Towards Playing Montezumas Revenge with Deep Reinforcement Learning

Blake Wulfe
wulfebw@stanford.edu

Abstract—**We analyze the task of learning to play the Atari 2600 game Montezuma's Revenge with an emphasis on the application of hierarchical reinforcement learning methods. This game is particularly challenging due to its sparse reward structure, partial observability, and hierarchy of diverse subtasks. We discuss potential solutions to these first two challenges, and, for the third, propose a simple method - stacking multiple recurrent neural network layers - that we conjecture enables the use of temporal abstraction in deep reinforcement learning models. We evaluate variations of this method on a set of test problems, and present promising results for one model-free, value-based variation that explicitly encodes temporal abstraction in the model.**

I. Introduction

Learning to play Atari 2600 games directly from screen input is a challenging reinforcement learning task. Methods for overcoming challenges presented by this task are well suited for application to real-world problems with high-dimensional state spaces. Real-world problems also frequently exhibit other challenges that are not present in many Atari games - for example, sparse rewards, partial observability, and hierarchical structure. Montezuma's Revenge is one game, however, that exemplifies these challenges. For this reason, this game is an appealing testing ground for reinforcement learning methods intended for complex, real-world application.

Deep reinforcement learning is a class of methods well-suited to application in high dimensional state spaces that has seen significant recent interest. A number of such methods have been proposed to address the three challenges presented by Montezuma's Revenge, but none, at least independently, seem sufficient to tackle this problem. In this paper, we briefly survey potential solutions to the first two problems - sparse rewards and partial observability - and propose a novel method of alleviating the third challenge through the incorporation of temporal abstraction.

A number of traditional hierarchical reinforcement learning methods exist, both in which abstract actions are provided manually (e.g., options [28], HAMQ [20], and MAXQ [6]) or automatically learned (e.g., HEXQ [9] and skill chaining [11]). While it is plausible that these latter methods may be adapted to problems such as Montezuma's Revenge, this has not yet been accomplished. Recent success in playing these games is largely due to advances in neural networks, to which no existing hierarchical methods seem immediately applicable. How can neural networks automatically learn increasingly complex skills? Central to learning skills is the ability to utilize state and temporal abstraction - i.e., the ability to only use the subset of available information that is relevant to a specific subproblem rather than the entirety of available information.

We present a method that we conjecture is capable of such temporal and spatial abstraction. Specifically, we propose the simple extension to Recurrent Q-Networks (RQNs) of adding additional recurrent layers, which we call stacked RQNs (sRQN). We demonstrate the effectiveness of this method on a set of test hierarchical learning tasks.

The rest of this paper is organized as follows:

1) Discussion of Montezuma's Revenge
2) Related Work
3) Background Topics
4) Stacked Recurrent Q-Networks
5) Experiments and Results
6) Discussion
7) Conclusion

II. Montezuma's Revenge

In Montezumas Revenge, the player controls an explorer with the goal of escaping from a maze while gathering as much loot as possible (see figures 1 and 2).

A. Challenges

This game is difficult for three reasons. First, it exhibits sparse rewards. For example, in the starting room of the maze, the explorer must perform a long sequence of complex actions to reach the key and receive any reward signal.

Second, Montezumas Revenge is partially observable when using a fixed number of frames to represent the state. This is because it is possible for the agent to (1) collect a key, (2) use that key to open a door, which removes it from the inventory, and (3) enter into a separate room of the maze. At this point the agent does not know whether it has previously opened a door and is unable to infer this information from the screen (see Appendix A for example sequence). As a result, an optimal solution must overcome partial observability.

Third, Montezumas Revenge consists of a hierarchy of diverse subtasks, each of which must be mastered to play the game successfully. For example, on the timescale of a few frames, the agent must learn to avoid attackers or climb up and down ladders. On the timescale of a few hundred frames, the agent must learn to navigate through a diverse set of individual rooms. On the timescale of thousands of frames, the agent must plan trajectories through the maze that will allow it to access new rooms and ultimately escape. Accomplishing these tasks seems to require some form of temporal abstraction.

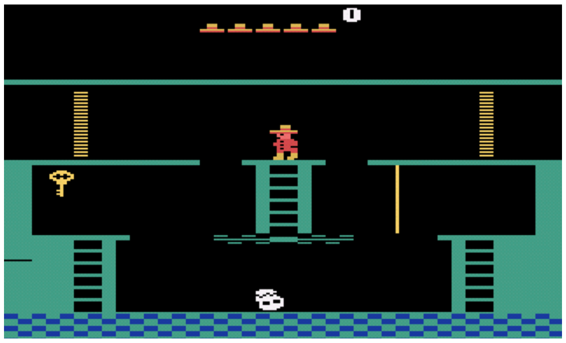

Fig. 1: The first room of the first-level maze in Montezuma's Revenge. The explorer, shown at center image in red, must retrieve the yellow key to open either of the two doors.

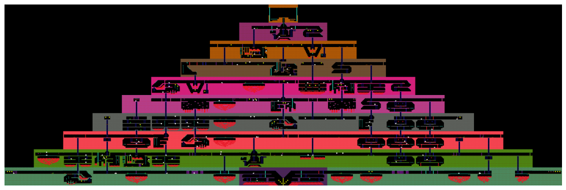

Fig. 2: The full maze of the first level of Montezuma's Revenge. The explorer must navigate from the top room in purple to the bottom-left room in dark green.

III. Related Work

A. Playing Atari 2600 Games

Development of a general video game playing agent specifically for the Atari 2600 was introduced in 2006 by Naddaf [18]. Bellemare et al. [2] formally introduced the Arcade Learning Environment framework and established a set of baselines using SARSA(λ) applied to tile-coded features as well as search-based methods that explore the game through saving and reloading its RAM contents. Offline, planning-based methods relying on this ability have achieved excellent results in playing Atari games [7]; however, the focus in this project is on methods that use information available to human players and that can execute in real-time. Within these constraints, Defazio et al. [5] compared the performance of SARSA(λ), Q(λ), next-step and per-time-step maximizing agents, Actor-Critic methods, and others for Atari game playing, finding that the best performing algorithm differed significantly between games. Minh et al. [17][16] introduced Deep Q-Networks (DQN), a batch Q-learning method that uses a neural network to learn state-action values.

B. Exploration

There are three recent extensions of the DQN that may provide solutions to the problem of exploration in sparse reward environments.

First, Stadie et al. [27] incorporated an autoencoder to learn state representations that were used in conjunction with a separate, learned model to incentivize exploration. This method resembles earlier work in intrinsic motivation [3] and artificial curiosity [24].

Second, Osband et al. introduced the Bootstrapped DQN [19], which samples randomly from a distribution over Q-value functions. This is accomplished by extending a DQN with a set of heads, one of which is chosen per episode to act greedily, thereby allowing for deep exploration as opposed to dithering strategies such as ϵ-greedy. This method as well as predictive models may be paired with Prioritized Experience Replay [22] to further decrease sample complexity in reward-sparse environments.

Third, Minh et al. [15] recently proposed a set of asynchronous deep reinforcement learning methods, one of which - Asynchronous Advantage Actor Critic (A3C) - is the current state-of-the-art in real-time Atari game playing. In this method, agents across multiple CPU threads asynchronously update shared weights. This approach significantly speeds learning (one day of training on a CPU achieves superior results to eight days of training a DQN on a GPU), as well as provides a number of natural extensions for improved exploration.

C. Partial Observability

Two approaches provide avenues for overcoming challenges presented by partial observability in Atari Games.

First, the Deep Recurrent Q-Network (DRQN or RQN) proposed by Hausknecht [8] adds a long short-term memory (LSTM) layer to the network. The RQN deals well with partial observability in the form of flickering game screens, but has not been tested on tasks involving partial observability over longer timescales.

Second, Schmidhuber proposed a method in which a RNN model learns the dynamics of the environment, and then shares connections with a controller [23]. Since the model hidden state must in theory capture all information of the input necessary for predicting future states and rewards, the controller should be able to treat the hidden state as a Markovian input for solving the underlying MDP. This in theory addresses the issue of partial observability, but also, as asserted in the paper, provides a mechanism for performing hierarchical learning. Related approaches have been applied, for example, in [14].

D. *Stacked Recurrent Neural Networks*

The method we propose for enabling temporal abstraction in deep reinforcement learning relies upon stacks of recurrent layers. This is one method of making RNNs deep, in which the outputs of lower layers are input to higher layers. [21][25]. The claimed benefit of this approach is that it captures patterns at differing timescales. While this assertion is not obviously true, there are a number of extensions to stacked RNNs, for example Gated Feedback Recurrent Networks [4] and Clockwork RNNs [12], which more explicitly learn to process input at different timescales.

One method that is particularly explicit in performing temporal abstraction is exemplified by the Hierarchical Neural Autoencoder [13], which passes the output of lower RNN layers to higher layers only at fixed intervals. This method can be viewed as a recursive network [26] altered to include recurrent connections at each layer.

IV. Background

A. *Reinforcement Learning*

We consider the standard reinforcement learning setting in which an agent interacts with an environment, taking action a in state s and receiving reward r and next state $s\prime$. The goal of the agent is to maximize the expected discounted return for each state s. This value for a given policy π is given by the state-action value $Q^{\pi}(s,a) = \mathbf{E}[R_t|s_t = s, a]$. The optimal state-action value $Q^{*}(s,a) = max_{\pi}Q^{\pi}(s,a)$ gives the best possible value achievable by any policy. The value of following a policy π starting at a state s is defined $V^{\pi} = \mathbf{E}[R_t|s_t = s]$, with equivalent optimal version as before.

B. *Q-Learning*

Q-learning is a model-free, value-based method that iteratively improves an estimate of the optimal state-action value function. It does this by minimizing the expected squared error between a bootstrapped estimate of the value of the next state plus reward and an estimate of the value of the current state: $min_{\theta}\mathbf{E}[(r_t + \gamma max_{a\prime}Q(s\prime, a\prime; \theta_{t-1}) - Q(s, a; \theta_t))^2]$.

C. *Deep Q-Networks*

Deep Q-Networks use neural networks to approximate the state-action value function. They do this in the discrete action case by using a network architecture that maps an input state representation to a series of outputs, each of which correspond to the estimated Q-value of taking a specific action in the current state.

Two techniques are generally used with DQNs to stabilize learning. The first is the use of a replay memory, which samples batches of previous experience from a collected dataset D to update network parameters. This speeds learning by reducing correlations between input samples. The second method is the use of a target network, in which a separate set of network parameters θ^{-} are maintained and used to produce target values in the Q-learning update. This eliminates a feedback effect where a parameter update can result in increasingly large updates producing divergent behavior.

DQNs thus minimize a similar object to that of Q-learning: $min_{\theta}\mathbf{E}_{(s,a,r,s\prime)\sim D}[(r_t + \gamma max_{a\prime}Q(s\prime, a\prime; \theta^{-}) - Q(s, a; \theta))^2]$

D. *Recurrent Q-Networks*

Recurrent Q-Networks (RQN) have a similar structure to DQNs but include a recurrent layer somewhere in the architecture, generally as the second to last layer of the network. This allows the model to in theory remember information from all previous states. Most related work and all recurrent models in this paper use Long short-term memory (LSTM) RNNs [10].

V. Stacked Recurrent Q-Networks

We propose to incorporate the temporal abstraction capabilities of stacked RNNs into the RQN. We conjecture that this model, the stacked Recurrent Q-Network (sRQN), will be able to capture patterns at different timescales, thereby enabling it to learn to perform tasks at different level of temporal abstraction. In Montezuma's Revenge, for example, the lowest layer of the network might learn to avoid obstacles and climb ladders, the layer above to navigate individual rooms, and a higher layer still to learn to escape a maze as a whole.

A. *Layer Connectivity*

Which layers should connect with the final, state-action value producing layer? We consider two options - one in which the final output of all recurrent layers is input into the final feed-forward layer (sRQN-merge), and a second in which only the output of the final recurrent layer is input to the feed-forward layer (sRQN).

How should the different layers be connected across timesteps? While it is possible to connect earlier timesteps of higher layers back to lower layers, we only consider here the relatively simpler options of full connectivity and partial connectivity across timesteps. In this latter option, lower-layer outputs are input to higher layers only at fixed intervals similarly to the hierarchical neural autoencoder [13]. See figure 3 for depictions of the different architectures.

We refer to this partially connected network as a hierarchical stacked recurrent Q-Network (hsRQN). There are two motivations for this architecture. First, it is more efficient that the fully connected version. This is a concern because training RNNs with a large number of unrolled timesteps (as we expect is necessary for learning to play Montezuma's Revenge) can have a high computational cost [8].

Second, by connecting layers only at fixed intervals, we explicitly encode temporal abstraction in the network. One

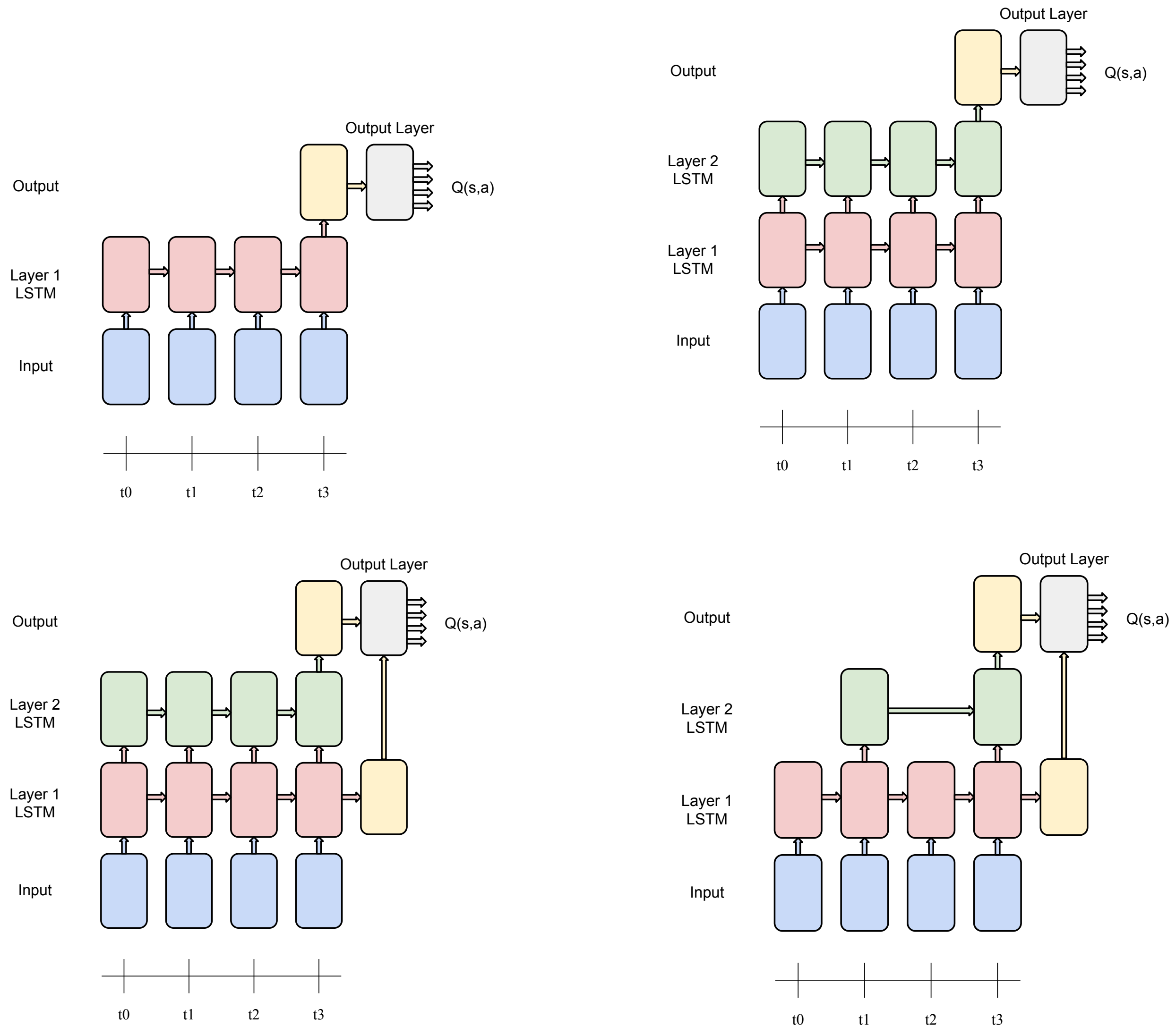

Fig. 3: The four architectures evaluated. Left Top: RQN Left Bottom: The sRQN-merge. Right Top: The sRQN-merge, in which each layers outputs are concatenated before being passed to the output layer. Right Bottom: The hierarchical sRQN, in which hidden states are passed to higher layers only at set intervals.

argument against this approach is that LSTMs should be able to learn these patterns automatically, so by only connecting layers at intervals we are enforcing a constraint that may limit the network. This is a valid concern and one we try to address empirically.

VI. Experiments

A. Experimental Setup

1) Four Room and Maze Domains: In order to assess the strengths of these different models, we evaluate them on a set of test problems. We consider the Four room domain presented originally in [28], using a formulation similar to that of [29] shown in figure 4. This is a classic hierarchical learning task, which can also easily be scaled to make more difficult or be adapted to the partially observable case.

The MDP is a square maze consisting of four rooms. The agent starts in the bottom left corner and the goal is to reach the top right corner. Doing so gives a reward of 1, and each step incurs a cost of -0.01. The agent may move east, west, north, or south. In each case, the agent is deterministically transitioned to the next state unless it runs into a wall, in which case it remains in the same location. Each episode is capped at $(2 * room - side - length * number - of - rooms)^2$ steps, which is selected based on a property of random walks.

2) Evaluation Criteria: To evaluate the performance of different algorithms on the maze MDP, we consider two metrics. First is the average episodic reward, which has the advantage

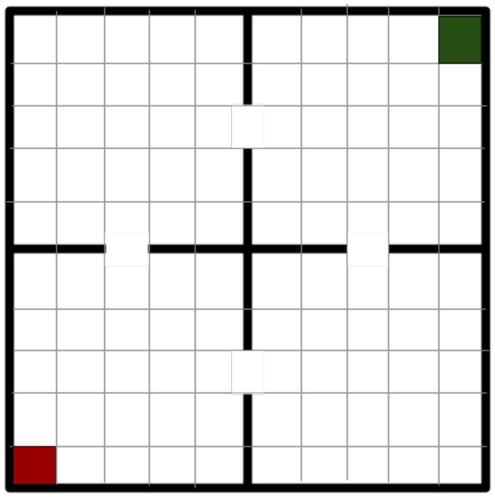

Fig. 4: The four room domain. Agent starts in bottom left with a goal state in upper right.

of being easy to gather and interpret, but the disadvantage that it does not necessarily convey how well an agent has learned the values of states.

The second metric is how quickly the estimated state-value of the initial state in the maze converges to its true value. This metric was used to assess a recently proposed hierarchical learning method [1], and has two advantages. First, it conveys how quickly value propagates through the state space, which may demonstrate the benefit of hierarchical approaches. Second, it relies on the agent learning the absolute value of a state correctly rather than just its relative value. The primary disadvantage of this metric is that it is highly sensitive to hyperparameter settings.

3) State Representation: What representation of the state should be used as input to the models? We empirically evaluated four options:

1) coordinates
2) one-hot tabular
3) one-hot row and column
4) one-hot row, column, and room

Figure 5 shows the results of training a DQN for fifty, single-episode epochs with the different state representations. From these results, we concluded that the row and column and the row, column, and room representations were both acceptable options. The tabular option likely performed poorly as a result of too little training time.

4) Number of Training Timesteps: For how many backpropagation through time (BPTT) steps should recurrent models be trained? Figure 6 shows the results of evaluating different number of rollout timesteps. These results indicate that values within a certain range perform comparably, but that once a high enough number of steps are unrolled, learning becomes difficult. This may be the case because for value to propagate to states early in the maze, it must do so through more layers in the network.

B. Results

Each network architecture was trained with a variety randomly selected hyperparameter values (learning rate, frozen target period, replay memory capacity, BPTT timesteps, state representation, exploration probability, number of hidden units) for a relatively low number of episodes. From this we decided upon a set of reasonable hyperparameters to use across all models. We then ran the four experiments below, simulating each model twice for between 400 to 500 episodes (constant within a given experiment) and taking the better run as reflected by reward and state-value graphs.

1) Four Room Domain: This first experiment assessed the performance of each model on the four room MDP. The hsRQN and RQN achieved similar reward and start-state value curves. As can be seen in figure 7 The hsRQN estimate of the start state value increased with the fewest episodes and ultimately slightly overestimated the correct value of the state. The RQN exhibits a similar learning curve though ultimately underestimated the value. The sRQN-merge model performed better than the stacked network without the merge, but both performed relatively worse than the hsRQN and RQN. To further visualize what the network learns, we plot heatmaps of the predicted value of states by the hsRQN after different episodes in the experiment in figure 8.

2) Single Room Maze: In our second experiment, we removed the walls from the maze thereby eliminating problem subtasks. We expected that this would diminish the advantage of the hsRQN over the RQN because this learning in this task should not benefit significantly from temporal abstraction. Performance of the models as measured by average episode reward is quite similar. The RQN propagates value back to the start state most quickly though overshoots the true value. The hsRQN converges to closest to the true value, again followed by the sRQN-merge and sRQN respectively. We were surprised that the hsRQN seemed to perform best at this task. This may be a result of the small sample size of runs for each model,

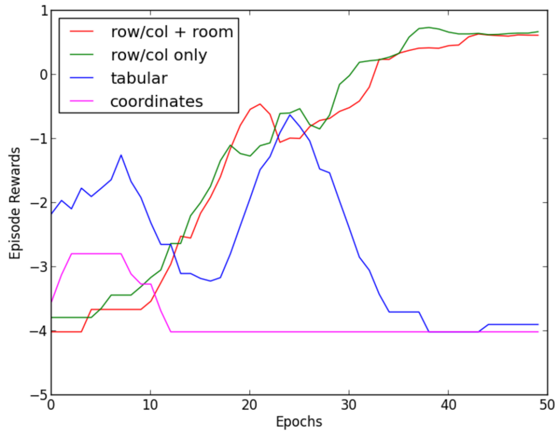

Fig. 5: Comparison of DQN performance with different state representations

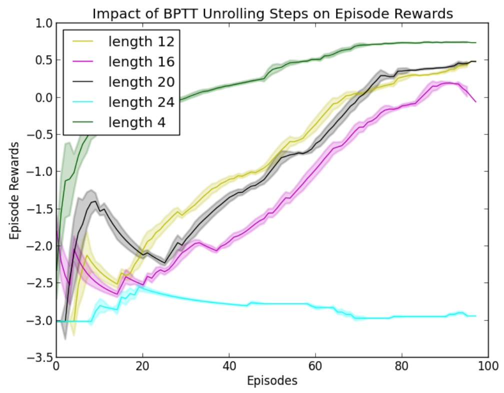

Fig. 6: Comparison of RQN performance with different numbers of BPTT timesteps

or the larger number of parameters in the hsRQN

3) Larger Mazes: By adding additional rooms and therefore subtasks, we conjectured that the stacked networks would gain a larger advantage over the RQN. We tested the models on a nine room maze with each room having a side length of three. In this experiment, the RQN and hsRQN again performed comparably, both converging to relatively close start state values. This similarity in performance may be due to the fact that this maze is smaller than the previous two, and therefore less effectively reflects differences between the models.

4) Partially Observable Mazes: The previous experiments are analogous to showing a player the entire maze in Montezumas Revenge. How do these models perform if we instead only provide the agents state in the current room? This last experiment attempted to answer that question. In this case, the agent is only provided with the one-hot row and column representation of its current room. As a result, the start-state value is not as easily interpreted. All four models have difficulty learning correct state-values, though the trend of values indicates that training for more iterations may have allowed the models to converge to the true values. The hsRQN again seems to perform best, though towards the end of the simulation is surpassed in performance by the RQN.

VII. Discussion

Our experimental results are inconclusive as to the ability of the stacked recurrent models to use temporal abstraction to improve learning. The hsRQN generally performed better than the RQN, whereas the other stacked models performed worse. These results may be due to the low number of sample runs of each network, or they may reflect significant underlying traits of the different models. We believe it is likely that such differences would be more clearly reflected in the performance of the models on a more complex task, for example in application to playing Montezuma's Revenge.

VIII. Conclusion

In this paper we proposed the stacked recurrent Q-Network, a novel model we conjecture is capable of taking advantage of temporal abstraction to improve learning ability. We assessed

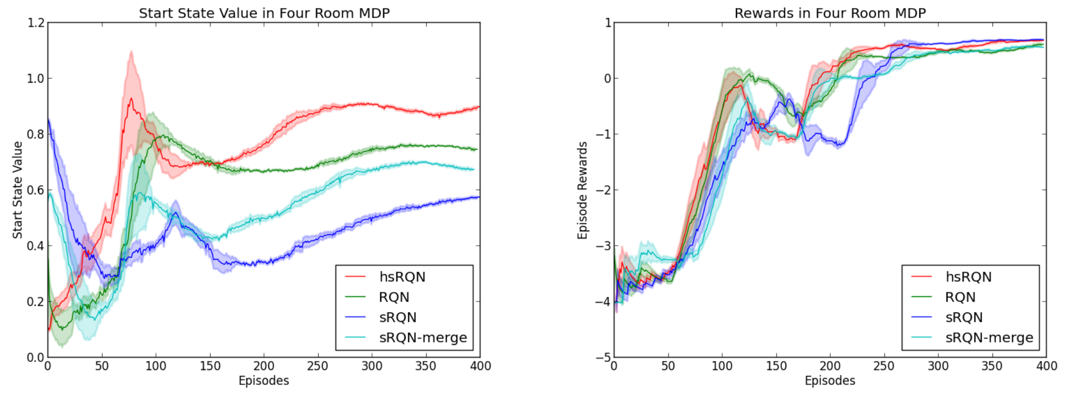

Fig. 7: Value and reward graphs for the four room MDP. The correct state value is 0.83

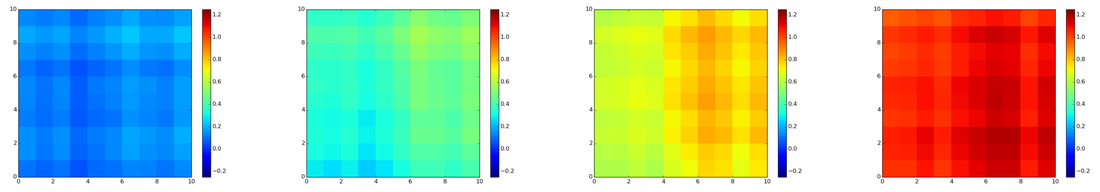

Fig. 8: Estimated state values as learning progresses.

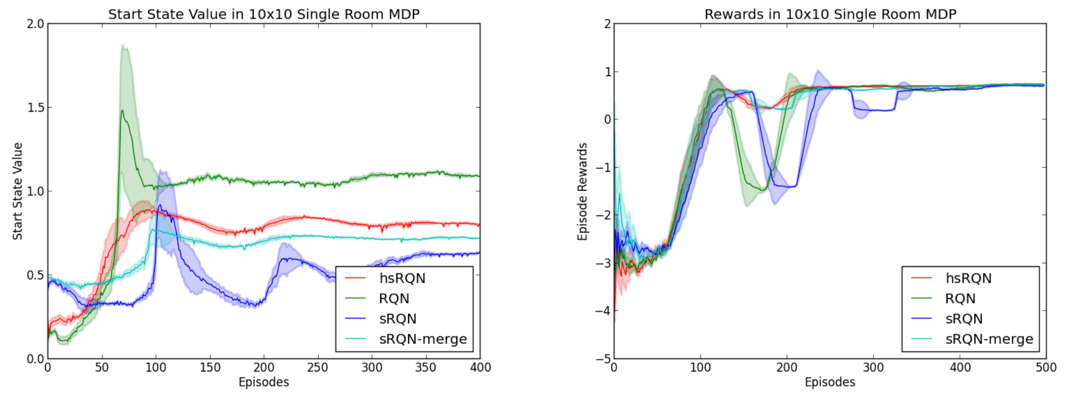

Fig. 9: Value and reward graphs for the single room maze.

variations of this model on a set of test problems. While these results indicate the proposed model shows promise, we are unable to say conclusively whether it truly provides much benefit over the relatively simpler RQN. We believe this can be determined by testing the model on more challenging tasks.

We also discussed the Atari 2600 game Montezumas Revenge, the main challenges presented by this game, and some of the potential solutions to these challenges. We consider Montezumas Revenge to be a particularly well suited testing ground for not only hierarchical methods, but also for those overcoming issues of sparse rewards and partial observability.

We next intend to compare the performance of the hsRQN and RQN in learning to play Montezuma's Revenge, as well as to consider certain extensions to the proposed model such as training on increasingly long sequences and adding additional layers during training.

Appendix A
Partial Observability in Montezumas Revenge

We present an example of partial observability in Montezumas Revenge where the agent must remember the full history to act optimally.

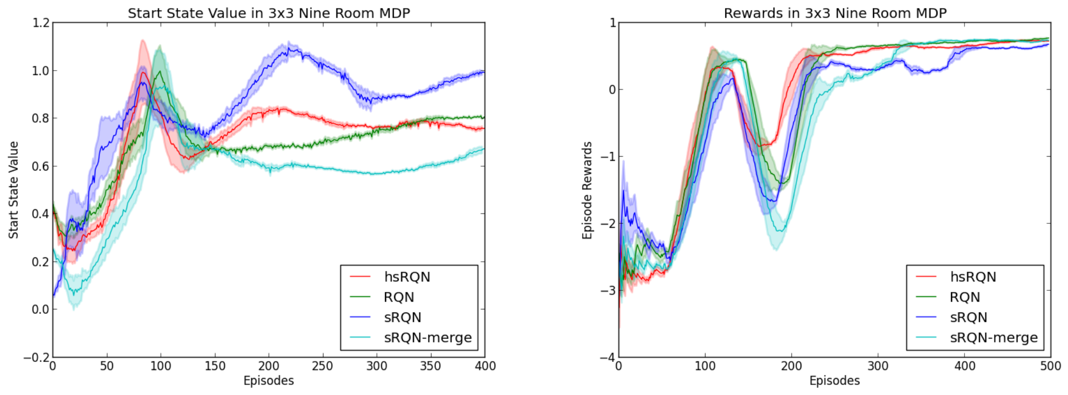

Fig. 10: Value and reward graphs for the nine room maze.

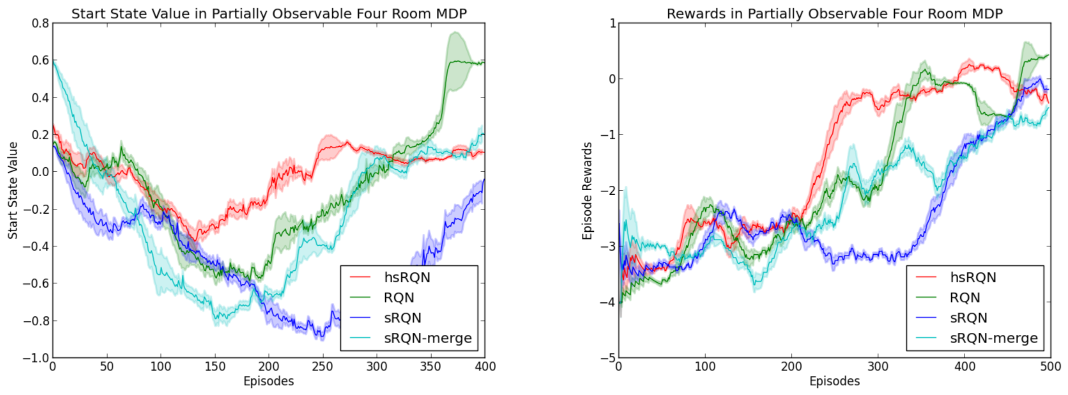

Fig. 11: Value and reward graphs for the partially observable case

Appendix B
Technical Details

All neural network models were developed using Theano [24] and Lasagne [25]. All code used for the project is available at https://github.com/wulfebw/hierarchical_rl. Optimization We use stochastic gradient descent with nesterov momentum to perform optimization. We found that learning progressed more smoothly than with adaptive learning rate methods such as Adam. Adam might have performed comparably with different learning rate, beta1, and beta2 settings, but it also seems that adapative learning rates may have disadvantages with nonstationary input distributions

We tested the models with a range of hidden layer sizes. In general it did not significantly impact learning so we used very few hidden units - between four and twenty for most tests.

We found that varying learning rate and the frozen target period had a dramatic effect on learning, particularly on how quickly value was propagated to the start state. We used a learning rate of 0.01 and frozen interval of 100 updates for training the networks in the above experiments. These values occasionally resulted in unstable learning, but seemed to produce the best results as measured by value propagation.

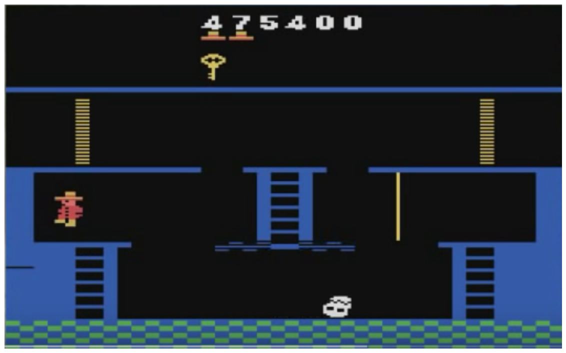

Fig. 12: The explorer acquires the key in the first room of the maze.

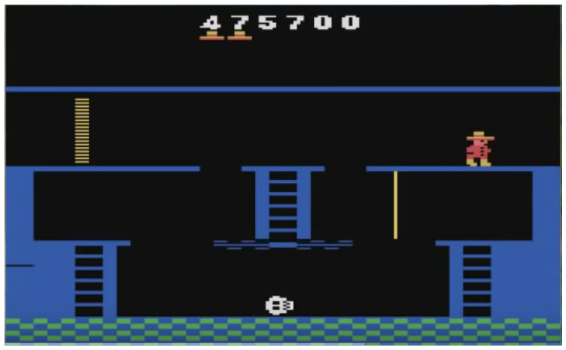

Fig. 13: The explorer opens a door with the key, losing it in the process and moves towards the next room. Note the second door on the left side of the room.

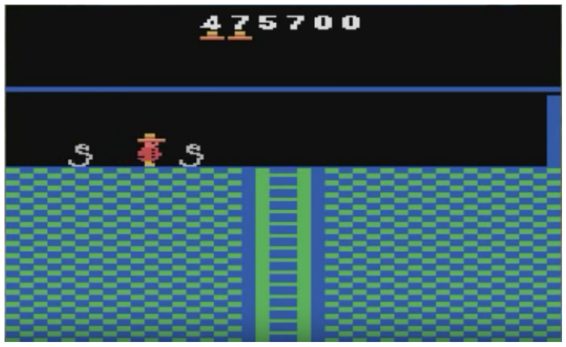

Fig. 14: The explorer enters into the second room. He no longer has the key in his inventory, so has no way of knowing whether he has just left the first room or whether he has already retrieved an additional key and unlocked the second door in the first room. The explorer may spend an arbitrary amount of time in this second room, and therefore may be required to maintain an arbitrary number of frames to know whether the second door is unlocked.

References

[1] Pierre-Luc Bacon and Doina Precup. “The option-critic architecture”. In: *NIPS Deep Reinforcement Learning Workshop*. 2015.

[2] Marc G Bellemare et al. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* (2012).

[3] Nuttapong Chentanez, Andrew G Barto, and Satinder P Singh. “Intrinsically motivated reinforcement learning”. In: *Advances in neural information processing systems*. 2004, pp. 1281–1288.

[4] Junyoung Chung et al. “Gated feedback recurrent neural networks”. In: *arXiv preprint arXiv:1502.02367* (2015).

[5] Aaron Defazio and Thore Graepel. “A comparison of learning algorithms on the arcade learning environment”. In: *arXiv preprint arXiv:1410.8620* (2014).

[6] Thomas G Dietterich. “Hierarchical reinforcement learning with the MAXQ value function decomposition”. In: *J. Artif. Intell. Res.(JAIR)* 13 (2000), pp. 227–303.

[7] Xiaoxiao Guo et al. “Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 3338–3346.

[8] Matthew Hausknecht and Peter Stone. “Deep recurrent q-learning for partially observable mdps”. In: *arXiv preprint arXiv:1507.06527* (2015).

[9] Bernhard Hengst. “Discovering hierarchy in reinforcement learning with HEXQ”. In:

[10] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[11] George Konidaris and Andre S Barreto. “Skill discovery in continuous reinforcement learning domains using skill chaining”. In: *Advances in Neural Information Processing Systems*. 2009, pp. 1015–1023.

[12] Jan Koutnik et al. “A clockwork rnn”. In: *arXiv preprint arXiv:1402.3511* (2014).

[13] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. “A hierarchical neural autoencoder for paragraphs and documents”. In: *arXiv preprint arXiv:1506.01057* (2015).

[14] Xiujun Li et al. “Recurrent Reinforcement Learning: A Hybrid Approach”. In: *arXiv preprint arXiv:1509.03044* (2015).

[15] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). URL: http://arxiv.org/abs/1602.01783.

[16] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[17] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[18] Yavar Naddaf et al. “Game-independent ai agents for playing atari 2600 console games”. PhD thesis. University of Alberta, 2010.

[19] Ian Osband et al. “Deep Exploration via Bootstrapped DQN”. In: *CoRR* abs/1602.04621 (2016). URL: http://arxiv.org/abs/1602.04621.

[20] Ronald Parr and Stuart Russell. “Reinforcement learning with hierarchies of machines”. In: ().

[21] Razvan Pascanu et al. “How to construct deep recurrent neural networks”. In: *arXiv preprint arXiv:1312.6026* (2013).

[22] Tom Schaul et al. “Prioritized Experience Replay”. In: *arXiv preprint arXiv:1511.05952* (2015).

[23] Juergen Schmidhuber. “On Learning to Think: Algorithmic Information Theory for Novel Combinations of Reinforcement Learning Controllers and Recurrent Neural World Models”. In: *arXiv preprint arXiv:1511.09249* (2015).

[24] Jürgen Schmidhuber. “Adaptive confidence and adaptive curiosity”. In: *Institut fur Informatik, Technische Universitat Munchen, Arcisstr. 21, 800 Munchen 2*. Citeseer. 1991.

[25] Jürgen Schmidhuber. “Learning complex, extended sequences using the principle of history compression”. In: *Neural Computation* 4.2 (1992), pp. 234–242.

[26] Richard Socher et al. “Parsing natural scenes and natural language with recursive neural networks”. In: *Proceed-*

ings of the 28th international conference on machine learning (ICML-11). 2011, pp. 129–136.

[27] Bradly C Stadie, Sergey Levine, and Pieter Abbeel. “Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models”. In: *arXiv preprint arXiv:1507.00814* (2015).

[28] Richard S Sutton, Doina Precup, and Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* 112.1 (1999), pp. 181–211.

[29] M Weiring and M Otterlo. *Reinforcement learning: State-of-the-Art, Vol. 12 of Adaptaion, Learning and Optimization*. 2012.