DQN Sort

Name : 藤原 大悟

Student Number : 6930-31-6255

Formulation

数字列の昇順ソートタスク

状態 s_t ： N桁の0~9の数字($x_1, x_2, \dots, x_N$)の列(ただし0が先頭でも良いこととした)

行動 a_t ： 数字の隙間(i=1~N-1)に対し，その両隣の数字を交換する操作.　確率1で交換後の一つの数字列を観測する.

報酬 r_t : 数列の昇順ソートに成功すると+1, それ以外の場合は一回の操作ごとに$-\frac{2}{N(N-1)}$ (罰則)

ニューロン数がNである隠れ層を二つ有する4層のニューラルネットワークでDQN強化学習を行う. 入力を状態s_t，出力を各行動a_t の有する価値としてQ関数のニューラルネットワークによる近似を行う.

誤差はTD誤差

$$TDError = y_t - Q_\theta(s_t, a_t)$$

$$y_t = \begin{cases} r_t & s_{t+1}\text{が終端状態} \\ r_t + \max_{a'} \hat{Q}_{\bar{\theta}}(s_{t+1}, a') & \text{otherwise} \end{cases}$$

を用い，これに対しHuberLossを考えた.

ターゲットネットワークの更新頻度Bを128とした.

ミニバッチサイズを16，キューのサイズを100としてExperience Replayを適用した.

最適化手法はAdamを採用した.

ϵ-greedy法のパラメータϵは初期値を1とし，以降指数的に各エポックごとに減衰し，0に向かうようにした.

学習の開始は更新回数がA=1000を超えてからとした.

N=4とした

Implementation

Tensorflowを用いて実装を行なった

In [143]:

```
import matplotlib.pyplot as plt
from collections import deque
import tensorflow as tf
import numpy as np
import math
from scipy.stats import multivariate_normal as multi_gauss
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

In [147]:

```
class DQN_SORT:
    def __init__(self,inputnum):
        self.N=inputnum
        #隠れ層1のニューロン数
        self.M=100
        #隠れ層2のニューロン数
        self.L=100
        #行動数
        self.Nact=self.N-1
        #ミニバッチ数
        self.minibatch_size = 16
        #キューのサイズ
        self.replay_memory_size = 1000
        self.epsilon=0.5
        # replay memory
        self.P = deque(maxlen=self.replay_memory_size)

        # model
        self.init_model()

        # variables
        self.current_loss = 0.0


    def init_model(self):
        # (0) 入力文字列
        self.x = tf.placeholder(tf.float32, [None, self.N],name='x')

        # (1) サイズ変更
        x1 = tf.reshape(self.x, [-1,self.N])

        # (2) 隠れ層1
        # 重みとバイアス
        self.w1 = tf.Variable(tf.truncated_normal([self.N,self.M],stddev=0.01))
        self.b1 = tf.Variable(tf.zeros([self.M]))
        x2 = tf.nn.relu( tf.matmul(x1, self.w1) + self.b1)

        # (3) 隠れ層2
        # 重みとバイアス
        self.w2 = tf.Variable(tf.truncated_normal([self.M,self.L],stddev=0.01))
        self.b2 = tf.Variable(tf.zeros([self.L]))
        x3 = tf.nn.relu( tf.matmul(x2,self.w2) + self.b2)

        # (3) 出力層
        # 重みとバイアス
        self.w3 = tf.Variable(tf.truncated_normal([self.L,self.Nact],stddev=0.01))
        self.b3 = tf.Variable(tf.zeros([self.Nact]))
        self.Qs = tf.matmul(x3, self.w3) + self.b3


        ##target network
        # (0) 入力文字列
        self.xtar = tf.placeholder(tf.float32, [None, self.N],name='x')

        # (1) サイズ変更
        x1tar = tf.reshape(self.xtar, [-1,self.N])

       # (2) 隠れ層1
        # 重みとバイアス
```

```
        self.w1tar = tf.Variable(self.w1)
        self.b1tar = tf.Variable(self.b1)
        x2tar = tf.nn.relu( tf.matmul(x1tar, self.w1tar) + self.b1tar)

        # (3) 隠れ層2
        # 重みとバイアス
        self.w2tar = tf.Variable(self.w2)
        self.b2tar = tf.Variable(self.b2)
        x3tar = tf.nn.relu( tf.matmul(x2tar, self.w2tar) + self.b2tar)

        # (3) 出力層
        # 重みとバイアス
        self.w3tar = tf.Variable(self.w3)
        self.b3tar = tf.Variable(self.b3)
        self.Qstar = tf.matmul(x3tar, self.w3tar) + self.b3tar


        # 損失関数
        self.y = tf.placeholder(tf.float32, [None])#バッチサイズ分の長さをもつ
        self.act_indices=tf.placeholder(tf.int32, [None,2])#選択アクションを含むindex
        self.Q= tf.gather_nd(self.Qs,self.act_indices)#バッチサイズ分の長さをもつQ値

        self.loss = tf.reduce_mean(tf.losses.huber_loss(self.y,self.Q))

        # train operation
        optimizer = tf.train.AdamOptimizer()
        self.training = optimizer.minimize(self.loss)

        # session
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())

    def Qs_func(self, state):
        res=self.sess.run(self.Qs, feed_dict={self.x: [state]})
        return res[0]

    def Qstar_func(self,state):
        res=self.sess.run(self.Qstar, feed_dict={self.xtar: [state]})
        return res[0]

    def target_update(self):
        tf.assign(self.w1tar,self. w1)
        tf.assign(self.b1tar, self.b1)
        tf.assign(self.w2tar, self.w2)
        tf.assign(self.b2tar, self.b2)
        tf.assign(self.w3tar, self.w3)
        tf.assign(self.b3tar, self.b3)

    def select_action(self, state):
        if np.random.rand() <= self.epsilon:
            # random
            return np.random.randint(self.Nact)
        else:
            # max_action Q(state, action)
            return np.argmax(self.Qs_func(state))

    def store_experience(self, state, action, reward, state_1, terminal):
        self.P.append((state, action, reward, state_1, terminal))

    def experience_replay(self):
        state_minibatch = []
```

```
        y_minibatch = []
        act_indices=[]
        index=0

        # sample random minibatch
        # キューに十分にたまるまでは全部使う
        minibatch_size = min(len(self.P), self.minibatch_size)
        minibatch_indexes = np.random.choice(np.arange(len(self.P)), minibatch_size,replace=False)

        for j in minibatch_indexes:
            state_j, action_j, reward_j, state_j_1, terminal = self.P[j]

            y_j= 0

            if terminal:
                y_j= reward_j
            else:
                y_j= reward_j + np.max(self.Qstar_func(state_j_1))

            state_minibatch.append(state_j)
            y_minibatch.append(y_j)
            act_indices.append([index,action_j])
            index+=1

        # training
        self.sess.run(self.training, feed_dict={self.x: state_minibatch, self.y: y_minibatch, self.act_in
        # for log
        self.current_loss = self.sess.run(self.loss, feed_dict={self.x: state_minibatch, self.y: y_miniba
```

In [148]:

```
class StateArray:
    def __init__(self,N):
        self.N=N
        self.arr=np.random.randint(0,9,N)
        self.frame=0

    def excute_action(self,action):
        temp=self.arr[action]
        self.arr[action]=self.arr[action+1]
        self.arr[action+1]=temp

    def is_sorted(self):
        res=True
        for i in range(self.N-1):
            res=(self.arr[i]<=self.arr[i+1]) and res
        return res

    def observe(self):
        self.reward=-1*2/(self.N*(self.N-1))
        self.terminal=False
        if self.is_sorted():
            self.reward=1
            self.terminal=True
        return self.arr,self.reward,self.terminal
```

In [149]:

```
# parameters
n_epochs = 100

decrese=0.001**(1/n_epochs)

N=4

dqn = DQN_SORT(N)

A=1024
B=1024

count=0

learning_curve=[]
average=0.0
buff=3

for e in range(n_epochs):
    # reset
    frame=0
    cum_reward=0.0
    sa=StateArray(N)
    state_t_1,reward_t,terminal=sa.observe()

    while not terminal:
        state_t = state_t_1

        # execute action in environment
        action_t = dqn.select_action(state_t)
        sa.excute_action(action_t)

        # observe environment
        state_t_1, reward_t, terminal = sa.observe()

        # store experience
        dqn.store_experience(state_t, action_t, reward_t, state_t_1, terminal)

        # experience replay
        if count>A:
            dqn.experience_replay()

        if count%B==0:
            dqn.target_update()


        cum_reward+=reward_t
        average+=cum_reward

        if frame%buff==buff-1:
            average/=buff
            learning_curve.append([count,average])

        count+=1
        frame+=1

    dqn.epsilon*=decrese
    if e%10==0 and count>A:
        print("epoch=",e,"loss=",dqn.current_loss)
```

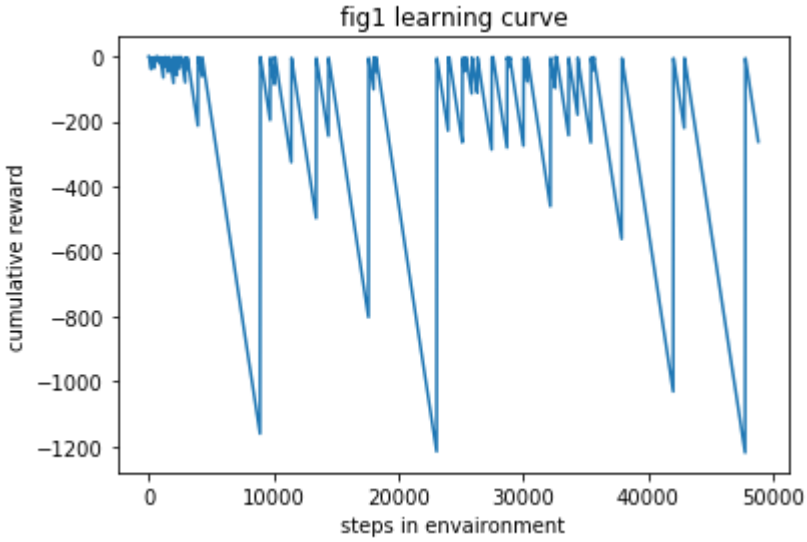
```
60
61 print("finished")
62 learning_curve=np.array(learning_curve)
63 learning_curve=learning_curve.T
```

```
epoch= 20 loss= 0.00044829276
epoch= 30 loss= 3.8358714e-05
epoch= 40 loss= 0.0006216031
epoch= 50 loss= 9.5349577e-07
epoch= 60 loss= 6.2366175e-06
epoch= 70 loss= 2.3341701e-05
epoch= 80 loss= 3.7558806e-05
epoch= 90 loss= 1.4010259e-05
finished
```

Plot Learning Curve

In [150]:

```
1 plt.title("fig1 learning curve")
2 plt.xlabel("steps in envaironment")
3 plt.ylabel("cumulative reward")
4 plt.plot(learning_curve[0],learning_curve[1])
5 plt.show()
```



Conclusion

急激に値が上昇している点はエポックの移り変わりを示すが，むしろ学習初期の方がエポック間のstep数が少ない(=ソートが早く終わっている)ことがわかる．学習がうまくいっていないようである．残念ながらレポート提出までに適切なパラメータを見つけることができなかった．

Advance

N-1個の行動に加えて，自らタスクを終了させる行動を加え，終了時点で正しくソートできていれば報酬を+1，できていなければ報酬を+0.

In [117]:

```
class DQN_SORT_ADV:
    def __init__(self,inputnum):
        self.N=inputnum
        #隠れ層1のニューロン数
        self.M=100
        #隠れ層2のニューロン数
        self.L=100
        #行動数
        self.Nact=self.N
        #ミニバッチ数
        self.minibatch_size = 32
        #キューのサイズ
        self.replay_memory_size = 10000
        self.epsilon=0.8
        # replay memory
        self.P = deque(maxlen=self.replay_memory_size)

        # model
        self.init_model()

        # variables
        self.current_loss = 0.0


    def init_model(self):
        # (0) 入力文字列
        self.x = tf.placeholder(tf.float32, [None, self.N],name='x')

        # (1) サイズ変更
        x1 = tf.reshape(self.x, [-1,self.N])

        # (2) 隠れ層1
        # 重みとバイアス
        self.w1 = tf.Variable(tf.truncated_normal([self.N,self.M],stddev=0.01))
        self.b1 = tf.Variable(tf.zeros([self.M]))
        x2 = tf.nn.relu( tf.matmul(x1, self.w1) + self.b1)

        # (3) 隠れ層2
        # 重みとバイアス
        self.w2 = tf.Variable(tf.truncated_normal([self.M,self.L],stddev=0.01))
        self.b2 = tf.Variable(tf.zeros([self.L]))
        x3 = tf.nn.relu( tf.matmul(x2,self.w2) + self.b2)

        # (3) 出力層
        # 重みとバイアス
        self.w3 = tf.Variable(tf.truncated_normal([self.L,self.Nact],stddev=0.01))
        self.b3 = tf.Variable(tf.zeros([self.Nact]))
        self.Qs = tf.matmul(x3, self.w3) + self.b3


        ##target network
        # (0) 入力文字列
        self.xtar = tf.placeholder(tf.float32, [None, self.N],name='x')

        # (1) サイズ変更
        x1tar = tf.reshape(self.xtar, [-1,self.N])

       # (2) 隠れ層1
        # 重みとバイアス
```

```
        self.w1tar = tf.Variable(self.w1)
        self.b1tar = tf.Variable(self.b1)
        x2tar = tf.nn.relu( tf.matmul(x1tar, self.w1tar) + self.b1tar)

        # (3) 隠れ層2
        # 重みとバイアス
        self.w2tar = tf.Variable(self.w2)
        self.b2tar = tf.Variable(self.b2)
        x3tar = tf.nn.relu( tf.matmul(x2tar, self.w2tar) + self.b2tar)

        # (3) 出力層
        # 重みとバイアス
        self.w3tar = tf.Variable(self.w3)
        self.b3tar = tf.Variable(self.b3)
        self.Qstar = tf.matmul(x3tar, self.w3tar) + self.b3tar


        # 損失関数
        self.y = tf.placeholder(tf.float32, [None])#バッチサイズ分の長さをもつ
        self.act_indices=tf.placeholder(tf.int32, [None,2])#選択アクションを含むindex
        self.Q= tf.gather_nd(self.Qs,self.act_indices)#バッチサイズ分の長さをもつQ値

        self.loss = tf.reduce_mean(tf.losses.huber_loss(self.y,self.Q))

        # train operation
        optimizer = tf.train.AdamOptimizer()
        self.training = optimizer.minimize(self.loss)

        # session
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())

    def Qs_func(self, state):
        res=self.sess.run(self.Qs, feed_dict={self.x: [state]})
        return res[0]

    def Qstar_func(self,state):
        res=self.sess.run(self.Qstar, feed_dict={self.xtar: [state]})
        return res[0]

    def target_update(self):
        tf.assign(self.w1tar,self. w1)
        tf.assign(self.b1tar, self.b1)
        tf.assign(self.w2tar, self.w2)
        tf.assign(self.b2tar, self.b2)
        tf.assign(self.w3tar, self.w3)
        tf.assign(self.b3tar, self.b3)

    def select_action(self, state):
        if np.random.rand() <= self.epsilon:
            # random
            return np.random.randint(self.Nact)
        else:
            # max_action Q(state, action)
            return np.argmax(self.Qs_func(state))

    def store_experience(self, state, action, reward, state_1, terminal):
        self.P.append((state, action, reward, state_1, terminal))

    def experience_replay(self):
        state_minibatch = []
```

```
        y_minibatch = []
        act_indices=[]
        index=0

        # sample random minibatch
        # キューに十分にたまるまでは全部使う
        minibatch_size = min(len(self.P), self.minibatch_size)
        minibatch_indexes = np.random.choice(np.arange(len(self.P)), minibatch_size,replace=False)

        for j in minibatch_indexes:
            state_j, action_j, reward_j, state_j_1, terminal = self.P[j]

            y_j= 0

            if terminal:
                y_j= reward_j
            else:
                y_j= reward_j + np.max(self.Qstar_func(state_j_1))

            state_minibatch.append(state_j)
            y_minibatch.append(y_j)
            act_indices.append([index,action_j])
            index+=1

        # training
        self.sess.run(self.training, feed_dict={self.x: state_minibatch, self.y: y_minibatch, self.act_in
        # for log
        self.current_loss = self.sess.run(self.loss, feed_dict={self.x: state_minibatch, self.y: y_miniba
```

In [118]:

```
class StateArrayAdv:
    def __init__(self,N):
        self.N=N
        self.arr=np.random.randint(0,9,N)
        self.frame=0
        self.reward=-1*2/(self.N*(self.N-1))
        self.terminal=False

    def excute_action(self,action):
        if action!=self.N-1:
            temp=self.arr[action]
            self.arr[action]=self.arr[action+1]
            self.arr[action+1]=temp
        else:
            self.terminal=True

    def is_sorted(self):
        res=True
        for i in range(self.N-1):
            res=(self.arr[i]<=self.arr[i+1]) and res
        return res

    def observe(self):
        self.reward=-1*2/(self.N*(self.N-1))
        if self.terminal:
            if self.is_sorted():
                self.reward=1
            else:
                self.reward=0

        return self.arr,self.reward,self.terminal
```

In [119]:

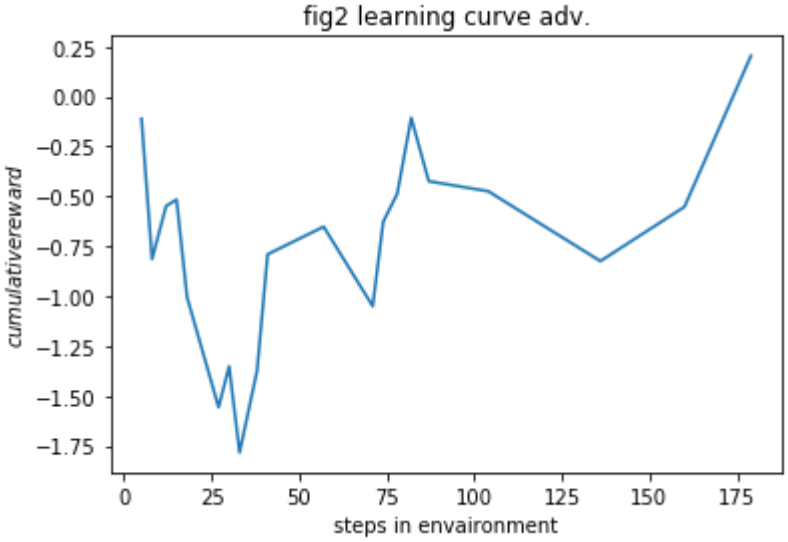
```
n_epochs=5000
dqn_adv = DQN_SORT_ADV(N)

count=0

learning_curve_adv=[]
average=0.0

N=4

A=1024
B=1024

for e in range(n_epochs):
    # reset
    frame=0
    cum_reward=0.0
    sa=StateArrayAdv(N)
    state_t_1,reward_t,terminal=sa.observe()

    while not terminal:
        state_t = state_t_1

        # execute action in environment
        action_t = dqn_adv.select_action(state_t)
        sa.excute_action(action_t)

        # observe environment
        state_t_1, reward_t, terminal = sa.observe()

        # store experience
        dqn_adv.store_experience(state_t, action_t, reward_t, state_t_1, terminal)

        # experience replay
        if count>A:
            dqn_adv.experience_replay()


        if count%B==0:
            dqn_adv.target_update()


        cum_reward+=reward_t
        average+=cum_reward

        if frame%buff==buff-1:
            average/=buff
            learning_curve_adv.append([count,average])

        count+=1
        frame+=1

    dqn_adv.epsilon*=decrese
    if e%100==0 and count>A:
        print("epoch=",e,"loss=",dqn_adv.current_loss)

print("finished")
learning_curve_adv=np.array(learning_curve_adv)
learning_curve_adv=learning_curve_adv.T
```

```
epoch= 1000 loss= 0.0015185372
epoch= 1100 loss= 0.00039484084
epoch= 1200 loss= 0.01871253
epoch= 1300 loss= 0.012573384
epoch= 1400 loss= 0.00016127527
epoch= 1500 loss= 0.021034678
epoch= 1600 loss= 0.026605979
epoch= 1700 loss= 0.015753468
epoch= 1800 loss= 0.014635619
epoch= 1900 loss= 7.38577e-06
epoch= 2000 loss= 0.0016783845
epoch= 2100 loss= 0.009081308
epoch= 2200 loss= 6.0189937e-05
epoch= 2300 loss= 0.014398257
epoch= 2400 loss= 0.0062923785
epoch= 2500 loss= 0.00012045413
epoch= 2600 loss= 0.0075981985
epoch= 2700 loss= 3.6846986e-06
epoch= 2800 loss= 0.00081098726
epoch= 2900 loss= 9.89033e-06
epoch= 3000 loss= 0.0011454413
epoch= 3100 loss= 0.00040335642
epoch= 3200 loss= 0.0006826106
epoch= 3300 loss= 0.002611473
epoch= 3400 loss= 0.00024435957
epoch= 3500 loss= 0.00018714493
epoch= 3600 loss= 0.0001550679
epoch= 3700 loss= 9.0620296e-07
epoch= 3800 loss= 4.152867e-05
epoch= 3900 loss= 7.973924e-05
epoch= 4000 loss= 1.8670858e-05
epoch= 4100 loss= 1.2622099e-06
epoch= 4200 loss= 0.00012013019
epoch= 4300 loss= 0.00052570656
epoch= 4400 loss= 9.479394e-06
epoch= 4500 loss= 1.4638972e-06
epoch= 4600 loss= 5.8158326e-05
epoch= 4700 loss= 2.265943e-05
epoch= 4800 loss= 2.5988376e-05
epoch= 4900 loss= 2.4094043e-06
finished
```

Plot Learning Curve

In [120]:

```
plt.title("fig2 learning curve adv.")
plt.xlabel("steps in envaironment")
plt.ylabel("$cumulative reward$")
plt.plot(learning_curve_adv[0],learning_curve_adv[1])
plt.show()
```



Conclusion

今度は学習がうまくいってる．また，終了することでstep数が抑えられ，非常に早く学習が終了する．step数を抑えることで，累積したマイナスの報酬が抑えられ，成功報酬の+1が有効に機能している．終了行動のない場合は累積したマイナスの報酬があまりにも大きな値になり，成功報酬+1と比較して相対的にほとんど効果をなしていなかった．その点こちらはこの効果が反映され学習にいい影響を及ぼしていると考えられる．