Introduction to Artificial Intelligence Course Report

With OpenAI Gym Virtual agents

李平山

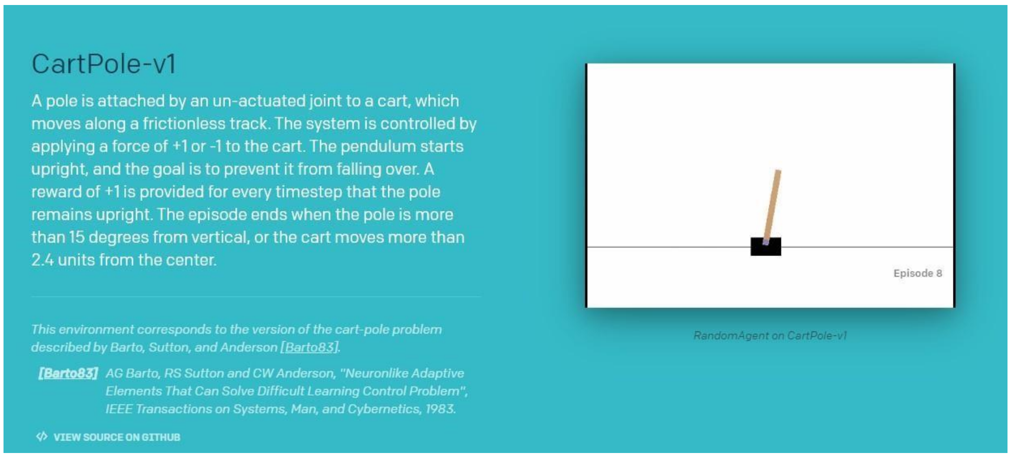
2017201980

Contents:

1. First attempt of agents with predefined rules

1.1 introduction to the environment 'CartPole-v1'

Using 'CartPole-v1' environment in the OpenAI gym, game introduction:



CartPole-v1

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [Barto83].

[Barto83] AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983.

VIEW SOURCE ON GITHUB

[Retrieved from https://gym.openai.com/envs/CartPole-v1/]

- Observation(object): an environment-specific object representing your observation of the environment.
- Reward(float): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- Done(boolean): whether it’s time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated.\
- Info(dict): diagnostic information useful for debugging. It can sometimes be useful for learning

We can get the variables in the CartPole-v1:

Observation:

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-24 deg	24 deg
3	Pole Velocity At Tip	-Inf	Inf

Actions:

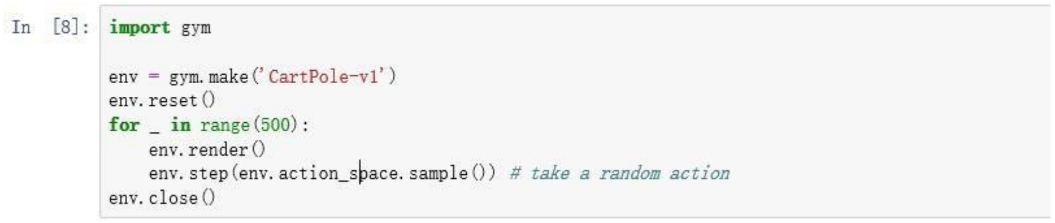
Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

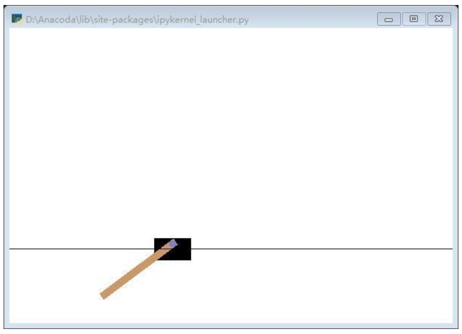
Test Environment: windows 10, python 3,

1.2 A random agent on ‘CartPole-v1’:

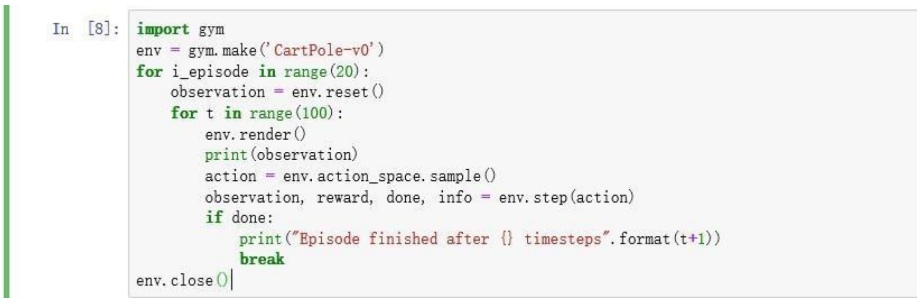
```
action = env.action_space.sample()
```



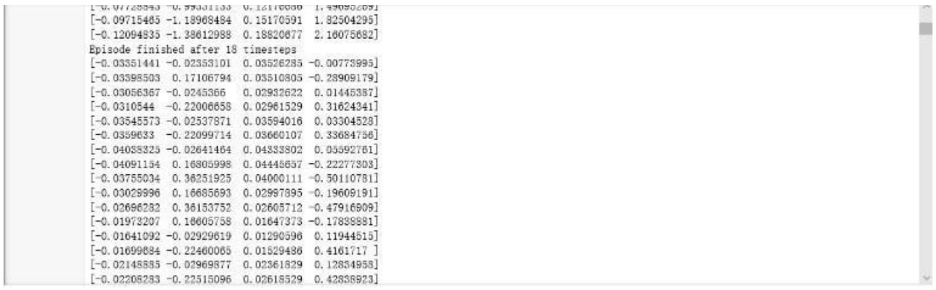
```
In  [8]: import gym

         env = gym.make('CartPole-v1')
         env.reset()
         for _ in range(500):
             env.render()
             env.step(env.action_space.sample()) # take a random action
         env.close()
```



results on random agents



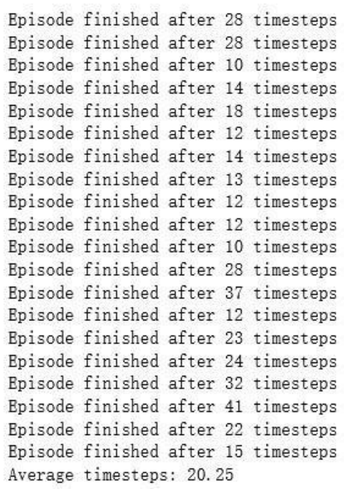
```
In  [8]: import gym
         env = gym.make('CartPole-v0')
         for i_episode in range(20):
             observation = env.reset()
             for t in range(100):
                 env.render()
                 print(observation)
                 action = env.action_space.sample()
                 observation, reward, done, info = env.step(action)
                 if done:
                     print("Episode finished after {} timesteps".format(t+1))
                     break
         env.close()
```



```
[-0.09715465 -1.18968484  0.15170591  1.82504295]
[-0.12094835 -1.38612988  0.18820677  2.16075682]
Episode finished after 18 timesteps
[-0.03351441 -0.02353101  0.03526285 -0.00773995]
[-0.03398503  0.17106794  0.03510805 -0.28909179]
[-0.03056367 -0.0245366   0.02932622  0.01445387]
[-0.0310544  -0.22006658  0.02961529  0.31624341]
[-0.03545573 -0.02537871  0.03594016  0.03304528]
[-0.0359633  -0.22099714  0.03660107  0.33684756]
[-0.04038325 -0.02641464  0.04333802  0.05592761]
[-0.04091154  0.16805998  0.04445657 -0.22277303]
[-0.03755034  0.36251925  0.04000111 -0.50110781]
[-0.03029996  0.16685693  0.02997895 -0.19609191]
[-0.02696282  0.36153752  0.02605712 -0.47916909]
[-0.01973207  0.16605758  0.01647373 -0.17838881]
[-0.01641092 -0.02929619  0.01290596  0.11944515]
[-0.01699884 -0.22460065  0.01529486  0.4161717 ]
[-0.02148885 -0.02969877  0.02361829  0.12834958]
[-0.02208283 -0.22515096  0.02618529  0.42838923]
```

[a snapshot on the output observations]

Count the timesteps of random agent after 20 episodes:



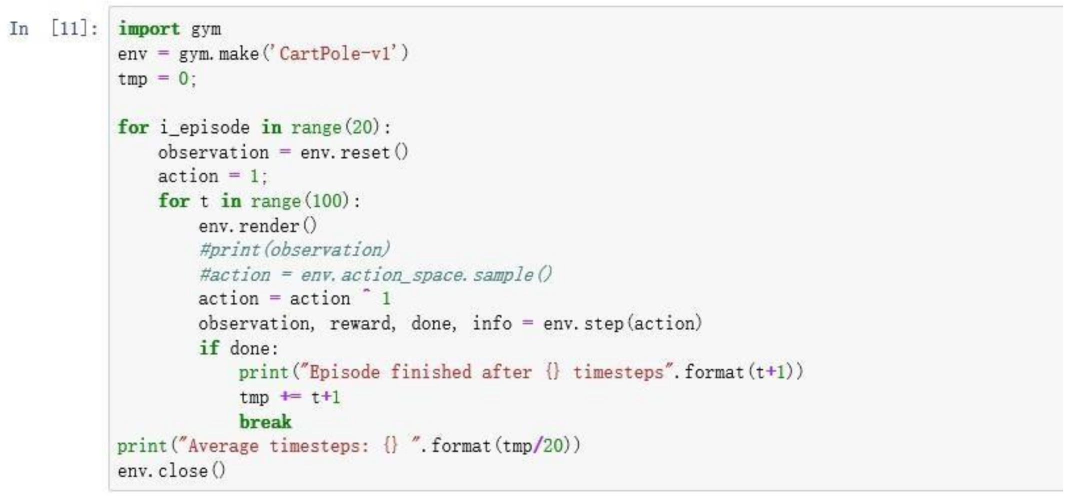
```
Episode finished after 28 timesteps
Episode finished after 28 timesteps
Episode finished after 10 timesteps
Episode finished after 14 timesteps
Episode finished after 18 timesteps
Episode finished after 12 timesteps
Episode finished after 14 timesteps
Episode finished after 13 timesteps
Episode finished after 12 timesteps
Episode finished after 12 timesteps
Episode finished after 10 timesteps
Episode finished after 28 timesteps
Episode finished after 37 timesteps
Episode finished after 12 timesteps
Episode finished after 23 timesteps
Episode finished after 24 timesteps
Episode finished after 32 timesteps
Episode finished after 41 timesteps
Episode finished after 22 timesteps
Episode finished after 15 timesteps
Average timesteps: 20.25
```

We can see the average result on random agent: 20.25

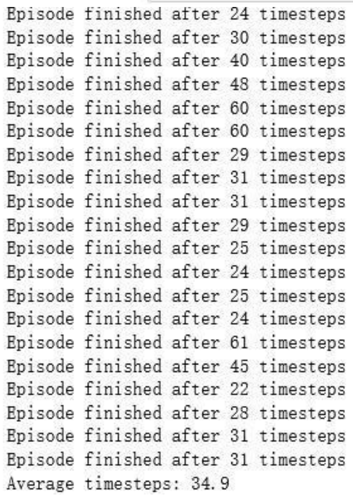
1.3 An agent based on greedy rules:

- A very simple naïve idea： actions of the agent cart changes frame-by-frame based and only based on the last action it took, heading to it’s opposite direction.

$$ACTIONn+1 = ACTIONN \,\hat{}\, 1$$



```
In  [11]: import gym
          env = gym.make('CartPole-v1')
          tmp = 0;

          for i_episode in range(20):
              observation = env.reset()
              action = 1;
              for t in range(100):
                  env.render()
                  #print(observation)
                  #action = env.action_space.sample()
                  action = action ^ 1
                  observation, reward, done, info = env.step(action)
                  if done:
                      print("Episode finished after {} timesteps".format(t+1))
                      tmp += t+1
                      break
          print("Average timesteps: {} ".format(tmp/20))
          env.close()
```



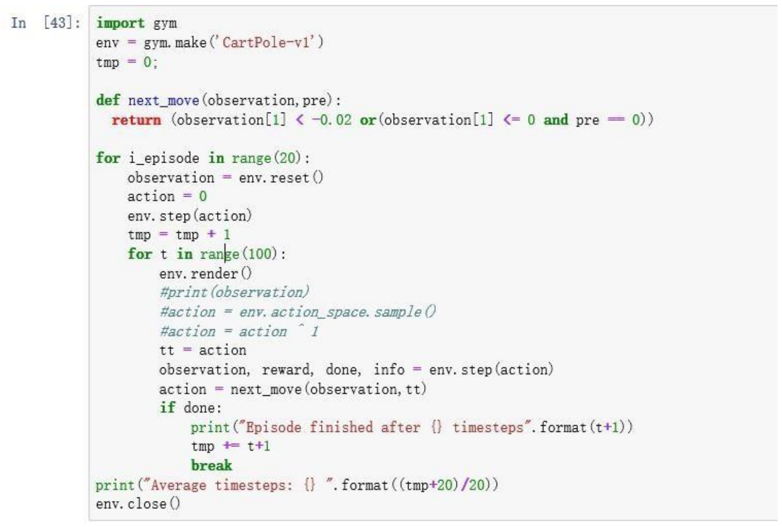
```
Episode finished after 24 timesteps
Episode finished after 30 timesteps
Episode finished after 40 timesteps
Episode finished after 48 timesteps
Episode finished after 60 timesteps
Episode finished after 60 timesteps
Episode finished after 29 timesteps
Episode finished after 31 timesteps
Episode finished after 31 timesteps
Episode finished after 29 timesteps
Episode finished after 25 timesteps
Episode finished after 24 timesteps
Episode finished after 25 timesteps
Episode finished after 24 timesteps
Episode finished after 61 timesteps
Episode finished after 45 timesteps
Episode finished after 22 timesteps
Episode finished after 28 timesteps
Episode finished after 31 timesteps
Episode finished after 31 timesteps
Average timesteps: 34.9
```

Still, count the timesteps of greedy agent after 20 episodes. We can see the average result on random agent: 34.9: Better performance.

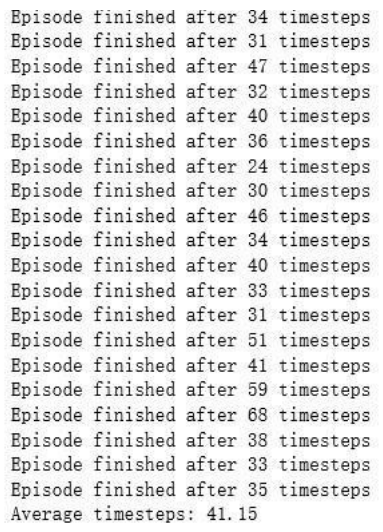
1.4 An agent based on predefined rules:

Define function next move, based on pre_action and last_observation:

```
def next_move(observation,pre):

 return (observation[1] < -0.02 or(observation[1] <= 0 and pre == 0))
```



```
In  [43]: import gym
          env = gym.make('CartPole-v1')
          tmp = 0;

          def next_move(observation, pre):
            return (observation[1] < -0.02 or(observation[1] <= 0 and pre == 0))

          for i_episode in range(20):
              observation = env.reset()
              action = 0
              env.step(action)
              tmp = tmp + 1
              for t in range(100):
                  env.render()
                  #print(observation)
                  #action = env.action_space.sample()
                  #action = action ^ 1
                  tt = action
                  observation, reward, done, info = env.step(action)
                  action = next_move(observation, tt)
                  if done:
                      print("Episode finished after {} timesteps".format(t+1))
                      tmp += t+1
                      break
          print("Average timesteps: {} ".format((tmp+20)/20))
          env.close()
```



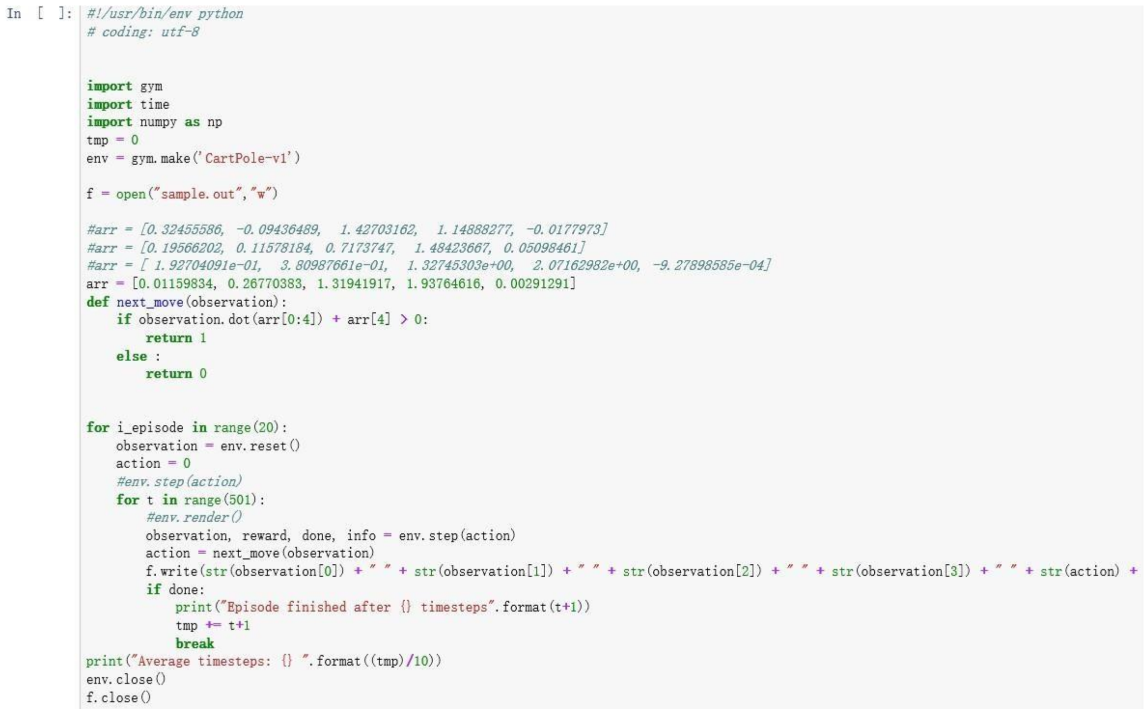
```
Episode finished after 34 timesteps
Episode finished after 31 timesteps
Episode finished after 47 timesteps
Episode finished after 32 timesteps
Episode finished after 40 timesteps
Episode finished after 36 timesteps
Episode finished after 24 timesteps
Episode finished after 30 timesteps
Episode finished after 46 timesteps
Episode finished after 34 timesteps
Episode finished after 40 timesteps
Episode finished after 33 timesteps
Episode finished after 31 timesteps
Episode finished after 51 timesteps
Episode finished after 41 timesteps
Episode finished after 59 timesteps
Episode finished after 68 timesteps
Episode finished after 38 timesteps
Episode finished after 33 timesteps
Episode finished after 35 timesteps
Average timesteps: 41.15
```

We can see the average result on random agent: 41.15: about the same result as greedy_agent, better performance than random_agent.

2. Machine Learning Agents with binary classification

For every decision making of action, observation variable is and is the only that matters. We use a well-trained agent in OpenAI platform(obtained from https://gym.openai.com/envs/CartPole-v0/) to be the well-performed example to generator data.

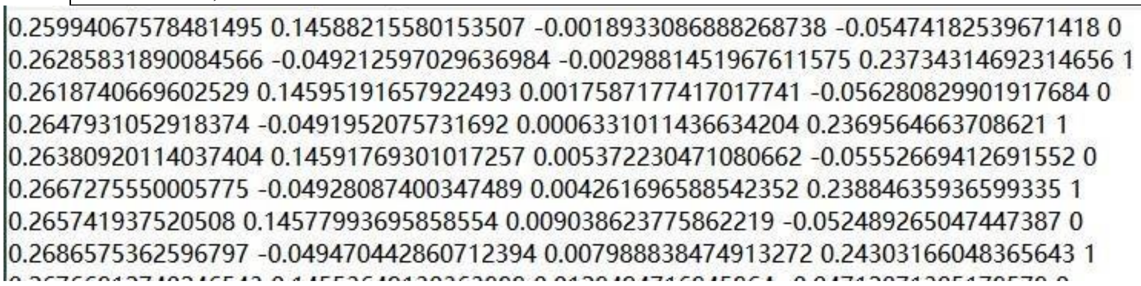
And use following program to generator 50,000 data



```
In [ ]: #!/usr/bin/env python
        # coding: utf-8


        import gym
        import time
        import numpy as np
        tmp = 0
        env = gym.make('CartPole-v1')

        f = open("sample.out","w")

        #arr = [0.32455586, -0.09436489,  1.42703162,  1.14888277, -0.0177973]
        #arr = [0.19566202, 0.11578184, 0.7173747,  1.48423667, 0.05098461]
        #arr = [ 1.92704091e-01,  3.80987661e-01,  1.32745303e+00,  2.07162982e+00, -9.27898585e-04]
        arr = [0.01159834, 0.26770383, 1.31941917, 1.93764616, 0.00291291]
        def next_move(observation):
            if observation.dot(arr[0:4]) + arr[4] > 0:
                return 1
            else :
                return 0


        for i_episode in range(20):
            observation = env.reset()
            action = 0
            #env.step(action)
            for t in range(501):
                #env.render()
                observation, reward, done, info = env.step(action)
                action = next_move(observation)
                f.write(str(observation[0]) + " " + str(observation[1]) + " " + str(observation[2]) + " " + str(observation[3]) + " " + str(action) +
                if done:
                    print("Episode finished after {} timesteps".format(t+1))
                    tmp += t+1
                    break
        print("Average timesteps: {} ".format((tmp)/10))
        env.close()
        f.close()
```

Test data is saved in sample.out in formation

Observation, action



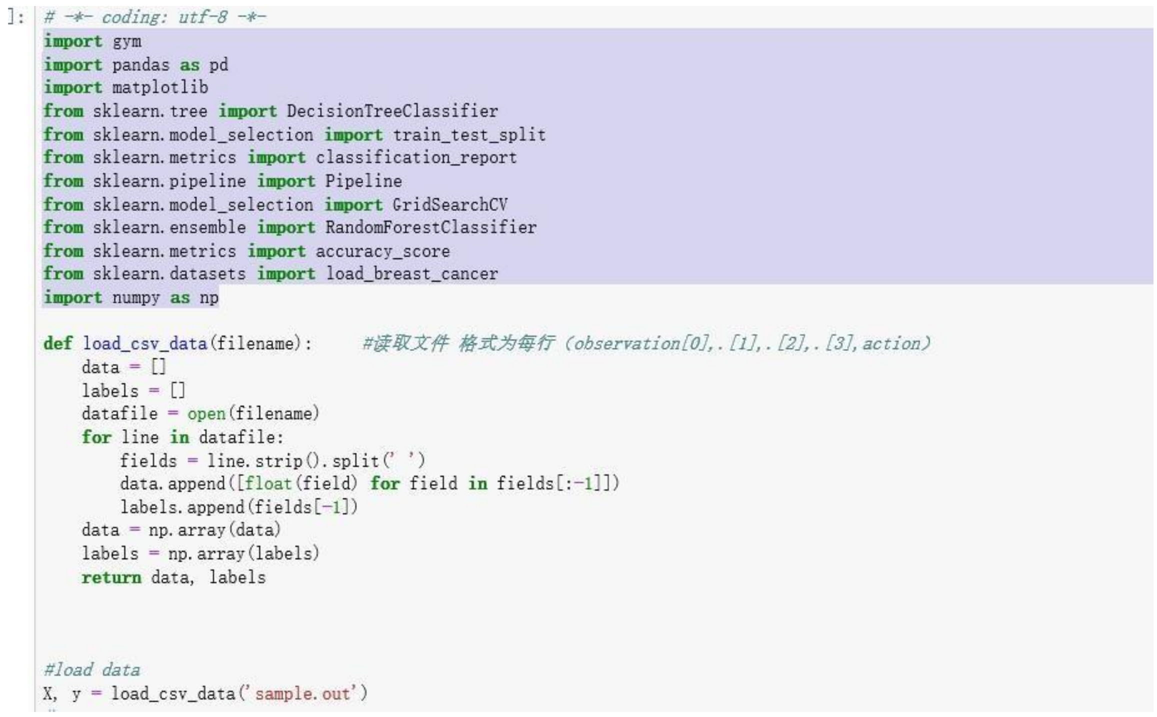
```
0.25994067578481495 0.14588215580153507 -0.0018933086888268738 -0.05474182539671418 0
0.26285831890084566 -0.049212597029636984 -0.0029881451967611575 0.23734314692314656 1
0.2618740669602529 0.14595191657922493 0.0017587177417017741 -0.056280829901917684 0
0.2647931052918374 -0.0491952075731692 0.0006331011436634204 0.2369564663708621 1
0.26380920114037404 0.14591769301017257 0.005372230471080662 -0.05552669412691552 0
0.2667275550005775 -0.04928087400347489 0.004261696588542352 0.23884635936599335 1
0.265741937520508 0.14577993695858554 0.009038623775862219 -0.05248926504744387 0
0.2686575362596797 -0.049470442860712394 0.007988838474913272 0.24303166048365643 1
```

2.1 Binary Classification and Cross-validation on test data

For binary classification, we are interested in classifying data into 0's and 1's in our data as action described in CartPole environment, using observation as features in classification. In this matter, we use different methods on test data:

- SVM Classifier
- Logistic Regression Classifier
- RandomForest
- voting_classify with methods above

Before training, we set the environment using sklearn:



```
# -*- coding: utf-8 -*-
import gym
import pandas as pd
import matplotlib
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer
import numpy as np

def load_csv_data(filename):     #读取文件 格式为每行 (observation[0],.[1],.[2],.[3],action)
    data = []
    labels = []
    datafile = open(filename)
    for line in datafile:
        fields = line.strip().split(' ')
        data.append([float(field) for field in fields[:-1]])
        labels.append(fields[-1])
    data = np.array(data)
    labels = np.array(labels)
    return data, labels


#load data
X, y = load_csv_data('sample.out')
```

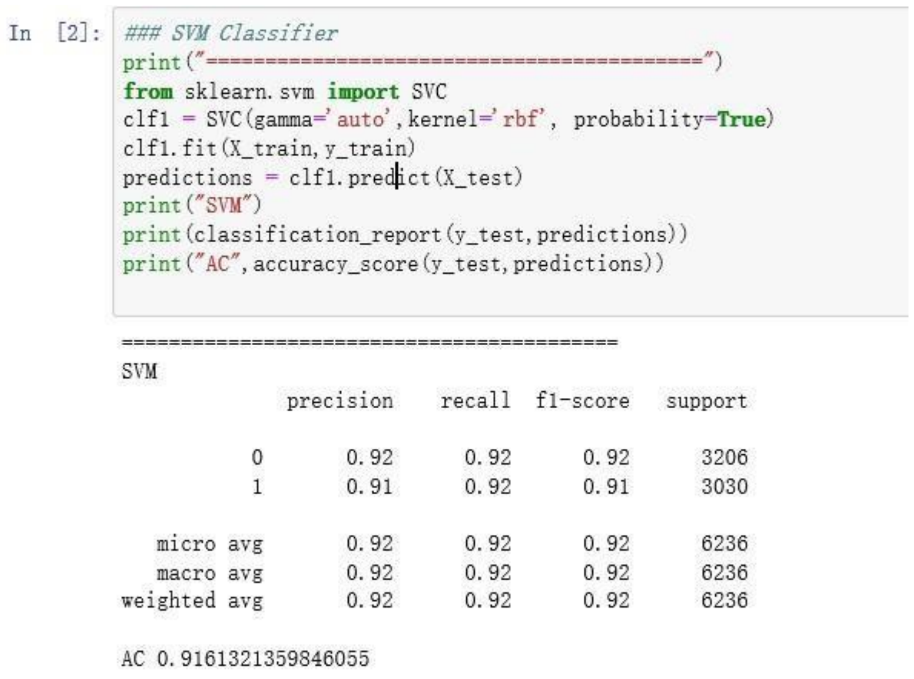
We use train_test_split for Cross-validation on test data:

(Split arrays or matrices into random train and test subsets)

```
X_train,X_test,y_train,y_test = train_test_split(X,y)
```

2.2 SVM Classifier and its performance:

Support Vector Machines (SVMs) are a type of classification algorithm that are more flexible - they can do linear classification, but can use other non-linear *basis functions*. The following uses a linear classifier to fit the observation-action pattern that separates the data into 0‘s and 1‘s:

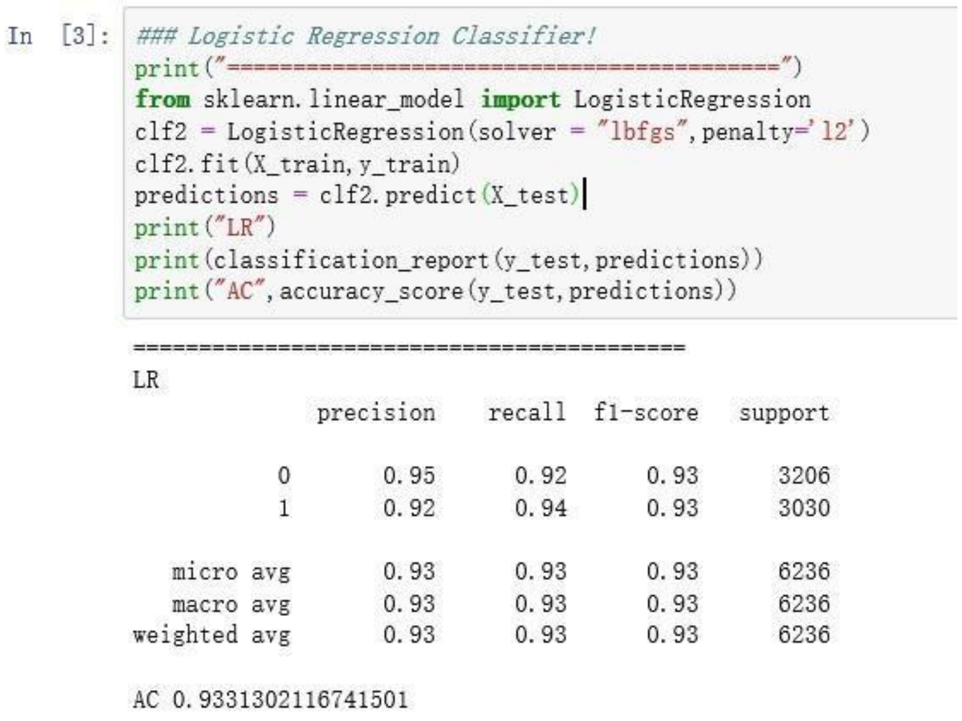


```
In  [2]: ### SVM Classifier
         print("==========================================")
         from sklearn.svm import SVC
         clf1 = SVC(gamma='auto',kernel='rbf', probability=True)
         clf1.fit(X_train,y_train)
         predictions = clf1.predict(X_test)
         print("SVM")
         print(classification_report(y_test,predictions))
         print("AC",accuracy_score(y_test,predictions))

         ==========================================
         SVM
                       precision    recall  f1-score   support

                    0       0.92      0.92      0.92      3206
                    1       0.91      0.92      0.91      3030

            micro avg       0.92      0.92      0.92      6236
            macro avg       0.92      0.92      0.92      6236
         weighted avg       0.92      0.92      0.92      6236

         AC 0.9161321359846055
```

Accuracy_score for SVM classifier: 0.9161321359846055

2.3 Logistic Regression Classifier and its performance:`

Logistic Regression is a type of Generalized Linear Model (GLM) that uses a logistic function to model a binary variable based on any kind of independent variables.

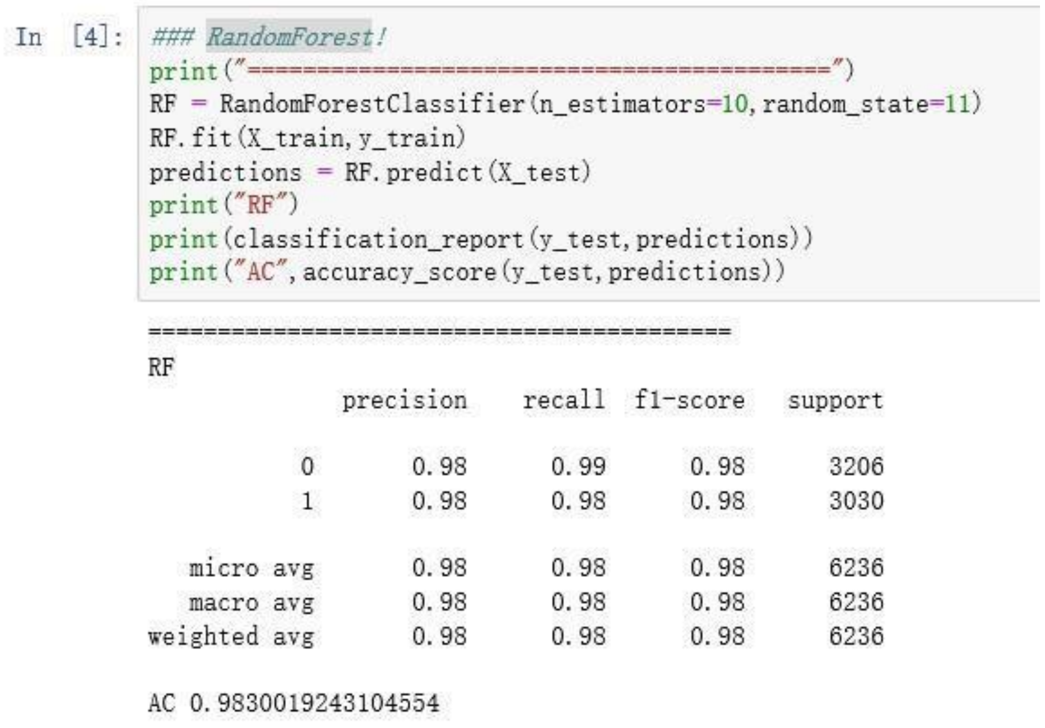


```
In  [3]: ### Logistic Regression Classifier!
         print("==========================================")
         from sklearn.linear_model import LogisticRegression
         clf2 = LogisticRegression(solver = "lbfgs",penalty='l2')
         clf2.fit(X_train,y_train)
         predictions = clf2.predict(X_test)
         print("LR")
         print(classification_report(y_test,predictions))
         print("AC",accuracy_score(y_test,predictions))

         ==========================================
         LR
                       precision    recall  f1-score   support

                    0       0.95      0.92      0.93      3206
                    1       0.92      0.94      0.93      3030

            micro avg       0.93      0.93      0.93      6236
            macro avg       0.93      0.93      0.93      6236
         weighted avg       0.93      0.93      0.93      6236

         AC 0.9331302116741501
```

Accuracy_score for Logistic Regression classifier: 0.9331302116741501

2.4 RandomForest and its performance:

Random Forests are an ensemble learning method that fit multiple Decision Trees on subsets of the data and average the results. We can again fit them using sklearn, and use them to predict outcomes, as well as get mean prediction accuracy



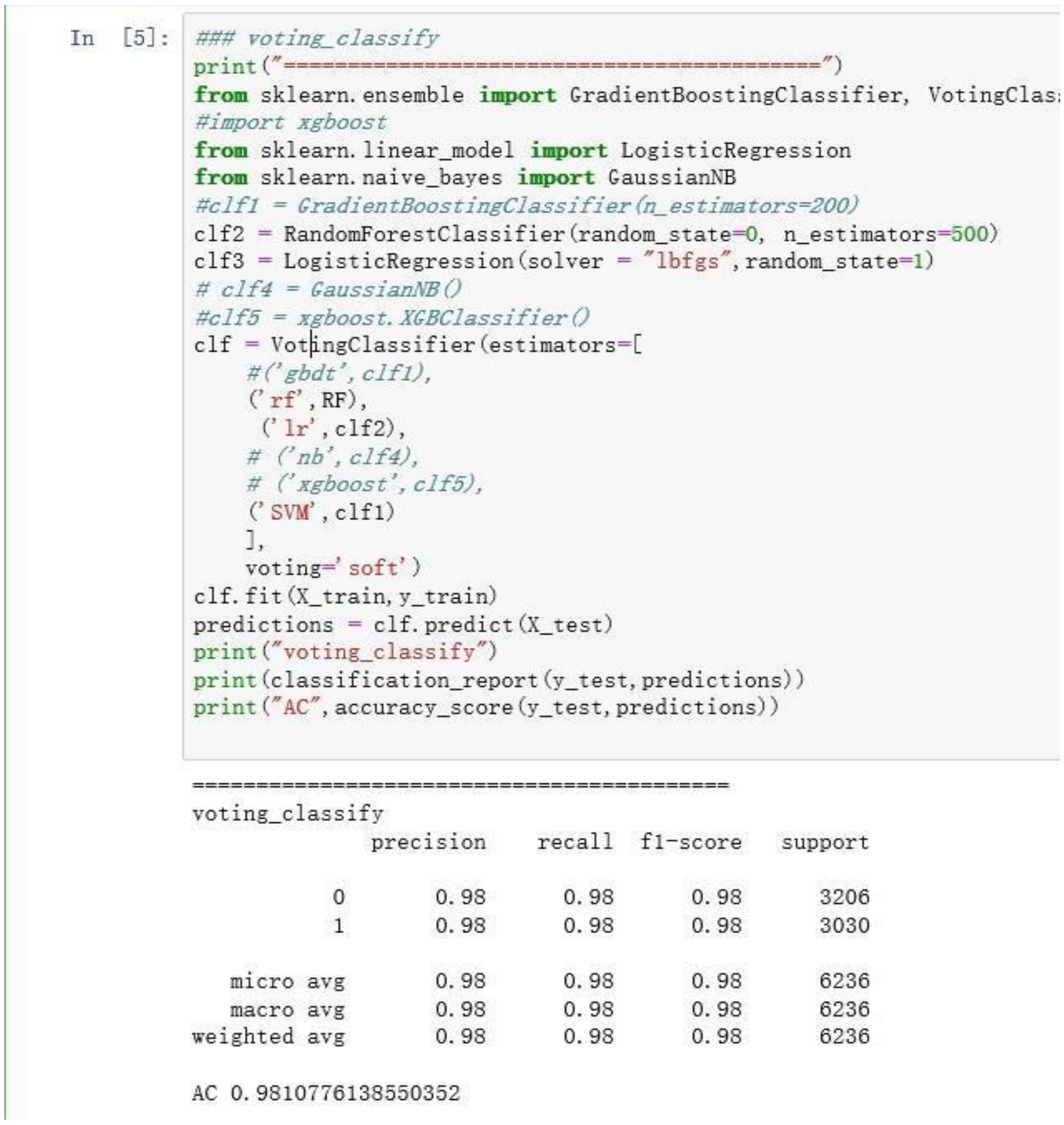
```
In  [4]: ### RandomForest!
         print("==========================================")
         RF = RandomForestClassifier(n_estimators=10,random_state=11)
         RF.fit(X_train,y_train)
         predictions = RF.predict(X_test)
         print("RF")
         print(classification_report(y_test,predictions))
         print("AC",accuracy_score(y_test,predictions))
```

```
==========================================
RF
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      3206
           1       0.98      0.98      0.98      3030

   micro avg       0.98      0.98      0.98      6236
   macro avg       0.98      0.98      0.98      6236
weighted avg       0.98      0.98      0.98      6236

AC 0.9830019243104554
```

Accuracy_score for RandomForest: 0.9830019243104554

2.5 Voting Classifier and its performance:`

In this matter, we combines the predictions from multiple machine learning algorithms (3 classifier discussed above). Mark that Voting classifier isn't an actual classifier but a wrapper for set of different ones that are trained and valuated in parallel in order to exploit the different peculiarities of each algorithm.



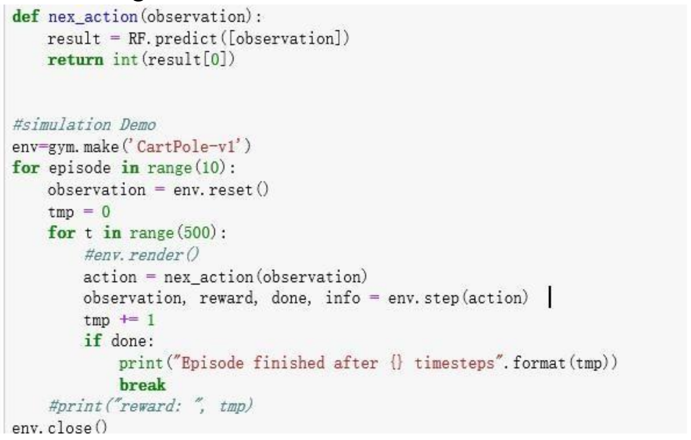
```
In  [5]: ### voting_classify
         print("===========================================")
         from sklearn.ensemble import GradientBoostingClassifier, VotingClas
         #import xgboost
         from sklearn.linear_model import LogisticRegression
         from sklearn.naive_bayes import GaussianNB
         #clf1 = GradientBoostingClassifier(n_estimators=200)
         clf2 = RandomForestClassifier(random_state=0, n_estimators=500)
         clf3 = LogisticRegression(solver = "lbfgs",random_state=1)
         # clf4 = GaussianNB()
         #clf5 = xgboost.XGBClassifier()
         clf = VotingClassifier(estimators=[
             #('gbdt',clf1),
             ('rf',RF),
              ('lr',clf2),
             # ('nb',clf4),
             # ('xgboost',clf5),
             ('SVM',clf1)
             ],
             voting='soft')
         clf.fit(X_train,y_train)
         predictions = clf.predict(X_test)
         print("voting_classify")
         print(classification_report(y_test,predictions))
         print("AC",accuracy_score(y_test,predictions))
```

```
==========================================
voting_classify
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      3206
           1       0.98      0.98      0.98      3030

   micro avg       0.98      0.98      0.98      6236
   macro avg       0.98      0.98      0.98      6236
weighted avg       0.98      0.98      0.98      6236

AC 0.9810776138550352
```

Accuracy_score for Voting Classifier 0.9810776138550352

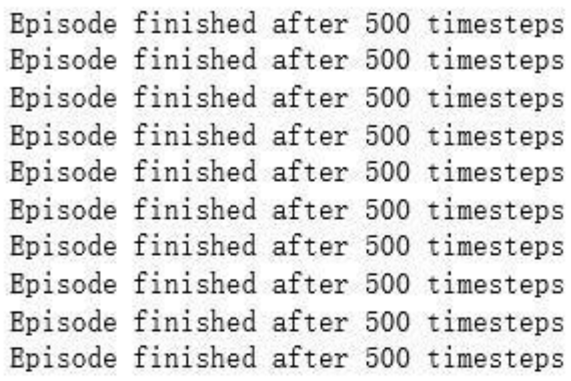
2.6 A simulation Demo using the best-performed classifier:

Using Random Forest to determine the action in CartPole game:



```
def nex_action(observation):
    result = RF.predict([observation])
    return int(result[0])


#simulation Demo
env=gym.make('CartPole-v1')
for episode in range(10):
    observation = env.reset()
    tmp = 0
    for t in range(500):
        #env.render()
        action = nex_action(observation)
        observation, reward, done, info = env.step(action)
        tmp += 1
        if done:
            print("Episode finished after {} timesteps".format(tmp))
            break
    #print("reward: ", tmp)
env.close()
```

Results on RF:



```
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
Episode finished after 500 timesteps
```

We can see from the episodes that all reached best score!

3. Deep Reinforcement Learning Agent using Policy Gradients & DQN

3.1 Brief introduction to Policy Gradients:

Policy gradients is a policy-based reinforcement learning technique.

In policy-based methods, instead of learning a value function(as it is in DQN) that tells us what is the expected sum of rewards given a state and an action, we learn directly the policy function that maps state to action, that is to say, we select actions without using a value function.

$$\pi_\theta(a|s) = P[a|s]$$

In policy search, we have our policy π that has a parameter θ. This π outputs a probability distribution of actions.

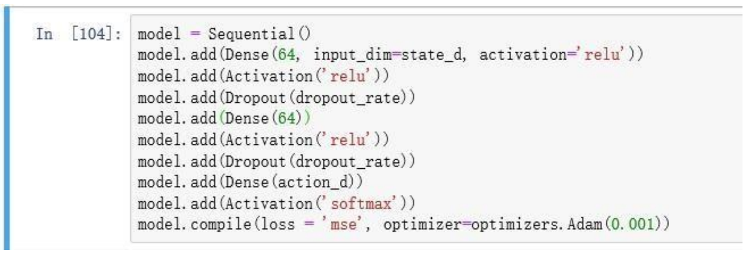
$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

Secondly, we build our Policy Score function in the model. Remember that policy can be seen as an optimization problem. We must find the best parameters (θ) to maximize a score function, J(θ). The main idea here is that J(θ) will tell us how good our π is. Policy gradient ascent will help us to find the best policy parameters to maximize the sample of good actions.

$$\begin{aligned} &Policy:\ \pi_\theta \\ &Objective\ function\ :J(\theta) \\ &Gradient:\ \nabla_\theta J(\theta) \\ &Update:\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \end{aligned}$$

And set the Policy gradient ascent. In normal cases, policy gradient ascent functioned as above.

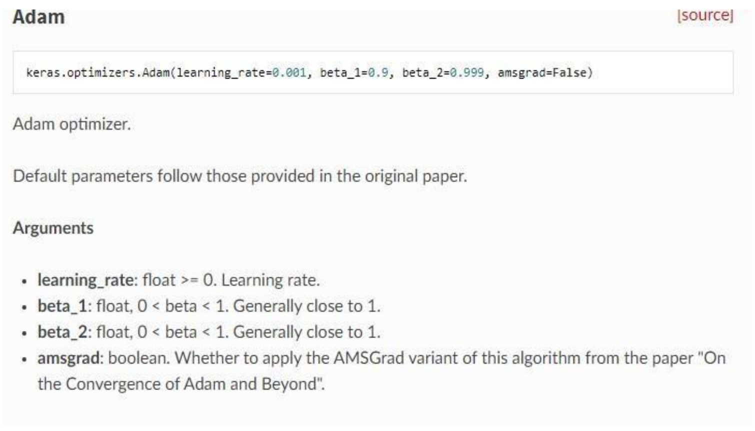
3.2 Set the model using Keras Sequential



```
In  [104]: model = Sequential()
           model.add(Dense(64, input_dim=state_d, activation='relu'))
           model.add(Activation('relu'))
           model.add(Dropout(dropout_rate))
           model.add(Dense(64))
           model.add(Activation('relu'))
           model.add(Dropout(dropout_rate))
           model.add(Dense(action_d))
           model.add(Activation('softmax'))
           model.compile(loss = 'mse', optimizer=optimizers.Adam(0.001))
```

We set two dense levels in the network, using ‘relu’ as activation function.

Note that we use Adam optimizer, as defined in keras reference:


Adam [source]

```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
```

Adam optimizer.

Default parameters follow those provided in the original paper.

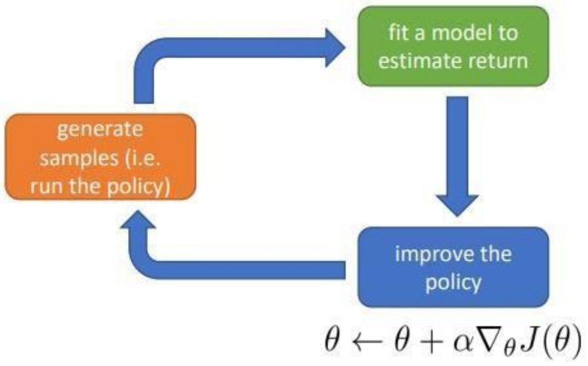
Arguments

- **learning_rate**: float >= 0. Learning rate.
- **beta_1**: float, 0 < beta < 1. Generally close to 1.
- **beta_2**: float, 0 < beta < 1. Generally close to 1.
- **amsgrad**: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

3.3 Set gradient ascent algorithm

Gradient ascent is the inverse of gradient descent. Remember that gradient always points to the steepest change. In gradient descent, we take the direction of the steepest decrease in

the function. In gradient ascent we take the direction of the steepest increase of the function.


fit a model to estimate return
improve the policy
generate samples (i.e. run the policy)
$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

In this particular case of CartPole-v1, we can compare gradient ascent to loss function, using discount_value to calculate for simplicity (as later proved to be of good performance)

```
model.fit(states, probs, sample_weight=discount_value, verbose=0)
```

Note that discount_value is used to calculate gradient ascent after normalization.

```
def discount_rewards(rewards, gamma=0.975):
prior = 0 out = np.zeros_like(rewards)
for i in reversed(range(len(rewards))):
prior = prior * gamma + rewards[i]
out[i] = prior
   return out / np.std(out - np.mean(out))
```

Using a decrease rate as gamma of 0.975.

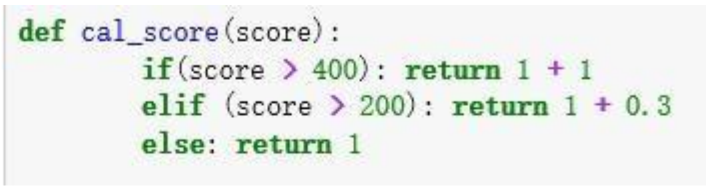
And that we calculate probs in one-hot coding to describe the actions.

eg: P(0.1,0.9) -> probs.add(0,1)

Also note that we use sample_weight in tensorflow.keras to cal loss function as defined below (from reference): Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples

3.4 Train the model

Before training, we can find that in CartPole, reward is forever 1 in every possible timestep,to accelerate the process of training the model, we update the reward in the following rules:



```
def cal_score(score):
        if(score > 400): return 1 + 1
        elif (score > 200): return 1 + 0.3
        else: return 1
```

Because it is pre-known that the winning condition in the CartPole-v0 and CartPole-v1 is 200 timesteps and 500 timesteps in one episode, we set the parameters in the function, 200 and 400 and the bonus rewards based on the rules and the actual performance.

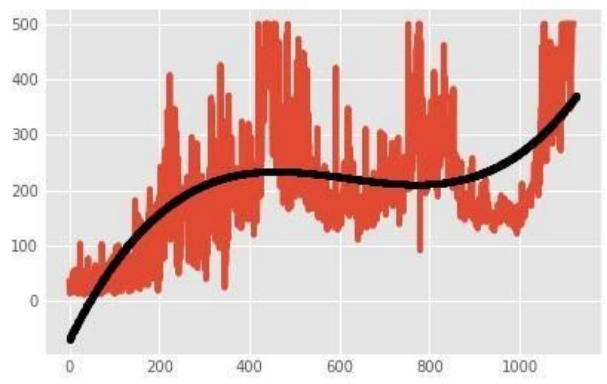
Set the max_episode to be 2000 and start the training :

```
for i in range(max_episodes):
s = env.reset()
score = 0
replay_records = []
while True:
a = act(s)        n
ext_s, r, done, _ = env.step(a)
r = cal_score(score)
replay_records.append((s, a, r))
score += r
s = next_s
if done:
        train(replay_records)
        score_list.append(score)
        break
```

3.5 Performance and test results with different various parameters:

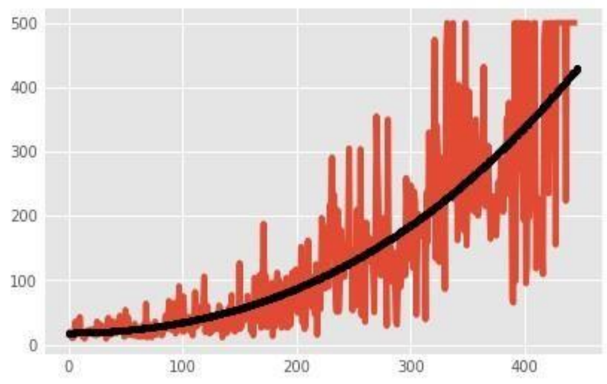
The performance of the model relay much on the parameters mentioned in sections above.

For example, a model using 'tahh' acceleration function without using optimized-reward algorithm and dropout strategy may look like this as performance:



It may encounter overfitting issues and use more episodes to solve the problem.

After adding dropout strategy, setting better parameters and optimize the functions, we can get a better performed model as below(in average)



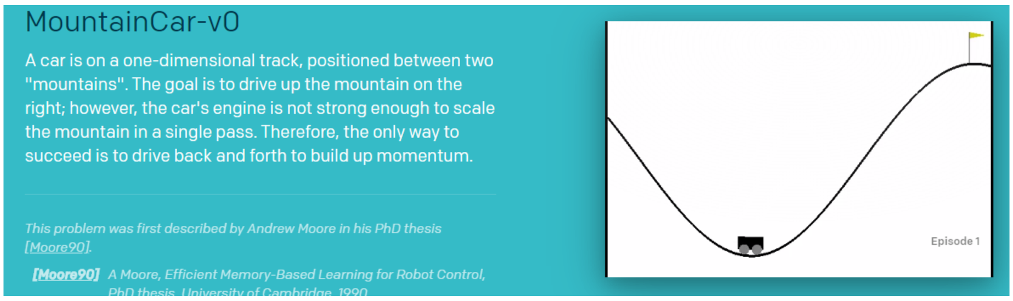
Check the model after reset the environment:

```
CartPole-v1 test starts:    test result using trained policy gradient
model: 500.0   test result using
trained policy gradient model: 500.0   test result using trained policy gradient
model: 500.0   test result using trained policy gradient model: 500.0   test
result using trained policy gradient model: 500.0
```

We can see from the episodes that all reached best score!

3.6 Further discussions using DQN and env 'MountainCar-v0 ':

For better understanding of deep reinforcement learning, we use another environment 'MountainCar-v0' using Deep Q-Learning Network.



MountainCar-v0

A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

This problem was first described by Andrew Moore in his PhD thesis [Moore90].

[Moore90] A Moore, Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, 1990.

In MountainCar-v0 environment, we have 3 actions for agents an 2 observations from the env.

Before DQN, we introduce the Q-learning:

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

```
Q[s][a] = (1 - lr) * Q[s][a] + lr * (reward + factor * max(Q[next_s]))
```

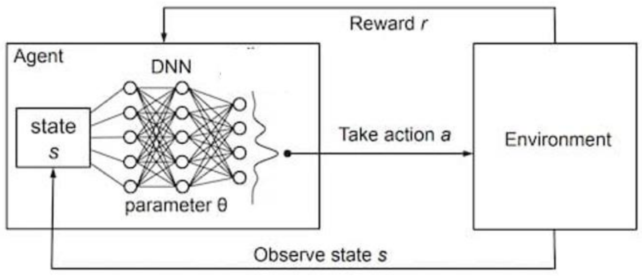
In this specific environment, we set the Q-table to be as the following:

State	Action 0	Action 1	Action 2
[0.2, -0.01]	10	-20	-15
[-0.3, 0.01]	80	15	10
…	…	…	…

As it is in most cases, when state is described in floating numbers, which are contiguous, meaning that there are an infinite number of states, thus updating the q-table value is impossible. As a result, we need to normalize the linear transformation of State.

While in DQN(Deep Q-Learning Network), we use Deep Netural Network (DNN) to replace q-table to realize the calculation of Q value:

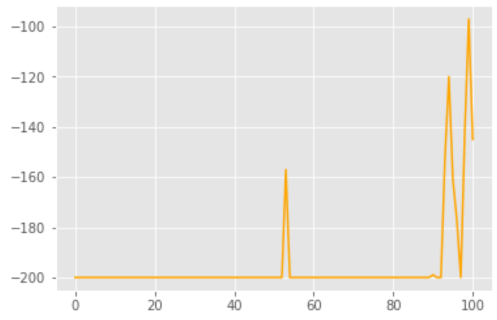
We compare neural network to a Function, and neural network instead of q-table is actually doing Function fitting, which can also be called Value Function Approximation.



Since there is a Universal approximation theorem, which shows that feedforward neural networks can fit functions of arbitrary complexity with arbitrary precision, as long as they have a single hidden layer and a finite number of neural units, we train this particular module with Full Connected Network in TensorFlow 2.0 Keras.

In MountainCar-v0, we set the network with one hidden layer and Alter the reward to +1 when the car reached +0.3.

Results and discussions:



After certain amount of training, the model can finish the task.

- The problem with value-based methods is that they can have a big oscillation while training. This is because the choice of action may change dramatically for an arbitrarily small change in the estimated action values.
- Policy gradients are more effective in high dimensional action space: the problem with Deep Q-learning is that their predictions assign a score on a given state.

References

Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.".

Simonini,T. (2018). *An introduction to Policy Gradients with Cartpole and Doom*. Retrieved from https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/

Wikipedia contributors. (2019, December 5). Deep reinforcement learning. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:44, December 26, 2019, from https://en.wikipedia.org/w/index.php?title=Deep_reinforcement_learning&oldid=929429114

ZhiHua, Z. (2016). *Machine Learning（机器学习）*. Qing hua da xue chu ban she.