is often a domain expert. In this paper we introduce a novel method for the full automation of creating and solving an AMDP to induce a potential function. We then show empirically that the potential function our method creates improves the sample efficiency of DQN in the domain in which we test our approach.

KEYWORDS

Reinforcement Learning, Reward Shaping, Abstraction

1 INTRODUCTION

Advances in recent years have thrust Reinforcement Learning (RL) into the spotlight. RL agents primarily learn through interaction with their environment. The aim for RL is for the agent to learn a suitable policy π that maps states to actions such that the expected cumulative reward is maximised over an episode of the environment. The "Curse of Dimensionality" [1] observes that the number of states in a system grows exponentially with the size of the state representation. This is an issue for many RL algorithms as each state (or those nearby) often need to be visited several times for a satisfactory policy to be learnt. This increase in the sample complexity can drastically slow the rate at which RL algorithms learn.

Reward Shaping (RS) can somewhat mitigate the effects of this "curse" [16], but in most approaches this comes with the cost of incorporating some domain knowledge that has to be provided by a human expert. In these approaches, an intrinsic reward function (the reward comes from within the agent, rather than from the environment) encodes domain knowledge which the agent receives in addition to the reward from the environment.

However, in many cases it is not straightforward and at times rather difficult to encode the required domain knowledge as a reward shaping function. There have been initial attempts at automatically constructing the intrinsic reward function [11], but these approaches require environment dynamics to be known a priori. Solutions to abstract representations of the RL task can also be used to yield shaping functions [4, 7, 12]. This still requires domain knowledge as a formulation of the abstract task. Nevertheless, this requires less domain knowledge than encoding the full shaping function. Unfortunately, the task of encoding or identifying such knowledge for many applications can still be quite difficult and time consuming. This motivates research into the further automation of the construction of RS functions. This would allow for reaping the benefits of reward shaping without the onus of manually encoding domain knowledge or abstract tasks.

Our main contribution is a novel RL technique, called Latent Property State Abstraction (LPSA) that creates its own intrinsic reward function for use with reward shaping which can speed up the learning process of Deep RL algorithms. In our work we demonstrate this speed-up on the widely popular DQN algorithm [13]. Our method works by learning its own abstract representation of environment states using an auto-encoder. Then it learns an abstract value function over the environment before utilising this to increase the learning rate on the ground level of the domain using reward shaping.

We empirically evaluate our novel approach against the baseline DQN agent as well as an implementation of a World Models [8] agent.

2 BACKGROUND AND PREREQUISITES

In this Section we introduce the background on Reinforcement Learning and Abstract Markov Decision Processes.

2.1 Reinforcement Learning

The principal component of Reinforcement Learning is interaction between an agent and an environment [19]. The environment is usually represented by a Markov Decision Process (MDP). We use the standard notation for MDPs:

$$\mathcal{M} = (S_{\mathcal{M}}, A_{\mathcal{M}}, R_{\mathcal{M}}, P_{\mathcal{M}})$$

$S_{\mathcal{M}}$ is the set of states of which $\mathcal{M}$ can take. $A_{\mathcal{M}} : S_{\mathcal{M}} \rightarrow A$ is a function mapping a state to the actions available from that state. $R_{\mathcal{M}}(s, a, s')$ denotes the reward received when transitioning from s to s' via action a. Finally, $P_{\mathcal{M}}$ denotes the probability of moving to state s' when in state s and using action a.

The subscript $\mathcal{M}$ is used to allow us to distinguish between multiple MDPs and also between Abstract Markov Decision Processes which we introduce later. For a more complete overview of RL, see [19].

Many methods exist for training an agent to learn a policy π (a function mapping the current state to the action to perform) that maximises the cumulative reward received by an agent over a single episode of the environment, such methods include simple tabular based methods like Q-Learning [19] and SARSA [17] , as well as newer deep learning approaches such as DQN [13] and PPO [18]. These methods typically suffer from the so-called "Curse of Dimensionality" [1], where the number of possible states increases exponentially with the size of the state-dimension. This often causes learning to be slow in large, complex environments.

2.2 Reward Shaping

An aid to combating this curse is that of Reward Shaping (RS). Reward Shaping operates by providing additional reward $F(s, a, s')$ to the agent after each transition. This additional reward represents external knowledge and is used to steer the agent towards more desirable behaviour.

It was shown in [14] that the optimal policy for a given environment will remain unchanged if F is defined as the difference of potential between two states. That is, we define

$$F(s, a, s') = \omega(\gamma\phi(s') - \phi(s))$$

Where γ is the discount factor and ω is a scaling factor. The total reward received by an agent after each transition has occurred is then $r_{total} = r_{extrinsic} + r_{intrinsic}$, where $r_{extrinsic}$ is the reward given by the environment and $r_{instrinsic} = F(s, a, s')$ is the additional reward for shaping.

The primary issue to consider is the origin of the shaping function. Manually designing a function that indicates the quality of a transition or state is in many cases time consuming or infeasible. This motivates research into automatically constructing such a function.

2.3 Reward Shaping With Abstract Markov Decision Processes

Useful shaping functions can be created from solutions to abstractions of tasks. Abstract Markov Decision Processes (AMDPs) provide a framework for representing these abstractions of MDPs:

$$\mathcal{A} = (S_{\mathcal{A}}, A_{\mathcal{A}}, R_{\mathcal{A}}, P_{\mathcal{A}})$$

Each element of the tuple corresponds to an abstraction of their MDP counterpart, operating on abstract states rather than ground states. Defining these abstract sets and functions is not trivial, and they are often constructed utilising domain knowledge from a human expert. However, even with a human expert, the questions of what abstract states should there be, and what abstract actions, rewards, and transitions should be defined, are often not easy to answer.

A function, Z, is also required in order to map ground states $s \in S_{\mathcal{M}}$ to abstract states $t \in S_{\mathcal{A}}$. The function Z is referred to as the state-abstraction function.

When multiple agents operate on separate levels of abstraction, we refer to the agent interacting with the MDP as the *ground* agent and the agent interacting with the AMDP as the *abstract* agent. We denote this similarly for ground and abstract learning processes.

AMDPs can be used along with Potential Based Reward Shaping (PBRS) [4, 7, 12]. Typically, once the AMDP is constructed, dynamic programming is used to compute a value V for each state. The state value is then used for the potential function for PBRS. That is, for state $s \in S_{\mathcal{M}}$, the potential function for shaping is $\phi(s) \leftarrow \omega V(Z(s))$, for some constant scaling variable ω and state abstraction function Z.

The AMDPs were hand-crafted by an expert in [4] and [7]. On the other hand, in [12], the AMDP is generated from an abstraction function Z along with a set of Options — macro actions — O provided by an expert. Primitive actions were able to be used if no such set of Options was available, but this limited abstract actions to ground actions and doing so would induce a very stochastic AMDP. The game of Othello was used in [12], and constructing the state abstraction function Z involved detailed knowledge about desirable states of the game, such as the agent having its pieces occupy corners and edges. This approach also involved constructing a basic Option that responded optimally within the abstract state-space. None of these approaches are sufficiently automated to scale-up to much larger domains in terms of the cost of encoding domain knowledge.

When utilising AMDPs for PBRS, there is an inherent trade-off to consider: The more "abstract" the AMDP is, the less knowledge the shaping function can encode, but the AMDP will consequently be easier to solve. The reverse also holds — less "abstract" AMDPs can provide more useful shaping functions at higher cost of solving the AMDP. The optimal degree of abstraction will depend ultimately on the size of the task and the level to which abstract knowledge is available or can be obtained.

Previous work [3] has also utilised small, low-level discrete state-space AMDPs to speed up an agent learning to solve continuous-state-discrete-action control problems with deep learning. Here the agent constructed the AMDP based on an initial exploration phase. The agent subsequently solved the AMDP and utilised it with PBRS to speed up its own learning process for the control problems. The biggest drawback of this work was the fact that the abstract states created were uniform in size across each state dimension — thus the AMDP consisted of a number of abstract states exponential in the size of the abstract state-dimension. This limited the domains for which this approach worked to small, low dimensional continuous control problems.

Our work here seeks to address this issue with this work — extending the complexity of the abstract state-space to handle larger, pixel-based state-representations by overcoming the exponential growth of the uniform state abstraction approach. To do this, we employ auto-encoders to learn a continuous latent representation of ground states and use this as our abstract state-space.

2.4 Auto-encoders

An auto-encoder [6] is a neural network architecture consisting of an encoder-decoder pair. The purpose of an auto-encoder is to simultaneously learn both an encoder and decoder by feeding the combined pair training examples (x, x) — that is, the network tries to learn the identity function. After training has completed, the encoder and decoder can be split and used in a modular fashion. Typically auto-encoders bottleneck in size towards the middle of the network — the end of the encoder and beginning of the decoder. This forces the network to learn a compressed representation of the data. The idea being that the encoder then compresses the data into a smaller form and the decoder then decompresses this smaller form into the original data.

A powerful type of auto-encoder is the Variational Auto-Encoder (VAE) [10]. Instead of the encoder outputting a vector of n dimensions, two n-dimensional vectors are output. Rather than outputting an exact vector in the latent space, the encoder now gives a vector of means and a vector of standard deviations $\boldsymbol{\mu} = [\mu_1, \mu_2, ..., \mu_n]$, $\boldsymbol{\sigma} = [\sigma_1, \sigma_2, ..., \sigma_n]$. A single n-dimensional vector is then sampled from each dimension distribution: $\boldsymbol{z} = [\mathcal{N}(\mu_1, \sigma_1), ..., \mathcal{N}(\mu_n, \sigma_n)]$.

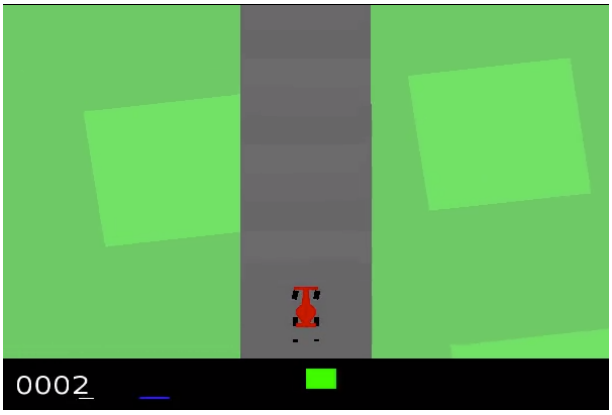


Figure 1: Visualisation OpenAI Gym's Car Racing domain, image captured from the domain running with visualisation on. This is what the agent "perceives" as an array of RGB pixel values.

This z is then fed to the decoder to produce an output. Intuitively, this adds some noise to the input of the VAE and forces a higher level of generalisation to take place.

In our work we use a variational auto-encoder to learn a salient abstract state-space to create the AMDP for PBRS.

3 ENVIRONMENT

The environment on which our approach is evaluated is an action-discretized version of OpenAI Gym's Car Racing domain [2], which has been widely used as a benchmark for Deep RL algorithms. A visual depiction of the environment is given in Figure 1.

In this environment the agent exerts control over a car, and must guide it around a procedurally generated track. The agent must try and keep the car within the track. The car receives a reward of -0.1 after each time step as well as $1000/N$ for each segment of track visited for N the total number of track segments.

The agent perceives a 96×96 pixel representation of the environment, including visual cues for the vehicle's speed, ABS (anti-lock braking system) sensors and the position of the steering wheel.

In the original environment the action-space is essentially $[0, 1]^3$, the accelerator, brakes and steering each take a continuous value between 0 and 1 denoting the extent to which the control is activated.

In our version of the environment we gave the car 20 discrete actions corresponding to different applications of the accelerator, brakes and steering. The reason for doing so is to evaluate agents with a focus on the high-dimensional state-space rather than a more complex action space. We are trying to show that utilising a reward shaping function derived from an agents interactions with an abstract environment can improve learning speed — a continuous action space is not necessary to demonstrate this.

4 RELATED APPROACHES

Recent work has also utilised auto-encoders within the field of RL. In the paper "World Models" [8], the authors train a VAE to reconstruct frames from the OpenAI Gym [2] Car Racing environment. This is used in tandem with a Recurrent Neural Network and a single-layer linear controller. In this approach the auto-encoder is used to construct latent-states from high-dimensional state-spaces. The RNN learns to model the effects of actions on the latent state-space so that these effects can be taken into consideration along with the latent states by the controller when selecting actions. *World Models* relies on an evolutionary approach known as CMA-ES (Covariance Matrix Adaption - Evolution Strategy) [9]. Under this approach, multiple members of the "population" are evaluated concurrently, each having their performance scored over a number of episodes (in our case, 4 episodes) taking the average of these scores to assess each member. After each member has been evaluated, the next generation is created based on properties and strategies favoured by the most fit members of the previous generation. The complete details are beyond the scope of this paper and are given in [9]. While this approach at the time of writing achieves the state-of-the-art performance on the Car Racing domain, it has a high computational time complexity and can take a long time identify strong policies.

Another VAE based approach is found in [5], where the authors attempt to tackle a domain based around manipulating a physical robotic arm in order to complete a variety of tasks. Here the auto-encoder is used to bring down the state-space of webcam frames down to a more manageable size, this latent representation of the images is combined with non-visual state dimensions such as joint angles and velocities. Further, the algorithm utilises "feature presence" on the latent space which essentially filters out latent features which are not highly "activated" in one area of the image. The idea behind this feature presence concept is to remove aberrations from lighting or camera anomalies, it also smooths out latent trajectories. The final algorithm then uses model-based RL to train on the present latent states and non-visual state dimensions.

The approach was evaluated on a robotics-based task, where the agent's action-space consists of a seven dimensional, continuous value, where each dimension corresponds to a torque value to be applied at a robotic arm's specified joint. There are a number of tasks for the robotic arm to complete, ranging from sliding a lego block to a designated position to scooping up a bag of rice with a spatula and transferring it to a bowl.

The VAE-based approach was far more capable than an agent learning without any visual information using the same model-based RL algorithm. Further, this approach also outperformed other auto-encoder-based approaches at the same task.

These methods show that the use of auto-encoders — particularly VAEs — have much potential for improving RL performance. We aim to extend the use of VAEs to Potential Based Reward Shaping.

5 METHOD

Here we give a broad overview of our proposed Latent Property State Abstraction (LPSA) method before detailing the individual steps in the proceeding sections.

The first step is to train the auto-encoder via random rollouts of episodes from the environment that produce samples of high-dimensional states. The auto-encoder trains in a self-supervised manner using these states.

Once the allotted training time for the auto-encoder has elapsed, the encoder and decoder are split and function as a state abstraction

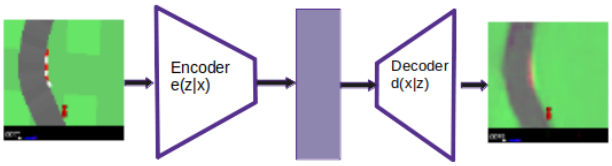


Figure 2: The basic architecture of an auto-encoder and the effect it has on its input.

function and its (approximate) inverse. In other words, the function mapping an MDP state to an AMDP state is defined by the encoding part of the auto-encoder, while the decoding part serves as the (approximate) inverse. For an illustration, see Figure 2. Now that the abstraction function is available (the encoder), the neural network representing the AMDP's Q-function can be trained using a suitable RL algorithm (such as DQN) on the environment but first passing the high-dimensional states from the environment through the encoder to yield abstract states. The AMDP's Q-function is then learning to associate states from the latent state-space to Q-values for each action.

Once the AMDP is trained, we can use it for shaping in the same manner as used in[3] and other shaping methods; the ground network is trained on the desired environment as normal, except that an additional reward is given to the agent based upon the discounted difference in the values of the abstract states it has just moved between.

As long as the learned abstraction function is mapping similar states to similar areas of the latent state-space — requiring the auto-encoder to be "good" — and the abstract network is learning to perform "acceptably" on the latent state-space then the reward shaping can boost the ground agent's learning, particularly early on in the training process.

5.1 Training the Auto-encoder

The encoder is the crux of the LPSA method. The quality of the abstraction function will determine how well the abstract agent can learn a policy for the latent state-space, as well as how applicable the shaping is to the current state.

We opt to use a variational auto-encoder primarily for its ability to create a densely populated latent state space — this will be pertinent for encoding states outside of those encountered by the autoencoder during training.

An agent interacts with the environment using a random policy for a specified number of time-steps. For each training step, a batch of states are drawn uniformly from the training set, the loss is calculated and the weights are updated using the standard VAE loss function [15]:

$$\theta, \phi = \arg\min_{\theta,\phi} \Big(\underset{x \sim e_\theta(z|x)}{-\mathbb{E}} \big[\log d_\phi(x \mid z) \big] + \mathbb{KL}(e_\theta(z \mid x), d_\theta(x)) \Big)$$

Where e_θ is the function represented by our encoder with weights θ, and d_ϕ is the function represented by the decoder with weights ϕ.

The exact architecture we use is given in Figure 3. Once enough episodes have elapsed the training ends.

For our experiments with the Car Racing domain a batch size of 128 was used for each training step and 5000 batches were used for training. Further, 128 dimensions were utilised for the latent state-space. The Car Racing frames are resized from $96 \times 96 \times 3$ to size $64 \times 64 \times 3$.

5.2 Training the Abstract Policy

Now that the auto-encoder has concluded its training, we can utilise the encoder portion of the auto-encoder to act as a state abstraction function. We denote this encoder as a function Z, mapping elements of the ground state-space to the latent state-space.

We train a neural network using DQN on the environment with the latent state-dimensions as the input. Each action is repeated three times by the agent to reduce the computation time required. Furthermore, the observation received by the agent is a stack of four latent states from when an action was last selected. Therefore the shape of each observation is (128×4) and consists of latent states from timesteps $t = 0, -3, -6, -9$, where $t = 0$ is the current time point.

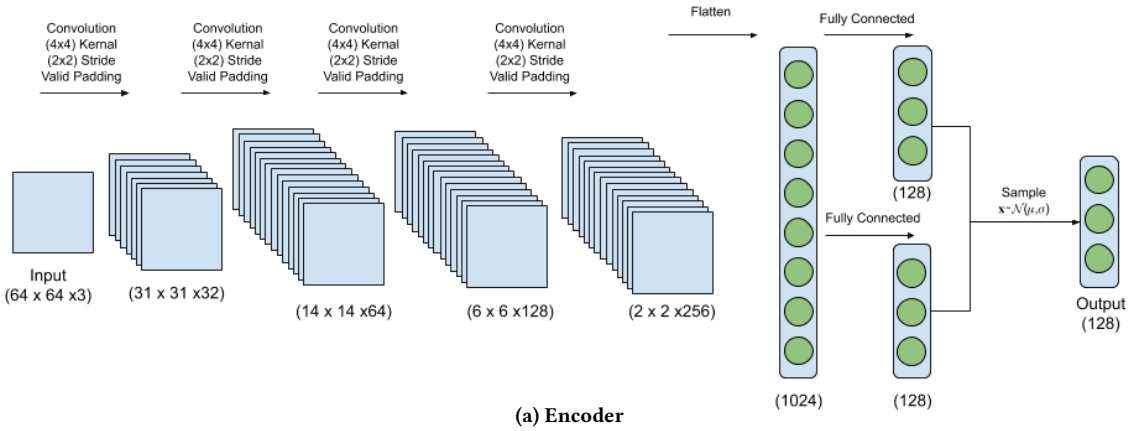
After each observation, the agent adds the following to its experience replay: (o, a, r, o') for current observation o, action a, new observation o' and immediate reward r. Note that o and o' are stacks of latent states as described in the prior paragraph. It is the observations that are added to the experience replay instead of the ground states as in ordinary DQN. From here on, DQN proceeds as normal to learn a policy over the latent state-space with the ground actions available.

It is important to note that even though the abstract agent is learning to associate stacks of latent states with values, the agent is interacting with the ground environment viewed through an abstract "lens". This means that no abstract transition function needs to be learned or provided to the agent — the abstract transition function of the environment is the ground transition function with states mapped into the latent space using the encoding part of the auto-encoder.

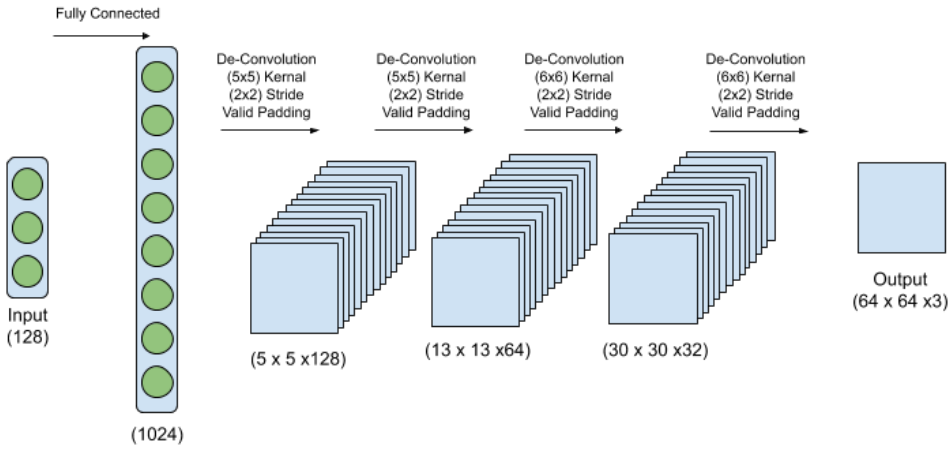
Slightly more formally, assuming that the ground transition function of the environment is $P_{\mathcal{M}}$, then the abstract transition function will be $P_{\mathcal{A}}(Z(s), a, Z(s')) = P_{\mathcal{M}}(s, a, s')$. Of course, this is not known to the agent at the point of learning since $P_{\mathcal{M}}$ is hidden, however this is the transition function that the abstract agent is subject to. Similarly, for rewards, the ground reward is given to the agent, but associated with latent values. $R_{\mathcal{A}}(Z(s), a, Z(s')) = R_{\mathcal{M}}(s, a, s')$.

Once a predetermined number of episodes or steps have elapsed, training finishes and the Q-Network represents the function mapping latent states and actions to values — that is, the abstract Q-function. For our experiments we chose to train both the ground agent and abstract agent for the same number of episodes (8000). A single episode took much less time for the abstract agent however, owing to the smaller neural network requiring fewer calculations to update the weights, as well as the abstract agent generally not surviving as long in the environment in a given episode.

Since the abstract agent is capable of interacting with the ground environment (because of the identical action-space), we can evaluate its performance directly. It is worth noting that we do not expect the abstract agent to perform spectacularly, its perception is limited



(a) Encoder



(b) Decoder

Figure 3: The encoder-decoder components of the auto-encoder architecture.

by only receiving latent states. However, we do not require that the abstract agent performs very well, only that the value function it learns is useful for reward shaping. Specifically, the reward shaping should signal to the agent which areas of the state-action space are more useful than others to explore. In this way, the reward shaping could, for example, discourage behaviour that it has identified as "bad".

For completion we will include the abstract agent's performance in results to verify that training directly on latent states is not inherently superior to Vanilla DQN. This will further support our hypothesis that the "abstractness" of the shaping is providing the boost to the learning performance, rather than access to a simpler environment model.

5.3 Utilising the Abstraction

The abstraction is used to shape the reward the agent receives. The agent interacts with the ground environment using a pre-existing RL algorithm (we used DQN). After a transition $s \rightarrow s'$, the abstraction function Z, which is a VAE, computes abstract states $Z(s)$ and $Z(s')$. From these the abstract Q-function then maps these to abstract values. That is:

$$V(s) = \max_{a \in A_{\mathcal{M}}} Q_{\mathcal{A}}(Z(s), a).$$

Potential Based Reward Shaping is then used to augment the reward received by the agent for the transition $s \rightarrow s'$:

$$F(s, a, s') = r + \omega(\gamma(V(Z(s')) - V(Z(s))).$$

Additionally the experience $(s, a, s', r + \omega(\gamma(V(Z(s)) - V(Z(S'))))$ is added to the agent's experience replay for training. This ensures that the shaped reward is always used for performing Q-value updates when sampled from the experience replay.

5.4 Training the Ground Network

Apart from the modified reward function (and subsequently the experience replay) to utilise reward shaping, the agent operating on the ground level is identical to another agent implementing the ground-algorithm. In our experiments to aid the learning of all agents there is a small amount of pre-processing performed

Parameter	Ground Agent	Abstract Agent
α	4×10^{-4}	4×10^{-4}
γ	0.95	0.95
τ	1×10^{-2}	1×10^{-2}
ω	2.5×10^{-3}	N/A
ϵ	$0.1 \rightarrow 0.01$	$0.1 \rightarrow 0.01$
Batch Size	128	128
Episodes	8000	8000
Warmup Steps	10000	10000

Table 1: Hyper-parameters used for experiments

to the environment states. The image is converted to grey-scale from colour, where each pixel can now take a value between 0 and 255. Each action is repeated three times by the agent to reduce the computation time required. Once again the observation received by the agent is a stack of four frames from when an action was last selected. Therefore the shape of each observation is $(96 \times 96 \times 4)$ and consists of the frames from timesteps $t = 0, -3, -6, -9$.

Table 1 details the hyperparameters used for the ground and abstract agents. These values were found empirically. The implementation of World Models used the hyperparameters from the original paper [8].

6 EXPERIMENTS AND RESULTS

Here we present the empirical evaluation results comparing the performance of our approach against benchmark agents (notably DQN and World Models). We end this section with the final results comparing the Vanilla DQN against our DQN augmented with LPSA. We also compare both of these methods against learning directly on the latent state-space (the AMDP) to ensure that the latent representation is not simply easier to learn on overall.

We opt to present the results in terms of average reward earned per episode as this gives the clearest measure of how successfully the agent is learning within the Car Racing environment. This is because, for the Car Racing domain, the length of an episode varies wildly in length depending on the agent's performance. As such, consistent improvements, beginning early, over the course of training are unfairly penalised when compared to sharp but late increases in performance. For three of the agents (DQN Vanilla, the Abstract Agent and DQN augmented with LPSA) we run each agent for 8000 episodes and they each take approximately 24 hours to complete. However, when we compare against the *World Models* method, the time taken per episode is vastly longer. We therefore limit the *World Models* agent to 24 hours and plot the completed episodes. This ensures a fair comparison that penalises neither steady improvements, nor advantages algorithms that take an inordinately long time to run.

Due to these discrepancies, we also provide the average number of steps performed per second by each agent in Table 2. The results given here highlight the difficulties that may occur when trying to compare methods that are very different. Since we may not wholly rely on comparisons by episodes or time alone a more detailed analysis is required, utilising elements from both.

Additionally, because we are trying to demonstrate the utility of the shaping function learnt, we do not include preprocessing time for any of the agents. For LPSA, this consists of training the auto-encoder and learning the shaping function, for *World Models* this consists of again training the auto-encoder, but also training a Recurrent Neural Network to learn the dynamics of the latent state space. While this may limit the direct practical application of this work, we argue that it emphasises the potential of agents to their own abstractions. For this reason, all plots begin at episode 0 even if they required preprocessing or other initial steps.

As can be seen in Figure 4a, both LPSA and DQN eventually converge on policies with similar scores, however LPSA has a far stronger initial performance. At the end of training, it does appear that DQN has a very slightly higher mean reward, however due to the size of the confidence intervals it is difficult to argue definitively that either outperforms the other once training has finished.

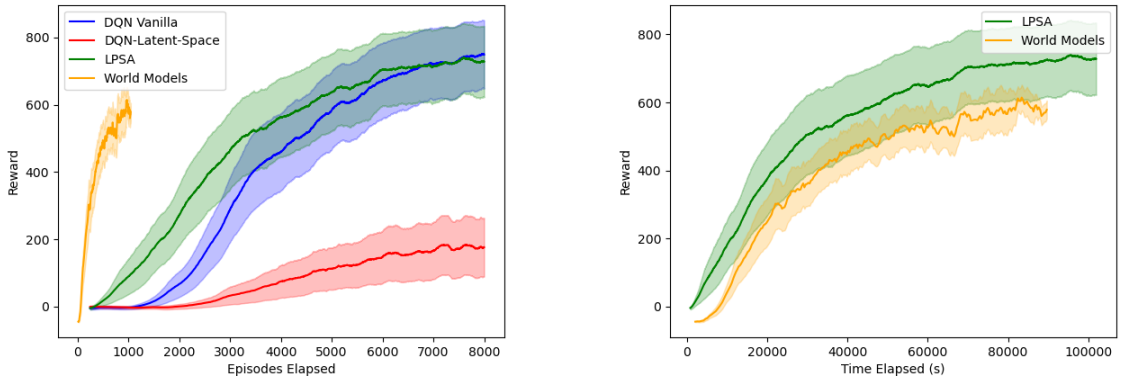
It can also be seen that the abstract agent can quickly converge to a sub-optimal policy. However this sub-optimal policy performs much worse than the that learnt by the other methods. The abstract agent is seemingly limited by the granularity of the abstract environment.

It therefore follows that it is the shaping function derived from the abstract agent that is giving LPSA the boost to learning performance and that it is not the case that the abstract representation of the environment is inherently easier and that we are simply passing the abstract agent's solution to the ground agent — otherwise the abstract agent's final policy would perform much better than it does. The abstract agent's solution to the abstraction of the environment is capturing key skills that are transferable to the ground environment. This makes it apparent that LPSA's use of an auto-encoder is successfully capturing important abstract aspects of the environment.

The aforementioned *World Models* paper [8] at the time of writing claims to have the highest performance on the Car Racing domain and is the current state of the art. We also compare our results against their method.

As Figure 4a shows, the *World Models* approach takes fewer episodes than LPSA to converge on a strong policy. However, this comparison is misleading since each individual episode takes vastly more time to complete for the *World Models* approach. Whilst their approach may allow for an ultimately more stable policy able to perform to a near-optimal level, the time taken to achieve this becomes disproportionately prohibitive with diminishing returns. We ran the *World Models* approach for a approximately the same amount of time as the other agents (roughly 24 hours) and report the results for the amount of episodes that elapsed. At the end of the set training time, *World Models* has a worse performance than either Vanilla DQN or LPSA. In order to demonstrate this, we plot the rewards received for both LPSA and *World Models* against time in Figure 4b, showing consistently stronger performance by LPSA when compared to *World Models*.

Due to *World Models* making use of a populational approach to learning we opt to disable the early-stopping mechanism for this agent in order to maximise computational throughput,. This way, each member of the population can always be learning, rather than waiting after potentially stopping early — the majority of this time will have to be spent anyway as certain agents begin to excel



(a) Rewards against Episodes

(b) Rewards against Time

Figure 4: The average reward received by an agent in the Car Racing domain plotted against either the number of episodes completed (a) or the elapsed time (b). Each agent was trained for approximately 24 hours. The averages are based on 25 repetitions of the learning process for each agent and the shaded regions represent a 95% confidence interval.

Vanilla DQN	AMDP	LPSA	World Models
18.45	22.42	13.38	10.63

Table 2: Average number of steps per second performed by each agent over the 25 training runs (rounded to two decimal places).

and would not trigger the early-stopping mechanism. There is a slight trade-off here as even the best agents do not always use the maximum amount of steps in this domain, but this seems the best choice when there is a large number of members in the population.

This is all relevant to our evaluation because each episode now takes many more steps to complete — at least initially in the training procedure. The effects of this are two-fold: first, each episode takes longer as it is comprised of more steps, and second, more insight can potentially be gleaned from each episode (again, there are more steps).

All of these factors together make for a difficult comparison. Nevertheless, our evaluation clearly shows that LPSA is able to utilise reward shaping from an automatically learned shaping function to perform on a similar level to the *World Models* approach, and has a significantly lower computational time complexity.

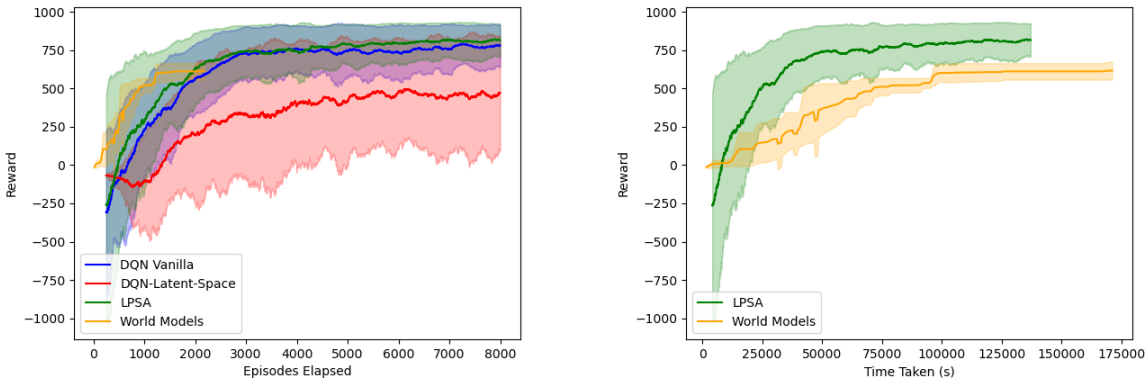
6.1 Further Experiments and Results

In addition to the experiments carried out above, we also performed experiments in a slightly modified instance of the Car Racing environment. This allows us to verify that our approach works in other scenarios. In this modified domain, we remove the termination of the episode for leaving the track and instead punish the agent for doing so. If the agent does not reach a new track segment in 12 actions (36 time steps), rather than terminating the episode, the agent instead receives −1 reward. The agent continues to receive this negative reward until it reaches the next track segment. The episode will continue until either 1000 time steps have elapsed or the agent completes the track.

Figure 5a shows the average results from five repetitions for the agents interacting in this environment, with three repetitions for *World Models*. Here the advantage of LPSA is less clear, but still present when compared to DQN. However, in this modified environment *World Models* has less of an episodic advantage and after a similar amount of training time has reached a less strong policy when compared to either DQN or LPSA. Further, we also plot the results against time for LPSA and *World Models* in Figure 5b — this highlights the performance of LPSA over *World Models* further.

7 CONCLUSION

In this paper we have introduced a new approach — LPSA — for generating state abstractions that can assist the learning rate of RL algorithms. LPSA is capable of improving an agent's learning performance through reward shaping by utilising variational auto-encoders to create an abstract state space. LPSA is capable of handling high dimensional state-spaces. Our results indicate that agents using DQN shaped with LPSA outperform Vanilla DQN in the Car Racing domain — with the caveat that the shaping function and auto-encoder have been learned a priori. Nevertheless, this is a significant contribution, as it shows an entirely automatic process for an agent to create its own abstraction and to then utilise this abstraction in order to give a pronounced boost to the learning performance.



(a) Reward against Episodes

(b) Reward against Time

Figure 5: The average reward received by an agent in the modified Car Racing domain (without early termination) plotted against either the number of episodes elapsed (a) or the elapsed time (b). All of the agents except for World Models were run for 8000 episodes. World Models was run a similar amount of time (approximately 45 hours). For DQN, DQN on the latent state space and LPSA the plot shows the mean result after five repetitions. For World Models the plot shows the mean result after three repetitions. The shaded regions denote one standard deviation in the results observed.

REFERENCES

[1] Richard Bellman. 2010. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

[3] John Burden and D. Kudenko. 2020. Uniform State Abstraction For Reinforcement Learning. In *ECAI*.

[4] Kyriakos Efthymiadis and Daniel Kudenko. 2014. A comparison of plan-based and abstract MDP reward shaping. *Connection Science* 26, 1 (2014), 85–99.

[5] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. 2015. Learning Visual Feature Spaces for Robotic Manipulation with Deep Spatial Autoencoders. *CoRR* abs/1509.06113 (2015). arXiv:1509.06113 http://arxiv.org/abs/1509.06113

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[7] M. Grzes and D. Kudenko. 2008. Plan-based reward shaping for reinforcement learning. In *2008 4th International IEEE Conference Intelligent Systems*, Vol. 2. 10–22–10–29. https://doi.org/10.1109/IS.2008.4670492

[8] David Ha and Jürgen Schmidhuber. 2018. World Models. *CoRR* abs/1803.10122 (2018). arXiv:1803.10122 http://arxiv.org/abs/1803.10122

[9] N. Hansen and A. Ostermeier. 1996. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*. 312–317.

[10] Diederik P. Kingma and Max Welling. 2019. An Introduction to Variational Autoencoders. *CoRR* abs/1906.02691 (2019). arXiv:1906.02691 http://arxiv.org/abs/1906.02691

[11] M. Marashi, A. Khalilian, and M. E. Shiri. 2012. Automatic reward shaping in Reinforcement Learning using graph analysis. In *2012 2nd International eConference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 111–116. https://doi.org/10.1109/ICCKE.2012.6395362

[12] Bhaskara Marthi. 2007. Automatic Shaping and Decomposition of Reward Functions. In *Proceedings of the 24th International Conference on Machine Learning* (Corvalis, Oregon, USA) *(ICML '07)*. ACM, New York, NY, USA, 601–608. https://doi.org/10.1145/1273496.1273572

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[14] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. 1999. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 278–287. http://dl.acm.org/citation.cfm?id=645528.657613

[15] Stephen G. Odaibo. 2019. Tutorial: Deriving the Standard Variational Autoencoder (VAE) Loss Function. *CoRR* abs/1907.08956 (2019). arXiv:1907.08956 http://arxiv.org/abs/1907.08956

[16] Jette Randløv and Preben Alstrøm. 1998. Learning to Drive a Bicycle Using Reinforcement Learning and Shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 463–471. http://dl.acm.org/citation.cfm?id=645527.757766

[17] G. Rummery and Mahesan Niranjan. 1994. On-Line Q-Learning Using Connectionist Systems. *Technical Report CUED/F-INFENG/TR 166* (11 1994).

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

[19] Richard S. Sutton and Andrew G. Barto. 2017. *Introduction to Reinforcement Learning* (2 ed.). MIT Press, Cambridge, MA, USA.