****

**Politechnika Świętokrzyska**

**Systemy rozpoznawania mowy i obrazu (projekt)**

Bartłomiej Bugara, Szymon Kołodziejczyk

4ID13A

**„Mario AI”**

**Klon gry Mario z siecią neuronową**

**Dokumentacja projektu**



1. **Opis projektu**

Tematem naszego projektu jest klon popularnej gry Super Mario Bros. wraz z wbudowanym edytorem map i systemem sztucznej inteligencji, który ma zdolność uczenia się, jak grać w tę grę. Głównym celem naszego projektu jest opracowanie efektywnych algorytmów uczenia maszynowego, które umożliwią agentowi osiągnięcie optymalnej zdolności do przechodzenia poziomów w grze.

**Opis gry:**

„Super Mario Bros.” to klasyczna platformówka polegająca na przemierzaniu poziomów, zbieraniu monet i omijaniu przeszkód, aby uratować księżniczkę Peach przed złym Bowserem. Jednakże nasz projekt stanowi jedynie rekreację tej gry, umożliwiając graczowi tworzenie i modyfikowanie poziomów oraz pozwalając na naukę agentowi AI, jak z powodzeniem je przechodzić.

**Funkcje projektu:**

Prócz klona gry nasza aplikacja zawiera interaktywny edytor map dzięki któremu możemy tworzyć oraz modyfikować poziomy. Edytor pozwala na ustawianie podstawowych bloków, pieniążków, a także przeciwników.

Najważniejszą funkcjonalnością projektu jest wielowarstwowa sieć neuronowa która ma za zadanie nauczyć się grać i osiągać jak najlepsze wyniki.

1. **Użyte technologie**

Projekt napisaliśmy wykorzystując Microsoft Visual Studio 2022 wraz z kompilatorem MSVC. Za obsługę grafiki posłużyła nam biblioteka Raylib zainstalowana do projektu poprzez menedżer pakietów NuGet.

Użyte oprogramowanie:

• GitHub Desktop – program do obsługi systemu kontroli wersji Git, dzięki któremu ułatwiona jest wspólna praca nad projektem przez kilka osób

jednocześnie

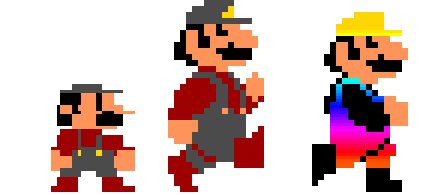
• Inkscape – program do tworzenia grafiki wektorowej, przy jego pomocy stworzyliśmy logo zawarte na pierwszej stronie oraz ikonkę

• GIMP – program do tworzenia grafiki rastrowej, przy jego pomocy wykonano tło oraz grafiki pixelart występujące w grze

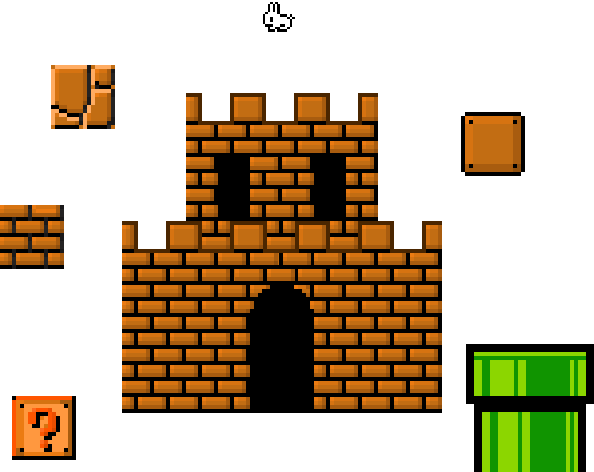
1. **Opis algorytmów**

Gra – algorytmy gry zostały odwzorowane względem oryginału.

1. Gracz:
   * Mały – zwykła postać Cześka, oznacza że mamy jedno życie
   * Duży – większa tekstura Czesia, 2 HP, aktywuje się po zebraniu kwiatka
   * Ekstra – strzelający ogniem Czesiek, po zebraniu kwiatka



1. Bloki:
   * Cegła – blok cegieł który się niszczy i pozostawia particle
   * Kamień – zwykły niezniszczalny blok
   * Stal – podobnie jak kamień tylko inna tekstura
   * Zamek – dekoracja końcowa planszy
   * Flaga – zaznacza końcowy etap poziomu
   * Blok z nagrodą – pozwala na spawn monet, grzybów oraz kwiatów
   * Rura – zwykła przeszkoda utrudniająca przejście



1. Ulepszenia:
   * Pieniążek – występuje na planszy lub wyskakuje z powyżej opisanego bloku. Dodaje nam 200 punktów, posiada animację.
   * Grzybek – po wyskoczeniu porusza się w lewo i prawo, w czasie kolizji zmienia kierunek ruchu, dzięki niemu gracz ma dodatkowe życie – zmienia teksturę z małego Cześka na dużego.
   * Kwiatek – pozwala graczowi na strzelanie ogniem



1. Przeciwnicy: Goomba i Coopa, pierwszy z nich jest grzybkiem poruszającym się w obie strony, wejście w niego powoduje utratę życia, posiada animację przy pomocy sprite’ów, aby go pokonać wskakujemy na niego, po chwili znika. Drugi zaś ma dwa kolory, czerwony oraz zielony, zielony na krawędzi spada, czerwony wraca. Podobnie jak Goomba posiada animację, główną różnicą jest zmiana Coopy w skorupę po wskoczeniu na niego.



Sieć neuronowa:

Nasza sieć została zawarta w klasie NN, którą następnie wykorzystujemy do uczenia gry.

|  |
| --- |
| #include "NN.h"  NN::NN()  {  for (int i = 0; i < inputsSize; i++)  {  Node n;  n.ID = nodes.size();  n.type = NodeType::input;  n.value = -1;  n.x = (i % inputsSizeW) \* cellSize;  n.y = (i / inputsSizeW) \* cellSize;  nodes.push\_back(n);  }  {  Node n;  n.ID = nodes.size();  n.type = NodeType::input;  n.value = 1;  n.x = ((inputsSize - 1) % inputsSizeW) \* cellSize;  n.y = (inputsSize / inputsSizeW + 1) \* cellSize;  nodes.push\_back(n);  }  for (int i = 0; i < outputsSize; i++)  {  Node n;  n.ID = nodes.size();  n.type = NodeType::output;  n.value = 0;  n.x = 96 \* cellSize;  n.y = i \* cellSize\*2;  nodes.push\_back(n);  }  }  NN::NN(nlohmann::json& j):NN()  {  for (int i = 0; i < j["N"].size(); i++)  {  Node n;  n.readFromFile(j["N"][i]);  nodes.push\_back(n);  }  for (int i = 0; i < j["C"].size(); i++)  {  Connection c;  c.readFromFile(j["C"][i]);  connections.push\_back(c);  }  }  void NN::addConnection()  {  int n = inputsSize + 1;  int hidenStart = inputsSize + outputsSize + 1;  int value = rand() % (nodes.size() - outputsSize);  Node from;  if (value < inputsSize + 1)  {  from = nodes[value];  }  else  {  from = nodes[value + outputsSize];  }  Node to = findNodeToConect(from);  Connection c;  c.from = from.ID;  c.to = to.ID;  c.w = (rand() % 20000) / 10000.0f - 1.0f;  c.active = true;  if(!hasConnection(c))  connections.push\_back(c);  }  struct NodesID {  int ID1, ID2;  };  int getNewID(int lastID, std::vector<NodesID> newIDs)  {  for (int i = 0; i < newIDs.size(); i++)  if (lastID == newIDs[i].ID2)  return newIDs[i].ID1;  return lastID;  }  bool randomSwapConnetions(Connection c, std::vector<Connection>& connections)  {  for(int i=0;i< connections.size();i++)  if (c.compare(connections[i]))  {  if (c.active == connections[i].active)  {  bool swap = rand() % 2;  if (swap)  connections[i] = c;  }  else  {  if (!c.active)  connections[i] = c;  }  return true;  }  return false;  }  NN\* NN::combineNNs(NN \*n)  {  NN\* toRet=new NN();  std::vector<NodesID> doubleNodes = std::vector<NodesID>();  std::vector<Node> nnnodes = std::vector<Node>();  ///Dodanie wszystkich Nodów z pierwszej sieci neuronowej  for (int i = inputsSize + 1 + outputsSize; i < nodes.size(); i++)  {  nnnodes.push\_back(nodes[i]);  }  ///Dodanie wszystkich Nodów z drugiej sieci neuronowej  for (int i = inputsSize + 1 + outputsSize; i < n->nodes.size(); i++)  {  nnnodes.push\_back(n->nodes[i]);  }  ///Usuniecie duplikatów  for (int i = 0; i < nnnodes.size(); i++)  {  for (int j = nnnodes.size() - 1; j > i; j--)  {  if (nnnodes[i].compare(nnnodes[j]))  {  NodesID s;  s.ID1 = nnnodes[i].ID;  s.ID2 = nnnodes[j].ID;  doubleNodes.push\_back(s);  nnnodes.erase(nnnodes.begin()+j);  }  }  }  ///Ustalenie ID Nodów dla nowej sieci  if (nnnodes.size() > 0)  {  int ID = toRet->nodes.size();  for (int i = 0; i < nnnodes.size(); i++)  {  if (nnnodes[i].ID != ID + i)  {  NodesID s;  s.ID1 = ID + i;  s.ID2 = nnnodes[i].ID;  doubleNodes.push\_back(s);  nnnodes[i].ID = ID + i;  }  toRet->nodes.push\_back(nnnodes[i]);  }  }  ///Dodanie połączeń z pierwszej sieci neuronowej  for (int i = 0; i < connections.size(); i++)  {  if (connections[i].from < toRet->nodes.size() && connections[i].to < toRet->nodes.size())  toRet->connections.push\_back(connections[i]);  }  ///Dodanie połączeń z drugiej sieci oraz usunięcie duplikatów  for (int i = 0; i < n->connections.size(); i++)  {  Connection con;  con.from = getNewID(n->connections[i].from, doubleNodes);  con.to = getNewID(n->connections[i].to, doubleNodes);  con.active = n->connections[i].active;  con.w = n->connections[i].w;  if (!randomSwapConnetions(con, toRet->connections) && con.from < toRet->nodes.size() && con.to < toRet->nodes.size())  toRet->connections.push\_back(con);  }  return toRet;  }  void NN::addNode()  {  int v = rand() % connections.size();  Connection c = connections[v];  connections[v].active = false;  Node from = nodes[c.from];  Node to = nodes[c.to];  Node newNode;  newNode.x = (from.x + to.x) / 2;  newNode.y = (from.y + to.y) / 2;  newNode.ID = nodes.size();  Connection newC1;  newC1.from = from.ID;  newC1.to = newNode.ID;  newC1.w = c.w;  newC1.active = true;  Connection newC2;  newC2.from = newNode.ID;  newC2.to = to.ID;  newC2.w = c.w;  newC2.active = true;  nodes.push\_back(newNode);  connections.push\_back(newC1);  connections.push\_back(newC2);  }  void NN::sortConnections()  {  std::vector<Node> nodes;  for (int i = inputsSize + 1 + outputsSize; i < nodes.size(); i++)  nodes.push\_back(this->nodes[i]);  for (int j = 0; j < nodes.size(); j++)  {  for (int i = 0; i < nodes.size() - 1; i++)  {  if (nodes[i].x > nodes[i + 1].x)  {  Node n = nodes[i];  nodes[i] = nodes[i + 1];  nodes[i + 1] = n;  }  }  }  for (int i = 0; i < outputsSize; i++)  nodes.push\_back(this->nodes[i + inputsSize + 1]);  int k = 0;  for (int j = 0; j < nodes.size(); j++)  {  int ID = nodes[j].ID;  for (int i = k; i < connections.size(); i++)  {  if (connections[i].to == ID)  {  if (k != i)  {  Connection c = connections[i];  connections[i] = connections[k];  connections[k] = c;  }  k++;  if (k >= connections.size())  return;  }  }  }  }  void NN::generateOutput()  {  for (int i = inputsSize + 1; i < nodes.size(); i++)  nodes[i].value = 0;  for (auto c : connections)  {  if(c.active)  nodes[c.to].value += nodes[c.from].value \* c.w;  }  }  void NN::changeValue()  {  int i = rand() % connections.size();  connections[i].w \*= ((rand() % 4001) / 1000) - 1;  }  Node NN::findNodeToConect(Node from)  {  std::vector<Node> nodesToConect;  for (int i = 0; i < outputsSize; i++)  {  nodesToConect.push\_back(nodes[i + inputsSize + 1]);  }  for (int i = inputsSize + 1 + outputsSize; i < nodes.size(); i++)  {  Node n = nodes[i];  if (n.x > from.x)  {  nodesToConect.push\_back(n);  }  }  int v = rand() % nodesToConect.size();  return nodesToConect[v];  }  void NN::mutate()  {  int mutationType = rand() % 200;  if (connections.size() <= 0 || mutationType <= 50)  {  addConnection();  }  else if(mutationType <= 100)  {  addNode();  sortConnections();  }  else  {  changeValue();  }  }  void NN::setInputs(int\* inputs)  {  for (int i = 0; i < inputsSize; i++)  {  nodes[i].value = inputs[i];  }  }  bool\* NN::getOutputs()  {  bool outputs[outputsSize];  for (int i = 0; i < outputsSize; i++)  {  outputs[i] = (nodes[i + inputsSize + 1].value > 0.0f);  }  return outputs;  }  #include "raylib.h"  Color getColorFromValue(float v)  {  if (v == 0)  return { 0,0,0,0 };  if (v == 1)  return { 255,255,255,255 };  if (v == -1)  return { 255,0,0,255 };  if (v == 2)  return { 0,255,0,255 };  if (v < 1 && v>-1)  return { (unsigned char)(255 \* (-v)),(unsigned char)(255 \* v),0,255 };  if (v >= 1)  return { 0,(255),0,255 };  if (v <= -1)  return { 255,0,0,255 };  return { 0,0,0,255 };  }  Color getColorLineFromValue(float v)  {  Color c;  c.r = 255 \* (v > 1 ? 0 : (v < -1) ? 1 : (2-v) / 2);  c.g = 255 \* (v > 1 ? 1 : (v < -1) ? 0 : v / 2);  c.b = 0;  c.a = 255;  return c;  }  void NN::draw(int x, int y)  {  int size = cellSize / 2;  for (auto n : nodes)  {  DrawRectangle(n.x + x, n.y + y, cellSize, cellSize, getColorFromValue(n.value));  DrawRectangleLines(n.x + x, n.y + y, cellSize, cellSize, BLACK);  }  for (auto c : connections)  {  if (!c.active)  continue;  Node form = nodes[c.from];  Node to = nodes[c.to];  DrawLine(form.x + size + x, form.y + size + y, to.x + size + x, to.y + size + y, getColorLineFromValue(c.w));  }  }  void NN::saveToFile(nlohmann::json &j)  {  int k = 0;  for (int i = inputsSize + 1 + outputsSize; i < nodes.size(); i++)  {  nodes[i].saveToFile(j["N"][k]);  k++;  }  k = 0;  for (auto c:connections)  {  c.saveToFile(j["C"][k]);  k++;  }  }  bool NN::hasConnection(Connection c)  {  for (auto con : connections)  if (c.compare(c))  return true;  return false;  } |

Opis głównych fragmentów powyższego kodu:

1. **Konstruktor NN::NN():**
   * Tworzy instancję sieci neuronowej.
   * Inicjuje węzły sieci jako wejścia i wyjścia z odpowiednimi parametrami, takimi jak typ, wartość, współrzędne itp.
2. **Konstruktor NN::NN(nlohmann::json& j):**
   * Konstruktor pomocniczy, który pozwala na tworzenie instancji NN na podstawie wcześniej zapisanej konfiguracji w formacie JSON.
   * Odtwarza węzły i połączenia na podstawie przekazanych danych.
3. **Metoda NN::addConnection():**
   * Dodaje nowe połączenie między węzłami w sieci.
   * Wybiera losowy węzeł źródłowy i węzeł docelowy do połączenia.
   * Losowo inicjuje wagę połączenia.
4. **Struktura NodesID i funkcja getNewID():**
   * Struktura NodesID przechowuje informacje o przypisanym ID węzła.
   * Funkcja getNewID() znajduje nowe ID węzła w przypadku duplikatów.
5. **Funkcje randomSwapConnections() i NN::combineNNs(NN n):**
   * Funkcja randomSwapConnections() losowo zamienia połączenia w sieci.
   * Funkcja NN::combineNNs() łączy dwie sieci neuronowe, usuwając duplikaty węzłów i połączeń.
6. **Metoda NN::addNode():**
   * Dodaje nowy węzeł pośredni w sieci między istniejącymi węzłami.
   * Tworzy nowe połączenia między węzłami w oparciu o nowy węzeł.
7. **Metoda NN::sortConnections():**
   * Sortuje połączenia w sieci na podstawie położenia węzłów, ułatwiając analizę i wizualizację.
8. **Metoda NN::generateOutput():**
   * Oblicza wyniki wyjściowe sieci neuronowej na podstawie wartości węzłów i wag połączeń.
9. **Metoda NN::changeValue():**
   * Zmienia losowo wartości wag połączeń w sieci.
10. **Metoda NN::findNodeToConect(Node from):**
    * Znajduje węzeł docelowy do nowego połączenia na podstawie węzła źródłowego.
11. **Metoda NN::mutate():**
    * Wykonuje mutacje na sieci neuronowej, co może polegać na dodawaniu połączeń, węzłów lub zmianie wartości wag połączeń.
12. *Metoda NN::setInputs(int inputs) i bool* NN::getOutputs():\*\*
    * Metoda setInputs() ustawia wartości wejściowe sieci neuronowej na podstawie tablicy wejść.
    * Metoda getOutputs() zwraca wyniki wyjściowe sieci w postaci tablicy wartości logicznych.
13. **Funkcje getColorFromValue() i getColorLineFromValue():**
    * Funkcje pomocnicze do przypisywania kolorów wizualizacji węzłów i połączeń na podstawie wartości.
14. **Metoda NN::draw(int x, int y):**
    * Rysuje wizualizację sieci neuronowej i połączeń w określonych współrzędnych x i y.

1. **Wykorzystanie sieci do nauki:**

Podczas uruchomienia gry tworzona jest generacja, każda z nich posiada 200 gier w czasie których sieć mutuje, krzyżuje i wybiera najlepsze wyniki w celu ewolucji swojej strategii.

Każda gra posiada swoją miarę oceny – fitness. Określa ona jak daleko agent gry był w stanie dojść w zakresie od 128 – co oznacza początek poziomu do 999999 co oznacza równe 100% przejścia. Prawidłowy zakres kończy się w okolicach 14-15 tysięcy. Warunkiem zakończenia nauki jest utrata życia lub zatrzymanie się AI w miejscu na czas powyżej 2 sekund.

W czasie nauki sieci wykonywaliśmy kilkaset generacji, pozwalających nam na dogodne eksperymenty. Wejściami naszego AI są kratki mapy, na których kolorem białym oznaczamy normalne obiekty, czerwonym takie które nas zabijają (przeciwnicy, lawa), zaś zielonym obiekty które są dodatkowym atutem dla ulepszenia gry – pieniądze, grzyby itp. Wyjściami są oczywiście funkcje odpowiadające za sterowanie graczem.

Początkowo ilość inputów wynosiła 11 na 17 kratek czyli 187 wejść, zaś wyjść była piątka, chodzenie lewo, prawo, skok, kucanie i strzał(rzadko używany nawet przez gracza).

Przy takiej ilości wejść sieć uczyła się dość długo, rzadko docierając w dalsze zakątki mapy, nie mówiąc o wygraniu gry. Po rozmowie z prowadzącym zaniżyliśmy ich ilość do 11x11 czyli razem 121 inputów. Kolejnym krokiem mającym na celu ulepszenie nauki było ograniczenie ilości wyjść jedynie do skoku oraz ruchu w prawo, jest to wystarczające w celu pokonania poziomu. Na tym etapie w okolicach 20-30 generacji AI potrafiło samoczynnie przejść większość gier.

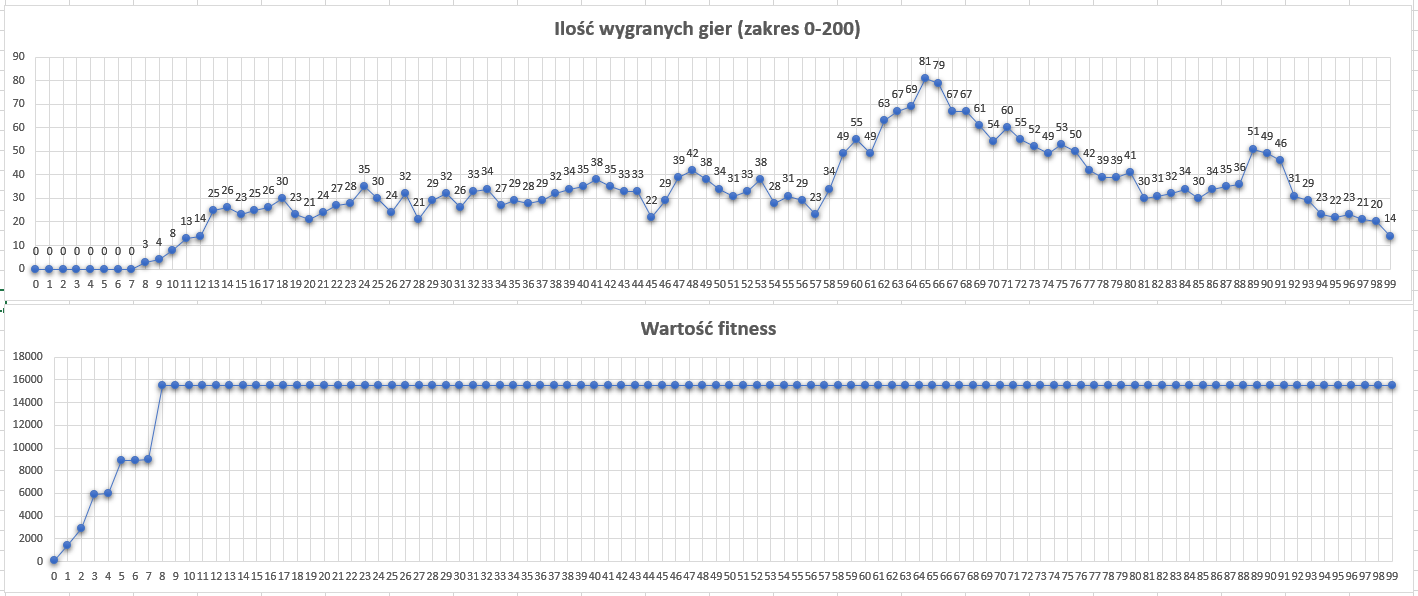
Rozmiar grid’a, ilość wejść i wyjść zawarte są w pliki NN.h

|  |
| --- |
| #pragma once  #include <vector>  #include <math.h>  #include "json.hpp"  #define inputsSizeH 7  #define inputsSizeW 7  #define inputsSize 49  #define outputsSize 2 |

1. **Wyniki końcowe i wykresy**

Ostatecznie ilość inputów zaniżyliśmy do 7x7 – 49 wejść odpowiednio pozwala sieci na naukę zarówno reakcji na dane etapy poziomu, przeciwników, przeszkody. Zmniejszony rozmiar pozwala na dużo szybszą naukę z podobnym czasem reakcji. Wyjściami sieci na końcowym etapie pozostał skok oraz ruch w prawo.

Poniższe wykresy przedstawiają ocenę wartości fitness w poszczególnych generacjach. Końcowe generacje przedstawiają wyniki które oznaczają iż AI bez problemu jest w stanie przejść poziom.

Próba 1: 100 generacji

System zaczął wygrywać już w momencie 8. Generacji, po czym zaczął zwiększać swoją efektywność. Wbrew oczekiwaniom ilość wygranych gier w kolejnych generacjach nie rosła a zmieniała swoje wartości, najwyższy wynik zaobserwować możemy w 65 generacji, jest to 81 wygranych gier, co równe jest 40% ogółu prób. Wartość fitness początkowo rosła drastycznie, następnie następowały niewielkie zmiany pomiędzy wartościami 15521 a 15536, rozbieżność ta bierze się z algorytmu liczącego fitness, uwzględnia on czas w jakim gra została wygrana. Również z tego powodu ilość wygranych gier zmienia się w momentach wahania się czasu przejścia.

Próba 2: 30 generacji

Ta generacja przyniosła nam najlepsze wyniki i uważamy ją za nasze złote dziecię. Pierwsze wygrane odniosła już w 5 generacji i to w ilości 13 gier. W końcowych generacjach wyniki wygranej były zadowalające, wręcz genialne. Zakres przegranych 5-20 gier to wynik który określa że system znalazł doskonałe rozwiązanie i czas. Od 14 generacji wartość fitness była stała, system uznał że jest to doskonałe rozwiązanie z najszybszym czasem którego nie można już przebić.

Próba 3: 30 generacji

Połowa próby ulegała stałej nauce aż dobrnęła do poprawnego rozwiązania. Ilość wygranych pomimo niewielkich wahań możemy uznać za stałą, okolica 30 gier jest dobrym wynikiem .

Wartość wskaźnika fitness idealnie obrazuje proces nauki, rośnie dość często aby jak najszybciej dojść do poprawnego rozwiązania – przejścia poziomu.

Próba 4: 30 generacji

Proces nauki w tym przypadku przypadł na dłuższy okres. Fitness nie rósł zbyt drastycznie, utrzymywał przez długi czas stały poziom, ale ostatecznie pozwoliło to na osiągniecie wyniku 47 wygranych, jest to prawie 25% ogólnej liczby gier. W przypadku sieci neuronowej, której działanie często opiera się na losowości jest to wynik zadowalający, dający nam pewność że w kolejnych generacjach system wypracowałby dużo lepsze wyniki czasowe.

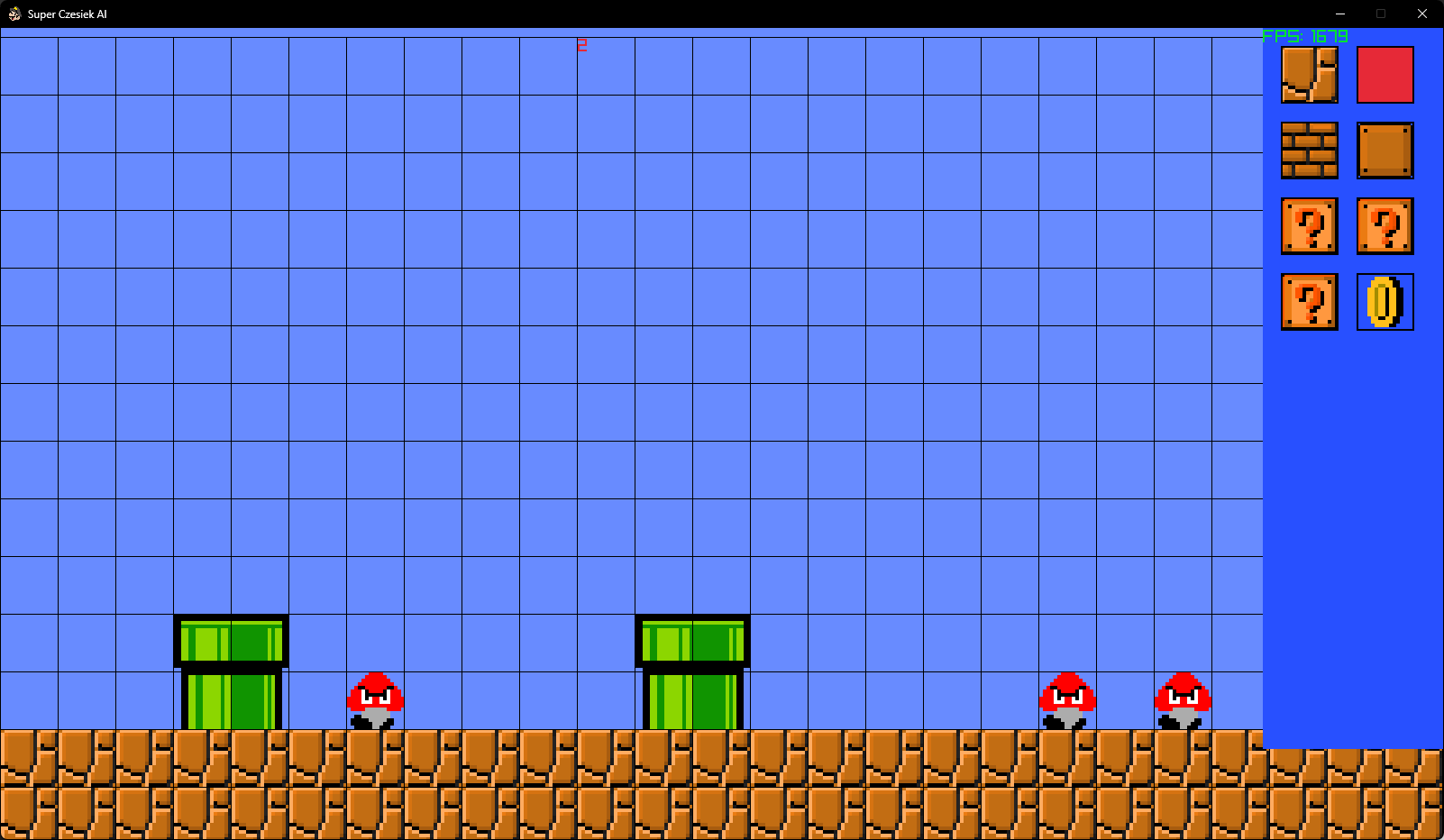
Próba 5: 30 generacji

Ta próba okazała się niepowodzeniem, wartość fitness na poziomie 9012 określa iż gra została pokonana w około 60%, jest to spora część poziomu który składa się z 8 chunków i wielu pułapek, lecz nie jest ona satysfakcjonująca, szczególnie biorąc pod uwagę inne próby które potrafiły osiągać wyniki co najmniej 30 wygranych.

1. **Zrzuty ekranu**
   * 1. **Menu gry**

****

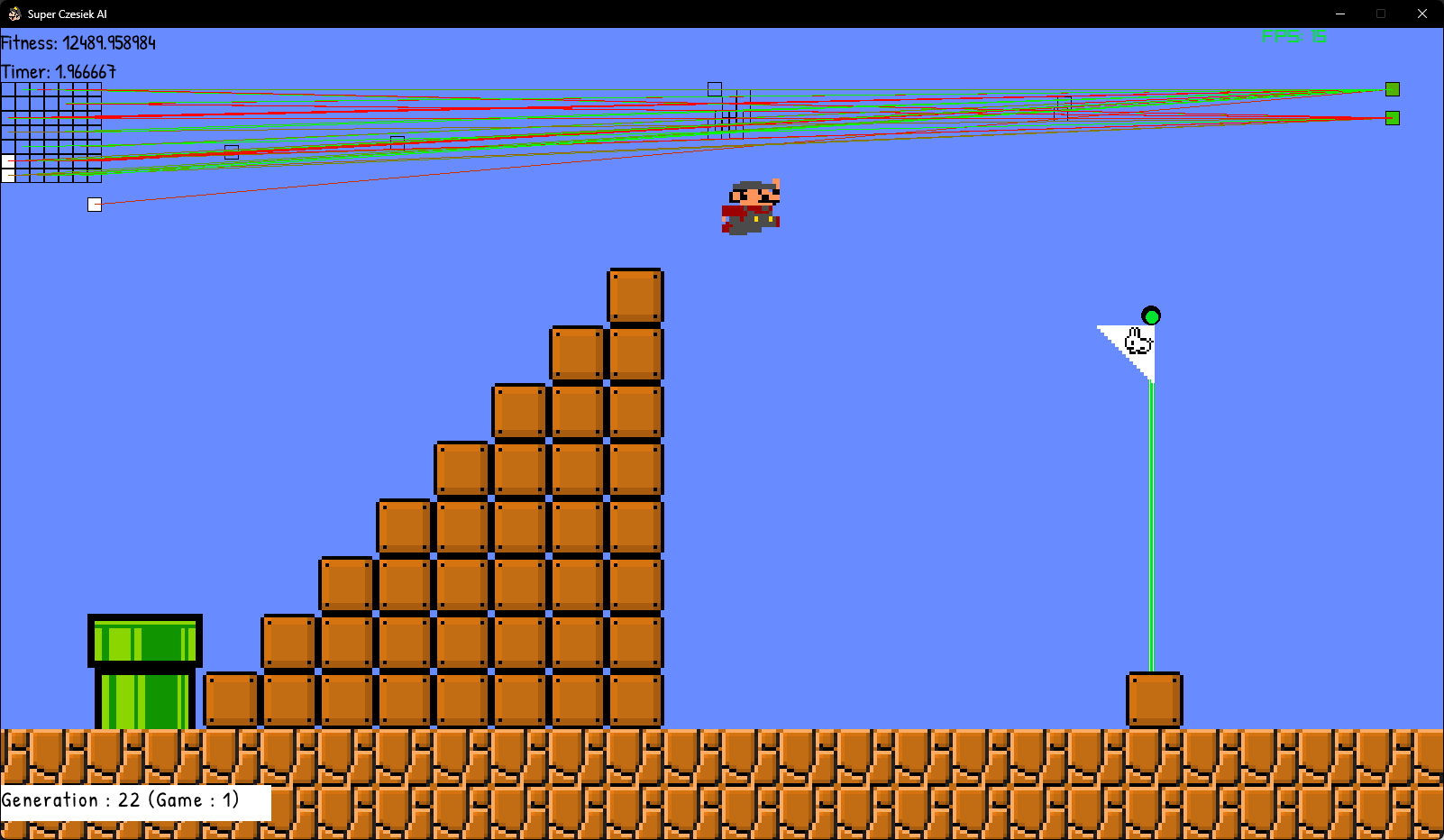
* + 1. **Edytor map**



* + 1. **Gra**



* + 1. **Nauka AI**

****