CS4619: Artificial Intelligence II

Reinforcement Learning, Again

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

In [3]: # This code comes from: http://mckinziebrandon.me/TensorflowNotebooks/2016/12/21/openai.html
   # You may need to install JSAnimation, e.g.: pip install git+https://github.com/jakevdp/JSAnimation.git
   from JSAnimation.IPython_display import display_animation
   from matplotlib import animation
   from IPython.display import display

   def display_frames_as_gif(frames):
       """ Displays a list of frames as a gif, with controls """
       patch = plt.imshow(frames[0])
       plt.axis('off')

       def animate(i):
           patch.set_data(frames[i])

       anim = animation.FuncAnimation(plt.gcf(), animate, frames = len(frames), interval=50)
       display(display_animation(anim, default_mode='once'))
Reinforcement Learning

- There are many algorithms for RL, but the one we studied previously was $Q$-Learning:

\[
\text{QLearning}(\epsilon)
\]

\[
\begin{align*}
\text{s} &= \text{SENSE}(); \\
\text{do forever} &\quad \text{end do}
\end{align*}
\]

\[
\begin{align*}
\circ \quad \text{rand} &= \text{a randomly-generated number in}[0, 1); \\
\circ \quad \text{if rand} < \epsilon &\quad \text{else}
\end{align*}
\]

\[
\begin{align*}
\circ \quad a &= \text{arg max}_a Q(s, a); \\
\circ \quad r &= \text{EXECUTE}(a); \\
\circ \quad s' &= \text{SENSE}(); \\
\circ \quad Q(s, a) &= r + \gamma \times \text{max}_{a'} Q(s', a'); \\
\circ \quad s &= s';
\end{align*}
\]

- The problem with algorithms such as $Q$-Learning is the table, which has an entry for every state paired with every action
  - This does not scale well to problems that have many states and/or many actions
  - It fails to generalise: similar states probably have similar $Q$-values
- The state-of-the-art solution is to use a deep neural network to represent and learn the $Q$-values

Deep $Q$-Learning

- In Deep $Q$-Learning we use a deep neural network to represent and learn the $Q$-values
- We're going to look at one way of doing this: a DQN (Deep $Q$-Network)
- A DQN predicts $Q$-values (regression!)
- One way of doing this:
  - Input layer takes in a state and an action
  - Output layer has just one neuron, which outputs the $Q$-value
  - To choose which action to take in a state, we must activate the network repeatedly: once for each action
- Another way of doing this:
  - Input layer takes in just a state
  - Output layer has one neuron per action to output their $Q$-values
  - To choose which action to take in a state, we activate the network just once and choose the action for the output neuron with the largest activation

This is what DQNs do
DQN

- Pseudocode (discussed in subsequent slides, below):

```plaintext
Deep-Q-Learning(c)

- Initialize replay memory: \( D = [ ] \)
- \( s = SENSE(); \)
- do forever
  - \( \text{random} = \) a randomly-generated number in \( [0, 1); \)
  - if \( \text{random} < c \)
    - Choose action \( a \) randomly;
  - else
    - \( a = \arg \max_a Q(s, a); \)
  - \( r = \text{EXECUTE}(a); \)
  - \( s' = SENSE(); \)
  - Store \( \langle s, a, r, s' \rangle \) in \( D \)
  - Randomly choose a mini-batch \( X \) of examples from \( D \)
  - For each \( \langle s, a, r, s' \rangle \in X \), calculate the target value, i.e. \( r + \gamma \max_{a'} Q(s', a') \)
  - Train the network on mini-batch \( X \) (one iteration only) using loss function
    \[
    \frac{1}{2} \sum_{(s,a,r,s') \in X} \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]^2
    \]
  - \( s = s'; \)
```

- Keep in mind that the \( Q \)-values above (\( Q(s, a) \) and \( Q(s', a') \)) are predicted by the neural network

### Exploration vs. Exploitation

- We discussed this previously
- The \( \epsilon \)-greedy policy is a simple solution

### DQN's loss function

- In everything we have done in previous lectures, the target values are fixed before learning begins
- In DQNs, the targets change!
- Consider a transition \( \langle s, a, r, s' \rangle \):
  - Suppose the agent is in state \( s \)
  - It feeds \( s \) into the network; the output neurons produce \( Q \)-values for each action
  - These \( Q \)-values are the predictions
  - It chooses the action \( a \) that has highest \( Q \)-value: \( \arg \max_a Q(s, a) \)
  - It executes \( a \), obtaining reward \( r \) and transitioning to state \( s' \)
  - It feeds \( s' \) into the network; the output neurons produce \( Q \)-values for each action
  - For action \( a \) (the one we chose), \( r + \gamma \max_{a'} Q(s', a') \) is a better estimate and so this is the target
  - For the other actions (the ones we did not choose), the target is the same as the prediction (error is 0)
- We can use mean-squared-error for the loss function:
  - The square of the difference between target and prediction
  - In effect, the loss is:
    \[
    \frac{1}{2} \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]^2
    \]
Experience replay

- The previous slide implies that, in each sense-plan-act cycle, we train on just one example
  - i.e. for transition \( \langle s, a, r, s' \rangle \), we described the target value and prediction
- But this takes ages to converge
- Instead, DQNs use experience replay:
  - All the experiences \( \langle s, a, r, s' \rangle \) are stored in a so-called ‘replay memory’
  - When training, the DQN chooses a random mini-batch from the replay memory

Further tweaks

- The presentation above simplifies the algorithm a little
- It ignores an extra if-statement for handling the situation where the loop terminates (e.g. when a game is over)
- More importantly, DQNs include lots of other ‘tricks’ to improve convergence, including:
  - \( c \) decreases, e.g. from 1 to 0.1 over time
  - The use of two neural networks (one for predicting targets and one for everything else)
    - Weights from the second one are copied into the first on periodically
  - Error clipping, reward clipping, …
- We won’t concern ourselves with these in CS4619!

DQNs for Atari Games

- The DQN algorithm was developed by a company called DeepMind (now owned by Google)
- In 2013/2014, they trained DQNs to play Atari 2600 video games
- States of the game
  - You might think that the state of the game is represented using a game-specific data structure
  - The cool part: instead, the state of the game is an image (array of pixels) — the same as you see on the screen
    - This is a generic way of representing the state of these games
    - It is what makes their DQN applicable to so many different games: all you need are images from the game
      after each user action
    - It does mean that the lower layers in the neural network need to be convolutional layers — to do some image processing
  - describes seven games, outperforming humans on three of them
  - describes 49 games, outperforming humans on half of them
States as images

- If states are images then, in principle, we can find out the positions of all the objects (e.g. the bricks and paddle in Breakout)
- But we cannot find out their speed or direction of travel
- So DeepMind chose to represent a single state by four images:
  - After an action, the new state is the current screen image and the preceding three
  - They also do a little preprocessing to reduce image sizes:
    - convert RGB to grayscale
    - scale down from 210 × 160 pixels to 84 × 84
- So, if $m$ is the mini-batch size, then the input shape is $(m, 84, 84, 4)$. Why?

OpenAI Gym

- We don’t want to programme Atari games from scratch ourselves
- OpenAI Gym ([https://github.com/openai/gym](https://github.com/openai/gym))
  - provides a range of agents and environments for working on RL
  - e.g. Atari games, board games, etc. ([https://gym.openai.com/envs/](https://gym.openai.com/envs/))
- Installation of OpenAI Gym:
  - Do a minimal installation, e.g. pip install gym
  - Then install the Atari game environments, e.g. pip install ‘gym[atari]’

```python
In [4]: import gym

In [5]: # Create an environment
ev = gym.make("BreakoutDeterministic-v4")

In [6]: # Find out what actions are possible
   env.action_space
Out[6]: Discrete(4)

In [7]: # Initialise and get the initial state, s
   s = env.reset()

In [8]: # Let’s say you want to execute action 2
   action = 2
   new_state, reward, done, info = env.step(action)

In [10]: # To display:
   #env.render()
```
The demo below repeatedly:
- renders the environment, but not to the screen — to an array, which is stored in a list called `frames`
- executes a random action and gets the new state
- Afterwards, it displays the contents of `frames` as an animation

```python
In [11]: # Run a demo of the environment
    observation = env.reset()
cum_reward = 0
    frames = []
    for t in range(5000):
        # Render into buffer.
        frames.append(env.render(mode = 'rgb_array'))
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            break
display_frames_as_gif(frames)
```

Keras-RL and TF-Agents
- There are two libraries that implement DQNs and other deep learning versions of RL
- Unfortunately, I cannot get either to work: incompatibilities with various versions of tensorflow
Concluding Remarks

- The lecture gives a flavour of using deep learning for RL
- There are many more complex variants (e.g. Dueling Deep $Q$-Networks and Double Deep $Q$-Networks) including ones that lift the assumption of a finite set of actions (e.g. Actor-Critic models)
- Let's also mention AlphaGo, AlphaGo Zero and AlphaZero
  - March 2016, AlphaGo beat 18 times world champion Lee Sedol 4-1
  - October 2017, AlphaGo Zero (trained on self-play only) beat AlphaGo 100-0
  - **40 days** – AlphaGo Zero surpasses all previous versions, becomes the best Go player in the world
  - December 2017, Alpha Zero beats StockFish at chess and Elmo at Shogi (Japanese chess)

In principle, we can train Alpha Zero to play any perfect information from self-play only