Quantum Neural Network for Continuous Reinforcement Learning

Erik Sorensen

Data Science Honors Project Proposal

Introduction

Quantum computing and machine learning are two exciting fields. Quantum computing brings a new perspective on computers by using properties of quantum mechanics to do computations, which can be much faster than the classical computers we know today. Not only are they faster, but some tasks that we thought were impossible with the computers we know today, can be accomplished using quantum computers. (CITATION) Machine learning learns complex problems from large amounts of data that are too difficult for humans to manually solve. It has been used in facial recognition (CITATION) and identifying weeds in precision agriculture (CITATION). A more recent sub-field of machine learning is *reinforcement learning*. Reinforcement learning uses a positive or negative reinforcement signal from an environment, known as the *reward*, to provide feedback to the learning system so that it may learn and improve to maximize this reward. Some areas that use reinforcement learning are economics and artificial intelligence in games. (CITATION) Reinforcement learning is an attractive tool to use. While other machine learning techniques often times require a well tuned, large dataset to learn, reinforcement learning only requires a reward from an environment and time to train. Even further, using a quantum computer to do reinforcement learning has the potential to solve problems that would be too computationally complex for classical computers to solve. This increases the scope of problems we can solve exponentially.

Reinforcement learning

Reinforcement learning may be applied to many fields. As an example, I will use a vacuuming robot to show a specific use case of how reinforcement learning can be applied in the world. Lets imagine if this robot used reinforcement learning techniques. The robots main goal is to vacuum an *environment*, while avoiding furniture and other obstacles.

With reinforcement learning, the robot can self-teach itself by exploring the room. The only feedback the robot gets is *observations* about its surroundings via sensors, and a *reward* signal that indicates how well the robot is doing at its job. Every action the robot takes it will receive either a positive reward signal for good actions or a negative reward signal for bad actions. The robots goal then is to maximize this positive reward signal so that it knows it is doing a good job. The part of the robot that decides the next *action* it should take, whether it be going left or right or going back to charge, we will call the *agent*.

How then can the robot learn to efficiently do this task with these tools it has at its disposal? This problem is called the reinforcement problem [1], which gives the instructions for most reinforcement learning algorithms. It is stated here:

> For an agent in a certain state at a certain time, what action should be taken that will maximize the overall reward, now and in the future?

To summarize, the goal of reinforcement learning is to find a *policy* which maximizes the *reward*. A *policy* (pi) is a probability distribution that describes the probability the agent will take a certain action given it is in a certain state. The *optimal policy* is a policy that on average receives the greatest total reward. This is useful, especially in terms of the vacuuming robot who does not know anything about the environment it has been placed in. By defining the rewards the robot will search toward, it will learn itself how to best maximize this reward in the environment. Since we defined the robots reward as avoiding furniture in the environment, it will learn the best paths it can take to avoid these obstacles. All methods of reinforcement learning have this same goal, which is to find the optimal policy to maximize reward.

Value Function and Action-Value Function

One method for discovering the optimal policy is to estimate the value of reward the agent receives in each state. Once this value is known, the best action an agent can take is the one that will accumulate the highest value across all future states. This method is called value iteration which returns a prediction of the expected accumulative, discounted, future reward, measuring how good each state is [2]. The expected total reward is called the return. The value function is described by,

$$V(s) = E\{R_t | s_t = s\}$$

where E is expected value operator, R_t is the total return that an agent receives given state s_t at time t, and s is the current state that the agent is in. Another solution to finding the optimal policy is called Q-learning. Q-learning expands on the value function where instead of determining the value of being in a certain state, it maximizes the reward over infinitely many successive steps, starting from the current state. The action-value function is given by

$$Q(s, a) = E\{r_{t+1} + \gamma V(s_{t+1}) | s_t = s, a_t = a\}$$

where $r_t + 1$ is the reward from moving into state $s_t + 1$ from s_t by taking action a [3]. The action-value function has many benefits for learning, one of which is that it integrates well with *online learning.* Online learning algorithms are executed on data acquired in sequence [2]. The agent will use the current estimate of the optimal policy while exploring the environment in search of a better estimate of the optimal policy.

Q-learning is a famous reinforcement learning algorithm that uses the action-value function to converge on the optimal policy iteratively, meaning that after each step it obtains a better estimate of the action-value and will eventually converge on the true optimal policy values for each step [3]. The formula for Q-learning is given by,

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

[4], where $Q(s_t, a_t)$ is the *action value*, s_{t+1} is the state moved to out of s_t by taking action a_t, r_t is the reward from this transition, α is the learning rate, and γ is the discount rate which describes the discount rate of future rewards.

Using the action-value function has two main advantages over the value function for learning. Firstly, it is easier to learn the action-value function to determine the optimal policy. Second, it is a simple task to evaluate policies with the action-value function because the next action of the agent is always determined by the best action that maximizes the action-value function (i.e. $a = argmax'_a Q(s, a')$).

Deep Reinforcement learning

Neural Networks

Often, the action-value function $Q(s, a)$ is represented as a table, where each state-action pair s, a maps to a particular reward value in the table. Generally, this table can get quite large with big environment spaces, sometimes they can be nearly *continuous* in size. In these scenarios, Q-learning becomes impractical because the table becomes much too large to converge on the optimal policy in a reasonable amount of time. Instead of this approach, we can try to approximate a function $f(s, a)$ of $Q(s, a)$ so that instead of directly learning each value in the table, we can learn the *parameters* of this function instead.

To do this, we can use *function approximation.* Function approximation methods expect to receive examples of the desired input-output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the s, a of each update as a training example. We then interpret the approximate function they produce as an estimated *value function* [5]. Therefore, instead of saving the values in the Q-table to determine the best next action, we are taking better actions at each time step in accordance to the learned function approximator.

Once we frame the problem as a function approximation problem, we can use *neural networks* to learn the approximation of these functions. Neural networks are a computer learning algorithm that is modeled after a simplified version of biological neurons that are grouped into *layers*, where each layer is the result of previous layers multiplied by a *weight*. Neural networks must be trained by a technique called *supervised learning*, which is a machine learning task of learning a function that maps an input to an output based on example input-output pairs collected from an environment. Usually, supervised learning is done for classification problems, but neural networks may be applied to other problems by carefully selecting an *activation function*. An activation function is a function that describes the output of a neuron. Usually, activation functions compresses the outputs of neurons into a sigmoidal shape within a certain range, similarly to how biological neurons are activated.

Q-Networks

When Neural Networks are applied to Q-learning they are called Q-Networks [6]. Recall the previous formula for Q-learning written previously. With neural networks, we want to update the weights of the neural network to reduce the error, more specifically the *Temporal Difference Error* (TD). The TD error is calculated as follows,

$$TD = (r + \gamma \max_a Q(s_{t+1}, a, \theta)) - Q(s_t, a_t, \theta).$$

As seen above, the TD error is the difference between the maximum possible value for the next state and the current prediction of the Q-value. Now that we can judge how well we are predicting the value of the next steps, we need a formula for improving the weights of our neural network so that it can continuously improve its predictions over each time step t. To do this, we can use *gradient descent*, which is used to update the weights (θ) of our neural network so as to minimize the TD error. Adding gradient descent to our formula we get,

$$\theta \leftarrow \theta + \alpha[r + \gamma \max_a Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta)]\nabla_\theta Q(s_t, a_t; \theta).$$

Q-Networks work very well on large environments, especially when there are many layers involved. Q-Networks with many layers of neurons can train much faster and are called *Deep Q-Networks*. However, Q-Networks have trouble when we have a continuous action space because they compute the maximum expected future reward for each possible action at each time step given some state. But when we have infinite possibilities of actions, computing the maximum expected future reward for each action becomes impossible.

Deterministic Policy Gradient and Actor Critic Algorithms

The *policy gradient* is one of the most popular methods of continuous variable learning algorithms. Instead of estimating a value function which becomes expensive with many actions, we learn directly the policy function that maps state to action. This means that we no longer have to optimize the state s and a, and save their results. The idea with policy gradient methods is to adjust the parameters θ toward the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$. The fundamental result underlying this idea is called the *policy gradient theorem* [8],

$$\nabla_\theta J(\pi_\theta) = \int_S p^\theta(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a \mathrm{d}s$$

$$= \mathbb{E}_{s\tilde{}p^\pi, a\tilde{}\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)]$$

This formula is simpler than Q-Networks because the policy gradient does not depend on the gradient of the state distribution which reduces the computation of the performance gradient to a simple approximation. Once we can measure the quality of our policy π we can use *gradient ascent* to maximize the expected reward of our performance gradient ($\nabla_p iJ(\pi_\theta)$). Because of the reduced computation, policy gradient algorithms learning is more stable than Q-Networks. Convergence is also a guarantee, whether its a local maximum (worst case) or global maximum (best case). The most important fact of policy gradients is that it is possible to learn in an environment with a continuous action space which makes policy gradients fundamental in learning in continuous spaces.

The *actor-critic* is a widely used architecture that combines ideas from Q-Networks and the policy gradient theorem [9]. The actor-critic contains two separate neural networks, an *actor* and a *critic*, that work together to learn an environment. The actor controls how the agent behaves by adjusting the parameters θ of the policy $\pi_\theta(s)$ by gradient ascent, similarly to the policy gradient theorem. However, instead of the unknown true action-value function $Q^\pi(s,a)$, the action-value function $Q^w(s,a)$ is used, with the parameter w. The critic measures how good the action taken by the agent is by estimating the action-value function $Q^w(s,a) \approx Q^\pi(s,a)$ using an evaluation algorithm such as temporal-difference learning *at each time step t*, similarly to how Q-Networks learn. Because we have two neural networks, we have two parameters θ and w that must be optimized separately in parallel for each of the neural networks,

$$\Delta\theta = \alpha\nabla_\theta(\log \pi_\theta(s,a))\hat{q}_w(s,a),$$

$$\Delta w = \beta(R(s,a) + \gamma\hat{q}_w(s_{t+1},a_{t+1}) - \hat{q}_w(s_t,a_t))\nabla_w\hat{q}_w(s_t,a_t),$$

where α and β are separate learning rates. The actor-critic model has many advantages over the policy gradient method. For one, learning is more stable and faster because the parameters are updated at each time step with TD learning instead of at the end of each episode. Another issue with policy gradients is that it takes the average reward over every step in an episode. That means it could identify an episode as good even if there were some bad actions because the total reward was extracted. With the actor-critic model, each action the actor takes is critiqued individually so it takes less episodes to converge on the optimal policy.

Modern computers can handle the computation of *deep reinforcement learning*, termed deep because of the many layers used in neural networks, much better than Q-learning because of recent techniques in parallelization and GPU matrix multiplication [7]. Furthermore, the framework of neural networks is highly flexible because of activation functions and can be adapted to advanced RL techniques such as Q-Networks, the policy gradient theorem, and actor-critic models. The continuous nature of neural networks also allow us to practice RL with *continuous variable quantum computing* to train large RL algorithms much quicker.

Continuous Variable Quantum Computing

Strawberry fields

Proposed Study

References

[1]: R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, 1998).

[2]: Li Yuxi, *Deep Reinforcement Learning* (????, 2018).

[3]: Melo, Francisco S, *Convergence of Q-learning: a simple proof* (ISR).

[4]: C. J. Watkins and P. Dayan, Machine Learning 8,279 (1992).

[5]: R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction. Second Edition.* (MIT Press, 1998) pages 161-162.

[6]: V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, ArXiv Preprint ArXiv:1312.5602 (2013).

[7]: Nvidia, *GPU-Based Deep Learning Inference: A Performance and Power Analysis*, (Nvidia, 2015)

[8]: R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, *Policy gradient methods for reinforcement learning with function approximation*, (Neural Information Processing Systems, 1999) pages 1057-1063.

[9]: J. Peters, S. Vijayakumar, S. Schaal, *Natural actor critic* (16th European Conference on Machine Learning, 2005) pages 180-291.