Deep Q Learning

Joseph Engler, PhD

This article extends our knowledge of Reinforcement Learning to those environments in which tabular quantification of the state, action reward tuple is intractable. In almost all, except the most trivial, environments, that are encountered by Reinforcement Learning practitioners, the number of states and action pairs is extremely large or possibly infinite. Thus maintaining a tabular record of each state action pair is simple not feasible. For such environments, the Q function must be approximated by some other method. Fortunately, a universal function approximation methodology is readily available to us in the form of deep neural networks. This article will focus on the use of deep learning to approximate the Q function for a reinforcement learning problem.

While not the focus of this article, consider, for the moment, video games such as the Atari classic, Breakout. Breakout consists of a screen with a a set of rows of blocks at the top of the screen, a paddle controlled by the user at the bottom of the screen, and a 'ball' that interacts with the paddle and the blocks as shown in Figure 1. Each frame of the game can be considered a new state. Thus, when the 'ball' moves even a single pixel a new state is encountered. Clearly, this would result in a very large number of states, certainly, more than would be feasible to maintain in a lookup table of manageable size.

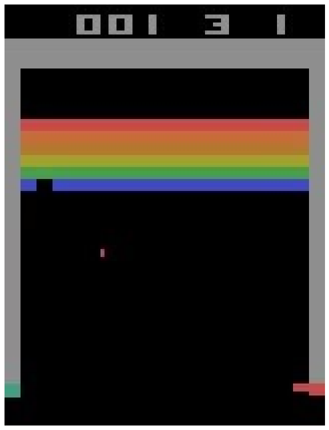

Figure 1: Sample Atari Breakout Screen

Due to the immense size of the table that would be required to be able to apply the techniques of tabular Q learning, a different methodology must be found. A celebrated paper, authored by a team of researchers from Google's DeepMind, illustrated that deep learning is a highly capable replacement methodology (Mnih et al., 2013, 2015). Deep learning is known as being capable of universal function approximation and as such is desirable to use for approximating the Q function of systems whose environments exhibit a large number of states (Schaul, Horgan, Gregor, & Silver, 2015).

According to (Lecun, Bengio, & Hinton, 2015), deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. At each layer in the system a set of mathematical functions are performed on the parameters of that layer. The parameters of each layer consist of a weight tensor and a bias vector. Deep learning consists of two phases for learning the representation of the data. First, the network is fed input data which flows through the network interacting with the parameters of each layer and undergoing mathematical transformations in the form of activation functions. Second, a loss is calculated between the output of the feed forward step and the ground truth output value associated with the input. The loss is then back propagated through the network to update the parameters of the network ensuring that future losses will slowly be minimized.

Figure 1, below, describes a typical feed forward neural network consisting of the input values, weights and biases at each layer, and output values. Each non-bias line in Figure 1 represents a value in a weight tensor. During the feed forward step, the values of the previous layer, or the input values as the case may be, are multiplied by the weight tensor, as shown. Those values are then summed up for each node in the current layer and the bias value for that node is added. Assume that h_{ij} is the i^{th}node in the j^{th} layer, then the value of h_{ij} is given as in Eq. 1.

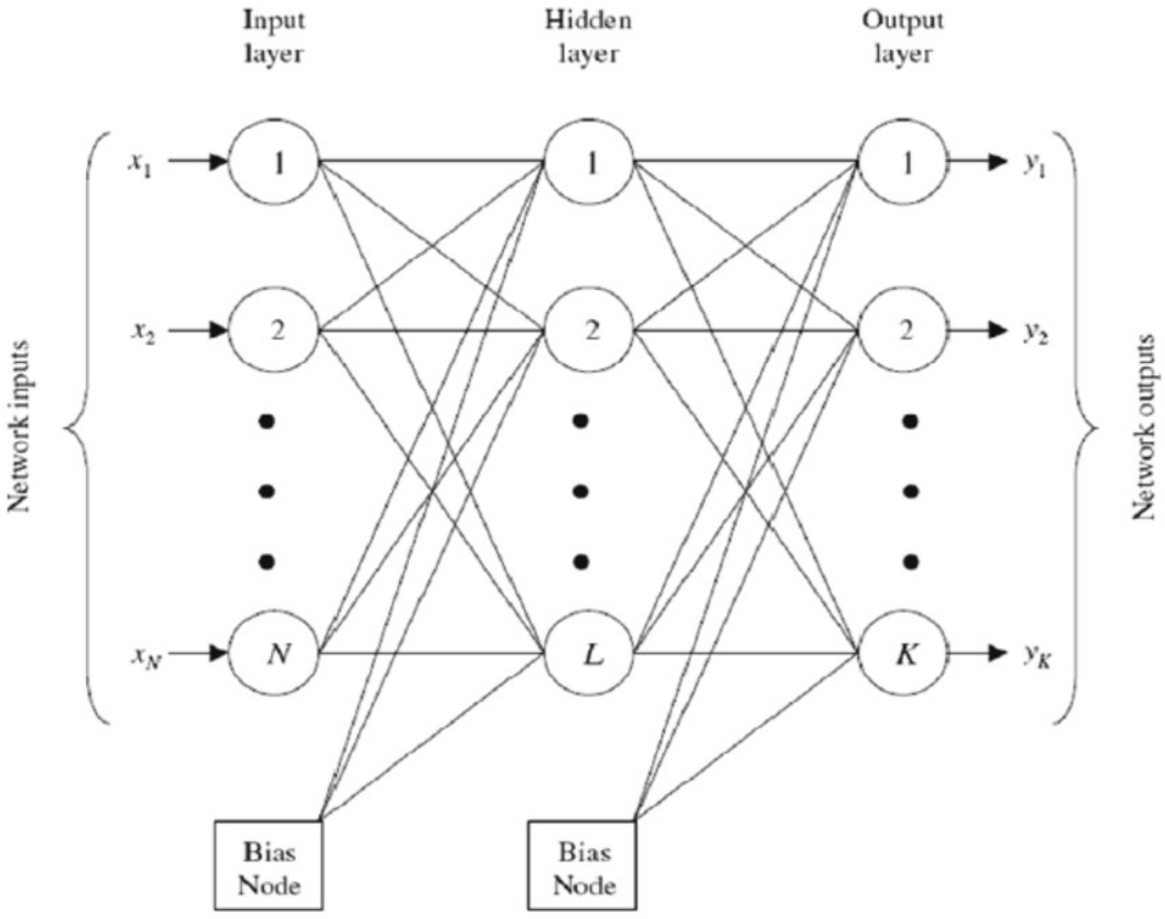

Figure 1: Example Feed Forward Deep Learning Neural Network

$$h_{ij} = \sum_{i=0}^{N} X_i W_i + b_i \qquad \text{Eq. 1}$$

The value of h_{ij} is then subjected to a mathematical transformation in the form of a, activation function such as the sigmoid or hyperbolic tangent function, as shown in the example of Eq. 2, whose resultant value then becomes either one of the inputs to the following layer, or one of the output values.

$$z_i = \tanh(h_{ij}) \qquad \text{Eq. 2}$$

Upon completing a pass through the network, as described above, the loss for the resultant output is calculated. Many different loss functions may be used depending on the type of network and the type of output that is expected. As reinforcement learning, and more to the point Q learning, is concerned with choosing the appropriate action, the output in this can be considered a vector of the probabilities of taking one of the actions. In that case, it is customary to use the cross entropy method for calculating the loss. Due to the fact that the outputs, in this case, are probabilities it is implied that the mathematical transformation of the output layer is most likely the Softmax function given in Eq. 3 below.

$$f(h_i) = \frac{e^{h_i}}{\sum_{k=0}^{n} e^{h_k}} \qquad \text{Eq. 3}$$

The Cross Entropy loss of the network's output and the target vector, *T*, is calculated as negative the sum of the products of the elements of *T* and the elements of the output vector. However, the ground truth vector, *T*, consists of a number of zero values and a single 1 value which represents the true action that should have been predicted. As such, it is possible to discard all the products except the element of *T* that has a value of 1. Given that information, Eq. 4 defines the Cross Entropy loss value.

$$CE = -\log\left(\frac{e^{h_i}}{\sum_{k=0}^{n} e^{h_k}}\right) \qquad \text{Eq. 4}$$

Once Cross Entropy loss is calculated, the gradient of that loss is determined and used, along with a learning rate, to determine the amount of change to be exerted on the weights and biases at each layer. The gradient of the loss function is calculated using partial derivatives. Most modern machine learning platforms, such as Tensorflow and PyTorch, include auto differentiation so a mathematical treatment of the determination of the gradient will not be given herein. However, (Goodfellow, Bengio, & Courville, 2016) have given an excellent treatment of the topic in their popular book, Deep Learning.

Deep Learning neural networks are highly appropriate for approximating the Q function. When using deep learning to approximate the Q function, it is standard practice to say that the Q function is parameterized by the weights and biases, θ, of the deep learning system. The purpose of Deep Learning then, with respect to the Q function, is to learn the best action to execute given the state of the environment. If this sounds familiar, it should. This is the very basis of Reinforcement Learning and was discussed extensively in the last article on Dynamic Programming.

To achieve the goal of selecting the optimal action for each state, Deep Q Learning utilizes a Bellman operator similar to that used in dynamic programming. Given a state (*s*), action (*a*), reward (*r*), and next-state (*s'*) tuple the Bellman operator update is given as in Eq. 5 for the Q function parameterized by the weights and biases, θ, of the deep learning system, where α is the learning rate of the updates and γ is the discount factor as discussed in the previous article. Thus, the Q value for the state-action pair (*s,a*) is only updated by a small factor (α) at each update.

$$Q^{\theta}(s,a)=(1-\alpha)Q^{\theta}(s,a)+\alpha(r+\gamma max_a Q^{\theta}(s',a)) \quad \text{Eq. 5}$$

Given a deep learning neural network model for determining the action at any given state, the first half of Eq. 5 can be easily described by the use of the neural network. The state, *s*, represented by a vector of values, is fed into the network. The network produces an an output a vector of action probabilities. The element of the vector with the highest probability is the action the Q function, approximated by the network, currently believes is the most optimal to execute. The agent would execute that action and receive a reward for the action, resulting in entering a new state, *s'*.

The second half of Eq. 5, choosing the maximum action for the new state, is not quite a clear when using deep learning to approximate the Q function. In some cases, the same network that was used to predict the action in the previous state is also used to predict the optimal action in the new state. However, as will be seen in the following article, this can cause issues in which the network does not learn the Q function well and ends up '*chasing its own tail*'. For the purposes of this article, though, we will utilize the same network to predict both the action for the current state and for the new state.

In addition to using deep learning to approximate the Q function, Deep Reinforcement Learning often utilizes a replay buffer of experiences on which the training of the neural network is performed. A replay buffer is simply a memory mechanism for holding the state, action, reward, next-state tuples. After each action is executed, the tuple for that action is added to the replay buffer. Once the replay buffer has reached a sufficient size, a batch of random samples are selected from the replay buffer and are used to train the network.

For each tuple in the randomly selected batch, the deep learning network utilizes the state of the that tuple as the input to the network and the target action for the new state as the example's ground truth or output label. The network is then trained for a single epoch. This can be a bit tricky to understand at first, so the following snippet of python code is given to help illustrate the technique.

```
minibatch = random.sample(memory, batch_size)
for state, action, reward, next_state, done in minibatch:
    target = reward
    if not done:
        target = (reward + gamma *  np.amax(model.predict(next_state)[0]))
      target_f = model.predict(state)
      target_f[0][action] = target
      model.fit(state, target_f, epochs=1, verbose=0)
```

The model in the above code snippet can be defined however the user determines is best for the problem they are attempting to solve. Using the above methodology has been shown to converge for standard Reinforcement Learning problems (with the previously mentioned caveat of reusing the same network for the prediction of the best action to take at the next state).

Deep Q Learning Example

An example of Deep Q Learning is given in the code below for OpenAI Gym ‘s Cart-Pole environment (OpenAI, n.d.). The Cart-Pole environment simulates a cart, on a track, with an upright pole hinged at its base. The actions of this environment are to move the cart to the right or to the left with the explicit goal being to keep the pole from tipping past a given angle θ. This is shown in Figure 2 below. A reward of 1 is given for each step (action step) that the pole remains upright (to within an angle of θ). The episode ends when the pole tips past the angle θ and the total reward for the episode is the total number of steps prior to the episode ending.

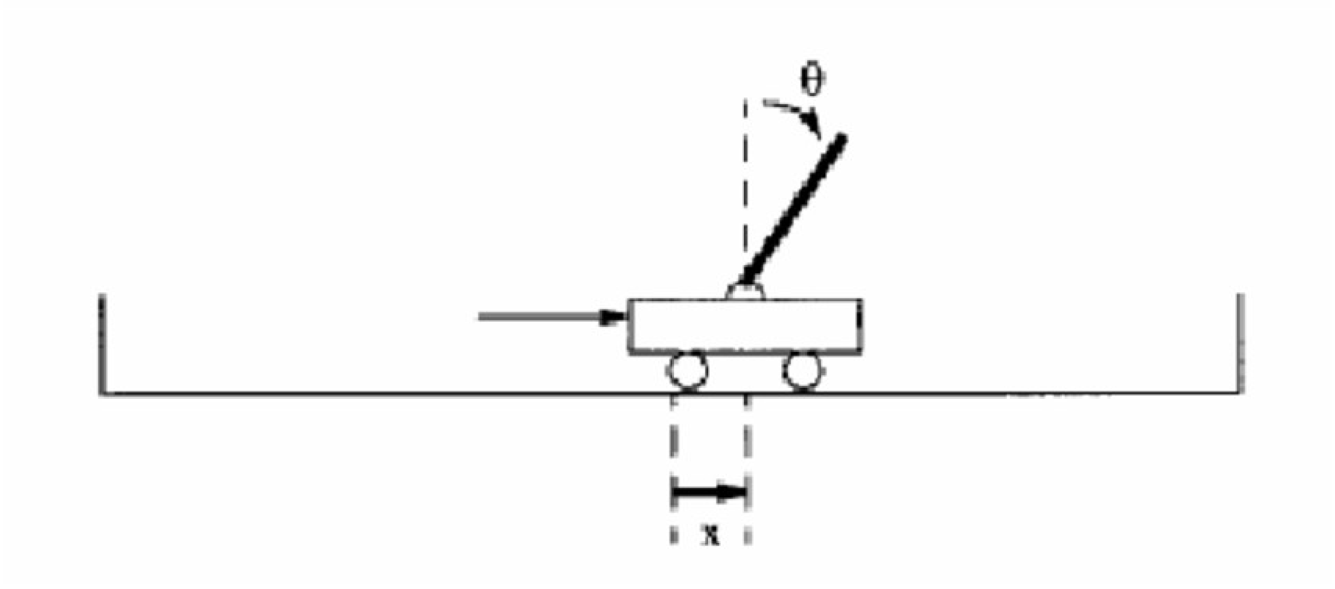

Figure 2: Cart Pole Diagram

Deep Q Learning can take a large number of episodes to properly learn the approximation of the Q function. In the case of the Cart Pole environment it is common to perform 1000 episodes to ensure the network has properly trained. As with tabular Q Learning, the agent begins by exploring the environment through randomly selected actions. As the network better approximates the Q Learning function, the exploration rate is reduced and the algorithm begins to exploit its learning by utilizing the network predicted actions rather than randomly selected actions.

The deep learning model used in this example is actually quite simple. Each state is represented by a vector of length 4. Thus, the input vector to the network is of length 4. There are two hidden layers of width 24 followed by an output layer of length 2 representing the two possible actions. This equates to a total of 770 trainable parameters for the network. The replay buffer, in the given example, is represented with a queue, or FIFO (first in, first out) structure, with a maximum size of 2000 tuples.

For each step in an episode, the agent selects an action for the given state, executes that action, receives a reward and the next state. That tuple is then stored in the replay buffer. Once the replay buffer reaches a minimum level (given as batch size in the example) the network is trained on a randomly selected batch of tuples. At the end of each episode the total score for the agent is printed. The code for this example is given below.

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

EPISODES = 1000

class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        model.summary()
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])  # returns action

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = (reward + self.gamma *
                          np.amax(self.model.predict(next_state)[0]))
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def load(self, name):
        self.model.load_weights(name)

    def save(self, name):
        self.model.save_weights(name)


if __name__ == "__main__":
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    agent = DQNAgent(state_size, action_size)
    done = False
    batch_size = 32

    for e in range(EPISODES):
        state = env.reset()
        env.render()
        state = np.reshape(state, [1, state_size])
        for time in range(500):
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            env.render()
            reward = reward if not done else -10
            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            if done:
                print("Attempt: {} of{}, score: {}"
                      .format(e, EPISODES, time))
                break
            if len(agent.memory) > batch_size:
                agent.replay(batch_size)
```

Works Cited

Bellman, R. (1954). Some Problems in the Theory of Dynamic Programming. *Econometrica*, *22*(1), 37. https://doi.org/10.2307/1909830

Blackwell, D. (1965). Discounted Dynamic Programming. *The Annals of Mathematical Statistics*, *36*(1), 226–235. https://doi.org/10.1214/aoms/1177700285

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Greaves, J. (2017). Understanding RL: The Bellman Equations. Retrieved September 20, 2010, from https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/

Howard, R. A. (1958). *Studies in Discrete Dynamic Programming*. 115.

Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv Preprint*, 1–9. Retrieved from http://arxiv.org/abs/1312.5602

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

OpenAI. (n.d.). OpenAI Gym. Retrieved from https://gym.openai.com/

PyPi. (n.d.). Retrieved from https://pypi.org/project/gym/0.7.4/

Schaul, T., Horgan, D., Gregor, K., & Silver, D. (2015). Universal value function approximators. *32nd International Conference on Machine Learning, ICML 2015*, *2*, 1312–1320.

Sutton, R. S., & Barto, A. G. (2014). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Tsitsiklis, J. N., & Van Roy, B. (2002). On average versus discounted reward temporal-difference learning. *Machine Learning*, *49*(2–3), 179–191.