Finding action-state Similarities in Tabular Reinforcement Learning Using Low-Dimensional Embeddings

Gabriel Rubin
Pontifical Catholic University of Rio Grande do Sul
gdearrud@gmail.com

Abstract

This paper proposes a method to automatically detect action-state similarities in temporal difference learning methods. Previous work showed that domain-specific action-state similarity functions can be used to speed up the training process of these learning methods on many domains, such as the popular video-game Super Mario Bros. We intend to use our automatic similarity function to speed up an agent on that same domain, without domain-specific information. In this paper, we will detail the technical aspects of our method, and analyze its performance against the standard Q-Learning method on the Super Mario Bros domain.

Introduction

Reinforcement learning (Sutton et al. 1998) methods are the state-of-the-art solutions in a variety of domains, from autonomous robots (Riedmiller et al. 2009) to recent developments that enabled agents to master classic video-games (Mnih et al. 2015). Such methods require an agent to learn by training: the agent explores and interacts with an unknown environment for several time-steps in order to learn how to perform an episodic or continuous task. However, this training process can often be expensive resource-wise and time consuming.

Many methods were developed in order to speed up an agent's training process from a designer's perspective, such as using domain-specific knowledge to shape its rewards (Ng, Harada, and Russell 1999) and custom strategies (Ribeiro 1995). Recent work (Rosenfeld, Taylor, and Kraus 2017) introduced a new method called SASS that can speed up this training process significantly for temporal difference methods, such as Q-Learning (Watkins and Dayan 1992), by *spreading* the Q-function estimates of an state to other similar states. SASS considers a custom similarity function that relies on a designer's input in order to compare action-state pairs. However, it is often desirable to automatize such designer-dependent processes in order to achieve a generalized solution as opposed to a custom one for each domain.

This paper proposes a novel method for automatically finding action-state similarities by comparing the distance between their low-dimensional embeddings (Nikol'skii 2012). These embeddings are generated by an autoencoder (Hinton and Salakhutdinov 2006) trained with action-state data from the domain. We tested the performance of our method against a classic Q-Learning agent on an unofficial Super Mario Bros domain used in many AI championships (Togelius et al. 2013).

In the following sections we will describe the technical aspects of our solution and how we tested our implementation. Then, we analyze our results against the Q-Learning method and elaborate on the strengths and weaknesses of our solution.

Background and Previous Work

Reinforcement learning problems and domains are usually represented through a Markov Decision Process (MDP). MDPs are a classical formalization of sequential decision making, where an agent interact with an environment through a set of actions. Taking actions yield immediate and delayed rewards to the agent, and changes the environment configuration (state). This change is expressed as a transition from one state to another, and perceived by the agent as an experience tuple $\langle s, a, r, s^{'} \rangle$, where s is the last state, a the action taken from state s, r the reward yielded from taking a from s, and $s^{'}$ the environment's next state. The environment have an action-space A and a state-space S which contain all the possible actions and states respectively.

Temporal difference methods, such as *Q-Learning*, learn to estimate the value of taking action a from state s by means of a function called *Q-Function* that receives an action-state pair $\langle a, s \rangle$ as input. This $\langle a, s \rangle$ pair is used as an index in a table called *Q-Table*, which stores the calculated *Q-Value* for that $\langle a, s \rangle$ pair. A *Q-Learning* agent calculates its *Q-Values* through a value iteration update according to its experiences while interacting with the environment.

Recent work (Rosenfeld, Taylor, and Kraus 2017) introduces a new Q-Learning variant, called SASS, that

accelerates the learning process by spreading the *Q-Value* of an action-state pair to similar action-state pairs according to a human-designed similarity function σ. The SASS agent utilizes a modified *Q-Learning* update function, where, for each experience $\langle s, a, r, s^{'} \rangle$, more than a single $\langle a, s \rangle$ entry is updated on the Q-Table. Each similar action-state $\langle \tilde{a}, \tilde{s} \rangle$ is updated according to Equation 1, where $\sigma(s, a, \tilde{s}, \tilde{a})$ is the similarity function, δ is the temporal difference error term, and α is the learning rate.

$$Q(\tilde{s}, \tilde{a}) = Q(\tilde{s}, \tilde{a}) + \alpha\sigma(s, a, \tilde{s}, \tilde{a})\delta \quad (1)$$

Automatic action-state Similarity Detection Using Embeddings

A typical method for finding similarities in data is to compare the distance of elements in the data using some distance metric, such as the Euclidian distance. In reinforcement learning, a state is often represented as a vector of features that describe the environment's current state of affairs. For complex environments, states can have many features and their vector representation will consequently be in a high dimensional space. Working with elements in high dimensions can lead to the *curse-of-dimensionality* (Aggarwal 2005): it is often computationally expensive, hard to visualize, and prune to over-fitting.

Low-dimensional embeddings are lower dimensional representations of high dimensional data that can hold the *semantics* of the original data. Embeddings can be generated by a type of neural network called autoencoder (Hinton and Salakhutdinov 2006). An autoencoder receives input data through an encoder that encode the incoming data into a low-dimensional embedding (or encoding), then, it decodes this embedding back to its original dimension. An autoencoder is trained by gradually adjusting the network's weights based on the difference (or error) of the input x to the output x'.

This paper's main contribution is our automatic similarity function $\sigma_A : S \times A \times S \times A \mapsto [0, 1]$, where A is the action-space and S is the state-space of a certain domain, that calculates the similarity of action-state pairs by comparing the distance between their embeddings. Our automatic function transforms the $\langle \tilde{a}, \tilde{s} \rangle$ pairs into a simplified representation through an autoencoder and then calculates the distance between these pairs using a distance measure. This function is implemented in a modified SASS agent, replacing the human-designed similarity function.

Our automated solution has the benefit of being domain independent and its performance does not rely on a designer's input. In the following section, we will detail how we implemented our solution and tested it in the Super Mario Bros domain.

Implementation

In order to implement the method proposed in this paper, we divided the project in two parts: implementing and training the autoencoder, and implementing an SASS agent that utilizes our automatic similarity method in the Mario AI Bros domain. Our implementation is based on the Java solution from the original SASS paper (Rosenfeld, Taylor, and Kraus 2017), which is based on the Mario AI Competition implementation (Togelius et al. 2013). In this section, we will describe in detail the Super Mario Bros domain, our autoencoder configuration and training and our agent's implementation.

Super Mario Bros domain

The Super Mario Bros domain is a custom, fan-made, version of the popular video-game Super Mario Bros, implemented in Java for the Mario AI Competition. In this domain, agents are able to play as Mario on randomly generated levels of Super Mario Bros. Agents are rewarded by how well they play each of the generated levels: how many coins they collect, number of enemies defeated, mushrooms obtained, and their level completion time. We will focus on the games action-state pairs and their representation, as rewards and reward shaping are out of the scope of this Paper.

The game state is represented through a vector with 10 values that represent information regarding the agent's surroundings and its own internal state. The state vector contains the following information:

1. Can Mario jump?
2. Is Mario on ground?
3. Is Mario able to shoot?
4. Mario's current direction.
5. Enemies close to Mario on 8 directions.
6. Enemies mid-range to Mario on 8 directions.
7. Enemies far from Mario on 8 directions.
8. Obstacles in front of Mario.
9. Closest enemy position in x.
10. Closest enemy position in y.

Figure 1 shows a frame from the Mario AI Competition game simulator and the grid with possible values for elements in the state vector that describe the agent's surroundings. Mario's actions are also represented as an integer with 11 possible values that represent input directions, jump and run buttons, and their combined activation.

Autoencoder Implementation

In order to compare the states from the Super Mario Bros domain, we implemented an autoencoder on Python using the Keras machine learning framework

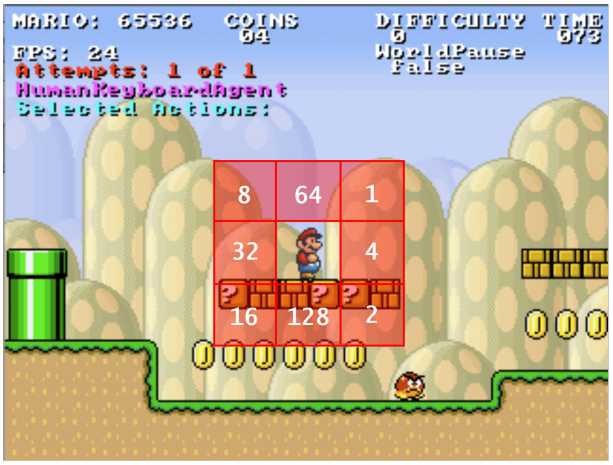

Figure 1: Mario AI Competiton game simulator with an example of Mario position scheme. States 4 - 7 use this scheme with different sizes in order to represent Mario's surroundings

(Chollet and others 2015). Figure 2 shows our autoencoder's configuration: the domain's state vectors are reduced to embeddings of 2 dimensions, greatly reducing the state's dimensionality.

We trained our autoencoder for 400 epochs with 2 million state data collected from a regular Q-Learning agent playing in the Super Mario Bros domain. Our autoencoder reached a loss value close to 0.1 and we find that this loss is acceptable considering its compression factor. We also experimented a configuration with embeddings of 3 dimensions, but the loss value was also close to 0.1.

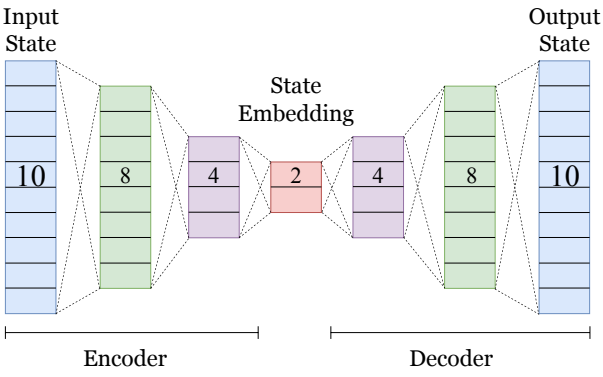

Figure 2: Our autoencoder structure.

Agent Implementation

We implemented an agent based on the SASS implementation, but it uses our own action-state similarity function σ_A replacing the SASS human-designed one σ. Our similarity method works by comparing the agent's last $\langle a, s\rangle$ pair embedding $E(a, s)$ with other $\langle \tilde{a}, \tilde{s}\rangle$ pair embedding $E(\tilde{a}, \tilde{s})$. Embeddings $E(a, s)$ and $E(\tilde{a}, \tilde{s})$ are calculated by calling our trained autoencoder model using the Java deep learning library DL4J (Deeplearning4j 2018).

Our similarity function σ_A was implemented considering 3 distance measures: Euclidian distance, Manhattan distance, and Cosine Similarity. Calculated distances were scaled by a threshold ϵ, allowing for control of the distance considered between different action-state pairs. Similarity σ_A is calculated using Equation 2, where DISTANCE is the distance measure function. The first part of the equation, the subtraction, is not taken into account when using the cosine similarity as DISTANCE function.

$$\sigma_A = 1 - \text{MIN}(\text{DISTANCE}(E(a, s), E(\tilde{a}, \tilde{s})) \cdot \frac{1}{\epsilon}, 1) \quad (2)$$

Our agent utilizes an action-state history buffer in order to select $\langle \tilde{a}, \tilde{s}\rangle$ pairs for comparison with the current $\langle a, s\rangle$ pair. Each time the Q-Value of a pair $\langle a, s\rangle$ is calculated and stored in the Q-Table, extra updates are performed for states in the history buffer with $\sigma_A(a, s, \tilde{a}, \tilde{s}) > 0$. After these extra updates, $\langle a, s\rangle$ is stored in the history buffer. The history buffer have a configurable storing capacity C_b in order to allow a control of how many $\langle \tilde{a}, \tilde{s}\rangle$ pairs are considered for extra updates.

Experiments

We tested different agent configurations in an easy setting of the Super Mario Bros domain. In each test, we measured the mean reward obtained by the agent for a period of 200.000 episodes. We carried experiments with the 3 distance measures for σ_A: Euclidean distance, Manhattan distance, and Cosine Similarity; And with different history buffer capacities C_b. For all tests analyzed in this section, the distance threshold ϵ was fixated at 1, since our initial experimentation showed that higher values can lead to unstable training. Our objective was to find what configuration is best suited for this environment. Finally, we compared our agents results with those obtained by a Q-Learning agent. Results were analyzed using the following criteria:

- Training stability: how steadily the mean rewards grow.
- Training speed: how fast the mean rewards grow.
- Final mean reward value: mean reward obtained at episode 200.000.

Our results for the 3 different distance measures are illustrated on Figure 3, where every agent had a history buffer with capacity $C_b = 50$. In this experiment, the Euclidean distance has a slightly better result when compared to the Manhattan distance, with slightly higher final mean reward and faster training speed. We believe this is due to the fact that both measures yield similar results. The cosine similarity was

clearly the worse performing configuration: mean rewards drops from the first episode and never recover. We believe the reason for this result is that this measure is giving small similarity values for almost every state, forcing extra updates in states that are not similar.

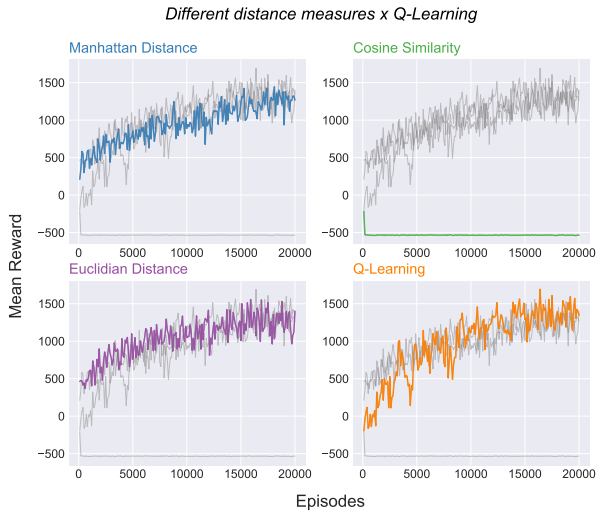

Figure 3: Results of agents implemented with different distance measures for σ_A compared to a Q-Learning agent.

After this first experiment, we tested how different history buffer capacities affected our agent's performance. We maintained the Euclidian distance as measure for this tests and compared the performance of agents with: $C_b = 25$, $C_b = 50$, and $C_b = 100$. Results are illustrated on Figure 4, where $C_b = 50$ yielded to higher mean rewards overall, while stability and speed were similar between configurations.

Our results show that our best configuration is: σ_A with Euclidian distance and $C_b = 50$. This configuration's performance was very similar to that of the Q-Learning agent. However, our agents have one advantage over the Q-Learning agent: it yields high mean rewards since the first training episodes, while the Q-Learning agent spend some time with lower, and negatives, mean rewards. We believe that this is because our agents effectively *spread* their Q-Values to similar states, accelerating the first training episodes. However, we also believe that our similarity function is not optimal for this domain, and the performance of our agents were limited by sub-optimal states with high Q-Values, resulted from wrong similarity estimations.

Conclusion

In this study, we implemented a novel reinforcement agent that combines the training acceleration of the SASS agent with a method for finding action-state similarities automatically and analyzed its performance in a popular and challenging domain: the Super Mario

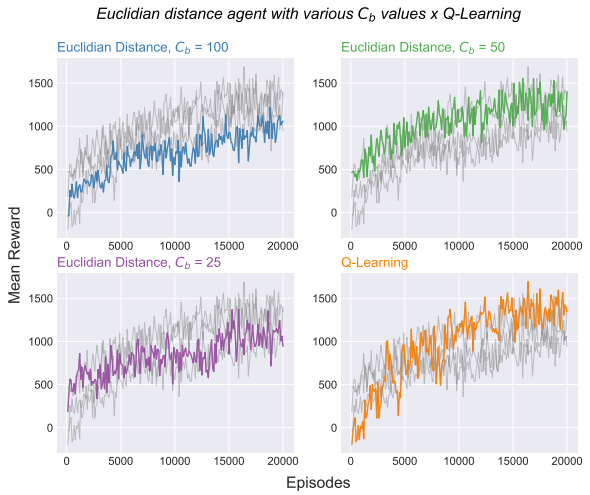

Figure 4: Results of agents using Euclidian distance as distance measure with different history buffer capacities compared to a Q-Learning agent.

Bros game. We demonstrated that, with the right configuration, our agent has a performance comparable to that of the Q-Learning agent, with the benefit of having faster training at the first training episodes.

Nevertheless, we also found that our similarity function is not optimal for this domain. We believe that this is due to the domain's state representation: it may not be the optimal one for the Super Mario Bros game. We intend to investigate our solution's performance in a domain that represent states via images, thus being less prune to design limitations. We also intend to investigate why the cosine similarity measure performed so poorly in our experiments

Finally, our results show that our automatic similarity function is a very interesting upgrade over the regular SASS agent with good results for a domain-independent solution. We believe that this approach should be tested in other domains, and intend to do this in future work.

References

Aggarwal, C. C. 2005. On k-anonymity and the curse of dimensionality. In *Proceedings of the 31st international conference on Very large data bases*, 901–909. VLDB Endowment.

Chollet, F., et al. 2015. Keras. https://keras.io.

Deeplearning4j. 2018. Deeplearning4j: Open-source distributed deep learning for the jvm at http://deeplearning4j.org.

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *science* 313(5786):504–507.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.;

Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, 278–287.

Nikol'skii, S. M. 2012. *Approximation of functions of several variables and imbedding theorems*, volume 205. Springer Science & Business Media.

Ribeiro, C. H. 1995. Attentional mechanisms as a strategy for generalisation in the q-learning algorithm. In *Proceedings of ICANN*, volume 95, 455–460.

Riedmiller, M.; Gabel, T.; Hafner, R.; and Lange, S. 2009. Reinforcement learning for robot soccer. *Autonomous Robots* 27(1):55–73.

Rosenfeld, A.; Taylor, M. E.; and Kraus, S. 2017. Speeding up tabular reinforcement learning using state-action similarities. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, 1722–1724. International Foundation for Autonomous Agents and Multiagent Systems.

Sutton, R. S.; Barto, A. G.; Bach, F.; et al. 1998. *Reinforcement learning: An introduction*. MIT press.

Togelius, J.; Shaker, N.; Karakovskiy, S.; and Yannakakis, G. N. 2013. The mario ai championship 2009-2012. *AI Magazine* 34(3):89–92.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.