卒業研究報告書

題目

シューティングゲームにおける自律的武器選択アルゴリズム

指導教員

石水 隆 講師

報告者

21-1-037-0255

内田 陸来

近畿大学理工学部情報学科

令和 7 年 2 月 2 日提出

概要

シューティングゲーム（以下,STG）は,敵の動きや攻撃パターンが多様で予測が難しく,戦略的な判断が求められるゲームである.また,多くの STG ではプレイヤーが操作する機体（以下,自機）が複数の特性を持つ武器を装備し,状況に応じて適切な武器を選択する必要がある.例えば,広範囲に攻撃できる武器や短時間で大きなダメージを与える武器などが挙げられ,それぞれの使用タイミングはプレイヤーの判断に依存する.このような特性を持つ STG は,ゲーム AI の評価や検証に適した題材であり,複雑な環境下での意思決定能力を強化学習で検証するためのモデルとして適している.

本研究では,強化学習手法である DQN（Deep Q Network）を用いて,複数の武器を持つ自機が敵の出現状況や攻撃パターンに応じて武器を選択し,効率的に敵を撃破する AI を開発することを目的とした.自作の STG 環境を構築し,通常弾,スピード弾,拡散弾という 3 種類の武器を使用できるように設計した.さらに,各武器の使用基準を定義し,AI が状況に応じて適切な武器を選択できるかを検証した.

DQN を用いて 10000 エピソードの学習を行い,100 エピソードごとにスコアの平均値を算出して学習の推移を分析した.その結果,AI のスコアにはばらつきがあり,学習回数が増加してもスコアの安定的な向上は確認されなかった.また,スピード弾が最も高い効果を示した一方で,使用時間が他の武器と比較して短い傾向が見られた.このことから,AI が報酬の増加と敵や敵弾の撃破との関連性を十分に学習できていないことが示唆された.さらに,敵や敵弾の数が多いエリアに自機がいる時間の割合が高いとスコアが低くなる傾向が見られたが,この関係性も安定せず,AI は敵弾回避の学習も十分に行えていないことが分かった.

これらの課題の要因として,報酬設計における複数の増加要因（攻撃行動や回避行動の成功）が存在し,AI が報酬の増加の原因を明確に理解できていないことが考えられる.また,学習エピソード数が十分でないことも一因であると推測される.これらを踏まえ,報酬設計を単純化し,例えば攻撃行動が成功すると報酬が増加し,回避行動が失敗すると報酬が減少するように設計することで,AI が明確な学習目標を持てるようにする必要がある.また,学習エピソード数を増加させることで,スコアの安定化や状況に応じた最適な武器選択が可能になると考えられる.

目次

1. 序論 1

1.1 本研究の背景 1
1.2 本研究の目的 1
1.3 自動シューティングゲームに関する既知の結果 1
1.4 本報告書の構成 2

2 本研究で作成した STG 2

2.1 ゲーム内容 2

3 強化学習 4

3.1 Q 学習の基礎 5
3.2 DQN のアルゴリズム 5
3.3 DQN の手順 5

4 自作 STG への強化学習の適用方法 6

4.1 DQN の選定 6
4.2 状態空間と行動空間 6
4.3 報酬設計 7
4.4 学習プロセス 7

5 自作 STG のプログラム 7

5.1 エージェント 7
5.2 自機 8
5.3 弾 8
5.4 敵 8
5.5 敵の弾 8
5.6 環境 9

6 結果・考察 10

7 結論 12

謝辞 13

参考文献 14

付録 15

1. 序論

1.1 本研究の背景

シューティングゲーム(以下 STG とする)は,弾を発射することができる武器が装備された自機を操作し,次々に現れる敵機および敵機が発射する弾を避けながら敵機を撃破していくゲームである. そのサブジャンルとして他の STG よりも大量の弾が出現し,それを避けながら敵を倒す弾幕 STG や画面が強制的にスクロールされるスクロール STG,一人称視点で行い世間では FPS と呼ばれるファーストパーソン STG などがある.どの STG も一見すると単なる反射神経を競うだけゲームのようであるが, 実際には予測困難な敵の動きや多様な攻撃パターンに対応する必要がある戦略的要素の強いゲームジャンルである.また,多くの STG では自機が複数の異なる特性を持つ武器を装備しており,敵の状況に応じて最適な武器を選択する能力が求められる.武器には,広範囲攻撃を得意とするものや,高速弾を発射するものがあり,それぞれの効果的な使用タイミングが勝敗を大きく左右する.このように,STG は単なるプレイヤーの反射神経を試すだけでなく,複雑な状況判断が必要であるため,意思決定を要するゲーム AI の評価や検証に非常によい題材である.

一方,AI 技術の分野では,複雑な環境下で最適な行動を選択するための強化学習が進展している.特に,DQN(Deep Q Network)は,深層学習の能力を利用して大規模な状態空間を効率的に探索し,報酬に基づいて最適な行動を学習できる強力な手法として知られている.DQN は様々な分野において適用可能であり, STG のような動的で複雑なゲーム環境においても,DQN は非常に有効であると考えられており,近年,多くの研究が行われている.

1.2 本研究の目的

本研究の目的は,DQN を用いて STG における AI の意思決定能力を向上させることである.具体的には,複数の武器を持つ自機が,敵の出現パターンや攻撃状況に応じて最適な武器を選択し,効率的に敵を撃破するとともに,自機の生存を維持する柔軟な戦略を学習する AI を開発することを目指す.また,AI の学習効率を高めるため,報酬設計や観測範囲の調整を行い,攻撃行動と回避行動をバランスよく学習できるようにする.この研究を通じて,人間のプレイに近いリアルな意思決定を行える AI モデルを提案することを目指す.

1.3 自動シューティングゲームに関する既知の結果

先行研究では,STG における AI 開発において,DQN の有用性が多く報告されている.野村らは,DQN を用いてプレイヤーや敵の位置情報など AI に与える情報を全体から自機周辺に限定することで学習効率を向上させる手法を提案した.この方法は弾幕 STG に適用され,狭い観測範囲での学習が効果的であることが示されたが,人間のプレイ水準には未だ及ばない結果が得られている[2].また,平井らは人間の視覚特性を AI に取り入れる手法を提案し,敵や敵の弾の位置など AI に与える情報に自機から遠いほど大きなズレを生じさせる処理をすることで人間の視覚を再現することで,人間らしい意思決定を実現する可能性を示した[3].藤本は,DQN を用いて報酬設計を工夫し,回避行動と攻撃行動を適切にバランスさせることで,AI の効率的な学習を実現したが,より高い精度の意思決定にはさらなる改良が必要であると述べている[4]. さらに,中川らの研究では,報酬を細分化することが AI の学習結果に大きく影響することが示された.具体的には,回避行動や攻撃行動の優先順位を明確化する報酬設計が,AI の意思決定能力を向上させると報告されている[5].

本研究では, これらの既存知見を踏まえつつ, STG の人工知能において特に複数の武器選択に焦点を当てた AI モデルを開発し, より人間に近い意思決定を可能にするアプローチを提案する.

1.4 本報告書の構成

本報告書は，研究内容，実際に使用したプログラム，結果および考察，結論の順で構成されている．第 2 章では，研究に使用したゲーム内容について述べる. 第 3 章では強化学習について述べる. 第 4 章では，強化学習の適用方法について述べる．第 5 章では，自作したプログラムのソースコードを説明する．第 6 章では，実験結果とその考察を行い，第 7 章で研究の結論をまとめる.

2 本研究で作成した STG

本研究では,強化学習の手法を用いて,STG における人間らしい動きを行える人工知能の構築法を検証する．人工知能の「人間らしい動き」を本研究は明確な理由のもと武器を選択することと定義する. 強化学習の適用対象として自作の STG を構築した.本ゲームの自機は 3 種類の武器（通常弾,スピード弾,拡散弾）を持ち,これらを自律的な意思決定のもと選択しながら敵を撃破し,敵弾を回避することが求められる.選択する武器と理由は以下のものとする.

- 通常弾: 敵を効率よく倒すために使用する.
- スピード弾: 短時間で多数の弾を発射し,素早く敵や敵の弾にダメージを与える.
- 拡散弾: 広範囲に発射され,敵弾が自機を囲む状況で回避を兼ねた攻撃を行う.

敵キャラクターは複数のタイプが存在し,それぞれ異なる HP や攻撃パターンを持つ.敵は一定時間経過後に再出現し,プレイヤーに衝突する,または自機弾に当たることでダメージを受ける仕組みとなっている.さらに,ゲームオーバー判定や無敵時間の管理機能も実装し,現実的なゲーム環境を模倣した.

本ゲームは Python および Pygame を用いて開発され,研究における AI の学習環境として機能する設計とした.

2.1 ゲーム内容

図 1 に本研究で作成した STG のプレイ画面を示す．以下に自作 STG の内容を記載する.

・自機　:黒色の船
　　ライフ　　: 初期値は 3 である.敵もしくは敵の弾に当たるとライフが 1 減る.
　　0 になるとゲーム終了.
　　無敵状態　:ライフが減ると 2 秒間敵や敵の弾に干渉しなくなる.
　　行動　　:左,右,上,下,狙撃,弾変更(通常弾,スピード弾,拡散弾)
・弾　　:鼠色の菱形
　　[種類:攻撃力,速度,クールタイム(秒), タイプ,特徴]
　　通常弾　　:5,7 ,0.5,Normal,標準装備
　　スピード弾: 1,12 ,0.2, Speed,他の球よりも速い
　　拡散弾　　: 3,7 ,0.7, Spread,3 方向（真上を基準とし左右に 30 度）に発射される
・敵 1　:緑色の戦車(弾は赤色のリンゴの芯)
　　体力　　:5
　　弾[速度,クールタイム(秒), タイプ,特徴]
　　:5,2,EnemyBullet,真下に発射される.

数　　:縦 2,横 11 の最大 22 体
リスポーン:3 秒

・敵 2　:赤色のヘリコプター(弾は赤色のリンゴの芯)
体力　　:15
弾[速度,クールタイム(秒), タイプ,特徴]
:5,2, EnemySnipeBullet,自機の位置に向けて発射される.
数　　:縦 2,横 11 の最大 22 体
リスポーン:3 秒

・敵 3　:水色の UFO(弾は赤色のリンゴの芯)
体力　　:50
弾[速度,クールタイム(秒), タイプ,特徴]
:5,3, EnemySpreadBullet,3 方向（真下を基準とし左右に 30 度）に発射される
数　　:最大 1 体
リスポーン:5 秒
移動　　:画面を左右に移動.

・その他
画面サイズ:幅 546,高さ 520
表示項目　:Bullet(弾のタイプを表示)
Action(行動を表示[行動に対応する各種番号は下記の行動空間で示す])
Score(AI の行動の評価を数値化したものの合計を表示)
画面分割　:画面を 36 個に分割し各エリアの敵や敵の弾の数を表示.
自機に近い(自機からの距離が 50 以内の)弾は水色の弾に変わる.

・ルール
自機を動かし敵の弾を避けながら敵に弾を撃ち, 倒していく.敵や敵の弾に 3 回当たるとゲームオーバーになりゲームが終了する.

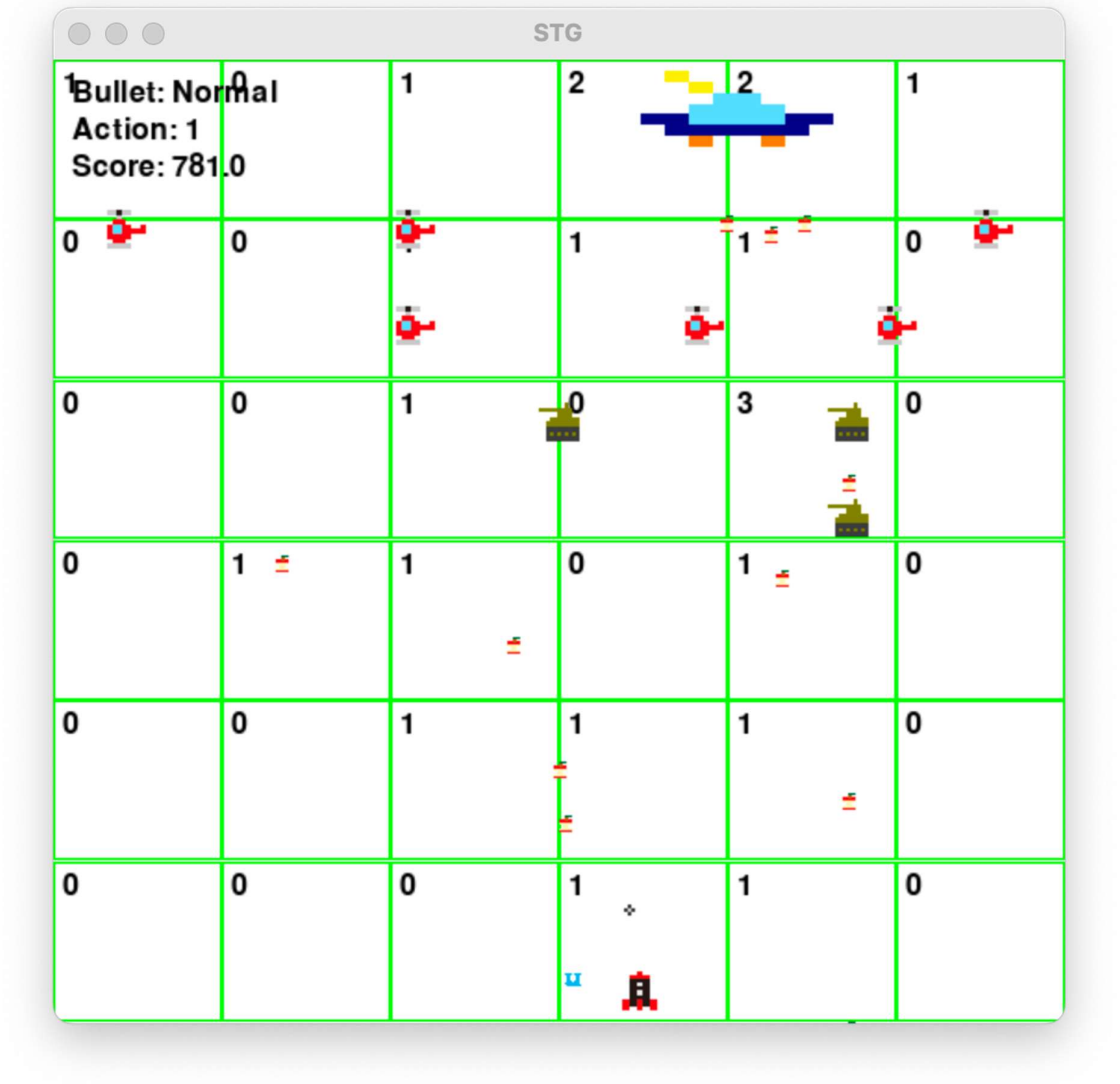

図 1 ゲーム画面

3 強化学習

本章では強化学習の手法について述べる.

強化学習は,AI などのエージェントが行動を選択し報酬を得ることを繰り返すことで報酬を最大化するための最適な方法を学習する手法である.この手法では,エージェントが状態 s において行動 a を選択し,環境から次の状態 s′と報酬 r を受け取ることで学習が進む.これを数学的にはマルコフ決定過程(Markov Decision Process, MDP) として表現する.DQN は,この強化学習を深層学習と組み合わせた手法であり,従来の Q 学習の Q 値関数 Q(s,a) を,ニューラルネットワークを用いて近似することで高次元な状態空間にも適用可能にしている[8].

3.1 Q 学習の基礎

Q 学習では,最適な行動方針を学習するために,以下のベルマン方程式を用いる：

$$Q*(s,a) = E_{s'\sim\varepsilon}[r + \gamma \max Q*(s',a') \mid s,a]$$

・$Q*(s,a)$は状態sで行動aを選択したときの最大累積報酬（期待値）を表す.

・γ は割引率であり,将来の報酬に対する重みを調整する.

・E は期待値を示す.

この方程式をもとに,Q 値を更新することで最適な方法が得られる.しかし,状態空間が高次元の場合,すべての(s, a)を計算しないといけないため,ニューラルネットワークを活用した DQN が提案された

3.2 DQN のアルゴリズム

DQN では,Q 値関数を近似するために重み θ を持つニューラルネットワーク Q(s,a;θ)を利用する.このネットワークの学習は,以下の損失関数を最小化することで行われる：

$$L(\theta) = E_{s,a\sim D}[(y - Q(s,a;\theta))^2]$$

・$y = E[r + \gamma \max Q*(s',a'\theta) \mid s,a]$はターゲット Q 値.

・Dはリプレイバッファと呼ばれる経験(過去の行動,状態,報酬など)の記憶データセット

リプレイバッファを用いることで,過去の経験を再利用し,学習の効率と安定性を向上させている.また,ターゲットネットワークは一定のタイミングで更新されるため Q 値の急激な変化がなく安定している.そのためそれを導入することにより,学習中の不安定さを抑えている.

3.3 DQN の手順

1,初期化:ニューラルネットワーク,ターゲットネットワーク,リプレイバッファを初期化
2.行動選択:現在の状態を基に探索率εの確率でタランダム 1-ε の確率で最も Q 値の高かった行動を選択する(ε-greedy 法).
3.環境で動作: 行動 a を実行し,次の状態 s'と報酬 r を観測.
4.経験の保存: (s,a,r,s′) をリプレイバッファに保存.
5.ミニパッチ更新: リプレイバッファから指定した分だけランダムに選び損失を計算.
6.パラメータ更新:損失関数の変化量を計算して重みを更新する.
7.ターゲットネットワークの更新:一定のタイミングでターゲットネットワークを更新
8.終了:終了条件を達成すると終わる.

4　自作 STG への強化学習の適用方法

本章では，自作 STG への強化学習の適用方法について述べる.

4.1 DQN の選定

本研究では,自作 STG の学習方法として，深層強化学習アルゴリズムである DQN を採用した.採用理由はゲームの高次元な状態空間（例: プレイヤーや敵の位置，弾丸の軌道）に対応できる点と,行動空間が離散的であることが DQN の特性と一致しているためである.さらに,DQN はニューラルネットワークを活用して複雑なパターンを学習し,経験再生やターゲットネットワークで学習を安定化させる.これにより,状況に応じた武器選択や敵の効率的な撃破が可能となるため,本研究で採用した.

4.2 状態空間と行動空間

本研究では，AI の入力として,以下の情報を状態空間として定義した.

・自機
　　:x 座標,y 座標,ライフ,弾のタイプ(ship_x,ship_y,ship_life,bullet_type)
・敵(自機から近い順に 5 体※敵 3 を除く)
　　: x 座標,y 座標,敵の種類(enemy_x, enemy _y, enemy _type)
・敵 3
　　: x 座標,y 座標 (enemy3_x, enemy 3_y)
・弾(自機から最も近い)
　　: x 座標,y 座標,速度(bullet _x, bullet _y, bullet _speed)
・敵の弾(自機から近い順に 5 つ)
　　: x 座標,y 座標,速度,弾のタイプ(enemy _bullet _x, enemy _bullet _y,
　　enemy _bullet _speed, enemy _bullet _ type)
・スコア
　　:直前のスコア(score)

行動空間は,自機が取り得るすべての行動を含み,以下のように定義した.

・行動　　:(Action 欄に表示される番号)
・左移動　　:0
・右移動　　:1
・上移動　　:2
・下移動　　:3
・発射　　:4
・弾変更(通常弾)　　:5
・弾変更(スピード弾)　:6
・弾変更(拡散弾)　　:7

4.3 報酬設計

学習を効果的に進めるため,以下の報酬設計を行った.

増加報酬

・敵に弾が当たった時: +2(敵 1), +6(敵 2), +10(敵 3)
・敵を倒した時: +10(敵 1), +30(敵 2), +50(敵 3)
・敵を生成されてすぐ倒した時:+10
・敵の弾を撃ち落とした時: +5, +10(近くの弾)
・次に被弾するまでの時間が 3 秒以上の時: +1
・敵や敵の弾を倒してから次に倒すまでの時間が一秒以内かつ弾の種類がスピード弾の時 :+5
・近くの敵の弾を撃ち落とした時の弾の種類が拡散弾の時 :+5
・敵を倒した時の弾の種類が通常弾の時 :+5

減少報酬

・弾が対象に当たらなかった時: -1
・ライフが減った時: -20(1,2 回), -60(3 回)
・近くに弾がある時: -1
・画面全体を分割し敵や敵の弾が最も少ないところにいる時:+1,それ以外: -1
・自機が画面枠を越えようとする行動をとる時 :-1

4.4 学習プロセス

本節では，AI エージェントの学習プロセスについて述べる.

・学習データの収集

AI エージェントは,自作 STG 内での試行錯誤を通じてデータを収集する.学習は 10000 エピソードを単位として実施し,エピソード終了時にはスコア,自機が敵や敵の弾の数が最も多い場所・少ない場所にいた時間,弾の種類ごとの使用時間・1 秒以内に倒した対象の数・倒した敵の数・撃ち落とした敵の近い弾の数を記録する.

・学習アルゴリズムの設定

DQN の実装において,以下のハイパーパラメータを設定した.

・学習率: 0.001
・割引率（γ）：0.99
・ε-greedy 法による探索率: 初期値 1.0 から徐々に減少（0.1 まで）
・経験再生バッファサイズ: 2000 ステップ

5 自作 STG のプログラム

本章では，本研究で作成したプログラムについて説明する.付録にソースプログラムを示す.

5.1 エージェント

・クラス
　・DQN:DQN を用いて,より最適な行動を学習するクラス
・メソッド
　・build_model:ニューラルネットワークを構築する.
　・remember:スコアや状態,行動を経験として保存.
　・choose_action:ε-グリーディ法で行動を選択.
　・train:リプレイバッファからサンプリングしたデータで Q ネットワークを更新.
　・update_target_network:ターゲットネットワークの更新.

:

5.2 自機

・クラス
　　・Ship:プレイヤーが操作する機体.弾を発射できる.
・メソッド
　　・update:移動や状態の更新.画面への描画.
　　・shoot:弾の種類に応じた射撃.
　　・take_damage:被弾した時の処理(残機減少,無敵状態への移行).
　　・invincible_timer:無敵状態の時間管理.
　　・gameover:残機が 0 になったときの処理.

5.3 弾

・クラス
　　・Bullet:通常弾.攻撃力が高い.:
　　・SpeedBullet:スピード弾.飛ぶ速度が速い.
　　・SpreaedBullet:拡散弾.弾を 3 方向に発射（直進,左 30 度,右 30 度）.

・メソッド
　　・update:各クラスにあった弾の移動処理.画面外にでた時の処理.

5.4 敵

・クラス
　　・Enemy1:体力 5,発射感覚 2 秒,緑色の戦車.
　　・Enemy2:体力 15,発射感覚 3 秒,赤色のヘリコプター.
　　・Enemy3:体力 50,発射感覚 3 秒,水色の UFO.
・メソッド
　　・update: 移動(Enemy3 のみ)や状態の更新.画面への描画.
　　・take_damage:自機の弾があったときのダメージ処理.
　　・hide:倒されたときに非表示にして再出現の待機.
　　・check_respawn:再出現処理.
　　・shoot:各敵が搭載する弾を発射.

5.5 敵の弾

・クラス
　　・EnemyBullet:敵の通常弾, Enemy1 に搭載.
　　・EnemySnipeBullet: 自機を狙って発射, Enemy2 に搭載.
　　・EnemySpreaedBullet: 敵の拡散弾, Enemy3 に搭載.

・メソッド
　　・update: 各クラスにあった弾の移動処理.画面外にでた時の処理.

5.6 環境

・クラス

・Game:ゲーム全体

・メソッド

・get_game_state:現在の状態を数値で表した配列を取得する.

・encode_bullet_type:弾の種類を数値で表現.

・encode_enemy_type: 敵の種類を数値で表現.

・encode_enemy_bullet_type: 敵の弾の種類を数値で表現.

・get_state_size:状態の大きさ(次元数).

・update_and_calculate_reward:ゲーム画面の描画と更新.報酬の計算.

・grid_screen:ゲーム全体を分割し各エリアの敵や敵の弾の数の測定.
自機が敵や敵の弾の数が最も多い,少ないエリアにいる時間の測定.

・check_boundary:自機が画面枠を超える行動をしたときに負の報酬を与える処理.

・reset_game:ゲームの状態を初期化.

・create_enemy_positions:敵の生成位置のリストの作成.

・spawn_enemy:敵 1,2 をランダムに作成.

・display_bullet_type: 現在選択されている弾の種類を画面に表示.

・display_action_and_score: 現在の行動とスコアを画面に表示.

・display_density_grid: ゲーム全体の分割,各エリアの敵や敵の弾の数を可視化.

・reset_episode_stats:各エピソードの記録を初期化.
(記録: 報酬, 最も多いエリアに自機がいる時間,
最も少ないエリアに自機がいる時間, 弾各種の使用時間,
弾各種の倒した敵, 弾各種の 1 秒間に当てた数,
撃ち落とした近くの弾の数, 弾を最初に当てた時間,
弾を使い始めた時間)

・log_episode_result:各エピソードの報酬, 最も多いエリアに自機がいる時間,
最も少ないエリアに自機がいる時間, 弾各種の使用時間を表示.

・update_bullet_usage_time:弾各種の使用時間を記録.

・log_hit_within_1_seconds:1 秒以内に敵や敵の弾に当てた数を記録.

・log_nearby_bullet_destruction: 撃ち落とした近くの弾の数を記録.

・log_kill:倒した敵を種類ごとに数を記録.

・log_final_results: 指定したエピソードの結果(log_episode_result と同じ)を最後に表示.

・log_n_episode_results:指定したエピソード数ごとに弾各種の倒した敵,
弾各種の 1 秒間に当てた数, 撃ち落とした近くの弾の数を記録,
表示.

・log_final_scores:全エピソードのスコアを羅列表示.

・end:ゲーム終了処理.

6　結果・考察

本研究では,DQN を用いて自作 STG のエージェントに対して学習を行い,その学習成果をスコアに基づいて評価した.学習回数とスコアの関係を図 2 に示す.この実験では 10,000 回のエピソードを学習させ,そのうち 100 回ごとのスコアの平均値を算出した.図 2 に示されるように,AI のスコアには一定のばらつきが見られ,学習の進行に伴うスコアの明確な増加傾向は観測されなかった.このことは,AI が適切な行動方針を一貫して学習できず,学習の安定性を欠いていることを示唆する.また,スコアのばらつきが大きいことから,学習中の行動選択が試行錯誤的であり,最適化に向かう一定の指針を獲得できていない可能性が考えられる.

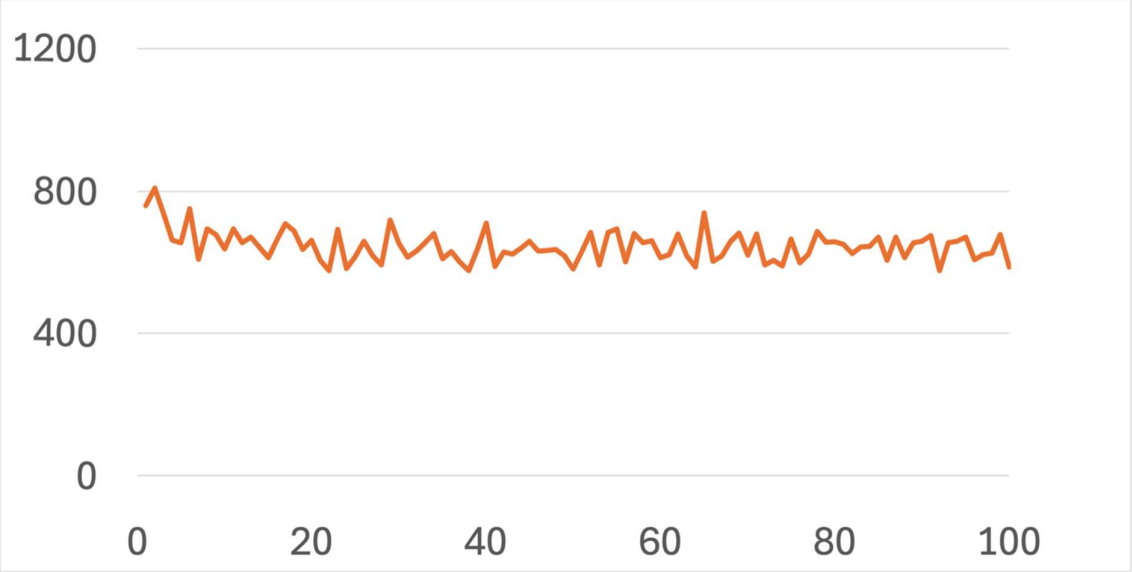

図 2 学習回数とスコア

次に,AI が学習中に用いた弾の種類ごとの使用状況およびその効果を解析した.その結果を図 3 および表 1 に示す.図 3 は 100 エピソードごとの各弾の種類ごとの総使用時間の平均の割合を示し,表 1 は全エピソードを通じた弾の種類ごとの各トータル記録を示している.

図 3 から,スピード弾の使用時間が他の弾種と比較して明らかに短いことがわかる.一方で,表 1 の記録から,スピード弾は全ての評価指標において最も高い効果を発揮している.この結果は,スピード弾が戦闘において極めて有効であるにもかかわらず,AI がその使用を効果的に学習できていないことを示している.このことから,AI が報酬の増加と敵や敵弾の撃破の関連性を十分に学習できていないと推察される.

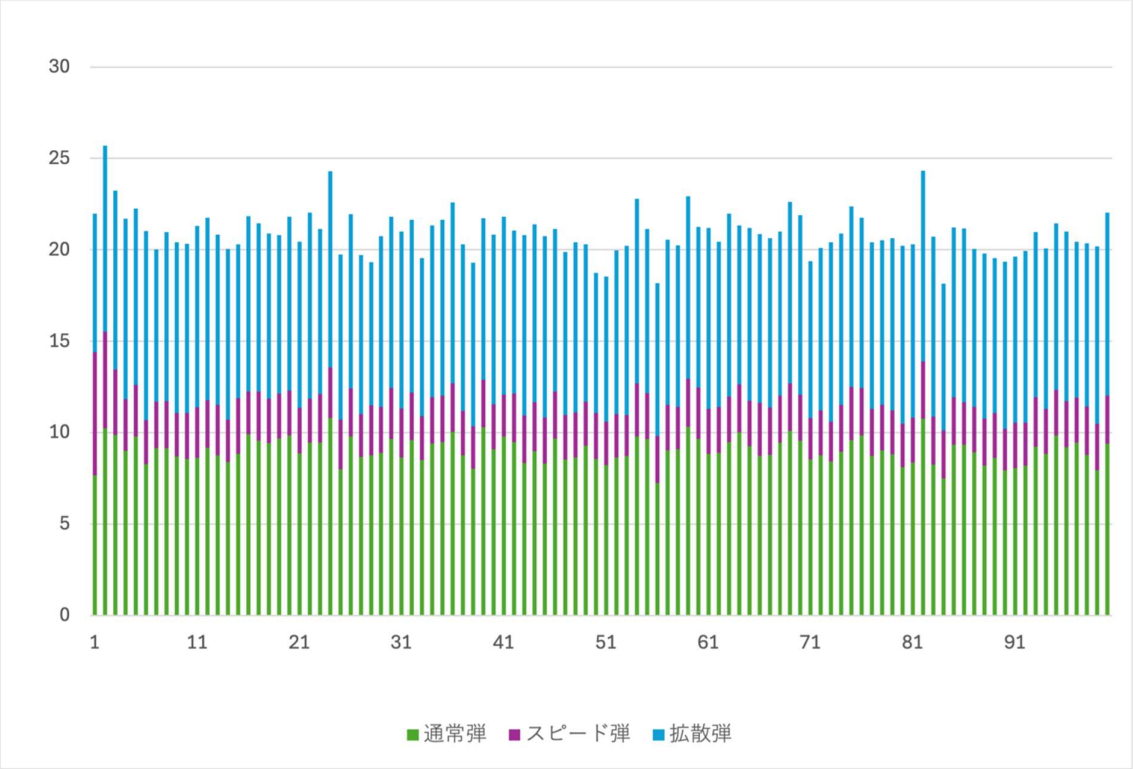

図 3 弾の種類ごとの使用時間

表 1 弾の種類ごとの各記録

	通常弾	スピード弾	拡散弾
1 秒以内に倒した対象の数	2568	11642	3188
倒した敵の数(敵 1)	41	174	35
倒した敵の数(敵 2)	5	10	2
倒した敵の数(敵 3)	0	0	0
撃ち落とした敵の近い弾の数	373	4330	383

次に画面を分割し,各エリアにおける敵や敵弾の数を計測し,その数が最も少ないエリアや多いエさらに,画面を分割し,各エリアにおける敵や敵弾の分布を計測するとともに,自機がそれぞれのエリアに滞在した時間を分析した.これにより,敵弾の密集度に応じた回避行動がスコアにどのように影響を与えているかを評価した.結果を図 4 に示す.青色の折れ線グラフは,1000 エピソードごとの最も少ないエリアと最も多いエリアに自機がいる時間の割合（多いエリアに自機がいる時間 / 少ないエリアに自機がいる時間）を示し,オレンジ色の折れ線グラフは 1000 エピソードごとのスコアを示している. 図 4 によると,スコアが高い場合には,多いエリアに自機がいる時間の割合が低くなる傾向が見られる.しかし,グラフ全体の傾向にはばらつきがあり,スコアおよび割合のどちらも安定していない.このことから,報酬の増加と敵弾の回避行動との関連性を AI が十分に学習できていないことが示された.

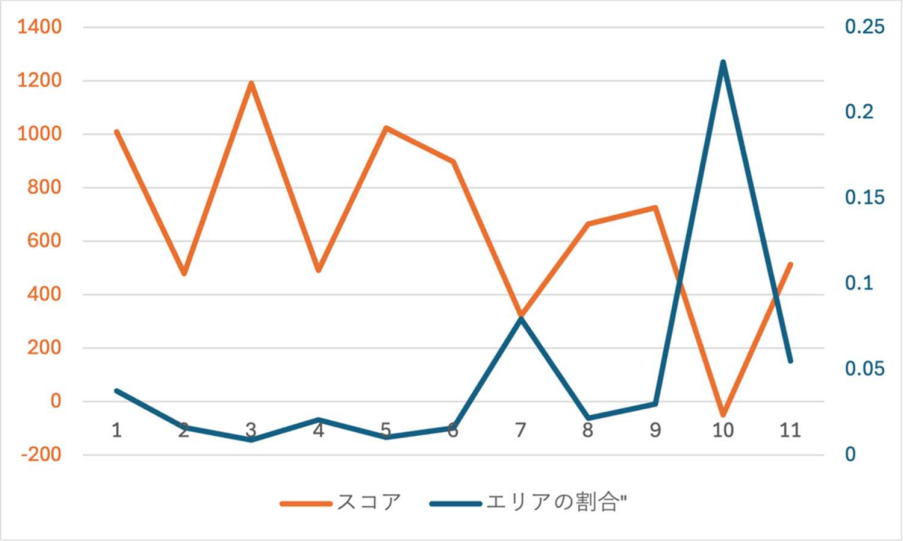

図 4 エリアの割合とスコア

報酬の増加と行動との関連性を AI が十分に学習できていない原因として,報酬設計における増加要因が回避行動と攻撃行動の両方に存在するため,AI は報酬の増加の原因を明確に理解できていないと考える.また,これらの要因を理解するための学習エピソード数が不足していた可能性がある.しその原因を解消するため報酬をたとえば攻撃行動を成功すると報酬の増加,回避行動ができない・失敗すると報酬の減少とするなど増加要因と減少要因となる行動を単純化し,エピソード数を増加させることで,スコアを安定させ,状況に応じた明確な武器選択が可能になると考える.

7 結論

本研究では,DQN を用いて STG における武器選択の学習可能性を検証した.結果として,AI のスコアは安定せず,弾選択や敵撃破の学習にも課題が見られた.特に,スピード弾の使用頻度が他の弾より少なく,報酬と敵撃破や弾回避の関連性を十分に学習できなかった.報酬設計における複数の増加要因や学習エピソード数の不足が影響したと考えられる.今後は報酬設計を単純化し,エピソード数を増加させることで,効率的な学習と武器選択の向上が期待される.

謝辞

本研究を行うにあたり,石水隆講師には研究の助言,レジュメや卒論の添削などのご指導を受けました.ここに感謝の意を表します.

参考文献

[1] 松浦健一郎, 司ゆき: シューティングゲームプログラミング, Softbank Creative (2006)

[2] 野村直也, 橋本剛: 弾幕シューティングゲームを対象とした汎用的学習法, 情報処理学会 研究報告ゲーム情報学, Vol.2018-GI-39, No.4, pp.1-7 (2018)
https://id.nii.ac.jp/1001/00186014/

[3] 平井弘一, Reijer Grimbergen: 弾幕の認識に人間の視覚特性を取り入れたシューティングゲーム AI の研究, 情報処理学会 ゲームプログラミングワークショップ 2016 論文集, Vol.2016, pp.158-161 (2016)
http://id.nii.ac.jp/1001/00175321/

[4] 藤本修嗣: 強化学習を用いた弾幕シューティングゲームを攻略するエージェントの作成, 情報処理学会 第 29 回ゲームプログラミングワークショップ 2024 論文集, pp.51-58 (2024)
http://id.nii.ac.jp/1001/00240606/

[5] 中川翔太, 永田裕一: DQN を用いたシューティングゲーム AI の制作, 第 48 回知能システムシンポジウム論文集, pp.2-1 - 2-6 (2021)
https://www.sice.or.jp/org/i-sys/is48/paper/SICE-IS_2021_paper_2.pdf

[6] 王耀: ゲーム AI における深層学習の応用可能性, 日本ゲーム学会誌, Vol.2020, No.5, pp.12-20 (2020)
https://www.jasga.or.jp/gameai2020.pdf

[7] 山田悠: 強化学習を用いたゲーム AI の最適化手法に関する研究, 京都大学学術出版局 (2023)
https://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/260193/1/AIS_KU2023.pdf

[8] Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Playing Atari with Deep Reinforcement Learning, arXiv preprint arXiv:1312.5602 (2013)
https://arxiv.org/pdf/1312.5602

付録

本研究で作成した STG のソースプログラムを以下に示す.

STG.py

```
from pygame import *
from pygame.locals import *
import sys
from random import *
import numpy as np
import math
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque

# 画像集
SHIP1 = image.load("ロケット.png")
SHIP2 = image.load("うんこさん.png")
BULLET1 = image.load("手裏剣.png")
BULLET2 = image.load("りんごの芯.png")
BULLET3 = image.load("パンの袋を留めるアレ.png")
ENEMY1 = image.load("戦車.png")
ENEMY2 = image.load("ヘリコプター.png")
ENEMY3 = image.load("UFO.png")
IMAGES = {
    'ship': SHIP1,
    'fuck': SHIP2,
    'bullet1': BULLET1,
    'bullet2': BULLET2,
    'bullet3':BULLET3,
    'enemy1': ENEMY1,
    'enemy2': ENEMY2,
    'enemy3': ENEMY3
}

# 色　R, G, B
WHITE = (255, 255, 255)
```

```
# スクリーンサイズ
SCREEN = display.set_mode((546, 520))
display.set_caption("STG")
SCREEN_WIDTH=546
SCREEN_HEIGHT = 520


class DQN:
    def __init__(self, state_size, action_size, gamma=0.99, lr=0.001, epsilon=1.0,
    epsilon_decay=0.99954, epsilon_min=0.1, memory_size=2000, batch_size=64, target_update=10):
        self.state_size = state_size #状態サイズ
        self.action_size = action_size #行動サイズ
        self.gamma = gamma   # 割引率
        self.lr = lr   # 学習率
        self.epsilon = epsilon   # 探索率
        self.epsilon_decay = epsilon_decay#探索率の減少率
        self.epsilon_min = epsilon_min #探索率の最低値
        self.memory = deque(maxlen=memory_size)   # 経験再生バッファ
        self.batch_size = batch_size#バッチサイズ
        self.target_update = target_update#ターゲットネットワークの更新頻度

        # Q ネットワークとターゲットネットワーク
        self.q_network = self.build_model()#Q ネットワーク構築
        self.target_network = self.build_model()#ターゲットネットワーク構築
        self.target_network.load_state_dict(self.q_network.state_dict())   # 同期
        self.target_network.eval()   # ターゲットネットワークは更新しない

        self.optimizer = optim.Adam(self.q_network.parameters(), lr=self.lr)#Q ネットワークのパラメータ学習
        self.loss_fn = nn.MSELoss()   # 現在の Q ネットワーク値とターゲット値の誤差を測定


    #ニューラルネットワークの構築
    def build_model(self):
        return nn.Sequential(
            nn.Linear(self.state_size, 128),#入力層
            nn.ReLU(),#学習を効率化
            nn.Linear(128, 128),#隠れ層
```

```
            nn.ReLU(),
            nn.Linear(128, self.action_size)#出力層
        )

    #経験を保存

    def remember(self, state, action, reward, next_state, done):



        self.memory.append((state, action, reward, next_state, done))

    # ε -グリーディ法で行動を選択
    def choose_action(self, state):
        if uniform(0, 1) < self.epsilon:#学習率が低くなるほど探索しなくなる
            return randint(0, self.action_size - 1)  # ランダムな行動
        state = torch.FloatTensor(state).unsqueeze(0)  # 入力をテンソル化し次元を追加
        with torch.no_grad():
            q_values = self.q_network(state)#Q 値出力
        return torch.argmax(q_values).item()  # 最大の Q 値を持つ行動

    #学習
    def train(self):
        if len(self.memory) < self.batch_size:
            return

        # ランダムサンプリング
        batch = sample(self.memory, self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        # データの形を直す
        states = torch.stack([
            state.squeeze(0) if isinstance(state, torch.Tensor) and state.ndimension() == 2 and
    state.size(0) == 1
            else torch.FloatTensor(state)
            for state in states
        ])
        next_states = torch.stack([
            state.squeeze(0) if isinstance(state, torch.Tensor) and state.ndimension() == 2 and
    state.size(0) == 1
```

```
                else torch.FloatTensor(state)
                for state in next_states
            ])
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
            dones = torch.FloatTensor(dones)



            # 現在の Q 値
            q_values = self.q_network(states).gather(1, actions.unsqueeze(1)).squeeze(1)

            # 次の状態の Q 値
            with torch.no_grad():
                max_next_q_values = self.target_network(next_states).max(1)[0]

            # ターゲット Q 値
            target_q_values = rewards + self.gamma * max_next_q_values * (1 - dones)

            # 損失を計算
            loss = self.loss_fn(q_values, target_q_values)

            # ネットワークの更新
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
        #ターゲットネットワークの更新
        def update_target_network(self):

            self.target_network.load_state_dict(self.q_network.state_dict())

# 自機
class Ship(sprite.Sprite):
    life = 3#残機
    invincible = False  # 無敵状態
    invincible_duration = 2000  # 無敵時間（ミリ秒）
    last_hit_time = 0  # 最後に当たった時間
    bullet_type = 'Normal'  # 初期設定は普通弾

    def __init__(self):
```

```
        sprite.Sprite.__init__(self)
        self.image1 = transform.scale(IMAGES['ship'], (26, 26))#自機の画像
        self.image2 = transform.scale(IMAGES['fuck'], (26, 26))#無敵状態の画像
        self.image = self.image1#現在の時期の状態の画像
        self.rect = self.image.get_rect(topleft=(273, 468))#自機のサイズ
        self.speed = 5#移動速度

        self.last_bullet_time = 0   # 最後に弾を発射した時間
        self.normal_bullet_cooldown = 500   # 通常弾のクールダウン（ミリ秒）
        self.speed_bullet_cooldown = 200   # スピード弾のクールダウン（ミリ秒）
        self.spread_bullet_cooldown = 750   # 拡散弾のクールダウン（ミリ秒）



def update(self):

        self.rect.clamp_ip(Rect(0, 0, 546, 520))#画面外に出ないように
        self.invincible_timer()
        self.gameover()

        game.screen.blit(self.image, self.rect)
#狙撃
def shoot(self):
        # 現在時刻を取得
        current_time = time.get_ticks()
        # 弾の種類によってクールダウン時間を設定
        #通常弾の場合
        if self.bullet_type == "Normal":
            if current_time - self.last_bullet_time > self.normal_bullet_cooldown:#クールタイムより発車経過時間が長い場合

                bullet = Bullet(self.rect.centerx, self.rect.top)
                game.all_sprites.add(bullet)
                game.bullets.add(bullet)
                self.last_bullet_time = current_time   # 最後の発射時間を更新
        #スピード弾の場合
        elif self.bullet_type == "Speed":
            if current_time - self.last_bullet_time > self.speed_bullet_cooldown:

                bullet = SpeedBullet(self.rect.centerx, self.rect.top)
```

```
                game.all_sprites.add(bullet)
                game.bullets.add(bullet)
                self.last_bullet_time = current_time  # 最後の発射時間を更新
        #拡散弾の場合
        elif self.bullet_type == "Spread":
            if current_time - self.last_bullet_time > self.spread_bullet_cooldown:

                # 真っ直ぐ
                bullet_center = SpreadBullet(self.rect.centerx, self.rect.top, 0)
                # 左に 30 度
                bullet_left = SpreadBullet(self.rect.centerx, self.rect.top, -30)
                # 右に 30 度
                bullet_right = SpreadBullet(self.rect.centerx, self.rect.top, 30)

                # まとめてグループに追加
                game.all_sprites.add(bullet_center, bullet_left, bullet_right)
                game.bullets.add(bullet_center, bullet_left, bullet_right)
                self.last_bullet_time = current_time

    def take_damage(self):
        current_time = time.get_ticks()
        if not self.invincible:  # 無敵状態でなければダメージを受ける
            self.life -= 1
            self.invincible = True  # 無敵状態に移行
            self.last_hit_time = current_time  # 最後にヒットした時間を記録
            self.image = self.image2  # 見た目を変更

    def invincible_timer(self):
        current_time = time.get_ticks()
        if self.invincible and current_time - self.last_hit_time >= self.invincible_duration:#無敵状態か
つ無敵時間が経過した時
            self.invincible = False  # 無敵状態を解除
            self.image = self.image1  # 見た目を元に戻す
    def gameover(self):
        if self.life <= 0:
            print("Game Over!")

# 弾クラス
class Bullet(sprite.Sprite):
```

```
    def __init__(self, x, y):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['bullet1'], (10, 10))#弾の画像
        self.rect = self.image.get_rect(center=(x, y))
        self.speed = -7  # 弾の速度（上方向）
        self.attack_power = 5  # 普通弾の攻撃力

    def update(self):
        self.rect.y += self.speed
        if self.rect.bottom < 0:
            self.kill()  # 画面外に出たら弾を削除する
        game.screen.blit(self.image, self.rect)

# スピード弾クラス
class SpeedBullet(sprite.Sprite):
    def __init__(self, x, y):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['bullet1'], (10, 10))
        self.rect = self.image.get_rect(center=(x, y))
        self.speed = -12
        self.attack_power = 1

    def update(self):
        self.rect.y += self.speed
        if self.rect.bottom < 0:
            self.kill()
        game.screen.blit(self.image, self.rect)

  # 拡散弾クラス
class SpreadBullet(sprite.Sprite):  # 拡散弾クラス
    def __init__(self, x, y, angle):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['bullet1'], (10, 10))
        self.rect = self.image.get_rect(center=(x, y))

        self.angle = angle
        self.speed = 7
        self.attack_power = 2
    def update(self):
```

```
            self.rect.x += self.speed * math.sin(math.radians(self.angle))#進む方向設定
            self.rect.y -= self.speed * math.cos(math.radians(self.angle))
            if self.rect.bottom < 0 or self.rect.left < 0 or self.rect.right > 546:
                self.kill()  # 画面外に出たら削除
            game.screen.blit(self.image, self.rect)

    # 敵 1
class Enemy1(sprite.Sprite):
    MAX_HP = 5#体力
    FIRE_RATE = 2000  # 2 秒ごとに弾を発射
    last_shot_time = 0  # 最後に弾を発射した時間

    def __init__(self, x, y):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['enemy1'], (26, 26))
        self.rect = self.image.get_rect(topleft=(x, y))
        self.spawn_time = None  # リスポーンの時間管理
        self.visible = True  # 敵の状態を管理(倒されたら False)
        self.original_position = (x, y)  # 元の位置を保持
        self.HP = self.MAX_HP  # HP を初期化

    def update(self):
        if self.visible:
            game.screen.blit(self.image, self.rect)
            self.shoot()

        if not self.visible:
            self.check_respawn()  # 非表示時はリスポーンさせる
        else:
            game.screen.blit(self.image, self.rect)
            self.shoot()

    def take_damage(self, damage):
        if self.visible:  # 表示されている場合のみダメージを受ける
            self.HP -= damage
            if self.HP <= 0:
                self.hide(3000)  # 一定時間非表示にして復活

    def hide(self, respawn_time):
```

```
            self.visible = False  # 敵を非表示にする
            self.spawn_time = time.get_ticks() + respawn_time  # 再生成時間を設定
            self.rect.topleft = (-100, -100)  # 画面外に移動して当たり判定を無効化

        def check_respawn(self):
            if not self.visible and time.get_ticks() >= self.spawn_time:
                self.visible = True  # 指定された時間後に再表示
                self.rect.topleft = self.original_position  # 元の位置に設定
                self.HP = self.MAX_HP  # HP をリセット
        def shoot(self):
            current_time = time.get_ticks()
            if current_time - self.last_shot_time > self.FIRE_RATE:  # 一定間隔で弾を発射
                bullet = EnemyBullet(self.rect.centerx, self.rect.bottom)#敵の通常弾
                game.all_sprites.add(bullet)
                game.enemy_bullets.add(bullet)
                self.last_shot_time = current_time
    # 敵 2
class Enemy2(sprite.Sprite):
    MAX_HP = 15
    FIRE_RATE = 3000
    last_shot_time = 0

    def __init__(self, x, y):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['enemy2'], (26, 26))
        self.rect = self.image.get_rect(topleft=(x, y))
        self.spawn_time = None
        self.visible = True
        self.original_position = (x, y)
        self.HP = self.MAX_HP

    def update(self):
        if self.visible:
            game.screen.blit(self.image, self.rect)
            self.shoot()
        if not self.visible:
            self.check_respawn()
        else:
            game.screen.blit(self.image, self.rect)
```

```
            self.shoot()

    def take_damage(self, damage):
        if self.visible:
            self.HP -= damage
            if self.HP <= 0:
                self.hide(3000)

    def hide(self, respawn_time):
        self.visible = False
        self.spawn_time = time.get_ticks() + respawn_time
        self.rect.topleft = (-100, -100)

    def check_respawn(self):
        if not self.visible and time.get_ticks() >= self.spawn_time:
            self.visible = True
            self.rect.topleft = self.original_position
            self.HP = self.MAX_HP

    def shoot(self):
        current_time = time.get_ticks()
        if current_time - self.last_shot_time > self.FIRE_RATE:

            chasing_bullet = EnemySnipeBullet(self.rect.centerx, self.rect.bottom, game.ship)     #敵の
    追尾弾
            game.all_sprites.add(chasing_bullet)
            game.enemy_bullets.add(chasing_bullet)
            self.last_shot_time = current_time
# 敵 3
class Enemy3(sprite.Sprite):
    MAX_HP = 50   # 最大 HP
    FIRE_RATE = 3000
    last_shot_time = 0
    enemy_type="enemy3"

    def __init__(self):
        sprite.Sprite.__init__(self)
        self.image = transform.scale(IMAGES['enemy3'], (130, 52))
        self.rect = self.image.get_rect(topleft=(208, 0))
```

```
        self.HP = self.MAX_HP
        self.spawn_time = None
        self.visible = True
        self.original_position = (208, 0)  # 初期位置
        self.direction = 1  # 右方向に移動
        self.speed = 3

    def update(self):
        if self.visible:

            self.rect.x += self.speed * self.direction#右に移動
            if self.rect.right >= SCREEN.get_width()+60 or self.rect.left <= -60:
                self.direction *= -1  # 壁にぶつかったら方向を反転
            game.screen.blit(self.image, self.rect)
            self.shoot()

        if not self.visible:
            self.check_respawn()

    def take_damage(self, damage):
        if self.visible:
            self.HP -= damage

            if self.HP <= 0:
                self.hide(3000)

    def hide(self, respawn_time):
        self.visible = False
        self.spawn_time = time.get_ticks() + respawn_time
        self.rect.topleft = (-100, -100)

    def check_respawn(self):
        if not self.visible and time.get_ticks() >= self.spawn_time:
            self.visible = True
            self.rect.topleft = self.original_position
            self.HP = self.MAX_HP

    def shoot(self):
        current_time = time.get_ticks()
```

```
        if current_time - self.last_shot_time > self.FIRE_RATE:

            bullet_center = EnemySpreadBullet(self.rect.centerx, self.rect.bottom, 0)#敵の拡散弾
            bullet_left = EnemySpreadBullet(self.rect.centerx, self.rect.bottom, -30)
            bullet_right = EnemySpreadBullet(self.rect.centerx, self.rect.bottom, 30)


            game.all_sprites.add(bullet_center, bullet_left, bullet_right)
            game.enemy_bullets.add(bullet_center, bullet_left, bullet_right)
            self.last_shot_time = current_time

# 敵の通常弾クラス
class EnemyBullet(sprite.Sprite):
    def __init__(self, x, y):
        sprite.Sprite.__init__(self)
        self.is_near = False  # 自機に近いか
        self.image1 = transform.scale(IMAGES['bullet2'], (10, 10))  # 敵の弾の画像
        self.image2 = transform.scale(IMAGES['bullet3'], (10, 10)) #近い敵の弾の画像
        self.image=self.image1
        self.rect = self.image.get_rect(center=(x, y))
        self.speed = 5

    def update(self):
        self.rect.y += self.speed
        if self.is_near:#近い場合画像を差し替える
            self.image=self.image2
        else:
            self.image=self.image1
        if self.rect.top > 520:
            self.kill()
        game.screen.blit(self.image, self.rect)
# 敵の拡散弾クラス
class EnemySpreadBullet(sprite.Sprite):
    def __init__(self, x, y, angle):
        sprite.Sprite.__init__(self)
        self.is_near = False
        self.image1 = transform.scale(IMAGES['bullet2'], (10, 10))
        self.image2 = transform.scale(IMAGES['bullet3'], (10, 10))
        self.image=self.image1
```

```
            self.rect = self.image.get_rect(center=(x, y))
            self.angle = angle
            self.speed = 7



        def update(self):
            self.rect.x += self.speed * math.sin(math.radians(self.angle))
            self.rect.y += self.speed * math.cos(math.radians(self.angle))
            if self.is_near:
                self.image=self.image2
            else:
                self.image=self.image1
            if self.rect.top > 520 or self.rect.left < 0 or self.rect.right > 546:
                self.kill()
            game.screen.blit(self.image, self.rect)

    class EnemySnipeBullet(sprite.Sprite):
        def __init__(self, x, y, target):
            sprite.Sprite.__init__(self)
            self.is_near = False
            self.image1 = transform.scale(IMAGES['bullet2'], (10, 10))
            self.image2 = transform.scale(IMAGES['bullet3'], (10, 10))
            self.image=self.image1
            self.rect = self.image.get_rect(center=(x, y))
            self.speed = 5
            self.target = target   # ターゲット（自機）
            target_x=target.rect.x
            target_y=target.rect.y
            # 目標への方向ベクトルを計算
            dx = target_x - x
            dy = target_y - y
            distance = math.sqrt(dx**2 + dy**2)
            if distance != 0:
                self.direction_x = dx / distance
                self.direction_y = dy / distance
            else:
                self.direction_x = 0
                self.direction_y = 1
```

```
    def update(self):
        #自機の方向に設定
        self.rect.x += self.direction_x * self.speed
        self.rect.y += self.direction_y * self.speed
        if self.is_near:
            self.image=self.image2
        else:
            self.image=self.image1


        if self.rect.top > 520 or self.rect.left < 0 or self.rect.right > 546:
            self.kill()
        game.screen.blit(self.image, self.rect)


class Game():
    def __init__(self):
        init()
        self.clock = time.Clock()#フレーム制限用
        self.ship = Ship()
        self.enemy3 = Enemy3()
        self.screen = SCREEN
        self.screen.fill(WHITE)
        self.font = font.Font(None, 24)# フォントの初期化
        # スコアと行動を表示する変数
        self.current_action = "NONE"  # 現在の行動
        self.current_score = 0             # 現在のスコア

        # 弾の種類を管理
        self.bullet_type = "Normal"
        self.bullet_usage_time = {"Normal": 0, "Speed": 0, "Spread": 0}
        self.bullet_kill_count = {"Normal": {}, "Speed": {}, "Spread": {}}  # 敵の種類ごと

        # 弾命中と弾破壊を弾種ごとに管理
        self.hits_within_1_seconds = {"Normal": 0, "Speed": 0, "Spread": 0}
        self.nearby_bullets_destroyed = {"Normal": 0, "Speed": 0, "Spread": 0}
        self.hits_start_time = time.get_ticks()
```

```
        # スプライトグループ
        self.all_sprites = sprite.Group()
        self.bullets = sprite.Group()  # 自機の弾
        self.enemy_bullets = sprite.Group()  # 敵の弾
        self.enemies = sprite.Group()#敵

        # 各グループに追加
        self.all_sprites.add(self.ship)
        self.all_sprites.add(self.enemy3)

        # 敵の生成を管理
        self.spawn_time = time.get_ticks()  # 最後に敵を生成した時間
        self.spawn_interval = randint(1000, 3000)  # ランダムな生成間隔
        self.remaining_enemies = self.create_enemy_positions()  # 敵のポジションリスト

        self.reset_episode_stats()#状態初期化
        self.time_in_most_dense = 0  # 敵が多い領域で過ごした時間
        self.time_in_least_dense = 0  # 敵が少ない領域で過ごした時間
        self.episode_start_time = time.get_ticks()#エピソード開始時間
        self.time_per_episode = []  # 各エピソードごとの時間記録

    #状態取得
    def get_game_state(self, ship,enemies, enemy_bullets,enemy3,bullets,current_score):
        #自機
        ship_x = ship.rect.x / SCREEN_WIDTH #正規化
        ship_y = ship.rect.y / SCREEN_HEIGHT #正規化
        ship_life = ship.life /3
        bullet_type = self.encode_bullet_type(ship.bullet_type)#弾の種類を数値で取得

        #敵
        enemies_state = []
        #近い順に並び替え
        nearest_enemys = sorted(
            enemies, key=lambda enemy: ((enemy.rect.x - ship.rect.x) ** 2 + (enemy.rect.y -
    ship.rect.y) ** 2) ** 0.5
        )
        for nearest_enemy in nearest_enemys[:5]:#敵が近い順で 5 体
            enemy_x = nearest_enemy.rect.x / SCREEN_WIDTH  # 正規化
            enemy_y = nearest_enemy.rect.y / SCREEN_HEIGHT  # 正規化
```

```
            enemy_type = self.encode_enemy_bullet_type(type(nearest_enemy).__name__)#敵の種類を
取得
            enemies_state.extend((enemy_x, enemy_y,enemy_type))

        # 敵が 5 体未満の場合は追加
        while len(enemies_state) < 5*3:
            enemies_state.extend((0, 0,   -1))

        enemy3_x = enemy3.rect.x /SCREEN_WIDTH #正規化
        enemy3_y = enemy3.rect.y / SCREEN_HEIGHT #正規化

        #自機の弾
        bullet_positions = []

        if bullets:#最も近い弾を取得
            nearest_bullet = min(
                bullets, key=lambda bullet: ((bullet.rect.x - ship.rect.x) ** 2 + (bullet.rect.y -
ship.rect.y) ** 2) ** 0.5
            )
            bullet_x = nearest_bullet.rect.x / SCREEN_WIDTH   # 正規化
            bullet_y = nearest_bullet.rect.y / SCREEN_HEIGHT   # 正規化
            bullet_speed = nearest_bullet.speed
            bullet_positions.extend((bullet_x, bullet_y, bullet_speed))
        else:
            bullet_positions.extend((0, 0, 0))   # 弾がない場は追加

        #敵の弾
        nearby_enemy_bullet_positions = []
        sorted_enemy_bullets = sorted(
            enemy_bullets, key=lambda bullet: ((bullet.rect.x - ship.rect.x) ** 2 + (bullet.rect.y -
ship.rect.y) ** 2) ** 0.5
        )
        for enemy_bullet in sorted_enemy_bullets[:5]:
            enemy_bullet_x = enemy_bullet.rect.x / SCREEN_WIDTH   # 正規化
            enemy_bullet_y = enemy_bullet.rect.y / SCREEN_HEIGHT   # 正規化
            enemy_bullet_speed = enemy_bullet.speed
            enemy_bullet_type = self.encode_enemy_bullet_type(type(enemy_bullet).__name__)   # 弾
の種類
```

```
            nearby_enemy_bullet_positions.extend((enemy_bullet_x, enemy_bullet_y,
enemy_bullet_speed, enemy_bullet_type))

        while len(nearby_enemy_bullet_positions) < 5*4:
            nearby_enemy_bullet_positions.extend((0, 0, 0, -1))#ない場合は追加
        #スコア
        score=current_score/100000#とるであろう最大値で正規化



        state =
np.array([ship_x,ship_y,ship_life,bullet_type,*enemies_state,*nearby_enemy_bullet_positions,ene
my3_x,enemy3_y,*bullet_positions,score])

        return state

#弾を数字で識別
def encode_bullet_type(self, bullet_type):
            if bullet_type == "Normal":
                return 0
            elif bullet_type == "Speed":
                return 1
            elif bullet_type == "Spread":
                return 2
            else:
                return -1   #  不明なタイプ
#敵を数字で識別
def encode_enemy_type(self, enemy_type):
            """敵の種類を数値にエンコード"""
            if enemy_type == "Enemy1":
                return 0
            elif enemy_type == "Enemy2":
                return 1
            elif enemy_type == "Enemy3":
                return 2
            else:
                return -1   #  不明なタイプ

#敵の弾を数字で識別
```

```
def encode_enemy_bullet_type(self, enemy_bullet_type):
        """敵の種類を数値にエンコード"""
        if enemy_bullet_type == "EnemyBullet":
            return 0
        elif enemy_bullet_type == "EnemySpreadBullet":
            return 1
        elif enemy_bullet_type == "EnemySnipeBullet":
            return 2
        else:
            return -1  # 不明なタイプ
#状態サイズ取得
def get_state_size(self):

    # 自機の状態次元: (x 座標, y 座標, ライフ, 弾タイプ)
    ship_dimension = 1 + 1 + 1 + 1

    # 敵の状態次元: (x 座標, y 座標, 種類) × 5(近い敵 5 体)
    enemy_dimension = 5 * (1 + 1 + 1)

    # 特定の敵 (enemy3) の状態次元: (x 座標, y 座標)
    enemy3_dimension = 1 + 1

    # 自機の弾の状態次元: (x 座標, y 座標, 速度) × 1（最も近い弾のみ）
    bullet_dimension = 1 + 1 + 1

    # 敵の弾の状態次元: (x 座標, y 座標, 速度, 種類) × 5（近い弾 5 個）
    enemy_bullet_dimension = 5 * (1 + 1 + 1 + 1)

    # 総次元数
    total_dimension = (
        ship_dimension
        + enemy_dimension
        + enemy3_dimension
        + bullet_dimension
        + enemy_bullet_dimension
        +1#スコア
    )

    return total_dimension
```

```
#報酬計算
def update_and_calculate_reward(self):

    reward = 0

    # 背景をリセット
    self.screen.fill(WHITE)

    # 敵のリスポーンをチェック
    for enemy in self.enemies:
        if not enemy.visible:
            enemy.check_respawn()



    # 敵の生成
    current_time = time.get_ticks()
    if current_time - self.spawn_time > self.spawn_interval:
        self.spawn_enemy()
        self.spawn_time = current_time#最新の敵生成時間として記録
        self.spawn_interval = randint(1000, 3000)

    #敵の弾が近いか判定
    for enemy_bullet in self.enemy_bullets:
        distance = math.sqrt((enemy_bullet.rect.x - self.ship.rect.x) ** 2 +
                             (enemy_bullet.rect.y - self.ship.rect.y) ** 2)

        # 自機との距離が指定範囲内の場合
        if distance <= 50:
            enemy_bullet.is_near = True
        else:
            enemy_bullet.is_near = False

    # 壁にぶつかる行動の場合は負の報酬を与える
    reward += self.check_boundary(action)



    #敵や敵の弾がいるエリア表示
    reward+=self.grid_screen(
                self.ship,
```

```
                self.enemies,
                self.enemy3,
                self.enemy_bullets,
                SCREEN_WIDTH,
                SCREEN_HEIGHT,
                )



    for bullet in self.bullets:
        # 自機の弾が敵に当たった場合
        hit_enemies = sprite.spritecollide(bullet, self.enemies, False)#衝突判定(スプライトグループ)
        for enemy in hit_enemies:
            self.log_hit_within_1_seconds(self.ship.bullet_type)#1 秒以内に敵や敵の弾に攻撃したか
            enemy.take_damage(bullet.attack_power)#敵にダメージ
            bullet.kill()#弾の描画削除
            #敵を倒した場合の報酬
            if enemy.HP <= 0:
                current_time = time.get_ticks()
                #生成されて 3 秒以内の場合+10
                if current_time - enemy.spawn_time < 3000:
                    reward+=10
                reward += 10 if isinstance(enemy, Enemy1) else 30#敵 1 の場合:+10,敵 2 の場合:+30
                self.log_kill(self.ship.bullet_type, type(enemy).__name__ )
            #敵に当てた場合
            else:

                reward += 2 if isinstance(enemy, Enemy1) else 6#敵 1 の場合:+2,敵 2 の場合:+6

        # 自機の弾が敵 3 に当たった場合
        if sprite.collide_rect(bullet, self.enemy3):#衝突判定
            self.log_hit_within_1_seconds(self.ship.bullet_type)
            self.enemy3.take_damage(bullet.attack_power)
            bullet.kill()
            if self.enemy3.HP <= 0:
                reward += 50
```

```
                self.log_kill(self.ship.bullet_type, "Enemy3")

                self.enemy3.kill()
            else:

                reward += 10


        # 自機の弾が敵弾を撃ち落とした場合
        hit_bullets = sprite.spritecollide(bullet, self.enemy_bullets, True)
        for hit_bullet in hit_bullets:
            self.log_hit_within_1_seconds(self.ship.bullet_type)
            if hit_bullet.is_near:
                self.log_nearby_bullet_destruction(self.ship.bullet_type)#近くの弾の場合

                reward +=5


            bullet.kill()


        #弾が何にも当たらず画面外になった場合:-1
        if bullet.rect.y < 0 or bullet.rect.x <0 or bullet.rect.x >546:
            reward -= 1




    # 自機が敵弾に当たった場合
    self.last_hit_time = getattr(self, "last_hit_time", time.get_ticks())#一度も記録してない場合は現在の時間を記録
    if sprite.spritecollide(self.ship, self.enemy_bullets, True) and self.ship.invincible==False:
        self.ship.take_damage()
        self.last_hit_time = time.get_ticks()  # 被弾時刻を記録
        if self.ship.life!=0:#残機がある場合
            reward -= 20
```

```
        else:
            reward -= 60

# 一定時間被弾していない場合の報酬
current_time = time.get_ticks()
time_since_last_hit = current_time - self.last_hit_time
if time_since_last_hit > 3000：  # 3 秒以上被弾していない場合
        reward += 1   # 長時間生存ボーナス
        self.last_hit_time=current_time

# 自機と敵が衝突した場合
for enemy in self.enemies:
        if sprite.collide_rect(self.ship, enemy)and self.ship.invincible==False:
            self.ship.take_damage()
            enemy.hide(3000)
            if self.ship.life!=0:
                reward -= 20

            else:
                reward -= 60


if sprite.collide_rect(self.ship, self.enemy3)and self.ship.invincible==False:
            self.ship.take_damage()
            if self.ship.life!=0:
                reward -= 20

            else:
                reward -= 60



#近くに敵の弾がある場合:-1
for bullet in self.enemy_bullets:

        if bullet.is_near:
            reward -=1
```

```
        # スプライトの更新と描画
        self.all_sprites.update()
        self.all_sprites.draw(self.screen)

        # 弾の種類を表示
        self.display_bullet_type()
        # 行動選択とスコアを画面に描画
        self.display_action_and_score()

        return reward
def grid_screen(self,ship, enemies, enemy3,bullets, screen_width, screen_height, grid_size=6):


        # グリッド幅と高さの計算
        grid_width = screen_width / grid_size
        grid_height = screen_height/grid_size

        # 画面を分割したマップ
        density_grid = np.zeros((grid_size, grid_size))

        # 敵の位置をマップに追加
        for enemy in enemies:
            enemy_x, enemy_y = enemy.rect.x, enemy.rect.y
            grid_x = int(enemy_x // grid_width)
            grid_y = int(enemy_y // grid_height)
            density_grid[grid_y, grid_x] += 1

        enemy3_x, enemy3_y = enemy3.rect.x, enemy3.rect.y
        grid_x = int(enemy3_x // grid_width)
        grid_y = int(enemy3_y // grid_height)
        density_grid[grid_y, grid_x] += 1
```

```
        # 弾の位置をマップに追加
        for bullet in bullets:
            bullet_x, bullet_y = bullet.rect.x, bullet.rect.y
            grid_x = min(max(int(bullet_x // grid_width), 0), grid_size - 1)
            grid_y = min(max(int(bullet_y // grid_height), 0), grid_size - 1)
            density_grid[grid_y, grid_x] += 1

        # 自機がいるグリッドを計算
        ship_x, ship_y = ship.rect.x, ship.rect.y
        ship_grid_x = min(max(int(ship_x // grid_width), 0), grid_size - 1)
        ship_grid_y = min(max(int(ship_y // grid_height), 0), grid_size - 1)

        # 現在の自機のエリア
        current_density = density_grid[ship_grid_y, ship_grid_x]

        # 最も多いエリアと最も少ないエリアの計算
        most_dense = np.max(density_grid)
        least_dense = np.min(density_grid)

        # 時間計測
        current_time = time.get_ticks()
        if current_density == most_dense:#最も多いエリアにいる時
            self.time_in_most_dense += current_time - self.episode_start_time
        elif current_density == least_dense:#最も少ないエリアにいる時
            self.time_in_least_dense += current_time - self.episode_start_time

        self.episode_start_time = current_time

        # 画面を分割して表示
        self.display_density_grid(density_grid, grid_size)

        return 1.0 if current_density == least_dense else -1.0#最も空かないところにいる場合:+1



    #自機が画面枠を越えようとするとき負の報酬を与える
    def check_boundary(self, action):

        negative_reward = 0
```

```
        # 現在の位置を取得
        x, y = self.ship.rect.x, self.ship.rect.y
        width, height = SCREEN_WIDTH, SCREEN_HEIGHT

        # 左端で左移動
        if action == 0 and x <= 0:
            negative_reward = -1
        # 右端で右移動
        elif action == 1 and x + self.ship.rect.width >= width:
            negative_reward = -1
        # 上端で上移動
        elif action == 2 and y <= 0:
            negative_reward = -1
        # 下端で下移動
        elif action == 3 and y + self.ship.rect.height >= height:
            negative_reward = -1

        return negative_reward

    #ゲームの初期状態に戻す
    def reset_game(self):

        print("Game Reset!")
        self.ship = Ship()
        self.all_sprites.empty()
        self.bullets.empty()
        self.enemy_bullets.empty()
        self.enemies.empty()
        self.all_sprites.add(self.ship)
        self.enemy3 = Enemy3()
        self.all_sprites.add(self.enemy3)
        self.remaining_enemies = self.create_enemy_positions()
        self.spawn_time = time.get_ticks()

    #敵の生成位置リストを作成
    def create_enemy_positions(self):
        positions = []
        #敵 1 を最大 2*11 で作る
```

```
        for row in range(2):
            for i in range(11):
                x = 0 + (i * 52)
                y = 182 + (row * 52)
                positions.append(('enemy1', x, y))
        #敵 2 を最大 2*10 で作る
        for row in range(2):
            for i in range(10):
                x = 26 + (i * 52)
                y = 78 + (row * 52)
                positions.append(('enemy2', x, y))
        return positions
    #敵をランダムに生成
    def spawn_enemy(self):
        if self.remaining_enemies:
            enemy_type, x, y = choice(self.remaining_enemies)#敵の生成位置から一つ選択
            #生成位置の敵のタイプによる場合分け
            if enemy_type == 'enemy1':
                enemy = Enemy1(x, y)
            else:
                enemy = Enemy2(x, y)
            self.enemies.add(enemy)
            self.all_sprites.add(enemy)
            self.remaining_enemies.remove((enemy_type, x, y))

    #弾の種類を画面に表示
    def display_bullet_type(self):

        bullet_text = f"Bullet: {self.ship.bullet_type}"#出力文字
        bullet_text_surface = self.font.render(bullet_text, True, (0, 0, 0))#黒文字で描画
        self.screen.blit(bullet_text_surface, (10, 10))



    #行動とスコアを画面に表示
    def display_action_and_score(self):

        # 行動の表示
        action_text = f"Action: {self.current_action}"
```

```
        action_surface = self.font.render(action_text, True, (0, 0, 0))
        self.screen.blit(action_surface, (10, 30))

        # スコアの表示
        score_text = f"Score: {self.current_score}"
        score_surface = self.font.render(score_text, True, (0, 0, 0))
        self.screen.blit(score_surface, (10, 50))

#画面分割を表示
def display_density_grid(self, density_grid, grid_size):

        grid_width = SCREEN_WIDTH / grid_size
        grid_height = SCREEN_HEIGHT / grid_size
        self.ship_grid_y = self.ship.rect.y // grid_size
        self.ship_grid_x = self.ship.rect.x // grid_size
        for y in range(grid_size):
            for x in range(grid_size):
                rect = Rect(x * grid_width, y * grid_height, grid_width, grid_height)
                color = (255, 0, 0) if (y, x) == (self.ship_grid_y, self.ship_grid_x) else (0, 255, 0)#自機がいるところは赤,それ以外は緑
                draw.rect(self.screen, color, rect, 1)

                density_text = f"{int(density_grid[y, x])}"
                text_surface = self.font.render(density_text, True, (0, 0, 0))#敵などの数表示
                self.screen.blit(text_surface, (rect.x + 5, rect.y + 5))
#エピソードごとの記録をリセット
def reset_episode_stats(self):

        self.previous_reward = 0#現在の報酬
        self.time_in_most_dense = 0#最も多いエリアに自機がいる時間
        self.time_in_least_dense = 0#最も少ないエリアに自機がいる時間
        self.episode_start_time = time.get_ticks()
        self.bullet_usage_time = {"Normal": 0, "Speed": 0, "Spread": 0}#弾各種の使用時間
        self.bullet_kill_count = {"Normal": {}, "Speed": {}, "Spread": {}}#弾各種の倒した敵
        self.hits_within_1_seconds = {"Normal": 0, "Speed": 0, "Spread": 0}#弾各種の 1 秒間に当てた数
        self.nearby_bullets_destroyed = {"Normal": 0, "Speed": 0, "Spread": 0}#撃ち落とした近くの弾の数
        self.hits_start_time = time.get_ticks()#弾を最初に当てた時間
        self.current_bullet_start_time = time.get_ticks()#弾を使い始めた時間
```

```
#エピソード結果を記録
def log_episode_result(self, episode, scores):

    dense_time = (self.time_in_most_dense, self.time_in_least_dense)
    scores.append((episode, self.current_score, dense_time, dict(self.bullet_usage_time)))
    print(f"=== Episode {episode} Result ===")
    print(f"   Score: {self.current_score}")
    print(f"   Most Dense Time: {dense_time[0]} ms")
    print(f"   Least Dense Time: {dense_time[1]} ms")
    for bullet_type, time_used in self.bullet_usage_time.items():
        print(f"   {bullet_type} Usage: {time_used // 1000}s")

#弾各種の使用時間を記録
def update_bullet_usage_time(self,bullet_type):
    elapsed_time = time.get_ticks() - self.current_bullet_start_time

    self.bullet_usage_time[bullet_type] += elapsed_time
    self.current_bullet_start_time = time.get_ticks()

#1 秒以内に敵や敵の弾に当てた数を記録
def log_hit_within_1_seconds(self, bullet_type):
    reward=0
    current_time = time.get_ticks()
    if current_time - self.hits_start_time <= 1000:  # 1 秒以内
        if bullet_type == "Speed":#弾の種類がスピード弾の場合報酬発生
            reward+=5
        self.hits_within_1_seconds[bullet_type] += 1
    else:
        self.hits_start_time = current_time
        self.hits_within_1_seconds[bullet_type] = 1
    return reward
#撃ち落とした近くの弾の数を記録
def log_nearby_bullet_destruction(self, bullet_type):
    reward=0
    if bullet_type == "Spread":#弾の種類が拡散弾の場合報酬発生
        reward+=5
    self.nearby_bullets_destroyed[bullet_type] += 1
    return reward
```

```
#倒した敵を種類ごとに数を記録
def log_kill(self, bullet_type, enemy_type):
    reward=0
    if bullet_type == "Normal":#弾の種類が普通弾の時報酬発生
        reward+=5
    if enemy_type not in self.bullet_kill_count[bullet_type]:
        self.bullet_kill_count[bullet_type][enemy_type] = 0
    self.bullet_kill_count[bullet_type][enemy_type] += 1
    return reward

#指定したエピソードの結果を最後に表示
def log_final_results(self, scores, episodes_to_log=None):
    print("=== Final Results ===")
    for episode, score, dense_time, bullet_times in scores:
        if episodes_to_log is not None and episode not in episodes_to_log:
            continue
        print(f"Episode {episode}:")
        print(f"    Score: {score}")
        print(f"    Most Dense Time: {dense_time[0]} ms")
        print(f"    Least Dense Time: {dense_time[1]} ms")
        for bullet_type, time_used in bullet_times.items():
            print(f"    {bullet_type} Usage: {time_used // 1000}s")#秒で表示
        print()

#指定したエピソード数ごとに記録
def log_n_episode_results(self, scores, episodes):

    if len(scores) % episodes == 0:#指定したエピソードごとに記録
        print(f"¥n=== Results After {len(scores)} Episodes ===")
        total_hits = {"Normal": 0, "Speed": 0, "Spread": 0}
        total_nearby_destroys = {"Normal": 0, "Speed": 0, "Spread": 0}
        total_kills = {"Normal": {}, "Speed": {}, "Spread": {}}

        for score in scores[-episodes:]:#最後から指定ぶん取得
            for bullet_type in ["Normal", "Speed", "Spread"]:
                total_hits[bullet_type] += score["hits_within_1_seconds"][bullet_type]
                total_nearby_destroys[bullet_type] +=
score["nearby_bullets_destroyed"][bullet_type]#指定した分の記録を合計
```

```
                    for enemy_type, count in score["bullet_kill_count"][bullet_type].items():
                        if enemy_type not in total_kills[bullet_type]:
                            total_kills[bullet_type][enemy_type] = 0#なかった場合は 0 を追加
                        total_kills[bullet_type][enemy_type] += count

            print(f"    Hits within 1 seconds by bullet type:")
            for bullet_type, count in total_hits.items():
                print(f"       {bullet_type}: {count}")

            print(f"    Nearby bullets destroyed by bullet type:")
            for bullet_type, count in total_nearby_destroys.items():
                print(f"       {bullet_type}: {count}")

            print("    Total kills by bullet type:")
            for bullet_type, kills in total_kills.items():
                print(f"       {bullet_type}: {kills}")
            print()

    # 全エピソードのスコアを羅列表示
    def log_final_scores(self, scores):

        result=0

        print("=== All Episode Scores ===")
        for score in scores:
            result=score[1]
            print(f"{result}")

    #終了
    def end(self):
        for e in event.get():
            if e.type == QUIT:
                quit()
                sys.exit()



if __name__ == "__main__":
```

```
scores = []
scores_episode = []
bullet_usage_time = {"NORMAL": 0, "SPEED": 0, "SPREAD": 0}
episodes=10000#エピソード数
current_bullet_start_time = time.get_ticks()  # 現在の弾の使用開始時刻

# ゲームインスタンスを作成
game = Game()
# 環境の状態次元と行動数
state_size = game.get_state_size()  # 状態の次元数を取得する関数
action_size = len(["LEFT", "RIGHT", "UP", "DOWN", "SHOOT", "NORMAL", "SPEED",
"SPREAD"])#行動次元数
agent = DQN(state_size, action_size)




# 学習ループ
for episode in range(episodes):
    # ゲームの初期化
    total_reward = 0
    game.reset_episode_stats()
    # ゲームの状態を取得
    state = game.get_game_state(game.ship, game.enemies, game.enemy_bullets, game.enemy3,
game.bullets,game.current_score)

    # NumPy 配列に変換し,さらに Torch の Tensor に変換
    state = torch.FloatTensor(state).unsqueeze(0)

    while True:
        # 行動を選択
        action = agent.choose_action(state)
        game.current_action = action  # 現在の行動を記録



        # 行動を実行
```

```
 if action == 0:#左移動
        game.ship.rect.x -= game.ship.speed
 elif action == 1:#右移動
        game.ship.rect.x += game.ship.speed
 elif action == 2:#上移動
        game.ship.rect.y -= game.ship.speed
 elif action == 3:#下移動
        game.ship.rect.y += game.ship.speed

 elif action == 4:#  射撃

        game.ship.shoot()

 elif action == 5:#通常弾に変更
        if game.ship.bullet_type != "Normal":
                game.ship.bullet_type = "Normal"#弾種変更
                game.update_bullet_usage_time(game.ship.bullet_type)

 elif action == 6:#スピード弾に変更
        if game.ship.bullet_type != "Speed":
                game.ship.bullet_type = "Speed"
                game.update_bullet_usage_time(game.ship.bullet_type)



 elif action == 7:#拡散弾に変更
        if game.ship.bullet_type != "Spread":
                game.ship.bullet_type = "Spread"
                game.update_bullet_usage_time(game.ship.bullet_type)


# ゲーム状態の更新と報酬の取得
 reward = game.update_and_calculate_reward()
 total_reward += reward
 game.current_score = total_reward   # スコアを更新
 done = game.ship.life <= 0   # ゲーム終了条件
 next_state =
                        game.get_game_state(game.ship,game.enemies,game.enemy_bullets,g
        ame.
```

```
                    enemy3,game.bullets,game.current_score)#次の状態を設定

    # 状態がタプルである場合,リストに変換し,数値だけを抽出
    next_state = [float(x) if isinstance(x, (int, float)) else 0.0 for x in next_state]
    # Q 値を記録
    agent.remember(state, action, reward, next_state, done)

    # 状態を更新
    state = next_state

    # 学習を行う
    agent.train()




    # 描画
    display.update()

    # 終了判定
    if done:
        game.log_episode_result(episode + 1, scores)
        episode_result = {
            "score": game.current_score,
            "hits_within_1_seconds": game.hits_within_1_seconds.copy(),
            "nearby_bullets_destroyed": game.nearby_bullets_destroyed.copy(),
            "bullet_kill_count": game.bullet_kill_count.copy(),
        }

        scores_episode.append(episode_result)
        game.log_n_episode_results(scores_episode, 1)

        game.reset_game()
        break

    game.clock.tick(30)
    # ターゲットネットワークの更新
if episode % agent.target_update == 0:
    agent.update_target_network()
```

```
        # 探索率の減少
        agent.epsilon = max(agent.epsilon_min, agent.epsilon * agent.epsilon_decay)
print("¥n--- 全エピソードのスコア ---")
game.log_final_results(scores,episodes_to_log=[1,1000,2000,3000,4000,
5000,6000,7000,8000,9000,10000])
game.log_n_episode_results(scores_episode, episodes)
game.log_final_scores(scores)
```