AlphaZero Documentations

AlphaZero

7x7 NoGo Example Workflow

for each iteration {

- 400 self-play games
  for each self-play game {
  - 75 MCTS simulations per move
  - first 15 moves use probabilistic policy from MCTS
  - 16th move and onwards play best move (argmax)

  }
- 20000 training examples will be sampled over all self-play games for this iteration
- training examples from the latest 10 iterations preserved
- 20000 * 10 examples feed into NN
  In NN:
  - 10 epochs
  - batch size = 64 randomly sampled
  - 20000 * 10 / 64 forward-and-backward propagation per epochs
  - lr = 0.001
- 160 arena games between old and new
  for each arena game {
  - 25 MCTS simulations per move
  - first 5 moves use probabilistic policy from MCTS
  - 6th move and onwards play best move (argmax)

  }
- 55% wins against old version, accepting new model

}

Core Functions

Coach.py

Handles the general workflow of AlphaZero.

Coach

class Coach(game, nnet, args)

Handles the general workflow of AlphaZero.

Parameters:

game : NogoGame
A NoGo game to provide game-specific APIs.

nnet : NNetWrapper
A neural networks wrapper to provide nn-specific APIs.

args : dotdict
A dictionary of all the parameters used during training. For example, number of training iterations, number of self-play games, CUDA support etc.

Returns:

None

- ### Coach.learn

  method Coach.learn()

  Performs some number of training iterations. Each iteration involves: self-play games -> retrain neural networks -> pit against the old version and decide whether to accept the new version.

  Parameters:

  None

  Returns:

  None

- ### Coach.getCheckpointFile

  method Coach. getCheckpointFile(iteration)

  Returns the checkpoint file name of iteration.

  Parameters:

  iteration : int
  The current index of iteration.

  Returns:

  None

- ### Coach.saveTrainExamples

  method Coach. saveTrainExamples(iteration)

  Saves the training examples of iteration to file.

  Parameters:

  iteration : int
  The current index of iteration.

  Returns:

  None

- ### `Coach.loadTrainExamples`

  `method` **`Coach.loadTrainExamples(iteration)`**

  Loads the training examples of iteration from file.

  Parameters:

  iteration : int

  The current index of iteration.

  Returns:

  None

MCTS.py

Executes simulations with Monte Carlo tree search.

MCTS

class MCTS(game, nnet, args)

A MCTS class that executes simulations and handles tree search.

Parameters:

game : NogoGame
A NoGo game to provide game-specific APIs.

nnet : NNetWrapper
A neural networks wrapper to provide nn-specific APIs.

args : dotdict
A dictionary of all the parameters used during training. For example, number of training iterations, number of self-play games, CUDA support etc.

Returns:

None

- ### MCTS.getActionProb

method MCTS.getActionProb(canonicalBoard, temp=1, verbose=False)

Executes number of simulations and formulate a probabilistic policy based on the visted counts. If the temperature is low (deep in the game), formulate the policy with one hot encoding corresponding to the most played move. For example, pi = [0,0,0,1,0,0,0,0,0] if A2 is the most played move.

Parameters:

canonicalBoard : numpy.ndarray
The input board.

temp : int
Temperature of the game. Temperature is high = 1if the game is shallow; temperature is low = 0 if the game is deep (some number of moves have been played).

verbose : bool
An option to print the policies given by neural networks and simulations to stderr. Default is False to not print.

Returns:

probs : list or None
The policy vector if there is at least one valid move to play. None if the game ended.

- ### `MCTS.search`
  `method` **`MCTS.search(canonicalBoard)`**
  Performs one simulation.
  Parameters:
  canonicalBoard : numpy.ndarray
  The input board.
  Returns:
  -v : int
  The negative value of the win/loss w.r.t. the current player.

- ### `MCTS.clear`
  `method` **`MCTS.clear()`**
  Clears the tree.
  Parameters:
  None
  Returns:
  None

SelfPlay.py

Executes self-play games and collect games records

init_processes

function init_processes(_counter)

The process initialization function for multiprocessing.

Parameters:

_counter : multiprocessing.sharedctypes.Synchronized

Returns:

None

SelfPlay

class SelfPlay(mcts, game, args)

Handles the executions of self-play games.

Parameters:

mcts : MCTS
game : NogoGame
args: dotdict

- ### SelfPlay.playGame

  method SelfPlay.playGame(total_beg)

  Executes one episode of self-play, starting with player 1. As the game is played, each turn is added as a training example to trainExamples. The game is played till the game ends. After the game ends, the outcome of the game is used to assign values to each example in trainExamples.

  Parameters:

  total_beg : float

  The total duration since the beginning of the whole self-play process.

  Returns;

  trainExamples: list

  A list of examples of the form (canonicalBoard, pi, v). pi is the MCTS informed policy vector, v is +1 if the player eventually won the game, else -1.

- ### SelfPlay.playGames

  method SelfPlay.playGames()

  Handles the whole self-play process with multiprocessing.

  Parameters:

  None

  Returns:

  iterationTrainExamples : list

  A list of examples from trainExamples of every self-play game.

Arena.py

Handles the arena games between the new version and the old version.

init_processes

function init_processes(_counter)

The process initialization function for multiprocessing.

Parameters:

_counter : multiprocessing.sharedctypes.Synchronized

Returns:

None

Arena

class Arena(player1, player2, game, numPs, display=None)

An Arena class where any 2 agents can be pit against each other.

Parameters:

player1 : MCTS

Player with the old neural networks.

player2: MCTS

Player with the new neural networks.

game : NogoGame

A NoGo game to provide game-specific APIs.

numPs : int

Number of processes for multiprocessing.

display : function

A function that takes board as input and prints it. (not used)

- ### Arena.playGame

  method Arena.playGame(eps, num, total_beg, verbose=False)

  Executes one episode of an arena game.

  Parameters:

  eps : int

  The index of this arena game.

  num : int

  The total number of arena games to play.

  total_beg : float

  The total duration since the beginning of the arena process.

  verbose : bool

  An option to print the game result. (not used)

  Returns:

  win : -1, 0, or +1

  +1 if player1 won; -1 if player2 won; 0 if draw.

- ## `Arena.playGames`

  `method` **`Arena.playGames(num)`**

  Plays num games in which player1 starts num/2 games and player2 starts num/2 games.

  Parameters:

  Num : int

  The number of arena games.

  Returns:

  oneWon : int

  The number of games that player1 won.

  twoWon : int

  The number of games that player2 won.

  draws : int

  The number of games that ended up in a draw.

utils.py

Stores some utility classes and functions.

dotdict

class dotdict(kwargs)**

A subclass of python <class 'dict'> in addition to support dot operator for accessing values of keys.

Parameters:

**kwargs : dict

In our usage, the input is a usual python dictionary object.

print_pi

function print_pi(pi, size, label)

Prints the policy pi to stderr.

Parameters:

pi : list

The policy pi.

size : int

Size of the board.

label : str

Label to be printed out.

Returns:

None

Game-specific Designs

Game.py

A template of game-specific APIs.

Game

class Game()

This class specifies the base Game class. To define your own game, subclass this class and implement the functions below. This works when the game is two-player, adversarial and turn-based. Use 1 for player1 and -1 for player2.

- ### Game.getInitBoard

  method Game.getInitBoard()

  Generates and returns the initial board.

  Returns:

  startBoard : numpy.ndarray
  Canonical board.

- ### Game.getBoardSize

  method Game.getBoardSize()

  Returns a tuple of board dimensions.

  Returns:

  (x, y) : tuple
  Board dimensions.

- ### Game.getActionSize

  method Game.getActionSize()

  Returns the number of all possible actions (including invalid actions).

  Returns:

  actionSize : int
  The number of actions.

- ### Game.getNextState

  method Game.getNextState(board, player, action)

  Takes the action on the board and returns the next state.\

  Returns:

  nextBoard : numpy.ndarray
  Canonical board after applying action.
  nextPlayer : int
  Player who plays in the next turn (should be -player).

- ### Game.getValidMoves

  method Game.getValidMoves(board, player)

  Returns a binary vector of length self.getActionSize(), 1 for moves that are valid from the current board and player, 0 for invalid moves.

  Returns:

  validMoves : numpy.ndarray
  A vector of valid moves.

- ### Game.getGameEnded

  method **Game.getGameEnded**

  Checks whether the game has ended and returns the winner: 0 if game has not ended. 1 if player won, -1 if player lost, small non-zero value for draw.

  Returns:

  r : int

  End game result.

- ### Game.getCanonicalBoard

  method **Game.getCanonicalBoard(board, player)**

  Returns the canonical form of the board. The canonical form should be independent of player. For e.g. in chess, the canonical form can be chosen to be from the pov of white. When the player is white, we can return board as is. When the player is black, we can invert the colors and return the board.

  Returns:

  canonicalBoard : numpy.ndarray

  Canonical form of the board.

- ### Game.getSymmetries

  method **Game.getSymmetries(board, pi)**

  Returns a list of [(board,pi)] where each tuple is a symmetrical form of the board and the corresponding pi vector. This is used when training the neural network from examples.

  Returns:

  symmForms : list

  List of (board, pi)

- ### Game.stringRepresentation

  method **Game.stringRepresentation(board)**

  Does a quick conversion of board to a string format, required by MCTS for hashing.

  Returns:

  boardString : str

  String format of the board.

- ### Game.beginSearch

- ### Game.inSearch

- ### Game.endSearch

NeuralNet.py

A template for nn-specific APIs.

NeuralNet

class NeuralNet()

This class specifies the base NeuralNet class. To define your own neural network, subclass this class and implement the functions below. The neural network does not consider the current player, and instead only deals with the canonical form of the board.

- ### NeuralNet.train

  method NeuralNet.train(examples)

  This function trains the neural network with examples obtained from self-play.

  Parameters:

  examples : list

  A list of training examples, where each example is of form (board, pi, v). pi is the MCTS informed policy vector for the given board, and v is its value. The examples has board in its canonical form.

- ### NeuralNet.predict

  method NeuralNet.predict(board)

  Execute a forward pass of nn and returns the results.

  Parameters:

  board : numpy.ndarray

  Current board in its canonical form.

  Returns:

  pi : numpy.ndarray

  A policy vector for the current board.

  v : float

  A float in [-1,1] that gives the value of the current board.

- ### NeuralNet.save_checkpoint

  method NeuralNet.save_checkpoint(folder, filename)

  Saves the current neural network (with its parameters) in folder/filename.

- ### NeuralNet.load_checkpoint

  method NeuralNet.load_checkpoint(folder, filename)

  Loads parameters of the neural network from folder/filename.