Sequence

In [4]:

```
import random


def MoveGenerator(self, mode, stage):
    dict = {'r': 0, 'p': 1, 's': 2}
    seqlist = 'rpsrpsrpsrpsrpsrpsrpsrpsrpsrsprps'           # the pattern sequence here
    self.seqIndex = 0 if self.seqIndex == len(seqlist)-1 else self.seqIndex + 1
    return dict[seqlist[self.seqIndex]]
```

General DRL (With 300 Episodes)

In [9]:

```
import numpy as np
from matplotlib import style
import matplotlib.pyplot as plt
from collections import deque
from tensorflow.keras.optimizers import Adam
from keras.layers import Dense, Dropout
from keras.models import Sequential
style.use('ggplot')


class RPSEnvironment():
    def __init__(self):
        self.action_space = [0, 1, 2]
        self.seqIndex = 0
        self.Opponent_PlayerMode = 'SEQ'
        self.Opponent_PlayerCount = [0, 0, 0]
        self.AgentCount = [0, 0, 0]
        self.window = 10
        self.cumWinRate, self.cumTieRate, self.cumLostRate = None, None, None
        self.cumWinCount, self.cumTieCount, self.cumLostCount = None, None, None
        self.winRateTrend, self.tieRateTrend, self.lostRateTrend = 0, 0, 0
        self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg = 0, 0, 0
        # put all the observation state in here; shape in Keras input format
        self.state = np.array([[
            None, None, None,
            self.winRateTrend, self.tieRateTrend, self.lostRateTrend,
            self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg
        ]])

    def reset(self):
        # reset all the state
        self.cumWinRate, self.cumTieRate, self.cumLostRate = 0, 0, 0
        self.cumWinCount, self.cumTieCount, self.cumLostCount = 0, 0, 0
        self.winRateTrend, self.tieRateTrend, self.lostRateTrend = 0, 0, 0
        self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg = 0, 0, 0
        return np.array([0, 0, 0, 0, 0, 0, 0, 0, 0])

    def step(self, action, moveCount, stage):
        # value mode is PRNG or SEQ
        # play one move from player2
        Opponent_PlayerMove = MoveGenrator(
            self, self.Opponent_PlayerMode, stage)
        self.Opponent_PlayerCount[Opponent_PlayerMove] += 1
        AgentMove = action
        self.AgentCount[AgentMove] += 1

        # check who won, set flag and assign reward
        win, tie, lost = 0, 0, 0
        if AgentMove == Opponent_PlayerMove:
            self.cumTieCount, tie = self.cumTieCount + 1, 1
        elif (AgentMove - Opponent_PlayerMove == 1) or (AgentMove - Opponent_PlayerMove == -2):
            self.cumWinCount, win = self.cumWinCount + 1, 1
        else:
            self.cumLostCount, lost = self.cumLostCount + 1, 1

        # update the running rates
        self.cumWinRate = self.cumWinCount / moveCount
        self.cumTieRate = self.cumTieCount / moveCount
        self.cumLostRate = self.cumLostCount / moveCount
        # calculate trend
        tmp = [0, 0, 0]
        self.winRateTrend, self.tieRateTrend, self.lostRateTrend = 0, 0, 0
        if moveCount >= self.window:
            if self.winRateMovingAvg < tmp[0]:
                self.winRateTrend = 1
            else:
                self.winRateTrend = 0
            # tie rate trend analysis
            if self.tieRateMovingAvg < tmp[1]:
                self.tieRateTrend = 1
            else:
                self.tieRateTrend = 0
            # lost rate trend analysis
            if self.lostRateMovingAvg < tmp[2]:
                self.lostRateTrend = 1
            else:
                self.lostRateTrend = 0
            self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg = tmp[
                0], tmp[1], tmp[2]
        # net reward in this round
        if win == 1:
            reward = 1
        elif tie == 1:
```

```
            reward = 0
        elif lost == 1:
            reward = -1
        # record the state and reshape it for Keras input format
        dim = self.state.shape[1]
        self.state = np.array([
            win, tie, lost,
            self.winRateTrend, self.tieRateTrend, self.lostRateTrend,
            self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg
        ]).reshape(1, dim)
        # this game is done when it hits this goal
        if self.seqIndex >= 31:
            done = True
        else:
            done = False
        return self.state, reward, done, dim


class DoubleDQN:
    def __init__(self, env):
        self.env = env
        # initialize the memory and auto drop when memory exceeds maxlen
        # this controls how far out in history the "expeience replay" can select from
        self.memory = deque(maxlen=2000)
        # future reward discount rate of the max Q of next state
        self.gamma = 0.7
        # epsilon denotes the fraction of time dedicated to exploration (as oppse to exploitation)
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.9910
        # model learning rate (use in backprop SGD process)
        self.learning_rate = 0.005
        # transfer learning proportion contrl between the target and action/behavioral NN
        self.tau = .125
        # create two models for double-DQN implementation
        self.model = self.DeepLearningModel()
        self.target_model = self.DeepLearningModel()
        # some space to collect TD target for instrumentaion
        self.TDtargetdelta, self.TDtarget = [], []
        self.Qmax = []

    def DeepLearningModel(self):
        model = Sequential()
        state_shape = self.env.state.shape[1]
        model.add(Dense(24, input_dim=state_shape, activation="relu"))
        model.add(Dense(24, activation="relu"))
        model.add(Dense(24, activation="relu"))
        # let the output be the predicted target value.  NOTE: do not use activation to squash it!
        model.add(Dense(len(self.env.action_space)))
        model.compile(loss="mean_squared_error",
                      optimizer=Adam(lr=self.learning_rate))
        print(model.summary())

        return model

    def action(self, state):
        # this is to take one action
        self.epsilon *= self.epsilon_decay
        self.epsilon = max(self.epsilon_min, self.epsilon)
        # decide to take a random exploration or make a policy-based action (thru NN prediction)
        if np.random.random() < self.epsilon:
            # return a random move from action space
            return random.choice(self.env.action_space)
        else:
            # return a policy move
            self.Qmax.append(max(self.model.predict(state)[0]))
            return np.argmax(self.model.predict(state)[0])

    def remember(self, state, action, reward, new_state, done):
        # store up a big pool of memory
        self.memory.append([state, action, reward, new_state, done])

    def Experience_replay(self):        # DeepMind "experience replay" method
        # the sample size from memory to learn from
        batch_size = 32
        # do nothing untl the memory is large enough
        if len(self.memory) < batch_size:
            return
        # get the samples
        samples = random.sample(self.memory, batch_size)
        # do the training (learning); this is DeepMind tricks of using "Double" model (Mnih 2015)
        for sample in samples:
            state, action, reward, new_state, done = sample
            target = self.target_model.predict(state)
            #print('target at state is ', target)
            if done:
```

```
                target[0][action] = reward
            else:
                Q_future = max(self.target_model.predict(new_state)[0])
                TDtarget = reward + Q_future * self.gamma
                self.TDtarget.append(TDtarget)
                self.TDtargetdelta.append(TDtarget - target[0][action])
                target[0][action] = TDtarget
            # do one pass gradient descend using target as 'label' to train the action model
            self.model.fit(state, target, epochs=1, verbose=0)

    def target_train(self):
        # transfer weights  proportionally from the action/behave model to the target model
        weights = self.model.get_weights()
        target_weights = self.target_model.get_weights()
        for i in range(len(target_weights)):
            target_weights[i] = weights[i] * self.tau + \
                target_weights[i] * (1 - self.tau)
        self.target_model.set_weights(target_weights)

    def save_model(self, fn):
        self.model.save(fn)

# ------------------------- MAIN BODY ----------------------------------------


def main():
    episodes, trial_len = 300, 300                  # lenght of game play 150,300
    cumReward, argmax = 0, 0                        # init for intrumentation
    steps, rateTrack = [], []
    avgQmaxList, avgQ_futureList, avgQ_targetmaxList, avgTDtargetList = [], [], [], []
    avgCumRewardList = []
    AgentRate, Opponent_PlayerRate = [], []
    # declare the game play environment and AI agent
    env = RPSEnvironment()
    dqn_agent = DoubleDQN(env = env)
    # ------------------------------------------- start the game ----------------------------------------
    print('STARTING THE GAME with %s episodes each with %s moves' %
          (episodes, trial_len), '\n')
    for episode in range(episodes):
        # reset and get initial state in Keras shape
        cur_state = env.reset().reshape(1, env.state.shape[1])
        cumReward = 0
        if (episode+1) % (episodes // totalStages) == 0:
        stage = episode // (episodes // totalStages)

        for step in range(trial_len):
            # AI agent take one action
            action = dqn_agent.action(cur_state)
            # play the one move and see how the environment reacts to it
            new_state, reward, done, info = env.step(action, step + 1, stage)
            cumReward += reward
            # record the play into memory pool
            dqn_agent.remember(cur_state, action, reward, new_state, done)
            # perform Q-learning from using |"experience replay": learn from random samples in memory
            dqn_agent.Experience_replay()
            # apply tranfer learning from actions model to the target model.
            dqn_agent.target_train()
            # update the current state with environment new state
            cur_state = new_state
            if done:
                break

        rateTrack.append([episode+1, env.cumWinRate,
                          env.cumTieRate, env.cumLostRate])
        if True:        # print ongoing performance
            print('EPISODE ', episode + 1),
            if env.Opponent_PlayerMode == 'SEQ':
                print('stage:', stage, ' sigma:', env.norm_sigma)
            print(' WIN RATE %.2f ' % env.cumWinRate,
                  '    tie rate %.2f' % env.cumTieRate,
                  'lose rate %.2f' % env.cumLostRate)

        # print move distribution between the players
        if True:
            AgentRate.append([env.AgentCount[0] / trial_len,
                              env.AgentCount[1] / trial_len, env.AgentCount[2] / trial_len])
            Opponent_PlayerRate.append([env.Opponent_PlayerCount[0] / trial_len,
                                        env.Opponent_PlayerCount[1] / trial_len, env.Opponent_PlayerCount[2] / trial_len])
            print(' Agent rock rate: %.2f paper rate: %.2f scissors rate: %.2f' %
                  (AgentRate[-1][0], AgentRate[-1][1], AgentRate[-1][2]))
            print(' Opponent_Player rock rate: %.2f paper rate: %.2f scissors rate: %.2f' %
                  (Opponent_PlayerRate[-1][0], Opponent_PlayerRate[-1][1], Opponent_PlayerRate[-1][2]))
            env.AgentCount, env.Opponent_PlayerCount = [0, 0, 0], [0, 0, 0]

        # summarize Qmax from action model and reward
        avgQmax = sum(dqn_agent.Qmax) / trial_len   # from action model
```

```
        avgQmaxList.append(avgQmax)

        avgCumReward = cumReward / trial_len
        avgCumRewardList.append(avgCumReward)
        if True:
            print(' Avg reward: %.2f Avg Qmax: %.2f' % (avgCumReward, avgQmax))
        dqn_agent.Qmax = []         # reset for next episode

    # ---------------- plot the main plot when all the episodes are done ---------------------------
    #
    if True:

        fig = plt.figure(figsize=(12, 5))
        plt.subplots_adjust(wspace=0.2, hspace=0.2)

        # plot the average Qmax
        rpsplot = fig.add_subplot(321)
        plt.title('Average Qmax from action model', loc='Left', weight='bold', color='Black',
                  fontdict={'fontsize': 10})
        rpsplot.plot(avgQmaxList, color='blue')

        # plot the TDtarget
        rpsplot = fig.add_subplot(323)
        plt.title('TD target minus Q target from experience replay', loc='Left', weight='bold',
                  color='Black', fontdict={'fontsize': 10})
        rpsplot.plot(dqn_agent.TDtarget, color='blue')

        # plot the TDtarget
        rpsplot = fig.add_subplot(325)
        plt.title('TD target from experience replay', loc='Left', weight='bold', color='Black',
                  fontdict={'fontsize': 10})
        rpsplot.plot(dqn_agent.TDtargetdelta, color='blue')

        # plot thte win rate
        rpsplot = fig.add_subplot(322)
        plt.title('Win-Tie-Lost Rate', loc='Left', weight='bold', color='Black',
                  fontdict={'fontsize': 10})
        rpsplot.plot([i[1] for i in rateTrack], color='green')
        rpsplot.plot([i[2] for i in rateTrack], color='blue')
        rpsplot.plot([i[3] for i in rateTrack], color='red')

        # plot thte win rate
        rpsplot = fig.add_subplot(324)
        plt.title('Player 2 move percentage', loc='Left', weight='bold', color='Black',
                  fontdict={'fontsize': 10})
        rpsplot.plot([i[0] for i in Opponent_PlayerRate], color='orange')
        rpsplot.plot([i[1] for i in Opponent_PlayerRate], color='red')
        rpsplot.plot([i[2] for i in Opponent_PlayerRate], color='green')

        # plot the reward
        rpsplot = fig.add_subplot(326)
        plt.title('Average Reward per Episode', loc='Left', weight='bold', color='Black',
                  fontdict={'fontsize': 10})
        rpsplot.plot(avgCumRewardList, color='green')
        plt.show(block=False)

if __name__ == "__main__":
    main()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 24)                240

 dense_9 (Dense)             (None, 24)                600

 dense_10 (Dense)            (None, 24)                600

 dense_11 (Dense)            (None, 3)                 75

=================================================================
Total params: 1,515
Trainable params: 1,515
Non-trainable params: 0
_________________________________________________________________
None
Model: "sequential_3"
```

In [ ]: