# Effectiveness of Recurrent Network for Partially-Observable MDPs

**Kyu-Young Kim**
Stanford University
kykim153@stanford.edu

## Abstract

Deep learning models have been successfully employed as effective function approximators in many reinforcement learning (RL) algorithms. In Deep Q-Networks (DQNs), for instance, deep learning models are used to estimate the action-value function of the Q-learning algorithm. However, many of the Deep RL algorithms rely on the assumption that full state information is provided at each decision point. Real world environments, in contrast, often provide incomplete state information that the agent somehow needs to account for. In this work, we explore the effectiveness of recurrent neural network (RNN), a popular sequence learning model, in dealing with such partial-observability in the context of DQN. Furthermore, we study the effects that different RL techniques such as varying exploration strategies have on convergence behavior.

## 1 Introduction

Recent developments in deep learning have advanced the state-of-the-art in a wide range of tasks such as speech recognition, object detection, and language understanding [6]. With its ability to automatically extract high-level features from raw input data, deep learning has allowed tackling many problems in AI that were previously considered intractable. Application of deep learning techniques to RL led to the emergence of the new field called deep reinforcement learning (DRL) that demonstrated its ability to train RL agents even on highly complex environments.

Deep Q-Network is a notable application of deep learning to RL. It was the first RL algorithm that demonstrated that agents trained on raw pixel inputs can learn human-level control policies [7]. Using a deep neural network as a function approximator, the agent compactly represents the action-value function (Q-function) and based on the prediction selects the action with the maximal expected reward at each step. The DQN agent was able to outperform a human expert player on several of the Atari 2600 games.

Despite the remarkable success, the original DQN is limited in that the agent only learns a mapping of a fixed number of fully-observable states to Q-value. Hence, the DQN agent would struggle in environments that require it to remember more distant past than it encountered during training. The environments in these cases become partially-observable Markov decision processes (POMDP) and pose greater challenge for the RL agent [3]. Real world environments often only provide such partially-observable state information, and the agent somehow needs to account for this limitation.

In this work, we investigate the capability of Deep Recurrent Q-Network (DRQN), a variant of DQN that employs an RNN model for learning temporal dependency, in handling partially-observable environment state. We analyze the performance of DRQN agents on environments with limited state information and compare them against DQN agents trained on the same environments with full state information. We also examine several common RL techniques such as various exploration strategies in depth to understand the effects they have on convergence behavior.

## 2 Background

In reinforcement learning, an *agent* interacts with an *environment* to learn to act in a way that maximizes the expected return. At a given time step $t$, an agent chooses an *action* $a_t$ from a state $s_t$ and transitions to a new state $s_{t+1}$. After each state transition, the environment provides a scalar *reward* $r_{t+1}$ to the agent as feedback. The discounted return defined as $R = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ with $\gamma \in [0, 1)$ represents discounted sum of future rewards the agent is to accumulate. The objective of the agent is to learn an optimal policy $\pi^*$ through its interaction with the environment in order to achieve the maximum expected discounted return [1]. That is:

$$\pi^* = \text{argmax}_\pi \mathbb{E}[R \mid \pi]$$

With the assumption that the next state is conditionally independent of the past given the current state and the action (called Markov property), the RL problem can be formalized as a Markov decision process (MDP). MDPs are defined by a set of states $S$, a set of actions $A$, a transition function $T(s' \mid s, a)$ representing the probability of transitioning to $s'$ from $s$ after executing $a$, and a reward function $R(s, a)$ representing the immediate reward given for executing $a$ from $s$. In many RL algorithms, the states are assumed to be *fully observable* to the agent. In real world applications, however, the states are often only partially observable, and the agent has to learn to account for this limited information. This generalization of MDPs are called partially observable MDPs or POMDPs.

Given the problem definition, there are some common challenges in RL that are worth noting. First, the agent must deal with temporal dependencies among a potentially long sequence of actions. This is especially important in environments with sparse rewards, where the agent needs to take a long sequence of actions before receiving a reward signal. This problem is often referred to as the credit assignment problem. Second, the observations that the agent receives depend on the actions it takes. Depending on the agent's strategy, it receives different observations which are then used to improve its policy. Hence, choice of an appropriate exploration strategy is critical for learning a good policy. Third, the agent needs to generalize to unforeseen states. In case the state space is discrete and small, the agent may be able to explore all possible states during its interaction with the environment. If the state space is large or even continuous, this would not be possible, and the agent must learn to generalize from its limited experience.

One popular class of methods for solving RL problems involves estimating the value function $V^\pi(s)$ representing the expected utility of being in state $s$ and following policy $\pi$ in subsequent steps. Given an optimal value function $V^*(s) = \max_\pi V^\pi(s)$ for all $s$, the corresponding optimal policy can be derived by selecting the action $a$ that maximizes the quantity $R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) V^*(s')$ for each $s$. Since the transition function is not available in RL, we instead use the action-value function $Q^\pi(s, a)$, which is the expected utility of executing action $a$ from state $s$ and subsequently following policy $\pi$. Given the definition, we have the following recursive relation called Bellman equation:

$$Q(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_t + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})][1]$$

This serves as the basis of Q-learning algorithm that iteratively improves the Q-function with the observed next state and reward as:

$$Q_{i+1}(s_t, a_t) \leftarrow Q_i(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_i(s_{t+1}, a) - Q_i(s_t, a_t))$$

where $\alpha$ is the learning rate. The algorithm converges to an optimal Q-function: $Q_i \to Q^*$ as $i \to \infty$.

In practice, however, this approach is impractical as Q-function only has estimates for the states and actions explicitly encountered during training. Instead, a function approximator parameterized by $\theta$ is often used to estimate the Q-function: $Q(s, a; \theta) \approx Q^*(s, a)$. We can use a simple linear function as the approximator or a more complex non-linear function. Using deep neural networks as function approximator leads to the Deep Q-networks (DQN) [7], which we explore in depth in this work.

## 3 Approach

DQN is considered as one of the early successful applications of deep learning to RL, demonstrating use of deep neural networks to train agents on environments with high-dimensional state spaces. One limitation of the original DQN algorithm is that it is only capable of handling a fixed-number of states that it encountered at training time. If the agent is required to remember more distant past, the

environment would appear non-Markovian, and the agent would likely to struggle [3]. We consider combining RNN models like Long Short Term Memory [4] that are widely used for sequence-learning tasks with DQN algorithm to handle such partial-observability. We found some of the techniques presented in [7] that helped to stabilize DQN training to be critical even for recurrent models.

### 3.1 Exploration

In the RL setup, the transition function and the reward model are not available. Given this limitation, it is critical for the agent to choose actions in a way that balances between utilizing what it has learned about the environment thus far and exploring new states that may lead to better outcomes. Hence, neither greedy nor random exploration strategy would likely to work very well.

One common strategy used is called $\epsilon$-greedy. In this approach, the agent greedily chooses the best action $a = \max_a Q(s, a; \theta)$ with probability $1 - \epsilon$ and a random action with probability $\epsilon$. This is the exploration strategy used for experiments in [7]. Borrowing the idea from learning rate schedule, we can also use different values of $\epsilon$ depending on the training progress. For instance, we can use a relatively large value of $\epsilon$ at the start of training to encourage more exploration and progressively lower the value to increasingly encourage exploitation in later stage of training.

Another strategy which we explore in this work is called Boltzmann sampling. In this strategy, we sample among the possible actions with probabilities proportional to the logits computed by the function approximator network. Rather than randomly selecting among non-optimal actions as done in $\epsilon$-greedy, we take the output Q-value vector from the neural network and sample according to the logits. The temperature parameter can be used to better balance exploration and exploitation. This hyperparameter, of course, can also be tuned as training progresses.

### 3.2 Experience Replay

Use of non-linear function approximators were considered difficult as it could lead the learning algorithm to diverge [8]. One technique called experience replay was employed in the DQN algorithm to mitigate this training instability. In this approach, we store the transitions that the agent experiences at each time step in memory and use random samples of experiences to perform network updates. This sampling approach allows greater data efficiency as each experience stored in the buffer can be used for many weight updates. But, perhaps more importantly, this random sampling breaks the strong correlations that exist among consecutive samples that can introduce bias if used to update network weights in an on-line fashion [7].

For the DQRN algorithm, we randomly choose a position in the buffer and take some number of consecutive transitions to update the RNN model weights.

### 3.3 Model Architecture

The overall architecture is as shown in the diagram below. A series of states are fed to a fully-connected projection layer to be encoded as input to the RNN layer and processed in sequence. The output from the final time step of the RNN layer is then projected to produce a Q-value vector of length equal to the number of possible actions.

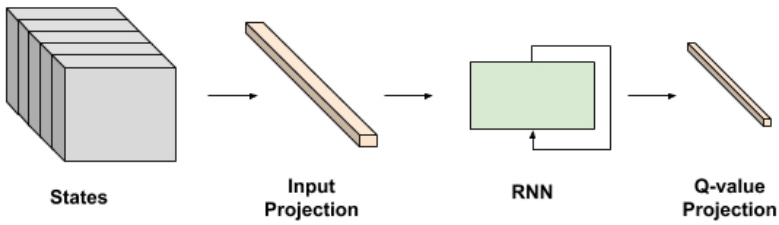



Figure 1: Deep Recurrent Q-network

**Algorithm 1** Deep Recurrent Q-learning

Initialize replay buffer with random exploration strategy
Initialize input projection layer, RNN model, and Q-value projection layer
**for** episode $\leftarrow 1$ to $M$ **do**
  **for** $t \leftarrow 1$ to $T$ **do**
    Select $a_t$ based on the exploration strategy (e.g., $\epsilon$-greedy, Boltzmann sampling)
    Execute $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
    Sample a random batch $b$ of consecutive transitions from replay buffer
    Compute $\hat{Q}(b;\theta)$ with the RNN model on the sequence of transitions
    Update weights of the network based on the TD loss $(y - \hat{Q}(b;\theta))^2$
  **end for**
**end for**

Note that the algorithm outlined above is quite similar to the DQN algorithm except that (a) an RNN model is used to predict the Q-values and (b) the batch contains a sequence of consecutive transitions that can be of an arbitrary length. Both the type of RNN model and the length of the sequence of transitions (which does not need to be fixed) are hyperparameters to be chosen. We illustrate the significance of these factors in the experiment results below.

## 4 Experiments

We have performed various experiments to understand how effective RNN-based deep Q-network is in handling partial observability. In particular, we have used two classic control environments for experiments.

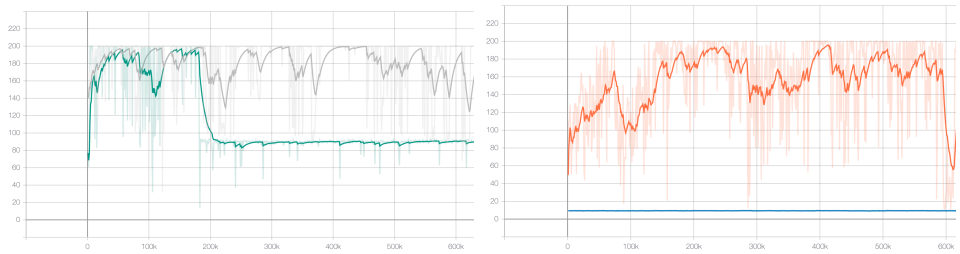
In the cart-pole environment, a pole is attached to a cart that is free to move along a friction-less, horizontal track [2]. The attached pole is also free to move, and the objective is to prevent it from falling over by applying a left or right force of a fixed magnitude at each time step. The state consists of four variables including (a) position of the cart on the track, (b) vertical angle of the pole, (c) velocity of the cart, and (d) rate of change of the vertical angle. All four components are made available to DQN agent, while only the position and velocity are provided to DRQN agent. The hypothesis is that RNN model should be capable of inferring the missing components based on a series of the other available components. A reward of +1 is given at each step that the pole remains upright. The episode ends when the pole moves more than 15 degrees from vertical, the cart moves more than 2.4 units away from the center, or the agent reaches the max time step of 200 that we set. Hence, the max reward the agent can attain is +200.

In the mountain car environment, a car is situated between two uphills, and the goal is to drive the car over the hill to the right. Since the car's engine is not strong enough, the car needs to be driven back and forth to build up enough momentum to climb up the hill. The state consists of position and velocity, and as before, only the position component is made available to simulate partial observability. A reward of -1 is given at each time step, so that the agent has a strong incentive to finish the episode as soon as possible. Here, we set the max time step to 200 and max reward threshold to -110. In other words, the max reward that can be attained is -110 and the min is -200.

The overall experiment setup is as follows. We start by populating the replay buffer with 100k transitions collected with random exploration. Once it becomes full, we start to train the agent using the stored transitions. We adopt an exploration strategy and based on it we select actions to execute. As the agent interacts with the environment, we store the new transitions encountered and also update the network. To update the weights, we use Adam optimizer [5] to reduce the temporal-difference loss. For all the experiments, we have used batch size of 64 and updated the target network every five steps for training stability.

### 4.1 Partial Observability

We have experimented with various configurations of hyperparmeters such as learning rate, training sequence length, and the parameter controlling the level of exploration (e.g., temperature in Boltzmann sampling). With an appropriate set of hyperparameter values, DRQN agent with an LSTM



(a) DQN (b) DRQN

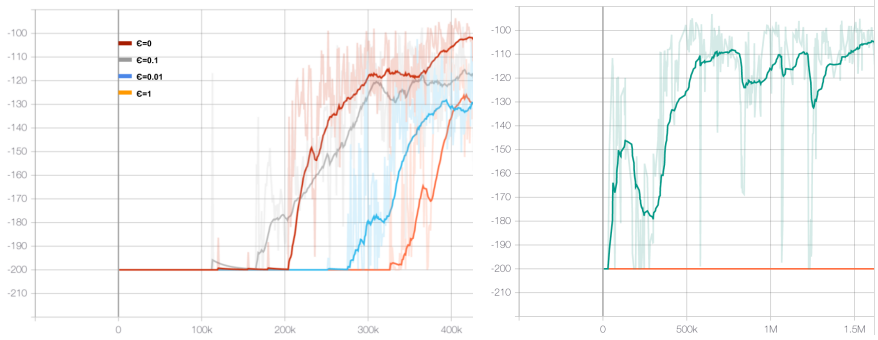
Figure 2: Average rewards on cart-pole with learning rates 0.1 and 1

network was able to achieve the maximum possible rewards in both environments given only partial observability into the states. DQN agent with full observability required less number of updates than DRQN agent with partial observability in order to reach the max average reward. This gap was fairly large depending on the complexity of the environment and the hyperparameter values. Nevertheless, DRQN agent successfully learned to account for the limitation in the two environments, showing sequence-learning models such as RNN models can be employed to effectively handle partially observable states. Two of the more important configurations that had notable impact on performance were the type of exploration strategy and the sequence length used for updating the network weights, which we discuss further below.

One behavior that was commonly observed in both DQN and DRQN agents was that using a learning rate that is too high not only led to a sub-optimal solution but also progressively degraded the quality as more updates were applied. This is in contrast to what is often observed in supervised learning, where using too high of a learning rate leads to an oscillatory behavior rather than a consistent degradation. This phenomenon is perhaps due to the nature of RL learning algorithm, where the network that we update in each step also affects the subsequent actions we choose to take and hence the future updates. The presence of such feedback loops can lead to unstable training, and that is why techniques such as the ones introduced in [7] were critical in making DQN work. Figure 2 shows an example of this behavior when learning rates of 0.1 and 1 were used to train DQN and DRQN agents on the cart-pole environment.

### 4.2 Exploration Strategy

We experimented with two strategies – $\epsilon$-greedy and Boltzmann sampling – with different values of the exploration parameter. Interestingly, we observed that agents that explored less converged faster than agents that explored more. This was the case for both $\epsilon$-greedy and Boltzmann sampling. In one extreme case, DRQN agents on the mountain car environment were able to attain non-worst rewards only when trained with greedy strategy. Figure 3b shows plots of average rewards attained by DRQN agents trained with $\epsilon$-greedy. Only the ones with $\epsilon$ of 0 were able to improve their policies.




(a) DQN (b) DRQN

Figure 3: Average rewards on mountain car with various $\epsilon$ values

| Environment | DQN | DRQN-5 | DRQN-10 | DRQN-20 |
|---|---|---|---|---|
| Cart-pole | 5k | 100k | 15k | 5k |
| Mountain car | 420k | >3M | >3M | 564k |

Table 1: Steps to first reach the max rewards

One possible explanation of this phenomenon is that the environments are relatively simple so that using a large enough replay buffer initialized with experiences collected prior to training is sufficient for the agents to learn from. And, hence, during the actual interaction with the environments, it is better for the agents to exploit than to explore.

### 4.3 Sequence Length

To train DRQN agents with a replay buffer, we randomly select a position in the buffer and take some number of consecutive transitions to update the weights of the RNN network. In the cart-pole environment, where only one missing state component needs to be inferred, using five consecutive transitions was sufficient to train a well-performing DRQN agent. We did observe that using a longer sequence length in general led to faster convergence. In the mountain car environment, where two state components need to be inferred, a sequence length that is too short often led to no policy improvements even after several millions of updates. Only with a length of 20, were we able to train an agent that reached the maximum average rewards. Overall, DRQN agent required more steps to reach the max rewards than DQN agent, and using a longer sequence length helped to converge faster. Table 1 shows the number of steps needed to reach the max average rewards for four different agents on the two environments.

## 5 Conclusion

Deep Q-network is one of early successful applications of deep learning to RL problems. In [7], DL models were used as function approximator to train agents that could learn human-level control policies from raw pixel inputs. One important assumption made was that the state is fully observable. In many real world applications, however, state is often only partially observable to the agent.

We studied effectiveness of RNN models, a popular family of DL models used for sequence-learning tasks, for handling partial-observability. The idea is that if the missing state components can be inferred from a series of other available components, RNN models should be able to do this inference. Our experimental results on two classic control environments demonstrate that with an appropriate configuration of such parameters as learning rate, exploration strategy, and training sequence length, DRQN agent can perform as well as DQN agent despite limited state information.

Depending on the complexity of the environment as measured by the number of missing state components, DRQN agent often required significantly more network updates to perform as well as DQN agent. Using a moderate learning rate was important, and too large of a learning rate led to progressively worse performance as training continued. We suspect that this is perhaps due to the feedback loop that is inherent in RL algorithms such as Q-learning. For exploration strategy, we observed only small difference between $\epsilon$-greedy and Boltzmann sampling, the two strategies we experimented with. We did find, however, that the parameter that controls the level of exploration was quite important. Finally, the number of consecutive transitions to use to train the network had significant impact on convergence. Using a longer sequence length in general led to faster convergence for DRQN agent.

As future work, it would be worth investigating ways to dramatically speed up training of DRL agents. One known disadvantage of Q-learning is that convergence can often be very slow. It might be useful to apply the techniques used to address this issue also in DRL setup. For instance, instead of randomly sampling experiences from replay buffer, we can use a more sophisticated selection strategy such as prioritized sweeping. Improving data efficiency of RL algorithms would allow expanding their applications to more real world problems.

## References


[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[2] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Artificial neural networks. chapter Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems, pages 81–93. IEEE Press, Piscataway, NJ, USA, 1990.

[3] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[8] John N. Tsitsiklis and Benjamin Van Roy. Analysis of temporal-difference learning with function approximation. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, pages 1075–1081, Cambridge, MA, USA, 1996. MIT Press.