Advanced neural networks: reinforcement learning

Erik Spence

SciNet HPC Consortium

13 November 2017

Today's data, code and slides

You can get the slides for today's class at the SciNet Education web page.

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage".

The code for today's class will need the 'pygame' and 'Box2D' Python packages. "pip install ..." should work. If it doesn't for Box2D, try the following commands:

```
ejspence@mycomp ~> git clone https://github.com/pybox2d/pybox2d
ejspence@mycomp ~> cd pybox2d
ejspence@mycomp ~> sudo python setup.py build
ejspence@mycomp ~> sudo python setup.py install
ejspence@mycomp ~>
```

Supervised versus unsupervised learning

One way of categorizing machine learning algorithms is based on whether they are supervised or unsupervised.

- Supervised learning means that all data comes in $(\mathbf{x}, \boldsymbol{y})$ pairs, where $\mathbf{x}$ is the input data and $\boldsymbol{y}$ is the 'label' or 'target'.
- All the neural networks we have trained so far have been supervised.
- Unsupervised learning means that you get $\mathbf{x}$ and the goal is to determine a $\boldsymbol{y}$.

There is a third category, which is the topic of today's class: semi-supervised learning.

Reinforcement learning

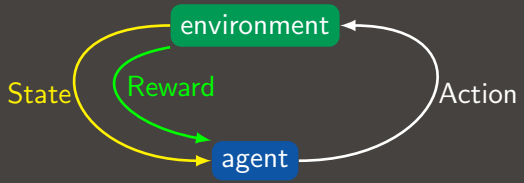
Suppose you want to teach a neural network to play a video game. There are several approaches you could use.

- You could use a supervised-learning approach, whereby you compile a data set of current states of the game, and associated 'best actions' to take given that data, and then train the NN in one of the ways we've already examined.
- But this is unsatisfying, since this is not at all how we learn to play video games in real life.
- In real life we learn to play by interacting with the game, figuring out strategies for getting points.
- This is how Reinforcement Learning (RL) works: the NN interacts with the game, and is rewarded for desirable results.
- As such, this method of training is 'semi-supervised', since rewards only arrive once in a while, and they are often time delayed.

The reinforcement learning framework

Reinforcement learning terminology:

- agent: this is you, the game player.
- environment: the game you are playing.
- state: the current state of the environment.
- action: actions the agent can perform in the environment.
- reward: positive or negative results from certain actions.
- policy: rules which dictate which actions to perform, give the state of the environment.



The actions and states, with the rules of the environment, make up a Markov decision process.

Markov decision processes

A collection of actions and changes to the environment (an "episode") forms a sequence of states (s), actions (a) and rewards (r):

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, \ldots, r_{n-1}, s_{n-1}, a_{n-1}, r_n, s_n$$

The Markov decision process relies on the Markov assumption, namely that the probability of the reaching the next state, s_{i+1}, only depends upon the current state, s_i, and the action, a_i. How we got to the current state does not matter.

Long-term success

To have the best performance in the long run, we need to consider not just the immediate rewards for any given action, but also the long-term rewards. How is this done?

Consider the total possible reward for a given number of actions:

$$R = r_1 + r_2 + r_3 + \cdots + r_{n+1} + r_n$$

The total future reward, starting from t, can be expressed as

$$R_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_{n+1} + r_n$$

Long-term success, continued

Sometimes the environment has a stochastic component to it. Consequently we can't be completely sure if we will get the same rewards the next time we perform the same actions. The farther into the future we go, the more and more it may diverge. It is common to use the "discounted future reward" to account for this:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n$$

where γ is the "discount factor", between 0 and 1. We can rewrite the discounted reward as a recursive expression

$$R_t = r_t + \gamma \left(r_{t+1} + \gamma \left(r_{t+2} + \dots \right) \right) = r_t + \gamma R_{t+1}$$

A good plan is to choose an action which maximizes the discounted future reward.

The discount factor

How do we choose a good value for the discount factor, γ?

$$R_t = r_t + \gamma R_{t+1}$$

- If we choose a small value, $\gamma \simeq 0$, then the model will be short-sighted, ignoring future rewards. This is appropriate if there is alot of randomness in the system.
- If we choose a large value, $\gamma \simeq 1$, then the model is assuming that the system is fairly deterministic, and the future can be well-predicted.

Choose a value of γ which is appropriate for the system you're dealing with.

Q-learning

Let us define a function $Q(a, s)$ which returns the maximum discounted future reward when we perform action a on state s.

$$Q(a_t, s_t) = \max R_{t+1}$$

This is the "Q function". It represents the "quality" of a certain action, given a state. You might legitimately wonder how we could know the score at the end of the game, given the current state and a given action. The truth is, we can't, but it's a useful theoretical construct.

Like the discounted future reward, we can express Q in terms of the Q value of the next state:

$$Q(a_t, s_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)$$

This is called the Bellman equation. It allows us a formulation around which to train a neural network.

Q-learning loss function

Let us create a neural network which will take the place of the Q function. Given the Bellman equation:

$$Q(a_t, s_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)$$

we can then define a loss function for our NN training:

$$L = \frac{1}{2}\left[\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)}_{\text{target}} - \underbrace{Q(a_t, s_t)}_{\text{prediction}}\right]^2$$

which is just the usual quadratic loss function.

Problems RL faces

Reinforcement learning has some challenges it needs to deal with.

- Problem 1: often, when you make a move which gets points, it has nothing to do with the immediately-preceding move. The time-delay between the move responsible for the point and receiving the point needs to be reconciled. This is called the "credit assignment problem".
- How do we deal with this? The standard approach is called "Experience Replay", which involves compiling a collection of moves, a "data set", against which to train. We choose a random mini-batch from the collection, which breaks the similarity between subsequent data points, which tends to push the network into a local minimum.

Our algorithm will create a collection of "observations" against which to train.

Problems RL faces, continued

Reinforcement learning has some challenges it needs to deal with.

- Problem 2: perhaps I come up with a strategy which is marginally effective at getting points. Should I stop there? Should I experiment to find new techniques? This is the "explore-exploit" problem.
- The standard method of addressing this problem is to throw randomness into the decision-making process. This can "encourage" the network to explore areas of the action space that it would otherwise avoid.

The collection of observations we train against starts out totally random; the randomness is reduced as the training of the network progresses.

RL example

Let's build a neural network to play a video game.

- Let's work with a modified version of the "cartpole" game. You can grab this from the class website.
- The object of the game is to move the cart back and forth to get the pendulum to swing. For each frame of the game that the pendulum is above the horizontal you get a point.
- You will need the pygame and Box2D packages for this game to run. You can try using 'pip install' for this.
- If you have trouble installing Box2D, try the following commands:

```
ejspence@mycomp ~> git clone https://github.com/pybox2d/pybox2d
ejspence@mycomp ~> cd pybox2d
ejspence@mycomp ~> sudo python setup.py build
ejspence@mycomp ~> sudo python setup.py install
ejspence@mycomp ~>
```

RL example, continued

What is the environment for this game?

- The state consists of 4 pieces of information: x, $\frac{\partial x}{\partial t}$, θ, $\frac{\partial \theta}{\partial t}$
- x is the position of the cart and θ is the angle of the pendulum.
- There are three actions: left, right, none.
- Pressing the left or right arrow keys applies a force in that direction.
- The cart and pendulum both have mass.
- There is a small amount of friction between the cart and the ground.
- Gravity points downward.

The details can be found in the cartpole file. It's pretty easy to read.

The pygame interface

If you've never worked with pygame before, it can be a bit confusing as to what is actually happening.

- First pygame is initialized.
- The cartpole object is initialized. This uses the Box2D package to create the environment (cart, pole, surface, bumpers, gravity, *etc.*).
- The initial display is plotted.
- We then enter the event loop. While the game is running:
  - when events (actions) happen, process them (move the moving parts of the game, update the score).
  - redraw the screen.
- Continue until the game is closed.

This is 'event activated' programming, which is quite different from the procedural programming you're probably accustomed to.

Computer playing cartpole

How does the computer play the game?

- First the QCartPolePlayer object is initialized.
- The usual pygame functions are then intercepted and reassigned:
  - The resetting of the screen is changed, to give the computer the opportunity to "press keys".
  - The capturing of events is changed, so that the keys which the computer "pressed" can be fed into pygame.
- The game is then run as usual.

Great. Now how do we add the Q-learning to the control of game?

Computer learning cartpole

$$L = \frac{1}{2}\left[\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)}_{\text{target}} - \underbrace{Q(a_t, s_t)}_{\text{prediction}}\right]^2$$

How do we implement Q-learning?

- We implement the function Q as a neural network.
  - The input is the state, s.
  - The output is the predicted reward for each possible action.
- A collection of "observations" is made, and stored. Each observation consists of: (previous_state, action, reward, current_state).
- To train the network:
  - Take a mini-batch of current_state observations; run them through the NN to get the predicted reward for each action, $Q(a_{t+1}, s_{t+1})$.
  - Use this to construct the right hand side of the above equation: $r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)$.
  - Use supervised learning to train Q.

Computer learning cartpole, the algorithm

What's the actual algorithm?

- Setup the cartpole game.
- Intercept the appropriate pygame commands so that the computer can play.
- Run the game with random actions to get a set of observations from which to start training the network.
- For each screen update of the game:
  - Get the reward for the last state and action.
  - Add the latest observation set to the collection of observations.
  - Train the neural network.
  - Get the next action, given the current state.

Q-learning with cartpole

```
def build_model():
  input_state = kl.Input(shape = (4,))
  input_actions = kl.Input(shape = (3,))
  x = kl.Dense(64, activation = "tanh")(input_state)
  x = kl.Dropout(0.4)(x)
  x = kl.Dense(32, activation = "tanh")(x)
  x = kl.Dense(16, activation = "tanh")(x)
  q = kl.Dense(3, activation = "relu")(x)
  action_q = kl.Dot(1)([q, input_actions])

  self.q_model = km.Model(inputs = input_state,
    outputs = q)
  self.applied_action_model = km.Model(
    inputs = [input_state, input_actions,
    outputs = action_q)

  self.applied_action_model.compile(optimizer =
    ko.SGD(lr = 1e-5, loss = "mean_squared_error")
```

```
# Q_Cartpole_Player.py, continued
def get_keys_pressed(self, reward):

  current_state = cartpole.get_state()
  self.observations.append([self.last_state,
    self.last_action, reward, current_state])

  self.train_model()

  self.last_state = current_state
  self.last_action, action_index =
    self.choose_next_action()

  if action_index == 0: action = [K_LEFT]
  elif action_index == 1: action = []
  else: action = [K_RIGHT]

  return action
```

Q-learning with cartpole, continued

```
# Q_Cartpole_Player.py, continued

# Train the NN.
def train_model():
  batch = random.sample(self.observations, self.mini_batch_size)

  previous = np.array([d[0] for d in batch]);   actions = np.array([d[1] for d in batch])
  rewards = np.array([d[2] for d in batch]);   current = np.array([d[3] for d in batch])

  expected_reward = []
  current_rewards = self.q_model.predict(current)

  for i in range(self.mini_batch_size):
    expected_reward.append(rewards[i] + self.future_reward_discount * np.max(current_rewards))

  loss = self.applied_action_model.train_on_batch([previous, actions], np.array(expected_reward))
```

Training our game player

There's not much to it really.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python q_cartpole_player.py
Using Theano backend.
Starting training.
Score is 173
⋮
```

The training will run until you close the game window or press Ctrl-C at the command line.

Notes about our player

Some notes about the player, and its training.

- Random moves are used to create the initial set of observations, before training of the NN begins.
- As training begins, the fraction of moves which are random, rather than from the NN, decreases slowly until only 5% of the moves are random.
- Random moves are still included in the training to help address the "explore-exploit" problem, forcing the network explore actions it wouldn't otherwise try.
- There is a maximum size to the collection of observations. Once it is filled, the older observations are discarded and replaced by the latest observations.

Read the code for all of the details of how the code works.

Notes about our player, continued

$$L = \frac{1}{2}\left[\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q\left(a_{t+1}, s_{t+1}\right)}_{\text{target}} - \underbrace{Q(a_t, s_t)}_{\text{prediction}}\right]^2$$

As you might expect, because Q in both terms in this equation, training may diverge.

- The NN has a habit of falling into a local minimum. This manifests itself as the cart sitting at one of the ends trying to push into it, not going anywhere.
- This is combated with a multi-pronged approach:
  - Collect a large number of observations before starting to train. The point of this is to make sure you have enough observations that actually earn a reward.
  - Use a large enough mini-batch size.
  - Use dropout.
  - Add several hidden layers.

Even with these improvements, then network will still get stuck about 1/3 of the time.

Final notes about Deep-Q RL

Since it's first introduction in 2014, there have been many suggested improvements to the basic Deep-Q RL algorithm. The latest can be found here:

`http://rll.berkeley.edu/deeprlworkshop`

An entire video-game "gym" has been created by OpenAI to foster RL development:

`https://gym.openai.com/envs`

Be aware that Google has patented Deep Q-learning.

Linky goodness

Reinforcement learning:

- https://arxiv.org/abs/1312.5602
- https://www.intelnervana.com/demystifying-deep-reinforcement-learning
- http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- http://edersantana.github.io/articles/keras_rl
- https://keon.io/deep-q-learning
- https://medium.com/@gtnjuvin/my-journey-into-deep-q-learning-with-keras-and-gym-3e779cc12762
- http://www.danielslater.net/2015/12/how-to-run-learning-agents-against.html