#include "Pong/DQN\_Manager/DQN\_Manager.h"

#include "Pong/Environment/Environment.h"

void UDQN\_Manager::Init(AEnvironment\* InOwner)

{

Environment = InOwner;

}

void UDQN\_Manager::Build(AActor\* inplyr, int innum, TArray<int> inLayS, TArray<FString> inLayT, double ineps, double inedecay, double inesdecay, double inmine, double inLR, double inLRd, double inminLR, double ingamma, int inbufsiz, int inbatch, int trenew, int trainrt, int intrainsteps) {

plyr = inplyr;

num = innum;

LayS = inLayS;

LayT = inLayT;

epsilon = ineps;

epsilon\_decay = inedecay;

epsilon\_stepdecay = inesdecay;

minepsilon = inmine;

LRate = inLR;

LRate\_decay = inLRd;

minLRate = inminLR;

gamma = ingamma;

BufferSize = inbufsiz;

batch = inbatch;

TargetRenewRate = trenew;

TrainRate = trainrt;

TotalTrainSteps = intrainsteps;

PolicyNN.build(LayS.Num(), LRate, LayS, LayT);

TargetNN.FullCopy(PolicyNN);

Buffer.Build(BufferSize);

}

void UDQN\_Manager::DQN()

{

TArray<double> input = Environment->GetState(num);

if (steps % 10 == 0) {

UE\_LOG(LogTemp, Warning, TEXT("Step: %d"),steps);

}

int act;

double ran = double(FMath::Rand()) /double (RAND\_MAX);

if (ran <= epsilon) {

//Random Action.

UE\_LOG(LogTemp, Display, TEXT("Accion aleatoria, epsilon = %f"), epsilon);

act = FMath::RandRange(0, 2);

}

else {

//Choose best action

UE\_LOG(LogTemp, Display, TEXT("Accion no aleatoria"));

TArray<double> out = PolicyNN.run(input);

double mx = out[0];

act = 0;

for (int i = 1; i < out.Num(); i++) {

if (mx < out[i]) {

mx = out[i];

act = i;

}

}

}

DoAction(act);

double reward = 0;

bool Done = Environment->GetReward(num, reward);

TArray<double> outstate = Environment->GetState(num);

FDataPiece data(input, act, reward, outstate, Done);

Buffer.Add(data);

steps++;

if (Buffer.current\_size >= batch and Buffer.current\_size>=Buffer.max\_size/5 and steps%TrainRate==0) {

train();

}

if (steps % TargetRenewRate == 0) {

TargetNN.FullCopy(PolicyNN);

}

//epsilon = FMath::Max(minepsilon, epsilon \* epsilon\_decay);

epsilon = FMath::Max(minepsilon, epsilon - epsilon\_stepdecay);

}

void UDQN\_Manager::DoAction(int act) {

if (act == 0) {

return;

}

if (act == 1) {

Environment->MoveRacket(plyr, 1.0);

return;

}

if (act==2) {

Environment->MoveRacket(plyr, -1.0);

return;

}

}

void UDQN\_Manager::train() {

TArray<FDataPiece> data = Buffer.RandomSample(batch);

double loss = 0;

PolicyNN.TotalClean();

for (int i = 0; i < batch; i++) {

double expected = data[i].reward;

if (!data[i].Done) {

TArray<double> targetout = TargetNN.run(data[i].OutputState);

TArray<double> policyout = PolicyNN.run(data[i].OutputState);

int mxact = 0;

double mx = policyout[0];

for (int j = 1; j < policyout.Num(); j++) {

if (mx < policyout[j]) {

mxact = j;

mx = policyout[j];

}

}

expected += gamma \* targetout[mxact];

}

PolicyNN.Clean();

TArray<double> ans = PolicyNN.run(data[i].InputState);

loss += (expected - ans[data[i].action]) \* (expected - ans[data[i].action]);

if (i == 0) {

UE\_LOG(LogTemp, Display, TEXT("Expected %f, answered %f"), expected, ans[data[i].action]);

}

PolicyNN.SingleBackprop(expected, data[i].action, batch);

}

PolicyNN.GradientClipping(1.0);

PolicyNN.Adam();

loss /= double(batch);

UE\_LOG(LogTemp, Display, TEXT("Loss = %f"), loss);

LRate = FMath::Max(minLRate, LRate \* LRate\_decay);

}