Note:

This is only a part of the code, showing fundamental aspects of the MCTS and neural network. Please note that other parts of the code, such as the game board, are not displayed here.

MCTS

In [ ]:
```
import numpy as np
import copy


def softmax(x):
    probs = np.exp(x - np.max(x))
    probs /= np.sum(probs)
    return probs


class TreeNode(object):
    """A node in the MCTS tree.

    Each node keeps track of its own value Q, prior probability P, and
    its visit-count-adjusted prior score u.
    """

    def __init__(self, parent, prior_p):
        self._parent = parent
        self._children = {}  # a map from action to TreeNode
        self._n_visits = 0
        self._Q = 0
        self._u = 0
        self._P = prior_p

    def expand(self, action_priors):

        for action, prob in action_priors:
            if action not in self._children:
                self._children[action] = TreeNode(self, prob)

    def select(self, c_puct):
        """Select action among children that gives maximum action value Q
        plus bonus u(P).
        Return: A tuple of (action, next_node)
        """
        return max(self._children.items(),
                   key=lambda act_node: act_node[1].get_value(c_puct))

    def update(self, leaf_value):
        """Update node values from leaf evaluation.
        leaf_value: the value of subtree evaluation from the current player's
            perspective.
        """
        # Count visit.
        self._n_visits += 1
```

```
        # Update Q, a running average of values for all visits.
        self._Q += 1.0*(leaf_value - self._Q) / self._n_visits

    def update_recursive(self, leaf_value):
        """Like a call to update(), but applied recursively for all ancestors.
        """
        # If it is not root, this node's parent should be updated first.
        if self._parent:
            self._parent.update_recursive(-leaf_value)
        self.update(leaf_value)

    def get_value(self, c_puct):

        self._u = (c_puct * self._P *
                   np.sqrt(self._parent._n_visits) / (1 + self._n_visits))
        return self._Q + self._u

    def is_leaf(self):
        """Check if leaf node (i.e. no nodes below this have been expanded)."""
        return self._children == {}

    def is_root(self):
        return self._parent is None


class MCTS(object):

    def __init__(self, policy_value_fn, c_puct=5, n_playout=10000):

        self._root = TreeNode(None, 1.0)
        self._policy = policy_value_fn
        self._c_puct = c_puct
        self._n_playout = n_playout

    def _playout(self, state):

        node = self._root
        while(1):
            if node.is_leaf():
                break
            # Greedily select next move.
            action, node = node.select(self._c_puct)
            state.do_move(action)

        action_probs, leaf_value = self._policy(state)
        # Check for end of game.
        end, winner = state.game_end()
        if not end:
            node.expand(action_probs)
        else:
            # for end state, return the "true" leaf_value
            if winner == -1:  # tie
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == state.get_current_player() else -1.0
                )

        # Update value and visit count of nodes in this traversal.
        node.update_recursive(-leaf_value)
```

```
    def get_move_probs(self, state, temp=1e-3):
        """Run all playouts sequentially and return the available actions and
        their corresponding probabilities.
        state: the current game state
        temp: temperature parameter in (0, 1] controls the level of exploration
        """
        for n in range(self._n_playout):
            state_copy = copy.deepcopy(state)
            self._playout(state_copy)

        # calc the move probabilities based on visit counts at the root node
        act_visits = [(act, node._n_visits)
                      for act, node in self._root._children.items()]
        acts, visits = zip(*act_visits)
        act_probs = softmax(1.0/temp * np.log(np.array(visits) + 1e-10))

        return acts, act_probs

    def update_with_move(self, last_move):
        """Step forward in the tree, keeping everything we already know
        about the subtree.
        """
        if last_move in self._root._children:
            self._root = self._root._children[last_move]
            self._root._parent = None
        else:
            self._root = TreeNode(None, 1.0)

    def __str__(self):
        return "MCTS"


class MCTSPlayer(object):
    """AI player based on MCTS"""

    def __init__(self, policy_value_function,
                 c_puct=5, n_playout=2000, is_selfplay=0):
        self.mcts = MCTS(policy_value_function, c_puct, n_playout)
        self._is_selfplay = is_selfplay

    def set_player_ind(self, p):
        self.player = p

    def reset_player(self):
        self.mcts.update_with_move(-1)

    def get_action(self, board, temp=1e-3, return_prob=0):
        sensible_moves = board.availables
        # the pi vector returned by MCTS as in the alphaGo Zero paper
        move_probs = np.zeros(board.width*board.height)
        if len(sensible_moves) > 0:
            acts, probs = self.mcts.get_move_probs(board, temp)
            move_probs[list(acts)] = probs
            if self._is_selfplay:
                # add Dirichlet Noise for exploration (needed for
                # self-play training)
                move = np.random.choice(
                    acts,
                    p=0.75*probs + 0.25*np.random.dirichlet(0.3*np.ones(len(prob
```

```
                )
                # update the root node and reuse the search tree
                self.mcts.update_with_move(move)
            else:
                # with the default temp=1e-3, it is almost equivalent
                # to choosing the move with the highest prob
                move = np.random.choice(acts, p=probs)
                # reset the root node
                self.mcts.update_with_move(-1)
#                location = board.move_to_location(move)
#                print("AI move: %d,%d\n" % (location[0], location[1]))

            if return_prob:
                return move, move_probs
            else:
                return move
        else:
            print("WARNING: the board is full")

    def __str__(self):
        return "MCTS {}".format(self.player)
```

Neural Network

In [ ]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np


def set_learning_rate(optimizer, lr):
    """Sets the learning rate to the given value"""
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr


class Net(nn.Module):
    """policy-value network module"""
    def __init__(self, board_width, board_height):
        super(Net, self).__init__()

        self.board_width = board_width
        self.board_height = board_height
        # common layers
        self.conv1 = nn.Conv2d(4, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        # action policy layers
        self.act_conv1 = nn.Conv2d(128, 4, kernel_size=1)
        self.act_fc1 = nn.Linear(4*board_width*board_height,
                                 board_width*board_height)
        # state value layers
        self.val_conv1 = nn.Conv2d(128, 2, kernel_size=1)
        self.val_fc1 = nn.Linear(2*board_width*board_height, 64)
        self.val_fc2 = nn.Linear(64, 1)
```

```
    def forward(self, state_input):
        # common layers
        x = F.relu(self.conv1(state_input))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        # action policy layers
        x_act = F.relu(self.act_conv1(x))
        x_act = x_act.view(-1, 4*self.board_width*self.board_height)
        x_act = F.log_softmax(self.act_fc1(x_act))
        # state value layers
        x_val = F.relu(self.val_conv1(x))
        x_val = x_val.view(-1, 2*self.board_width*self.board_height)
        x_val = F.relu(self.val_fc1(x_val))
        x_val = F.tanh(self.val_fc2(x_val))
        return x_act, x_val


class PolicyValueNet():
    """policy-value network """
    def __init__(self, board_width, board_height,
                 model_file=None, use_gpu=False):
        self.use_gpu = use_gpu
        self.board_width = board_width
        self.board_height = board_height
        self.l2_const = 1e-4  # coef of l2 penalty
        # the policy value net module
        if self.use_gpu:
            self.policy_value_net = Net(board_width, board_height).cuda()
        else:
            self.policy_value_net = Net(board_width, board_height)
        self.optimizer = optim.Adam(self.policy_value_net.parameters(),
                                    weight_decay=self.l2_const)

        if model_file:
            net_params = torch.load(model_file)
            self.policy_value_net.load_state_dict(net_params)

    def policy_value(self, state_batch):
        """
        input: a batch of states
        output: a batch of action probabilities and state values
        """
        if self.use_gpu:
            state_batch = Variable(torch.FloatTensor(state_batch).cuda())
            log_act_probs, value = self.policy_value_net(state_batch)
            act_probs = np.exp(log_act_probs.data.cpu().numpy())
            return act_probs, value.data.cpu().numpy()
        else:
            state_batch = Variable(torch.FloatTensor(state_batch))
            log_act_probs, value = self.policy_value_net(state_batch)
            act_probs = np.exp(log_act_probs.data.numpy())
            return act_probs, value.data.numpy()

    def policy_value_fn(self, board):
        """
        input: board
        output: a list of (action, probability) tuples for each available
        action and the score of the board state
        """
        legal_positions = board.availables
```

```python
        current_state = np.ascontiguousarray(board.current_state().reshape(
                -1, 4, self.board_width, self.board_height))
        if self.use_gpu:
            log_act_probs, value = self.policy_value_net(
                    Variable(torch.from_numpy(current_state)).cuda().float())
            act_probs = np.exp(log_act_probs.data.cpu().numpy().flatten())
        else:
            log_act_probs, value = self.policy_value_net(
                    Variable(torch.from_numpy(current_state)).float())
            act_probs = np.exp(log_act_probs.data.numpy().flatten())
        act_probs = zip(legal_positions, act_probs[legal_positions])
        value = value.data[0][0]
        return act_probs, value

    def train_step(self, state_batch, mcts_probs, winner_batch, lr):
        """perform a training step"""
        # wrap in Variable
        if self.use_gpu:
            state_batch = Variable(torch.FloatTensor(state_batch).cuda())
            mcts_probs = Variable(torch.FloatTensor(mcts_probs).cuda())
            winner_batch = Variable(torch.FloatTensor(winner_batch).cuda())
        else:
            state_batch = Variable(torch.FloatTensor(state_batch))
            mcts_probs = Variable(torch.FloatTensor(mcts_probs))
            winner_batch = Variable(torch.FloatTensor(winner_batch))

        # zero the parameter gradients
        self.optimizer.zero_grad()
        # set learning rate
        set_learning_rate(self.optimizer, lr)

        # forward
        log_act_probs, value = self.policy_value_net(state_batch)
        # define the loss = (z - v)^2 - pi^T * log(p) + c||theta||^2
        # Note: the L2 penalty is incorporated in optimizer
        value_loss = F.mse_loss(value.view(-1), winner_batch)
        policy_loss = -torch.mean(torch.sum(mcts_probs*log_act_probs, 1))
        loss = value_loss + policy_loss
        # backward and optimize
        loss.backward()
        self.optimizer.step()
        # calc policy entropy, for monitoring only
        entropy = -torch.mean(
                torch.sum(torch.exp(log_act_probs) * log_act_probs, 1)
                )
        return loss.item(), entropy.item()

        #for pytorch version >= 0.5 please use the following line instead.
        #return loss.item(), entropy.item()

    def get_policy_param(self):
        net_params = self.policy_value_net.state_dict()
        return net_params

    def save_model(self, model_file):
        """ save model params to file """
        net_params = self.get_policy_param()  # get model params
        torch.save(net_params, model_file)
```

Traning Process

In [ ]:

```
from __future__ import print_function
import random
import numpy as np
from collections import defaultdict, deque
from game import Board, Game
from mcts_pure import MCTSPlayer as MCTS_Pure
from mcts_alphaZero import MCTSPlayer
from policy_value_net_pytorch import PolicyValueNet  # Pytorch


class TrainPipeline():
    def __init__(self, init_model=None):
        # params of the board and the game
        self.board_width = 6
        self.board_height = 6
        self.n_in_row = 4
        self.board = Board(width=self.board_width,
                           height=self.board_height,
                           n_in_row=self.n_in_row)
        self.game = Game(self.board)
        # training params
        self.learn_rate = 2e-3
        self.lr_multiplier = 1.0  # adaptively adjust the learning rate based on
        self.temp = 1.0  # the temperature param
        self.n_playout = 400
        self.c_puct = 5
        self.buffer_size = 10000
        self.batch_size = 512  # mini-batch size for training
        self.data_buffer = deque(maxlen=self.buffer_size)
        self.play_batch_size = 1
        self.epochs = 5  # num of train_steps for each update
        self.kl_targ = 0.02
        self.check_freq = 50
        self.game_batch_num = 1500
        self.best_win_ratio = 0.0
        # num of simulations used for the pure mcts, which is used as
        # the opponent to evaluate the trained policy
        self.pure_mcts_playout_num = 1000
        if init_model:
            # start training from an initial policy-value net
            self.policy_value_net = PolicyValueNet(self.board_width,
                                                   self.board_height,
                                                   model_file=init_model)
        else:
            # start training from a new policy-value net
            self.policy_value_net = PolicyValueNet(self.board_width,
                                                   self.board_height)
        self.mcts_player = MCTSPlayer(self.policy_value_net.policy_value_fn,
                                      c_puct=self.c_puct,
                                      n_playout=self.n_playout,
                                      is_selfplay=1)

    def get_equi_data(self, play_data):
        """augment the data set by rotation and flipping
        play_data: [(state, mcts_prob, winner_z), ..., ...]
        """
```

```
        extend_data = []
        for state, mcts_porb, winner in play_data:
            for i in [1, 2, 3, 4]:
                # rotate counterclockwise
                equi_state = np.array([np.rot90(s, i) for s in state])
                equi_mcts_prob = np.rot90(np.flipud(
                    mcts_porb.reshape(self.board_height, self.board_width)), i)
                extend_data.append((equi_state,
                                    np.flipud(equi_mcts_prob).flatten(),
                                    winner))
                # flip horizontally
                equi_state = np.array([np.fliplr(s) for s in equi_state])
                equi_mcts_prob = np.fliplr(equi_mcts_prob)
                extend_data.append((equi_state,
                                    np.flipud(equi_mcts_prob).flatten(),
                                    winner))
        return extend_data

    def collect_selfplay_data(self, n_games=1):
        """collect self-play data for training"""
        for i in range(n_games):
            winner, play_data = self.game.start_self_play(self.mcts_player,
                                                          temp=self.temp)
            play_data = list(play_data)[:]
            self.episode_len = len(play_data)
            # augment the data
            play_data = self.get_equi_data(play_data)
            self.data_buffer.extend(play_data)

    def policy_update(self):
        """update the policy-value net"""
        mini_batch = random.sample(self.data_buffer, self.batch_size)
        state_batch = [data[0] for data in mini_batch]
        mcts_probs_batch = [data[1] for data in mini_batch]
        winner_batch = [data[2] for data in mini_batch]
        old_probs, old_v = self.policy_value_net.policy_value(state_batch)
        for i in range(self.epochs):
            loss, entropy = self.policy_value_net.train_step(
                    state_batch,
                    mcts_probs_batch,
                    winner_batch,
                    self.learn_rate*self.lr_multiplier)
            new_probs, new_v = self.policy_value_net.policy_value(state_batch)
            kl = np.mean(np.sum(old_probs * (
                    np.log(old_probs + 1e-10) - np.log(new_probs + 1e-10)),
                    axis=1)
            )
            if kl > self.kl_targ * 4:  # early stopping if D_KL diverges badly
                break
        # adaptively adjust the learning rate
        if kl > self.kl_targ * 2 and self.lr_multiplier > 0.1:
            self.lr_multiplier /= 1.5
        elif kl < self.kl_targ / 2 and self.lr_multiplier < 10:
            self.lr_multiplier *= 1.5

        explained_var_old = (1 -
                             np.var(np.array(winner_batch) - old_v.flatten()) /
                             np.var(np.array(winner_batch)))
        explained_var_new = (1 -
                             np.var(np.array(winner_batch) - new_v.flatten()) /
```

```
                              np.var(np.array(winner_batch)))
        print(("kl:{:.5f},"
               "lr_multiplier:{:.3f},"
               "loss:{},"
               "entropy:{},"
               "explained_var_old:{:.3f},"
               "explained_var_new:{:.3f}"
               ).format(kl,
                        self.lr_multiplier,
                        loss,
                        entropy,
                        explained_var_old,
                        explained_var_new))
        return loss, entropy

    def policy_evaluate(self, n_games=10):
        """
        Evaluate the trained policy by playing against the pure MCTS player
        Note: this is only for monitoring the progress of training
        """
        current_mcts_player = MCTSPlayer(self.policy_value_net.policy_value_fn,
                                         c_puct=self.c_puct,
                                         n_playout=self.n_playout)
        pure_mcts_player = MCTS_Pure(c_puct=5,
                                     n_playout=self.pure_mcts_playout_num)
        win_cnt = defaultdict(int)
        for i in range(n_games):
            winner = self.game.start_play(current_mcts_player,
                                          pure_mcts_player,
                                          start_player=i % 2,
                                          is_shown=0)
            win_cnt[winner] += 1
        win_ratio = 1.0*(win_cnt[1] + 0.5*win_cnt[-1]) / n_games
        print("num_playouts:{}, win: {}, lose: {}, tie:{}".format(
                self.pure_mcts_playout_num,
                win_cnt[1], win_cnt[2], win_cnt[-1]))
        return win_ratio

    def run(self):
        """run the training pipeline"""
        try:
            for i in range(self.game_batch_num):
                self.collect_selfplay_data(self.play_batch_size)
                print("batch i:{}, episode_len:{}".format(
                        i+1, self.episode_len))
                if len(self.data_buffer) > self.batch_size:
                    loss, entropy = self.policy_update()
                # check the performance of the current model,
                # and save the model params
                if (i+1) % self.check_freq == 0:
                    print("current self-play batch: {}".format(i+1))
                    win_ratio = self.policy_evaluate()
                    self.policy_value_net.save_model('./current_policy.model')
                    if win_ratio > self.best_win_ratio:
                        print("New best policy!!!!!!!!")
                        self.best_win_ratio = win_ratio
                        # update the best_policy
                        self.policy_value_net.save_model('./best_policy.model')
                        if (self.best_win_ratio == 1.0 and
                                self.pure_mcts_playout_num < 5000):
```

```
                        self.pure_mcts_playout_num += 1000
                        self.best_win_ratio = 0.0
        except KeyboardInterrupt:
            print('\n\rquit')


if __name__ == '__main__':
    training_pipeline = TrainPipeline()
    training_pipeline.run()
```

```
batch i:1, episode_len:11
batch i:2, episode_len:12
batch i:3, episode_len:12
batch i:4, episode_len:9
batch i:5, episode_len:12
batch i:6, episode_len:14
kl:0.00960,lr_multiplier:1.500,loss:4.526646614074707,entropy:3.5794434547424316,
explained_var_old:-0.000,explained_var_new:0.096
batch i:7, episode_len:12
kl:0.01102,lr_multiplier:1.500,loss:4.3851318359375,entropy:3.5677266120910645,ex
plained_var_old:0.126,explained_var_new:0.281
batch i:8, episode_len:9
kl:0.00336,lr_multiplier:2.250,loss:4.448668003082275,entropy:3.5769829750061035,
explained_var_old:0.073,explained_var_new:0.114
batch i:9, episode_len:24
kl:0.01561,lr_multiplier:2.250,loss:4.28877067565918,entropy:3.5379462242126465,e
xplained_var_old:0.180,explained_var_new:0.258
batch i:10, episode_len:9
kl:0.01832,lr_multiplier:2.250,loss:4.388433933258057,entropy:3.578704357147217,e
xplained_var_old:0.103,explained_var_new:0.167
batch i:11, episode_len:13
kl:0.03075,lr_multiplier:2.250,loss:4.429268836975098,entropy:3.5323004722595215,
explained_var_old:0.109,explained_var_new:0.136
batch i:12, episode_len:18
kl:0.03027,lr_multiplier:2.250,loss:4.295185565948486,entropy:3.5308995246887207,
explained_var_old:0.180,explained_var_new:0.247

quit
```

In [ ]: