Neural network programming: reinforcement learning

Erik Spence

SciNet HPC Consortium

30 April 2026

SciNet

Today’s code and slides

You can get the slides and code for today’s class at the SciNet Education web page.

`https://scinet.courses/1400`

Click on the link for the class, and look under "Lectures" on the right side-bar, click on "Reinforcement learning".

The code for today’s class will need the "gymnasium" package. The command "pip install gymnasium" should work. You’ll also probably need 'pygame-ce' installed.

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 $\mathbf{x}$ are the input features and $\boldsymbol{y}$ is the 'label' or 'target'. The goal is develop a model, $\boldsymbol{f}$, such that $\boldsymbol{f}(\mathbf{x}) = \boldsymbol{y}$.
- Most of the neural networks we have trained so far have been supervised, with the exception of the generative networks: VAEs and Diffusion models.
- Unsupervised learning means that you only get (or use) $\mathbf{x}$ and the goal is to determine patterns in $\mathbf{x}$ which may be useful.

There are other categories, one of 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.

Reinforcement learning, applications

Who cares? Isn't this only used for video games?

No, reinforcement learning has been used in a number of important applications, including areas of science and engineering.

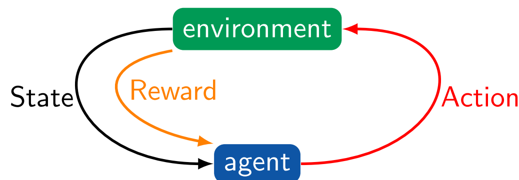
- Robotics: teaching robots to move around and interact with their environment.
- Business: inventory and delivery management.
- Finance: development of trading strategies.
- Science: protein folding, drug discovery, materials discovery.
- Games: AlphaGo, StarCraft II.

Any situation where you can set up a "agent" which interacts with an "environment" can use a reinforcement-learning approach.

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: a set of rules which dictate which actions to perform, given 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 (random) 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

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

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

SciNet

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 total discounted future reward when we perform action a on state s.

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

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:

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

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

SciNet

Q-learning loss function

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

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

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

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

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

What other problems does reinforcement learning have?

- 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.

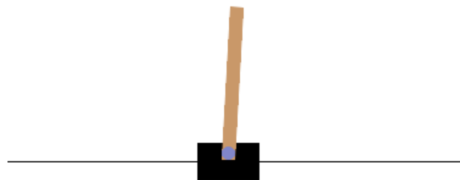
The collection of observations we train against starts out totally random; the randomness is reduced as the training of the network progresses.

RL example

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

- We'll use cartpole, which is available through the "gymnasium" package.
- The goal is to move the cart back and forth to keep the pendulum vertical.
- You get points for the amount of time the pendulum is near vertical.
- The game ends if
  - the pendulum exceeds 15 degrees from vertical, in either direction,
  - the cart moves off the screen.

The OpenAI Gym has been designed to be easy to use to develop reinforcement learning algorithms.



RL example, continued

What is the environment for this game?

- The state consists of 4 pieces of information: x, $\frac{dx}{dt}$, θ, $\frac{d\theta}{dt}$
- x is the position of the cart and θ is the angle of the pendulum.
- There are two possible actions: left and right. This refers to the motion of the cart.
- The cart and pendulum both have mass.
- Gravity points downward.

SciNet

Computer playing cartpole

How does the computer play the game?

- First the Cartpole environment is initialized.
- Then, while the game is not done:
  - The current state of the game is rendered to the screen.
  - The next action of the game is (somehow) decided.
  - The state of the game is stepped forward, based on the chosen action.
  - Repeat.

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

Computer learning cartpole

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

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 discounted future reward for each possible action.
- A collection of "observations" is made, and stored. Each observation consists of: (previous_state, action, reward, current_state, is_finished).
- To train the network:
  - Take a 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 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.

SciNet

Implementing the loss function

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

You may have noticed a peculiarity about the loss function.

- Given a state s_t we can calculate Q for every possible action, a_t.
- However, we only have the right side of the equation for one of the values of a_t, the action that we end up choosing, the action that leads to s_{t+1}, and results in r_{t+1}.
- If our neural network produces one value of Q for each possible action, how to do train the network for the actions which aren't chosen? There's no right side of the equation for those actions!
- It turns out that setting the right side to zero for those actions ruins the training.
- A better approach is to set the right side to $Q(a_t, s_t)$ for those a_t which are not selected.

SciNet

Computer learning cartpole, the algorithm

What's the actual algorithm?

- Setup the cartpole game.
- 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 current state.
  - Get the next best action, given the current state.
  - Step the game forward.
  - Get the reward for the chosen action.
  - Add the latest observation set to the collection of observations.
  - Train the neural network on a mini-batch of the observations.

SciNet

Deep Q-learning with cartpole

```python
# Q_Cartpole_Player.py
def build_model(self):

  input_state = kl.Input(
    shape = (self.state_dim,))

  x = kl.Dense(24,
    activation = "relu")(input_state)
  x = kl.Dense(24,
    activation = "relu")(x)
  q = kl.Dense(self.num_actions,
    activation = "linear")(x)

  self.q_model = km.Model(inputs =
    input_state, outputs = q)
  self.q_model.compile(loss = "mse",
    optimizer = "adam")
```

```python
# Train the NN.
def train_model(self):

  batch = random.sample(self.observations,
    self.batch_size)

  previous, actions, rewards, current, dones = \
    list(zip(*batch))

  q_update = rewards + (1 - np.array(dones)) *
    self.future_reward_discount *
    np.max(self.q_model.predict(np.array(current)))
  q_values = self.q_model.predict(np.array(previous))
  q_values[range(self.batch_size), actions] = q_update

  self.q_model.fit(np.array(previous), q_values,
    verbose = 0)
```

SciNet

Training our game player

There's not much to it really.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python
q_cartpole_player.py

Using Tensorflow backend.
Starting training.
Score is 173
⋮
```

The code will run 200 games, and train on them.

```python
# Train the player.
player = QCartPolePlayer()

for i in range(200):
  state = player.env.reset()[0];   total = 0;    done = False

  while (not done):
    player.env.render()
    prev_state = state
    action = player.get_next_action(prev_state)
    state, reward, done, _, info = player.env.step(action)

    reward = reward if not done else -reward
    total += reward
    player.remember([prev_state, action, reward, state, done])
    player.train_model()

  print(i, "reward is ", total)
```

SciNet

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.

Notes about our player, continued

Note that the algorithm which we've been examining, Deep Q-learning, is only appropriate, as presented, for discrete decision-making. Continuous decision-making, in which we output a value instead of a category, requires a different algorithm. There are many others out there:

- Deep Deterministic Policy Gradient (DDPG),
- Continuous DQN (CDQN or NAF)
- Cross-Entropy Method (CEM)
- Actor-critic methods (A2C and A3C)
- Proximal Policy Optimization

If you end up going down this road you will need to familiarize yourself with these other algorithms.

Linky goodness

Reinforcement learning:

- https://arxiv.org/abs/1312.5602
- https://www.davidsilver.uk/teaching
- http://edersantana.github.io/articles/keras_rl
- https://medium.com/@gtnjuvin/my-journey-into-deep-q-learning-with-keras-and-gym-3e779cc12762
- https://spinningup.openai.com/en/latest/algorithms/ddpg.html

SciNet