Chess with Deep Reinforcement Learning: User Manual

Tuur Vanhoutte

Contents

1 Introduction

I made a chess engine that uses deep reinforcement learning to play moves. It is possible to host the AI model on a GPU-equipped server, or host the model locally.

This software has multiple features:

- Create a training set using a specific AI model, by playing chess games against itself.
- Deploy a whole cluster of servers and clients to create a training set in parallel.
- Train the network using a training set created by self-play.
- Evaluate two different AI models by playing a given number of games against each other.
- Play against an AI

2 Creating a training set

To create a training set, you first need to either create your own AI model, or use my pretrained model.

2.1 Create your own AI model

To create your own AI model, run the following command:

```
python rlmodelbuilder.py --model-folder <FOLDER> --model-name <NAME>

# For a detailed description of the parameters, run:
python rlmodelbuilder.py --help
```

If you want to change parameters like the number of hidden layers, the input and output shapes (if you want to use the AI for other games), or the amount of convolution filters, change values in the config.py file.

2.2 Use my pretrained model

You can find a link to my pretrained model in the README.md file in the repository. Copy the model.h5 file to the 'models/' folder.

2.3 Create a training set using the chosen model

There are two ways to create a training set:

- Play full chessgames with one model playing white, and a copy of the model playing black.
- Solve chess puzzles using the AI model.

The first method can easily be deployed using the docker-compose file:

```
# in repository's code/ folder:
docker-compose up --build
```

This will deploy one prediction server and 8 clients ⇒ 8 parallel games will play. The data for the training set will be stored in ./memory. The amount of clients can be changed in the docker-compose file. You can also use the two docker images in a cluster like Kubernetes, to reliably deploy many more servers and clients in parallel.

To manually run self-play or to solve puzzles, you can run the selfplay.py file with specific arguments. For a detailed description of the arguments, run:

```
python3 selfplay.py --help
```

2.3.1 Manual (non-dockerized) self-play with full games

```
# if you want to send predictions to a server, start the server first:
python3 server.py
```

```
# if you want to predict locally instead of on a server, add --local-predictions:
python3 selfplay.py --local-predictions
```

The games will be saved in the './memory' folder.

2.3.2 Solving puzzles (non-dockerized)

You can download the puzzles file here: `https://database.lichess.org/#puzzles`. This is a .csv file consisting of more than 2 million chess puzzles, with many different types ("mateIn1", "mateIn2", "endgame", "short", etc...).

Training with these puzzles is faster than training with full games, and the AI can more easily learn patterns like mating moves, or other patterns that are difficult to solve.

```
python3 selfplay.py --type puzzles \
    --puzzle-file <PATH-TO-CSV-FILE> \
    --puzle-type <TYPE>
```

Just like with self-play, you can add --local-predictions if you want to predict locally instead of on a server. If the puzzle ends in a checkmate within the move limit, the game will be saved to memory. That is why for now, only puzzles of type "mateInX" are supported. The puzzle move limit can be changed in the config.py file.

3 Training the AI using a created training set

```
python3 train.py \
    --model <PATH-TO-MODEL> \
    --data-folder <PATH-TO-TRAINING-SET-FOLDER>

# For a full description of the parameters, run:
python3 train.py --help
```

You can change parameters like batch size and learning rate in the config.py file.

4 Evaluating two models

```
python3 evaluate.py evaluate.py <model_1> <model_2> <nr_games>

# For example, to run 100 matches between two models, run:
python3 evaluate.py models/model_1.h5 models/model_2.h5 100

# For a full description of the parameters, run:
python3 evaluate.py --help
```

The white player in chess inherently has a slightly higher chance of winning, because they can play the first move. Therefore, to evaluate two models, each model has to play as white and black an equal amount of games.

Therefore, running n matches means the evaluation will consist of $2 \cdot n$ games, because each model will play both colors once per match.

The output of this command will be an overview of the results of the matches:

```
Evaluated these models for 100 matches:
    Model 1 = models/model_1.h5, Model 2 = models/model_2.h5
The results:
Model 1: X wins
Model 2: X wins
Draws: X
```

5 Playing against the AI

To play against the AI, you can run the following command:

```
# [brackets] indicate optional arguments
python3 main.py [--player <white|black>] \
    [--local-predictions] \
    [--model <PATH-TO-MODEL>]
```

- If you don't specify a player, a random side will be chosen for you.
- If you add --local-predictions, you also have to specify a model.

Figure 1: The chessboard GUI

- You can click on a piece to move it, then click a destination square
- When you click a piece, the square lights up to show it is selected

Figure 2: Promoting a pawn

5.1 Changing the AI difficulty

- You can change the difficulty of the AI by changing the amount of MCTS simulations per move in the config.py file.
- This will also change the amount of time it takes for the AI to make a move.