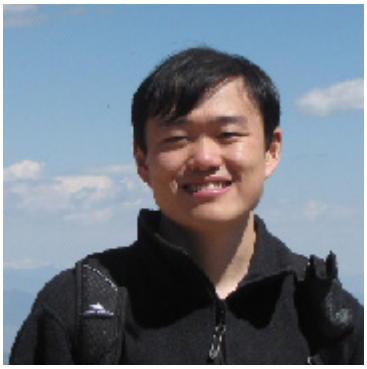
Tutorial on Deep Reinforcement Learning and Games

Presenter



Yuandong Tian

Research Scientist, Facebook AI Research

Email: yuandong [at] fb [dot] com

Prerequisites

Programming language:

Python, C/C++

Machine Learning:

Basic knowledge about deep learning and basic math skills. Note that math skills are probably more important.

Content

Introduction: AI and Games (15 min)

Basic Knowledge in Reinforcement Learning (1 hour)

- Q-learning
- Policy Gradient
- Actor-Critic Models

Advanced Topics (40 min)

- Soft-Q learning
- Model-based RL
- Hierarchical RL

Game Related Approaches (15 min)

- Alpha-beta pruning

- Monte-Carlo Tree Search (MCTS)

Case Study (40 min)

- AlphaGo Fan and AlphaGo Zero
- Our work
  - DarkForest
  - Doom AI bot
  - ELF platform
  - House3D

Interactive Tasks and Q&A (1 hour)

- Training MiniRTS model
- Evaluate and visualize MiniRTS model
- Add new features in the C side, recompile and read it from python

Slides

You can download the slides here.

A previous version can be found here

Installation

Linux

```
# Download miniconda and install.
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O $HOME/miniconda.sh
/bin/bash $HOME/miniconda.sh -b
$HOME/miniconda3/bin/conda update -y --all python=3

# Add the following to ~/.bash_profile (if you haven't already) and source it:
export PATH=$HOME/miniconda3/bin:$PATH

# Create a new conda environment and install the necessary packages:
conda create -n elf python=3
source activate elf
# If you use cuda 8.0, try conda install pytorch cuda80 -c soumith
conda install pytorch -c soumith

pip install --upgrade pip
pip install msgpack_numpy
conda install tqdm
conda install libgcc

# Install cmake >= 3.8 and gcc >= 4.9
# This is platform-dependent.
# If you use AWS, you can just skip this.
```

```
# Install tbb
# For ubuntu
sudo apt-get install libtbb-dev
# For Fedora / CentOS
sudo yum install libtbb-devel

# Clone and build the repository:
cd ~
git clone https://github.com/facebookresearch/ELF
cd ELF/rts/
mkdir build && cd build
cmake .. -DPYTHON_EXECUTABLE=$HOME/miniconda3/bin/python -DGAME_DIR=../game_MC
make

# Make sure that python3 is the alias of python
alias python3=$HOME/miniconda3/bin/python

# Train the model
sh ./train_minirts.sh --gpu 0
```

MacOS

```
# Download miniconda and install.
wget https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -O $HOME/miniconda.sh
/bin/bash $HOME/miniconda.sh -b
$HOME/miniconda3/bin/conda update -y --all python=3

# Add the following to ~/.bash_profile (if you haven't already) and source it:
export PATH=$HOME/miniconda3/bin:$PATH

# Create a new conda environment and install the necessary packages:
conda create -n elf python=3
source activate elf
# If you use cuda 8.0, try conda install pytorch cuda80 -c soumith
conda install pytorch -c soumith

pip install --upgrade pip
pip install msgpack_numpy
conda install tqdm
conda install libgcc

# Install cmake >= 3.8 and gcc >= 4.9
# This is platform-dependent.
brew install cmake
brew install tbb

# Clone and build the repository:
cd ~
git clone https://github.com/facebookresearch/ELF
cd ELF/rts/
mkdir build && cd build
```

```
cmake .. -DPYTHON_EXECUTABLE=$HOME/miniconda3/bin/python -DGAME_DIR=../game_MC
make

# Make sure that python3 is the alias of python
alias python3=$HOME/miniconda3/bin/python

# Train the model
sh ./train_minirts.sh --gpu 0
```

Windows

We don't have time to test it on windows. Please let me know if you manage to get it working.

Basic Tasks

Training MiniRTS model

Once you have downloaded `ELF`, compile it and run the following command to start training (make sure you are in the root directory of `ELF`):

```
game=./rts/game_MC/game model=actor_critic model_file=./rts/game_MC/model python3 train.py --batchsize 128 --freq_update 1 --players "type=AI_NN,fs=50,args=backup/AI_SIMPLE|start/500|decay/0.99;type=AI_SIMPLE,fs=20" --num_games 1024 --tqdm --T 20 --additional_labels id,last_terminal --trainer_stats winrate --keys_in_reply V
```

Without `--gpu [gpu_id]` it will run on CPU and is very slow, but is still running (and uses all your CPUs). Sample output looks like the following:

Evaluate MiniRTS model

Once the model has been trained, you can ask it to play against a rule-based AI. We can start with the pre-trained model (click into this page and click download) and run the following command to evaluate its performance (Here I suppose that you save the pre-trained model in `$HOME/model.bin`):

```
game=./rts/game_MC/game model_file=./rts/game_MC/model model=actor_critic python3 eval.py  --num_games 1024 --batchsize 128 --tqdm --load $HOME/model.bin --players "fs=50,type=AI_NN;fs=50,type=AI_SIMPLE" --eval_stats winrate --num_eval 1000 --additional_labels id,last_terminal,seq --shuffle_player --num_frames_in_state 1 --greedy --keys_in_reply V
```

With CPU-only machine, the evaluation is quite slow and might take 15min to evaluate 1000 games. With GPU, 10000 games will only take 2min. Sample output looks like the following (when rule-based AI has frame skip of 50):

```
Load from /private/home/yuandong/model.bin
Loaded = /private/home/yuandong/model.bin
100%|██████████| 10000/10000 [01:10<00:00, 141.37it/s]
new_record: True
count: 0
best_win_rate: 0.8358641358641276
str_win_rate: [0] Win rate: 0.836 [8367/1643/10010], Best win rate: 0.836 [0]
str_acc_win_rate: Accumulated win rate: 0.836 [8367/1643/10010]
Final statistics:
failed:
Result:0/0/4
  Win rate (as player 0): 0 0/4
  Win rate (as Player 1): 0 0/4

player 0:
Result:4054/4311/8365
  Win rate (as player 0): 0.484638 4054/8365
  Win rate (as Player 1): 0.515362 4311/8365

player 1:
Result:840/802/1642
  Win rate (as player 0): 0.511571 840/1642
  Win rate (as Player 1): 0.488429 802/1642

Base loc0 rate: 0.257217 2575/10011
Base loc1 rate: 0.256817 2571/10011
Base loc2 rate: 0.240236 2405/10011
Base loc3 rate: 0.24573 2460/10011
```

You might also try changing the frame skip of the opponent (e.g., from 50 to 20) and check the change of win rate. Frame skip controls how often the AI makes decisions and thus with a small frame skip, the AI is more fast-reacting and is stronger.

Visualize MiniRTS models

It seems that the win rate is quite high. Are you excited about how the trained AI win over the rule-based system? You can visualize replays using frontend tools provided in `ELF/rts/frontend`. First, save the replays from evaluation by adding a switch `--save_replay_prefix replay`, and rerun the evaluation code. Your scripts looks like the following:

```
game=./rts/game_MC/game model_file=./rts/game_MC/model model=actor_critic python3 eval.py  --num_games 1024 --batchsize 128 --tqdm --load $HOME/model.bin --players "fs=50,type=AI_NN;fs=50,type=AI_SIMPLE" --eval_stats winrate --num_eval 1000 --additional_labels id,last_terminal,seq --shuffle_player --num_frames_in_state 1 --greedy --keys_in_reply V --save_replay_prefix replay
```

The command will generate lots of replay files (`replay*.rep`) in the current folder (sorry for the flooding!). You can pick any replay file and load it in the webpage using the following command:

```
cd ~/ELF/rts/build
./minirts-backend replay --load_replay replay0-0.rep --vis_after 0
```

The command will wait for the websocket to open.

```
Players: dummy,dummy,spectator=0=/Users/yuandong/replays_decay_nofow.tar?new_replay427-4.rep
Dealing with player = dummy
Dealing with player = dummy
Dealing with player = spectator=0=/Users/yuandong/replays_decay_nofow.tar?new_replay427-4.rep
Loaded replay, size = 467
Waiting for websocket client ...
```

Then open the webpage at `~/ELF/rts/frontend/minirts.html` using your favorite browser, and you will see the replay. You can switch player aspect, pause or rollback to a previous time stamp by dragging the progress bar.

You can also tar all replays together into a `.tar` file, by using the following command:

```
tar cvf replays.tar *.rep
```

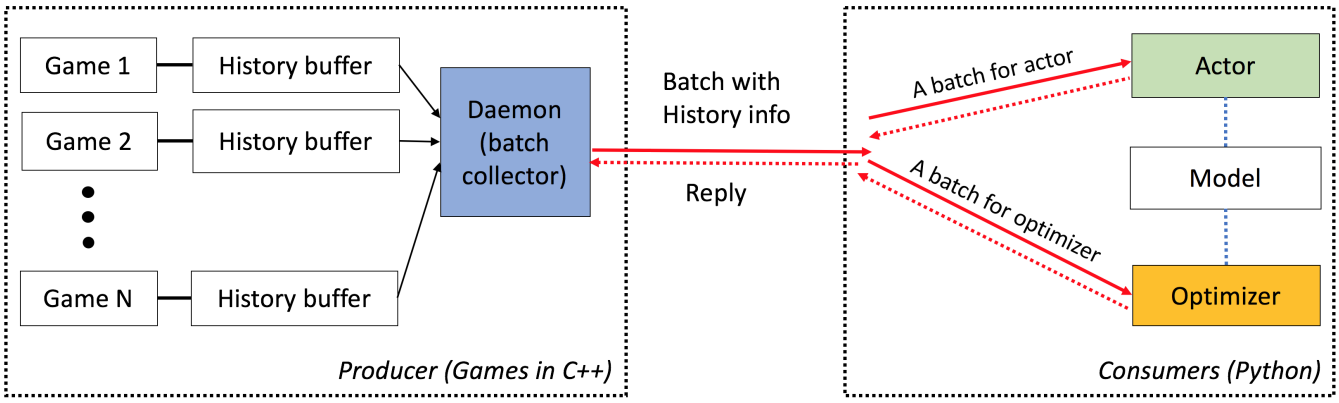
and then specify the replay using `--load_replay replays.tar?replay_name.rep`.

Here are a few video links: replay1 replay2 replay3 replay4 replay5

Advanced Tasks

Understand how ELF works

Running reinforcement learning algorithms efficiently requires a good collaboration of CPUs and GPUs. For GPU, it is more efficient to run a batch, and when GPU is doing its job, CPU can run the game simulator at the same time. Therefore, ELF uses the following architecture to simulate multiple games concurrently, and provide a batch of state to machine learning algorithm for training.



Any games with C++ interface could be incorporated into ELF. A game instance in a thread extracts features and sends it to ELF, waiting the response. ELF collects features from different game instance, wait until the batch size is reached and hands them over to machine learing side (Python). When machine learning side predicts the actions for the batch, they will be sent back to C++-side and resume corresponding game threads.

Make ELF customizable

You might already get tired of the simple MiniRTS game. How could we change the way features are extracted? Could we get a new variable from the C++ side (e.g., the x and y coordinates of our current base)? The answer is yes.

Description	Location
The features are defined in `GameState`	`rts/game_MC/python_options.h`
The features are extracted from `GameEnv` to `GameState`	`rts/game_MC/state_feature.cc`
Python gets the dimension of the features from `GameContext`	`rts/game_MC/python_wrapper.cc`
Python gets hyper-parameters of the environments from `GameContext`	`rts/game/MC/python_wrapper.cc`
Input and reply features defined in Python	`rts/game_MC/game.py`
Register callback function in `GCWrapper`. When a batch comes, these functions are notified	`train.py`

Could you modify the code in C++/Python so that Python knows where the base is?

Get the location of the base in `GameEnv`

`GameEnv` is a class that stores state information of MiniRTS game, e.g., the locations, Hit Points, moving speed of all units. There is a function in `GameEnv` that directly gives you the location of your base `FindClosestBase`. Use that function you can get a `UnitId`, then you can get the location of the unit using `GameEnv::GetUnit(unit_id)->GetPointF()`.

Save the location of the base in `GameState`

Once you get the location, in the feature extraction code, you can save them to `GameState`. Note that you will also need to add fields in GameState to hold them. For example:

```
struct GameState {
   ...
   float base_x, base_y;
   ...
   DECLARE_FIELD(GameState, [your current fields], base_x, base_y);
};
```

Do not forget to add the newly introduced field in the C macro `DECLARE_FIELD`, so that ELF's internal code knows how to deal with it.

Add the dimension information in C++/Python binding

Also you need to add the dimension information here. Juse add a new if-condition to return an `EntryInfo` that contains dimension information. All tensors returned by ELF are of size [#Hist, BatchSize, ...], where the dimenions in ... are specified by `EntryInfo`. For example, the state `s` contains all feature layers (20-by-20-by-22), so its `EntryInfo` is called by `key`, `type_name` and three numbers specifying the dimensions of the tensor:

```
if (key == "s")
    return EntryInfo(key, type_name,
        { (int)MCExtractor::Size() * _num_frames_in_state,  // Num of planes
          _context->options().map_size_y,                   // Size in Y
          _context->options().map_size_x                    // Size in X
        }
    );
```

On the other hand, the action `a` is a single discrete variable so its `EntryInfo` has only two parameters:

```
if (key == "a") return EntryInfo(key, type_name);
```

So similarly, you can also add a line:

```
if (key == "base_x" || key == "base_y") return EntryInfo(key, type_name);
```

Add the corresponding keys in Python side

Finally, add the keys in Python side here:

```
return dict(
        batchsize=self.args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "terminal", "base_x", "base_y"])),
        reply=dict(T=1, keys=set(reply_keys + self._unit_action_keys())),
    )
```

Test it

Finally we can test it. Recompile your code, add a breakpoint in the Python side. This can be done by changing

```
GC.reg_callback("train", trainer.train)
```

into

```
def train(batch):
    import pdb
    pdb.set_trace()
    return trainer.train(batch)

GC.reg_callback("train", train)
```

Then you can rerun training and see whether you got two additional fields `base_x` and `base_y`. This can be done by checking `batch.hist(0)["base_x"]` and `batch.hist(0)["base_y"]`, both should be a `FloatTensor`.

Enjoy!