МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Школа информационных технологий и робототехники

Направление подготовки 09.04.01 «Информатика и вычислительная техника»

Выпускающее подразделение: ОИТ (Отделение информационных технологий)

Лабораторная работа

«Использование методов машинного обучения при создании игры»

По дисциплине **«Интеллектуальные системы»**

Выполнил:

Студент группы 8ВМ13 Никитась С.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Работу принял:

Ассистент (ОИТ, ИШИТР) Григорьев Д.С. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Томск, 2021

**Реферат**

Отчёт по лабораторной работе 8 стр., 2 рисунка, 1 приложение.

Ключевые слова: методы машинного самообучения (ML), генетический алгоритм, игра, Python.

Объектом исследования являются: методы машинного самообучения в игре.

Предметом исследования является выработанный в процессе машинного самообучения генетический алгоритм.

Цель работы: рассмотреть возможности применения методов машинного обучения в игре при помощи соответствующего кода на языке Python, сделать выводы относительно способности алгоритма к обучению.

Используемая платформа: PyCharm

**Введение**

В ходе работы, был применён алгоритм генетический машинного обучения для управления простейшим вариантом игры. Игра включает в себя препятствия, сгенерированные случайным образом, которые необходимо преодолевать.

Задаются изменяемые и неизменяемые параметры, определяющие ход игры.

* Изменяемые параметры:

«Score» (число полученных очков), «Generation» (число завершённых попыток), «Process Time» (время, пройденное алгоритмом в течение каждой попытки, выраженное в секундах).

* Неизменяемые параметры:

«Population» (значение, выделенное для поля популяции или, проще, множество проверочных тестов, которые потом в качестве параметров генетического алгоритма будут формировать множество успешных попыток, равно 100 популяциям), «Average Fitness» и «Highest Fitness» (среднее и наибольшее число возможных попыток, показатели успешности которых варьируется с учётом совокупности прочих отобранных показателей, располагается в пределах 58), «Mutation Rate» (частота мутаций в генетическом алгоритме, равна 0.1).

**Результаты работы**

В данном случае, результатом работы является информация о возможностях применения генетического алгоритма в игровом процессе. Самообучающаяся система пытается научиться способности преодолевать препятствия, которые заранее неизвестны. Цикл обучения достигал 10000, наблюдение за работой алгоритма происходило в течение нескольких дней.

На рисунке 1 мы видим часть программного кода, в котором задана функция, позволяющая игре функционировать в обычном режиме. Также на данном рисунке можно отследить изменяемые и неизменяемые показатели.

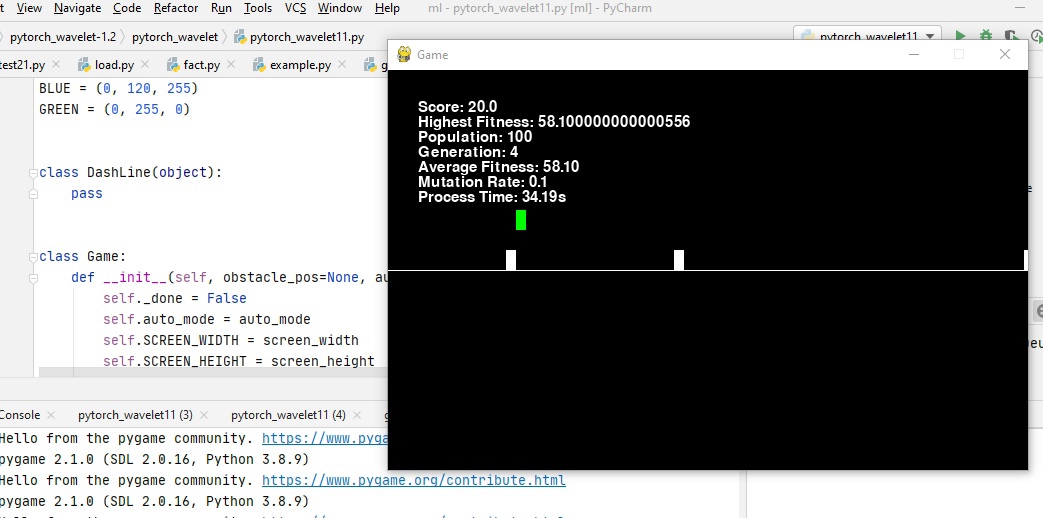


Рисунок 1

На рисунке 2 показано изменение, по сравнению в предыдущим рисунком, числа попыток «generation». Стоит отметить, что в процессе апробации алгоритма число попыток достигало не более 10000 (следовательно, экспериментальным методом было выявлено максимально возможное число попыток). Была проведена попытка выявления взаимного влияния между показателями «generation» и «process time», однако, между показателями «process time» и «score» чёткой взаимосвязи не было установлено.



Рисунок 2

**Вывод**

В результате запуска кода, наблюдается постепенный процесс обучения программы. Для получения более удовлетворительных результатов, требуется дополнительное изучение различных параметров настройки данного алгоритма.

**Приложение 1**

Используемый код

DTYPE = torch.complex64  
EPS = torch.finfo(DTYPE).eps \* 10  
  
  
def \_roll(x, N):  
 return torch.cat((x[-N:, ...], x[:-N, ...]), dim=0)  
  
  
def \_haar2(x, dtype=torch.complex64, device=**'cpu'**):  
 a0 = (1 / 2) \* (x[0::2, 0::2] + x[0::2, 1::2] + x[1::2, 0::2] + x[1::2, 1::2])  
 a1 = (1 / 2) \* (x[0::2, 0::2] + x[0::2, 1::2] - x[1::2, 0::2] - x[1::2, 1::2])  
 a2 = (1 / 2) \* (x[0::2, 0::2] - x[0::2, 1::2] + x[1::2, 0::2] - x[1::2, 1::2])  
 a3 = (1 / 2) \* (x[0::2, 0::2] - x[0::2, 1::2] - x[1::2, 0::2] + x[1::2, 1::2])  
 return [a0, a1, a2, a3]  
  
  
def \_ihaar2(a, dtype=torch.complex64, device=**'cpu'**):  
 Np = [a[0].shape[0] \* 2, a[0].shape[1] \* 2]  
 x = torch.zeros(Np, dtype=dtype, device=device)  
 x[0::2, 0::2] = (1 / 2) \* (a[0] + a[1] + a[2] + a[3])  
 x[0::2, 1::2] = (1 / 2) \* (a[0] + a[1] - a[2] - a[3])  
 x[1::2, 0::2] = (1 / 2) \* (a[0] - a[1] + a[2] - a[3])  
 x[1::2, 1::2] = (1 / 2) \* (a[0] - a[1] - a[2] + a[3])  
 return x  
  
  
def \_multi\_level\_haar2(x, Nlayers, dtype=torch.complex64, device=**'cpu'**):  
 if Nlayers == 1:  
 return \_haar2(x, dtype=dtype, device=device)  
 else:  
 a = \_haar2(x, dtype=dtype, device=device)  
 a\_low = \_multi\_level\_haar2(a[0], Nlayers - 1, dtype, device)  
 return [a\_low] + a[1:]  
  
  
def \_multi\_level\_ihaar2(a, Nlayers, dtype=torch.complex64, device=**'cpu'**):  
 if Nlayers == 1:  
 return \_ihaar2(a, dtype=dtype, device=device)  
 else:  
 a\_low = \_multi\_level\_ihaar2(a[0], Nlayers - 1, dtype, device)  
 x = \_ihaar2([a\_low] + a[1:], dtype=dtype, device=device)  
 return x  
  
  
def \_wavelet\_block(a):  
 a\_top = torch.cat((a[0], a[1]), 0)  
 a\_bottom = torch.cat((a[2], a[3]), 0)  
 return torch.cat((a\_top, a\_bottom), 1)  
  
  
def \_visualize(a):  
 if len(a[0]) != 4:  
 return \_wavelet\_block(a)  
 else:  
 a\_lower = \_visualize(a[0])  
 return \_wavelet\_block([a\_lower] + a[1:])  
  
  
def visualize(x, Nlayers, dtype=torch.complex64, device=**'cpu'**):  
 with torch.no\_grad():  
 a = \_multi\_level\_haar2(x, Nlayers, dtype=dtype, device=device)  
 return \_visualize(a)  
  
  
class Haar2():  
 def \_\_init\_\_(self, dtype=torch.float32, device=**'cpu'**):  
 self.dtype = dtype  
  
 self.\_check\_inverse(device)  
  
 def forward(self, x, shiftx=False, shifty=False, dtype=torch.complex64, device=**'cpu'**):  
 if shiftx: x = \_roll(x, 1)  
 if shifty: x = \_roll(x.permute(1, 0), 1)  
 return \_haar2(x, dtype, device)  
  
 def inverse(self, a, shiftx=False, shifty=False, dtype=torch.complex64, device=**'cpu'**):  
 x = \_ihaar2(a, dtype, device)  
 if shifty: x = \_roll(x, -1).permute(1, 0)  
 if shiftx: x = \_roll(x, -1)  
 return x  
  
 def \_check\_inverse(self, device=**'cpu'**):  
 with torch.no\_grad():  
 x = torch.rand((256, 256), dtype=self.dtype, device=device)  
 WHWx = self.inverse(self.forward(x, device=device), device=device)  
 out = torch.sum(torch.abs(x - WHWx))  
 assert out < EPS \* x.shape[0] \* x.shape[1], **'Inverse test failed!'**class HaarN():  
 def \_\_init\_\_(self, Nlayers, dtype=torch.complex64, device=**'cpu'**):  
 self.Nlayer = Nlayer  
  
 self.dtype = dtype  
  
 self.\_check\_inverse(device)  
  
 def forward(self, x, shiftx=False, shifty=False, dtype=torch.complex64, device=**'cpu'**):  
 if shiftx: x = \_roll(x, 1)  
 if shifty: x = \_roll(x.permute(1, 0), 1)  
 return \_multi\_level\_haar2(x, self.Nlayers, dtype, device)  
  
 def reverse(self, a, shiftx=False, shifty=False, dtype=torch.complex64, device=**'cpu'**):  
 x = \_ihaar2(a, dtype, device)  
 if shifty: x = \_roll(x, -1).permute(1, 0)  
 if shiftx: x = \_roll(x, -1)  
 return x  
  
 def \_check\_inverse(self, device=**'cpu'**):  
 with torch.no\_grad():  
 x = torch.rand((256, 256), dtype=self.dtype, device=device)  
 WHWx = self.inverse(self.forward(x, device=device), device=device)  
 out = torch.sum(torch.abs(x - WHWx))  
 assert out < EPS \* x.shape[0] \* x.shape[1], **'Inverse test failed!'**