Knowledge-free domain-independent automated planning for games

Raphael Lopes Baldi
Pontifical Catholic University of Rio Grande do Sul, Brazil
Ipiranga Av., 6681, 90619-900
Porto Alegre, RS, Brasil

Abstract

Classical planning algorithms require us to first formalize problems as a symbolic model. That represents a knowledge acquisition bottleneck since it requires us to either get help from a specialist or acquire the knowledge ourselves, after which we need to describe the problem's domain formally. In both situations, the person in charge of creating the model needs to collect all states and transitions to use a classical planner. In this work, we intend to explore a technique to learn a transition function from the latent representation of the domain's states, and use it as input for an A* based planner.

Introduction

Planning is one of the various techniques Artificial Intelligence researchers use to solve problems. It is a process to select and organize actions to get to a target environment's state, given the current environment's state. Agents are capable of, using some formalization of the problem's domain, anticipate the outcome of actions while creating the plan (ordered sequence of actions) to achieve the goal state. Automated planning is the area of AI in which we design, implement and evaluate the deliberation process to select and order actions, computationally [4].

One of the most time-consuming tasks when dealing with automated planning is the creation of a formal model for environments. Usually, the job requires a broad knowledge of the system and this knowledge not always falls under the AI developer's expertise, which makes the modeling task time-consuming and prone to errors [1]. The primary objective of this work is to use a technique to automatic extract the domain definition from images (game's frames), and use that definition with a planner to make the agent capable of playing Atari games.

It is hard to define games as a planning problem manually due to the exponential explosion of states. Taking the Atari video game as an example, the console renders the screen 60 times (or frames) per second, and for each of those frames, the player can choose one action from the 17 available - including the no operation action. If we were to expand all possible transitions, at the end of the first second, we would have to evaluate 17^{60} transitions. In 2017, Asai and Fukunaga introduced Latplan, which purpose goes in line with our objective: it uses a series of neural networks to learn an encoding of the domain, and a transition function that allows a planner to use search methods - like A* - by sampling such networks.

Our goal is to develop an approach to automatically play video games using planning algorithms, without prior knowledge of the problem's domain, and without the need to engineer the domain formalization by using a technique for extracting the domain definition from game images and memory [1]. All implementation, configuration, datasets, and instructions on how to execute the solution we developed are publicly available at https://github.com/raphaelbaldi/latplan-ale.

Game Latplan

In this work, we develop adaptations, and extensions to the LatPlan planning technique of Asai and Fukunaga in order to create a system capable of planning in video games environment without prior knowledge of the video games' domains. Our architecture introduces some *new elements* to the original LatPlan. First, a framework to extract frames from Atari games using the *Arcade Learning Environment*. Second, an autoencoder capable of learning a latent representation for the frames of a specific game; and third, a neural network to convert the sequences of images that comprise LatPlan's output into an executable plan for a video game playing agent controlled by our planner.

We introduce four significant changes to Latplan. First, we limit the number of actions we execute every frame: since the autoencoder neural network can infer states from its built-in knowledge, we do not need to provide it with every possible state change as LatPlan does. Second, our architecture does not need to have frames corresponding to all possible states for a given video game to work. Third, we evaluate training the State Autoencoder at the same time we generate the training and test datasets. Finally, we introduce an artificial neural network – a Command Categorizer – to allow our architecture to execute the plans we generate. The Command Categorizer learns a transition function $\gamma = S \times S \rightarrow C$, where S is the set of states' latent representations obtained from the State Autoencoder, and C is the set of valid commands in the game. This chapter describes the modifications we did to build our Game Latplan architecture.

Game Latplan architecture

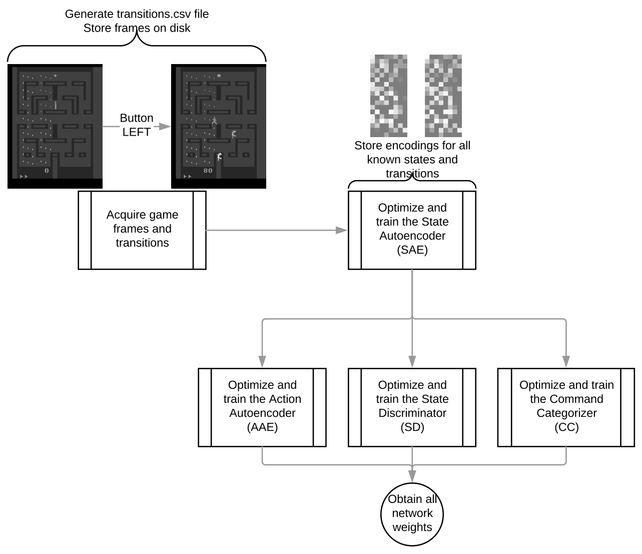
Generate transitions.csv file
Store frames on disk
Button LEFT
Store encodings for all known states and transitions
Acquire game frames and transitions
Optimize and train the State Autoencoder (SAE)
Optimize and train the Action Autoencoder (AAE)
Optimize and train the State Discriminator (SD)
Optimize and train the Command Categorizer (CC)
Obtain all network weights

Figure 1: Game Latplan learning phase.

Our architecture is composed of six components: frame acquisition, State Autoencoder, Action Autoencoder, State Discriminator, Command Categorizer, and a classical planner. We can split our process into two phases. In the first phase, that we illustrate in Figure 1, we learn a model for the game we want to play. In Figure 2 we show the second phase, in which we use the model we learned to generate plans for the game. The next sections describe in detail each of the components in our architecture, which are, in summary as follows:

1. the frame acquisition component is responsible for acquiring data from the game we are interested in planning;
2. the State Autoencoder then learns a latent representation of the state space and generates the latent representation for each known state;
3. using the latent representations from the previous step, we train an Action Autoencoder;
4. we use the latent representation for the states, alongside the commands we collect during dataset acquisition, to train the Command Categorizer;
5. we generate fake latent representations and use it in combination with the latent representations we obtain in the second step to train a State Discriminator;
6. using the neural networks we trained, we run the planner to obtain a sequence of states' latent representations that represents the plan to go from a given initial state to a goal state.
7. we use the Command Categorizer to transform the sequence of latent representations into a sequence of commands to send to the video game emulator to execute the plan.

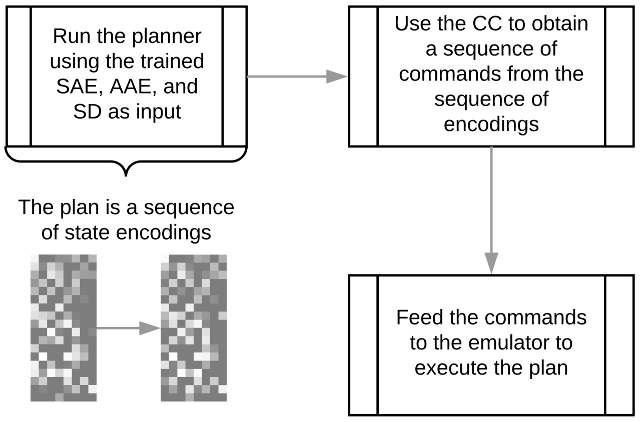
Run the planner using the trained SAE, AAE, and SD as input
Use the CC to obtain a sequence of commands from the sequence of encodings
The plan is a sequence of state encodings
Feed the commands to the emulator to execute the plan

Figure 2: Game Latplan planning phase.

Dataset acquisition and frame processing

We chose to work with an Atari game called Alien. This game is similar to Pac-Man: we have to run through a maze collecting dots, that grants us points, without getting killed by the ghosts – in Alien's case, by the monsters. But while in Pac-Man we can kill ghosts by collecting an energy ball, in Alien we have to collect the energy ball, collect ammunition for our weapon, and then shoot the monsters once in close range. The new set of actions requires the player to select a path taking both ammunition and the energy balls into account.

We implemented two methods to extract our training dataset from game playing sessions in the Arcade Learning Environment [2] Atari emulator. The first method runs the simulator in random mode: for each step it takes, it selects a random action until it reaches either a game-over state or a given number of frames. The second method allows a human to guide the exploration of the transition function by playing the game. We used a technique to skip frames while generating the dataset (frame skip): the emulator repeats each command it receives for the number of frames we set before returning the next state of the game. In both cases, we set the frame skip to 3, and the horizon to 3600 frames, meaning that for each frame extraction episode we collect at most a minute of gameplay[1]. To do the frame skip, we repeated the selected action for the number of frames we wanted to skip, and then we captured the resulting frame.

Our choice to skip three frames per capture has two key reasons, illustrated in Figure 4. First, since the difference between two sequential frames is small we found the State Autoencoder could generalize better with fewer, but more different states. Second, since the Alien game renders half the content for the level for each frame, we need to skip an odd number of frames to be able to collect a full rendering of the game screen from the domain. Figure 3 illustrates these two phenomena, note how odd-numbered frames only render game state on the right side of the screen, and how the

[1]The Atari video game renders the screen 60 times per second, hence generating 3600 frames will produce at most one minute of game footage.

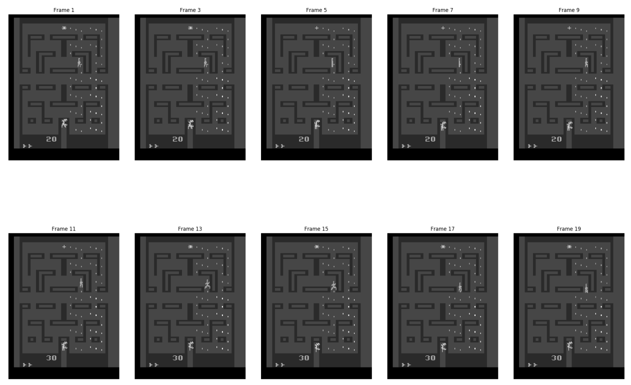

Figure 3: 20 frames, collecting every other frame.

difference between two frames is very often a single pixel. If we were to select an even number of frames to skip, we would have to deal with two consequences: the State Autoencoder would become unable to generalize even frames that we skipped during the dataset extraction phase; and the human players would have a hard time moving through the level since they would see just half of the screen.

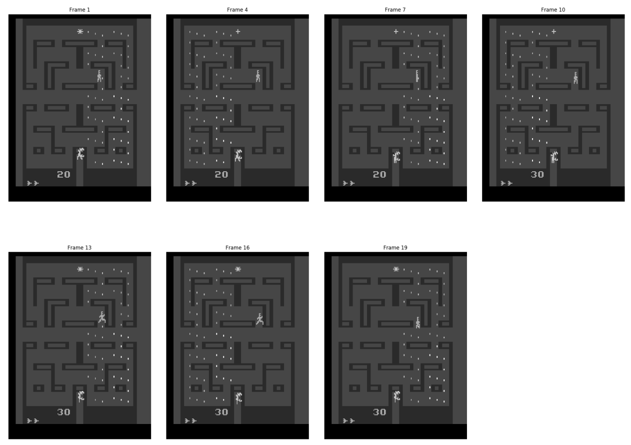

Figure 4: 20 frames, collecting every third frame.

Artificial neural networks

Our goal while defining the infrastructure for training and testing the artificial neural networks we wanted was to create a framework that could make it simple for us to make changes to the hyperparameters and architecture of the components, run hyperparameter search independently of the network architecture while minimizing memory usage. Asai and Fukunaga uses images that are significantly smaller than Atari frames, which results in memory exhaustion when loading the full dataset to train the neural network. To solve this, we implemented a system capable of running both training, testing, and hyperparameter search using generators. A generator is a feature from the toolset we used that allows us to generate – in our case, load from disk – the training and test sets as needed by the network we are training. This way we could keep training data on disk until we needed to use it for the current batch.

State Autoencoder

The State Autoencoder is the artificial neural network responsible for learning the latent representations of the game's frames. We implemented it as a fully convolutional neural network that we split into two parts: an encoder which takes an image as input and converts it into a latent representation; and a decoder which takes a latent representation and reconstructs the initial image. Our training objective is reducing the mean squared error of the image reconstruction, using Adam [6] as the optimizing algorithm. In Figure 5 we illustrate the architecture for the State Autoencoder, with three convolutional layers. We kept the neural network implementation as flexible as possible to allow for easy iteration over multiple combinations of hyperparameters, hence the number of attributes to set.

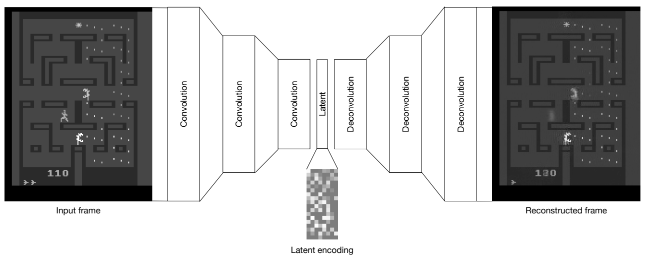

Figure 5: The State Autoencoder architecture.

The resulting artificial neural network for the Alien game contains 3,178,593 trainable parameters. Our first layer deals with reshaping the input data into a standardized input, while the second one adds Gaussian noise to the data. Adding noise to the input has two effects: it reduces the possibility of the State Autoencoder overfitting the training data, and it makes the State Autoencoder robust to noisy input. Since we use it to reconstruct frames from latent representations that we reconstruct from the output of the Action Autoencoder, robustness to noise allows us to pull closer to valid frames during the reconstruction phase of the planner. We disable the noise layer during predictions.

Each *convolution* box we show in Figure 5 is a pair of a convolutional layer followed by a *max pooling* layer. The objective of each max pooling layer is downscaling the image to reduce the number of weights when connecting the convolutional layers to our latent layer. Using such pairs makes the state encoder (the first part of the State Autoencoder) to apply a series of filters (convolution), and then select the maximum value for each area (max pooling), discretizing the input data. We set the number of filters using the *clayer* hyperparameter, and the size of each filter using the *ker-*

nel_size hyperparameter. The final layer of the encoder is a fully-connected layer that holds the latent representation for the input.

Our decoder's input is the latent layer – the last layer of the encoder. Connected to the latent layer we have a series of pairs of *upsampling* layers connected to deconvolutional layers. We show those pairs of layers as *Deconvolution* boxes in Figure 5. The *upsampling* layers are responsible for upscaling the information from the previous layer by replicating each data point to fit the filter size, and the deconvolutional layers are learning to reconstruct the input data. The final two layers are responsible for adjusting the data back to the correct input by padding it with zeroes – if necessary – and reshaping the data back to the make it fit the input shape. After training the network, we output the latent representations for all the frames we collected – using the system we describe in Section – for the target game, and store it in a file. We also store latent representations for the known transitions in our dataset, which we generate by combining the latent representation of prior and successor states for all known transitions.

Action Autoencoder

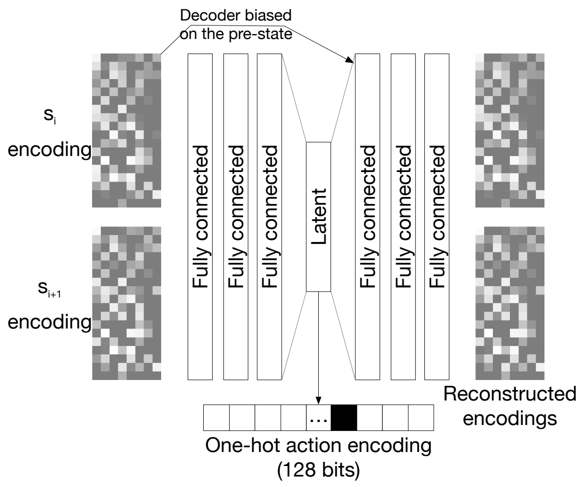

Figure 6: The Action Autoencoder architecture.

The Action Autoencoder is the network responsible for learning the transition function in our domains. Its input is a pair of latent representations for known states – s_i and s_{i+1}, a state and its successor – and the objective of it is to reconstruct those while making the bottleneck layer to converge into a one-hot categorical vector. Figure 6 ilustrates the general architecture of the Action Autoencoder. Analogously to the State Autoencoder, we describe in Section , the Action Autoencoder contains two components. The first one is an encoder that takes the pair of state's latent representations as input and outputs an action's latent representation (one-hot categorical vector). The second component is a decoder that takes a state latent representation, and an action latent representation as input and outputs the reconstruction of the state latent representation and the successor state latent representation after the application of such action.

Each *Fully connected* box in Figure 6 is a sequence of three layers: a fully-connected layer, a batch normalization layer, and then a dropout layer. It is important to notice that the Action Autoencoder will use a small temperature as the minimum value for the temperature annealing as we want the action latent representation to converge into a one-hot categorical vector.

State Discriminator

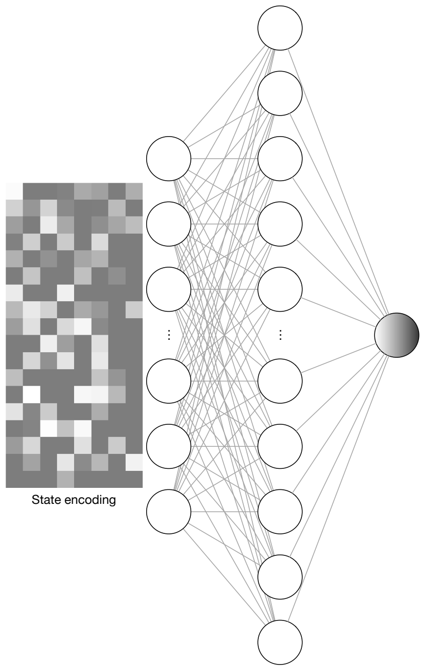

Figure 7: The State Discriminator architecture.

One crucial artificial neural network in our architecture is the State Discriminator, as the Action Autoencoder generates invalid successor states during planning, and we need a way to avoid expanding those. The State Discriminator is responsible for learning what is a valid state, by analyzing the state's latent representation. We use it during the planning phase to prune out states that are not valid for the game we are running. The input of the network is a state's latent representation, and its output is the probability of the state be valid. Figure 7 shows the architecture of the State Discriminator artificial neural network.

Analogously to Latplan, we use the PU-Learning framework [3] to train this classifier. PU stands for positive and unlabeled samples, and the framework introduces a system to train a classifier when we have only positive and unlabeled samples. The framework introduces the idea of doing a set expansion, from a set of positive samples we have, to obtain a mixed set of samples from which we can train a classifier. We can generate the mixed set using the State Autoencoder (see Section) as a generative network, by inputting samples to the decoder's input. An essential difference between our architecture and Latplan's architecture is the fact that we do not have a way to know all valid states in the game. Latplan relies on the ability to enumerate all states for a given puzzle during the generation of fake states, to ensure it is not labeling positive samples as negative. We cannot do that for a video game, due to a large number of possible states, even for a short gameplay session.

To generate the fake state latent representations we need to train the discriminator, we generate a vector of random latent representations. We then run this vector through the State Autoencoder's decoder, to obtain reconstructions for the random input. Finally, we run the frame reconstruction from the previous step through the State Autoencoder's encoder, to obtain a new latent representation. We repeat the process until we reach a reconstruction loss threshold that approximates reconstruction loss of the fake states to the reconstruction loss of valid states, but still large enough to ensure those frames are not valid. Given the two sets – valid states, and fake states – we create a new, mixed set, where we label valid states as 1 and fake states as 0. We keep generating fake latent representations until we draw a set as big as the set of valid state's latent representations. After training the State Discriminator, we compute the PU constant by obtaining the mean value for the test set, selecting only valid states. Figure 8 shows a few samples of the fake states we generated to train the State Discriminator. Since we do not have a way to tell a frame is not valid – we do not have a method to check for it on the emulator – we assume that generating frames from latent representations we generate at random and reconstructing them while the reconstruction loss is high, yield invalid reconstructions.

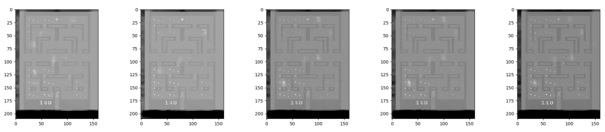

Figure 8: Fake states we generate to train the State Discriminator.

Our State Discriminator has 99.37% accuracy on the positive set and 1% accuracy on the negative set. It means our discriminator is not helping the planner to prune out invalid states, due to the high percentage of false positives we are obtaining. From Figure 8 we can conclude that even using a reconstruction loss threshold (10^{-2}) to generate fake states that are orders of magnitude larger than the reconstruction loss (10^{-4}), the resulting frames are still too similar to valid frames for the discriminator to tell them apart.

Command Categorizer

One fundamental limitation of Latplan [1] as a technique for playing video games is that we cannot execute the resulting plans since the output of the planner is a sequence of states' latent representations. We can convert these latent representations to frame's reconstructions, but just that is not enough to make plans applicable in our environment. In this work, we define a command to be the input we give to the video game in order to act in the computer game (a button press), and an action to be the formalization of a transition in the problem's domain. We propose a new artificial neural network to allow us to execute the resulting plans on the Atari emulator. This network learns the command to execute to make the video game advance from one state to a successor state given the latent representations for both states. Given two states' latent representations from the game, the Command Categorizer outputs the command we should perform to go from the first state to the second state.

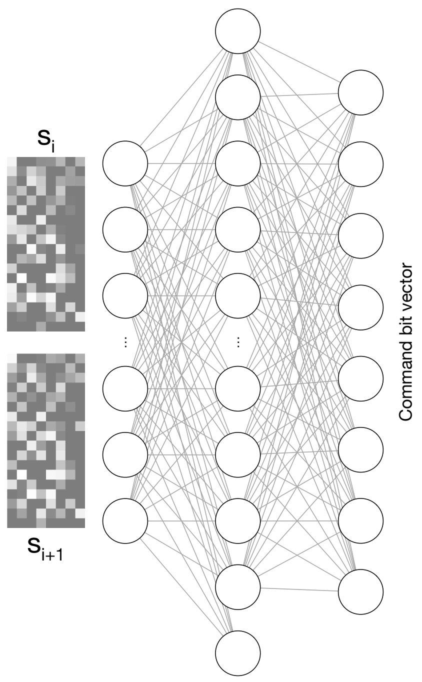

Figure 9: The Command Categorizer architecture.

We tested two different architectures for this network. In the first one, we extracted the resulting command – the combination of all buttons we are pressing – as a category, and then we created a single category classifier. The output layer had a softmax activation, and the network goal was to detect the command as a category. Since we have multiple ways to go from one state to the other, representing it as a category yields poor results. This network's accuracy was of 6%, which is slightly better than guessing[2]. Figure 9 shows the organization of the artificial neural network we use for this task.

The second approach we took was to consider the byte containing all buttons' state as the output of a multicategory classification network. The Arcade Learning Environment exposes all Atari commands and combines them in a byte for executing on the emulator [2], so this approach is also valid. During execution, we prune out commands that are out of range (only the first 5 bits are used to represent buttons). Doing so we thought the network would learn the most important button to press for each transition. Our net-

[2]Since we have 18 possible combinations of commands, a random guess would have a 5.5% chance of obtaining the right result.

work has 36% accuracy for recognizing commands given the latent representations of two states in the Alien game, using the command combination approach. We also trained the Command Categorizer using the game's frames as input, and even though the results were similar, when using the frames' reconstruction in a fully-convolutional network the result was worse (15% of accuracy).

Classical planner

Our classical planner, which is an adaptation of Asai and Fukunaga planner to work with non-binary data, uses a search algorithm to go from the initial state, given its latent representation, to the goal state, also using its latent representation. In our implementation, we use an infrastructure that allows for changing the search algorithm and heuristics with ease. For the Alien game, our distance heuristic is the difference between the latent representation for the current state and the latent representation of the goal state. We use the Greedy Best-First Search algorithm to explore the state space.

Our planning starts by evaluating what actions the Action Autoencoder knows. We retrieve the list of known transitions in our system and generate the action latent representation for each one, by running each pair $(pre, succ)$ through the Action Autoencoder. The planner is an adaptation of search algorithms: we compute the list of successor states using the Action Autoencoder. We combine the current state latent representation with each one of the actions the Action Autoencoder learned, and decode it.

The next step after expanding each state is pruning those that are not valid. We check each latent representation we obtain from reconstructing the successor state using the Action Autoencoder using the State Discriminator. If the neural network tells us the state is invalid, we stop the expansion. We then use the State Autoencoder to check for the frame reconstruction, decoding it, and then autoencoding the resulting frame again. We verify if the mean squared error between the state expanded by the Action Autoencoder and the state we reconstruct using the State Autoencoder is less than a threshold (0.01), pruning all the states that fail the test.

For each state, the planner uses the State Autoencoder to check if it is the goal state, once again by checking the mean squared error between the goal state reconstruction and the current state reconstruction. Once we reach the goal state, we have a sequence of state's latent representations. We run each pair of latent representations through the Command Categorizer to obtain the command we need to execute on the emulator, and then execute the resulting plan on the *Arcade Learning Environment*, by feeding each action, one after the other.

Results

When testing our architecture, we were unable to make the planner produce effective plans. Since the Action Autoencoder produces a large number of invalid transitions, and we do not have an implementation equivalent to the Action Discriminator, the planner expands a lot of invalid states. Our State Discriminator is pruning very few invalid states as well, meaning the number of states we expand grows exponentially. In our tests, the State Discriminator pruned 13% of the frames for each expansion, meaning the planner adds 86 new states per step[3].

Conclusion

We found that even though the technique of Asai and Fukunaga is suitable to solve the planning problem without prior knowledge of the domains – as we can see in their results – that relies on us designing an artificial neural network capable of both reconstructing states, and encoding them as categorical vectors, which still requires us to think about domain-specific solutions. Our approach, which uses non-categorical latent representations of the domain's states also produces sub-optimal results.

Moving forward we want to test our State Autoencoder with all video games available for the *Arcade Learning Environment*, and datasets removing features that may lead to poor reconstruction (scores, and blank spaces around the gameplay area). We want to experiment with color images, instead of grayscale, and evaluate changes to our State Autoencoder, and hyperparameters that might help us make it to successfully force the bottleneck layer into a categorical representation. We want to experiment with different methods to train a State Discriminator, potentially using a Generative Adversarial Network [5].

References

[1] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *CoRR*, abs/1705.00154, 2017.

[2] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Int. Res.*, 47(1):253–279, May 2013. ISSN 1076-9757.

[3] Charles Elkan and Keith Noto. Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 213–220, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401920.

[4] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, New York, NY, USA, 1st edition, 2016. ISBN 1107037271, 9781107037274.

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[3] In our test case our Action Autoencoder only used 96 bits to represent all known transitions.