بسم الله الرحمن الرحیم

گزارش کار فاز دوم قسمت دوم

استاد مربوطه: استاد حسین کارشناس

یونس ایوبی راد 4013613011

پویا اسفندانی 4013613005

محیط های ناشناخته

# کلاس شبکه عصبی

در این کلاس در گام اول یک کلاس شبکه عصبی تعریف شده.

class QNetwork(nn.Module):  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  
 super(QNetwork, self).\_\_init\_\_()  
 self.FC = nn.Sequential(  
 nn.Linear(input\_size, hidden\_size),  
 nn.ReLU(),  
 nn.Linear(hidden\_size, hidden\_size),  
 nn.ReLU(),  
 nn.Linear(hidden\_size, output\_size)  
 )  
 self.init\_weights()  
  
 def init\_weights(self):  
 for layer in self.FC:  
 if isinstance(layer, nn.Linear):  
 nn.init.kaiming\_uniform\_(layer.weight, nonlinearity='relu')  
  
 def forward(self, x):  
 return self.FC(x)

این کد یک شبکه عصبی (Fully Connected Feedforward Neural Network) است که برای پیش‌بینی مقدار Q در تقویت یادگیری طراحی شده است. شبکه سه لایه دارد: ورودی، مخفی، و خروجی. لایه‌ها شامل توابع ریاضی هستند که داده‌ها را پردازش می‌کنند تا الگوهای پیچیده یاد گرفته شوند. از ReLU برای غیرخطی کردن مدل و از مقداردهی اولیه مناسب برای وزن‌ها استفاده شده تا یادگیری بهتر انجام شود. این مدل وضعیت محیط را می‌گیرد و برای هر اقدام ممکن یک مقدار Q پیش‌بینی می‌کند، که نشان می‌دهد هر اقدام چقدر پاداش دارد. هدف آن انتخاب بهترین اقدام در هر وضعیت است و کاربرد آن بیشتر در بازی‌ها و تصمیم‌گیری‌های هوشمند است.

در ابتدای کار، شبکه هیچ چیز نمی‌داند، و وزن‌ها (ارتباط بین نودها) به‌طور تصادفی تنظیم می‌شوند. روش "کايمينگ یونیفرم" که در اینجا استفاده شده، یک روش خاص برای مقداردهی وزن‌ها است تا مطمئن شویم یادگیری شبکه سریع‌تر و پایدارتر خواهد بود.

# کلاس حافظه

. class ReplayBuffer:  
 def \_\_init\_\_(self, length):  
 self.memory = deque(maxlen=length)  
  
 def add(self, member):  
 self.memory.append((  
 torch.tensor(member[0], dtype=torch.float32),  
 torch.tensor(member[1], dtype=torch.long),  
 torch.tensor(member[2], dtype=torch.long),  
 torch.tensor(member[3], dtype=torch.float32),  
 torch.tensor(member[4], dtype=torch.bool)  
 ))  
  
 def sample(self, batch\_size):  
 batch = random.sample(self.memory, batch\_size)  
 states, actions, rewards, next\_states, dones = zip(\*batch)  
 return (  
 torch.stack(states).to(device),  
 torch.stack(actions).to(device),  
 torch.stack(rewards).to(device),  
 torch.stack(next\_states).to(device),  
 torch.stack(dones).to(device)  
 )  
  
 def \_\_len\_\_(self):  
 return len(self.memory)

کلاس **ReplayBuffer** یک حافظه حلقه‌ای برای ذخیره و نمونه‌گیری تجربیات در تقویت یادگیری است. تجربیات شامل وضعیت فعلی، اقدام، پاداش، وضعیت بعدی، و علامت پایان اپیزود هستند. این داده‌ها به‌صورت تانسور ذخیره می‌شوند و هنگام آموزش مدل، به‌صورت تصادفی نمونه‌گیری می‌شوند. این فرآیند از همبستگی داده‌های متوالی جلوگیری کرده و یادگیری را پایدارتر می‌کند. همچنین، وقتی حافظه پر شود، تجربیات قدیمی حذف می‌شوند تا جا برای داده‌های جدید باز شود. هدف اصلی، بهبود آموزش مدل از تجربیات گذشته است.

# کلاس Agent

کلاس Agent نماینده‌ی عامل هوشمندی است که در محیط یادگیری تقویتی عمل می‌کند. عامل با استفاده از شبکه‌های عصبی تصمیم می‌گیرد و خودش را آموزش می‌دهد تا پاداش بیشتری در محیط کسب کند

سازنده:

عامل شامل دو شبکه عصبی است:

PolicyNetwork : برای انتخاب اقدامات.

Target Network: برای پیش‌بینی پاداش‌های آینده، که هر از گاهی به‌روزرسانی می‌شود.

از الگوریتم Adam برای بهینه‌سازی وزن‌های شبکه و از huber loss برای محاسبه خطا (loss) استفاده می‌کند.

حافظه‌ای از نوع ReplayBuffer برای ذخیره تجربیات دارد.

پارامترهای یادگیری تقویتی:

Epsilon: احتمال انتخاب اقدام تصادفی (برای کاوش).

Gamma: ضریب تخفیف برای ارزش پاداش‌های آینده.

epsilon\_decay: نرخ کاهش مقدار epsilon با گذر زمان.

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  
 self.n\_games = 0  
 self.epsilon = epsilon  
 self.epsilon\_min = epsilon\_min  
 self.epsilon\_decay = epsilon\_decay  
 self.gamma = gamma  
 self.memory = ReplayBuffer(memory\_size)  
 self.policy = QNetwork(input\_size, hidden\_size, output\_size).to(device)  
 self.target = QNetwork(input\_size, hidden\_size, output\_size).to(device)  
 self.target.load\_state\_dict(self.policy.state\_dict())  
 self.optimizer = optim.Adam(self.policy.parameters(), lr=learning\_rate)  
 self.loss\_fn = nn.SmoothL1Loss(beta=1.0)  
 self.loss\_history = []

تابع انتخاب:

تابع select\_action اقدام عامل را با استفاده از استراتژی ε-greedy انتخاب می‌کند:

اگر مقدار تصادفی از epsilon کمتر باشد، اقدام تصادفی برای کاوش (Exploration) انتخاب می‌شود.

در غیر این صورت، اقدام بهینه بر اساس شبکه عصبی Policy پیشبینی میشود.

وضعیت ورودی (state) به Tensor تبدیل و محاسبه به GPU ارسال می‌شود.

شبکه Q-Value هر اقدام را محاسبه کرده و اقدام با بیشترین مقدار انتخاب می‌شود.

تابع Train :

این تابع عامل را با استفاده از داده‌های ذخیره‌شده در حافظه آموزش می‌دهد. مراحل به شرح زیر است:

**بررسی شرط آموزش:**  
اگر داده‌های حافظه کمتر از اندازه مینی‌باتچ باشد، تابع متوقف می‌شود.

**نمونه‌گیری از حافظه:**  
تجربیات ذخیره‌شده (شامل وضعیت‌ها، اقدامات، پاداش‌ها، وضعیت‌های بعدی، و وضعیت پایان) به‌صورت تصادفی نمونه‌گیری می‌شوند.

**محاسبه مقادیر Q فعلی:**  
شبکه **Policy** مقادیر Q را برای وضعیت‌های فعلی پیش‌بینی می‌کند. سپس با استفاده از اقدامات انجام‌شده، مقادیر مرتبط انتخاب می‌شوند.

**محاسبه مقادیر Q هدف (Target Q):**

با استفاده از **Target Network**، بیشترین مقدار Q برای وضعیت‌های بعدی پیش‌بینی می‌شود.

**محاسبه و به‌روزرسانی خطا (Loss):**  
خطای بین مقادیر Q فعلی و مقادیر Q هدف با استفاده از تابع از دست دادن (**loss function**) محاسبه می‌شود. سپس، وزن‌های شبکه **Policy** با استفاده از گرادیان نزولی به‌روزرسانی می‌شوند.

**به‌روزرسانی ε (Epsilon):**  
مقدار **ε** کاهش می‌یابد تا عامل به‌تدریج بیشتر بهره‌برداری کرده و کمتر کاوش کند.

# تابع های کلاس مین:

def choose\_step(env, agent, state,previous\_pigs):  
 pushState = np.append(np.array(state),np.array(previous\_pigs))  
 action = agent.select\_action(pushState)  
 next\_state, reward,pigs, done = env.step(action)  
 temp = reward  
 if state == next\_state:  
 temp = -10  
 if done:  
 agent.n\_games = agent.n\_games + 1  
 if agent.n\_games % 10 == 0:  
 agent.target.load\_state\_dict(agent.policy.state\_dict())  
 agent.epsilon = agent.epsilon \* agent.epsilon\_decay  
 pushNextState = np.append(np.array(next\_state),np.array(pigs))  
 agent.memory.add([pushState, action, temp, pushNextState, done])  
 return pigs, next\_state, action, reward, done

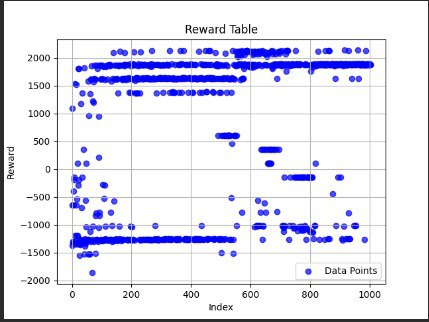
این تابع 5 ورودی دارد محیط، عامل‌هوشمند، استیت کنونی، لیست خوک های سابق که در قسمت پوش استیت لیست خوک ها را با استیت کنونی append می‌کند و پوش استیت جدیدی به وجودی میاورد سپس از تابع select action از ایجنت را فرا میخواند و استیت را می‌خواند، سپس کار را در محیط انجام می‌دهد و استیت جدید و خوک ها و پاداش و نتیجه بازی را میگیرید و بعد از آن جایزه را در تمپ میریزم و اگراستیت کنونی برابر با استیت قبلی بود تمپ را به منفی ده تبدیل میکنم(برای جلوگیری از رفتن مکرر به دیوار) و اگر بازی اتمام یافته بود تعداد بازی های ایجنت یک عدد اضافه میشود و اپسیلون اپدیت میشود و اگر مود تعداد بازی ها به ده برابر با صفر بود درخت پالیسی را در درخت تارگت کپی میکند و سپس پوش استیت بعدی و خوک های آینده را در یک آرایه نامپای قرار میدهد و آن ها را به حافظه ایجنت اضافه میکند و خوک ها، استیت بعدی، کار انجام شده، جایزه و نتیجه بازی را بر میگرداند.

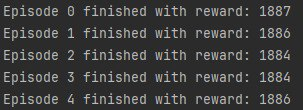
# تابع مین:

if \_\_name\_\_ == "\_\_main\_\_":  
 device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
  
 env = UnknownAngryBirds()  
 screen, clock = PygameInit.initialization()  
 episode\_reward = []  
 agent = Agent(10,128,4)  
 for episode in range(1000):  
 screen, clock = PygameInit.initialization()  
 state = env.reset()  
 visited\_node = []  
 running = True  
 total\_reward = 0  
 pigs = [1,1,1,1,1,1,1,1]  
 while running:  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 pygame.quit()  
 env.render(screen)  
 pigs, next\_state, action, reward, done = choose\_step(env, agent, state,pigs)  
 visited\_node = list  
 state = next\_state  
 total\_reward += reward  
 if len(agent.memory) > 256:  
 agent.train()  
  
 if done:  
 print(f"Episode {episode} finished with reward: {total\_reward}")  
 episode\_reward.append(total\_reward)  
 running = False

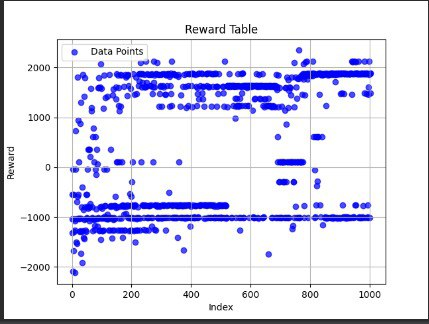
# چندین نمونه کار از کد:

مثال 1

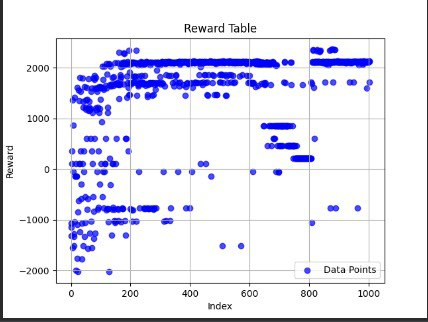


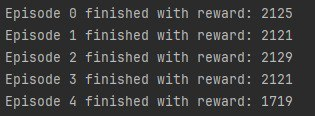


تست دوم( در اینجا در کنار پرنده آبی TNT قرار داشت و احتمال برخورد به آن بود)

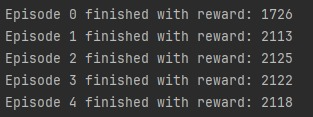
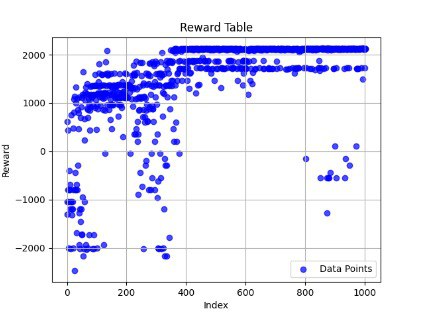


تست سوم





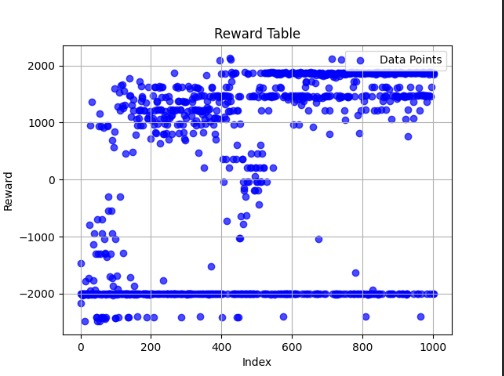
تست چهارم



همراه با فیلم

تست پنجم

TNT در ایندکس [1,1] بود

  
همراه با فیلم

مثال ششم

