Quantum Dyna Q-Learning

Ethan Duryea, Wei Hu

1 Introduction

Reinforcement learning is a sub-field of machine learning that has tremendous strides in recent years. Unlike supervised learning, where an algorithm attempts to learn a function given a labeled dataset, reinforcement learning simply attempts to maximize a reward signal. The simplicity of a reinforcement learning agent's goal enables that agent to learn a wide range of tasks. These tasks can include optimizing power consumption and performance of computer systems and data centers [1][2]. When coupled with deep neural networks, reinforcement learning algorithms have been able to achieve master level scores in several atari arcade games [3]. AlphaGo was a program that combined deep neural networks, supervised learning, and reinforcement learning to learn how to play the full sized version of the board game Go [4]. Given that a 19x19 Go board has $2.08168199382 \times 10^{170}$ legal game configurations, this is a huge achievement for reinforcement learning.

1.1 Reinforcement Learning

1.1.1 Markov Decision Process

A Markov Decision Process (MDP) provides a mathematical framework that can be used to study a wide range of optimization problems. An MDP consists of an agent interacting with an environment. At a given time t, the agent will be in state s and can choose to take an action a available in s. The agent a in s corresponds to a time step $t+1$, where the agent has transitioned to s' based on a probability given by a: $P_a(s, s')$. The transition from s to s' will yield a reward r with probability $R_a(s, s')$. The agent's goal is to maximize the discounted return. The discounted return is defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where γ is a parameter, $0 \leq \gamma \geq 1$ is a discount factor which controls the importance of future rewards. This function can also be written recursively as

$$G_t = r_t + \gamma G_{t+1}.$$

The agent uses a value function to develop a policy for choosing actions. A policy, $\pi(s|a)$, maps state s to the probability of taking the possible action a. The value function V^π of state s while operating under policy π yields the expected return when beginning in state s and following π. $V^\pi(s)$ is defined as

$$V^\pi(s) = \mathbb{E}_\pi[G_t|s_t = s].$$

We can estimate the value function through the agents interactions with the environment by keeping an average the returns for each individual states. As time approaches infinity, the average will converge to $V^\pi(s)$.

1.1.2 Q-Learning

Q-learning is a popular off-policy, temporal difference learning algorithm [5]. Q-learning uses an action-value function, $Q(s, a)$ as opposed to a state-value function $V(s)$. One step Q-learning is defined as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{\mathbf{a}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)].$$

Q-learning is considered to be off-policy because Q directly approximates the optimal action-value function q_* regardless of the policy being followed. We are updating $Q_t(s, a)$ by using target formed immediately after performing an action and receiving a reward. The target in Q-learning is $r_t+\gamma \max_{\mathbf{a}} Q(s_{t+1}, a)$. The update value is simply the difference between the target estimate and the current estimate, hence temporal difference.

1.1.3 Double Q-learning

It has been observed that Q-learning will often overestimate action-values in a stochastic environment [6] [7], resulting in poor learning. The overestimation is due to using the same value function to select and evaluate an action. To fix solve this problem, [8] proposed to decouple the selection and evaluation by introducing a second Q function. At a given time t, the Double Q-learning algorithm selects to update either Q_A or Q_B. The one step update for Double Q-learning becomes

$$Q_A(s_t, a_t) \leftarrow Q_A(s_t, a_t) + \alpha[r_t + Q_B(s_{t+1}, a^*) - Q_A(s_t, a_t)]$$

where $a^* = \max_{\mathbf{a}} Q_A(s_{t+1}, a)$. It was observed by [8] [9] that Double Q-learning can provide more stable and reliable learning.

1.1.4 Multiple Q-learning

In order to achieve more stable and robust learning, Double Q-learning has been extended to Multiple Q-learning [10] [11]. Like Double Q-learning, Multiple Q-learning decouples the selection and evaluation of an action. The key difference lies in the use of multiple Q functions facilitate stable and accurate evaluation.

For a Multiple Q-learning algorithm with $Q_1(s,a), Q_2(s,a), ..., Q_N(s,a)$, the update for Q_A becomes

$$Q_A(s_t,a_t) \leftarrow Q_A(s_t,a_t) + \alpha[r_t + \frac{1}{N}\sum_{i=1,i\neq A}^{N} Q_i(s_{t+1},a^*) - Q_A(s_t,a_t)]$$

where $a^* = \max_{\mathbf{a}} Q_A(s_{t+1},a)$, α is the learning rate, and $P(A=i) = \frac{1}{N}\forall i \in [1..N]$. The stability of the estimated value, $Q(s_{t+1},a*)$, increases with the use of more Q-functions, which leads to better learning for some problems.

1.1.5 Dyna Q-learning

A model can be used in a reinforcement learning algorithm to assist with finding a optimal policy. By model, we mean anything that can an agent can use to predict the dynamics of an environment. An agent can use either a distribution model or a sample model. A distribution model contains all of an environment's possibilities, including state transitions and rewards, and their probabilities. A sample model will return only one possibility, sampled according to the probabilities. Sample models are much easier to obtain and are thus more commonly used in reinforcement learning. Dyna Q-learning a version of Q-learning that uses a model of the environment to help compute its value function and in turn, find an optimal policy. The algorithm develops its model through direct interactions with the environment. At each transition $s_t, a_t \rightarrow r_{t+1}, s_{t+1}$, Dyna Q stores these values in a lookup table. When a s_t, a_t are given as inputs, the lookup table return the last r_{t+1}, s_{t+1} received from the environment. Real experiences in Dyna Q is used in two ways, direct learning or model learning. The direct learning phase uses the same update as one-step Q-learning. Throughout training, the model is used to create simulated experiences. The model can generate a single transition, or a complete episode. The simulated experiences are used to update the value function following one-step Q-learning.

1.2 Quantum Computing

Quantum Computing has been the focus of rapid development in recent years. As digit computers approaches their limits in processing power, quantum computing offers to provide a dramatic speedup. Currently, IBM provides an web interface for users to access and program a 5 qubit quantum computer. IBM has also developed 20 qubit and 50 qubit processors [12]. To demonstrate the future of quantum computing and quantum supremacy [13], Google has a 72 qubit processor.

In basic unit of classical computing is the bit. A bit can be in one of two state, 0 or 1. A string of n bits can be in one of 2^n at a time. The quantum counter part of the bit is the qubit. Like a bit, a qubit can be either $|0\rangle$ or $|1\rangle$. Unlike the bit, the qubit can be in a superposition of $|0\rangle$ and $|1\rangle$, known mathematically as $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. If a quantum computer has n qubits, it has the potential to be in

a superposition of up to 2^n states. Quantum computers are also capable of entanglement and teleportation.
Some quantum algorithms have been able to achieve drastic speedups compared to their classical counterparts. Some of the most notable are Shor's algorithm [14], which can factor integers in polynomial time and Grover's algorithm [15], which can search an unordered list in $O(\sqrt{N})$.

1.2.1 Grover's algorithm and amplitude amplification

Grover's algorithm uses the principles of amplitude amplification to increase the probability of selecting the correct item from an unordered list. The algorithm uses an oracle function that returns $f(x) = 1$ if x is the element we are searching for and $f(x) = 0$ if it is not. The oracle can be thought of as a quantum black box. Once a call to the oracle is made, Grover's algorithm will amplify the amplitude of the element x which returned $f(x) = 1$. The amplitude corresponds to the probability of a state being observed when a quantum system is measured. If we measure the quantum state $\alpha|0\rangle + \beta|1\rangle$, then the probability of observing $|0\rangle$ is $|\alpha|^2$ and the probability of observing $|1\rangle$ is $|\beta|^2$. If we are given that same quantum state, and wish to observe $|0\rangle$, we can use Grover's algorithm to amplify α such that $|\alpha|^2 = 1$.

1.2.2 Quantum Reinforcement Learning (QRL)

With the progress of quantum computing has sparked an interest in quantum machine learning algorithms [16]. Quantum Reinforcement Learning [17] combines reinforcement learning with quantum computing techniques and achieves drastic speedups in problem solving. The main idea is to encode the states and actions of an environment using qubits. This allows a superposition of all possible states and actions to be created.

2 Methods

Dyna QRL attempts to combine the benefits of planning with a model and Quantum Reinforcement Learning. The main idea of Dyna QRL is to use real experiences to build a quantum dyna model of the environment. The quantum dyna model is a superposition of experiences

$$\frac{1}{\sqrt{T}} \sum_{j=0}^{T} |s_j\rangle|a_j\rangle|r_j\rangle|s'_j\rangle$$

where $|s_j\rangle|a_j\rangle$ is a quantum register holding the inputs to the dyna model and $|r_j\rangle$ and $|s'_j\rangle$ are ancillary qubits that hold the output of the dyna model given $|s_j\rangle|a_j\rangle$. The model behaves as a function $M(s, a) \rightarrow (r, s')$ As the agent interacts with the environment, it will generate experiences and store them in the model. At the end of each time step, the algorithm will use the model to update

the Q-function. In classical Dyna Q-learning, only a subset of the experiences in the model are used. Because Dyna QRL's model is a superposition of all previously seen states and actions, we are able to update using the entire model simultaneously.

To increase effectiveness in environments with stochastic reward function, Dyna QRL can use multiple models to approximate the expected reward. At time t, a Dyna QRL algorithm with $M_1, M_2, ..., M_K$ models will choose a model with index A to update with a real experience. Index k is chosen with probability $P(A = k) = \frac{1}{K} \forall k \in [1..K]$. During the planning phase, the algorithm will choose a single model M_k to update the Q-function and perform action amplitude amplification. All of the state-action pairs (s_j, a_j) in M_k will be used for the update. We create a set $\mathcal{M} = \{M_1, M_2, ..., M_K\}$ and set $\mathcal{M}_{(s_j,a_j)} = \{M_k | M_k \ni (s_j, a_j) \forall k \in [1..K]\}$ that contains all the models with the experience (s_j, a_j). $\mathcal{M}_{(s_j,a_j)}$ will be used to calculate the expected value of the reward r_j that will be used for the update.

Algorithm 1 Dyna QRL

Initialize $Q_1(s,a), Q_2(s,a), ..., Q_N(s,a)$, $|a_s\rangle$ and $M_1, M_2, ...M_K$
loop for each episode :
  do until s is terminal :
    Choose M_k from $\mathcal{M}$ with probability $\frac{1}{K}$
    Choose Q_n with probability $\frac{1}{N}$
    $s \leftarrow$ current (nonterminal) state
    Observe a from $|a_s\rangle$
    Take action a, observe s' and r
    Store $M_k(s,a) \rightarrow (r, s')$
    For each $(s,a) \in M_k$:
      $l = \text{length}(\mathcal{M}_{(s,a)})$
      $r = \frac{1}{l} \sum [r_i \leftarrow M_i(s,a) \forall M_i \in \mathcal{M}_{(s,a)}]$
      $s' \leftarrow M_k(s,a)$
      Compute $V'_s = max'_a \frac{1}{n} \sum_{i=1}^{n} Q_i(s', a')$
      Compute $L = min\{k(r + V'_s), int(\frac{\pi}{4\theta} - \frac{1}{2})\}$
      Update $|a_s\rangle \leftarrow \hat{U}_g^L |a_s\rangle$
      Choose $b = argmax'_a Q_n(s', a')$
      Update $Q_n(s,a) \leftarrow Q_n(s,a) + \alpha[r + \frac{1}{N} \sum_{i=1, i \neq n}^{N} Q_i(s', b) - Q_n(s,a)]$
    $s \leftarrow s'$

3 Results

We used a 4x4 grid world to compare the performance of the algorithms. In each state, the agent can take four possible actions: up, down, right and left. The environment has two terminal states, a goal and a pit. If the agent reaches the

goal, it will receive a reward of +10. Falling into the pit will generate a reward of -10. For the other 24 states, the reward is generated either by a deterministic function or a stochastic function, depending on the test we are running. If the agent is in any of the 12 border positions, and takes an action that corresponds with moving into the border, the agents new state will be identical to its previous state and will receive an appropriate reward. The same rule is applied to the "wall" position marked **W** in Figure 1. The agent is not allowed to enter this position. If it tries to move into the wall, it will be returned to its original state and given a reward corresponding to moving into said state. At each time step, the agent has a complete representation of the current state of the grid world, including its current position and the positions of the goal, pit, and wall.

In order to compare the performance of the algorithms, we will use the following metrics: number of episodes/steps to converge to the optimal path, q-value estimates for the starting grid state, and the average reward curve. The grid world has two shortest paths to the goal state, each defined by the string of actions that the agent will take. The shortest path is defined as "33111" and the second is defined as "31311". Of these two paths, the second path is less optimal due to its proximity to the pit.

In this section, our results will demonstrate the areas which our proposed Quantum Dyna-Q algorithm outperforms others Q-learning based algorithms. We will also show potential shortcomings of our proposed algorithm.

3.1 Learning Curves

It is critical that learning algorithms converge to the optimal policy quickly, with as few episodes as possible. In this subsection we compare the speed of convergence of each algorithm by comparing the number of episodes needed to converge to the optimal policy. We will first use an environment with a deterministic reward function, than later compare the algorithms with different stochastic functions. It has already been show that QRL drastically outperforms its classical counterpart, what we wish to demonstrate here is how adding a dynamic backup subroutine will impact the speed of convergence.

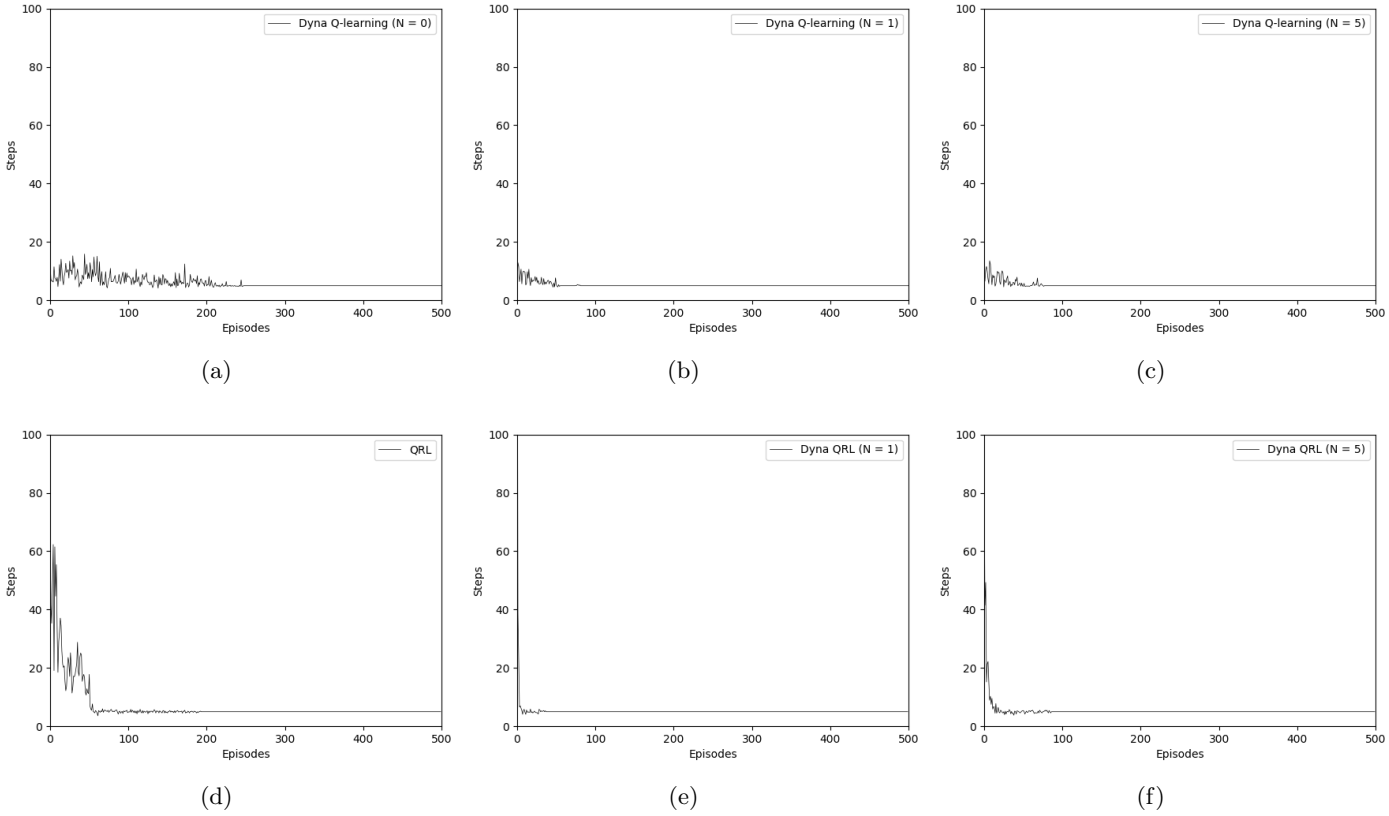


Figure 1: Learning Curves for Q-learning, Dyna Q-learning, QRL, and Dyna QRL. These figures compare the number of steps per episode of the four algorithms during the training process. For this experiment, $\alpha = 0.06$ and $\sigma = 0$.

In a deterministic environment, the speedup achieved by the Dyna-QRL algorithm is substantial. The Dyna-QRL ($N = 1$) algorithm took on average 11.6 episodes and 160.8 steps to reach the optimal policy. The Dyna-QRL ($N = 5$) algorithm took an average of 19.2 episodes and 223.8 steps to reach the optimal policy. It would appear that in a determinstic environment, an increased number of dyna models does not help increase the speed of learning but in fact will slow learning down. The increased cost of implementing multiple models is also an important factor to consider when comparing the Dyna-QRL algorithm.

In our tests, QRL took an average of 61.2 episodes and 1281.3 steps to converge to the optimal policy. Classical Double Q-learning took an average of 134.05 episodes and 1189.8 steps. Classical Dyna Double Q-learning performed well in this experiment, converging in 25.75 episodes and 188.55 steps. Based on these results, Dyna-QRL performed the best of the algorithms we tested. Dyna-

QRL converged 81% faster than QRL, 91% faster than Double Q-learning, and 55% faster than Dyna Double Q-learning.

The results are similar when a stochastic reward function is used. We ran the algorithms in an environment with two stochastic reward functions, one with a standard deviation of $\sigma = 5$ and other where $\sigma = 10$.

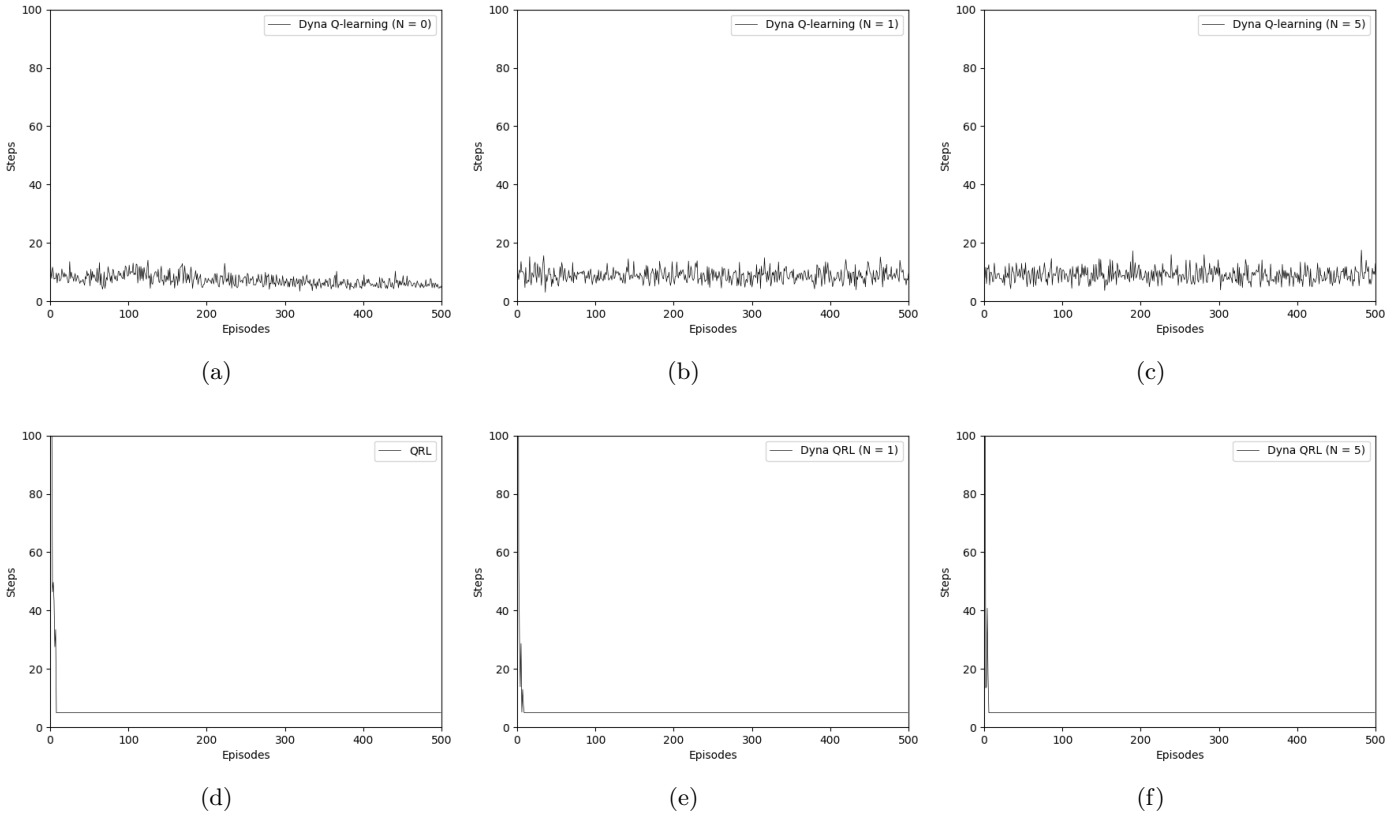


Figure 2: Learning Curves for Q-learning, Dyna Q-learning, QRL, and Dyna QRL in an environment with a stochastic reward function. These figures compare the number of steps per episode of the four algorithms during the training process. For this experiment, $\alpha = 0.06$ and $\sigma = 5$.

The classical dyna algorithms did not perform well when faced with a stochastic reward function. The model representation used is too simplistic to handle uncertain responses from the environment which cause the algorithm to be unable to learn. The model-free Q-learning algorithm outperforms Dyna Q-learning by a factor of 4.5 when $\sigma = 5$.

While classical Dyna Q-learning did not perform well in a stochastic environment, we found that its quantum counterpart did. Both the Dyna QRL al-

gorithms we tested out performed the model-less QRL algorithm. When $\sigma = 5$, Dyna1 QRL optimized its policy in 4.7 episodes and 1088 steps on average. Dyna5 QRL took 1256 steps in 3.3 to converge and QRL took 4.1 episodes and 1679.6 steps. Both Dyna algorithms clearly outperform QRL, especially in the number of steps needed to maximize the reward signal.

The second stochastic reward function we used had $\sigma = 10$. In this case, QRL took an average of 3.6 episodes and 1151.0 steps to reach the optimal action-selection policy. Classical Q-learning converged in 335.75 episodes and 2999.25 steps while classical Dyna Q-learning converged in 1001.0 episodes and 9027.7 steps. Dyna QRL was able to converge on an average of 3.0 episodes and 2164.6 steps while Dyna QRL ($N = 5$) took 3.5 episodes and 3356.3 steps.

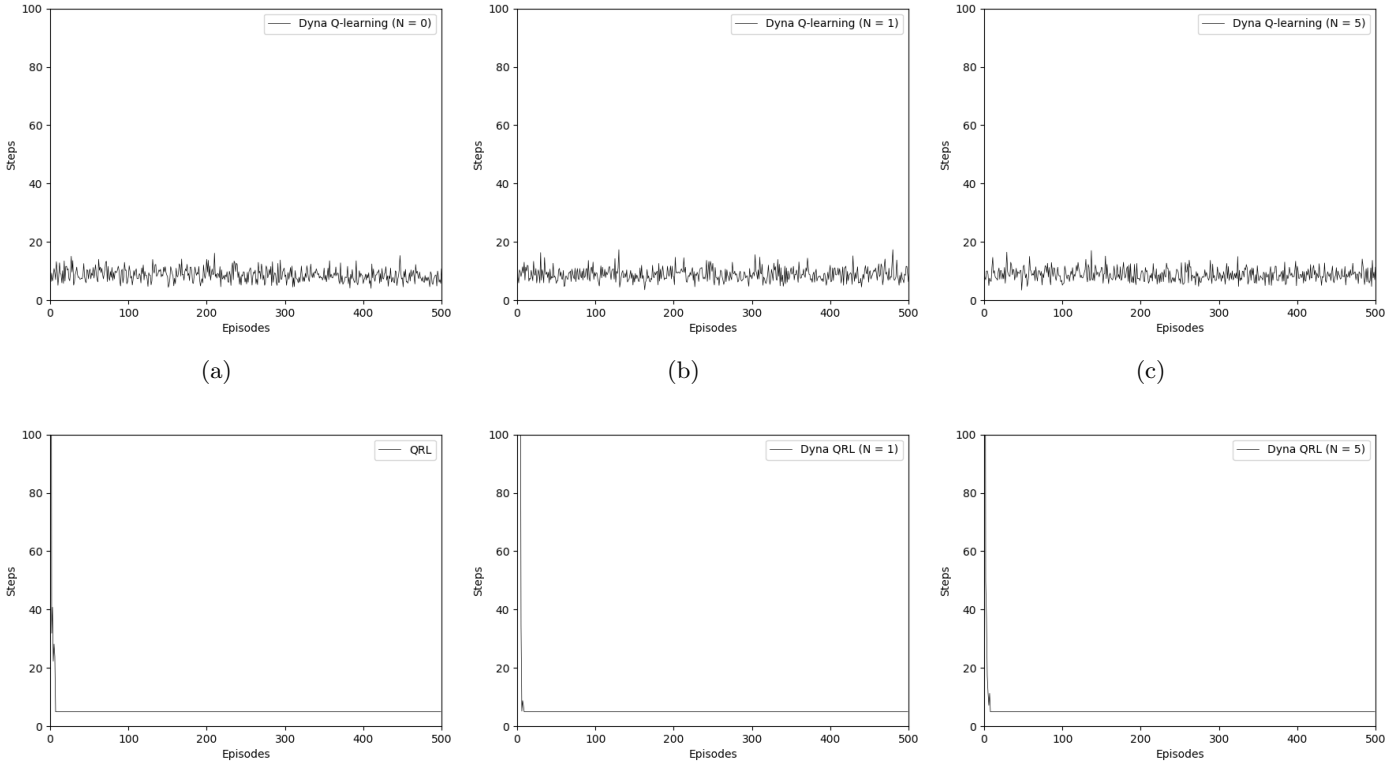


Figure 3: Learning Curves for Q-learning, Dyna Q-learning, QRL, and Dyna QRL in an environment with a stochastic reward function. These figures compare the number of steps per episode of the four algorithms during the training process. For this experiment, $\alpha = 0.06$ and $\sigma = 10$.

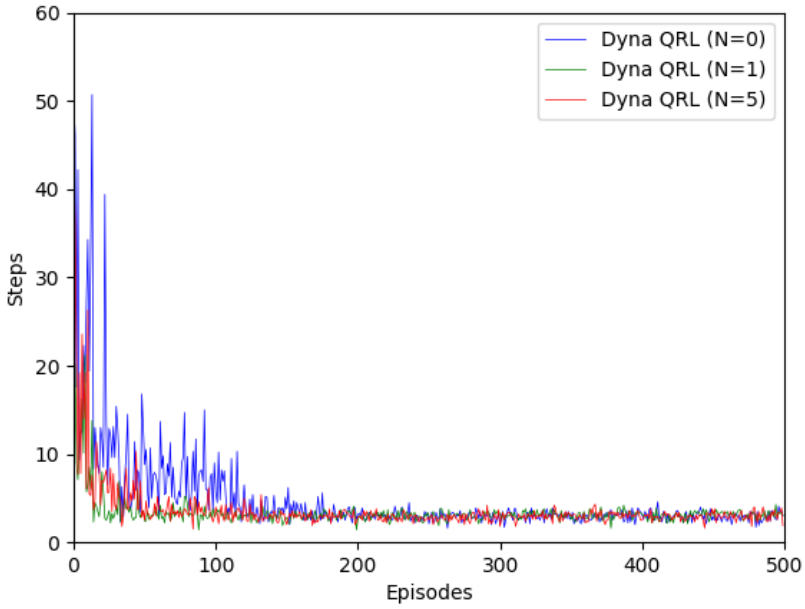
Clearly, if we measure performance as the number of episodes it takes an

algorithm to converge to the optimal policy, Dyna QRL performed the best. But if we compare the number of steps, QRL performed the best.

The robustness of utilizing amplitude amplification to form an action-selection policy is clearly demonstrated here. While the classical dyna Q algorithm failed to quickly find the optimal path to the goal, the quantum dyna QRL algorithm found it in under 10 episodes.

3.2 Learning Curves with stochastic starting positions

We also tested and analyzed the algorithms in an environment where the agent begins each episode in a random state. The randomness of the starting position requires an algorithm to fully explore the gridworld and come up with accurate state-action value estimates.



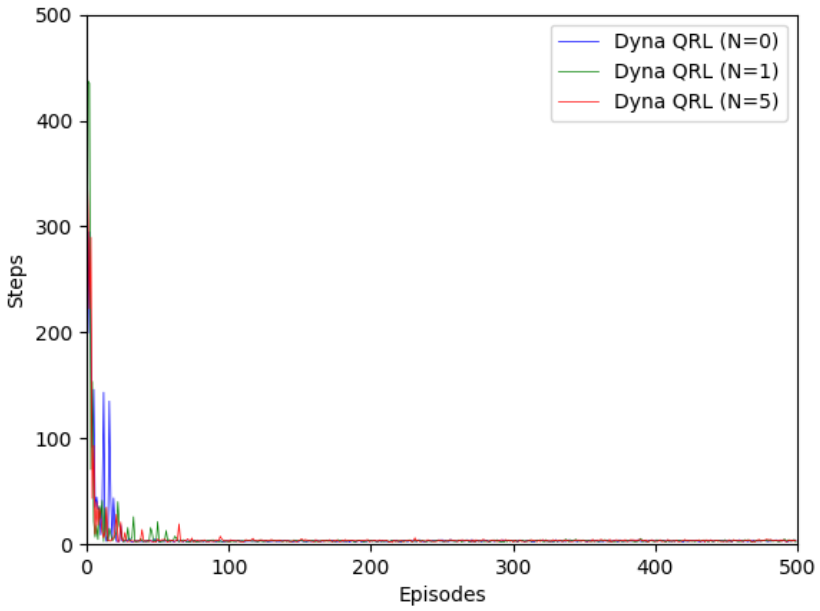
(a)

Figure 4: Learning curves of QRL and Dyna QRL algorithms in an environment where the starting position is randomly chosen at the beginning of each episode. In this experiment $\alpha = 0.06$ and $\sigma = 0$.

In our testing, we found Dyna QRL more capable of handling this randomness. Figure ? shows the number of actions taken per episode. Dyna QRL ($N = 1$) perform exceptionally well on this task, finding the optimal path for each starting state in 29 episodes and 188 steps. In shape contrast, the model-

less QRL algorithm took 104 episodes and 1005. In this setting, Dyna QRL had a performance gain of 81% with regard to steps and 72%.

Figure ? shows the learning curves from an environment with a stochastic starting state and a stochastic reward function. Both Dyna QRL and QRL needed significantly more steps to find the optimal paths. QRL converged in 2029 steps while Dyna QRL took 1436. With the increased randomness to the environment, Dyna QRL still outperforms QRL by 29%.

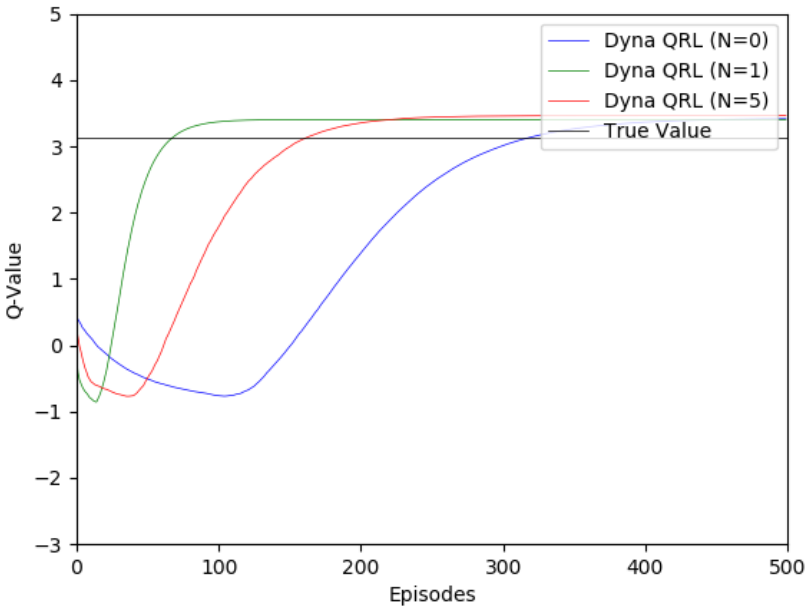


(a)

Figure 5: Learning curves of QRL and Dyna QRL algorithms in an environment where the starting position is randomly chosen at the beginning of each episode. In this experiment $\alpha = 0.06$ and $\sigma = 5$.

3.3 Action Value Estimates of QRL and Dyna QRL

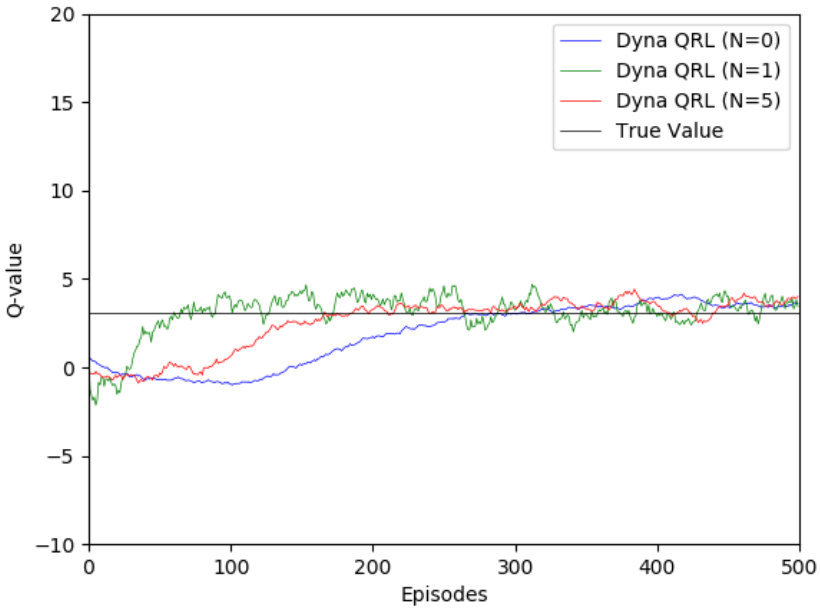
An key property of a reinforcement learning algorithm is its ability to accurate estimate the value of a given state. It has been observed that in a stochastic environment, using multiple Q-functions increases the accuracy of an agent's value estimates [2][11][10]. The curves for each algorithm's value estimate of the initial state are shown in figure ?. The true value of this state is 3.2.



(a)

Figure 6: These graphs show the Q-values of QRL and Dyna QRL for the initial state of the environment. The true Q-value for this state is 3.122. In this experiment $\alpha = 0.06$ and $\sigma = 0$.

Using a deterministic reward function, each algorithm we tested was able to estimate the true value of the initial state. Both Dyna QRL algorithms were faster in their value estimates, converging to the true value in approximately 50 episodes. QRL took significantly longer to converge, reaching the true value after 300 episodes. Dyna QRL runs backup simulations for each previously seen state after every action, resulting in many more value function updates during training. Through episodes 15 and 150, QRL's value estimates dropped below zero. QRL an average took 1281.3 steps to converge to the optimal path. The majority of the steps returned a negative reward switch translates to low value estimates. Because Dyna QRL found the optimal path in only 287.8 steps, it did not face this problem.



(a)

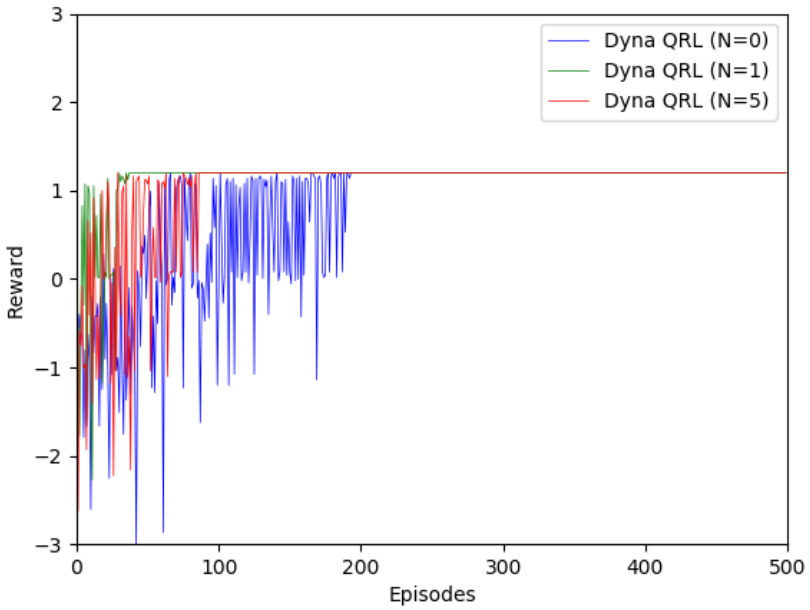
Figure 7: These graphs show the Q-values of QRL and Dyna QRL for the initial state of the environment. The true Q-value for this state is 3.122. In this experiment $\alpha = 0.06$ and $\sigma = 5$.

Figure ? shows the value estimate curves for each algorithm in environments with stochastic reward function. Each algorithm was successful in finding the true Q-value. The rate at which the algorithm converged to the true value was similar in the stochastic environment as the deterministic environment. Each algorithm took fewer episodes to converge to the optimal policy, but the number of steps per episode increased dramatically. The uncertainty of the reward function caused the algorithms to explore more during the early stages of training, allowing the algorithms to find the optimal path to the goal in fewer episodes at the cost of more steps.

3.4 Average Return Rates of QRL and Dyna QRL

The goal of a reinforcement learning algorithm its to discover an action selection policy that will allow an agent to maximize a reward signal. The reward from the environment determines if a particular action in a particular state is a good one. A good algorithm is able to maximize a reward signal quickly and thus achieve a greater return long term. Figure 8 shows the average reward per episode for QRL and two Dyna QRL algorithms, N=1 and N=5. It is clear from

the graphs that both Dyna QRL algorithms are able to return the maximum possible reward of 3.4 faster than QRL. This is of course do Dyna QRL ability to quickly find and converge to the optimal action selection policy. Figure 9 shows how the average accumulated reward for the three algorithms.



(a)

Figure 8: These graphs show the average reward per each episode for QRL and Dyna QRL. The maximum possible reward per episode is 3.4. In this experiment $\alpha = 0$ and $\sigma = 0$.

Figure 9 shows the averaged accumulated reward for the three algorithms. These graphs clearly show the benefit of algorithm that learn at a faster rate. For each algorithm, the episodes in the early stages of training yield negative rewards. These poor returns are due to the high amount of exploration that happen before the agents discovers the optimal policy. For Dyna QRL (N=1), the negative reward period ends quickly, in less than 25 episode. QRL takes many more episodes to begin increasing its average accumulated reward. Towards the end of training, Dyna QRL (N=1) reaches accumulated rewards of also three, while QRL barely obtains two.

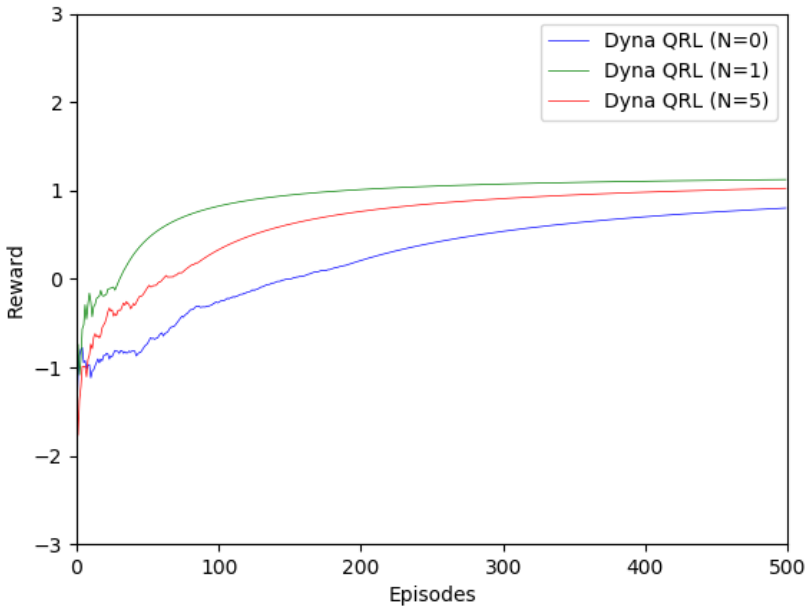


Figure 9: This figure shows the accumulated reward divided by the episode for QRL and Dyna QRL. In this experiment, $\alpha = 0.06$ and $\sigma = 0$.

3.5 Comparison of Dyna QRL with Multiple Models

In this section, we discuss the benefits of using multiple models inside the Dyna QRL algorithm to simulate an environment with a stochastic reward function where $\sigma = 5$. Figure 10 shows the average Q-values and figure 11 shows the standard deviation of the Q-values for each algorithm. We used a learning of $\alpha = 1.0$ to demonstrate how the use of more models can increase the robustness of the Dyna QRL algorithm. As shown in Figure 10, Dyna QRL ($N = 1 - 5$) have Q-Value estimates that are well above the true value while ($N = 6 - 15$) estimates are much more accurate. The use of more models makes Dyna QRL less dependent on hyper-parameter choice, giving a robust algorithm. The standard deviation of the Q-values also decreases with more models. In this environment $\sigma = 5$ and with the chosen hyper-parameters $\alpha = 1.0$, the decrease in SD levels off at ($N = 4$). This shows that more models increase the stability of the algorithm's value estimates.

For these tests, the multiple models were used to find the average reward for each action. Each non terminal move has an expected value of -1. The more models our algorithm has, the better it will estimate this value, which results in a more accurate representation of the environment. This technique can also be used in an environment where action a in state s has a probability

of transitioning to state s', given by a function $P_a(s, s')$. By using a sufficient number of models, we can approximate the transition function as well as the reward function.

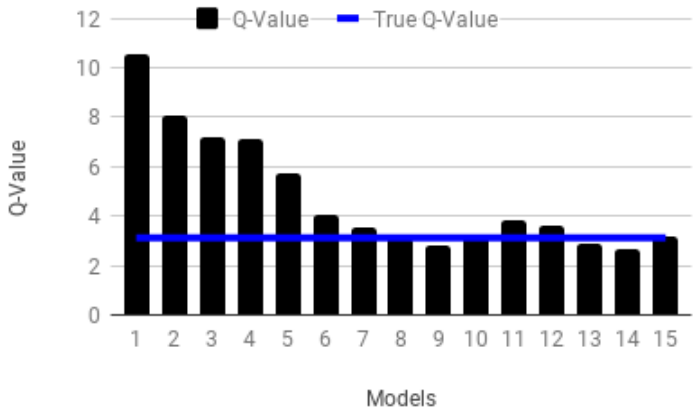


Figure 10: This figure shows 15 Dyna QRL algorithms' initial state Q-Values. The number of models for each algorithm is show on the horizontal axis. The true Q-Value of the initial state is 3.122. In this experiment $\alpha = 1.0$ and $\sigma = 5$.

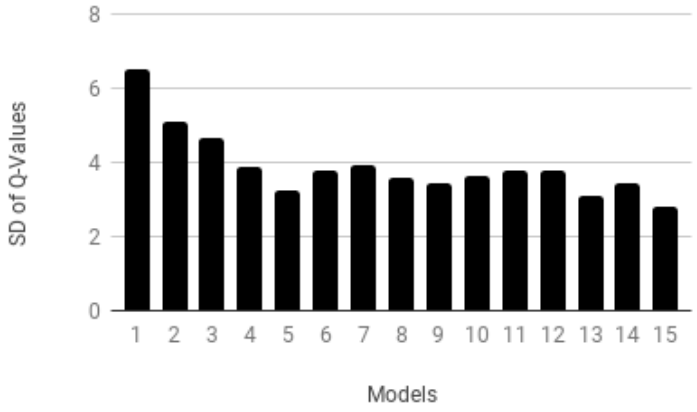


Figure 11: This figure shows the standard deviation of the Q-Values for the initial state. The number of models for each Dyna QRL algorithm is show on the horizontal axis. In this experiment $\alpha = 1.0$ and $\sigma = 5$.

References

[1] Gerald Tesauro, Rajarshi Das, Hoi Chan, Jeffrey Kephart, David Levine, Freeman Rawson, and Charles Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1497–1504, 2008.

[2] Xavier Dutreilh, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. From data center resource allocation to control theory and back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 410–417. IEEE, 2010.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[5] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[6] James E Smith and Robert L Winkler. The optimizers curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

[7] Eric Van den Steen. Rational overoptimism (and other biases). *American Economic Review*, 94(4):1141–1151, 2004.

[8] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[10] Ethan Duryea, Michael Ganger, and Wei Hu. Exploring deep reinforcement learning with multi q-learning. *Intelligent Control and Automation*, 7(04):129, 2016.

[11] Michael Ganger, Ethan Duryea, and Wei Hu. Double sarsa and double expected sarsa with shallow and deep learning. *Journal of Data Analysis and Information Processing*, 4(04):159, 2016.

[12] Ibm announces advances to ibm q systems & ecosystem, Nov 2017.

[13] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *arXiv preprint arXiv:1608.00263*, 2016.

[14] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. Ieee, 1994.

[15] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[16] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195, 2017.

[17] Daoyi Dong, Chunlin Chen, Hanxiong Li, and Tzyh-Jong Tarn. Quantum reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(5):1207–1220, 2008.