Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»

Кафедра «Системы обработки информации и управления»

Отчет по лабораторной работе №2 Reinforcement Learning по дисциплине

«Инструменты бизнес-аналитики»

ИСПОЛНИТЕЛЬ:

Березин И.С.

Группа ИУ5-43М

_____

"__"_____2020 г.

Москва 2020

Цель лабораторной работы: ознакомиться с теорий обучения с подкреплением (reinforcement learning), выполнить 2 практические задачи в данной сфере.

Задачи:

1) Есть N бандитов, для каждого есть число -100< x <100, которое подаётся на вход функции pullBandit

```
def pullBandit(bandit): #Сгенерировать случайное число
  result = np.random.randn(1)
  if result > bandit: #Выигрыш
      return 1
  else: #Проигрыш
      return -1
```

Значение результата используется в качестве награды.

Политика RL: policy gradient. Задача найти номер лучшего бандита

2) Как можно лучше стабилизировать перевернутый маятник (https://github.com/openai/gym/wiki/CartPole-v0)

Теория обучения с подкреплением (reinforcement learning).

Основное отличие обучения с подкреплением (reinforcement learning) от классического машинного обучения заключается в том, что искусственный интеллект обучается в процессе взаимодействия с окружающей средой, а не на исторических данных. Соединив в себе способность нейронных сетей восстанавливать сложные взаимосвязи и самообучаемость агента (системы) в reinforcement learning, машины достигли огромных успехов.

В случае reinforcement learning вместо того, чтобы создавать алгоритм, который обучается на наборе пар «факторы — правильный ответ», в обучении с подкреплением необходимо научить агента взаимодействовать с окружающей средой, самостоятельно генерируя эти пары. Затем на них же он будет обучаться через систему наблюдений (observations), выигрышей (reward) и действий

(actions). Очевидно, что теперь в каждый момент времени нет постоянного правильного ответа, поэтому задача становится немного сложнее.

В нашей постановке задачи есть n игровых автоматов (бандитов), в каждом из которых фиксирована вероятность выигрыша. Тогда цель агента — найти слот-машину с наибольшим ожидаемым выигрышем и всегда выбирать именно ее. Для простоты у нас будет всего четыре игровых автомата, из которых нужно будет выбирать.

По правде говоря, эту задачу можно с натяжкой отнести к reinforcement learning, поскольку задачам из этого класса характерны следующие свойства:

- **Разные действия приводят к разным выигрышам.** К примеру, при поиске сокровищ в лабиринте поворот налево может означать кучу бриллиантов, а поворот направо — яму ядовитых змей.
- **Агент получает выигрыш с задержкой во времени.** Это значит, что, повернув налево в лабиринте, мы не сразу поймем, что это правильный выбор.
- **Выигрыш зависит от текущего состояния системы.** Продолжая пример выше, поворот налево может быть правильным в текущей части лабиринта, но не обязательно в остальных.

В задаче о многоруком бандите нет ни второго, ни третьего условия, что существенно ее упрощает и позволяет нам сконцентрироваться лишь на выявлении оптимального действия из всех возможных вариантов. На языке reinforcement learning это означает найти «правило поведения» (policy). Мы будем использовать метод, называемый policy gradients, при котором нейросеть обновляет свое правило поведения следующим образом: агент совершает действие, получает обратную связь от среды и на ее основе корректирует веса модели через градиентный спуск.

В области обучения с подкреплением есть и другой подход, при котором агенты обучают value functions. Вместо того, чтобы находить оптимальное действие в текущем состоянии, агент учиться предсказывать, насколько выгодно находиться в данном состоянии и совершать данное действие. Оба подхода дают хорошие результаты, однако логика policy gradient более очевидна.

Policy Gradient

Как мы уже выяснили, в нашем случае ожидаемый выигрыш каждого из игровых автоматов не зависит от текущего состояния среды. Получается, что нейросеть будет состоять лишь из набора весов, каждый из которых соответствует одному игровому автомату. Эти веса и будут определять, за какую ручку нужно дернуть, чтобы получить максимальный выигрыш. К примеру, если все веса инициализировать равными 1, то агент будет одинаково оптимистичен по поводу выигрыша во всех игровых автоматах.

Для обновления весов модели мы будем использовать “e-жадную” линию поведения. Это значит, что в большинстве случаев агент будет выбирать действие, которое увеличивает ожидаемый выигрыш до максимума, однако иногда (с вероятностью равной *e*) действие будет случайным. Так будет обеспечен выбор всех возможных вариантов, что позволит нейросети «узнать» больше о каждом из них.

Совершив одно из действий, агент получает обратную связь от системы: 1 или -1 в зависимости от того, выиграл ли он. Это значение затем используется для расчета функции потерь:

$$\text{Loss} = -\log(\pi) * A$$

A (advantage) — важный элемент всех алгоритмов обучения с подкреплением. Он показывает, насколько совершенное действие лучше, чем некий baseline. В дальнейшем будем использовать более сложный baseline, а пока примем его равным 0, то есть *A* будет просто равен награде за каждое действие (1 или -1). *n* — это правило поведения, вес нейросети, соответствующий ручке слот-машины, которую мы выбрали на текущем шаге.

Интуитивно понятно, что функция потерь должна принимать такие значения, чтобы веса действий, которые привели к выигрышу увеличивались, а те, которые привели к проигрышу, уменьшались. В результате веса будут обновляться, а агент будет все чаще и чаще выбирать игровой автомат с наибольшей фиксированной вероятностью выигрыша, пока, наконец, он не будет выбирать его всегда.

Решение первой задачи:

Сначала создадим наших бандитов (игровых автоматов). В нашем примере их будет 5. Функция *pullBandit* генерирует случайное число из стандартного нормального распределения, а затем сравнивает его со значением бандита и возвращает результат игры. Чем дальше по списку находится бандит, тем больше вероятность, что агент выиграет, выбрав именно его. Таким образом, мы хотим, чтобы наш агент научился **всегда** выбирать последнего бандита (рисунок 1).

In [19]:
```
import tensorflow as tf
import numpy as np

bandits = [] # Список наших бандитов
k = 5 # кол-во бандитов
for i in range(0, k):
    i = np.random.randn(1)
    bandits.append(i)
print (bandits)
num_bandits = len(bandits)
def pullBandit(bandit):
    result = np.random.randn(1) #Сгенерировать случайное число
    if result > bandit:
        return 1 #Выигрыш
    else:
        return -1 #Проигрыш
```

```
[array([0.01072654]), array([-0.383953]), array([0.89364761]), array([-1.20883045]), array([0.25878564])]
```

Рисунок 1 – создание “бандитов”

Блок кода ниже создает простого агента, который состоит из набора значений для бандитов. Каждое значение соответствует выигрышу или проигрышу в зависимости от выбора того или иного бандита. Чтобы обновлять веса агента мы используем policy gradient, то есть выбираем действия, которое минимизирует функцию потерь (рисунок 2).

In [20]:
```
tf.reset_default_graph()
# Эти 2 строчки создают feed-forward часть нейросети. Здесь и происходит выбор действия.
weights = tf.Variable(tf.ones([num_bandits]))
chosen_action = tf.argmax(weights,0)

# Следующие 6 строчек устанавливают процедуру обучения. Нейросеть принимает на вход действие и
# его результат, чтобы оценить функцию потерь и обновить веса сети.
reward_holder = tf.placeholder(shape=[1],dtype=tf.float32)
action_holder = tf.placeholder(shape=[1],dtype=tf.int32)
responsible_weight = tf.slice(weights,action_holder,[1])
loss = -(tf.log(responsible_weight)*reward_holder)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
update = optimizer.minimize(loss)
```

Рисунок 2 – создание агента

Будем обучать агента, путем выбора определенных действий и получения выигрышей или проигрышей. Используя полученные значения, мы будем знать,

как именно обновить веса модели, чтобы чаще выбирать бандитов с большим ожидаемым выигрышем (рисунок 3).

In [21]:
```
total_episodes = 1000 #Количество итераций обучения
total_reward = np.zeros(num_bandits)
e = 0.1 #Вероятность случайного выбора
init = tf.global_variables_initializer()
```

In [22]:
```
with tf.Session() as sess:
    sess.run(init)
    i = 0
    while i < total_episodes:

        if np.random.rand(1) < e:
            action = np.random.randint(num_bandits)
        else:
            action = sess.run(chosen_action)

        reward = pullBandit(bandits[action])

        #Обновляем веса
        _,resp,ww = sess.run([update,responsible_weight,weights],
                     feed_dict={reward_holder:[reward],action_holder:[action]})

        #Обновляем общий выигрыш каждого бандита
        total_reward[action] += reward
        if i % 50 == 0:
            print("Общий выигрыш сейчас равен " + str(num_bandits) +
            " бандитов: " + str(total_reward))
        i+=1
print("Агент думает, что бандит №" + str(np.argmax(ww)+1) + " лучший")
```

Рисунок 3 – процесс обучения агента

Результат работы программы представлен на рисунке 4:

```
Общий выигрыш сейчас равен 5 бандитов: [-1.  0.  0.  0.  0.]
Общий выигрыш сейчас равен 5 бандитов: [-1. -2.  0. 31.  1.]
Общий выигрыш сейчас равен 5 бандитов: [-1. -1.  1. 67.  1.]
Общий выигрыш сейчас равен 5 бандитов: [  0.  -1.   1. 100.   1.]
Общий выигрыш сейчас равен 5 бандитов: [  1.  -1.   0. 138.   1.]
Общий выигрыш сейчас равен 5 бандитов: [  1.  -2.  -1. 173.   0.]
Общий выигрыш сейчас равен 5 бандитов: [  1.  -3.  -2. 213.   0.]
Общий выигрыш сейчас равен 5 бандитов: [  0.  -3.  -2. 249.  -1.]
Общий выигрыш сейчас равен 5 бандитов: [  0.  -2.  -3. 284.   0.]
Общий выигрыш сейчас равен 5 бандитов: [ -1.   0.  -3. 324.   1.]
Общий выигрыш сейчас равен 5 бандитов: [  0.   1.  -3. 354.   1.]
Общий выигрыш сейчас равен 5 бандитов: [  0.   1.  -2. 382.   0.]
Общий выигрыш сейчас равен 5 бандитов: [  2.   1.  -4. 419.  -1.]
Общий выигрыш сейчас равен 5 бандитов: [  2.   2.  -7. 459.  -1.]
Общий выигрыш сейчас равен 5 бандитов: [  1.   2.  -7. 494.  -3.]
Общий выигрыш сейчас равен 5 бандитов: [  1.   2.  -8. 523.  -3.]
Общий выигрыш сейчас равен 5 бандитов: [  0.   2. -10. 551.  -4.]
Общий выигрыш сейчас равен 5 бандитов: [  0.   4. -10. 589.  -4.]
Общий выигрыш сейчас равен 5 бандитов: [ -1.   3. -10. 627.  -4.]
Общий выигрыш сейчас равен 5 бандитов: [  0.   4. -11. 658.  -4.]
Агент думает, что бандит №4 лучший
```

Рисунок 4 – результат выполнения программы

Задача стабилизации перевернутого маятника

Воспользуемся OpenAI Gym — платформой для разработки и тренировки AI ботов с помощью игр и алгоритмических испытаний и возьмем классическую задачу оттуда: задача стабилизации перевернутого маятника или Cart-Pole. В нашем случае суть задачи заключается в том, чтобы как можно дольше удерживать стержень в вертикальном положении, двигая тележку по горизонтали (рисунок 5):

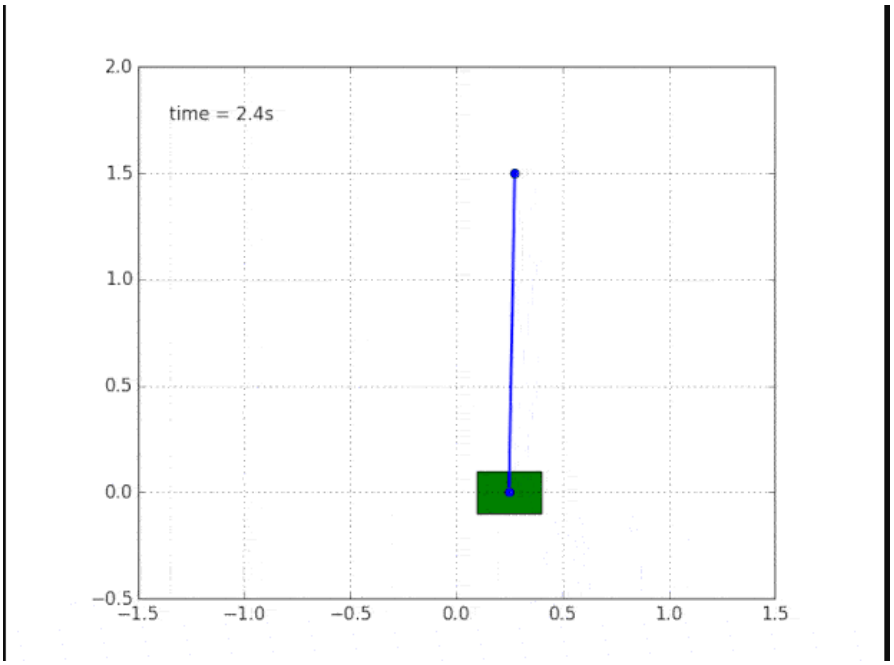

Рисунок 5 – перевернутый маятник

В отличии от задачи ранее (о бандитах), в данной системе есть:

- **Наблюдения.** Агент должен знать, где стержень находится сейчас и под каким углом. Это наблюдение нейросеть будет использовать для оценки вероятности того или иного действия.
- **Отсроченный выигрыш.** Необходимо двигать тележку таким образом, чтобы это было выгодно как на данный момент, так и в будущем. Для этого будем сопоставлять пару «наблюдение — действие» со скорректированным значением выигрыша. Корректировка осуществляется функцией, которая взвешивает действия по времени.

Чтобы учитывать задержку выигрыша во времени, нам нужно использовать policy gradient метод с некоторыми поправками. Во-первых, теперь необходимо обновлять агента, который имеет более одного наблюдения в единицу времени. Для

этого все наблюдения мы будем собирать в буфер, а затем использовать их одновременно, чтобы обновить веса модели. Этот набор наблюдений за единицу времени затем сопоставляется с *дисконтированным* выигрышем.

Таким образом, каждое действие агента будет совершено с учетом не только мгновенного выигрыша, но и всех последующих. Также теперь мы будем использовать скорректированный выигрыш в качестве оценки элемента *A (advantage)* в функции потерь.

Импортируем библиотеки и загрузим среду задачи Cart-Pole (рисунок 6):

```
In [15]: import tensorflow as tf
         import tensorflow.contrib.slim as slim
         import numpy as np
         import gym
         import matplotlib.pyplot as plt

In [16]: try:
             xrange = xrange
         except:
             xrange = range

         env = gym.make('CartPole-v0') #загружаем среду задачи
```

Рисунок 6 – импорт библиотеки и загрузка среды задачи Cart-Pole

Сначала создадим функцию, которая будет дисконтировать все последующие выигрыши на текущий момент (рисунок 7):

```
In [17]: gamma = 0.99 # коэффициент дисконтирования

In [18]: def discount_rewards(r):
             """ принимая на вход вектор выигрышей,
             вернуть вектор дисконтированных выигрышей"""
             discounted_r = np.zeros_like(r)
             running_add = 0
             for t in reversed(xrange(0, r.size)):
                 running_add = running_add * gamma + r[t]
                 discounted_r[t] = running_add
             return discounted_r
```

Рисунок 7 – функция, дисконтирующая все последующие выигрыши на текущий момент

Теперь создадим нашего агента (рисунок 8).

In [19]:
```
class agent():
    def __init__(self, lr, s_size,a_size,h_size):
        #Ниже инициализирована feed-forward часть нейросети.
        #Агент оценивает состояние среды и совершает действие
        self.state_in= tf.placeholder(shape=[None,s_size],dtype=tf.float32)
        hidden = slim.fully_connected(self.state_in,h_size,
                     biases_initializer=None,activation_fn=tf.nn.relu)
        self.output = slim.fully_connected(hidden,a_size,
                     activation_fn=tf.nn.softmax,biases_initializer=None)
        self.chosen_action = tf.argmax(self.output,1) # выбор действия

        #Следующие 6 строк устанавливают процедуру обучения.
        #Нейросеть принимает на вход выбранное действие
        # и соответствующий выигрыш,
        #чтобы оценить функцию потерь и обновить веса модели.
        self.reward_holder = tf.placeholder(shape=[None],dtype=tf.float32)
        self.action_holder = tf.placeholder(shape=[None],dtype=tf.int32)

        self.indexes = tf.range(0,
        tf.shape(self.output)[0])*tf.shape(self.output)[1] + self.action_holder

        self.responsible_outputs = tf.gather(tf.reshape(self.output, [-1]),
        self.indexes)
        #функция потерь
        self.loss = -tf.reduce_mean(tf.log(self.responsible_outputs)*
        self.reward_holder)

        tvars = tf.trainable_variables()
        self.gradient_holders = []
        for idx,var in enumerate(tvars):
            placeholder = tf.placeholder(tf.float32,name=str(idx)+'_holder')
            self.gradient_holders.append(placeholder)

        self.gradients = tf.gradients(self.loss,tvars)

        optimizer = tf.train.AdamOptimizer(learning_rate=lr)
        self.update_batch = optimizer.apply_gradients(zip(self.gradient_holders,
        tvars))
```

Рисунок 8 – создание агента

На рисунке 9 представлен процесс обучения агента:

In [20]:
```
tf.reset_default_graph() #Очищаем граф tensorflow

myAgent = agent(lr=1e-2,s_size=4,a_size=2,h_size=8) #Инициализируем агента

total_episodes = 10000 #Количество итераций обучения
max_ep = 999
update_frequency = 5

init = tf.global_variables_initializer()

#Запуск графа tensorflow
with tf.Session() as sess:
    sess.run(init)
    i = 0
    total_reward = []
    total_lenght = []

    gradBuffer = sess.run(tf.trainable_variables())
    for ix,grad in enumerate(gradBuffer):
        gradBuffer[ix] = grad * 0

    while i < total_episodes:
        s = env.reset()
        running_reward = 0
        ep_history = []
        for j in range(max_ep):
            #Выбрать действие на основе вероятностей, оцененных нейросетью
            a_dist = sess.run(myAgent.output,feed_dict={myAgent.state_in:[s]})
            a = np.random.choice(a_dist[0],p=a_dist[0])
            a = np.argmax(a_dist == a)

            s1,r,d,_ = env.step(a) #Получить награду за совершенное действие
            ep_history.append([s,a,r,s1])
            s = s1
            running_reward += r
            if d == True:
                #Обновить нейросеть
                ep_history = np.array(ep_history)
                ep_history[:,2] = discount_rewards(ep_history[:,2])
                feed_dict = {myAgent.reward_holder:ep_history[:,2],
                             myAgent.action_holder:ep_history[:,1],
                             myAgent.state_in:np.vstack(ep_history[:,0])}
                grads = sess.run(myAgent.gradients, feed_dict=feed_dict)
                for idx,grad in enumerate(grads):
                    gradBuffer[idx] += grad

                if i % update_frequency == 0 and i != 0:
                    feed_dict = dictionary = dict(zip(myAgent.gradient_holders,
                                                      gradBuffer))
                    _ = sess.run(myAgent.update_batch, feed_dict=feed_dict)
                    for ix,grad in enumerate(gradBuffer):
                        gradBuffer[ix] = grad * 0

                total_reward.append(running_reward)
                total_lenght.append(j)
                break


            #Обновить общий выигрыш
        if i % 100 == 0:
            print(np.mean(total_reward[-100:]))
        i += 1
```

Рисунок 9 – процесс обучения агента

Результат работы программы представлен на рисунке 10:

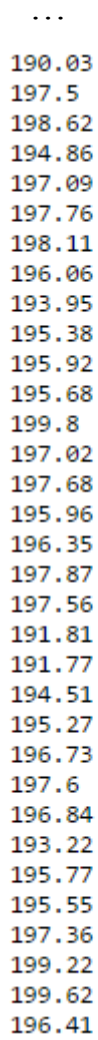

Рисунок 10 – результаты выполнения программы

Заключение

Таким образом, в данной лабораторной работе было рассмотрено обучение с подкреплением (reinforcement learning), отличия данного метода обучения нейронных сетей от классического. Выполнены две поставленные задачи на практике с использованием стандартных библиотек по машинному обучению (tenserflow и др.), а также платформы для разработки и тренировки AI ботов с помощью игр и алгоритмических испытаний OpenAI Gym.