Deep Q Network

Denver Quant Society

Yuki Kitayama

Contents

Introduction

Deep Q Network (DQN)

- One of many reinforcement learning algorithms using deep learning (**“Deep reinforcement learning”**)
- Proposed in 2013 by DeepMind (AI company in UK under Google) in paper “Playing Atari with Deep Reinforcement Learning”.
- **Neural network** is used to process state data and to output action-value (Q value)

Why I introduce this?

- Agent **learns from scratch** without human intervention and achieves results **better than human**.
- Many reinforcement learning papers use DQN as **benchmark**.

Example

Space Invaders

- A game to kill as many enemies as possible to get a high score.
- State: Game screen
- Action: Fire, move right, move left, move right and fire, move left and fire, do nothing
- Reward: Score you can get when you kill an enemy

Approach

- Use convolutional neural network to process game screen, and outputs what’s the best in a certain state every time step.

YouTube

- https://www.youtube.com/watch?v=W2CAghUiofY

Concept - Big picture

For loop

- Get action ← **<u>Epsilon greedy algorithm</u>**
- Get next state from the action
- Collect data ← **<u>Experience replay</u>**
- Train model ← **<u>Q-learning</u>**
- Current state = next state

Concept - Epsilon greedy algorithm

Idea

- Model outputs might be a locally optimal, so let the agent ignore the model, take a different action, and explore an environment.

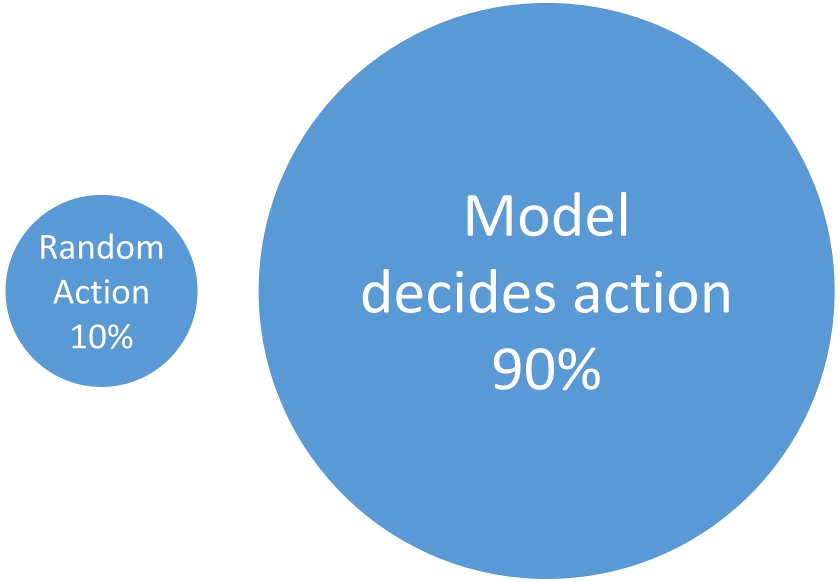
Benefit

- Through the experience, we force the agent to learn.

Algorithm

- Set a value (ε) between 0.0 and 1.0
    - Generate random number between 0.0 and 1.0
        - If the random number is smaller than ε,
            - Take a random action
        - If larger,
            - Take an action from the model

Epsilon = 0.1



Concept - Experience replay

Idea

- It's a buffer (called **replay memory**) of experienced data which we collect through iteration.
- Becomes a dataset to train our model.

Benefit

- Efficient because it stores a large amount of data, but sample mini-batch to train model.
- Reduce data correlation through sequentially collecting experience but randomly sampling it

Algorithm

- In each time step, collect a tuple of state, action, reward, and next state
- Add them to a list of replay memory.
- Generate random indices and batch size to sample data
- Train neural network.

Replay memory (size = 1M)

$[State_{t=1}, Action_{t=1}, Reward_{t=1}, Next\ state_{t=1}]$

$[State_{t=2}, Action_{t=2}, Reward_{t=2}, Next\ state_{t=2}]$

…

↓

Mini-batch (size = 32)

$[State_{t=24}, Action_{t=24}, Reward_{t=24}, Next\ state_{t=24}]$

$[State_{t=73}, Action_{t=73}, Reward_{t=73}, Next\ state_{t=73}]$

…

↓

Train neural network

Concept - Q-learning

Model

- Use neural network to predict action-values, Q(s, a), (“**Q value**”)
- Shape of Input and output depends on an environment.

Q-learning

- Target
    - Reward + discount factor * max(Q(next state, action))
- Estimate
    - Q(current state, action)

Update

- Loss function is mean of (Target - Estimate)^2
    - E [r + gamma * max(Q(s’,a’)) – Q(s,a; theta)]
- Differentiate the loss function w.r.t. theta
- Gradient descent and update weights in neural network.
    - Theta = theta – alpha * gradient of theta.

State

Neural network

Action	Q value
Fire	5.5
Right	4.3
Left	8.6
Right & fire	2.1
Left & fire	2.5
Nothing	4.6

Action

Move left

Implementation - Code

OpenAI Gym CartPole

- State: Cart position, cart velocity, pole angle, pole velocity
- Action: Move cart right, move cart left
- Reward: Every time step, you gain 1.
- Environment ends when cart moves away from center or when pole loses standing state.
- Video of real setting: https://www.youtube.com/watch?v=XiigTGKZfks&feature=youtu.be
- Code:
  - Python: https://github.com/yukikitayama/reinforcement-learning/blob/master/dqs/dqn_cartpole_train.py
  - Notebook: https://github.com/yukikitayama/reinforcement-learning/blob/master/dqs/dqn_cartpole.ipynb

Epsilon greedy algorithm

```python
def get_action(state, num_actions, model, epsilon):
    if np.random.random() < epsilon:
        return np.random.choice(num_actions)
    else:
        return np.argmax(model(np.atleast_2d(state.astype('float32')))[0])
```

Experience replay

```python
class ExperienceReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = {'state': [],
                       'action': [],
                       'reward': [],
                       'next_state': [],
                       'done': []}
    def size(self):
        return len(self.buffer['state'])

    def store(self, experience_dict):
        if self.size() >= self.capacity:
            for key in self.buffer.keys():
                self.buffer[key].pop(0)
        for key, value in experience_dict.items():
            self.buffer[key].append(value)
```

Implementation - Code

Q-learning

```python
def get_model(num_states, num_actions):
    inputs = Input(shape=(num_states,))
    x = Dense(256, activation='relu')(inputs)
    x = Dense(256, activation='relu')(x)
    outputs = Dense(num_actions, activation='linear')(x)
    model = Model(inputs, outputs)
    return model
```

```python
def update_model(model, target_model, memory, optimizer,
                 batch_size, gamma, num_actions):
    index = np.random.randint(low=0, high=memory.size(), size=batch_size)
    states = np.asarray([memory.buffer['state'][i] for i in index])
    actions = np.asarray([memory.buffer['action'][i] for i in index])
    rewards = np.asarray([memory.buffer['reward'][i] for i in index])
    next_states = np.asarray([memory.buffer['next_state'][i] for i in index])
    dones = np.asarray([memory.buffer['done'][i] for i in index])

    next_state_action_values = np.max(target_model(next_states), axis=1)
    target_values = np.where(dones,
                             rewards,
                             rewards + gamma * next_state_action_values)

    with tf.GradientTape() as tape:
        action_values = tf.math.reduce_sum(
            model(np.atleast_2d(states.astype('float32'))) * tf.one_hot(actions, num_actions),
            axis=1
        )
        # Get loss function
        loss = tf.math.reduce_mean(tf.square(target_values - action_values))
    # Get weights
    variables = model.trainable_variables
    # Get gradients
    gradients = tape.gradient(loss, variables)
    # Gradient descent
    optimizer.apply_gradients(zip(gradients, variables))
```

Implementation - Result

Result