Learning To Play Surround

Eilif Solberg

September 30, 2018

1 Introduction

The game of Surround was designed for the Atari 2600 platform and was launched in 1977. We will use a modified version of the game, where the players take turns making their moves. You might think of it as a "board game version" of Surround. The game has two players, 'white' and 'black', where 'white' player goes first. You will randomly be assigned to play with either 'white' or 'black' each game, and you will also randomly start on the left or right side of the board. The players have four actions or moves: "left", "right", "up" and "down". Each time the player moves it leaves a "block" in its previous positions. The first player to crash into any block, either its own or the opponents, or crashes into the border of the board, looses. As the game is turn-based, there are no draws. A reward of 1 is given to the winner and -1 to the looser. We do not discount rewards.

The strategy we shall use for learning this game is called *self-play reinforcement learning*. In the normal reinforcement learning setting, we only try to optimize the expected (discounted) reward for an agent in an assumed fixed environment. In self-play reinforcement learning we create and update the environment as well. The environment consist of two parts. The simulator, which job is to apply the moves of the agents, update the board and return a view of the board and a reward for the agent. The simulator is fixed during the whole period, i.e. the rules of the game does not change. The second part of the environment is an *opponent*, in self-play reinforcement learning this is a clone of ourselves! To be precise our training shall proceed as follows. We initialize the agent, and then the environment with a clone of the agent as opponent. Then we train the agent in the environment, while keeping the opponent fixed, until our agent becomes clearly superior to the opponent. When the agent wins a certain percentage of the games (e.g. 60 percent), we update the opponent to a clone of the current agent. In that way we always play against an opponent that are about our level.

Note that although we said before that you will be randomly assigned to play either "black" or "white", in the observation (which may also be used as a state) returned form the environment, you will always look green with a blue *head* (the current position of the player) while your opponent will be red with a purple head. From the perspective of the opponent, he will also see himself as green and you as red. In this way can we use the same policy function to play both black and white.

2 Training

An environment for the agent is created with

```
environment = simulator.Environment(height, width, opponent, turn_based=True)
```

where *height* and *width* are the height and with of the board. After we have created an environment for the agent, we have transformed the problem into the standard reinforcement learning setting. Note that the opponent should be an object with a `next_action` method associated with it. The environment uses this to create the next action for the opponent, i.e.

```
action_opponent = opponent.next_action(state)
```

A new game and initial state is created for the agent with

```
player = random.choice(["white", "black"])
game = environment.new_game(player=player)
state = game.init_state()
```

To obtain the next state and reward from the the environment you should apply your action by

```
new_state, reward, END_OF_GAME = game.apply_action(action)
```

Here *new_state* is a numpy array of shape [height, width, 3] and reward is 0 is except if END_OF_GAME is true, otherwise it is -1 if we lost and 1 if we won. END_OF_GAME is a boolean which is true if the game is over.

We will in this problem extend the policy-gradient framework with a value function, we thus have an *actor-critic* setup. You should read up on this in the lecture notes on reinforcement learning. The network and training of the state-value function is already set up for you, though you are free to make changes to this.

Note that as checkpointing is implemented so you may stop and restart the training as you like.

3 Evaluation against baselines

We will measure progress in two ways. Firstly, we continuously monitor how well we are playing against our opponent. In addition to this, we have two baseline policies that we evaluate against. You will find both implemented in **baseline.py**. The first baseline opponent is a true random opponent, it picks 'left', 'right', 'up' and 'down' with equal probability. Note that this has at least a 1 in 4 chance of loosing every time it makes a move (except the first move), as moving in the opposite direction always leads to a crash. A slightly smarter baseline is the one that tries to avoid immediate loss. It first looks at all positions 'left', 'right', 'up' and 'down' and figures out which of them are safe to play. Then it chooses either of these with equal probability. If all actions will lead to a loss, it picks any action at random.

4 Your tasks

The only file you should need to edit is **actor_critic.py** You should create a policy-network

```
class PolicyNetwork:

  def __init__(self, height, width, sess):
    # you may want more arguments then the ones show above
    pass

  def next_action(self, state):
    pass

  # ... more methods ...
```

You will need to create the update operation for the policy network and run this where and when you find it appropriate. For this you will need to figure out how to use the value function to create a *critic*. You will need to update both the `train` and `initialize_networks` functions.

A few summaries are already implemented and logged to TensorBoard, you may e.g. see example games under the *IMAGES* tab. You may add more of these if you like. It is optional, but might be useful for debugging purposes. E.g. plotting the *entropy* of the outputs of the policy network could be useful. If the entropy for most states are zero, the policy is basically deterministic, while if the entropy is typically around $log(4)$, the network is sampling uniformly among the four actions.

Some hints that may help you with the solution.

- **Hint 1**: Note that the last form of the actor-critic from the notes

$$\nabla_\theta E_{\pi_\theta}(G_0) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\tau^{(i)}}\left(\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})((R_{t+1}^{(i)} + \gamma\hat{v}_\pi(S_{t+1}^{(i)})) - \hat{v}_\pi(s_t^{(i)}))\right)$$

  can be simplified into an online update at each time step, which you may find the easiest to implement.

- **Hint 2**: You may take inspiration from the `ValueNetwork` class and how updates are done there.

- **Hint 3**: This is related to the previous hint, but deserves some further explanation. The surround game has several symmetries that can be exploited. Making a "left" move from a certain position should be no worse or better than moving "right" from the board we get when we mirror the board around the vertical axis (i.e. left-right flip the image). Similarly moving "up" from a certain position, and moving "down" from the position in the board upside-down should be equivalent. See e.g the functions `mirror` and `augment_with_mirroring`.

- **Hint 4**: Focus on making sure you understand how things work, including your own implementation. Don't spend a lot of time randomly trying things to see if things improve/converge. Even a correct implementation may have no/slow convergence and your pass/fail will not depend on this. You will want to try different learning rates though. My experiments so far have not been very promising...

The file **actor_critic.py** is currently set up with *train dir* and *learning rate* as arguments, so from the command line it could be called by e.g.

```
TRAIN_DIR = "train_dir"
LEARNING_RATE=1e-5
python3 actor_critic.py $TRAIN_DIR" $LEARNING_RATE
```

You may of course change this as you like. Learning rate here refers to the learning rate (denoted by α in the notes) of the value network.

5 Hand in

We should hand in your assignment using Devilry. You upload should have three elements:

- Your code.
- A short report (need not be longer than a page) answering the questions:
  - how you implemented the agent update
  - if you have seen signs of improvement/convergence
  - if you tried to change something else, and what the result of it was
- Perhaps a zipped-version of a log-directory illustrating any of the points described in your report (there is an upload size limit on Devilry, perhaps 100MB).

You may optionally include some feedback on the assignment itself.