Deep Q-Learning with Keras

环境定义

```
next_state, reward, done, info = env.step(action)
```

Agent在环境中执行Action，环境反馈执行后的状态、奖励等。

`done` is a Boolean value telling whether the game ended or not. The old `state` information paired with `action` and `next_state` and `reward` is the information we need for training the agent.

Implementing Simple Neural Network using Keras

```
# Neural Net for Deep Q Learning

# Sequential() creates the foundation of the layers.
model = Sequential()

# 'Dense' is the basic form of a neural network layer
# Input Layer of state size(4) and Hidden Layer with 24 nodes
model.add(Dense(24, input_dim=self.state_size, activation='relu'))
# Hidden layer with 24 nodes
model.add(Dense(24, activation='relu'))
# Output Layer with # of actions: 2 nodes (left, right)
model.add(Dense(self.action_size, activation='linear'))

# Create the model based on the information above
model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
```

This training process makes the neural net to predict the reward value from a certain `state`.

```
model.fit(state, reward_value, epochs=1, verbose=0)
```

After training, the model now can predict the output from unseen input. When you call `predict()` function on the model, the model will predict the reward of current state based on the data you trained. Like so:

```
prediction = model.predict(state)
```

Implementing Mini Deep Q Network (DQN)

The loss is just a value that indicates how far our prediction is from the actual target. We want to decrease this gap between the prediction and the target (loss). We will define our loss function as follows:

$$loss = (Target - Prediction)^2 Target = r + \gamma \max_{a'} \hat{Q}(s, a') Prediction = Q(s, a) loss = \left(r + \gamma \max_{a'} \hat{Q}(s, a') - Q(s, a) \right)^2$$

首先，执行动作 a，并观察奖励 r 和新状态 s'。然后，计算最大目标Q值，并考虑折扣因素。最后，加上当前奖励，得到目标值；减去当前的预测得到loss。平方值使较大的loss值变大，且将负值作为正值对待。

定义目标：

```
target = reward + gamma * np.amax(model.predict(next_state))
```

Keras does all the work of subtracting the target from the neural network output and squaring it. It also applies the learning rate we defined while creating the neural network model. This all happens inside the `fit()` function. This function decreases the gap between our prediction to target by the learning rate. The approximation of the Q-value converges to the true Q-value as we repeat the updating process. The loss will decrease。

Memorize

在神经网络中，算法会以新的经验覆盖之前的经验。因此，就需要用之前的经验再次进行模型训练。在DQN中，经验包括的内容有当前状态、动作、奖励、下一时刻状态。
One of the challenges for DQN is that neural network used in the algorithm tends to forget the previous experiences as it overwrites them with new experiences. So we need a list of previous experiences and observations to re-train the model with the previous experiences. We will call this array of experiences `memory` and use `memorize()` function to append state, action, reward, and next state to the memory.

In our example, the memory list will have a form of:

```
memory = [(state, action, reward, next_state, done)...]
```

And memorize function will simply store states, actions and resulting rewards to the memory like below:

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

`done` is just a Boolean that indicates if the state is the final state.

Replay

A method that trains the neural net with experiences in the `memory` is called `replay()`. First, we sample some experiences from the `memory` and call them `minibath`.

```
minibatch = random.sample(self.memory, batch_size)
```

The above code will make `minibatch`, which is just a randomly sampled elements of the memories of size `batch_size`. We set the batch size as 32 for this example.

To make the agent perform well in long-term, we need to take into account not only the immediate rewards but also the future rewards we are going to get. In order to do this, we are going to have a 'discount rate' or 'gamma'. This way the agent will learn to maximize the discounted future reward based on the given state.

```
# Sample minibatch from the memory
minibatch = random.sample(self.memory, batch_size)

# Extract informations from each memory
for state, action, reward, next_state, done in minibatch:

    # if done, make our target reward
    target = reward

    if not done:
      # predict the future discounted reward
      target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])

    # make the agent to approximately map
    # the current state to future discounted reward
    # We'll call that target_f
    target_f = self.model.predict(state)
    target_f[0][action] = target

    # Train the Neural Net with the state and target_f
    self.model.fit(state, target_f, epochs=1, verbose=0)
```

How The Agent Decides to Act

Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward. `np.argmax()` is the function that picks the highest value between two elements in the `act_values[0]`.

```python
def act(self, state):
    if np.random.rand() <= self.epsilon:
        # The agent acts randomly
        return env.action_space.sample()

    # Predict the reward value based on the given state
    act_values = self.model.predict(state)

    # Pick the action based on the predicted reward
    return np.argmax(act_values[0])
```

`act_values[0]` looks like this: [0.67, 0.2], each numbers representing the reward of picking action 0 and 1. And argmax function picks the index with the highest value. In the example of [0.67, 0.2], argmax returns **0** because the value in the 0th index is the highest.

Hyper Parameters

- `episodes` - a number of games we want the agent to play.
- `gamma` - aka decay or discount rate, to calculate the future discounted reward.
- `epsilon` - aka exploration rate, this is the rate in which an agent randomly decides its action rather than prediction.
- `epsilon_decay` - we want to decrease the number of explorations as it gets good at playing games.
- `epsilon_min` - we want the agent to explore at least this amount.
- `learning_rate` - Determines how much neural net learns in each iteration.

Coding DQN Agent

```python
# Deep Q-learning Agent
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))
        return model

    def memorize(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
```

Train the Agent

```
if __name__ == "__main__":

    # initialize gym environment and the agent
    env = gym.make('CartPole-v0')
    agent = DQNAgent(env)

    # Iterate the game
    for e in range(episodes):

        # reset state in the beginning of each game
        state = env.reset()
        state = np.reshape(state, [1, 4])

        # time_t represents each frame of the game
        # Our goal is to keep the pole upright as long as possible until score of 500
        # the more time_t the more score
        for time_t in range(500):
            # turn this on if you want to render
            # env.render()

            # Decide action
            action = agent.act(state)

            # Advance the game to the next frame based on the action.
            # Reward is 1 for every frame the pole survived
            next_state, reward, done, _ = env.step(action)
            next_state = np.reshape(next_state, [1, 4])

            # memorize the previous state, action, reward, and done
            agent.memorize(state, action, reward, next_state, done)

            # make next_state the new current state for the next frame.
            state = next_state

            # done becomes True when the game ends
            # ex) The agent drops the pole
            if done:
                # print the score and break out of the loop
                print("episode: {}/{}, score: {}".format(e, episodes, time_t))
                break

        # train the agent with the experience of the episode
        agent.replay(32)
```