

POLITECHNIKA KOSZALIŃSKA



WYDZIAŁ ELEKTRONIKI I INFORMATYKI

INFORMATYKA

Przetwarzanie i Eksploracja Danych

inż. Patryk Banaś
U-15568

Generowanie światów Minecrafta z wykorzystaniem
algorytmów AI

Generating Minecraft worlds using AI algorithms

Praca magisterska wykonana pod kierunkiem
dr inż. Robert Świta

Koszalin 2023

Streszczenie

Celem pracy jest opracowanie podejścia opartego na sztucznej inteligencji do generowania światów gry na podstawie danych z Minecraft. Tradycyjne metody generowania światów opierają się głównie na losowości, co ogranicza kontrolę nad końcowym wynikiem. W niniejszej pracy zaproponowano model oparty na sieciach neuronowych, który umożliwia bardziej precyzyjne generowanie świata, dając użytkownikom możliwość wpływania na takie elementy jak kształt terenu i gęstość jaskiń. Omówiono zastosowane podejścia i mechanizmy, takie jak Generatywne Sieci Przeciwwstawne oraz mechanizmy uwagi, w kontekście ich wykorzystania do generowania światów w grze Minecraft. Dane treningowe zostały przygotowane poprzez ekstrakcję i formatowanie informacji z gry, redukując przy tym ich nadmiarowość. Model został oceniony pod kątem jakości generowanych struktur za pomocą odpowiednich metryk stosowanych w tego typu problemach. Wyniki badań wskazują, że sztuczna inteligencja może być skutecznym narzędziem do generowania światów w grach komputerowych. Zaproponowany model umożliwia tworzenie spersonalizowanych i kontrolowanych środowisk, co otwiera nowe możliwości w projektowaniu gier.

Generatywna Sztuczna Inteligencja, Generatywne Sieci Przeciwwstawne, Generowanie Wirtualnych Środowisk, Minecraft, Proceduralne Treści

Abstract

The aim of this thesis is to develop an artificial intelligence-based approach for generating game worlds using Minecraft data. Traditional methods of world generation rely primarily on randomness, which limits control over the final outcome. In this work, a neural network-based model is proposed that enables more precise world generation, giving users the ability to influence elements such as terrain shape and cave density. The applied approaches and mechanisms, such as Generative Adversarial Networks and attention mechanisms, are discussed in the context of their use for generating worlds in Minecraft. The training data was prepared through extraction and formatting of information from the game, while reducing data redundancy. The model was evaluated for the quality of generated structures using appropriate metrics commonly applied to such problems. The research results indicate that artificial intelligence can be an effective tool for generating worlds in computer games. The proposed model allows for the creation of personalized and controlled environments, opening up new possibilities in game design.

Generative Artificial Intelligence, Generative Adversarial Networks, Virtual Environment Generation, Minecraft, Procedural Content

Spis treści

1. Wprowadzenie	4
1.1 Motywacja	5
1.2 Cele pracy	6
2. Minecraft i generacja świata	7
2.1 Koncepcja chunków	7
2.2 Proceduralne generowanie świata w Minecraft	8
2.3 Wady generowania proceduralnego w grze Minecraft	9
2.4 Generowania świata opartego na sztucznej inteligencji	9
3. Sztuczna inteligencja i modele generatywne	10
3.1 Konwolucyjne sieci neuronowe (CNNs)	10
3.2 Generatywne sieci przeciwstawne (GAN)	11
3.3 Architektura U-Net	12
3.4 Mechanizmy uwagi	13
3.5 Techniki kodowania danych	14
3.6 Szum w sieciach GAN	14
4. Projekt i architektura modelu	15
4.1 Ogólna architektura	15
4.2 Sieć generatora	15
4.3 Sieć dyskryminatora	17
4.4 Wejścia i wyjścia modelu	18
4.5 Uwagi dotyczące treningu	19
4.6 Podsumowanie	19
5. Przygotowanie danych	20
5.1 Generacja świata	20
5.2 Ekstrakcja danych	21
5.3 Generowanie dodatkowych danych	22
5.4 Kodowanie i normalizacja danych	23
5.5 Uzasadnienie	23
6. Trenowanie modelu	24
6.1 Przygotowania do uczenia sieci	24
6.2 Funkcje straty	24
6.3 Warianty modeli	25

6.4 Wyniki uczenia sieci	26
7. Metryki oceny	29
7.1 Dystans Levenshteina	29
7.2 Tile Pattern Kullback-Leibler Divergence (TPKL-Div)	30
7.3 Średnia różnica wielkość gradientu (MGD)	31
8. Analiza wyników	33
8.1 Wyniki TPKL-Div	33
8.2 Wyniki dystansu Levenshteina	34
8.3 Porównanie histogramów bloków	35
8.4 Porównanie map terenów	36
8.5 Wnioski	37
9. Podsumowanie	38
Bibliografia	40
Spis rysunków	43
Spis tabel	44

1. Wprowadzenie

Proceduralne generowanie treści (*ang. Procedural Content Generation, PCG*) jest ważnym obszarem w rozwoju gier, obejmujący algorytmiczne tworzenie treści gry z ograniczonym lub pośrednim udziałem użytkownika. Obejmuje techniki, które dynamicznie generują zasoby gry, takie jak poziomy, mapy, zasady, historie i postacie, często zapewniając wyjątkowe doświadczenia podczas każdej rozgrywki. [1] [2]

PCG odgrywa kluczową rolę w nowoczesnych grach z kilku powodów:

1. **Zwiększona powtarzalność:** PCG wzbogaca rozgrywkę poprzez tworzenie nowych elementów przy każdej sesji gry. Gry takie jak *Rogue* [3] i inne z gatunku gier roguelike¹ wykorzystują PCG do tworzenia nowych układów lochów, zapewniając, że gracze napotkają unikalne wyzwania przy każdej rozgrywce. [4]
2. **Efektywne tworzenie treści:** Ręczne tworzenie rozległych światów gry jest czasochłonne i wymaga dużych zasobów. PCG pozwala programistom na algorytmiczne produkowanie ogromnych ilości treści, co skraca czas i koszty rozwoju. Ta efektywność jest szczególnie korzystna dla niezależnych programistów z ograniczonymi zasobami.
3. **Skalowalność i eksploracja:** PCG pozwala na generowanie niezwykle rozległych światów, których ręczne zaprojektowanie byłoby praktycznie niemożliwe. Na przykład, *No Man's Sky* wykorzystuje PCG do generowania całego wszechświata z ponad 18 kwintylionami planet, oferując graczom praktycznie nieograniczone możliwości eksploracji. [5]
4. **Dostosowywanie i personalizacja:** PCG może dostosowywać zawartość do indywidualnych preferencji lub akcji gracza, tworząc spersonalizowane doświadczenia. Adaptacyjne systemy trudności i zawartości reagujące na działania gracza, zwiększają zaangażowanie i satysfakcję.
5. **Innowacja i kreatywność:** Deweloperzy mogą używać PCG do eksperymentowania z nowymi pomysłami i mechanikami rozgrywki. Algorytmiczne generowanie treści może prowadzić do nieoczekiwanych i nowych doświadczeń, które mogą nie pojawić się w tradycyjnych procesach projektowania. [6]

Losowość jest podstawowym elementem w wielu tradycyjnych metodach PCG, zapewniając zmienność i nieprzewidywalność. Generatory losowych liczb są często używane do tworzenia różnorodności w treści, takich jak cechy terenu, rozmieszczenie przedmiotów albo konfiguracje wrogów w grze. Jednak poleganie wyłącznie na losowości wprowadza kilka ograniczeń:

1. **Brak kontroli nad generowaną treścią:** Czysto losowe generowanie może wytworzyć niespójną lub niegrywalną treść. Poziomy mogą być zbyt łatwe lub nawet niemożliwe do ukończenia, a narracje mogą nie mieć logicznego postępu. Ta nieprzewidywalność może zabrać przyjemność z grania w grę.
2. **Niespójna jakość:** Losowość nie zawsze gwarantuje wysokiej jakości ani zgodności z zasadami projektowania. Generowana treść bez odpowiednich ograniczeń może nie spełniać estetycznych standardów lub wymagań rozgrywki, co może prowadzić do chaotycznych doświadczeń. [7]

¹ rodzaj gry komputerowej, która z założenia ma być podobna do *Rogue* [3]

3. **Powtarzalność pomimo losowości:** Podczas gdy losowość ma na celu stworzenie różnorodności, czasami może skutkować powtarzającymi się wzorcami, szczególnie jeśli algorytm nie jest wystarczająco złożony. Gracze mogą zauważyć powtarzające się elementy, zmniejszając przy tym poczucie różnorodności generowanych treści.
4. **Trudność w osiągnięciu konkretnych celów projektowych:** Projektanci mogą mieć konkretne tematy lub konkretną wizję gry, które chcą przekazać. Generowanie losowe utrudnia w zapewnieniu, że treść jest zgodna z tymi celami, ponieważ działa bez świadomości o szerszych intencjach projektowanej gry. [8]
5. **Frustracja gracza:** Nieprzewidywalne skoki trudności lub niesprawiedliwe scenariusze generowane przez losowe algorytmy mogą prowadzić do frustracji gracza. Bez mechanizmów równoważących treści gry, gracze mogą napotkać nie do pokonania wyzwania lub nieangażującą rozgrywkę.

Aby zniwelować te ograniczenia, nowoczesne podejścia w PCG obejmują bardziej wyrafinowane techniki, które łączą losowość z kontrolowanym projektem. Te metody obejmują:

- Systemy oparte na regułach: Wdrażanie reguł i ograniczeń służących do kierowania generowaniem treści gwarantuje, że dane wyjściowe będą zgodne ze szczegółowymi parametrami projektu. [9]
- Problemy spełnienia ograniczeń (*ang. Constraint satisfaction problems, CSPs*): Korzystanie z technik CSP umożliwia generowanie treści, które spełniają zestaw wstępnie zdefiniowanych ograniczeń, poprawiają spójność i grywalność. [10]
- Uczenie maszynowe i sztuczna inteligencja: Wykorzystanie sztucznej inteligencji umożliwia tworzenie treści, które uczą się na podstawie istniejących danych, rejestrując wzorce i style zgodne z celami projektowymi. [11]
- Modelowanie gracza: Dostosowanie treści na podstawie zachowań i preferencji gracza prowadzi do spersonalizowanych doświadczeń, które zwiększają zaangażowanie.

Dzięki integracji tych zaawansowanych metod programiści mogą wykorzystać zalety PCG, ograniczając jednocześnie niedogodności związane z czystą losowością.

1.1 Motywacja

Tradycyjne metody proceduralnego generowania treści (PCG), oparte głównie na losowości, często ograniczają możliwości pełnej kontroli nad tworzonymi światami, zarówno dla deweloperów, jak i użytkowników. To ograniczenie skłoniło badaczy i twórców do wykorzystania technik sztucznej inteligencji (AI) w celu zwiększenia precyzji i personalizacji w generowaniu treści. [11] [12]

Dzięki metodom uczenia się maszynowego, możliwe staje się analizowanie wzorców z istniejących danych i tworzenie światów, które spełniają określone założenia projektowe. Trenowanie modeli AI na zestawach danych umożliwia generowanie treści zgodnych z parametrami zdefiniowanymi przez użytkownika, co pozwala na kontrolowanie takich aspektów jak struktura terenu, rozmieszczenie zasobów czy styl architektury.

Jednym z najbardziej znaczących przykładów badań nad generowaniem treści w Minecraft jest konkurs *Generative Design in Minecraft* (GDMC), który koncentruje się na generowaniu osad dostosowanych do specyfiki terenu. W tym konkursie algorytmy AI mają za zadanie tworzyć osady, które nie tylko spełniają funkcjonalne wymagania, ale także adaptują

się do różnych topografii, takich jak wyspy, pasma górskie czy różnorodne biomy [13] [14]. Konkurs ten inspiruje liczne prace naukowe, które badają kreatywność i adaptacyjność AI w kontekście proceduralnego generowania złożonych i dynamicznych struktur.

W niniejszej pracy omówiono zastosowanie sztucznej inteligencji do generowania światów w grze Minecraft. Celem jest zbadanie, w jaki sposób AI może przewyższyć ograniczenia tradycyjnych metod PCG, zapewniając jednocześnie różnorodność i kontrolę nad generowanymi treściami. Poprzez zastosowanie sieci neuronowych możliwe jest generowanie światów zgodnych z indywidualnymi preferencjami graczy, a jednocześnie zachowanie unikalności typowej dla proceduralnego podejścia.

1.2 Cele pracy

Głównym celem tej pracy jest opracowanie podejścia opartego na sztucznej inteligencji do generowania światów w grze Minecraft, które pozwala na większą kontrolę i personalizację w porównaniu z tradycyjnymi metodami proceduralnego generowania treści (PCG). Wykorzystując różne architektury sieci neuronowych, w szczególności Generative Adversarial Networks (GAN), dąży się do stworzenia modelu zdolnego generować świat gry, który jest zgodny z parametrami zdefiniowanymi przez użytkownika, takimi jak kształt terenu (mapa wysokości) i gęstość jaskiń. To podejście oparte na sztucznej inteligencji ma na celu przewyższenie ograniczeń wynikających z losowości nieodłącznie związanej z typowym generowaniem świata Minecrafta, umożliwiając zarówno programistom, jak i graczom większą personalizację określonych aspektów środowiska gry.

Aby osiągnąć ten główny cel, ustalono następujące cele cząstkowe:

Projektowanie i wdrażanie modelu: Stworzenie architektury sieci neuronowej, która może przetwarzać wejściowe mapy wysokości i dane o gęstości jaskiń w celu generowania spójnych i ustrukturyzowanych światów Minecrafta. Model ten będzie zawierał warunkowe dane wejściowe, aby umożliwić elastyczne projektowanie świata na podstawie kryteriów zdefiniowanych przez użytkownika.

Trenowanie modelu: Model zostanie wytrenowany na zbiorze danych z wygenerowanych wcześniej światów Minecrafta, aby nauczyć się generować nowe struktury świata na podstawie wzorców obecnych w danych treningowych. Proces uczenia będzie obejmował dostosowanie różnych hiperparametrów², aby znaleźć optymalną równowagę między różnorodnością generowanych światów a zgodnością z danymi wejściowymi użytkownika.

Ocena i porównanie: Ocena wygenerowanych światów będzie polegać na porównaniu z rzeczywistymi danymi Minecrafta, aby zbadać zgodność modelu z oczekiwaną dystrybucją typów bloków oraz strukturą generowanych środowisk. Analiza będzie obejmować ocenę rozmieszczenia bloków, zgodności mapy wysokości, gęstości jaskiń oraz innych istotnych cech. Różne wskaźniki, takie jak Tile Pattern Kullback-Leibler Divergence (TPKL-Div) i odległość Levenshteina, zostaną zastosowane do pomiaru jakości generowanych struktur względem innych modeli. Celem oceny jest nie tylko weryfikacja różnorodności generowanego świata, ale również zapewnienie, że model tworzy realistyczne i spójne środowiska, zgodne z parametrami ustalonymi przez użytkownika oraz wymogami gry.

² Regulowane parametry, które umożliwiają sterowanie procesem trenowania modelu.

2. Minecraft i generacja świata

Minecraft to sandboxowa gra wideo stworzona przez Mojang Studios i wydana po raz pierwszy w 2011 roku. Pozwala ona graczom eksplorować praktycznie nieskończony, proceduralnie generowany świat 3D złożony z bloków reprezentujących różne materiały, takie jak ziemia, kamień, rudy, woda czy lava. Gra słynie z kreatywnej swobody, umożliwiając graczom budowanie struktur, wytwarzanie przedmiotów i angażowanie się w wyzwania związane z przetrwaniem. [15]



Rysunek 1. Zrzut ekranu typowego krajobrazu Minecrafta prezentującego środowisko zbudowane z bloków.

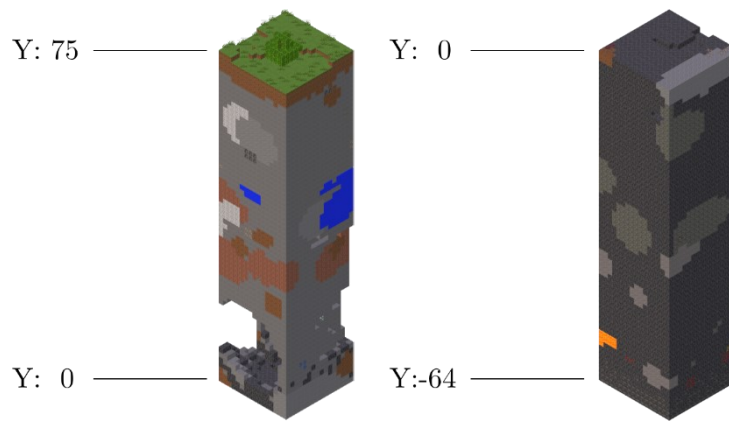
Minecraft stał się jedną z najlepiej sprzedających się gier wideo wszech czasów, z ponad 200 milionami sprzedanych egzemplarzy na wszystkich platformach i dużą, aktywną społecznością [16]. Jego prosta, ale potężna mechanika sprawiła, że stał się platformą dla kreatywności, edukacji, a nawet badań naukowych.

2.1 Koncepcja chunków

W grze Minecraft świat jest podzielony na regiony zwane chunkami. Chunk to obszar o wymiarach 16×16 bloków, który rozciąga się pionowo od dołu świata ($Y = -64$) do góry ($Y = 320$ w najnowszych wersjach) [17]. Chunki są podstawowymi jednostkami generowania, ładowania i przechowywania świata w grze Minecraft.

Chunki służą kilku istotnym celom w grze:

- Generowanie świata: Chunki są generowane proceduralnie, gdy gracz eksploruje świat, co pozwala na efektywnie nieskończony świat gry bez nadmiernego zużycia pamięci.
- Optymalizacja wydajności: Dzieląc świat na chunki, Minecraft może dynamicznie ładować i rozładowywać sekcje świata na podstawie pozycji gracza, optymalizując wykorzystanie zasobów.
- Zarządzanie danymi: Chunki umożliwiają wydajne przechowywanie i pobieranie danych świata, ponieważ każdy chunk jest przechowywany oddzielnie w plikach zapisu gry.



Rysunek 2. Wizualizacja części podziemnej pojedynczego chunka przy wysokości 75 i głębokości -64.

Mechanika wczytywania chunków polega na tym, że gdy gracz znajduje się w pobliżu chunka, silnik gry ładuje go do pamięci i renderuje znajdujące się w nim bloki i jednostki. Gdy gracz się oddala to chunki, które nie znajdują się już w odległości renderowania, są rozładowywane w celu zwolnienia zasobów systemowych.

2.2 Proceduralne generowanie świata w Minecraft

Generowanie świata w Minecraft jest przede wszystkim proceduralne, co oznacza, że wykorzystuje algorytmy do dynamicznego tworzenia terenu i struktur, zamiast polegać na gotowej zawartości. Takie podejście pozwala na tworzenie rozległych, unikalnych światów przy minimalnym ręcznym projektowaniu.

Teren w grze Minecraft jest generowany przy użyciu szumu Perlina we wcześniejszych wersjach i szumu *Simplex* w nowszych aktualizacjach [18]. Obie są funkcjami szumu opartymi na gradiencie, opracowanymi przez Kena Perlina w celu generowania płynnych, pseudolosowych zmian w przestrzeni. Te funkcje szumu pomagają określić różne cechy świata, takie jak wysokość terenu i rozkład biomów. [19]

Świat Minecrafta składa się z różnych biomów, takich jak lasy, pustynie i oceany, z których każdy ma różne warunki środowiskowe i typy bloków [20]. System generowania proceduralnego określa rozmieszczenie biomów i odpowiednio generuje elementy terenu.

Generowane funkcje obejmują:

- Góry i wzgórza: Podwyższony teren utworzony przez zmianę wyjścia funkcji szumu.
- Jaskinie i wąwozy: Podziemne struktury wyrzeźbione przy użyciu progów szumu.
- Roślinność: Drzewa, rośliny i kwiaty umieszczone na podstawie specyficznych zasad dla danego biomu.
- Struktury: Wioski, świątynie i inne budynki wygenerowane przy użyciu predefiniowanych szablonów i rozmieszczenia proceduralnego.

Rola losowości

Losowość jest integralną częścią proceduralnego generowania świata. Każdy świat jest generowany w oparciu o *seed*, czyli wartość liczbową, która inicjuje generator liczb losowych. Korzystanie z tego samego *seed'a* tworzy ten sam układ świata [21]. Losowość zapewnia, że

każdy świat jest wyjątkowy, zapewniając graczom nowe doświadczenia w każdej grze. Rudy i inne zasoby są losowo rozmieszczane w świecie, wpływając na rozgrywkę i eksplorację.

2.3 Wady generowania proceduralnego w grze Minecraft

Choć na ogół generowany świat jest nieprzewidywalny dla przeciętnego gracza, mogą pojawić się pewne wzorce, prowadzące do powtarzalności terenu w niektórych obszarach. Gracze mają ograniczoną możliwość wpływania na generowanie świata bez korzystania z zewnętrznych narzędzi lub modyfikacji, co ogranicza spersonalizowane doświadczenia.

Proceduralnie generowane struktury mogą czasami być bezsensowne lub pozbawione spójności, np. latające wyspy lub ucięte jaskinie. Zautomatyzowane generowanie może nie osiągnąć tego samego poziomu szczegółowości lub kreatywności, co ręcznie tworzona zawartość, potencjalnie zmniejszając atrakcyjność wizualną.

Generowanie złożonych światów w czasie rzeczywistym może być wymagające obliczeniowo, wpływając na wydajność słabszych systemów. Ponieważ gracz porusza się szybko, gra musi szybko generować nowe chunki, co może prowadzić do opóźnień.

2.4 Generowania świata opartego na sztucznej inteligencji

Ograniczenia generowania proceduralnego w grze Minecraft podkreślają potencjalne korzyści płynące z integracji technik sztucznej inteligencji.

Modele sztucznej inteligencji mogą uczyć się na podstawie rzeczywistych danych terenu lub krajobrazów zaprojektowanych przez graczy, aby tworzyć bardziej realistyczne i zróżnicowane środowiska. Modele uczenia maszynowego mogą być warunkowane określonymi danymi wejściowymi, pozwalając graczom bezpośrednio wpływać na cechy terenu, rozmieszczenie biomeów lub systemy jaskiń. Sztuczna inteligencja może generować struktury i krajobrazy o większej spójności i kreatywności, przewyższając niektóre wady związane z losowością.

3. Sztuczna inteligencja i modele generatywne

W tym rozdziale omówiono teoretyczne podstawy niezbędne do zrozumienia rozwoju i implementacji generatora świata Minecrafta opartego na sztucznej inteligencji, z wykorzystaniem sieci neuronowych. Przedstawiono kluczowe koncepcje, takie jak konwolucyjne sieci neuronowe (CNN), generatywne sieci przeciwstawne (GAN), architektura U-Net, mechanizmy uwagi, dyskryminator PatchGAN, techniki kodowania danych oraz role szumu w sieciach GAN. Zgłębiając te zagadnienia, czytelnik zyska wgląd w fundamenty, które wspierają architekturę modeli i procesy uczenia opisane w kolejnych rozdziałach.

3.1 Konwolucyjne sieci neuronowe (CNNs)

Klasa głębokich sieci neuronowych powszechnie wykorzystywanych do przetwarzania danych o topologii podobnej do siatki, takich jak obrazy (siatki 2D) i dane wolumetryczne (siatki 3D). CNN są zaprojektowane do automatycznego i adaptacyjnego uczenia się przestrzennych hierarchii cech poprzez propagację wsteczną (ang. *backpropagation*) przy użyciu wielu bloków konstrukcyjnych, takich jak warstwy konwolucyjne (splotowe), warstwy łączące (ang. *pooling layers*) i warstwy w pełni połączone (ang. *fully connected layer*). [22]

Warstwy konwolucyjne

Są podstawowymi elementami CNN. Stosują one zestaw uczących się filtrów (kernele), które łączą dane wejściowe w celu utworzenia map cech. Każdy filtr aktywuje pewne cechy w określonych pozycjach przestrzennych na wejściu. [23]

$$Output(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} Input(i+m, j+n) \times Kernel(m,n)$$

Wejście 2D		Kernel 3x3		Wyjście																													
<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	1	0	1	0	0	0	1	1	0	0	0	1	0	1	1	0	*	<table border="1"><tr><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>-1</td></tr><tr><td>0</td><td>-1</td><td>-1</td></tr></table>	1	1	0	1	0	-1	0	-1	-1	=	<table border="1"><tr><td>0</td><td>-1</td></tr><tr><td>-2</td><td>-1</td></tr></table>	0	-1	-2	-1
1	0	1	0																														
0	0	1	1																														
0	0	0	1																														
0	1	1	0																														
1	1	0																															
1	0	-1																															
0	-1	-1																															
0	-1																																
-2	-1																																

Stride (krok): 1

Rysunek 3. Wizualizacja operacji splotu z kernelem 3x3 na wejściu 2D o wymiarach 4x4.

Na Rysunek 3 *strides* oznacza krok przesunięcia okna filtra, zazwyczaj jest to 1. Dla każdego wymiaru można ustalić inną wartość kroku, zazwyczaj na każdym wymiarze jest ustawiana ta sama wartość.

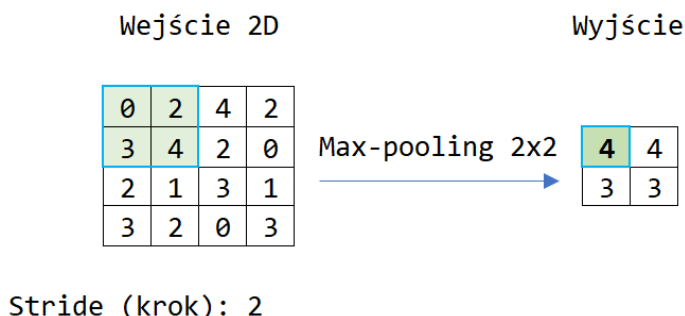
Funkcje aktywacji

Wprowadzają nieliniowość do sieci, umożliwiając jej uczenie się złożonych wzorców. Typowe funkcje aktywacji obejmują ReLU (Rectified Linear Unit), sigmoid i tanh. [24]

$$ReLU(x) = \max(0, x)$$

Warstwy łączące (ang. *pooling layers*)

Zmniejszają wymiary przestrzenne map cech, co zmniejsza złożoność obliczeniową i pomaga zapobiegać nadmiernemu dopasowaniu. Typowe rodzaje łączenia obejmują łączenie maksymalne i łączenie średniej. [25]



Rysunek 4. Wizualizacja operacji max-pool 2x2 z przesunięciem okna równym 2.

Tak samo jak w warstwach konwolucyjnych, tutaj też możemy ustalić krok przesunięcia okna.

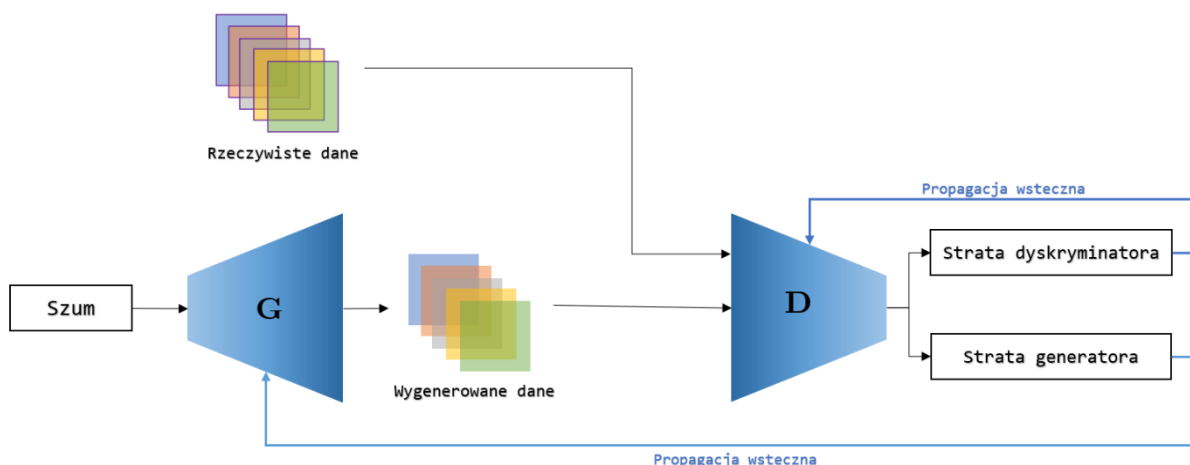
Konwolucje 3D

W przypadku danych wolumetrycznych, takich jak chunki Minecrafta, stosowane są konwolucje 3D. Rozszerzają one koncepcję splotów 2D, stosując kernel w trzech wymiarach, przechwytyując hierarchie przestrzenne w przestrzeni wolumetrycznej. [26]

$$Output(i, j, k) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{p=0}^{P-1} Input(i + m, j + n, k + p) \times Kernel(m, n, p)$$

3.2 Generatywne sieci przeciwstawne (GAN)

GAN to sposób uczenia maszynowego, gdzie dwie sieci neuronowe, generator i dyskryminator, są trenowane jednocześnie za pomocą procesów przeciwstawnych. Generator ma na celu generowanie danych nieodróżnialnych od danych rzeczywistych, podczas gdy dyskryminator próbuje odróżnić dane rzeczywiste od wygenerowanych przez generator. [27]



Rysunek 5. Schemat ilustrujący interakcję między generatorem a dyskryminatorem w sieci GAN.

Szkolenie GAN odpowiada grze *minimax*, sformalizowaną przez następującą funkcję celu:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Gdzie:

- $D(x)$ jest oszacowaniem przez dyskryminator prawdopodobieństwa, że instancja danych x jest prawdziwa.
- $G(z)$ jest wyjściem generatora przy danym szumie z .
- $p_{data}(x)$ jest rzeczywistym rozkładem danych.
- $p_z(z)$ jest rozkładem szumu.

Warunkowe sieci GAN (cGAN) rozszerzają strukturę GAN poprzez uzależnienie zarówno generatora, jak i dyskryminatora od dodatkowych informacji y :

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x|y)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z|y)))]$$

Pozwala to na kontrolowane generowanie danych w oparciu o warunki wejściowe, co ma kluczowe znaczenie dla zadań wymagających określonych wyników, takich jak generowanie terenu o określonych cechach. [28]

Oryginalna sieć GAN wykorzystuje binarną stratę entropii krzyżowej. Jednak warianty, takie jak Wasserstein GAN (WGAN), proponują alternatywne funkcje strat w celu poprawy stabilności treningu. [29]

Funkcja straty Wasserstein'a:

$$L = \mathbb{E}_{\tilde{x} \sim P_G} [D(\tilde{x})] - \mathbb{E}_{x \sim P_{real}} [D(x)]$$

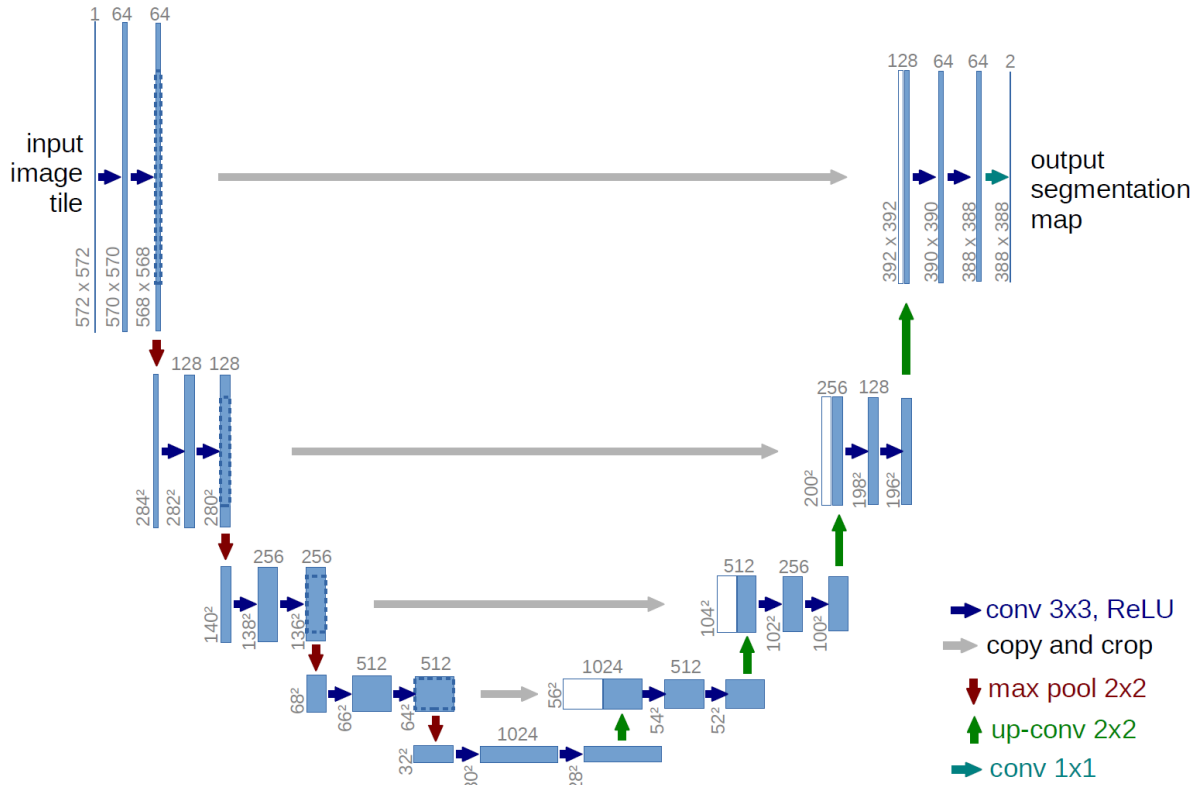
Jest to różnica między średnią predykcji dyskryminatora na rzeczywistych danych a średnią predykcji na wygenerowanych danych.

3.3 Architektura U-Net

U-Net to rodzaj konwolucyjnej sieci neuronowej pierwotnie zaprojektowanej do segmentacji obrazów biomedycznych. Ma strukturę kodera-dekoder z połączeniami pomijającymi, które łączą odpowiednie warstwy w ścieżkach kodera i dekoder. [30]

Struktura koder-dekoder

- Koder (ścieżka umowna): Składa się z wielokrotnego zastosowania warstw konwolucyjnych i łączących, przechwytyujących informacje kontekstowe.
- Dekoder (ścieżka rozszerzająca): Używa transponowanych konwolucji i próbkowania w celu zwiększenia wymiarów przestrzennych, rekonstruując dane wyjściowe.
- Pomiń połączenia: Bezpośrednio łączą warstwy kodera z warstwami dekoder, umożliwiając dostęp do cech o wysokiej rozdzielczości utraconych podczas próbkowania w dół.



Rysunek 6. Schemat architektury U-Net przedstawiający połączenia kodera, dekodera i pomijania. Źródło: [48]

3.4 Mechanizmy uwagi

Mechanizmy uwagi (z ang. *Attention*) pozwalają sieciom neuronowym skupić się na określonych częściach danych wejściowych, poprawiając wydajność w zadaniach, w których ważny jest kontekst i relacje. „Uwaga”, pierwotnie opracowana do przetwarzania języka naturalnego, została zaadaptowana do zadań widzenia komputerowego. [31]

Self-Attention (Samo-uwaga)

Mechanizm samo-uwagi oblicza odpowiedź na danej pozycji w sekwencji poprzez uwzględnienie wszystkich pozycji i obliczenie ich ważonej średniej. W zadaniach wizji komputerowej samo-uwaga pomaga modelowi skupić się na istotnych obszarach przestrzennych. [31]

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Gdzie:

- Q , K i V to macierze zapytań, kluczy i wartości.
- d_k jest wymiarem wektorów kluczy.

W sieciach CNN można zintegrować mechanizmy uwagi w celu ulepszenia reprezentacji cech. Obejmuje to uwagę przestrzenną, uwagę kanałów i podejścia łączone. [32]

3.5 Techniki kodowania danych

Kodowanie One-Hot

Reprezentuje zmienne kategoryczne jako wektory binarne. Dla każdej kategorii tworzony jest wektor binarny, w którym tylko indeks odpowiadający kategorii ma wartość 1, a wszystkie inne indeksy mają wartość 0.

Przykład

Dla bloków typu {Air, Dirt, Stone}:

- Air: [1, 0, 0]
- Dirt: [0, 1, 0]
- Stone: [0, 0, 1]

Normalizacja

Skaluje dane numeryczne do standardowego zakresu, zazwyczaj [0, 1] lub [-1, 1], poprawiając zbieżność podczas uczenia. Techniki obejmują skalowanie min-max i normalizację z-score.

Min-Max Scaling:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Normalizacja zapobiega dużym nachyleniom, które mogą destabilizować trening. Przyspiesza zbieżność, zapewniając równy udział cech i zapewnia, że dane są w formacie odpowiednim dla architektury sieci.

3.6 Szum w sieciach GAN

W sieciach GAN szum jest wprowadzany na wejście generatora, aby zachęcić do różnorodności generowanych wyników. Pomaga to zapobiec uczeniu się przez generator deterministycznego mapowania, promując kreatywność i zapobiegając nadmiernemu dopasowaniu. [27]

Techniki stosowania szumu

1. Losowe wektory szumu: Próbkowane z rozkładu (np. gaussowskiego) i wprowadzane do generatora.
2. Szum wejściowy: Dodawany do danych wejściowych w celu uodpornienia modelu na zmiany.
3. Szum cech: wstrzykiwany do warstw pośrednich.

W zaimplementowanym modelu, dyskretny szum jest nakładany na objętość wejściową poza generatorem. Takie podejście wprowadza zmienność i zapobiega prostemu kopiowaniu objętości wejściowej przez generator.

4. Projekt i architektura modelu

Zaprojektowanie efektywnej architektury sieci neuronowej jest kluczowe do opracowania generatora świata Minecraft opartego na sztucznej inteligencji. Architektura musi uchwycić złożone wzorce, struktury terenów i jaskiń, a jednocześnie uwzględnić dane wejściowe warunkowe, które umożliwiają użytkownikowi kontrolę nad tymi elementami. Ten rozdział zawiera przegląd architektury modelu zastosowanej w tym projekcie, podkreślając sieci generatorów i dyskryminatorów, ich komponenty strukturalne oraz mechanizmy zintegrowane w celu zwiększenia wydajności.

4.1 Ogólna architektura

Model przyjmuje strukturę generatywnej sieci przeciwstawnej (GAN), składającej się z dwóch podstawowych komponentów: generatora i dyskryminatora. Obie sieci oparte są na architekturze U-Net, która dobrze się nadaje do zadań wymagających precyzyjnej lokalizacji i zrozumienia kontekstu ze względu na symetryczną strukturę kodera-dekodera z pomijanymi połączeniami (technika zwana z ang. skip connections).

Sieć generująca ma na celu tworzenie realistycznych chunków Minecrafta uwarunkowanych cechami wejściowymi, takimi jak mapy wysokości i gęstości jaskiń. Sieć dyskryminacyjna ocenia autentyczność wygenerowanych chunków, odróżniając je od prawdziwych chunków wyodrębnionych z gry.

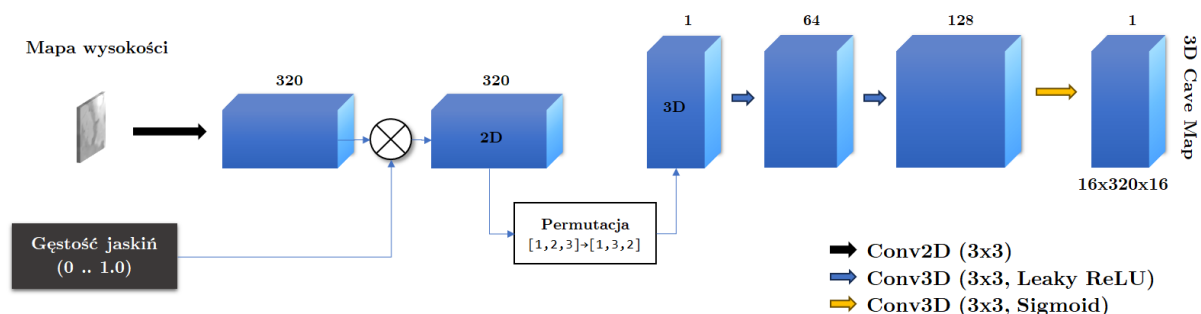
4.2 Sieć generatora

Generator został zaprojektowany do tworzenia chunków 3D o rozmiarze $16 \times 320 \times 16$ z sześcioma typami bloków. Jako dane wejściowe przyjmuje zaszumioną objętość wejściową, mapę wysokości i wartość gęstości jaskini. Zniekształcona objętość wejściowa to wstępnie ukształtowana objętość chunka z dodanym szumem, zachęcająca generator do uczenia się znaczących struktur zamiast zapamiętywania danych wejściowych. Mapa wysokości reprezentuje wysokość powierzchni w każdej współrzędnej (x, z) , a gęstość jaskiń jest wartością skalarną wskazującą pożądaną gęstość jaskiń w obrębie chunka.

Generator zawiera kilka kluczowych komponentów:

- **U-Net Backbone:** Struktura kodera-dekodera z pomijanymi połączeniami w celu zachowania informacji przestrzennych między warstwami.
- **Integracja wejść warunkowych:** Uwzględnia mapę wysokości i gęstość jaskini w procesie generowania.
- **Mechanizm uwagi w jaskini:** Mechanizm inspirowany uwagą, który prowadzi generator do skupienia się na strukturach jaskiń w różnych skalach.
- **Warstwa Cave Map:** Niestandardowa warstwa, która przetwarza mapę wysokości i gęstość jaskini w celu utworzenia mapy jaskini warunkującej wynik generatora.

Conditional Cave Map Layer



Rysunek 8. Ilustracja przepływu przetwarzania w warstwie warunkowej Cave Map.

Warstwa Conditional Cave Map Layer to niestandardowy komponent, który generuje mapę jaskiń 3D na podstawie mapy wysokości i gęstości jaskiń. Przetwarza ona mapę wysokości i gęstość jaskiń w celu utworzenia mapy jaskini, która warunkuje dane wyjściowe generatora. Warstwa przekształca mapę wysokości, stosuje zwoje 2D w celu wyodrębnienia cech, mnoży cechy przez gęstość jaskini w celu uwarunkowania mapy, transponuje i przekształca dane wyjściowe do objętości 3D i stosuje serię zwojów 3D w celu wygenerowania mapy jaskini. Wynikowa mapa jaskiń podkreśla regiony, w których należy podkreślić jaskinie.

Zastosowanie szumu

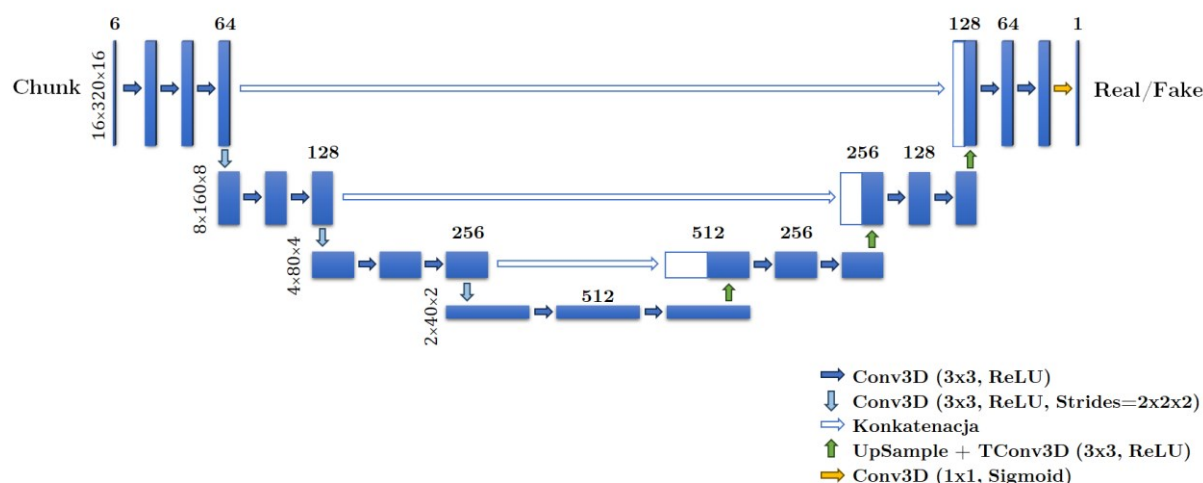
Aby zapobiec zwykłemu kopiowaniu objętości wejściowej przez generator, szum jest stosowany do objętości wejściowej poza modelem. Losowy dyskretny szum jest generowany i dodawany do objętości wejściowej, wprowadzając zmienność. Zachęca to generator do uczenia się podstawowego rozkładu danych i generowania realistycznych zmian.

4.3 Sieć dyskryminatora

Zadaniem dyskryminatora jest rozróżnianie między rzeczywistymi i wygenerowanymi chunkami. Jego architektura wykorzystuje pełną sieć U-Net z pomijanymi połączeniami, podobnie jak w przypadku generatora. Początkowo rozważano zastosowanie dyskryminatora PatchGAN [33], który ocenia lokalne fragmenty pod kątem realizmu. W ostatecznym projekcie zdecydowano się jednak na strukturę U-Net, która lepiej sprawdziła się w praktyce.

Dyskryminator odzwierciedla architekturę generatora, przechwytyując zarówno globalny kontekst, jak i drobne szczegóły. Połączenia pomijające zachowują informacje przestrzenne i pozwalają dyskryminatorowi ocenić autentyczność chunka w wielu skalach. Wyjściem jest pełnowymiarowy chunk, reprezentujący prawdopodobieństwo, że każdy blok jest prawdziwy lub wygenerowany.

Struktura U-Net w dyskryminatorze



Rysunek 9. Schemat architektury U-Net dyskryminatora.

Uzasadnienie dla dyskryminatora U-Net

Wyniki empiryczne wykazały, że dyskryminator U-Net zapewnia lepsze informacje zwrotne dla generatora w porównaniu z początkowym podejściem PatchGAN. Pełne dane wyjściowe chunka pozwalają dyskryminatorowi ocenić autentyczność w całym obrębie chunka, promując generowanie wyższej jakości.

4.4 Wejścia i wyjścia modelu

Generator otrzymuje kilka sygnałów wejściowych:

- Objętość wejściowa z szumem: Zakodowany poprzez one-hot, chunk, posiadający jedynie informacje o objętości z dodanym szumem, o kształcie (16,320,16,6).
- Mapa wysokości: Znormalizowane wartości wysokości powierzchni o kształcie (16,16).
- Gęstość jaskini: Wartość skalarna reprezentująca pożądaną gęstość jaskini.

Dane wyjściowe generatora:

- Wygenerowany chunk: kodowany one-hot, prawdopodobieństwa bloków dla każdego punktu, o kształcie (16,320,16,6).
- Wygenerowana mapa jaskiń: Mapa jaskini używana w modelach ze stratą mapy jaskini, o kształcie (16,320,16,1).

Dyskryminator ocenia chunki, przyjmując jako dane wejściowe zakodowany chunki o kształcie (16,320,16,6). Jego wyjściem jest mapa autentyczności, wskazująca prawdopodobieństwo, że każdy blok jest prawdziwy, o kształcie (16,320,16,1).

4.5 Uwagi dotyczące treningu

Zastosowanie szumu

Szum jest nakładany na objętość wejściową przed wprowadzeniem jej do generatora, zapobiegając nadmiernemu dopasowaniu generatora do danych wejściowych i zachęcając go do uczenia się znaczących wzorców.

Warunkowe możliwości GAN

Model został zaprojektowany jako warunkowa sieć GAN, wykorzystująca mapę wysokości i gęstość jaskini jako warunki. Chociaż architektura obsługuje mechanizmy warunkowego przełączania (między cGAN a GAN), nie były one aktywnie wykorzystywane w ostatecznej implementacji.

4.6 Podsumowanie

Architektura modelu łączy w sobie kilka zaawansowanych technik generowania realistycznych chunków Minecrafta. Architektura U-Net z pomijanymi połączeniami zapewnia solidną bazę do przechwytywania zarówno globalnych, jak i lokalnych cech. Włączenie warunkowych danych wejściowych i mechanizmu uwagi jaskini pozwala użytkownikowi kontrolować teren i struktury jaskiń, zwiększając elastyczność modelu. Projekt dyskryminatora, wykorzystujący strukturę U-Net, poprawia jakość informacji zwrotnych podczas treningu, prowadząc do lepszej wydajności generatora. Dzięki integracji tych komponentów model skutecznie uczy się generować spójne i kontrolowane światy Minecrafta, eliminując ograniczenia tradycyjnych metod generowania treści proceduralnej.

5. Przygotowanie danych

Skuteczność modelu sieci neuronowej w dużym stopniu zależy od jakości i przydatności danych użytych do treningu. W kontekście generowania światów Minecraft przy użyciu algorytmów AI, niezbędne jest przygotowanie zestawu danych, który dokładnie reprezentuje środowisko gry, a jednocześnie jest łatwy w zarządzaniu do przetwarzania. W tym rozdziale opisano kroki podejmowane w celu przygotowania danych użytych do trenowania sieci neuronowej, w tym generowanie świata, ekstrakcję danych, wstępne przetwarzanie i kodowanie.

5.1 Generacja świata

W tym projekcie wykorzystano wersję Minecraft 1.20.1 ze względu na jej stabilność i zgodność z niezbędnymi modyfikacjami. W celu umożliwienia integracji dodatkowych narzędzi, wymaganych do przygotowania danych, została zainstalowana modyfikacja *Forge* [34] (modyfikacja plików gry, która tworzy standardowe API do modyfikowania gry). Jest to bazowa modyfikacja, która jest wymagana do wielu modyfikacji stworzonych przez społeczność gry. Jedną z takich niezbędnych modyfikacji jest *Chunky* [35], jest to *pregenerator* chunków, który znacznie ułatwia i przyspiesza proces generowania dużych światów Minecraft.

Chunky to modyfikacja, która umożliwia wstępne generowanie chunków w świecie Minecraft. Wstępne generowanie chunków jest kluczowe dla zapewnienia, że dane światy są łatwo dostępne do ekstrakcji bez konieczności eksploracji w grze. Następujące kroki zostały wykonane przy użyciu *Chunky*:

1. Instalacja: Modyfikacja Chunky została umieszczona w zmodyfikowanej wersji Minecraft 1.20.1 Forge.
2. Konfiguracja świata: Nowy świat został stworzony z określonymi ustawieniami jak:
 - a. Jeden biot: Generowanie świata zostało ograniczone do pojedynczego biotu, konkretnie biotu „równin” (ang. *Plains*), aby zmniejszyć złożoność i zmienność danych.
 - b. Pominięcie struktur: Cechy takie jak wioski i inne monumenty zostały wyłączone z generacji, aby wyłącznie skupić się na terenie i formacjach podziemnych.
3. Polecenia wstępnej generacji: Polecenie `chunky corners 0 0 5120 5120` zostało użyte do zdefiniowania obszaru do wstępnej generacji. To polecenie ustawia współrzędne rogów prostokątnego regionu, który ma zostać wygenerowany, co skutkuje obszarem o wymiarach 320×320 chunków. Każdy chunk to wymiary 16×16 bloków.
4. Rozpoczęcie generacji: Polecenie `chunky start` inicjuje proces wstępnej generacji dla wybranego obszaru.





Ograniczając świat do jednego biotu i wyłączając dodatkowe struktury, proces ekstrakcji danych staje się prostszy. To podejście zmniejsza liczbę unikalnych typów bloków i cech specyficznych dla biotu, pozwalając skupić się na ogólnym terenie i strukturach jaskiń. Zapewnia również spójność w całym zestawie danych, co jest korzystne przy trenowaniu sieci neuronowej.

5.2 Ekstrakcja danych

Aby wyodrębnić dane ze wstępnie wygenerowanego świata, wykorzystano bibliotekę **amulet-core** [36]. Jest to biblioteka Pythona przeznaczona do edycji i manipulowania zapisanymi światami gry. Zapewnia funkcjonalność odczytu i zapisu danych świata programowo, co jest niezbędne do wyodrębnienia niezbędnych informacji do trenowania.

Minecraft oferuje szeroką gamę typów bloków, z ponad 830 blokami, które można umieścić w świecie gry, których wiele różni się jedynie parametrami, takimi jak obrót, stan czy rodzaj materiału [37]. Ta różnorodność stanowi wyzwanie w przetwarzaniu tych danych i uczeniu modelu ze względu na wysoką wymiarowość i rozrzedzenie (ang. *sparsity*) danych. Włącznie wszystkich typów bloków znacznie zwiększyłoby złożoność modelu bez zapewnienia proporcjonalnych korzyści.

Aby sprostować temu wyzwaniu, podjęto decyzję o zmniejszeniu liczby typów bloków do najbardziej podstawowych i powszechnie występujących w biomie „równin” (ang. *plains*). Wybrane bloki to:

1. Air (powietrze): Reprezentuje pustą przestrzeń, nad powierzchnią i wewnątrz jaskiń.
2.  Dirt (ziemia): Praktycznie zawsze występuje na warstwie powierzchniowej.
3.  Sand (piasek): Występuje w małych łatach na biomie równin. Często występuje przy spadkach terenu idących poniżej poziomu morza.
4.  Stone (kamień): Najbardziej rozpowszechniony podziemny blok. Jest głównym blokiem kształtującym teren.
5. Cave air (powietrze jaskiniowe): Podobnie jak Air, z taką różnicą, że jest używany do kształtowania jaskiń poniżej poziomu morza. Jaskinie stworzone z tego typu bloku nie są zalewane wodą, jak w przypadku Air występującego poniżej poziomu morza.
6.  Bedrock (podłoże skalne): Według zasad gry, jest to niezniszczalna warstwa bloków na dnie świata.

Skupiając się na tych sześciu typach bloków, zbiór danych staje się prostszy w zarządzaniu, a jednocześnie nadal obejmuje istotne cechy terenu i poćmionych struktur. Każdemu typowi bloku przypisano wartość całkowitą od 0 do 5 w celu identyfikacji.

Tabela 1. Mapowanie bloków na identyfikatory całkowite i ich kodowanie one-hot.

	Air	Dirt	Sand	Stone	Cave air	Bedrock
ID	0	1	2	3	4	5
One-hot	[1,0,0,0,0,0]	[0,1,0,0,0,0]	[0,0,1,0,0,0]	[0,0,0,1,0,0]	[0,0,0,0,1,0]	[0,0,0,0,0,1]

Proces ekstrakcji obejmuje następujące kroki:

1. Ładowanie chunków: Używając amulet-core, każdy chunk w obrębie wstępnie wygenerowanego obszaru został załadowany sekwencyjnie.

2. Mapowanie bloków: Bloki w obrębie każdego chunka zostały zmapowane na odpowiadające im identyfikatory na podstawie zredukowanej listy typów bloków.
3. Naprawieni jaskiń: W grze jaskinie są reprezentowane wyłącznie za pomocą bloków Air, bloki typu Cave air są jedynie używane przez silnik gry w trakcie generacji, a później są wymieniane na Air. Aby odróżnić powietrze nad ziemią od przestrzeni jaskiń pod ziemią, wszystkie bloki Air poniżej poziomu morza (warstwa 62 w grze) zostały przeklasyfikowane jako powietrze jaskiniowe.
4. Przechowywanie danych: Przetworzone chunki zostały zapisane jako tablice *NumPy* o wymiarach odpowiadających rozmiarowi chunka i wymiarom wcześniej wygenerowanego świata. Każdy element w tablicy reprezentuje identyfikator typu bloku.

5.3 Generowanie dodatkowych danych

Oprócz mapowań typów bloków wygenerowano dodatkowe dane, aby zapewnić sieci neuronowej szerszy kontekst.

Mapy wysokości (ang. *heightmaps*)

Reprezentują położenie najwyższego bloku wzdłuż osi Y niebędącego blokiem typu Air, na każdej współrzędnej (X, Z) w obrębie chunka. Mapy wysokości zostały wygenerowane dla każdego chunka poprzez iterację od góry chunka (Y=320) w dół.

Mapy wysokości dostarczają informacji i wysokości powierzchni, co jest kluczowe dla zrozumienia kształtu terenu i ukierunkowania modelu do generowania terenu.

Gęstość jaskiń

Liczba bloków powietrza jaskiniowego (Cave air) w obrębie chunka, szczególnie poniżej poziomu morza, aby podkreślić, że chodzi o jaskinie, które nie będą wypełnione blokiem wody.

Dla każdego chunka liczba bloków Cave air została zliczona w warstwach poniżej poziomu morza.

Gęstości jaskiń służy jako wskaźnik złożoności podziemnych struktur jaskiń i jest używana do warunkowania modelu w celu generowania jaskiń na podstawie pożądanej gęstości.

Chunki z samym modelem terenu (objętość wejściowa)

Dla każdego chunka terenu przygotowano jego wolumetryczny model. Na podstawie mapy wysokości, bloki poniżej powierzchni zostały wypełnione kamieniem, a bloki powyżej powierzchni zostały wypełnione powietrzem. Model terenu służy jako aproksymacja kształtu terenu, pozwalając modelowi skupić się na dopracowaniu i dodawaniu szczegółów, takich jak jaskinie, warstwy i zmiany powierzchni.

5.4 Kodowanie i normalizacja danych

Kodowanie danych jest niezbędne do przygotowania danych w formacie odpowiednim do trenowania sieci neuronowej.

Identyfikatory typu bloku zostały przekonwertowane na wektory kodowane metodą one-hot. W przypadku sześciu typów bloków, każdy jest reprezentowany przez sześćelementowy wektor z wartością 1 przy indeksie odpowiadającym typowi bloku i zera w innych miejscach. Kodowanie wszystkich bloków można znaleźć w Tabeli 1.

Przykład: Blok kamienia (identyfikator 3) jest kodowany jako [0, 0, 0, 1, 0, 0].

Ciągłe cechy wejściowe, takie jak mapy wysokości i gęstości jaskiń, zostały znormalizowane do zakresu od 0 do 1, aby ułatwić szybszą konwergencję podczas nauczania. Mapy wysokości podzielono przez maksymalną możliwą wysokość (320). Gęstość jaskiń podzielono przez maksymalną możliwą gęstość jaskiń na podstawie fragmentu chunka, w którym one występują.

Ostateczne kształty tablic danych użytych do uczenia to:

- Mapy wysokości: (N, 16, 16), gdzie N to liczba próbek.
- Gęstości jaskiń (N, 1)
- Modele terenu: (N, 16, 320, 16, 6) po kodowaniu one-hot.
- Rzeczywiste chunki: (N, 16, 320, 16, 6) po kodowaniu one-hot.
- Chunki z zapadniętymi jaskiniami: (N, 16, 320, 16, 6) po kodowaniu one-hot.

5.5 Uzasadnienie

Bloki wodne zostały celowo wykluczone ze zbioru danych. W grze Minecraft woda jest zazwyczaj generowana przez zastąpienie bloków powietrza poniżej poziomu morza, określonego przez grę. Włączenie bloków wodnych mogłoby wprowadzić dodatkową złożoność bez znaczących korzyści dla reprezentacji kształtu terenu. Skupiając się na blokach, które są głównymi budulcami terenu i powietrzem, model może skuteczniej uczyć się podstawowej struktury terenu.

Zmniejszenie liczby typów bloków i wykluczenie złożonych struktur, takich jak drzewa i wioski, pozwala modelowi skupić się na uczeniu się terenu, warstw i formacji jaskiń. To uproszczenie jest kluczowe w zarządzaniu zasobami obliczeniowymi i czasem trenowania, szczególnie podczas pracy z dużymi tablicami danych 3D.

Chociaż zbiór danych jest uproszczony, metodologia pozwala na przyszłą rozbudowę. Dodatkowe typy bloków lub odmiany biomu można włączyć, rozszerzając listę typów bloków i odpowiednio dostosowywać procesy ekstrakcji i kodowania danych.

6. Trenowanie modelu

Faza uczenia modelu sieci neuronowej jest krytycznym krokiem w opracowaniu podejścia opartego na sztucznej inteligencji do generowania światów Minecraft. W tym rozdziale omówiono proces uczenia, w tym różne badane warianty modelu, zastosowane funkcje strat, uzyskane wyniki i uzasadnienie wyboru najbardziej obiecującego modelu do dalszej analizy.

6.1 Przygotowania do uczenia sieci

W tym etapie pracy do uczenia został wykorzystany zestaw danych, który składa się z 4096 chunków świata Minecraft wyodrębnionych i sformatowanych z gry. Każdy fragment jest reprezentowany jako tablica 3D o wymiarach $16 \times 320 \times 16$, w której każdy wymiar odpowiada kolejno za szerokość (X), wysokość (Y) i głębokość (Z). Dane obejmują informacje o typach bloków, mapach wysokości terenu i gęstości jaskiń.

Dane wyjściowe są kodowane metodą one-hot, aby ułatwić klasyfikację bloków podczas trenowania sieci.

Modele trenowano przy użyciu platformy *Google Colab*, która umożliwia dostęp do wysoce wydajnych kart graficznych w chmurze, a w szczególności karty NVIDIA A100. Ta karta zapewnia znaczącą moc obliczeniową do trenowania dużych sieci neuronowych. Trening przeprowadzono na czterech epokach dla każdej wersji modelu z krokiem trenującym na podzbiorze o wielkości 8, aby zrównoważyć ograniczenia pamięci i szybkości uczenia.

Wykorzystane środowisko wykonawcze „A100 GPU” które bazuje na maszynach oznakowanych „a2-highgpu-1g” w *Google Cloud*. [38]

Specyfikacja maszyny:

- GPU: **NVIDIA A100 40 GB**
 - o Przepustowość pamięci: 1555 GB/s
 - o Tensor Float 32: 156 TF (*na tej precyzji były trenowane modele*)
- VM memory: **85 GB**
- vCPU count: **12**

6.2 Funkcje straty

Generator

Celem generatora jest produkowanie podobnych chunków do tych w grze Minecraft, które w najlepszym przypadku są nieodróżnialne od rzeczywistych danych. Funkcja strat generatora składa się z dwóch głównych komponentów:

1. **Adversarial Loss** (L_{adv}): Mierzy, jak dobrze generator oszukuje dyskryminator. Oblicza się ją, używając binarnej entropii krzyżowej (ang. *Binary cross entropy*) między predykcją dyskryminatora na wygenerowanych danych a docelową etykietą oznaczającą prawdziwe dane (1).

$$L_{adv} = \text{CrossEntropy}(1, D(G(z)))$$

2. **Content Loss** ($L_{content}$): Zachęca generator do generowania wyników podobnych danych rzeczywistych. Jest obliczana przy użyciu **kategorycznej** entropii krzyżowej między rzeczywistymi chunkami a wygenerowanymi chunkami, ważonej przez wagi

specyficzne dla klas, aby rozwiązać nierówności w uczeniu między klasami.

$$L_{content} = \sum_i w_i \cdot CrossEntropy(Real_i, Generated_i)$$

Gdzie w_i to wagi klas dla każdego typu bloku.

Dodatkowo jest jeszcze opcjonalny komponent, *Cave Map Loss* (L_{cave}) został wprowadzony w niektórych modelach, z nadzieją nakierowania generatora, a konkretnie warstwy *Cave Map Layer*, w tworzeniu struktur jaskiń. Jest obliczany przy użyciu binarnej entropii krzyżowej między rzeczywistą mapą jaskiń a wygenerowaną mapą jaskiń.

Całkowita strata generatora:

$$L_{gen} = L_{adv} + L_{content} + L_{cave}$$

Dyskryminator

Dyskryminator ma na celu rozróżnienie rzeczywistych chunków od wygenerowanych. Jego funkcja straty obejmuje:

1. Real Loss: Binarna entropia krzyżowa pomiędzy predykcją dyskryminatora na rzeczywistych danych a docelową etykietą oznaczającą prawdziwe dane (1).

$$L_{real} = CrossEntropy(1, D(Real))$$

2. Fake Loss: Binarna entropia krzyżowa pomiędzy predykcją dyskryminatora na danych wygenerowanych a etykietą oznaczającą fałszywe dane (0).

$$L_{fake} = CrossEntropy(1, D(G(z)))$$

3. Gradient Penalty: Zapewnia płynność wyjścia dyskryminatora względem jego wejścia, obliczanej przy użyciu podejścia Wasserstein GAN z karą gradientową (WGAN-GP). [39]

Całkowita strata dyskryminatora:

$$L_{disc} = L_{real} + L_{fake} + \lambda \cdot L_{gp}$$

Gdzie λ jest współczynnikiem wagowym dla kary gradientowej.

6.3 Warianty modeli

Eksperymentowano z czterema różnymi wariantami modelu, aby zidentyfikować najskuteczniejsze podejście do generowania chunków świata. Główne różnice między tymi modelami leżą w reprezentacji wejściowej i uwzględnieniu funkcji straty mapy jaskiń (Cave Map Loss).

Model 1: Embedding + Cave Map Loss

Dane wejściowe są przetwarzane przez warstwę osadzania (z ang. *embedding*), co pozwala modelowi na bardziej złożoną reprezentację danych. Funkcja straty generatora dodatkowo uwzględnia funkcję straty mapy jaskiń, która wpływa na poprawność generowania jaskiń.

Model 2: Embedding

Podobnie jak w Modelu 1, dane wejściowe są przetwarzane przez warstwę osadzania, ale w tym przypadku funkcja straty nie bierze pod uwagę funkcję straty mapy jaskiń. Model optymalizuje jedynie na podstawie dyskriminatora bez dodatkowego nacisku na generowanie jaskiń.

Model 3: One-Hot + Cave Map Loss

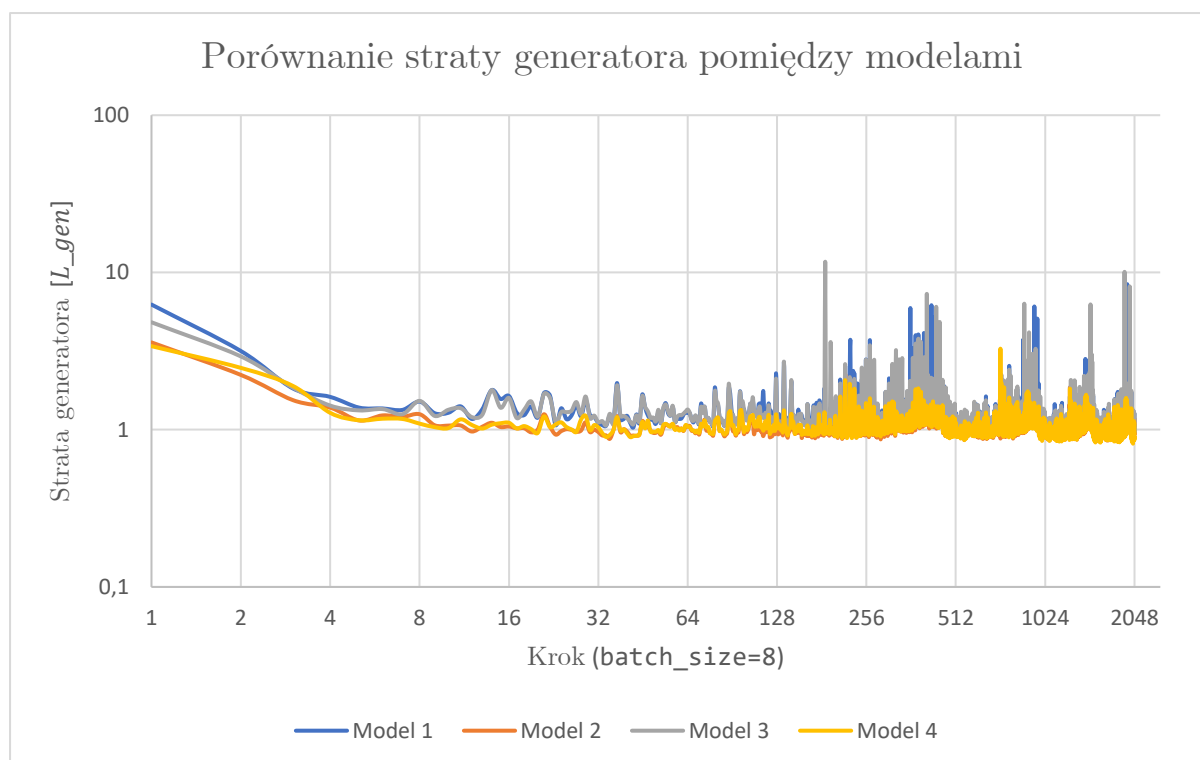
Dane wejściowe są wstępnie zakodowane jako one-hot, co oznacza, że każda wartość jest reprezentowana przez wektor. Funkcja straty generatora uwzględnia funkcję straty mapy jaskiń, co ma na celu poprawienie generowania jaskiń.

Model 4: One-Hot

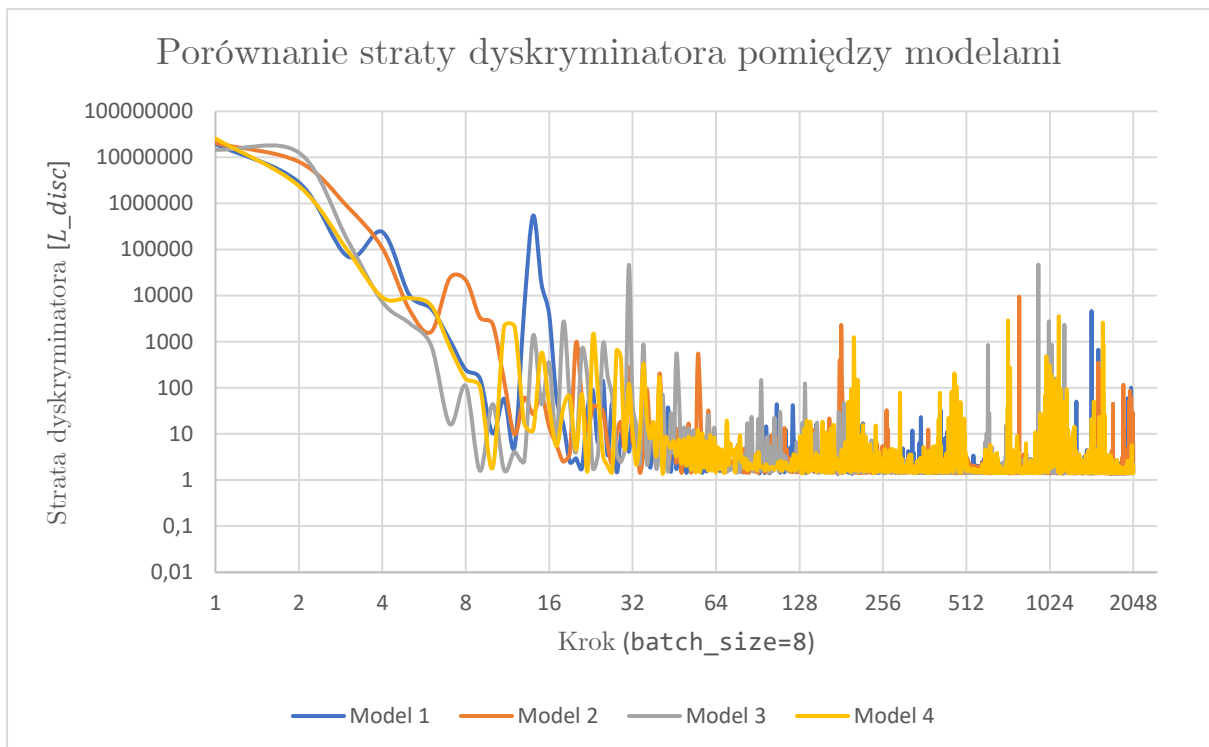
Dane wejściowe są również zakodowane jako one-hot, ale funkcja straty nie uwzględnia funkcji straty mapy jaskiń. Model optymalizuje jedynie na podstawie standardowych kryteriów, bez dodatkowego wpływu na generowanie jaskiń.

6.4 Wyniki uczenia sieci

Każdy model był trenowany przez cztery epoki, dla wszystkich danych trenujących na każdej epoce. Straty generatora i dyskriminatora rejestrowano w każdym kroku trenującymi, jeden krok trenował sieć na 8 danych z zestawu. Z kolei statystyki odnośnie dystrybucji bloków były generowane pod koniec każdej epoki.



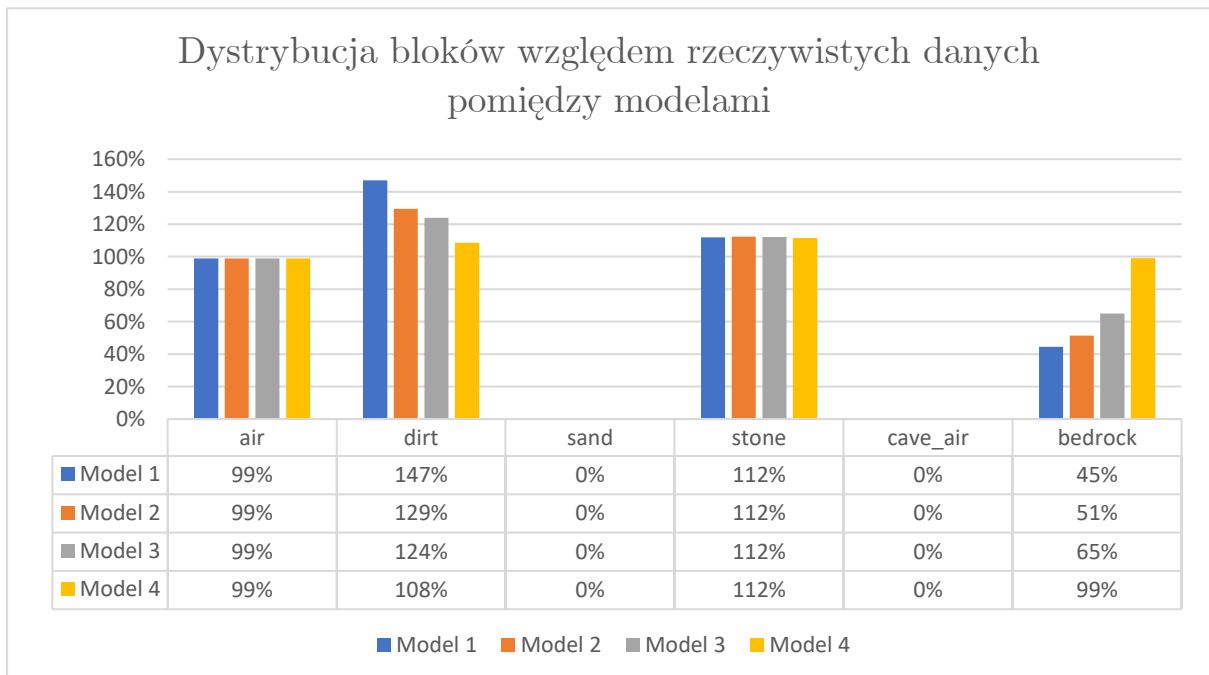
Rysunek 10. Wykres porównujący straty generatora pomiędzy modelami przez cały okres uczenia (4 epoki, 2048 kroków).



Rysunek 11. Wykres porównujący straty dyskryminatora pomiędzy modelami przez cały okres uczenia (4 epoki, 2048 kroków).

Porównanie dystrybucji bloków

Wygenerowane chunki zostały przeanalizowane w celu porównania dystrybucji typów bloków z rzeczywistymi danymi. Wartość bliska 100% wskazuje, że model wygenerował podobną liczbę bloków tego typu, jak w rzeczywistych danych.



Rysunek 12. Wykres dystrybucji bloków względem rzeczywistych danych pomiędzy modelami.

Obserwacje

- Generowanie powietrza jaskiniowego (*Cave air*): Wszystkie modele nie wygenerowały bloków powietrza jaskiniowego, co wskazuje na trudności w uczeniu się struktur jaskiń obecnych w rzeczywistych danych.
- Dystrybucja bloków Dirt i Bedrock: Modele bez nałożonej straty mapy jaskiń (Cave Map Loss) miały tendencję do generowania dystrybucji bloku Bedrock bliższej rzeczywistym danym. Nadreprezentacja bloku Dirt i mała reprezentacja bloku Bedrock w niektórych modelach sugerują brak równowagi w uczeniu się tych klas.
- Wpływ straty mapy jaskiń: Włączenie straty mapy jaskiń nie poprawiło znacząco generowania struktur jaskiniowych. Modele ze stratą Cave Map Loss nie wykorzystały skutecznie dodatkowych wytycznych.
- Reprezentacja danych wejściowych: **Modele wykorzystujące kodowanie One-Hot dla danych wejściowych, uczyły się szybciej** i wykazywały nieco lepszą dystrybucję bloków w porównaniu z modelami wykorzystującymi warstwę osadzenia (embedding layer).

Sieć najwyraźniej uczy się wszystkiego poza strukturami jaskiń. Dyskryminator ma problemy ze zrozumieniem niektórych aspektów świata. Próba nakierowania modelu na wygenerowanie trójwymiarowej mapy jaskiń nie doprowadziła do wykorzystania przez modele tych informacji do wygenerowania struktur. W tym przypadku nakierowanie sieci odbywało się przy użyciu binarnej entropii krzyżowej (ponieważ w mapach jest tylko jeden kanał danych, jest jaskinia, nie ma jaskini). Opierało się to na porównaniu rzeczywistej mapy jaskiń z wygenerowaną mapą jaskiń.

Dalsze eksperymenty z różnymi podejściami, takimi jak stosowanie funkcji strat do określonych aspektów jaskiń, nie przyniosły znaczących ulepszeń. Sugeruje to, że zadanie jest bardziej złożone niż początkowo zakładano lub sama architektura w aktualnej wersji może nie nadawać się do przechwytywania struktur jaskiń. Podejrzewa się, że model mógł działać lepiej, gdyby pewne części generatora, w szczególności warstwa odpowiedzialna za kształt jaskiń, została przeniesiona do osobnego modelu i wytrenowana pod to konkretne zadanie.

Na podstawie powyższych wniosków, **Model 4** został wybrany do dalszego uczenia i dogłębnej analizy. Jego wydajność wskazuje, że ma lepszą zdolność do uczenia się ogólnej struktury chunków Minecrafta bez dodatkowej złożoności funkcji straty mapy jaskiń. Przyszłe prace będą koncentrować się na zwiększeniu zdolności tego modelu do generowania struktur jaskiń, możliwie poprzez modyfikację architektury lub wprowadzenie nowych funkcji straty, specjalnie zaprojektowanych do uchwycenia złożoności jaskiń.

7. Metryki oceny

Ocena jakości generowanych treści jest ważnym aspektem w tworzeniu modeli generatywnych, szczególnie w kontekście złożonych struktur, takich jak świat Minecraft. Tradycyjne metryki stosowane w generowaniu obrazów, takie jak Inception Score [40] lub Fréchet Inception Distance [41], nie mają bezpośrednio zastosowania na danych 3D opartych o *woksele*. Dlatego konieczne są specjalistyczne metryki, aby ocenić, jak bardzo wygenerowane światy przypominają rzeczywiste światy Minecrafta i określić ilościowo określone cechy strukturalne chunków.

W tym rozdziale omawiamy metryki ewaluacyjne wykorzystane do oceny wydajności opartego na sztucznej inteligencji generatora świata Minecrafta opracowanego w tej pracy. Skupiamy się na trzech podstawowych metrykach:

- **Odległość Levenshteina:** Mierzy podobieństwo między wygenerowanymi fragmentami a rzeczywistymi fragmentami w oparciu o minimalną liczbę edycji pojedynczych bloków wymaganych do przekształcenia jednego fragmentu w drugi.
- **Tile Pattern Kullback-Leibler Divergence (TPKL-Div):** Ocenia podobieństwo lokalnych wzorców między wygenerowanymi i rzeczywistymi chunkami poprzez porównanie rozkładów małych wzorców bloków 3D.
- **Średnia różnica wielkość gradientu (MGD):** Porównuje gradienty map wysokości między wygenerowanymi a rzeczywistymi chunkami, mierząc różnicę w zmianach wysokości wzdłuż osi X i Z. MGD analizuje podobieństwo kształtów terenu, uwzględniając płynność i naturalność przejść wysokości.

Metryki Levenshteina i TPKL-Div zostały zaadoptowane z pracy „World-GAN: A Generative Model for Minecraft Worlds” [42], umożliwiając bezpośrednie porównanie wydajności modelu z istniejącymi podejściami.

7.1 Dystans Levenshteina

Dystans Levenshteina, znany również jako dystans edycji, jest metryką łańcuchową do pomiaru różnicy między dwiema sekwencjami. Jest ona definiowana jako minimalna liczba jednoznakowych zmian (wstawień, usunięć lub podstawień) wymaganych do zmiany jednej sekwencji w drugą [43]. W kontekście chunków 3D w grze Minecraft koncepcję tę można rozszerzyć, aby zmierzyć różnicę między dwoma chunkami, traktując je jako sekwencje typów bloków.

Aby zastosować dystans Levenshteina na chunkach Minecrafta, każdy 3D chunk jest spłaszczany do jednowymiarowej sekwencji poprzez łączenie typów bloków w spójnej kolejności (np. przechodzenie przez chunk w określonej kolejności x, y, z). Spłaszczone (ang. *flatten*) sekwencje rzeczywistego chunka i wygenerowanego chunka są porównywane przy użyciu algorytmu Levenshteina. Obliczana jest minimalna liczba edycji bloku potrzebna do przekształcenia wygenerowanego chunka w chunk rzeczywisty.

Tabela 2. Przykład obliczania dystansu Levenshteina między dwoma ciągami (chunkami).

		Air [1]	Air [2]	Bedrock [3]	Stone [4]
	0	1	2	3	4
Dirt [1]	1	1	2	3	4
Cave air [2]	2	2	2	3	4
Bedrock [3]	3	3	3	2	3
Sand [4]	4	4	4	3	3
Dystans Levenshteina					3
Podobieństwo					25%

Podobieństwo:

$$\frac{|\text{Lev}_{a,b} - \text{length}(a)|}{\text{length}(a)} \cdot 100\%$$

Gdzie:

- $\text{Lev}_{a,b}$, to dystans Levenshteina między ciągiem a i b (w tym przypadku, sekwencje typów bloku)
- $\text{length}(a)$, to długość ciągu a

Dystans Levenshteina:

$$\text{Lev}_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0 \\ \min \begin{cases} \text{Lev}_{a,b}(i-1, j) + 1 \\ \text{Lev}_{a,b}(i, j-1) + 1 \\ \text{Lev}_{a,b}(i-1, j-1) + k \end{cases} & \text{otherwise.} \end{cases}$$

Gdzie:

- k równa się 0 jeżeli $a_i = b_i$, w innym przypadku 1

Pomimo swoich ograniczeń, dystans Levenshteina zapewnia mierzalną miarę ogólnego podobieństwa między wygenerowanymi i rzeczywistymi chunkami. Jest przydatna do wychwytywania różnic w składzie bloków i może podkreślać rozbieżność w ogólnej strukturze chunków.

7.2 Tile Pattern Kullback-Leibler Divergence (TPKL-Div)

Metryka zaprojektowana do porównywania rozkładów lokalnych wzorców (kafelków) między wygenerowanymi i rzeczywistymi danymi. Mierzy rozbieżność między rozkładami prawdopodobieństwa wzorców wyodrębnionych z obu zestawów danych, zapewniając wgląd w zdolność modelu do replikowania lokalnych struktur znalezionych w rzeczywistych światach Minecraft. [44]

Dywergencja Kullbacka-Leiblera (KL-Divergence, zwana też entropią względną lub relatywną entropią) to miara z teorii informacji, która określa mierzalnie, jak jeden rozkład prawdopodobieństwa odbiega od drugiego, oczekiwanego rozkładu prawdopodobieństwa [45]. Jest ona zdefiniowana jako:

$$D_{\text{KL}}(P \parallel Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

Gdzie P i Q są rozkładami prawdopodobieństwa dla tego samego zbioru zdarzeń.

Dywergencja KL jest asymetryczna, co oznacza, że $D_{\text{KL}}(P \parallel Q) \neq D_{\text{KL}}(Q \parallel P)$. Ta asymetria może stanowić problem, gdy chcemy uzyskać symetryczną miarę rozbieżności między dwoma rozkładami.

Symetryzowana dywergencja KL

Aby rozwiązać problem asymetrii, powszechnym [45] podejściem jest obliczenie ważonej sumy rozbieżności KL w obu kierunkach. W kontekście TPKL-Div definiujemy symetryczną rozbieżność KL jako:

$$D_{\text{sym}}(P, Q) = w \cdot D_{\text{KL}}(Q \parallel P) + (1 - w) \cdot D_{\text{KL}}(P \parallel Q)$$

Gdzie w jest parametrem wagi w zakresie od 0 do 1, który określa wkład każdego kierunku. Gdy $w = 0.5$, rozbieżność staje się średnią dwóch rozbieżności KL, zapewniając symetryczną miarę.

Zastosowanie we wzorcach kafelkowych (Tile Patterns)

1. Wyodrębnianie wzorów kafelków: Dla każdego chunka wyodrębniamy wszystkie możliwe wzorce kafelków (małe bloki 3D) o określonym rozmiarze (np. $5 \times 5 \times 5$), poprzez przesuwanie okna przez chunk.
2. Zliczanie wzorców: Zliczamy wystąpienia każdego unikalnego wzorca zarówno w rzeczywistych, jak i wygenerowanych fragmentach, uzyskując liczbę wzorców dla każdego zbioru danych.
3. Obliczanie prawdopodobieństw: Obliczamy rozkłady prawdopodobieństwa P (z rzeczywistych chunków) i Q (z wygenerowanych chunków) poprzez normalizację liczby wzorców w stosunku do całkowitej liczby wzorców.
4. Obliczanie dywergencji KL: Obliczamy $D_{\text{KL}}(P \parallel Q)$ i $D_{\text{KL}}(Q \parallel P)$ przy użyciu obliczonych prawdopodobieństw.
5. Obliczanie dywergencji symetrycznej: Używając parametru wagi w (oznaczonego jako waga w kodzie), obliczamy symetryczną dywergencję KL, $D_{\text{sym}}(P, Q)$.
6. Agregacja: Obliczamy TPKL-Div dla każdego wygenerowanego chunka i agregujemy wyniki (obliczamy średnią i wariancję) dla wszystkich wygenerowanych chunków.

7.3 Średnia różnica wielkość gradientu (MGD)

Mean Gradient Magnitude Difference (MGD) to metryka wprowadzona w celu porównania kształtów map wysokości generowanych przez model z tymi znalezionymi w rzeczywistych danych Minecrafta. Wskaźnik ten ocenia podobieństwo gradientów terenu między wygenerowanym a rzeczywistym światem, koncentrując się na płynności i przepływie zmian wysokości w terenie, zamiast po prostu porównywać wysokości bezwzględne.

MGD jest obliczany poprzez pomiar gradientu wzdłuż osi X i Z dla każdego punktu na mapie wysokości, przy użyciu odległości euklidesowej między sąsiednimi punktami. Dla każdej pary rzeczywistych i wygenerowanych map wysokości obliczana jest różnica między ich wielkościami gradientu, a następnie uśredniana dla całej mapy. Zapewnia to wgląd w to, jak dobrze model oddaje ogólny kształt i przepływ terenu.

Wzór na wielkość gradientu jest następujący:

$$\Delta h = \sqrt{(h_{x+1,z} - h_{x,z})^2 + (h_{x,z+1} - h_{x,z})^2}$$

Gdzie $h_{x,z}$ to wartość wysokości na współrzędnych x i z .

Następnie MGD oblicza się jako:

$$\text{MGD} = \frac{1}{n} \sum_{i=1}^n |\Delta h_{real}(i) - \Delta h_{gen}(i)|$$

Gdzie:

- Δh_{gen} to wielkość gradientu dla wygenerowanej mapy wysokości,
- Δh_{real} to wielkość gradientu dla rzeczywistej mapy wysokości,
- n to liczba porównywanych punktów na mapie.

Po obliczeniu MGD dla każdej próbki w zbiorze danych, średnia i wariancja różnic we wszystkich próbkach są obliczane w celu zapewnienia ogólnej oceny podobieństwa między wygenerowanymi i rzeczywistymi mapami wysokości. Ostateczny wynik jest wyrażany jako procent podobieństwa, znormalizowany przez maksymalny możliwy błąd (tj. maksymalną różnicę wysokości 320 w grze):

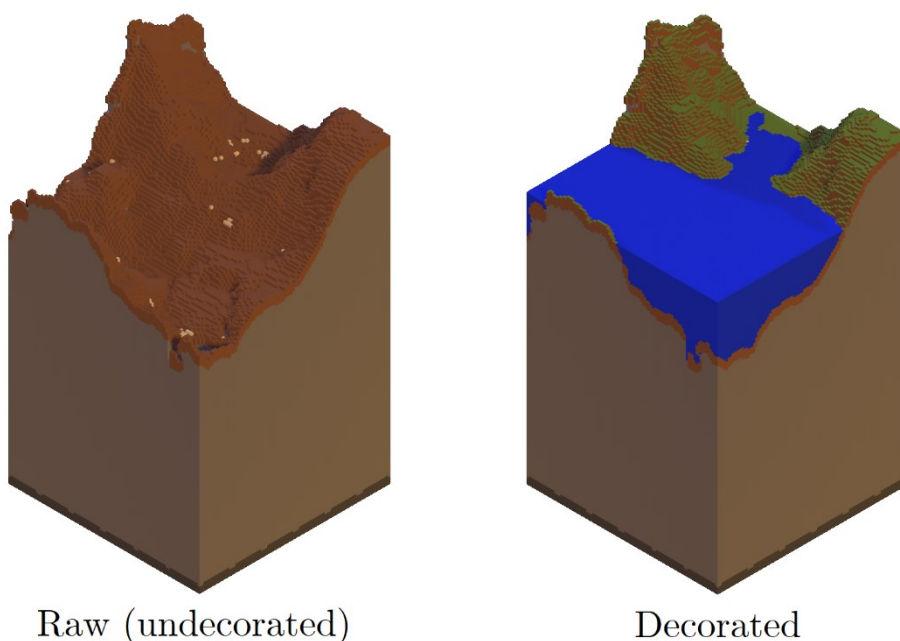
$$\text{Podobieństwo (\%)} = \left(1 - \frac{\text{MGD}}{320}\right) \cdot 100$$

Metryka ta jest szczególnie przydatna w identyfikowaniu subtelnych rozbieżności w kształtowaniu terenu, takich jak nienaturalne płaskie obszary lub ostre przejścia, które mogą nie być oczywiste, patrząc tylko na poszczególne wartości wysokości.

8. Analiza wyników

Niniejszy rozdział przedstawia wyniki oceny ostatecznego modelu, który wykorzystywał kodowanie one-hot na danych wejściowych i nie stosował straty `cave_loss` podczas uczenia. Model został wytrenowany przy użyciu 8192 próbek w ciągu ośmiu epok, z których każda przechodziła przez cały zbiór danych. Chociaż proces trenowania nie wykazał znaczących postępów po początkowych etapach, wyniki ewaluacji modelu dostarczają cennych informacji na temat jego mocnych i słabych stron.

Wygenerowany świat



Rysunek 13. Przykład wygenerowanego świata za pomocą opracowanego modelu (Model 4). Świat o rozmiarach 6x6 chunków. Po lewej: surowy świat, jest to bezpośredni wynik modelu. Po prawej: "udekorowany" świat, został wypełniony woda do poziomu morza i pokryty trawą.

8.1 Wyniki TPKL-Div

Tile Pattern Kullback-Leibler Divergence (TPKL-Div) został użyty do porównania wygenerowanych próbek z rzeczywistymi danymi Minecrafta pod względem lokalnych wzorców bloków. Ta metryka ocenia, jak dokładnie model odwzorowuje rozkład wzorców w rzeczywistych chunkach Minecrafta. Model został przetestowany przy użyciu wzorców o rozmiarze $5 \times 5 \times 5$ i $10 \times 10 \times 10$.

Tabela 3. Szczegółowe wyniki TPKL-Div dla opracowanego modelu.

Chunk	Rozmiar wzorca 5x5x5 (TPKL-Div)	Rozmiar wzorca 10x10x10 (TPKL-Div)
1	2,8505	4,5362
2	2,8478	4,5324
3	2,8431	4,5513
4	2,8501	4,5372
5	2,8813	4,5528

6	2,8636	4,5427
7	2,8676	4,5514
8	2,8551	4,5556
Średnia	2,8574	4,5449
Wariancja	0,0001396	$6,97398 \cdot 10^{-5}$
Ogólna średnia	3,7012	

Tabela 4. Porównanie wyniku TPKL-Div opracowanego modelu z innymi modelami.

Model	Średnia TPKL-Div (dla biotopu <i>plains</i>)
Opracowany model	3,70
World-GAN	23,05
TOAD-GAN	22,79

Niższy wynik TPKL-Div wskazuje, że wygenerowane wzorce ściśle pasują do tych znalezionych w rzeczywistych danych. Przy średnim wyniku TPKL-Div wynoszącym **3,7**, opracowany model przewyższa zarówno World-GAN [46], jak i TOAD-GAN [47]. Sugeruje to, że model jest bardzo skuteczny w replikowaniu lokalnych wzorców, szczególnie w biotopie równin. Niewielka rozbieżność z rzeczywistymi danymi jest prawdopodobnie związana z trudnościami modelu w generowaniu jaskiń, które są bardziej złożone i wprowadzają dodatkową zmienność do danych.

8.2 Wyniki dystansu Levenshteina

Dystans Levenshteina został wykorzystany do pomiaru zmienności i różnic między wygenerowanymi chunkami poprzez porównanie ich w wielu próbkach. Średnia odległość Levenshteina między każdą wygenerowaną próbką a wszystkimi innymi próbkami jest pokazana poniżej:

Tabela 5. Szczegółowe wyniki dystansu Levenshteina dla opracowanego modelu.

Próbka (<i>chunk</i>)	1	2	3	4	5	6	7	8	Średnia odległość Levenshteina między próbką N a wszystkimi próbkami
1	0	162	166	181	150	164	148	181	144,00
2	162	0	143	150	145	149	160	159	133,50
3	166	143	0	161	144	153	136	155	132,25
4	181	150	161	0	157	164	172	150	141,88
5	150	145	144	157	0	163	138	149	130,75
6	164	149	153	164	163	0	174	174	142,63
7	148	160	136	172	138	174	0	157	135,63
8	181	159	155	150	149	174	157	0	140,63
Wariancja między wszystkimi próbkami									2829,16
Ogólna średnia									137,66

Tabela 6. Porównanie dystansu Levenshteina dla opracowanego modelu z innymi modelami.

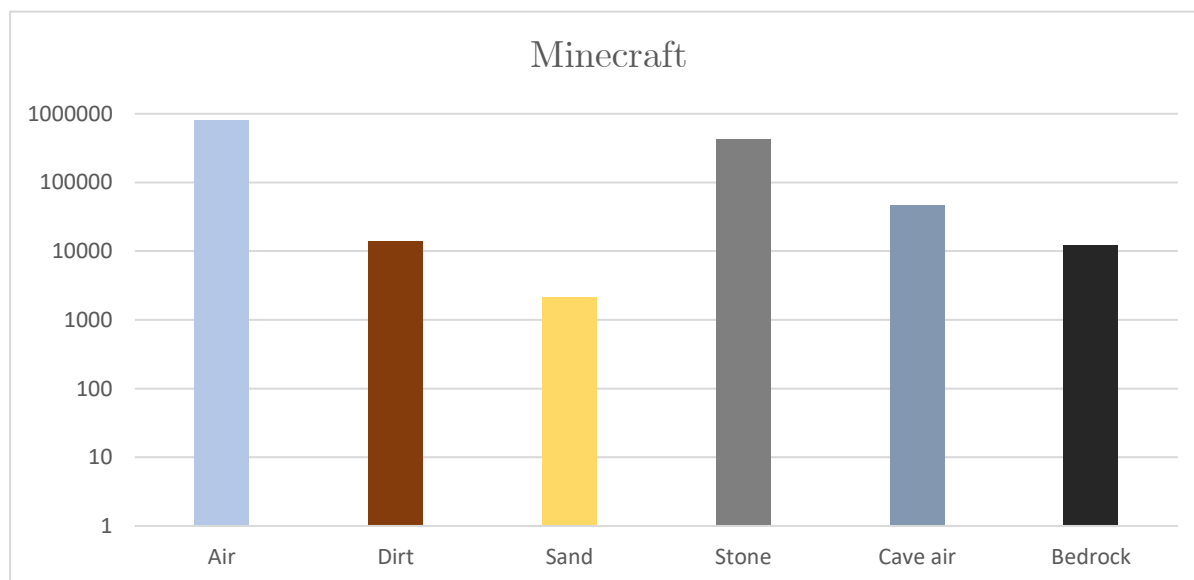
Model	Średni dystans Levenshteina (dla biomu <i>plains</i>)
Opracowany model	137.66
World-GAN	5314.43
TOAD-GAN	3895.96

Niższy dystans Levenshteina wskazuje na mniejszą zmienność wśród wygenerowanych próbek. Przy średniej odległości Levenshteina wynoszącej **137,66**, opracowany model wykazuje znacznie niższą zmienność w porównaniu zarówno do World-GAN, jak i TOAD-GAN. Chociaż sugeruje to, że model generuje bardzo spójne chunki, podkreśla to również potencjalną słabość: model może być nadmiernie dopasowany do danych treningowych, nie wprowadzając wystarczającej zmienności, szczególnie w obszarach takich jak generowanie jaskiń, gdzie oczekuje się bardziej zróżnicowanych struktur.

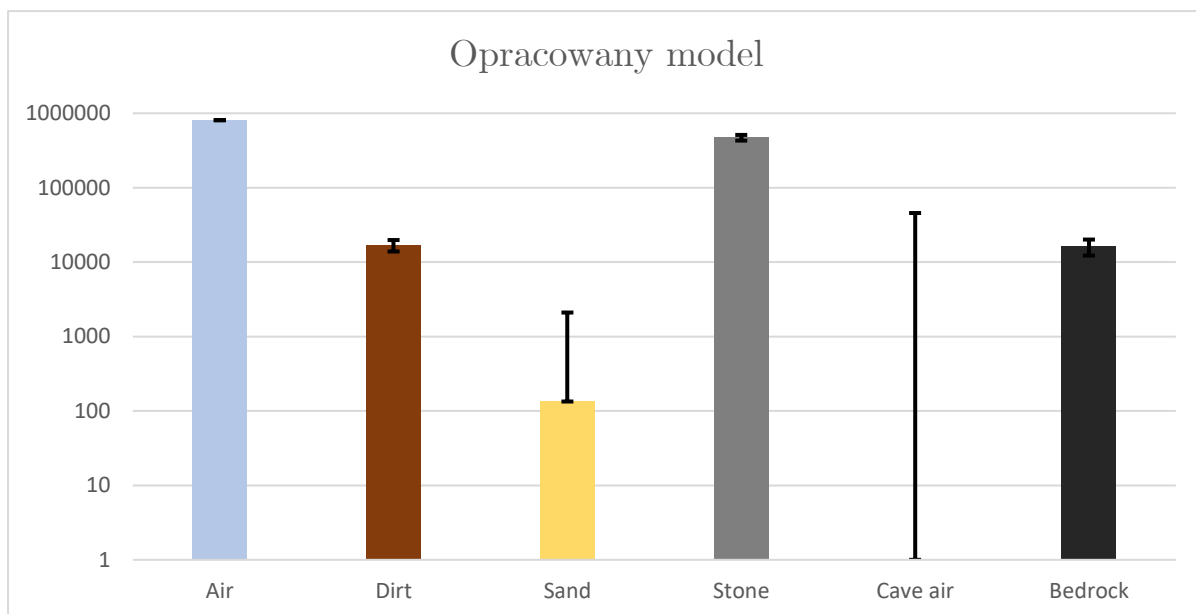
Brak zmienności jest prawdopodobnie spowodowany albo zbyt silnym dyskriminatorem, zmuszającym generator do zbyt dokładnej replikacji rzeczywistych danych, albo szumem, który nie wprowadza wystarczającej losowości. Dodatkowo niezdolność do generowania jaskiń, odzwierciedlona w brakujących blokach *Cave air*, może być kolejnym czynnikiem.

8.3 Porównanie histogramów bloków

Histogramy bloków przedstawiają liczbę każdego typu bloków wygenerowanych przez model w porównaniu z rzeczywistymi danymi Minecrafta.



Rysunek 14. Histogram bloków w rzeczywistych chunkach z gry Minecraft na przestrzeni 16 chunków.



Rysunek 15. Histogram bloków wygenerowanych chunków przez opracowany model na przestrzeni 16 chunków i uwzględnionym błędem w stosunku do histogramu z Rysunek 14.

8.4 Porównanie map terenów

Użyto metryki Mean Gradient Magnitude Difference (MGD), aby ocenić, jak dobrze model odwzorowuje kształty terenu w wygenerowanych chunkach w porównaniu do rzeczywistych danych Minecrafta. MGD mierzy różnicę w nachyleniu (gradientach) map wysokości wzdłuż osi X i Z co pozwala dokładniej ocenić podobieństwo kształtu terenu.

Tabela 7. Szczegółowe wyniki MGD między wygenerowanymi a rzeczywistymi chunkami w opracowanym modelu.

Para (real, generated)	Średnia różnica gradientu (MGD)
1	0,5815
2	0,4068
3	0,2391
4	0,4302
5	0,3318
6	0,0670
7	0,1266
8	0,2849
9	0,3188
10	0,0651
11	0,0374
12	0,0717
13	0,0829
14	0,0855
15	0,0000
16	0,0348
Ogólna różnica gradientów	0,1978
Wariancja	0,0287
Średnia podobieństwa	99,94%

Ogólna różnica gradientów wynosi **0,175**, a wariancja to **0,022**. Średnia podobieństwa, obliczona na podstawie MGD, wynosi **99.95%**, co wskazuje na bardzo wysokie podobieństwo kształtu terenu między wygenerowanymi chunkami a rzeczywistymi mapami wysokości z Minecrafta.

8.5 Wnioski

Rozkład bloków wygenerowany przez opracowany model jest w większości zgodny z rzeczywistymi danymi, prawidłowo odwzorowując bloki *Air*, *Dirt*, *Stone* oraz *Bedrock*. Chociaż bloki *Sand* występują najrzadziej, model wygenerował je, choć w mniejszych ilościach. Jednakże model nie generował bloków *Cave air*, co sugeruje problem z generowaniem jaskiń. Informacja ta wyklucza wpływ liczebności bloków na to, czy są generowane przez model.

Brak jaskiń wskazuje na problem z aktualnym podejściem, który może wynikać z niewystarczającej liczby danych dotyczących jaskiń lub z niewłaściwego sposobu modelowania tej cechy. Mimo to model radzi sobie doskonale z generowaniem powierzchni terenu i ogólnym odwzorowaniem struktury świata.

Wyniki pokazują, że opracowany model doskonale odwzorowuje powierzchniowe struktury świata Minecrafta, szczególnie biom „równiny”, co znajduje odzwierciedlenie w niskich wynikach TPKL-Div. Model ma jednak trudności z generowaniem zróżnicowanych struktur, co uwiadamia niski dystans Levenshteina i brak jaskiń. Przyszłe badania powinny skupić się na poprawie generowania jaskiń oraz zwiększeniu zmienności generowanych chunków poprzez dostosowanie siły dyskriminatora lub wprowadzenie większej ilości szumu.

9. Podsumowanie

Celem niniejszej pracy było opracowanie opartego na sztucznej inteligencji podejścia do generowania światów w grze Minecraft, które pozwala na większą kontrolę nad procesem generacji i personalizację środowiska gry. Wykorzystano model bazujący na sieciach neuronowych, w szczególności na Generative Adversarial Networks (GAN), z architekturą U-Net oraz mechanizmami uwagi (z ang. *Attention*).

Przeprowadzono szczegółowe przygotowanie danych, obejmujące ekstrakcję, przetwarzanie informacji z gry, redukcję liczby typów bloków do najbardziej istotnych oraz kodowanie danych w formacie odpowiednim dla sieci neuronowej. W procesie trenowania modelu zbadano różne warianty, w tym wykorzystanie kodowania one-hot oraz warstw osadzenia (z ang. *embedding*), a także wpływ funkcji straty związanej z mapą jaskiń.

Badania potwierdziły, że sztuczna inteligencja może być skutecznym narzędziem w generowaniu światów w grach komputerowych. Opracowany model z powodzeniem generował spójne i realistyczne struktury powierzchniowe świata Minecraft, odwzorowując kształt terenu i rozmieszczenie podstawowych typów bloków, takich jak kamień, ziemia czy piasek. Niskie wartości metryki Tile Pattern Kullback-Leibler Divergence (TPKL-Div) wskazują na wysoką zgodność lokalnych wzorców z rzeczywistymi danymi z gry.

Jednakże model napotkał trudności w generowaniu struktur jaskiniowych. Pomimo wprowadzenia funkcji straty związanej z mapą jaskiń oraz różnych wariantów architektury, model nie nauczył się efektywnie generować jaskiń, co przejawiało się brakiem bloków powietrza jaskiniowego w wygenerowanych chunkach. Sugeruje to, że złożoność tych struktur wymaga dalszych badań i być może większych zmian w architekturze modelu.

Ponadto zauważono ograniczoną zmienność w generowanych chunkach, co może wskazywać na nadmierne dopasowanie modelu do danych treningowych lub niewystarczająca ilość wprowadzonego zaszumienia w procesie generacji. Niskie wartości odległości Levenshteina między wygenerowanymi próbkami potwierdzają tę obserwację.

Przyszłe prace

Aby poprawić obecne rozwiązanie i rozwiązać napotkane problemy, warto rozważyć kilka możliwych działań:

- **Modyfikacja Architektury Modelu:** Wprowadzenie specjalizowanych podsieci lub modułów dedykowanych generowaniu struktur jaskiniowych. Może to obejmować zastosowanie dodatkowych mechanizmów uwagi skupionych na podziemnych formacjach.
- **Zwiększenie Różnorodności Danych:** Rozszerzenie zbioru danych o różne biomy i typy bloków, co może pomóc modelowi w nauce szerszego spektrum wzorców i struktur. Uwzględnienie większej liczby chunków zawierających jaskinie może poprawić zdolność modelu do ich generowania.
- **Dostosowanie Funkcji Straty:** Eksperymentowanie z innymi funkcjami straty lub modyfikacja istniejących poprzez nadanie większych wag klasom rzadziej występującym, takim jak *Cave air* (powietrze jaskiniowe). Może skłonić model do większej uwagi w generowaniu tych elementów.

- Zwiększenie Zmienności: Udoskonalenie sposobu wprowadzania szumu do modelu, aby zwiększyć różnorodność generowanych chunków. Można również rozważyć zastosowanie technik regularyzacji³, takich jak dropout⁴ czy augmentacja danych, aby zapobiec nadmiernemu dopasowaniu.

Praca ta stanowi istotny krok w kierunku wykorzystania sztucznej inteligencji do generowania światów w grach komputerowych, oferując większą kontrolę nad procesem i możliwość personalizacji środowiska gry. Pomimo napotkanych wyzwań związanych z generowaniem złożonych struktur, takich jak jaskinie, wyniki badań są obiecujące i wskazują na duży potencjał dalszego rozwoju w tym obszarze.

Implementacja proponowanych ulepszeń w przyszłych pracach może prowadzić do stworzenia bardziej zaawansowanych modeli, zdolnych do generowania pełniejszych i bardziej zróżnicowanych światów, które lepiej odzwierciedlają złożoność i bogactwo środowiska Minecrafta. Tym samym, badania te przyczyniają się do poszerzenia wiedzy na temat zastosowań sztucznej inteligencji w projektowaniu gier i otwierają nowe możliwości dla twórców oraz użytkowników.

³ wprowadzenie dodatkowej informacji do rozwiązywanego zagadnienia źle postawionego w celu polepszenia jakości rozwiązania.

⁴ losowe ustawienia wychodzących krawędzi ukrytych jednostek (neuronów tworzących ukryte warstwy) na 0 przy każdej aktualizacji fazy treningu.

Bibliografia

- [1] J. Togelius, G. N. Yannakakis, K. O. Stanley i C. Browne, „Search-based Procedural Content Generation: A Taxonomy and Survey,” *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 172-186, 2011.
- [2] J. Togelius, N. Shaker i M. J. Nelson, „Introduction,” w *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer, 2016, pp. 1-15.
- [3] M. Toy, G. Wichman i K. Arnold, „Rogue,” 1980.
- [4] B. Wirtz, „The Rise of the Roguelikes: What Is A Roguelike?,” Lake House Media, LLC, 2023. [Online]. Available: <https://www.gamedesigning.org/gaming/roguelike/>.
- [5] „The algorithms of No Man’s Sky,” Rambus Press, 2016. [Online]. Available: <https://www.rambus.com/blogs/the-algorithms-of-no-mans-sky-2/>.
- [6] A. Liapis, G. N. Yannakakis i J. Togelius, „Discussion,” w *Computational Game Creativity*, 2014.
- [7] N. Shaker, G. Smith i G. N. Yannakakis, „Evaluating content generators,” w *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer, 2016, pp. 215-224.
- [8] N. A. Schudlo, „Ease of Authoring the Story/Creative Control,” w *Development of an Emergent Narrative Generation Architecture for Videogames*, 2014.
- [9] P. Nilsson i O. Nyholm, *A Comparison Between Evolutionary and Rule-Based Level Generation*, 2017.
- [10] S. Lee-Urban, „Procedural Content Generation,” 2018. [Online]. Available: https://faculty.cc.gatech.edu/~surban6/2018sp-gameAI/lectures/2018_04_05-ProceduralContentGeneration_concluded.pdf.
- [11] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen i J. Togelius, *Procedural Content Generation via Machine Learning (PCGML)*, 2018.
- [12] „Machine learning in video games,” [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning_in_video_games.
- [13] „Generative Design in Minecraft (GDMC),” [Online]. Available: <https://github.com/williamcwi/GDMC>.
- [14] C. Salge, M. C. Green, R. Canaan i J. Togelius, „Generative design in minecraft (GDMC): settlement generation competition,” *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 2018.
- [15] „Minecraft,” Mojang Studios, 2011.

- [16] A. Barbosa, „Minecraft Has Sold Over 200 Million Copies, Over 126 Million Monthly Players,” 2020. [Online]. Available: <https://www.gamespot.com/articles/minecraft-has-sold-over-200-million-copies-over-12/1100-6477383/>.
- [17] „Chunk,” [Online]. Available: <https://minecraft.wiki/w/Chunk>.
- [18] „Noise generator,” [Online]. Available: https://minecraft.wiki/w/Noise_generator.
- [19] „World generation,” [Online]. Available: https://minecraft.wiki/w/World_generation.
- [20] „Biome,” [Online]. Available: <https://minecraft.wiki/w/Biome>.
- [21] „Seed (world generation),” [Online]. Available: [https://minecraft.wiki/w/Seed_\(world_generation\)](https://minecraft.wiki/w/Seed_(world_generation)).
- [22] Y. LeCun, L. Bottou, Y. Bengio i P. Haffner, „Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, tom 86, nr 11, pp. 2278-2324, 1998.
- [23] I. Goodfellow, Y. Bengio i A. Courville, „Convolutional Networks,” w *Deep Learning*, MIT Press, 2016.
- [24] V. Nair i G. E. Hinton, „Rectified linear units improve restricted boltzmann machines,” w *Proceedings of the 27th International Conference on International Conference on Machine Learning*, Omnipress, 2010, p. 807–814.
- [25] D. Scherer, A. Müller i S. Behnke, „Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition,” w *Artificial Neural Networks – ICANN 2010*, Springer Berlin Heidelberg, 2010, pp. 92-101.
- [26] D. Maturana i S. Scherer, „VoxNet: A 3D Convolutional Neural Network for real-time object recognition,” w *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 922-928.
- [27] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville i Y. Bengio, *Generative Adversarial Networks*, 2014.
- [28] M. Mirza i S. Osindero, *Conditional Generative Adversarial Nets*, 2014.
- [29] M. Arjovsky, S. Chintala i L. Bottou, *Wasserstein GAN*, 2017.
- [30] O. Ronneberger, P. Fischer i T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015.
- [31] A. Vaswani , N. Shazeer , N. Parmar , J. Uszkoreit , L. Jones , A. N. Gomez , L. Kaiser i I. Polosukhin, *Attention Is All You Need*, 2023.
- [32] S. Woo , J. Park , J.-Y. Lee i I. S. Kweon, *CBAM: Convolutional Block Attention Module*, 2018.

- [33] P. Isola , J.-Y. Zhu , T. Zhou i A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, 2018.
- [34] „Minecraft Forge,” [Online]. Available: <https://github.com/MinecraftForge/MinecraftForge>.
- [35] „Chunky,” [Online]. Available: <https://github.com/pop4959/Chunky>.
- [36] „Amulet Core,” [Online]. Available: <https://github.com/Amulet-Team/Amulet-Core>.
- [37] D. N., „How Many Blocks Are In Minecraft (The True Count!),” ThatVideoGameBlog, 2023. [Online]. Available: <https://thatvideogameblog.com/how-many-blocks-are-in-minecraft/>.
- [38] „GPU machine types,” Google, [Online]. Available: <https://cloud.google.com/compute/docs/gpus>.
- [39] I. Gulrajani , F. Ahmed , M. Arjovsky , V. Dumoulin i A. Courville, *Improved Training of Wasserstein GANs*, 2017.
- [40] „Inception score,” [Online]. Available: https://en.wikipedia.org/wiki/Inception_score.
- [41] „Fréchet inception distance,” [Online]. Available: https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance.
- [42] M. Awiszus, F. Schubert i B. Rosenhahn, „World-GAN: a Generative Model for Minecraft Worlds,” w *Proceedings of the IEEE Conference on Games*, 2021.
- [43] V. I. Levenshtein, „Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet physics. Doklady*, tom 10, pp. 707-710, 1965.
- [44] S. M. Lucas i V. Volz, „Tile pattern KL-divergence for analysing and evolving game levels,” w *Proceedings of the Genetic and Evolutionary Computation Conference*, ACM, 2019.
- [45] „Kullback–Leibler divergence,” [Online]. Available: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence.
- [46] „World-GAN,” [Online]. Available: <https://github.com/Mawiszus/World-GAN>.
- [47] „TOAD-GAN,” [Online]. Available: <https://github.com/Mawiszus/TOAD-GAN>.
- [48] „U-Net: Convolutional Networks for Biomedical Image Segmentation,” [Online]. Available: <https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>.

Praca została przeredagowana przy użyciu narzędzia ChatGPT, model językowy w wersji 4o.

Spis rysunków

RYSUNEK 1. ZRZUT EKRANU TYPOWEGO KRAJOBRAZU MINECRAFTA PREZENTUJĄCEGO ŚRODOWISKO ZBUDOWANE Z BLOKÓW.	7
RYSUNEK 2. WIZUALIZACJA CZĘŚCI PODZIEMNEJ POJEDYNCZEGO CHUNKA PRZY WYSOKOŚCI 75 I GŁĘBOKOŚCI -64.	8
RYSUNEK 3. WIZUALIZACJA OPERACJI SPLOTU Z KERNELEM 3X3 NA WEJŚCIU 2D O WYMIARACH 4X4.	10
RYSUNEK 4. WIZUALIZACJA OPERACJI MAX-POOL 2X2 Z PRZESUNIĘCIEM OKNA RÓWNYM 2.	11
RYSUNEK 5. SCHEMAT ILUSTRUJĄCY INTERAKCJĘ MIĘDZY GENERATOREM A DYSKRYMINATOREM W SIECI GAN.	11
RYSUNEK 6. SCHEMAT ARCHITEKTURY U-NET PRZEDSTAWIAJĄCY POŁĄCZENIA KODERA, DEKODERA I POMIJANIA. ŹRÓDŁO: [48]	13
RYSUNEK 7. SCHEMAT ARCHITEKTURY SIECI U-NET GENERATORA Z POŁĄCZENIAMI POMIJAJĄCYMI I CAVE ATTENTION.	16
RYSUNEK 8. ILUSTRACJA PRZEPŁYWU PRZETWARZANIA W WARSTWIE WARUNKOWEJ CAVE MAP.	17
RYSUNEK 9. SCHEMAT ARCHITEKTURY U-NET DYSKRYMINATORA.	18
RYSUNEK 10. WYKRES PORÓWNUJĄCY STRATY GENERATORA POMIĘDZY MODELAMI PRZECZ CAŁY OKRES UCZENIA (4 EPOKI, 2048 KROKÓW).	26
RYSUNEK 11. WYKRES PORÓWNUJĄCY STRATY DYSKRYMINATORA POMIĘDZY MODELAMI PRZECZ CAŁY OKRES UCZENIA (4 EPOKI, 2048 KROKÓW).	27
RYSUNEK 12. WYKRES DYSTRYBUCJI BLOKÓW WZGLĘDEM RZECZYWISTYCH DANYCH POMIĘDZY MODELAMI.	27
RYSUNEK 13. PRZYKŁAD WYGENEROWANEGO ŚWIATA ZA POMOCĄ OPRACOWANEGO MODELU (MODEL 4). ŚWIAT O ROZMIARACH 6X6 CHUNKÓW. PO LEWEJ: SUROWY ŚWIAT, JEST TO BEZPOŚREDNI WYNIK MODELU. PO PRAWEJ: "UDEKOROWANY" ŚWIAT, ZOSTAŁ WYPEŁNIONY WODA DO POZIOMU MORZA I POKRYTY TRAWĄ.	33
RYSUNEK 14. HISTOGRAM BLOKÓW W RZECZYWISTYCH CHUNKACH Z GRY MINECRAFT NA PRZESTRZENI 16 CHUNKÓW.	35
RYSUNEK 15. HISTOGRAM BLOKÓW WYGENEROWANYCH CHUNKÓW PRZECZ OPRACOWANY MODEL NA PRZESTRZENI 16 CHUNKÓW I UWZGLĘDNIONYM BŁĘDEM W STOSUNKU DO HISTOGRAMU Z RYSUNEK 14.	36

Spis tabel

TABELA 1. MAPOWANIE BLOKÓW NA IDENTYFIKATORY CAŁKOWITE I ICH KODOWANIE ONE-HOT.	21
TABELA 2. PRZYKŁAD OBLICZANIA DYSTANSU LEVENSHTTEINA MIĘDZY DWOMA CIĄGAMI (CHUNKAMI).	30
TABELA 3. SZCZEGÓŁOWE WYNIKI TPKL-DIV DLA OPRACOWANEGO MODELU.	33
TABELA 4. PORÓWNANIE WYNIKU TPKL-DIV OPRACOWANEGO MODELU Z INNYMI MODELAMI.	34
TABELA 5. SZCZEGÓŁOWE WYNIKI DYSTANSU LEVENSHTTEINA DLA OPRACOWANEGO MODELU.	34
TABELA 6. PORÓWNANIE DYSTANSU LEVENSHTTEINA DLA OPRACOWANEGO MODELU Z INNYMI MODELAMI.	35
TABELA 7. SZCZEGÓŁOWE WYNIKI MGD MIĘDZY WYGENEROWANYMI A RZECZYWISTYMI CHUNKAMI W OPRACOWANYM MODELU.	36