České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství

Procedurální generování prostředí

Procedural Generation of Environment

BAKALÁŘSKÁ PRÁCE

Vypracoval: Miroslav Baša
Vedoucí práce: Ing. Josef Nový, Ph.D.
Rok: 2023

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství

Akademický rok 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Miroslav Baša**
Studijní program:	**Aplikace přírodních věd**
Obor:	**Aplikace softwarového inženýrství**
Název práce česky:	**Procedurální generování prostředí**
Název práce anglicky:	**Procedural Generation of Environment**
Jazyk práce:	**Čeština**

Pokyny pro vypracování:

1. Nastudujte problematiku procedurálního generování prostředí.
2. Porovnejte a vyberte vhodnou technologii pro generování a následnou visualizaci.
3. Navrhněte aplikaci pro generování a vizualizaci prostředí.
4. Navrženou aplikaci implementujte a otestujte.

Doporučená literatura:

[1] CHENG, Wei-Chien, Wen-Chieh LIN a Yi-Jheng HUANG, 2014. Controllable and Real-Time Reproducible Perlin Noise. *Smart Graphics* [online]. Cham: Springer International Publishing, 2014, 86-97 [cit. 2022-10-20]. Lecture Notes in Computer Science. ISBN 978-3-319-11649-5. doi:10.1007/978-3-319-11650-1_8

[2] JoLIU, Jialin, Sam SNODGRASS, Ahmed KHALIFA, Sebastian RISI, Georgios N. YANNAKAKIS a Julian TOGELIUS, 2021. Deep learning for procedural content generation. *Neural Computing and Applications* [online]. 33(1) [cit. 2022-10-20]. ISSN 0941-0643. doi:10.1007/s00521-020-05383-8

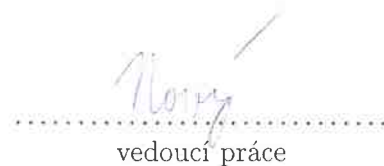
[3] SCHEIBENPFLUG, Andreas, Johannes KARDER, Susanne SCHALLER, Stefan WAGNER a Michael AFFENZELLER, 2016. Evolutionary Procedural 2D Map Generation using Novelty Search. *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion* [online]. New York, NY, USA: ACM, 2016-07-20, 39-40 [cit. 2022-10-20]. ISBN 9781450343237. doi:10.1145/2908961.2909047

Jméno a pracoviště vedoucího práce:

Ing. Josef Nový, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze

Jméno a pracoviště konzultanta:

–

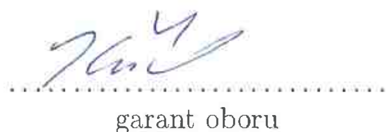


vedoucí práce

Datum zadání bakalářské práce: 15. 10. 2022

Termín odevzdání bakalářské práce: 2. 8. 2023

Doba platnosti zadání je dva roky od data zadání.



garant oboru

vedoucí katedry

děkan

V Praze dne 15. 10. 2022

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Děčíně dne ....

Miroslav Baša

Poděkování

Rád bych vyjádřil upřímné poděkování Ing. Josefu Novému, Ph.D. za jeho vedení během mé bakalářské práce a za všechny cenné podněty, které přispěly k obohacení práce.

Miroslav Baša

Název práce:
Procedurální generování prostředí

Autor: Miroslav Baša

Studijní program: Aplikace přírodních věd
Obor: Aplikace softwarového inženýrství
Druh práce: Bakalářská práce

Vedoucí práce: Ing. Josef Nový, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze
Konzultant: –

Abstrakt: V této bakalářské práci se zaměřuji na porovnání různých metod pro práci s neuronovou síti typu GAN. Konkrétně jsem naučil neuronovou síť na datasetu regionálních map. V úvodu se věnuji zavedení základních pojmů souvisejících s problematikou neuronových sítí, jejich využitím v praxi a složitostí. V další části se zabývám různými druhy šumů, jejich matematickou složitostí a možnými uplatněními v reálném světě. Závěrem práce jsou nově vygenerované mapy, které byly porovnány a zhodnoceny studenty ČVUT v Děčíně, a následné vyhodnocení výsledků.
Klíčová slova: Procedurální generování obsahu, Hluboké učení, Výpočetní a umělá inteligence, GAN

Title:
Procedural Generation of Environment

Author: Miroslav Baša

Abstract: In this bachelor's thesis, I focus on comparing various methods for working with a GAN-type neural network. Specifically, I trained a neural network on a dataset of regional maps. In the introduction, I will introduce basic concepts related to the issue of neural networks, their use in practice and complexities.
In the next part, I deal with different types of noise, their mathematical complexity and possible applications in the real world. The conclusion of the work is the newly generated maps, which were compared and evaluated by the students of the Czech Technical University in Děčín, and the subsequent evaluation of the results.
Key words: Procedural content generation, Deep learning, Computational and artificial intelligence, GAN

Obsah

Úvod **11**

1 Procedurální generování **13**
1.1 Úvod . . . 13
1.2 Využití v herním průmyslu . . . 13
1.2.1 Úrovně . . . 13
1.2.2 Obličej/charakter . . . 14
1.2.3 Text . . . 14
1.2.4 Hudba . . . 14
1.3 Generování fraktálů . . . 15
1.4 Buněčné/celulární automaty . . . 16
1.5 L-systémy . . . 17
1.6 Neuronové sítě . . . 17

2 Šumy a metody **19**
2.1 Základní definice šumu . . . 19
2.1.1 Koherence . . . 19
2.2 Perlinův šum . . . 19
2.3 Simplexový šum . . . 22
2.4 Worleyho šum . . . 23
2.5 Value šum . . . 23
2.6 Voronoiho šum . . . 24
2.7 Parametry generování šumu . . . 25
2.7.1 Oktávy . . . 25
2.7.2 Lacunarity . . . 25
2.7.3 Persistence . . . 26
2.7.4 Frekvence . . . 26
2.8 Porovnání šumů . . . 26

3 Neuronové sítě **27**
3.1 Analogie neuronu . . . 27
3.1.1 Biologické neurony . . . 27
3.1.2 Umělé neurony . . . 28
3.2 Struktura neuronové sítě . . . 29
3.3 Historie neuronových sítí . . . 29
3.4 Typy neuronových sítí . . . 30
3.4.1 Perceptron . . . 30

3.4.2 Dopředné neuronové sítě (FFNN) 31
3.4.3 Rekurentní neuronové sítě (RNN) 31
3.4.4 Konvoluční neuronové sítě (CNN) 32
3.5 Učení neuronové sítě 33
3.5.1 Učení bez učitele 33
3.5.2 Učení s učitelem 33
3.5.3 Zpětnovazební učení 34

4 Návrh programu **35**
4.1 Výběr technologií 35
4.2 Návrh systému 36
4.3 Vrstvy GAN 36
4.4 Modifikace GAN 38
4.4.1 Conditional GAN 38
4.4.2 Wasserstein GAN 38
4.4.3 Style GAN 39
4.5 Vizualizace 39
4.5.1 Generátor 39
4.5.2 Diskriminátor 40

5 Aplikace a implementace **43**
5.1 Výběr výchozího vzorku 43
5.2 Úprava datasetu 43
5.3 Rozšíření datasetu 45
5.4 Počáteční testování parametrů sítě 45
5.5 Problémy a úpravy pro zlepšení sítě 46
5.6 Použití šumů v praxi 48
5.7 Implementace generátoru 50
5.8 Implementace diskriminátoru 53

6 Vizualizace výsledků **57**
6.1 Uživatelské rozhraní 57
6.2 Výsledky generování 58
6.2.1 Náhodný šum 58
6.2.2 Perlinův šum 59
6.2.3 Simplexový šum 61

Závěr **71**

Literatura **72**

Přílohy **77**

A Externí příloha **77**

Úvod

V této bakalářské práci se zaměřuji na využití neuronové sítě typu GAN pro generování map z náhodného šumu, Perlinova šumu a Simplexového šumu na základě trénovaných dat.

Tato práce umožňuje vytvářet mapy, které mohou sloužit jako základ pro další tvorbu prostředí. Pro trénování sítě jsem použil vlastní dataset víceregionální mapy. Cílem práce bylo vytvořit funkční generátor, který je schopen vytvářet mapy prostředí, které budou vizuálně podobné reálným mapám. Zároveň cílem práce bylo otestovat různé druhy vstupních paramatrů pro procedurální generování.

Uplatnění mé práce by mohlo být prakticky využito jako začátek pro herní průmysl nebo i v architektuře například pro modelování nových oblastí nebo plánování tras.

Práce je rozdělena do šesti hlavních kapitol. První kapitola se věnuje různým metodám procedurálního generování. Popisuji zde základní metody, jako jsou L-systémy až po pokročilejší metody, včetně neuronových sítí. Následující kapitola se zabývá různými druhy šumů, které sloužili jako vstupní parametr. Popisuji zde jejich výhody a nevýhody a porovnávám jejich vlastnosti vůči sobě. V další kapitole představuji základy neuronových sítí, které jsem použil ve své práci. Popisuji zde základní principy neuronu a různé typy dalších neuronových sítí.

Poté se přesouvám k návrhu mé práce, kde popisuji prostředí, ve kterém jsem pracoval a také alternativní přístupy, které jsem zvažoval. Následující kapitola se zaměřuje na aplikaci již navrhnuté práce, zmiňuji zde proces generování mého datasetu a detailně popisuji problémy, se kterými jsem se potýkal. Vysvětluji zároveň opatření, která jsem musel podniknout. Poslední kapitola popisuje vizualizaci a formu zhodnocení mých výsledků.

Kapitola 1

Procedurální generování

1.1 Úvod

Procedurální generování je technika používaná k automatickému vytváření obsahu pomocí algoritmů a pravidel. Tento přístup je často využíván v oblasti vývoje videoher, grafiky a dalších digitálních nástrojů pro rychlé a efektivní vytváření velkého množství obsahu.

Proces procedurálního generování může zahrnovat různé kroky, jako jsou definování parametrů, stanovení pravidel a iterace algoritmů pro generování obsahu. Cílem je produkovat obsah, který je unikátní, rozmanitý a zajímavý pro uživatele.

Procedurální generování je výhodné v několika směrech. Ušetří čas a úsilí v procesu tvorby obsahu, umožní vytváření rozsáhlých a různorodých krajinných útvarů nebo úrovní a poskytuje uživatelům čerstvý a jedinečný zážitek při interakci s obsahem. Existuje několik metod procedurální generace, včetně následujících: fraktální generace, buněčné/celulární automaty, L-systémy a nebo neuronové sítě, které jsem využíval pro mojí práci. Tyto generace popisuji v následujících kapitolách.

1.2 Využití v herním průmyslu

Procedurální generování je často využívaným nástrojem pro tvorbu herních světů. V následujících podkapitolách rozepíši jednotlivé současné, ale i budoucí použití.

1.2.1 Úrovně

Úrovně her jsou běžným obsahem ve videohrách. Jedná se o prostory ve dvou nebo třech rozměrech, kterými hráči musí projít. Tyto úrovně musí být hratelné a musí dodržovat funkční omezení, jako je neprostupná geometrie, nutné předměty pro dokončení úrovně a nepřátelé, které hráči mohou porazit.

Procedurální generování se často používá ve 2D platformových hrách jako je Spelunky nebo akademických výzkumných projektů [28], jako je Mario AI Framework. Další typy 2D úrovní zahrnují rogue-like nebo dungeon-crawler úrovně, úrovně ve střílečkách z pohledu první osoby a bojové mapy pro strategické hry.

1.2.2 Obličej/charakter

Hluboké učení výrazně zlepšilo tvorbu obsahu v oblastech jako jsou tváře a modely postav [28], ale tyto pokroky zatím nebyly široce implementovány ve hrách. Dataset Celeb-A a podobné sady dat byly použity k vývoji nových variací, což vedlo k ohromujícím průlomům v generování obličejů.

Mnoho her vyžaduje lidské tváře pro různé role jako například pro generování NPC postav nebo design postav. Nicméně metody založené na strojovém učení nebyly dosud plně využity.

1.2.3 Text

Procedurální generování umožňuje vytvářet složitější a lépe strukturované věty, které mohou lépe zapadat do herního kontextu. Avšak integrace těchto metod do her přináší své výzvy, především kvůli omezenému ovládání procesu generování textu [28]. Hry potřebují zajistit, aby vygenerovaný text nejen byl kvalitní, ale také aby byl konzistentní s pravidly a cíli hry.

Jedním z příkladů hry, která dokázala úspěšně využít téměř nekontrolovatelnou generaci textu, je AI Dungeon 2. Tato hra umožňuje hráčům zasahovat do příběhu prostřednictvím svých akcí a rozhodnutí, a algoritmus poté generuje odpovídající text, který reaguje na hráčovy volby. I přes jistou nepředvídatelnost a občasnou nekonzistenci textu se tato metoda ukázala být zábavným a inovativním způsobem, jak hráčům umožnit být součástí tvorby příběhu.

1.2.4 Hudba

Většina her má soundtrack skládající se z hudby a zvukových efektů. Omezení na soundtrack jsou relativně mírná ve srovnání s jinými typy obsahových omezení [28]. Zvukové efekty by měly odpovídat akcím ve hře a hudba by měla vyjadřovat atmosféru okamžiku.

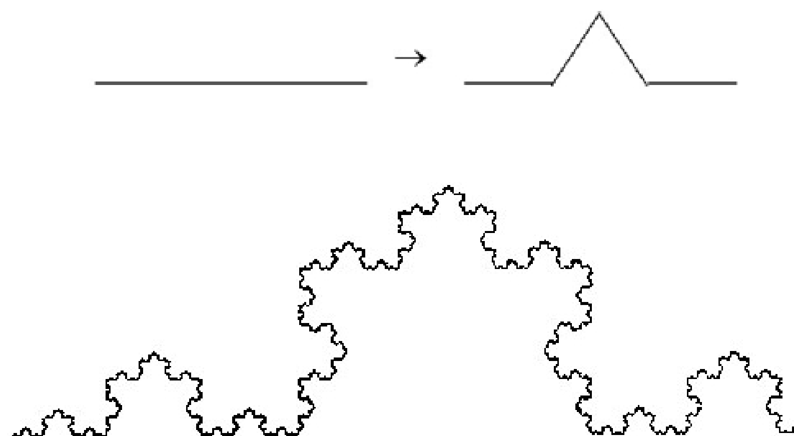
Některé hry využívají procedurálně generovaný soundtrack a proběhly výzkumné projekty zaměřené na generování hudby v reálném čase s ohledem na emocionální změny. Hluboké učení dosáhlo pokroku při generování hudby s určitou ovladatelností a pomalu se objevují snadno využitelné nástroje pro automatické skládání hudby.

1.3 Generování fraktálů

Fraktály jsou matematické struktury, které mají vlastnosti jako je samo-opakování a nekonečná složitost. Fraktály lze použít k vytváření terénu.

Generování fraktálů může vytvářet velmi detailní a realistické krajiny. Použití matematických vzorců zajistí, že generovaný obsah je konzistentní a dodržuje určité vzorce.

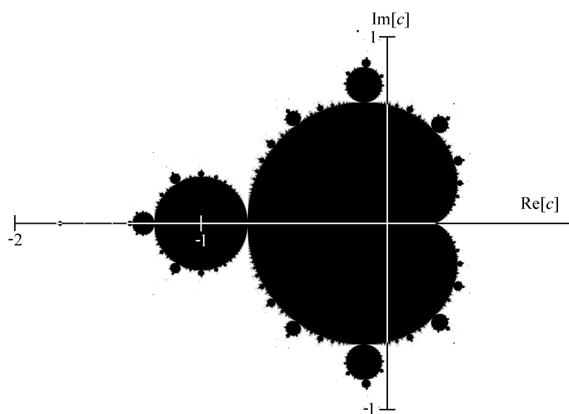
Jeden z klasických fraktálů je Kochova křivka, nazývaná též Kochova vločka. Tato křivka má několik zajímavých vlastností, neobsahuje žádné úsečky nebo hladké segmenty a nemá derivaci v žádném bodě.



Obrázek 1.1: Ukázka postupu a následná Kochova vločka [38]

Na obrázku 1.1 ukazuji základní postup, kdy měním všechny úsečky na trojúhelníky a zároveň ukazuji opakování 5 iterací, po kterých mi vznikne Kochova vločka.

Další metodou pro tvorbu fraktálů je například Mandelbrotova množina, která je generováná iterativním výpočtem v komplexním číselném prostoru. na přiloženém obrázku 1.2 ukazuji Mandelbrotovu množinu v kontinuálně barevném prostředí.



Obrázek 1.2: Ukázka Mandelbrotovy množiny [36]

Vzorec [38] pro Mandelbrotovu množinu je definován tak, že pro každý bod c v komplexním číselném prostoru se iteruje následující vzorec

$$\mathbf{z_{n+1} = z_n^2 + c}$$

kde $z_0 = 0$ a c je daný bod v komplexním číselném prostoru. Pokud hodnota z iteruje k nekonečnu nebo překročí určený limit, je bod považován za součást Mandelbrotovy množiny.

1.4 Buněčné/celulární automaty

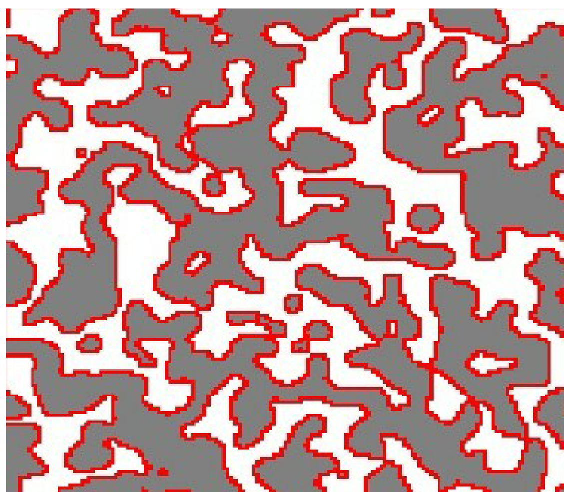
Celulární automat je složen z mřížky buněk, které jsou uspořádány v n-rozměrném prostoru. Každá buňka v této mřížce může nabývat jednoho stavu z množiny možných stavů.

Procedurální generování pomocí celulárních automatů [35] funguje na principu iterativního vyhodnocování stavů jednotlivých buněk na základě pravidel, nazývaných přechodové funkce. Přechodová funkce bere v úvahu stav vyhodnocované buňky a stavy jejích sousedních buněk. Na základě těchto informací se určí nový stav vyhodnocované buňky v dalším časovém kroku (další "generace"buněk).

Buněčné automaty jsou zajímavé pro svou schopnost vykazovat nelineární a nepředvídatelné chování. V závislosti na použitých pravidlech, stavech, sousedství a dalších parametrech mohou buňky v mřížce vykazovat různé vzory chování.

Celulární automaty mohou vytvářet složité vzory, které je obtížné dosáhnout jinými metodami. Používají se pro jednoduché simulace fyzikálních jevů. V kontextu procedurálního generování obsahu do počítačových her nejsou kvůli nedostatečným možnostem jejich ovládání nejlepší pro procedurální generování, avšak lze je použít v kombinaci s dalšími metodami. Celulární automaty se používají například k simulování šíření ohně nebo proudění kapalin. Díky své nižší výpočetní náročnosti než analytická řešení a dostatečné přesnosti pro herní účely jsou pro tyto účely výhodné.

Konkrétněji, jedna z metod generování nekonečných jeskynních 2D komplexů pomocí celulárního automatu funguje na principu rozdělení herní plochy na čtvercovou mřížku buněk. Každá buňka může být ve stavu skála (neprůchozí), podlaha (průchozí) nebo stěna (na rozhraní skály a podlahy). Na začátku algoritmu jsou všechny buňky ve stavu podlahy a určité procento z nich je náhodně změněno na stav skály. Poté se iterativně vyhodnocují stavy buněk na základě okolních buněk a pravidel. Výsledkem je generovaný jeskynní komplex, který lze postupně generovat, když se hráč pohybuje herním světem. Přikládám příklad tohoto generování 1.3. Toto řešení umožňuje parametrizaci pro různá chování a výsledky buněčného automatu.



Obrázek 1.3: Ukázka jeskyně složená z 9 mřížek o velikosti 50x50 buněk vygenerovaná pomocí celulárního automatu. [35]

1.5 L-systémy

L-systémy následují soubor pravidel, která postupně transformují počáteční řetězec na složitější sekvenci. Každá iterace generuje nový řetězec nahrazením určitých symbolů na základě předem definovaných produkčních pravidel. Tento iterativní proces umožňuje L-systémům vytvářet složité a podrobné struktury opakovaným použitím stejných pravidel, což vede k vzniku komplexních vzorů. V kontextu pro-

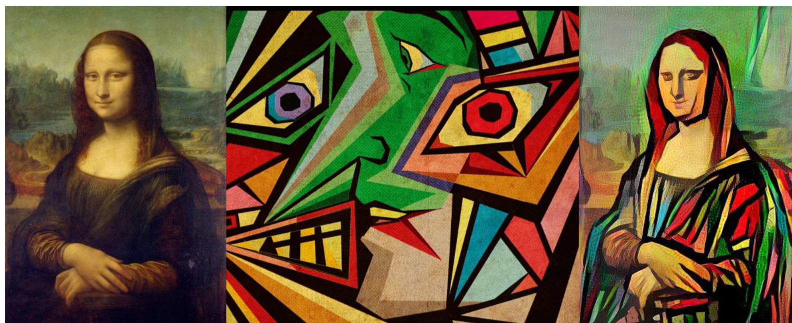
Obrázek 1.4: Ukázka vykreslení několika L-systémů v podobě rostlin [34]

cedurálního generování jsou L-systémy zvláště populární pro generování struktur podobných rostlinám, protože dokážou efektivně simulovat větvení a růstové vzory různých rostlin. Upravováním parametrů a pravidel L-systému mohou tvůrci a vývojáři dosáhnout široké škály tvarů a forem rostlin. Pro názornou ukázku přikládám převzatý obrázek 1.4.

1.6 Neuronové sítě

Neuronové sítě se mohou učit ze stávajícího obsahu a generovat nový obsah, který je podobný stylu a kvalitě. Mohou generovat vysoce realistický a detailní obsah, který je obtížné dosáhnout jinými metodami. Neuronové sítě také mohou být trénovány k

generování obsahu, který splňuje specifické požadavky.Příkladem může být generování hudby, která odpovídá určité náladě, nebo textury, která odpovídá konkrétnímu stylu. Neuronové sítě jsou také flexibilní a mohou se přizpůsobit změnám vstupních dat, což je činí vhodnými pro použití v dynamických prostředích, jako jsou právě hry. Typickým zástupcem neuronových sítí určených ke generování obsahu jsou sítě s architekturou GAN (Generative Adversarial Networks). Ukázkou možným výstupem této sítě je například přenos obrázku do určitého uměleckého stylu, kterou ukazuji na obrázku 1.5.



Obrázek 1.5: Příklad přenosu uměleckého stylu [16]

Rozhodl jsem se pro svou práci využít právě neuronové sítě, které více popisuji v následujících kapitolách.

Kapitola 2

Šumy a metody

2.1 Základní definice šumu

Šum je základním prvkem pro procedurální generování světa. Je to náhodný signál bez užitečného významu. Šum může být popsán pomocí matematické funkce, která produkuje náhodné hodnoty v určitém rozsahu. Šum může být jednorozměrný, dvourozměrný nebo i vícerozměrný.

2.1.1 Koherence

Jedna z velmi důležitých vlastností následujících zmíněných šumů je koherence. Koherence je klíčová vlastnost, která zajišťuje, že výsledná textura vypadá hladce, bez ostrých přechodů. Tyto tři podmínky [20] musí platit pro to, aby byl šum označen jako koherentní :

1. Předání stejné vstupní hodnoty vždy vrátí stejnou výstupní hodnotu.
2. Malá změna vstupní hodnoty vyvolá malou změnu výstupní hodnoty.
3. Velká změna vstupní hodnoty vyvolá náhodnou změnu výstupní hodnoty

Právě použití těchto šumů je využito pro generování různých prvků ve hrách jako jsou terén, textury, oblaka. Procedurální generování světa využívá různé druhy šumů k přidání náhodnosti a složitosti vstupních dat, což může pomoci síti lépe generalizovat a snížit přetrénování. Následným implementováním šumu lze dosáhnout náhodnosti výsledků a nerepetitivnosti. V následující části jsem vybral některé ze známějších šumů.

2.2 Perlinův šum

Přirozeně vypadající vzory nelze snadno vytvořit pomocí standardního generování šumu [11]. Proto Ken Perlin vytvořil Perlinův šum 2.1. Tento šum byl vytvořen v roce

1983 [24] a od té doby se stal jedním z nejpopulárnějších algoritmů pro generování realistických textur, terénu a dalších prvků. Perlinův šum vyhlazuje rozdíly mezi sousedními body, což vede k více křivkovitým grafům v jednorozměrném prostoru a k oblakovitým rysům ve dvourozměrném prostoru.

Perlinův šum se vyznačuje svou schopností vytvářet jemné a přirozené detaily, které jsou často obtížné dosáhnout jinými způsoby.

Základní myšlenkou Perlinova šumu je vytvořit sekvenci náhodných gradientů a interpolovat mezi nimi, aby se vytvořila hladká, spojitá funkce. Gradienty jsou obvykle reprezentovány jako vektory, přičemž každý vektor směřuje náhodným směrem v n-rozměrném prostoru (kde n je počet dimenzí šumu).

Algoritmus pro generování Perlinova šumu [25] lze shrnout takto:

1. Vygeneruje se mřížka náhodných gradientových vektorů. Pro každý bod v mřížce se přiřadí náhodný gradientový vektor.
2. Pro každý bod ve výstupní mřížce se zjistí, do které buňky mřížky spadá. Spočítá se skalární součin mezi gradientovým vektorem v každém rohu buňky mřížky a vektorem od bodu k rohu.
3. Interpoluje se mezi skalárními součiny v každém rohu buňky mřížky pomocí hladké interpolační funkce (např. kubické interpolace).
4. Následně se opakují kroky 2-4 pro více oktáv a výsledky se spojí pomocí váženého součtu.

Základní algoritmus pro generování 2D Perlinova šumu [8]:

Nechť funkce šumu je definována pro libovolný reálný bod $P = (x, y) \in \mathbb{R}^2$.

$$Q = (x_0, y_0) \cup (x_0, y_1) \cup (x_1, y_0) \cup (x_1, y_1) \in \mathbb{Z} \tag{2.1}$$

Nechť body Q jsou nejbližší body s celočíselnými souřadnicemi, kde:

1. Pro každý ze čtyř bodů Q získáme pseudonáhodný gradient G z tabulky gradientů. Gradient pro daný bod je získán pomocí funkce:
$$G_t[P_t[P_t[x] + y]] \tag{2.2}$$
kde G_t a P_t jsou předem vypočítané tabulky gradientů a permutací.
2. Pro každý z čtyř bodů Q se vypočte vliv odpovídajícího rohového bodu tím, že se vypočítá hodnota
$$G \cdot (P - Q) \tag{2.3}$$
3. Nasledně se interpolujou dvě vyšší a dvě nižší hodnoty vlivu použitím plynulé křivky v každém kroku (např. $3t^2 - 2t^3$).

4. V posledním kroku se interpoluje poslední dvě hodnoty pomocí stejné plynulé křivky a vrátí se jako výsledek funkce Perlinova šumu v bodě. (x, y).

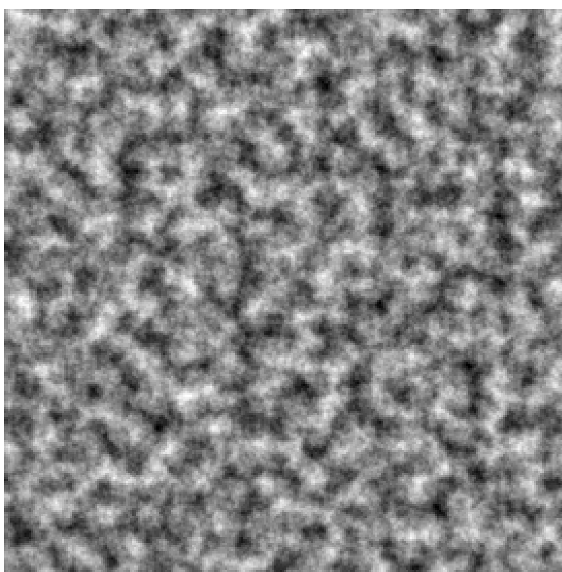
Původně Ken Perlin používal Hermiteho míchací funkci:

$$f(t) = 3t^2 - 2t^3 \tag{2.4}$$

Později funkci změnil na polynom pátého stupně:

$$f(t) = 6t^5 - 15t^4 + 10t^3 \tag{2.5}$$

Důvodem bylo to, že je velmi žádoucí mít spojitou druhou derivaci pro šumovou funkci. Tyto dvě funkce jsou velmi podobné, ale křivka pátého stupně také má nulovou druhou derivaci na svých koncových bodech, což způsobuje, že šumová funkce má spojitou druhou derivaci všude. Díky tomu je šumová funkce lépe vhodná pro běžné úlohy počítačové grafiky, jako je deformace povrchu a bump mapping.



Obrázek 2.1: Ukázka Perlinova šumu

Perlinův šum se používá k procedurálnímu generování různých prvků v počítačových hrách, jako jsou například terény, textury, stromy, mraky a další. Tento druh šumu je oblíbený pro svou schopnost vytvářet realistické a přirozené detaily, které se přirozeně snoubí do celkového vizuálního stylu aplikace. Perlinův šum je též relativně rychlý a snadno implementovatelný, což ho činí vhodným pro použití v real-time aplikacích.

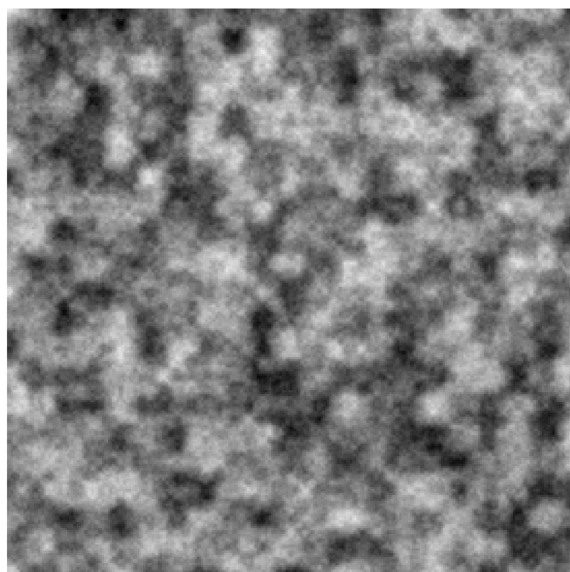
Nicméně tento šum má i své nevýhody. Jeho použití může vést k přílišně vyhlazenému vzhledu, pokud se nepoužívají další techniky pro vytváření detailů. Také může být náročné vytvořit vysoké detaily Perlinového šumu pro větší plochy.

2.3 Simplexový šum

Simplexový šum na obrázku 2.2 je pokročilý algoritmus šumu, který byl vynalezen Kenem Perlinem v roce 2001 jako vylepšení původního Perlinového algoritmu. Tento typ šumu se používá v mnoha aplikacích, jako jsou například počítačové hry, animace, filmy a další vizuální efekty.

Simplexový šum se vyznačuje větší efektivitou a rychlostí než původní Perlinův šum a zároveň produkuje méně artefaktů. Tento druh šumu je definován jako funkce, která produkuje hodnoty v určitém rozsahu. Stejně jako u Perlinového šumu, je i tento šum vytvářen pomocí gradientových funkcí a vyhlazovacích funkcí.

Pro výpočet Simplexového šumu se používá simplex mřížka, která je jednodušší a efektivnější než mřížka použitá u Perlinového šumu. Tato mřížka rozděluje prostor na trojúhelníky namísto čtverečků. Každý bod v mřížce má svůj vlastní náhodný vektor, který se používá k výpočtu gradientových funkcí. Tyto gradientové funkce se následně používají k výpočtu váh pro každý bod v mřížce. Tyto váhy jsou poté interpolovány pomocí vyhlazovacích funkcí, aby se vytvořil konečný Simplexový šum. Tento typ šumu se používá pro generování terénu, textur a dalších prvků v počítačo-



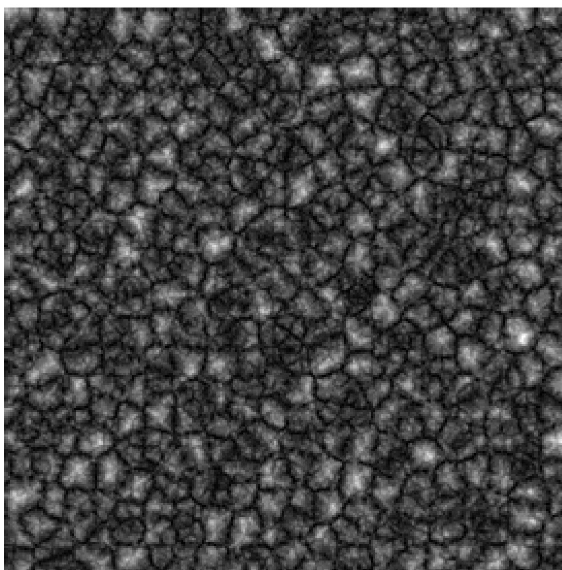
Obrázek 2.2: Ukázka Simplexového šumu

vých hrách. Například, Simplexový šum se často používá pro vytváření pohyblivých oblačných textur, které jsou nezbytné pro realistické prostředí ve hrách.

Nicméně, stejně jako u Perlinového šumu, použití Simplexového šumu má též své nevýhody. Jeho použití může vést k příliš vyhlazenému vzhledu, pokud se nepoužívají další techniky pro vytváření detailů. Přesto je Simplexový šum stále jedním z nejefektivnějších a nejpoužívanějších algoritmů pro generování procedurálního šumu v různých vizuálních aplikacích.

2.4 Worleyho šum

Worleyho šum, ukázka na obrázku 2.3, je druh šumu, který se používá především pro procedurální generování textur a terénu. Jedná se o šum [31], který je vytvořen náhodně rozmístěnými body v prostoru. Tyto body se nazývají vnitřní body a jsou umístěny v náhodných pozicích ve 3D prostoru.



Obrázek 2.3: Ukázka Worleyho šumu

Výsledný obrazový efekt vzniká v závislosti na vzdálenosti od nejbližšího vnitřního bodu. Čím blíže se pozice vstupního bodu nachází k vnitřnímu bodu, tím větší hodnotu bude mít výstupní hodnota.

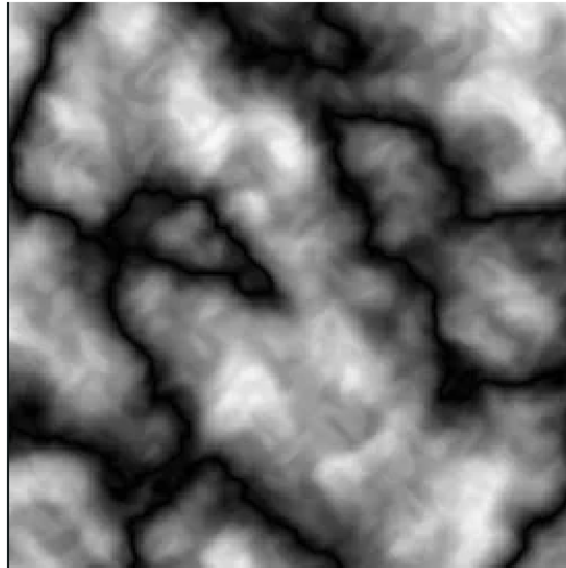
Existují různé metody pro výpočet Worleyho šumu, ale nejběžnější je tzv. "Manhattan distance"nebo též "taxi-cab distance", což je vzdálenost mezi dvěma body v metrice, kde se používá pouze vertikální a horizontální směr, nikoliv diagonální.

Tento šum se často používá pro generování kamenných a dřevěných textur, nebo pro vytváření terénu. Tyto aplikace jsou možné díky tomu, že Worleyho šum dokáže vytvořit přirozeně vypadající uspořádání objektů, jako jsou například kameny nebo dřevo.

2.5 Value šum

Value šum 2.4 známý jako white noise, je jeden z nejjednodušších typů šumu používaných v počítačové grafice. Jeho principem [39] je náhodně vygenerovat hodnoty a tyto hodnoty rozložit v prostoru. Jedná se o stochastický proces, kde každý bod v prostoru má svou vlastní náhodnou hodnotu.

Pro vytvoření tohoto šumu se používá grid [40], což je mřížka, která rozděluje prostor na malé krychle. Každý bod v této mřížce má svou vlastní náhodnou hodnotu. Tyto hodnoty se dále interpolují, aby se vytvořil plynulý šum.



Obrázek 2.4: Ukázka Value šumu

Value šum se využívá především pro procedurální generování textur a terénu. V případě textur se hodnoty value šumu používají jako faktory, které ovlivňují barvu nebo průhlednost pixelů textury. V případě terénu se hodnoty value šumu používají pro určení výšky terénu a tím vytváří rozdílné vrstvy krajiny.

Tento šum se často kombinuje s dalšími typy šumu, jako je například Perlinův šum nebo Simplexový šum, aby se dosáhlo více realistického vzhledu výsledné textury nebo terénu. Tyto kombinace mohou být velmi účinné, protože každý typ šumu přináší své vlastní vlastnosti, které se mohou navzájem doplňovat a vytvářet tak komplexnější efekty.

Jeho využití v procedurálním generování textur a terénu umožňuje dosáhnout realistických výsledků s relativně snadnou implementací.

2.6 Voronoiho šum

Voronoiho šum 2.5 je další druh šumu používaný pro procedurální generování světa. Tento typ šumu vychází z Voronoi diagramu, který je matematickým modelem pro rozdělení prostoru na nejblíže sousedící oblasti.

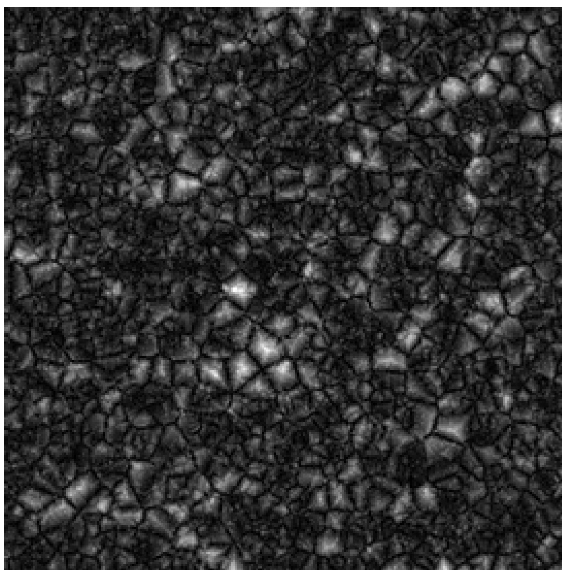
Voronoiho šum [41] se vyznačuje tím, že každý bod v prostoru má svůj nejbližší sousední bod, který určuje hodnotu v daném bodě. Výsledkem je textura, která se skládá z polygonů s různými tvary a velikostmi.

Pro výpočet Voronoiho šumu se používají náhodně generované body v prostoru, které se nazývají semena (seeds). Pro každý bod ve výsledné textuře se spočítá vzdálenost k nejbližšímu semenu. Tato vzdálenost se následně použije jako hodnota pro tento bod.

Voronoiho šum [30] se používá pro různé aplikace v počítačové grafice, například

pro generování terénu, textur nebo pro simulaci přirozených vzorů v přírodě. Tento druh šumu je oblíbený pro svou schopnost vytvářet přirozeně vypadající uspořádání objektů, které se hodí do celkového vizuálního stylu aplikace.

Nicméně, Voronoiho šum má své nevýhody. Jeho výsledná textura může být příliš "hrubá"nebo nepřirozená, pokud se nepoužívají další techniky pro vyhlazení. Zároveň může být náročné vytvořit vysoce detailní Voronoiho šum pro větší plochy.



Obrázek 2.5: Ukázka Voronoiho šumu

2.7 Parametry generování šumu

V procesu generování šumu existuje několik klíčových prvků, které ovlivňují celkovou texturu a vzhled výsledného obrazu. Každý z prvků má svou specifickou roli při utváření různých detailů. V následující části stručně popíšu jednotlivé prvky [20].

2.7.1 Oktávy

Oktávy představují počet průchodů algoritmem přes mapu. S vyšším počtem oktáv se provede více průchodů s různou frekvencí [20]. Každý z těchto průchodů přidává svůj příspěvek k výslednému šumu. Každá oktáva má svou vlastní frekvenci, která ovlivňuje, jak rychle se šum mění mezi jednotlivými body. Kombinováním různých hodnot oktáv můžeme dosáhnout bohatších textur na mapě.

2.7.2 Lacunarity

Lacunarity určuje násobný faktor frekvence mezi jednotlivými oktávami [20]. Příkladem pokud hodnota je 2.0, znamená to, že frekvence se bude násobit 2 pro každou

oktávu. První oktáva má nejnižší frekvenci a tvoří základní vzor. Druhá oktáva bude mít dvojnásobnou frekvenci, což přidává jemné detaily k základnímu vzoru. S vyšší lacunaritou (např. 6 nebo 8) se frekvence mezi jednotlivými oktávami více odlišuje, což vytváří složitější textury na mapě.

2.7.3 Persistence

Persistence určuje amplitudu každé oktávy po první [20]. S hodnotou persistence nižší než 1 bude vliv každé následující oktávy postupně klesat, což povede k hladšímu a méně detailnímu šumu. Naopak vyšší hodnota persistence (např. 2 nebo 3) bude mít za následek větší rozdíly mezi jednotlivými oktávami a výsledný šum bude hrubší na texturu.

2.7.4 Frekvence

Frekvence určuje, jak daleko jsou nejbližší hlavní rysy na mapě [20]. Vysoká frekvence znamená, že hlavní rysy jsou blízko sebe, což vytváří častější změny v nadmořské výšce a výsledkem jsou často hory nebo kopce. Naopak nízká frekvence znamená, že hlavní rysy jsou vzdálenější a výsledkem jsou rozlehlá údolí nebo roviny.

2.8 Porovnání šumů

Pro porovnání různých druhů šumů používáme různé kritéria, jako jsou:

Vzhled: Každý druh šumu má svůj vlastní vzhled. Porovnáním vzhledu různých druhů šumů lze zjistit, který druh šumu vytváří nejlepší vzhled pro procedurální generování světa.

Rychlost generování: Rychlost generování šumu je důležitá, zejména pokud se používá v real-time aplikacích, jako jsou počítačové hry. Různé druhy šumů mají různou rychlost generování.

Detaily: Dalším kritériem, kterým můžeme porovnat různé druhy šumů, jsou detaily. Některé druhy šumů jsou více detailní, což může být výhodné pro tvorbu textur a terénu. Na druhé straně, některé druhy šumů jsou méně detailní, ale mohou se lépe hodit pro generování obecných tvarů.

Kapitola 3

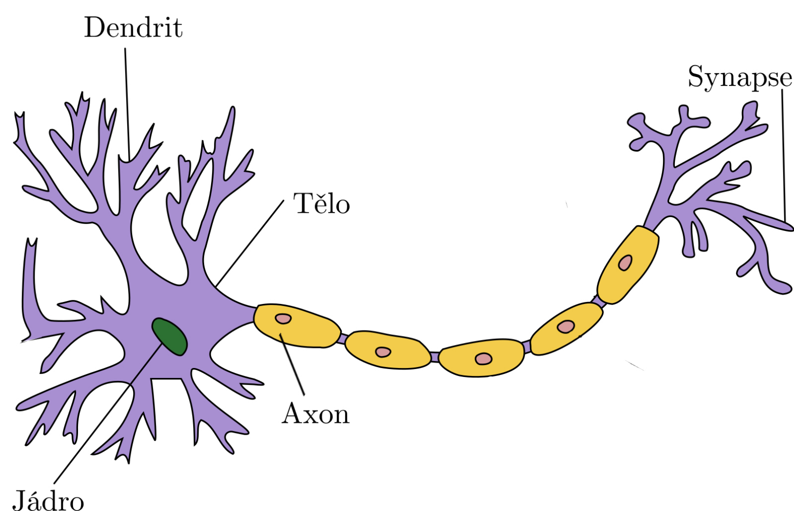
Neuronové sítě

3.1 Analogie neuronu

Neuronové sítě jsou matematické modely inspirované biologickými sítěmi nervových buněk. Proto je vhodné prvotně vysvětlit základní biologický neuron a jeho následnou analogii k umělému neuronu v neuronových síť.

3.1.1 Biologické neurony

Biologické neurony jsou specializované buňky, které mají schopnost přijímat, zpracovávat a přenášet informace v podobě elektrických impulzů. Každý neuron se skládá ze tří hlavních částí: buněčného těla (soma), dendritů a axonu. Ukázku biologického neuronu 3.1 zde přikládám.



Obrázek 3.1: Ukázka biologického neuronu [33]

Buněčné tělo obsahuje jádro a většinu buněčných organel. Zde se zpracovávají a integrují vstupní signály z dendritů a rozhoduje se o vygenerování výstupního signálu.

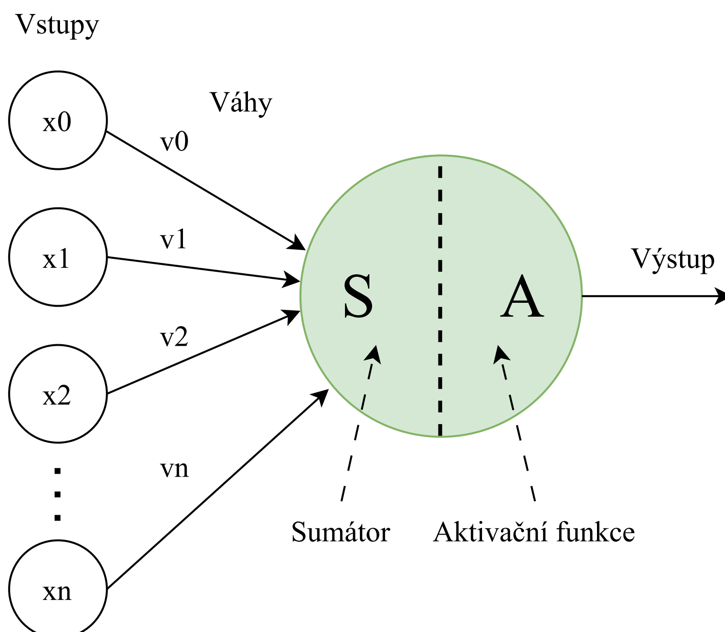
Dendrity jsou výběžky neuronu, které slouží k přijímání vstupních signálů od jiných neuronů nebo senzorů. Tyto signály jsou přenášeny v podobě elektrických impulzů, nazývaných akční potenciály.

Axon je výběžek neuronu, který slouží k přenosu informace do dalších neuronů nebo efektorů, jako jsou svaly nebo žlázy. Akční potenciály se šíří po axonu v podobě elektrických impulzů. Na konci axonu jsou synapse, které umožňují přenos signálů mezi neurony pomocí neurotransmiterů.

Komunikace mezi neurony probíhá prostřednictvím elektrických impulzů a neurotransmiterů. Když elektrický impulz dorazí do synapse na konci axonu, uvolní se neurotransmitery do prostoru mezi dvěma neurony, nazývaném synaptická štěrbina. Tyto neurotransmitery se vážou na receptorové molekuly na dendritech přijímajícího neuronu a přenášejí signál na další neuron.

3.1.2 Umělé neurony

Biologická neuronální struktura byla využita jako inspirace pro vytvoření umělých neuronů. Umělý neuron je matematický model, který simuluje základní funkce biologických neuronů 3.2. Jednoduchý model umělého neuronu se skládá ze vstupních vah, sumátoru, aktivační funkce a výstupu.



Obrázek 3.2: Ukázka úmělého neuronu

Vstupní váhy představují sílu a významnost vstupních signálů. Každý vstup je vážen určitou vahou, která ovlivňuje, jakým způsobem bude signál ovlivňovat výstup neuronu.

Sumátor sčítá vážené vstupní signály a vytváří váženou sumu. Tato vážená suma je poté předána aktivační funkci.

Aktivační funkce určuje, zda se neuron stane aktivním nebo ne. Pokud je dosažena určitá prahová hodnota, neuron generuje výstupní signál. Existuje mnoho různých typů aktivačních funkcí, které popisuji v dalších kapitolách.

Oba biologický a umělý neuron mají určité podobnosti, které jsou například:

- Biologický neuron přijímá vstupní signály od ostatních neuronů prostřednictvím dendritů. Stejně tak perceptron přijímá svá data od ostatních perceptronů pomocí vstupních neuronů, které přijímají čísla.
- Místa spojení mezi dendrity a biologickými neurony se nazývají synapse. Obdobně se spojení mezi vstupy a perceptrony nazývají váhy. Tyto váhy měří důležitost každého vstupu.
- V biologickém neuronu jádro produkuje výstupní signál na základě signálů poskytnutých dendrity. Stejně tak jádro v perceptronu provádí výpočty na základě vstupních hodnot a produkuje výstup.
- V biologickém neuronu je výstupní signál přenášen axonem. Stejně tak axon v perceptronu je výstupní hodnota, která bude vstupem pro další perceptrony.

3.2 Struktura neuronové sítě

Umělé neurony jsou propojeny do neuronové sítě, což je soubor vzájemně propojených umělých neuronů.

Signály se přenášejí mezi jednotlivými vrstvami sítě prostřednictvím propojení mezi umělými neurony. Každý neuron v jedné vrstvě je propojen s neurony ve vrstvě následující. Signály jsou přenášeny prostřednictvím vstupních vah a aktivací jednotlivých neuronů.

Tímto způsobem se informace postupně zpracovávají v neuronové síti a výstup je generován na základě vah a vzorců naučených během tréninku sítě.

3.3 Historie neuronových sítí

Historie neuronových sítí sahá až do roku 1943 [15], kdy Warren McCulloch a Walter Pitts vytvořili první model umělé neuronové sítě. Tento model byl inspirován biologickými nervovými buňkami a byl navržen pro řešení jednoduchých problémů v oblasti logiky. Model sestával z jednoduchých prvků, které byly schopny přijímat signály a vytvářet výstupy na základě těchto signálů. Tento model byl základem pro další vývoj neuronových sítí.

V 50. a 60. letech se začaly vyvíjet další modely neuronových sítí, jako například Perceptron od Franka Rosenblatta, který byl schopen klasifikovat jednoduché obrázky. Tento model se stal velmi populárním a byl využíván v různých aplikacích, jako například ve vojenských technologiích.

V 70. a 80. letech se začaly objevovat nové typy neuronových sítí, jako například Backpropagation a Hopfieldova síť. Backpropagation umožnila trénování neuronových sítí s více vrstvami a Hopfieldova síť byla schopna ukládat a obnovovat vzory. Tyto nové modely umožnily využití neuronových sítí v nových oblastech, jako například v rozpoznávání řeči a zpracování přirozeného jazyka.

V 90. letech se začaly používat rekurentní neuronové sítě, které byly schopny pracovat s časovými řadami a posloupnostmi dat. Tyto sítě byly například využívány v oblastech překladu řeči nebo vývoji robotů.

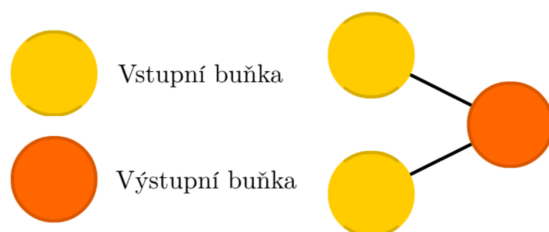
V posledních letech se neuronové sítě staly velmi populárními v oblasti strojového učení a umělé inteligence. Díky výpočetním výkonům moderních počítačů a množství dostupných dat se podařilo vytvořit složitější a úspěšnější modely neuronových sítí, které jsou schopné řešit různé problémy v oblasti umělé inteligence a strojového učení, jako například klasifikaci obrázků, rozpoznávání řeči, překlad přirozeného jazyka, predikce tržních trendů a mnoho dalších.

3.4 Typy neuronových sítí

Existuje mnoho typů neuronových sítí, z nichž každý má své vlastní specifické vlastnosti. V následujících podkapitolách zmíním nejvíce používané.

3.4.1 Perceptron

Perceptron je model umělého neuronu, který slouží jako učící algoritmus pro binární klasifikaci. Jedná se o nejjednodušší typ neuronových sítí, který v podstatě představuje jeden neuron. Pokud je použit jako architektura sítě, označuje se jako single-layer perceptron (SLP) s jednou vrstvou perceptronů [9]. Ukázka této sítě je na obrázku 3.3. Vícevrstvé perceptrony s více vrstvami perceptronů se označují jako multilayer perceptron (MLP) nebo jako feedforward neural network (FNN). Tyto sítě jsou využívány pro klasifikaci a regresi dat.

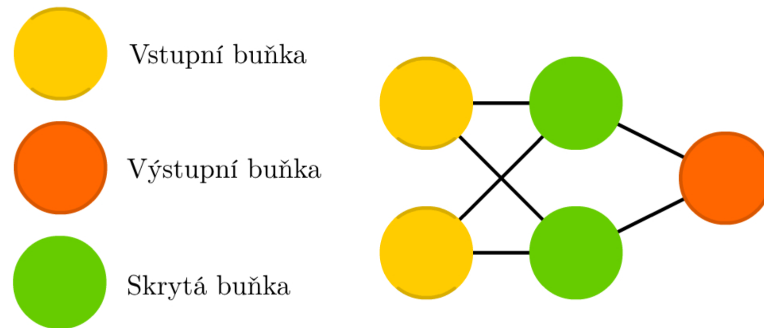


Obrázek 3.3: Ukázka jednovrstvého perceptronu [9]

3.4.2 Dopředné neuronové sítě (FFNN)

Dopředné neuronové sítě (FFNN) jsou sítě, kde neurony jsou uspořádány do vrstev. Signál se šíří jednosměrně od vstupních přes skryté až po výstupní neurony.

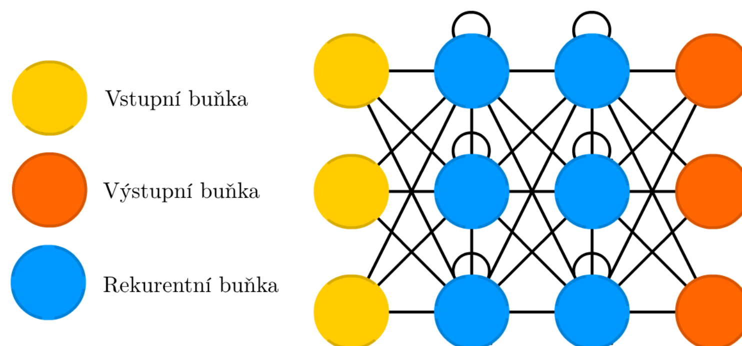
Pokud má tato síť více skrytých vrstev, označuje se jako hluboká dopředná neuronová síť (DFFNN). Na obrázku 3.4 ukazuji příklad této sítě.



Obrázek 3.4: Ukázka dopředné neuronové sítě [9]

3.4.3 Rekurentní neuronové sítě (RNN)

Rekurentní neuronové sítě (RNN) jsou podobné dopředným neuronovým sítím, ale mají vnitřní stav (paměť) [9], což jim umožňuje zpracovávat sekvence dat, jako jsou časové řady. RNN mají díky tomu schopnost uchovávat informace o předchozích krocích a používat je k predikci dalších kroků. Problémem RNN je mizející nebo explodující gradient, což vede ke ztrátě informace v čase. Pro řešení tohoto problému se používají sítě Long short-term memory (LSTM), které mají explicitní paměť a tři brány (vstupní, výstupní a zapomínací). Příkladem RNN sítě muže být následující obrázek 3.5



Obrázek 3.5: Ukázka rekurentní neuronové sítě [9]

3.4.4 Konvoluční neuronové sítě (CNN)

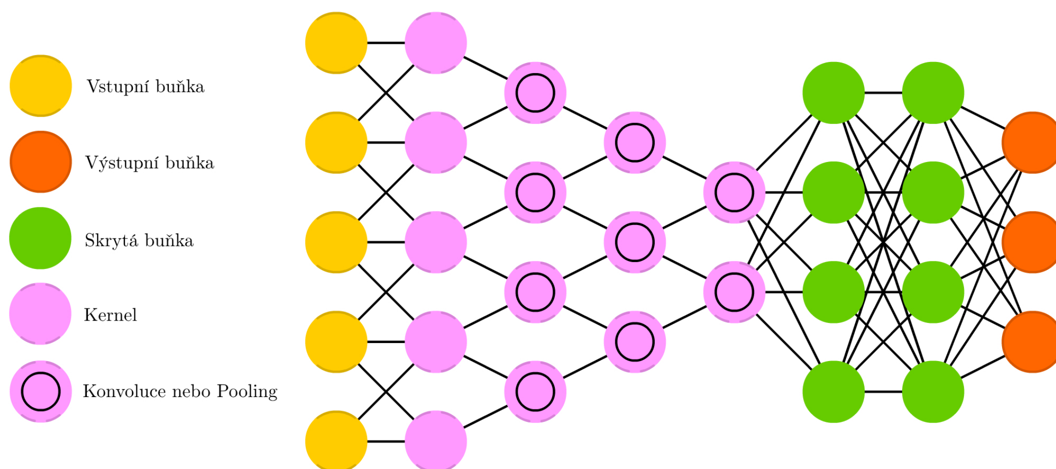
Konvoluční neuronové sítě (CNN) jsou vhodné pro zpracování obrazových dat. Používají konvoluční vrstvy k identifikaci různých vizuálních prvků na obrázku, jako jsou hrany, tvary, textury a další [9]. Pro zpracování velkých obrázků, které by byly pro klasické dopředné neuronové sítě nepraktické, jsou konvoluční sítě ideální.

CNN čtou obraz postupně po malých částech pomocí filtru konvolučního jádra (kernel) o určité velikosti. Filtr se posouvá po obrazu a hodnoty filtru se násobí s hodnotami obrazu pod ním. Součet těchto násobků vytváří hodnotu výstupní matice konvoluční vrstvy. Tímto způsobem je celý obraz postupně přečten. Pro posun se používá krok (stride), aby nedocházelo ke ztrátě informací, a okraje obrázku se ošetřují přidáním nulových hodnot (padding).

Výstupem konvoluční vrstvy je matice s velikostí jako vstupní obraz, na kterou lze aplikovat další konvoluce.

CNN obsahují několik konvolučních vrstev, které postupně zachycují důležité vlastnosti obrázku, od jednoduchých rysů jako jsou hrany nebo barvy v prvních vrstvách, přes textury a vzory v následujících vrstvách, až po komplexní objekty v posledních vrstvách. S konvulučními vrstvami se používají často i vrstvy pooling. Pooling je technika, která slouží k odstranění detailů. Častou metodou je max pooling, kde se v malé oblasti zachovává nejvyšší intenzita pixelu (například s největším množstvím červené). Příklad konvoluční neuronové sítě ukazuji na obrázku 3.6.

Pro mou bakalářskou práci jsem se zaměřil na využití konvolučních neuronových sítí, zejména s použitím metody GAN pro generování vizuálních dat.



Obrázek 3.6: Ukázka konvoluční neuronové sítě [9]

3.5 Učení neuronové sítě

Obecně neuron v neuronové síti provádí transformaci vstupních vektorů na výstupní hodnotu. To je proces, kdy se na základě vstupních dat neuron aktivuje a generuje výstup. Klíčovým prvkem neuronu jsou jeho parametry, které jsou na začátku konstantní. Adaptační dynamika je proces, který tyto parametry postupně upravuje tak, aby neuron dosahoval požadované transformace vstupních dat na výstup.

Proces učení neuronu probíhá iterativně na základě množiny dvojic vstupních hodnot a odpovídajících výstupů. Tyto dvojice představují trénovací data, na kterých se neuron učí. Adaptační algoritmy se zaměřují na adaptaci vah neuronu na základě těchto příkladů. Váhy neuronu slouží jako jeho paměť, ve které je zakódovaná zkušenost z předchozích trénovacích kroků.

Cílem adaptačních algoritmů je nalézt takové váhy neuronu, které umožní dostatečně obecnou transformaci dat, aby neuron byl schopen zpracovávat i nové, neznámé příklady dané oblasti. Algoritmy se inspirují známými příklady a hledají řešení na základě analogie, aby dosáhly co nejlepšího výkonu v učení.

3.5.1 Učení bez učitele

Existují situace, kdy nemáme k dispozici kritérium správnosti transformace vstupních dat, a přesto chceme, aby se neuronová síť naučila nějakým způsobem generalizovat a najít skryté vzory v datech. V takových případech se používá učení bez učitele [13].

Adaptační algoritmy pro učení bez učitele nepotřebují k dispozici korektní odpovědi, a proto se nezaměřují na přesné transformace dat. Namísto toho se algoritmy snaží nalézt podobné prvky ve vstupním prostoru dat a seskupit je do shluků. Takový přístup může pomoci odhalit strukturu a skryté vzory v datech, aniž bychom měli explicitní cíl nebo kritérium.

Adaptační algoritmy bez arbitra mohou být použity i během aktivní dynamiky sítě, což znamená, že síť může neustále učit a adaptovat se na nová data, aniž by potřebovala velké množství trénovacích dat.

3.5.2 Učení s učitelem

Učení s učitelem [13] je častým přístupem k učení neuronových sítí, zejména když máme k dispozici množinu dvojic vstupních dat a odpovídajících korektních výstupů.

V takovém případě jsou adaptační algoritmy schopny používat tyto příklady správného chování k tomu, aby se naučily, jak transformovat vstupní data na odpovídající výstupy. Trénovací a testovací množiny jsou využívány pro adaptaci a ověření výkonnosti neuronu nebo celé neuronové sítě.

Při trénování neuronové sítě dochází k postupnému optimalizování vah (tj. parametrů) jednotlivých neuronů tak, aby byla minimalizována chyba sítě při vstupu

určitého datového vstupu. Tento proces probíhá iterativně a zahrnuje několik kroků. Nejprve je potřeba definovat architekturu sítě, což zahrnuje počet neuronů v každé vrstvě, typy vrstev (např. konvoluční, plně propojené atd.) a způsob propojení neuronů mezi vrstvami.

Poté se předávají trénovací data síti, která jsou vstupem do sítě. V každé iteraci se vstup postupně předá sítí a výstup sítě se porovná s očekávaným výstupem. Tento rozdíl se nazývá chyba sítě.

Následně se provede zpětná propagace chyby, kdy se chyba sítě přepočte do jednotlivých neuronů v síti a na základě toho se upraví jejich váhy tak, aby se minimalizovala chyba při dalším průchodu sítě.

Provedení zpětné propagace chyby je založeno na algoritmu zvaném backpropagation. Tento algoritmus spočívá v tom, že se vypočítají parciální derivace chyby vůči váhám jednotlivých neuronů v síti a na základě těchto derivací se upraví hodnoty vah tak, aby se minimalizovala chyba sítě. Po provedení zpětné propagace chyby se upravené váhy použijí pro další iteraci trénování. Tento proces se opakuje pro všechna trénovací data, dokud není dosaženo dostatečného množství iterací nebo dokud není dosaženo požadované úrovně přesnosti sítě.

Během trénování se také často používají různé techniky regularizace, jako je například dropout, které pomáhají zabránit přetrénování sítě a zlepšit její obecnou schopnost generalizace.

Cílem tohoto procesu je minimalizovat odchylku, aby síť byla schopna co nejlépe aproximovat požadované vztahy mezi vstupem a výstupem.

Pro vyhodnocení úspěšnosti adaptace se používá validační množina, která slouží k zastavení adaptace, když chyba nad ní po určitou dobu neklesá. Testovací množina je potom použita k ověření výkonnosti naučeného neuronu nebo sítě, a jejich úspěšnost je kriteriem hodnocení adaptace.

V praxi se používá též dalších technik, jako je předtrénování sítě, které umožňuje vytvořit efektivní inicializaci vah a zlepšit výsledky trénování, a transfer learning, což je technika používající předtrénované sítě pro řešení nových problémů, což může urychlit trénování a zlepšit výsledky.

3.5.3 Zpětnovazební učení

Zpětnovazební učení je učení, kde agent v neznámém prostředí vybírá akce s cílem maximalizovat svůj užitek. Agent je umístěn do nějakého prostředí a pozoruje jeho stav, který může ovlivnit provedením akce. Vykonáním akce změní agent stav prostředí a dostane odměnu. Na základě této zpětné vazby si agent aktualizuje své znalosti, které používá při provádění dalších akcí. Agent disponuje omezenými informacemi pro rozhodování v daném okamžiku a bere v úvahu pouze částečnou zpětnou vazbu, což je odlišností od učení s učitelem. Cílem agenta je tedy maximalizovat svoji odměnu a tím dosahovat požadovaného výsledku.

Kapitola 4

Návrh programu

4.1 Výběr technologií

V mé práci jsem se rozhodl pro použití jazyka Python. Před započetím mé práce jsem zhodnotil, že Python nabízí různé výhody pro tvorbu GAN modelů.

Jednou z hlavních výhod Pythonu pro GAN bylo pro mě jeho velký ekosystém knihoven, včetně TensorFlow, PyTorch a Keras, které poskytují vysokoúrovňová API, která zjednodušují implementaci GAN modelů a usnadňují trénování neuronových sítí. Kromě toho tyto knihovny nabízejí předtrénované modely, které lze využít pro konkrétní úlohy generování obrazů, což šetří vývojářům cenný čas.

Také má snadno použitelnou syntaxi, která je přístupná i pro začátečníky v oblasti strojového učení a hlubokého učení. To umožňuje výzkumníkům a vývojářům prototypovat své nápady a testovat své modely rychle bez starostí o detaily na nízké úrovni. Kromě toho má Python aktivní komunitu vývojářů, kteří přispívají do jeho knihoven, nástrojů a frameworků, poskytují podporu, tutoriály a příklady, které pomáhají vývojářům efektivně začít pracovat s GAN.

Mimo jiné má také vynikající kompabilitu s jinými programovacími jazyky a nástroji, což usnadňuje integraci modelů GAN s jinými softwarovými komponenty, jako jsou databáze, webová rámcování a další modely strojového učení.

I přestože jiné programovací jazyky, jako je C++, Java a MATLAB, jsou vhodné pro generování obrázků pomocí GAN, tak tyto jazyky mají omezenější knihovny pro hluboké učení, což může být obtížnější pro implementaci složitějších GAN modelů. Nemají předtrénované modely pro konkrétní úlohy generování obrázků a mají menší komunity ve srovnání s Pythonem, což vede mimo jiné ke snížené podpoře a omezeným příkladům. Pokud jde o vývoj aplikací v Pythonu, lze vybrat z široké škály nástrojů k výběru, včetně textových editorů, integrovaných vývojových prostředí (IDE) a specializovaných platforem navržených pro konkrétní účely.

Zvolil jsem si jeden ze známějších Python IDE a to PyCharm, který nabízí řadu výhod oproti jiným nástrojům, které jsou například:

1. Inteligentní kódový editor: PyCharm nabízí inteligentní kódový editor, který

poskytuje dokončování kódu, zvýraznění chyb a analýzu kódu. Zrychluje tedy psaní kodu s menším počtem chyb.

2. Integrace s nástroji: PyCharm se lehce integruje s nástroji, jako je například Git. To mi umožnilo pracovat s přehledem předchozích verzí mého programu na githubu a usnadnilo hledání případných chyb.

3. Přizpůsobitelné rozhraní: Rozhraní PyCharmu je velmi přizpůsobitelné, což umožňuje vývojářům vytvořit personalizované vývojové prostředí, které vyhovuje jejich potřebám.

Kromě obecných výhod pro vývoj v jazyce Python nabízí PyCharm také několik funkcí, které ho dělají dobře přizpůsobený pro vývoj neuronových sítí.

1. Podpora knihoven: PyCharm podporuje knihovny specifické pro dělání neuronových sítí jako TensorFlow, Keras a PyTorch.

2. Vizuální ladící nástroj: vizuální ladící nástroj PyCharm umožnuje vidět strukturu a chování neuronových sítí, což mi usnadnilo ladění složitějších modelů.

3. Pomoc s kódem pro hluboké učení: PyCharm poskytuje pomoc s kódem, včetně automatického dokončování a zvýraznění chyb pro funkce.

4.2 Návrh systému

Generování map je jedním z hlavních cílů této práce. Pro tento účel byla zvolena neuronová síť typu GAN (Generative Adversarial Network).

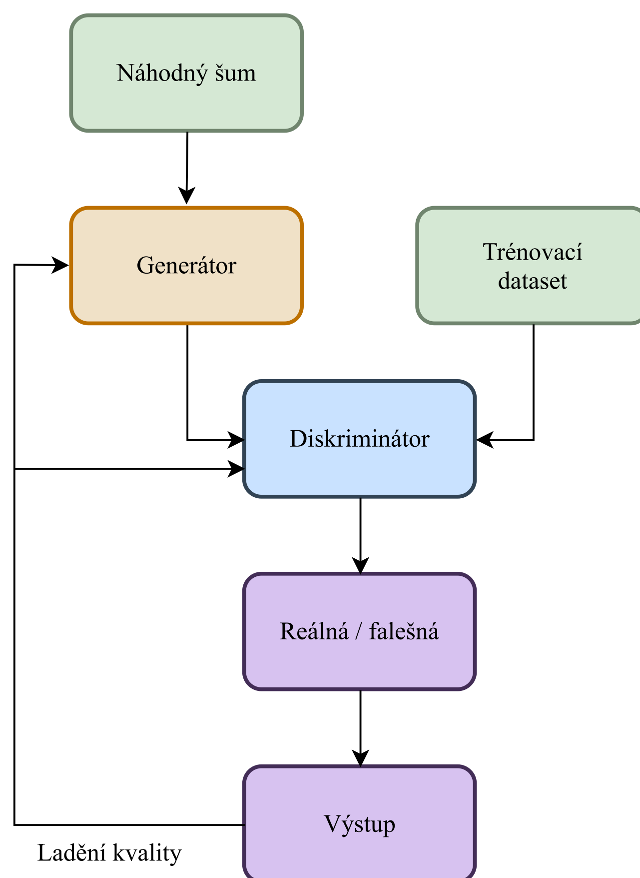
Koncept této architektury sítě se zakládá na dvou neuronových sítí generátoru a diskriminátoru.

Diskriminátor vyhodnocuje a odhaduje pravděpodobnost toho, zdali jeho vstup pochází z trénovacích dat. Diskriminátor je trénován na trénovacím datasetu. Trénovaný diskriminátor je testován tím, že jsou do něj vloženy data z generátoru a z trénovací sady.

Generátor generuje nová data (padělky) přepracováním náhodného vektoru šumu na nový šum podobný příkladům v trénovací sadě, aniž by přímo viděl trénovací data. Data pak posílá do diskriminátoru, který se snaží klasifikovat, zda vzorek pochází z trénovacích dat a je skutečný, nebo falešný.

4.3 Vrstvy GAN

Výsledkem je vzájemná dynamika, kdy oba modely postupně zlepšují své schopnosti, což nakonec vede k vytvoření realistických a kvalitních padělků. Výslednou generaci mohu po naučení sítě a vygenerování vah získat zpětném načtením uložených vah do generátoru. Přikládám obrázek 4.2 pro lepší vizualizaci jednotlivého průběhu sítě.



Obrázek 4.1: Ukázka GAN průběhu

V této části ukazuji jednotlivé vrstvy pro generátor a diskriminátor, ktere se dali využít pro mou práci.

- Lineární vrstvy (Dense): Tyto vrstvy přijímají vstupní šum a transformují ho na vektor s vyšší dimenzionalitou. Používají se různé aktivační funkce, jako například ReLU nebo LeakyReLU, pro přidání nelinearity do modelu.
- Normalizace (BatchNormalization): Normalizační techniky se často používají k vyrovnání rozložení dat a zlepšení stability a konvergence modelu. Normalizace se provádí na každý vstup do vrstvy nebo skupinu vrstev.
- Konvoluční vrstvy (Conv2D): Konvoluční vrstvy se používají k extrakci různých vlastností a informací z vstupních dat. Používají se různé aktivační funkce, jako například ReLU nebo LeakyReLU, pro přidání nelinearity.
- Konvoluční transpozice (Conv2DTranspose): Tato technika umožňuje upsampling dat, čímž se zvyšuje jejich rozměr a přibližuje se k požadovanému výstupnímu rozlišení. Různé parametry, jako velikost kernelu, krok a padding, mohou být nastaveny podle potřeby.
- Aktivační funkce: Každá vrstva generátoru je následována aktivační funkcí, která zavádí nelinearitu do modelu. Používají se různé aktivační funkce, jako

například ReLU, LeakyReLU, sigmoid nebo tanh, v závislosti na konkrétním problému.

- Dropout vrstva: Náhodně nastavuje vstupní jednotky na hodnotu 0 s frekvencí *rate* během každého kroku trénování. Například pro $rate = 0.2$ bude pravděpodobnost deaktivace každé jednotky 20%. Tím pomáhá předcházet přeučení modelu. Vstupy, které nejsou nastaveny na 0, jsou zvětšeny o $\frac{1}{1-\text{rate}}$, aby se zachovala suma všech vstupů.

- Výstupní vrstva: Generátor má výstupní vrstvu, která generuje finální výstup modelu. Pro můj projekt, kde se generují obrázky se používal výstup s aktivací tanh, která omezí hodnoty pixelů na určitý rozsah, například [-1, 1].

  U diskriminátoru je výstup zploštěn do jedné dimenze a předán do plně propojené vrstvy s jediným výstupem, která slouží k rozhodování, zda jsou vstupní data skutečná nebo falešná. V tomto případě je často používána sigmoidní aktivace, která vrací hodnoty v rozmezí [0,1]. Použití sigmoidu je dostačující, protože diskriminátor pouze informuje generátor, zda jsou data skutečná nebo falešná, a není tedy potřeba využívat širší rozsah hodnot.

4.4 Modifikace GAN

V mé práci budu využívat pouze obyčejný GAN, nicméně existují různé varianty GANů, které mohou zlepšit jednotlivé aspekty generování. V této podkapitole zmíním některé z nich, které jsou známější.

4.4.1 Conditional GAN

Conditional Generative Adversarial Network neboli cGAN přidává oproti obyčejnému GANu do generátoru další informace, jako jsou třídy nebo štítky. To znamená, že během tréninku se generátoru přidávájí obrázkům i jejich štítky například pro rostliny (růže, tulipán, slunečnice ....) díky kterým se síť naučí rozpoznávat rozdíl mezi nimi. Tímto získáme schopnost požádat model, aby generoval obrázky konkrétního druhu. Příkladem kromě rostlin můžou být obrázky s uměleckým stylem či generování hudby na základě určitých charakteristik nebo hudebních stylů.

4.4.2 Wasserstein GAN

Hlavní rozdíl mezi WGAN a GAN spočívá v použití ztrátové funkce během tréninku. V původním GANu jsou generátor a diskriminátor trénovány pomocí Jensen-Shannonovy divergence nebo Kullback-Leiblerovy divergence. Tyto ztrátové funkce mohou však trpět mizícími gradienty a mode collapse, což znamená, že generátor nedokáže produkovat různorodé vzorky.

Pro překonání těchto problémů WGAN používá Wassersteinovu vzdálenost jako ztrátovou funkci. Wassersteinova vzdálenost měří, kolik "hmoty"je potřeba přemístit, aby se jedna distribuce převedla na druhou. V kontextu GANů kvantifikuje rozdíl mezi distribucí reálných dat a distribucí generovaných dat.

Tato síť se převážně používá pro práci s komplexnějšími a náročnějšími daty, kde může vznikat nestabilita tréninku a špatná kvalita generovaných dat.

4.4.3 Style GAN

StyleGAN byl navržen tak, aby poskytoval jemnější kontrolu nad vzhledem generovaných obrázků přičemž cílová funkce a diskriminátor zůstávají stejné jako u běžného GANu, pouze se mění architektura generátoru. Tímto způsobem síť umožňuje manipulovat s různými aspekty stylů, jako je jas, barva a textura. Díky této vlastnosti může uživatel lépe ovládat výstupy generátoru a dosáhnout požadovaných vizuálních efektů.

4.5 Vizualizace

Samotná implementace Generátoru a Diskriminátoru je klíčovou částí. Před tím, než jsem se pustil do implementace, bylo důležité nejprve promyslet a rozvrhnout strukturu obou částí GANu, abych dosáhli co nejlepších výsledků.

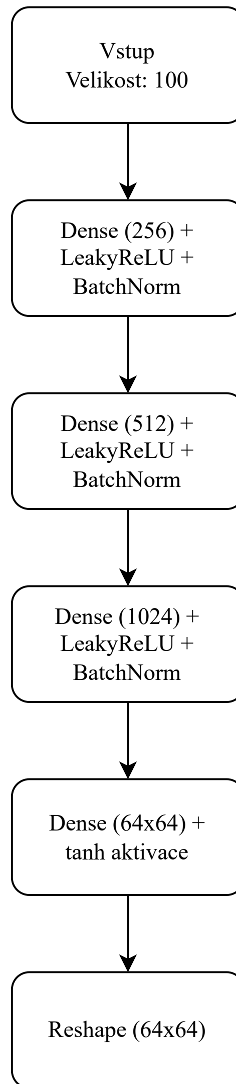
4.5.1 Generátor

Pro generátor jsem navrhl vstupní vrstvu, která bude přijímat šum v rozměru 100. Dále aplikuji plně propojenou Dense vrstvu, která vezme vstupní náhodný šum a převede ho na určitý počet neuronů, v mém případě zvolím 256. Větší počet neuronů poskytne generátoru větší prostor pro ukládání informací o vzorcích a strukturách, které má generovat. Díky tomu bude generátor schopen dosáhnout větší komplexnosti generovaných dat. Po Dense vrstvě následuje LeakyReLU vrstva s parametrem alpha=0.2, což umožní generátoru generovat složitější vzorce a struktury.

Pro další vrstvy zvolím postupně zvětšující se počet jednotek v Dense vrstvách (512 a 1024). Generátor tím bude mít dostatečně velký prostor pro ukládání informací o vzorcích a strukturách, které má generovat. Zároveň přidávám pro každou vrstvu Dense BatchNormalization s parametrem momentum=0.8, což pomáhá stabilizovat učení generátoru a urychluje konvergenci.

Na závěr generátoru přidávám plně propojenou Dense vrstvu s aktivací tanh a reshape vrstvu, která změní tvar výstupu z husté vrstvy na 2D obraz, například velikosti 64x64.

Na následujícím obrázku 4.5.1 ukazuji diagram, jak by má síť mohla probíhat. Je důležité zmínit, že při praktickém využití sítě budu měnit a přidávat jednotlivé vrstvy pro generování co nejvěrohodnější mapy. Nejedná se o finální uspořádání.



Obrázek 4.2: Ukázka návrhu generátoru pro mou práci

4.5.2 Diskriminátor

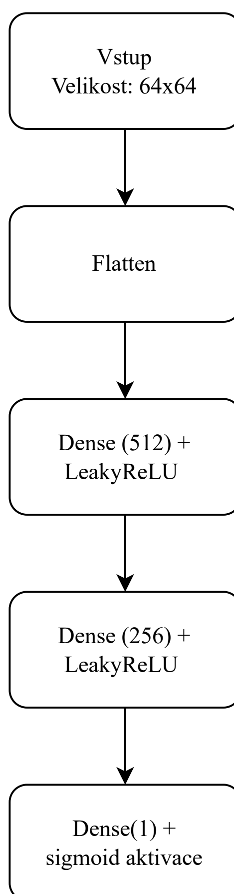
Pro diskriminátor jsem navrhl architekturu kde vstupní vrstva přijímá obraz o rozměrech 64x64. Poté následuje vrstva Flatten, která rovná vstupní obraz a připravuje ho pro další zpracování.

Následuje plně propojená Dense vrstva s 512 jednotkami, která má za úkol zpracovat data z ploché vrstvy. Pro aktivaci využiji funkci LeakyReLU s parametrem alpha=0.2, která pomáhá dosáhnout lepší stability a umožňuje diskriminátoru lépe rozpoznávat vzorce a struktury.

Další vrstvou je plně propojená Dense vrstva která obsahuje 256 jednotek a opět využívá aktivační funkci LeakyReLU s parametrem alpha=0.2.

Na závěr jsem udělal výstupní plně propojenou Dense vrstvu s jednou jednotkou a aktivací sigmoid. Tato vrstva vrací hodnotu v rozmezí 0 až 1, která reprezentuje pravděpodobnost, že vstupní obraz je reálný namísto generovaný generátorem.

Na následujícím obrázku 4.3 ukazuji diagram, jak by má síť mohla probíhat.



Obrázek 4.3: Ukázka návrhu diskriminátoru pro mou práci

Kapitola 5

Aplikace a implementace

5.1 Výběr výchozího vzorku

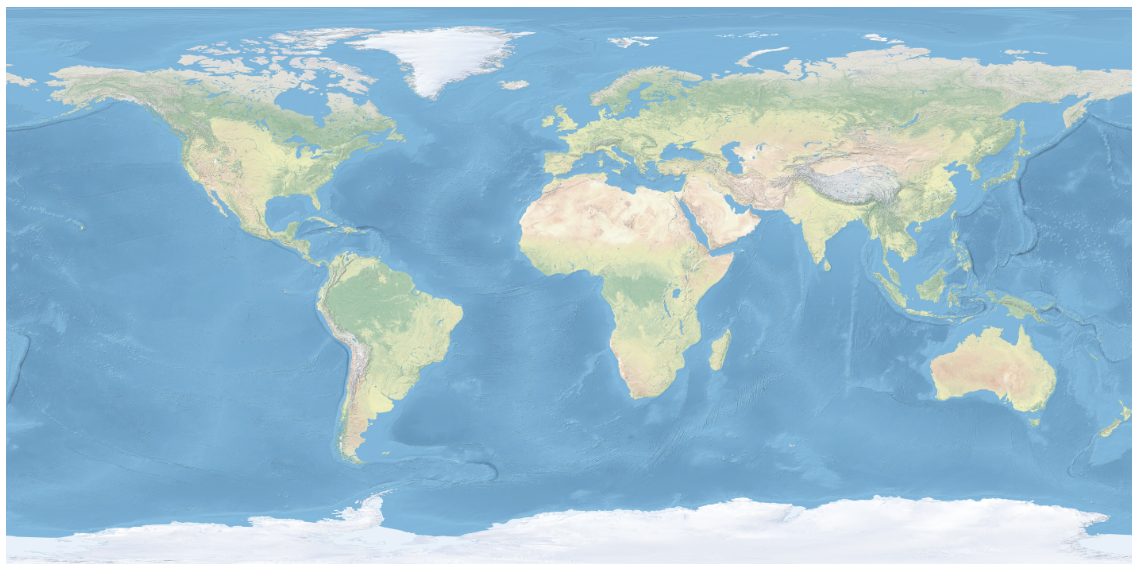
V této části se zaměřím na proces generování vhodného datasetu pro mou práci. Vzhledem k omezené dostupnosti existujících dat jsem se rozhodl vytvořit vlastní dataset pomocí technik augmentace dat. Popíšu zde podrobnosti o postupu, který jsem použil k získání obrázků pro trénování a testování mé modelové sítě.

Prvním krokem při generování datasetu byl pečlivý výběr výchozího vzorku dat, který by odpovídal potřebám mé práce. Chtěl jsem získat kolekci obrázků, které mají relevanci pro dané téma a obsahují klíčové prvky pro správné trénování mého modelu.

Při tomto výběru jsem se zpočátku zaměřil na barevné obrázky o menších velikostech 64x64 px. Důvodem volby menších rozměrů obrázků byla výpočetní náročnost učení modelů GAN sítí. Učení těchto sítí vyžaduje značné množství výpočetních prostředků a časových zdrojů. S využitím menších obrázků jsem mohl snížit nároky na výpočetní kapacitu a zrychlit celý proces učení modelu. Po zjištění správnosti generování jsem zvýšil velikost na 256x256 pro zvýšení parametrů a zlepšení detailů vůstupních dat. Při výběru výchozího vzorku jsem také bral v úvahu dostupné datasety bez licenčních omezení. Prozkoumal jsem různé zdroje, které poskytovaly vhodné obrázky, nicméně jsem nenarazil na mé požadovaná kritéria, jako například množství obrázků v datasetu nebo kvalita obrázků. Kvůli tomu jsem se rozhodl vytvořit vlastní dataset z veřejně dostupné rastrové mapy. Tím jsem zajistil dostatečný a přesně přizpůsobený výchozí vzorek dat pro další fázi generování datasetu. Přikládám obrázek 5.1 od "Natural Eath data", který jsem využil pro generování mého datasetu.

5.2 Úprava datasetu

Během testování s volně dostupnými datasety jsem narazil na několik problémů. Jedním z nich byl například to, že obrázky měly odlišná rozlišení, což znamenalo,



Obrázek 5.1: Ukázka mapy pro tvoření datasetu

že jeden obrázek mohl mít rozměry 600x600 pixelů, zatímco další mohl mít 900x900 pixelů. Dalším problémem byl nerovnoměrný poměr stran, který způsoboval, že obrázky nebyly ve formátu čtverce. Nebo například některé obrázky v datasetu byly černobílé. Bylo tedy nutné provést úpravy obrázků před jejich načtením, protože jsem potřeboval co nejvíce konzistentních obrázků v datasetu. Abych řešil tyto problémy, využil jsem knihovnu pro úpravu obrázků v Pythonu nazvanou PIL (Python Imaging Library).

Prvním krokem bylo zajistit, že obrázek je ve formátu RGB, což je nejčastěji používaný formát pro obrázky. K tomu jsem použil metodu convert('RGB'), která převede obrázek do daného formátu.

Následně jsem provedl ořezání obrázků tak, aby měly stejnou velikost. Použil jsem metodu image.crop((left, top, right, bottom)), kde jsem určil obdélníkovou oblast, kterou jsem chtěl zachovat. Specificky jsem zvolil oblast tak, aby její rozměry odpovídaly menší straně původního obrázku. Tímto způsobem jsem dosáhl, že všechny oříznuté obrázky měly stejnou velikost.

Dalším krokem byla změna velikosti obrázků na rozměry 256x256 pixelů. K tomu jsem použil metodu image.resize((256, 256)), která umožňuje nastavit požadovanou velikost obrázků. Díky tomu měli všechny obrázky v datasetu jednotnou velikost pro další zpracování.

Existuje také řada dalších knihoven, které jsem mohl použít pro úpravu obrázků. Například knihovna OpenCV poskytuje podobné funkce pro manipulaci s obrázky a je často využívána v oblasti zpracování obrazu. Další možností byla knihovna scikit-image, která nabízí různé algoritmy pro úpravu a zpracování obrazu. Nicméně, rozhodl jsem se použít knihovnu PIL, protože mi přišla relativně intuitivní a jednoduchá.

5.3 Rozšíření datasetu

Pro zvýšení množství dat v mém konkrétním datasetu jsem využil technik augmentace. Tyto techniky jsem aplikoval na výchozí vzorek dat, abych vytvořil nová data pro dataset a také jsem umožnil modelu pracovat s různými variantami dat, které by se mohly vyskytovat v reálných situacích.

Jednou z použitých technik bylo ořezání obrázků na různé části, otáčení a převrácení nebo zvyšování a snižování kontrastu a sytosti, kde jsem opět využil knihovnu PIL. Tímto způsobem jsem získal mnohem více vstupů při malých úpravách, model se díky tomu naučil rozpoznávat objekty nejen z různých perspektiv, ale také v různých orientacích. Obrázky byly vybírány náhodně z velkého zdroje, což výrazně snížilo pravděpodobnost střetu stejných obrázků a zajišťovalo tak jejich plnou náhodnost.

Při generování datasetu jsem zvažoval další techniky augmentace, které by mohly přinést rozmanitost a zlepšit schopnost modelu generovat kvalitní obrázky. Mezi tyto techniky patří:

1. Přidání šumu: Přidání náhodného šumu do obrázků, jako je gaussovský šum nebo šum typu salt and pepper, který by mohl zlepšit odolnost modelu vůči šumu přítomnému v reálných datech.
2. Geometrické deformace: Aplikace různých geometrických transformací, jako je zkosení, stlačení, roztažení.
3. Kombinace textur: Kombinace různých textur z různých zdrojů, které mohou vytvořit unikátní vzory a povrchy.

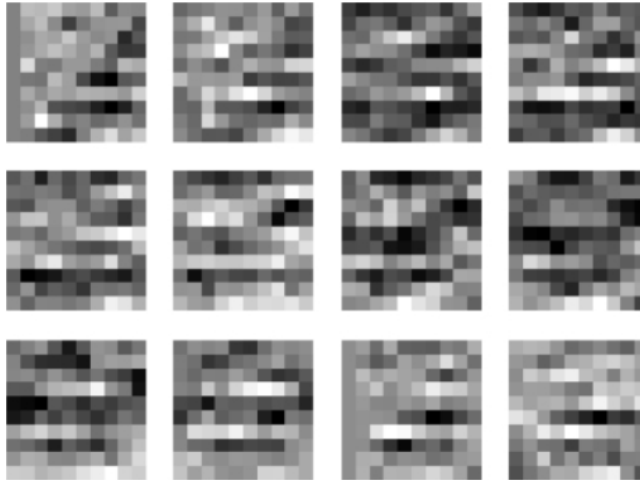
Je třeba poznamenat, že tyto techniky augmentace byly zvažovány, ale vzhledem k dostatečnému množství dat a výpočetnímu omezení jsem je v rámci mé práce nepoužil. Nicméně jsou to další možnosti, které by bylo vhodné zvážit pro budoucí rozšíření a vylepšení generování datasetu. V této kapitole se zaměřím na průběh učení při procedurálním generování sítí pomocí GAN. Představím zde své experimenty a výsledky, které jsem získal při testování mé sítě na generování datových sad.

5.4 Počáteční testování parametrů sítě

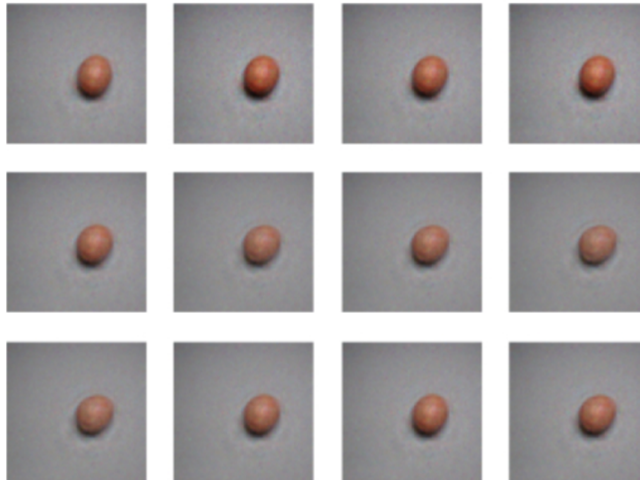
V úvodní fázi mého výzkumu jsem se rozhodl provést testování mé sítě na generování datasetu obsahujícího vajíčka. Tento krok byl zaměřen na získání počátečního porozumění základních principů a částí generátoru. Cílem bylo také zhodnotit fungování sítě a její schopnosti při generování obrazových dat.

V průběhu dalších experimentů jsem dospěl k závěru, že je vhodnější využít známý dataset MNIST, který obsahuje číselné obrázky. Proto jsem se rozhodl provést další testy a ověřit, zda síť trpí "mode collapse", tedy jevem, kdy generátor produkuje pouze omezený počet podobných vzorů. Mode collapse ukazuji i na obrázku 5.2 původního datasetu s vajíčky na obrázku.

Vstupní Perlinův šum



Výstup po 10000 epochách



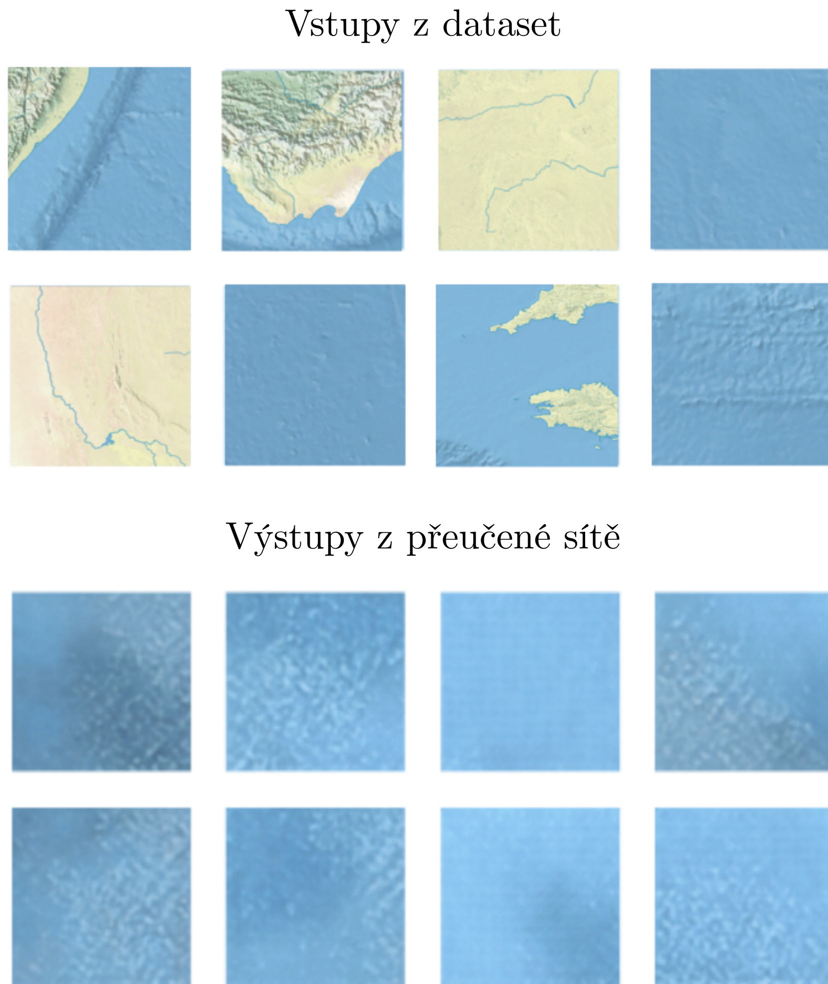
Obrázek 5.2: Ukázka mode collapse na datasetu s vajíčky

"Mode collapse"nastává, kdy se generátor naučí generovat pouze omezenou různorodost vzorů, čímž ztrácí schopnost vytvářet pestrou škálu vstupních dat. Tento jev může nastat, pokud se síť příliš zaměří na nejčastější a dominantní vzory ve vstupním datasetu.

5.5 Problémy a úpravy pro zlepšení sítě

Po získání počátečního porozumění principům mé sítě jsem se přesunul k samotnému procesu učení mého datasetu. Při prvotním učení jsem narazil na problém kterým

byl ten, že síť se přeučila na generování pouze vodních ploch a produkovala obrázky s touto konkrétní charakteristikou 5.3. Tento jev může nastat, zejména pokud v datasetu není dostatek různorodých dat, což omezuje schopnost generátoru vytvářet různé typy vzorů. Je však důležité zdůraznit, že můj dataset obsahoval 5000 obrázků, takže nedostatek dat nebyl zásadním problémem.



Obrázek 5.3: Ukázka líného učení GAN na vodních plochách

Hlavním problémem, se kterým jsem se potýkal, bylo vyšší zastoupení vodních ploch ve vstupním datasetu ve srovnání s pozemními plochami. Síť se tedy zaměřila na generování vodních ploch, protože to byla pro ni nejjednodušší cesta k dosažení pozitivního ohodnocení ze strany diskriminátoru. Tento nepříznivý poměr výskytu jednotlivých typů ploch v datasetu přispěl k přeučení sítě na generování specifického typu vzorů a omezení její schopnosti produkovat rozmanitou škálu výstupních dat. Proto jsem se rozhodl brát v úvahu pouze určitou plochu obrázku, která obsahuje dostatečně různorodé detaily. Na obrázku 5.4 lze vidět mapu o velikosti 1000x1000 pixelů, kterou jsem použil pro konečné výstupy.

Obrázek 5.4: Ukázka zvolené mapy o velikosti 1000x1000 pixelů

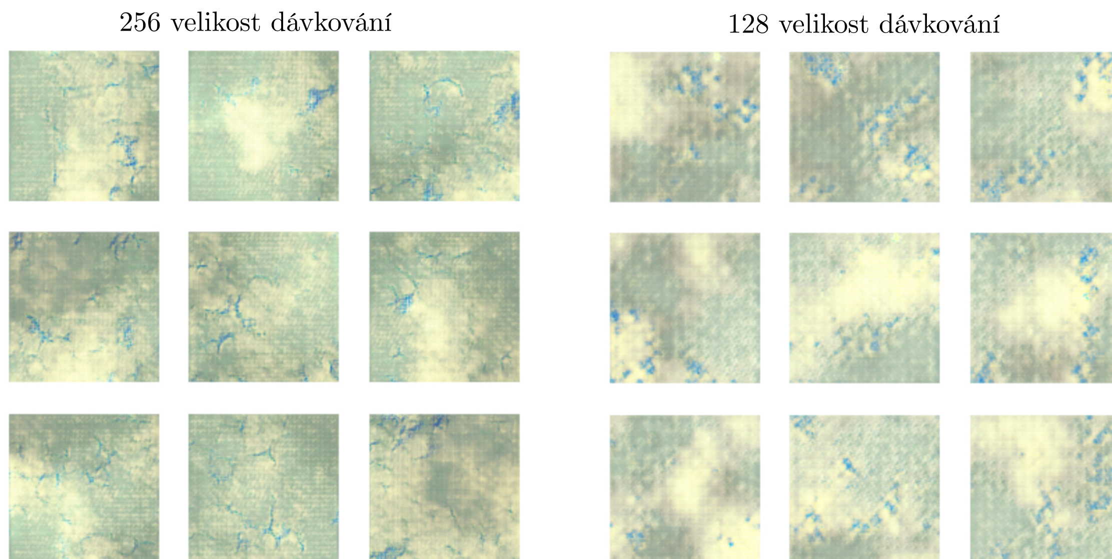
Další významnou úpravou mé sítě bylo trénování na rozšířeném datasetu s rozlišením 256x256 pixelů, na rozdíl od původního datasetu s rozlišením 64x64 pixelů. Vyšší rozlišení datasetu umožnilo síti zachytit drobné nuance a textury, což vedlo k vytváření realističtějších a přesnějších vizuálních výstupů.

Současně jsem při trénování upravoval i velikost dávkování (batch size). Zvýšení velikosti dávkování mi umožnilo získat větší rozmanitost vzorů během každé iterace učení. To mělo za následek zlepšení schopnosti sítě generovat lepší vzory. Nicméně, obě tyto změny, tedy zvýšení rozlišení datasetu a velikosti dávkování, způsobily pomalejší tempo učení sítě. Přikládám obrázek 5.5 při generaci map pomocí náhodného šumu pro 128 velikosti dávkování a 256 velikosti dávkování. Díky přístupu ke školnímu serveru jsem však mohl trénování nechat běžet i přes noc a využít jeho výpočetní kapacity.

5.6 Použití šumů v praxi

V rámci mého projektu jsem využíval Perlinův a Simplexový šum na několika místech k zobrazení rozdílu a následnému porovnání. Díky různé možnosti nastavení parametrů, jako je škála, lucarita, perzistence a oktavy jsem mohl přizpůsobit generování specifickým potřebám.

Pozoroval jsem, že Perlinův a Simplexový šum vykazují větší počet iterací a zabírají více času při generování přijatelnějších výstupů než u náhodného šumu. Toto může být způsobeno špatným nastavením parametrů, které jsem použil. I přesto, že jsem provedl určité změny, výsledky byly spíše horší. Jak Perlinův tak Simplexový šum



Obrázek 5.5: Porovnání generování při stejném počtu epoch na 128 a 256 velikosti dávkování

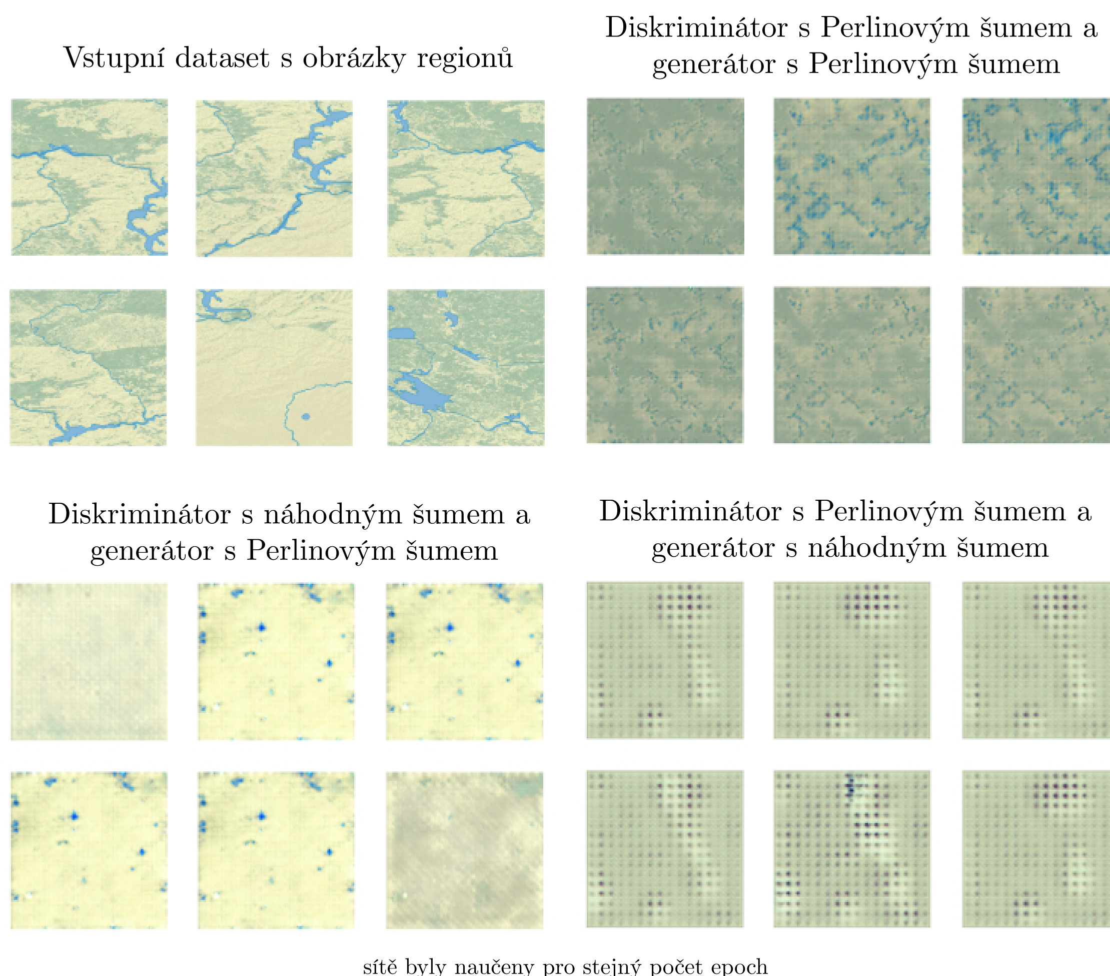
jsou deterministické šumy, což znamená, že při stejných vstupních parametrech vždy generují stejný výstup. Tato vlastnost je při generování obrazu výhodná, protože vede k větší konzistenci výsledků oproti náhodnému šumu.

Po vyzkoušení těchto druhů šumu jsem se zaměřil na zkoumání vlivu použití šumu pouze pro generátor a poté pouze pro diskriminátor. Během tohoto experimentu jsem sledoval, jaké výsledky byly dosaženy pomocí různých metod a zaznamenával jsem délku trvání a přesnost obrázku po 20000 epochách. Na obrázku 5.6 můžeme vidět jednotlivá chování perlinového šumu. Je zřejmé, že žádný výsledek není dokonalý, což pravděpodobně způsobuje menší míra tréninku sítě.

Nicméně již nyní je patrné, že když použijeme perlinový šum jak pro generátor, tak pro diskriminátor, výsledné data se vizuálně nejvíce přibližují našemu referenčnímu datasetu. Zaznamenal jsem také, že když je perlinový šum aplikován pouze na generátor a diskriminátor zůstane na náhodném šumu, generování obrázků vyžaduje výrazně více epoch.

Při posledním pokusu, kdy byl perlinový šum aplikován pouze na generátor, jsem pozoroval opakování stejných obrázků při zvyšování epoch, které diskriminátor klasifikoval s 100% přesností. Existuje několik možných důvodů pro tento jev, jako například nedostatečná komplexnost generátoru nebo neoptimální nastavení parametrů trénování. Nicméně jsem se rozhodl ponechat parametry sítě stejné, abych mohl sledovat vliv generování při konzistentních podmínkách.

Obdobné výsledky jsem dostával také u simplexového šumu.



Obrázek 5.6: Ukázka trénování pomocí Perlinového šumu

5.7 Implementace generátoru

Po aplikaci mého návrhu generátoru jsem zjistil, že nevyhovuje přesně mému zadání. V následující části ukazuju použitý kod, který zaručil generování výstupu, které uvádím v závěru práce.

Prvotně vytvářím instanci modelu Sequential, která slouží k postupnému přidávání vrstev do modelu.

```
---------------------------------------
def build_generator(self):
    model = Sequential()
---------------------------------------
```

Dále přidávám vrstvu Dense, která je plně propojená s 128 * 16 * 16 neurony a jako vstup přijímá latentní rozměr. Vrstva slouží k transformaci vstupního šumu na husté reprezentace.

Následuje vrstva LeakyReLU, která aplikuje leaky rectified linear unit aktivaci s parametrem alpha=0.2, která přidává nelinearitu do modelu.

Poté jsem přidal vrstvu Reshape, která přetvaruje výstup z předchozí vrstvy na tvar (16, 16, 128). Zmenšování a následné zvětšování vrstev v generátoru jsem uskutečnil to, že generuju detailnější a kvalitnější obrázky postupným rozšiřováním prostorového rozlišení a využíváním informací z předchozích vrstev.

```
-----------------------------------------------------------
model.add(Dense(128 * 16 * 16))
model.add(LeakyReLU(alpha=0.2))
model.add(Reshape((16, 16, 128)))
-----------------------------------------------------------
```

V navazujícím kroku přidávám vrstvu Conv2DTranspose, která provádí dekonvoluci (transponovanou konvoluci) s 128 filtry, jádrem velikosti 4x4, krokem 2 a zachováním původního tvaru (padding='same').

Následuje opět vrstva LeakyReLU s parametrem alpha=0.2 a vrstva BatchNormalization s parametrem momentum=0.8, které slouží k normalizaci a stabilizaci výstupů z předchozí vrstvy.

Následně jsem přidal Dropout vrstvu s hodnotou 0.4, která slouží, jak už bylo zmíněno ke zmenšení případného přeučení (overfitting) tím, že náhodně deaktivuje určité neurony během trénování.

```
--------------------------------------------------------
model.add(Conv2DTranspose(128, kernel_size=4, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))
model.add(Dropout(0.4))
--------------------------------------------------------
```

Tento vzor se opakuje pro další tři bloky vrstev. Postupně snižují rozměr a zvyšují počet filtrů a u poslední vrstvy vynechávám dropout:

```
--------------------------------------------------------
model.add(Conv2DTranspose(64, kernel_size=4, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))
model.add(Dropout(0.4))
model.add(Conv2DTranspose(32, kernel_size=4, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))
model.add(Dropout(0.4))
model.add(Conv2DTranspose(16, kernel_size=4, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))


--------------------------------------------------------
```

V poslední části jsem přidal vrstvu, která používá 3 filtry, což odpovídá výstupnímu kanálu RGB obrazu. Snižuju krok na 1 a zvětšuju jádro na 5x5. Zároveň používám aktivaci tanh, která umožňuje generování hodnot v rozmezí [-1, 1]. Stručný výpis modelu mi zajišťuje model.summary().

```
--------------------------------------------------------
model.add(Conv2DTranspose(3, kernel_size=5, strides=1,
padding='same', activation='tanh'))
model.summary()
--------------------------------------------------------
```

Za účelem lepší vizualizace příkladám kod pro generátor:

```
1
2  def build_generator(self, noise_type):
3          model = Sequential()
4
5          # Generator layers
6          model.add(Dense(128 * 16 * 16))
7          model.add(LeakyReLU(alpha=0.2))
8          model.add(Reshape((16, 16, 128)))
9
10         model.add(Conv2DTranspose(128, kernel_size=4, strides=2, padding='same'))
11         model.add(LeakyReLU(alpha=0.2))
12         model.add(BatchNormalization(momentum=0.8))
13         model.add(Dropout(0.4))
14
15         model.add(Conv2DTranspose(64, kernel_size=4, strides=2, padding='same'))
16         model.add(LeakyReLU(alpha=0.2))
17         model.add(BatchNormalization(momentum=0.8))
18         model.add(Dropout(0.4))
19         model.add(Conv2DTranspose(32, kernel_size=4, strides=2, padding='same'))
20         model.add(LeakyReLU(alpha=0.2))
21         model.add(BatchNormalization(momentum=0.8))
22         model.add(Dropout(0.4))
23         model.add(Conv2DTranspose(16, kernel_size=4, strides=2, padding='same'))
24         model.add(LeakyReLU(alpha=0.2))
25         model.add(BatchNormalization(momentum=0.8))
26
27         model.add(Conv2DTranspose(3, kernel_size=5, strides=1, padding='same',
       activation='tanh'))
```

```
28          noise = Input(shape=(self.LATENT_DIM,))
29          img = model(noise)
30          model.summary()
31
32
33          return Model(noise, img)
```

5.8 Implementace diskriminátoru

V následující části ukazuju použitý kód, pro diskriminátor, který zaručil generování výstupu, které uvádím v závěru práce.

Tak jako u generátoru i zde prvně vytvářím instanci modelu Sequential, která slouží k postupnému přidávání vrstev do modelu.

```
------------------------------
def build_discriminator(self):
    model = Sequential()
------------------------------
```

Dále přidávám vrstvu Conv2D, která je konvoluční vrstvou s 32 filtry, jádrem velikosti 3x3, krokem 2 a se zachováním původního tvaru (padding='same'). Následuje vrstva LeakyReLU, která aplikuje leaky rectified linear unit aktivaci s parametrem alpha=0.2. Následně jsem přidal Dropout vrstvu s hodnotou 0.4.

```
-----------------------------------------------
model.add(Conv2D(32, kernel_size=3, strides=2,
    input_shape=(256, 256, 3), padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.4))
-----------------------------------------------
```

Tento vzor se opakuje pro další tři bloky vrstev, přičemž počet filtrů se postupně zvyšuje na 64, 128 a 256.

```
----------------------------------------------
model.add(Conv2D(64, kernel_size=3, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.4))

model.add(Conv2D(128, kernel_size=3, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.4))

model.add(Conv2D(256, kernel_size=3, strides=2,
    padding='same'))
model.add(LeakyReLU(alpha=0.2))


-----------------------------------------------
```

Následně jsem přidal vrstvu Flatten, která slouží k převedení výstupu z předchozích vrstev do jednorozměrného vektoru, který lze předat do plně propojené vrstvy.

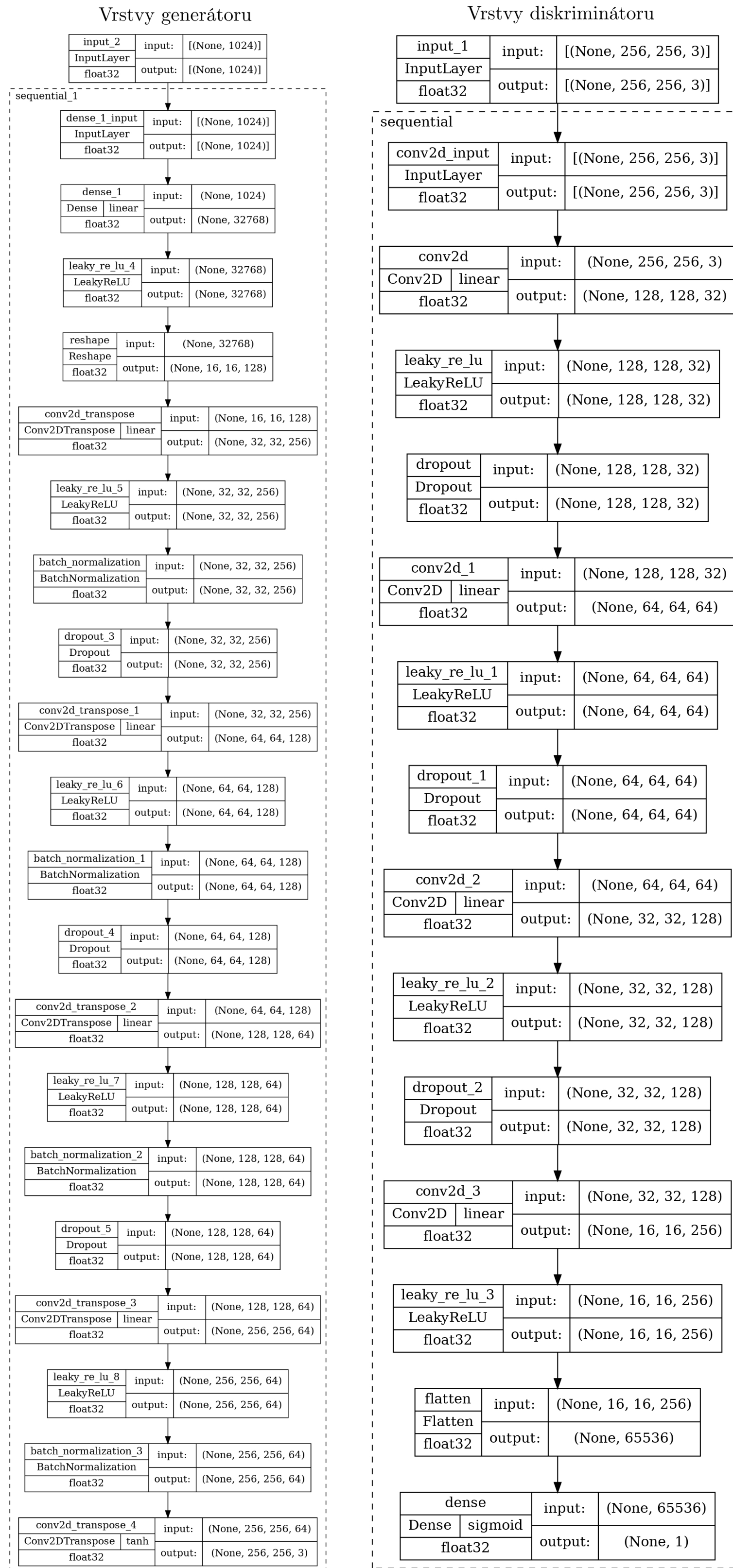
Nakonec jsem přidal plně propojenou vrstvu Dense s jedním neuronem a aktivací sigmoid. Tato vrstva slouží k binární klasifikaci (rozhodnutí, zda je obrázek reálný nebo generovaný).Stručný výpis modelu mi zajišťuje model.summary().

```
----------------------------------------
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.summary()
----------------------------------------
```

I zde za účelem lepší vizualizace příkladám kod pro diskriminátor:

```
1
2  def build_discriminator(self):
3          model = Sequential()
4
5          # Discriminator layers
6          model.add(Conv2D(32, kernel_size=3, strides=2, input_shape=(256, 256, 3),
       padding='same'))
7          model.add(LeakyReLU(alpha=0.2))
8          model.add(Dropout(0.4))  # Dropout layer
9          model.add(Conv2D(64, kernel_size=3, strides=2, padding='same'))
10
11         model.add(LeakyReLU(alpha=0.2))
12         model.add(Dropout(0.4))  # Dropout layer
13         model.add(Conv2D(128, kernel_size=3, strides=2, padding='same'))
14
15         model.add(LeakyReLU(alpha=0.2))
16         model.add(Dropout(0.4))  # Dropout layer
17         model.add(Conv2D(256, kernel_size=3, strides=2, padding='same'))
18
19         model.add(LeakyReLU(alpha=0.2))
20         model.add(Flatten())
21         model.add(Dense(1, activation='sigmoid'))
22
23         model.summary()
24
25         img = Input(shape=(256, 256, 3))
26         validity = model(img)
27
28         return Model(img, validity)
```

Pro lepší náhled na architekturu mého modelu a jednotlivých vrstev jak generátoru, tak diskriminátoru 5.7 jsem využil funkci plot_model. To mi umožnilo graficky zobrazit celkovou strukturu modelu.



Obrázek 5.7: Vygenerovaný diagram struktury generátoru a diskriminátoru skrz funkci plot_model

Kapitola 6

Vizualizace výsledků

6.1 Uživatelské rozhraní

Má bakalářská práce se zaměřuje nejen na generaci, ale i vizualizaci. Pro tyto účely jsem naprogramoval program s uživatelským rozhraním (UI), který slouží k generování šumu pomocí GAN. Tento program umožňuje uživatelům interaktivně ovládat proces generování šumu a vizualizovat výsledky.

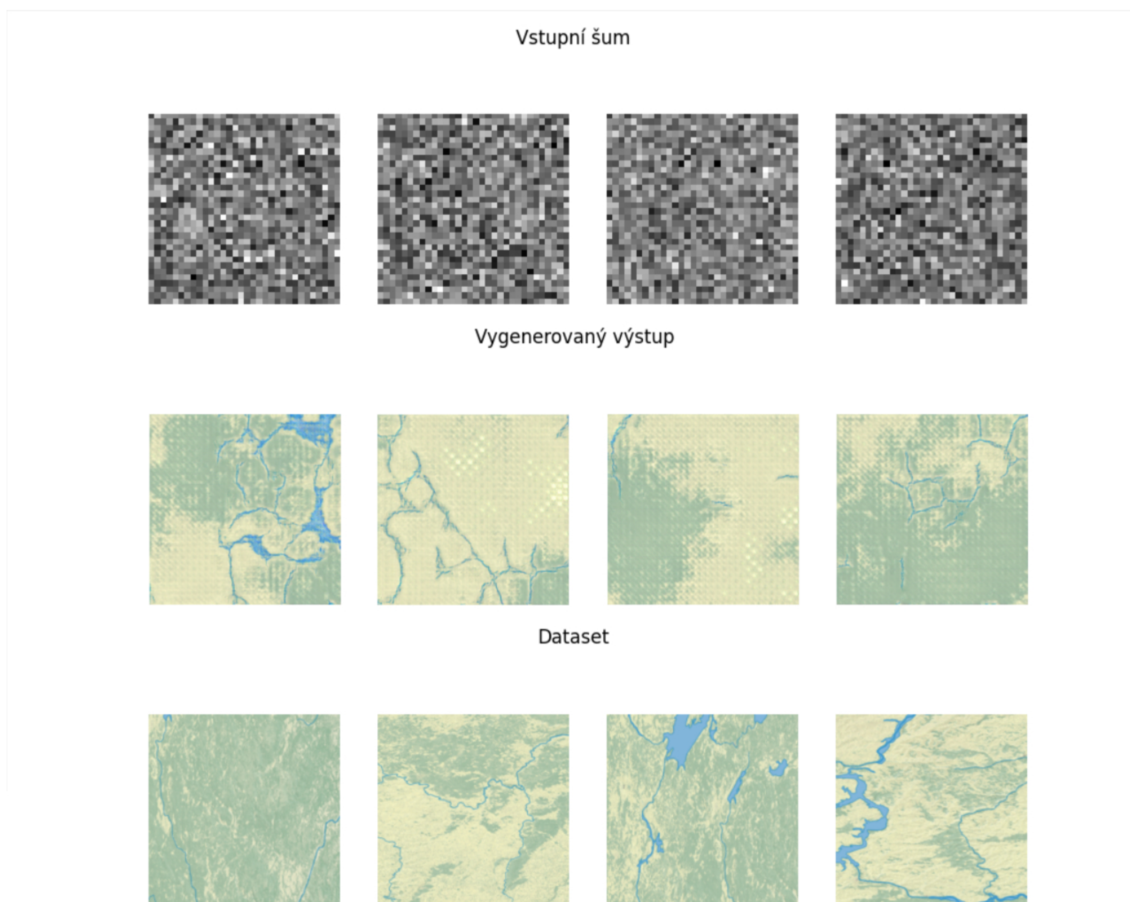
UI programu byl navržen pomocí modulů knihovny Tkinter. Okno programu má rozměry 1000x600 pixelů. V horní části UI jsou umístěny prvky pro výběr typu šumu a vstupní pole pro zadání počtu epoch, které určují druh zvolených vah. Uživatel má možnost vybrat typ šumu z nabídky, která obsahuje možnosti "random", "Perlin" a "Simplex". Dále jsou v horním panelu dvě tlačítka, "Generovat" a "Smazat obrázky". Po kliknutí na tlačítko "Generovat" program začne generovat obrázky na základě zvolených parametrů typu šumu a počtu epoch a tlačítko "Smazat" umožní smazání všech vygenerovaných obrázků pro aktuálně zvolený typ šumu v dané složce. Na přiloženém obrázku 6.1 zobrazuji popisovaný horní panel.

Obrázek 6.1: Ukázka horního panelu

Samotné generování probíhá pomocí předtrénované GAN neuronové sítě, která je implementována v modulu "main". Program načte váhy neuronové sítě z příslušné složky podle zvoleného typu šumu. Následně provede generování na základě zadaného počtu epoch. Vygenerované obrázky jsou uloženy do složky "data/output"a jsou zobrazeny na UI. Při výstupu vidím v UI nejen vygenerovaný obrázek, ale i počáteční šum jako takový a náhodně zvolené obrázky z datasetu, což mi umožňuje vizuálně posoudit výsledky generování. V následujícím obrázku 6.2 ukazuji obsah okna pro "random" tedy náhodný šum při 15000 epochách.

Program je nastaven na počáteční hodnoty pro typ šumu random (náhodný) a počtu

epoch na 15000. Uživatel má možnost tyto hodnoty dále upravit podle svých preferencí.



Obrázek 6.2: Ukázka obsahu okna pro náhodný šum při 15000 epochách

6.2 Výsledky generování

V této kapitole se zaměřím na prezentaci výsledků, které jsem dosáhl v rámci mého výzkumu. Budu popisovat různé způsoby a metody, které jsem zvolil. Před následujícími podkapitolami je potřeba zmínit, že v mé práci jsem pracoval s náhodným, Perlinovým a Simplexovým šumem. Jednotlivé šumy jsem nageneroval s vlastními váhami, které jsem následně různě kombinoval pro porovnání.

6.2.1 Náhodný šum

Náhodný šum generoval překvapivě rychle působivé mapy. Proces učení probíhal po dobu 123000 epoch, při intervalu 1000 obrázků jako kontrolních vzorků. Nicméně už při 80000 epochách se tvořily uhlazenější obrázky. Výstupní velikost obrázku

byla 256x256 pixelů v mřížce 3x4 pro lepší porovnání. Bylo vyzkoušeno více metod a různých změn parametrů. Úpravy, které nejvíce ovlivnily generování sítě a její výstupy z mých pozorování, byly větší počet volání samotného generátoru, přidání dalších vrstev a zmenšení hodnoty dropout vrstvy pro obě neuronové sítě.

Provedl jsem sérii testů, ktere nyní budu prezentovat. Začněme obrázkem 6.3, kde jsem trénoval, ale i generoval síť na náhodném šumu. Na obrázku lze vidět náhodný generovaný šum a následně vygenerované obrázky po 123000 epochách při nejlépe natrénované a nagenerované síti z mých pokusů. Po prozkoumání obrázku lze vidět, že řeky se tvořily spojitě a hladce, dokonce v určitých částech se podařilo nagenerovat i menší jezírka. Při okolním pozorování lze vidět určité artefakty v podobě menších nevýrazných čtverečků. Tyto čtverečky vznikaly na začátku mých pokusů převážně nedostatečnou velikostí mého datasetu. Poté se tyto artefakty objevovaly při nedostatečném počtu epoch na trénování či malou velikostí dávky. V tomto případě je možné, že generované obrázky dosáhly určité komplexity, kdy bylo pro síť obtížné provést perfektní rekonstrukci obrázku a byla by potřeba zvolit modifikovanou síť.

Jako další pokus, který ukazuji na obrázku 6.4 jsem využil již natrénované váhy z náhodného šumu a zkusil jsem vygenerovat mapy pomocí Perlinova šumu. Přikládám obrázek mého snažení.

U posledního experimentu, který lze vidět na obrázku 6.5 jsem se pokusil natrénované váhy aplikovat pro generování simplexového šumu.

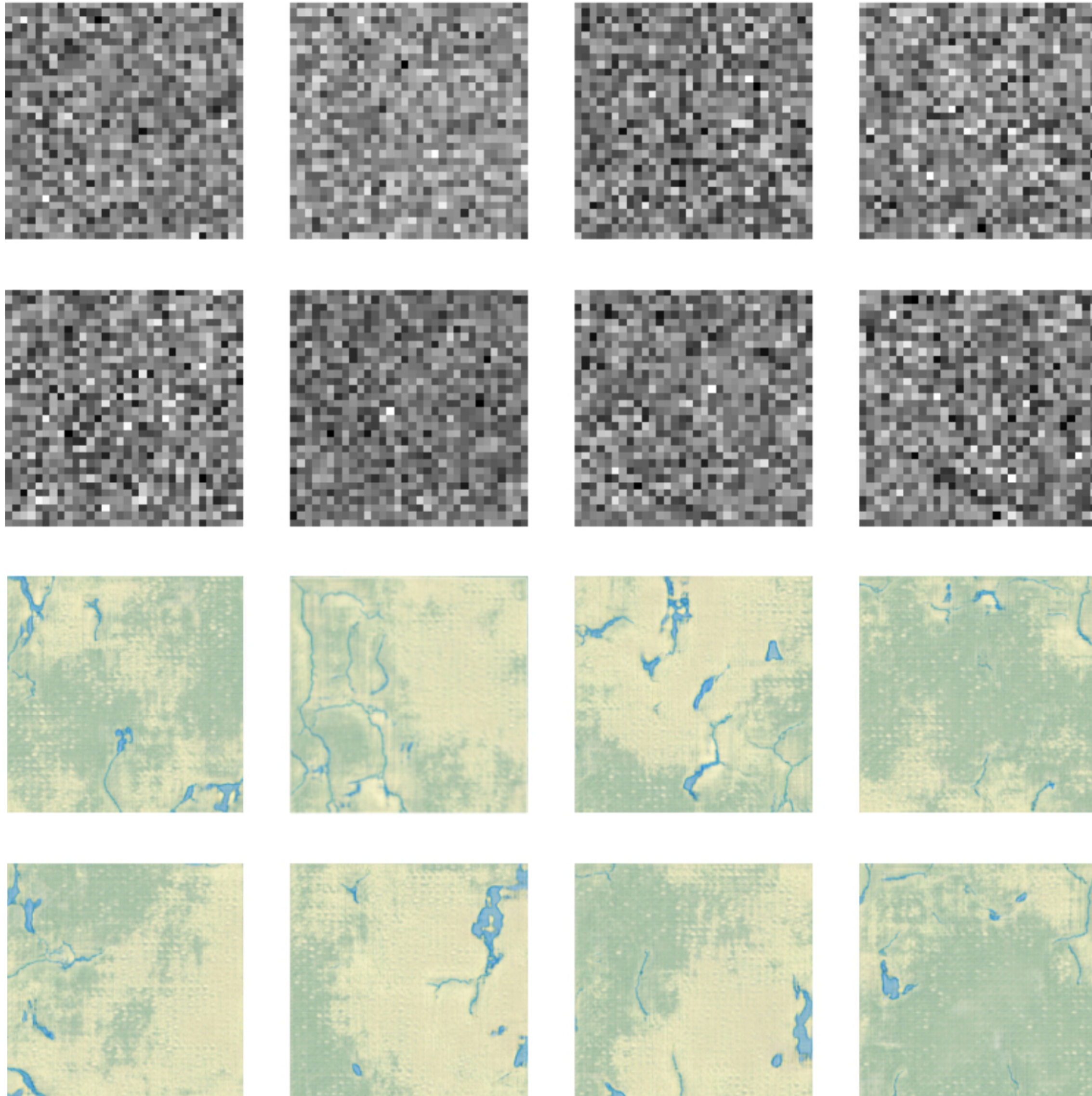
Můžeme pozorovat, že při obou šumech měl generátor problém vygenerovat vizuálně optimální mapy.

6.2.2 Perlinův šum

Pro další část jsem využil variantu, která spočívala v podobě využití Perlinova šumu. Perlinův šum bohužel negeneroval idální výsledky. Proces učení trval 66000 epoch při intervalu 1000 obrázků jako kontrolních vzorků, kdy se začli vzorky cyklit po 55000. Výstupní velikost obrázku je 256x256 pixelů v mřížce 3x4 pro lepší porovnání. Vyzkoušel jsem plno variant počínaje předěláním jednotlivých vrstev sítě pro generátor i diskriminátor, změnu počtu volání generátoru až po úpravu dávkování nebo algoritmu optimalizace Adam. Oktávy, perisistence a lucenarita byly nastavený na 8, 0.5 a 1.5. Na následujícím obrázku 6.6 prezentuji aplikaci vygenerovaných vah a následnou generaci za použití Perlinova šumu. Lze vidět tvoření modrých ploch, ale bez spojení.

Jako další kombinaci, která je ukázána na obrázku 6.7, je generování výstupu při zvolení Simplexového šumu na váze vygenerované Perlinovém šumu. Při této kombinace lze vidět, že na generovaném obrázku se začínají více propojovat jednotlivé části modrých ploch oproti původní kombinaci.

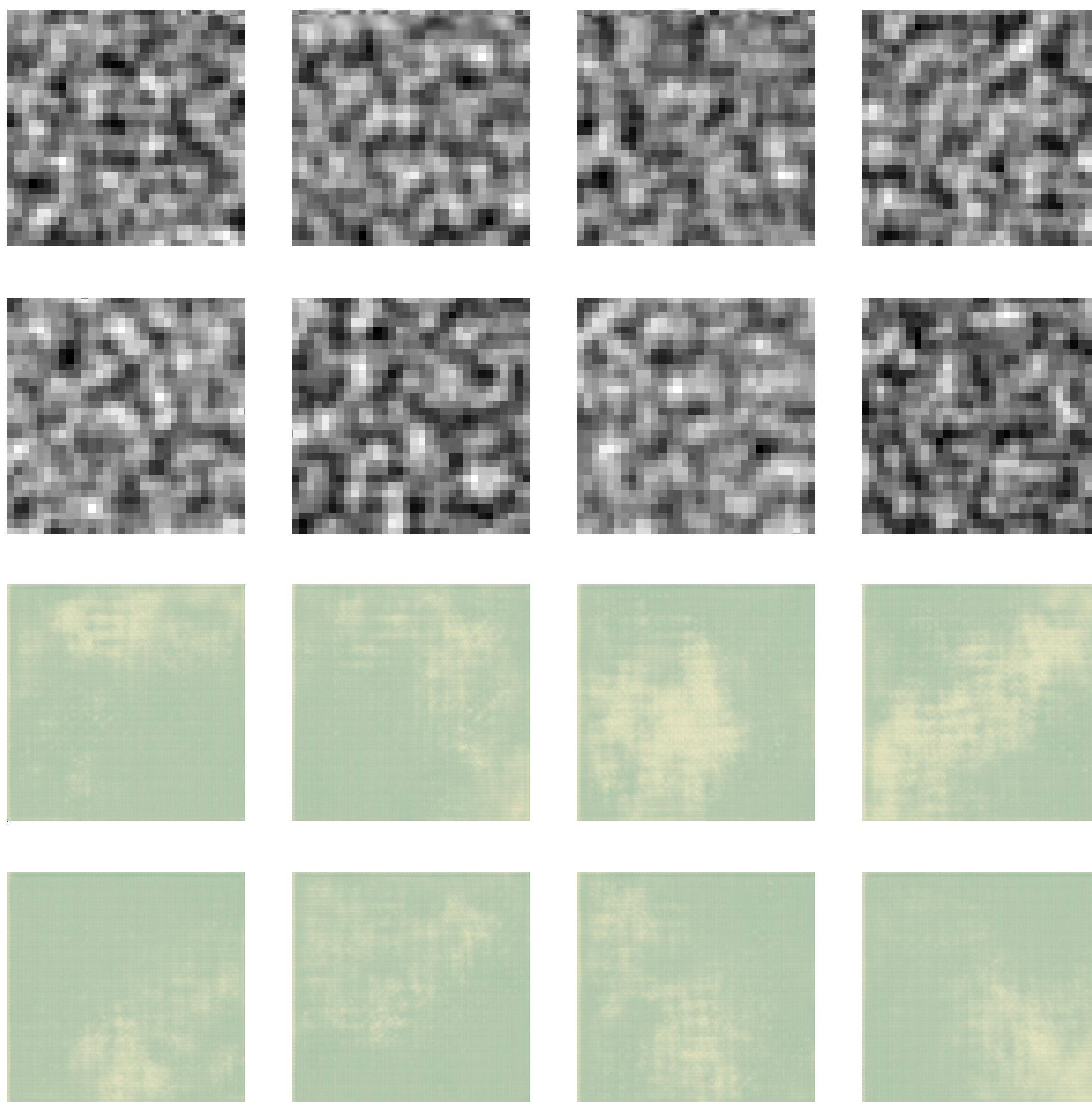
Poslední kombinací, kterou jsem zvolil, je generování výstupu při náhodném šumu na naučených váze, kterou představuji na obrázku 6.8. Jak jde vidět, obrázek se velmi barevně zdeformoval a je z těchto kombinací nejvzdálenější ke skutečnému datasetu. Zaroveň se zde velice často projevovali artefakty v podobě menších čtverečků.



Obrázek 6.3: Ukázka natrénované a negenerované sítě pomocí náhodného šumu

Na obrázkách ukazuji další alternativu. V této sítí byly u generátoru přidány další konvoluční vrstvy, zvýšení velikosti dense vrstvy a snížení v posledních vrstvách hodnota kernelu. Zminuju ji takhle vedle proto, protože tato alternativa bohužel nezlepšila generování vodních ploch, nicméně obrázek působí více uhlazenější. Na obrázku 6.9 ukazuji všechny tři varianty tzn. generace dle vah podle Perlinového, náhodného a Simplexového šumu. Síť byla naučena na 67000 epochách. Oktávy, perisistence a lucenarita byly nastavený na 8, 0.5 a 1.5. Proces učení trval 67000 epoch při intervalu 1000 obrázků jako kontrolních vzorků, kdy se začli vzorky cyklit po 30000 epochách.

Náhodný šum 2. varianta



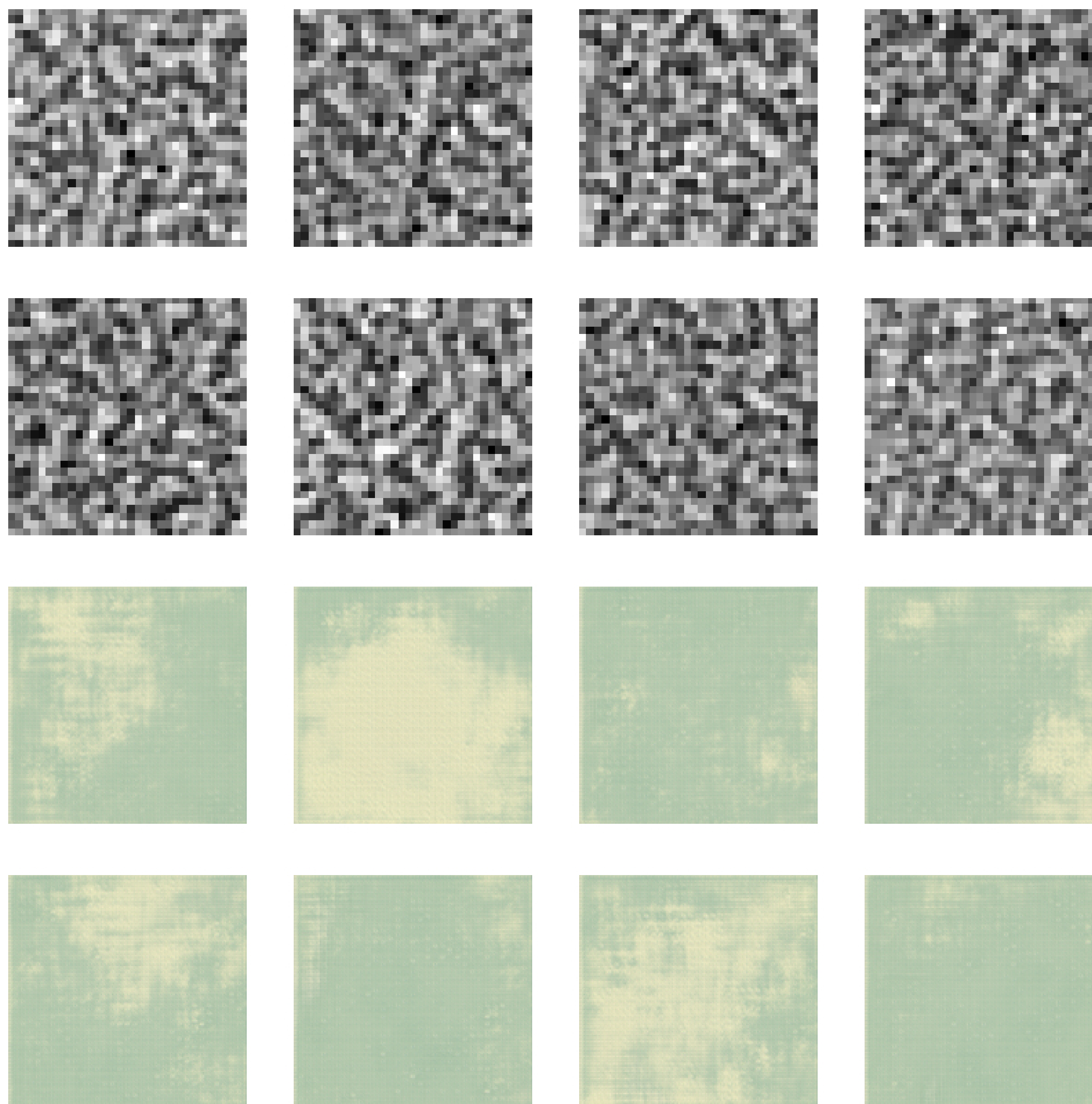
Obrázek 6.4: Ukázka natrénované sítě na náhodném šumu a negenerované sítě na Perlinovém šumu

6.2.3 Simplexový šum

Nyní jsem udělal poslední variantu, která spočívala v podobě využití vah z naučené sítě na Simplovém šumu.

Simplexový šum bohužel negeneroval dobré vzorky. Proces učení trval více jak 60000 epoch při intervalu 1000 obrázků jako kontrolních vzorků, kdy po 50000 síť začla cyklit výstupní obrázky. Výstupní velikost obrázku byla 256x256 pixelů v mřížce 3x4 pro lepší porovnání. Pozoroval jsem, že při některých epochách se začli tvořit jemné náznaky modrých oblastí, avšak ty zmizli do dalšího kontrolního vzorku. Tak jako u Perlinového šumu jsem vyzkoušel plno variant ať začínaje předěláním

Náhodný šum 3. varianta



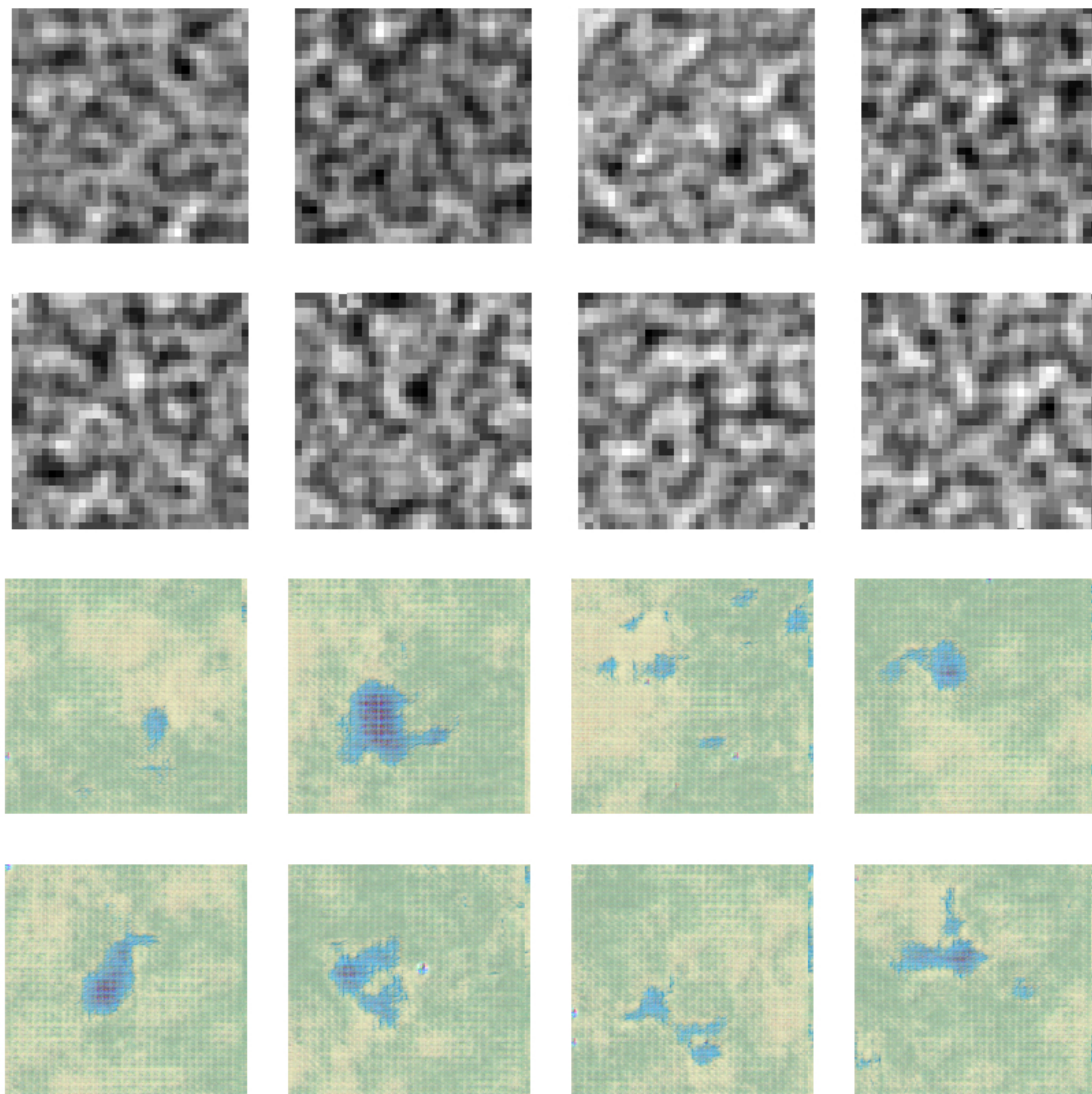
Obrázek 6.5: Ukázka natrénované sítě na náhodném šumu a negenerované sítě pomocí Simplexového šumu

jednotlivých vrstev sítě pro generátor i diskriminátor, změnu počtu volání generátoru až po úpravu dávkování nebo algoritmu optimalizace Adam. Spolu se změnami v paramatrech daný šumů byly mizivé. Zvětšením oktáv a lucenarity bylo dosaženo lepších výsledků, než při jiných nastaveních. Na následujícím obrázku 6.10 ukazuji použití vygenerovaných vah a následné generování při využití Simplexového šumu.

Jako další experiment, který zde ukazuji je využití natrénovaných vah ze Simplexového šumu, které jsem použil na generování map pomocí Perlinova šumu. Přikládám obrázek 6.11 mého snažení.

Poslední alternativou, která je prezentována na obrázku 6.12, jsem využil již natréno-

Perlinův šum 1. varianta

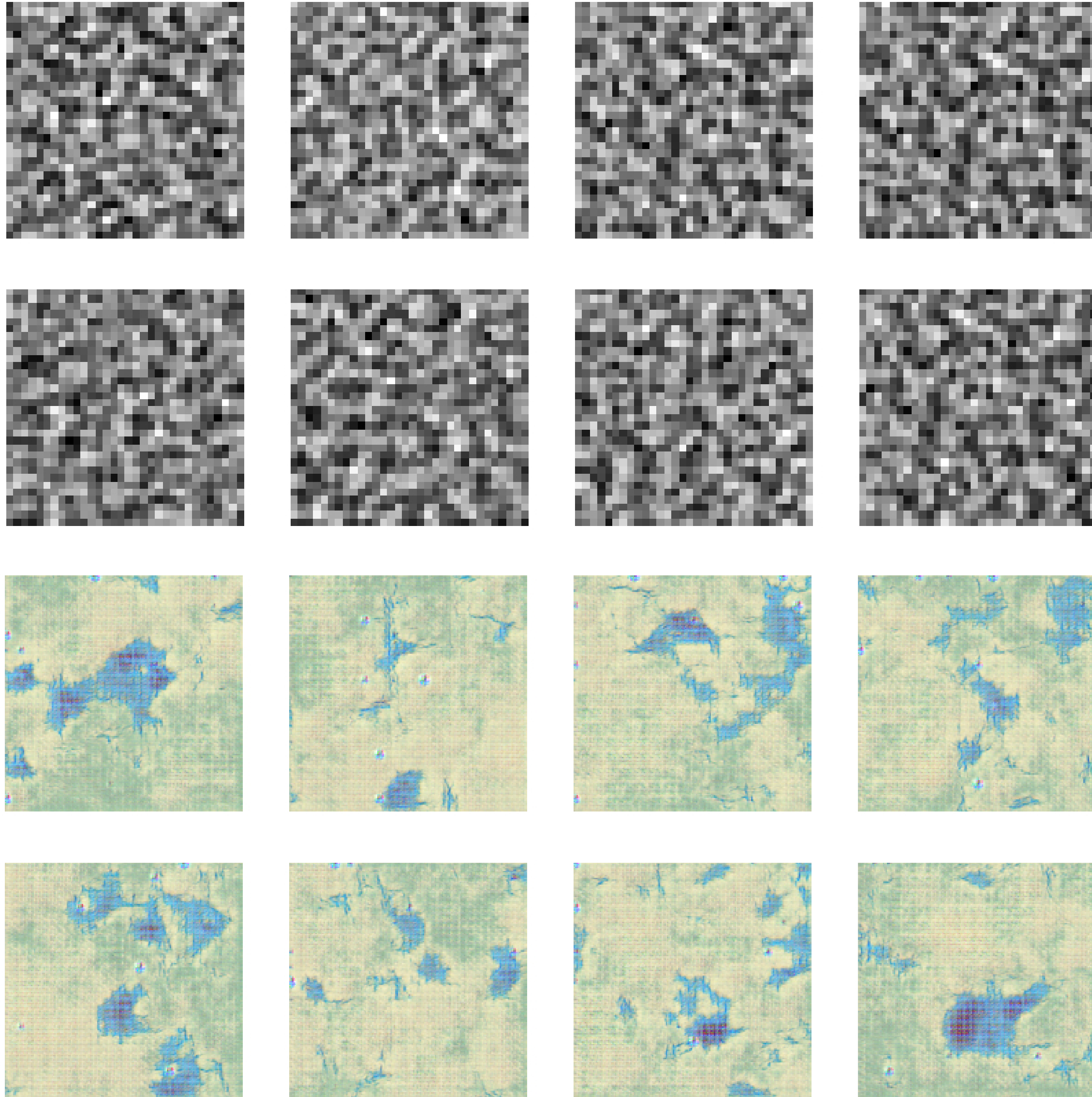


Obrázek 6.6: Ukázka natrénované a negenerované sítě na Perlinovém šumu

vané váhy ze Simplexového šumu a pokusil se vygenerovat mapy pomocí náhodného šumu.

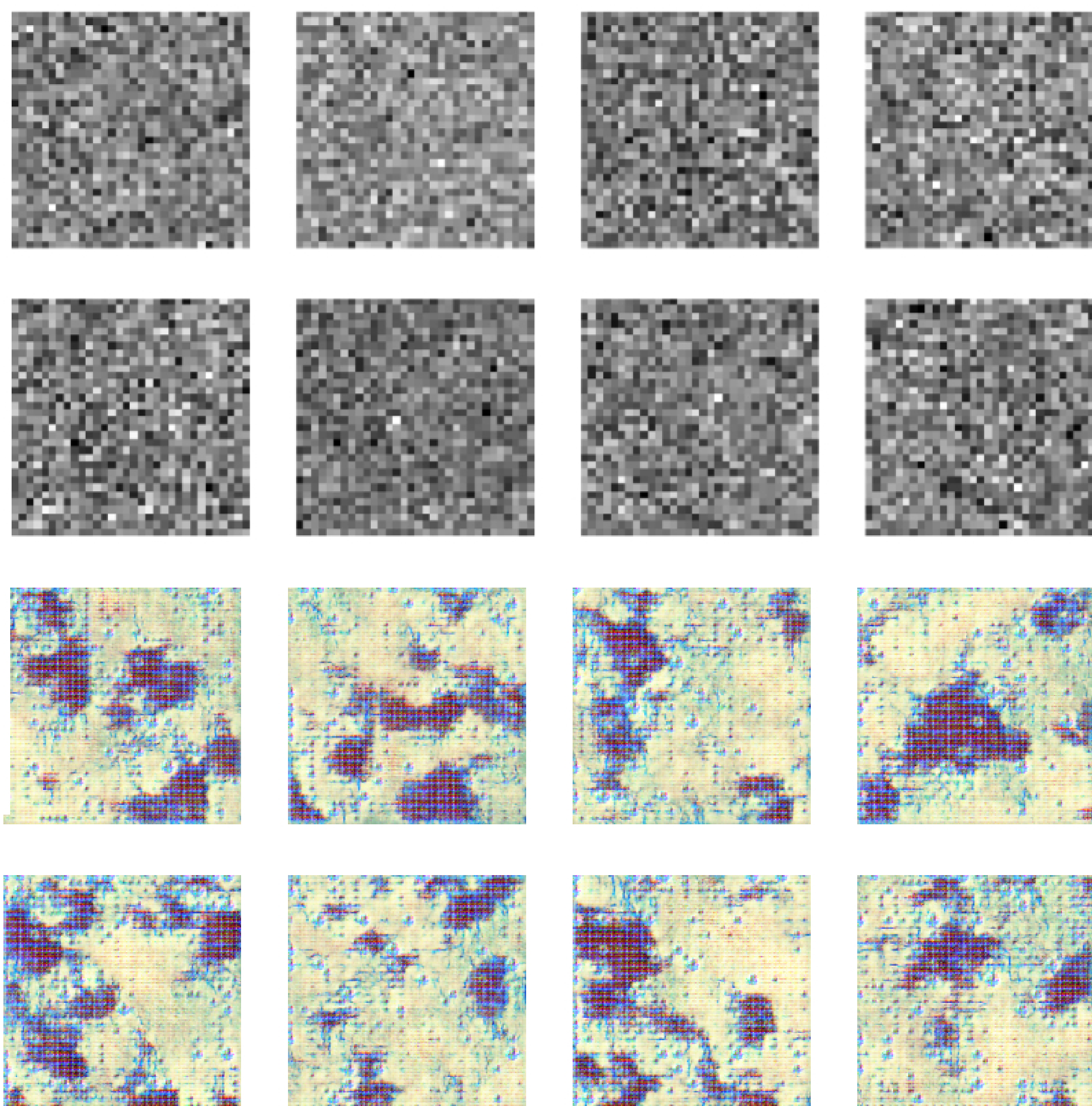
Na obrázku lze vidět, že se začli na mapě objevovat daleko více modrých spojitých ploch. Okolní prostředí avšak není zcela dobře vygenerované a vznikají zde znatelně viditelné artefakty v podobě čtverečků či bílých koleček.

Perlinův šum 2. varianta



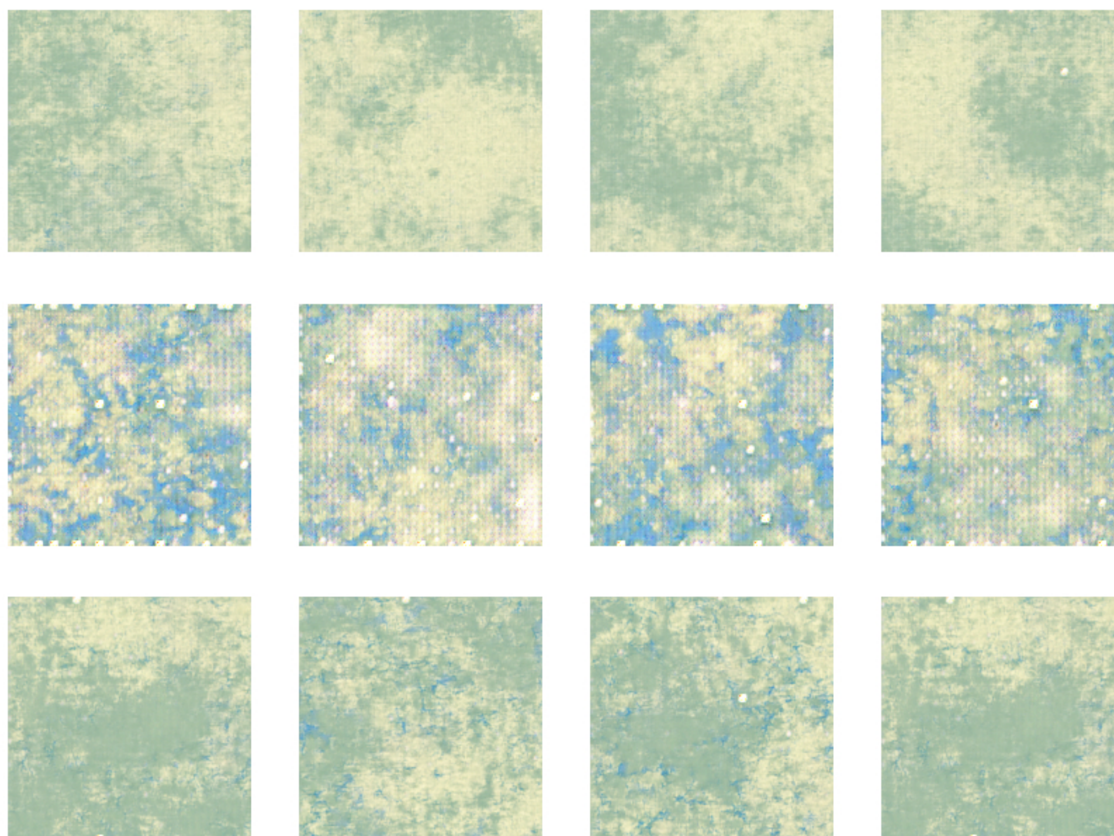
Obrázek 6.7: Ukázka natrénované sítě na Perlinovém šumu a negenerované sítě na Simplexovém šumu

Perlinův šum 3. varianta



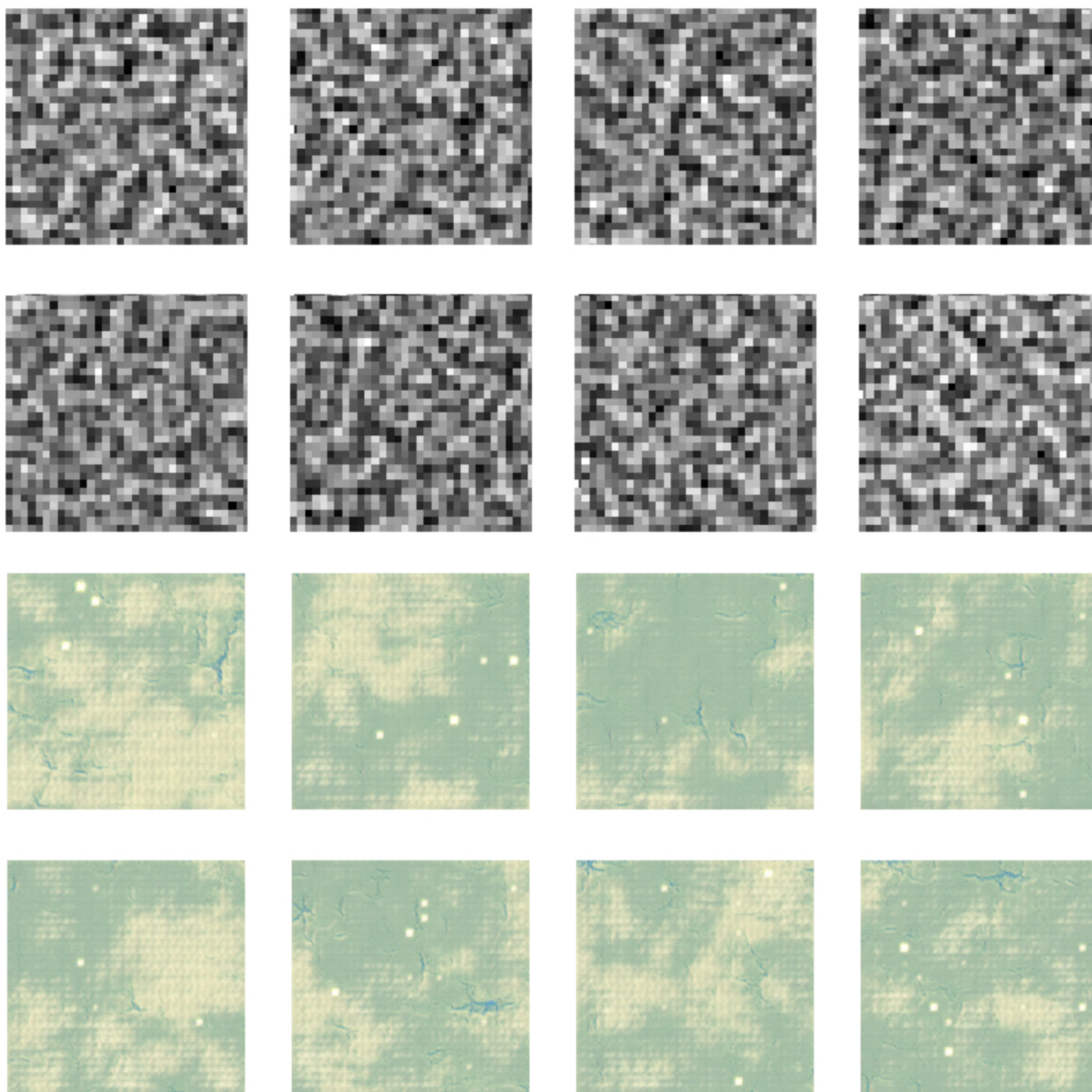
Obrázek 6.8: Ukázka natrénované sítě na Perlinovém šumu a negenerované sítě na náhodném šumu

Příklad jiné architektury sítě - Perlinův šum



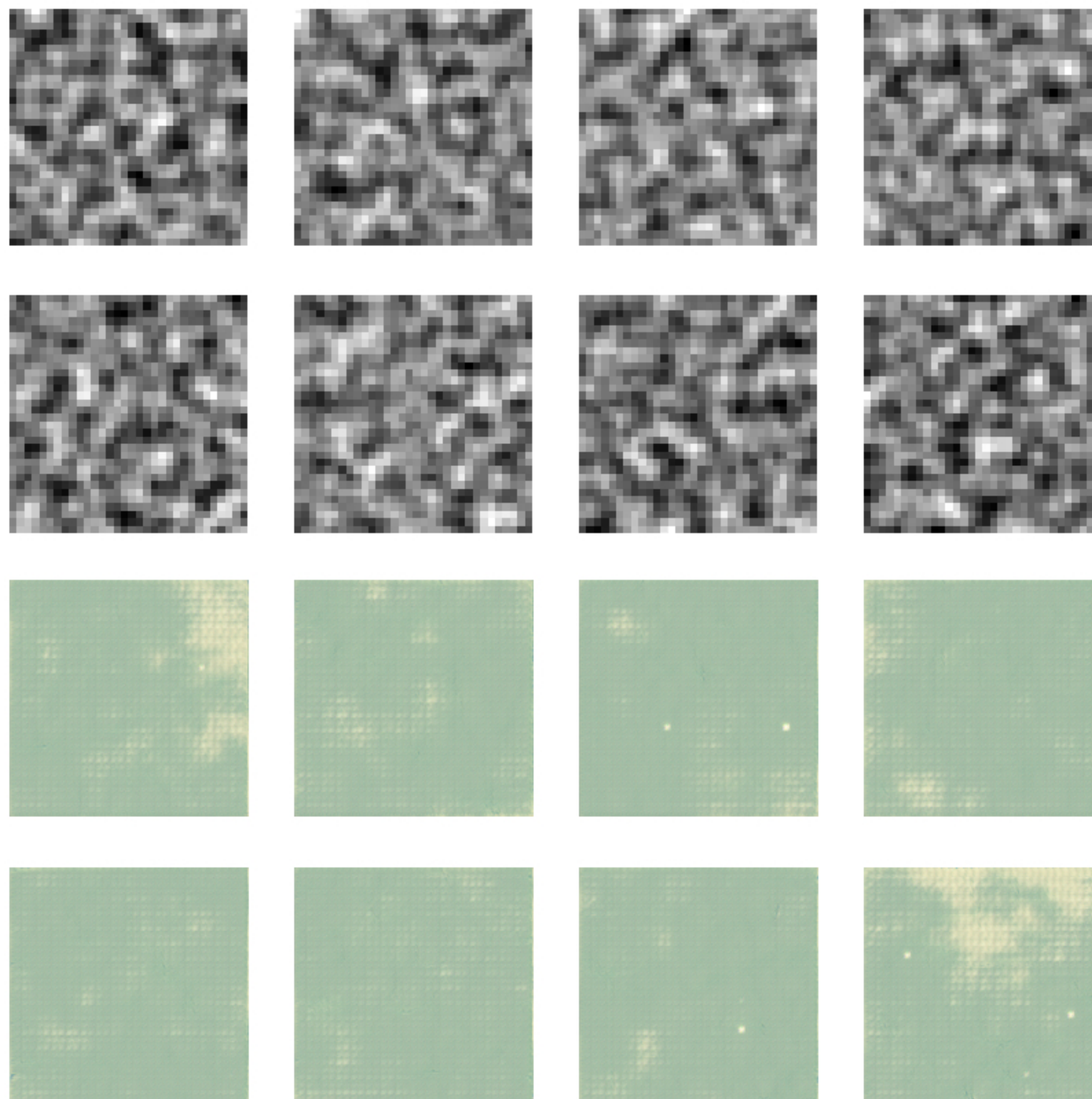
Obrázek 6.9: Ukázka natrénované sítě na Perlinovém šumu a nagenerované sítě na Perlinovém, náhodném a následně Simplexovém šumu

Simplexový šum 1. varianta



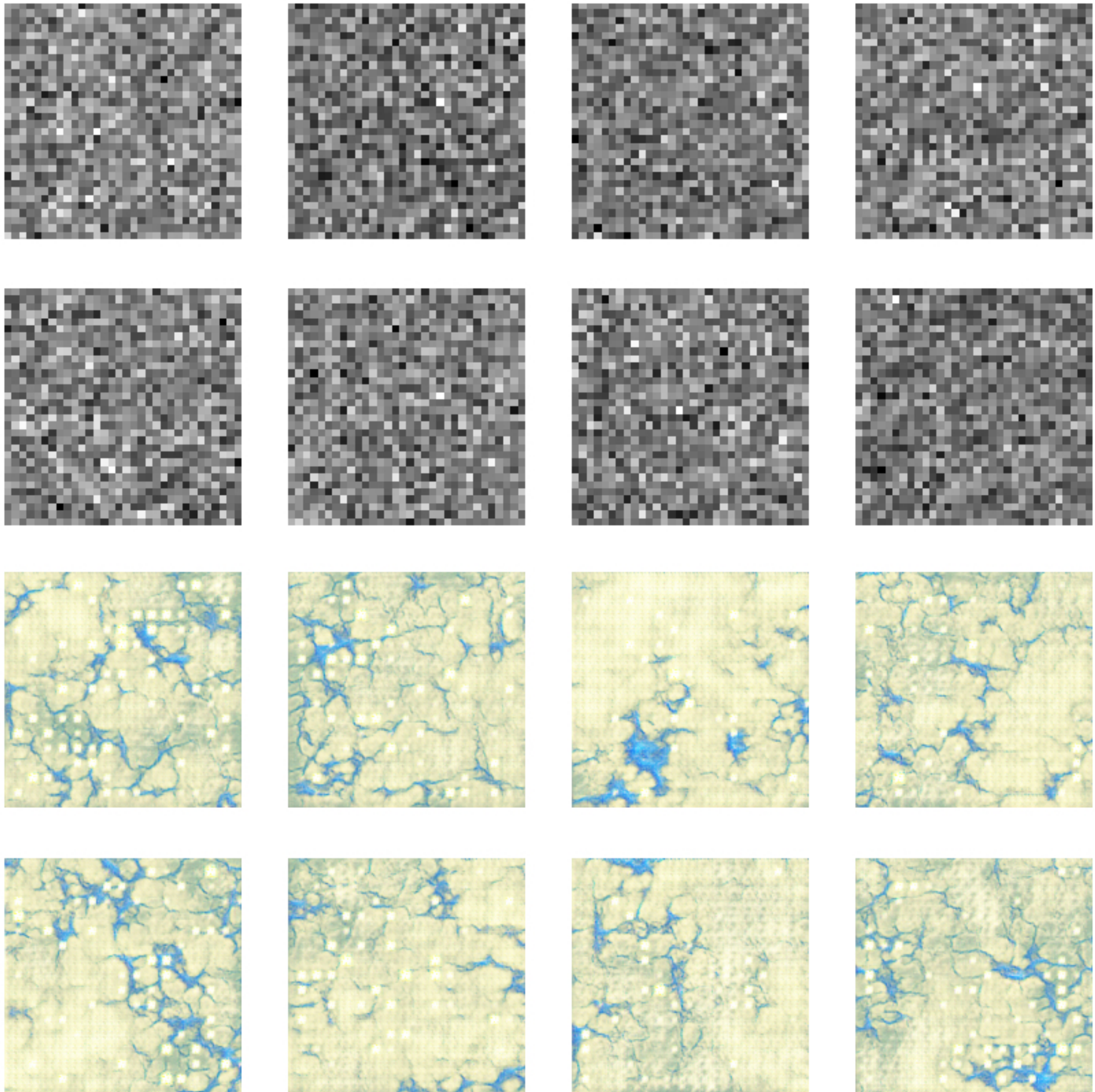
Obrázek 6.10: Ukázka natrénované a negenerované sítě na Simplexovém šumu

Simplexový šum 2. varianta



Obrázek 6.11:

Simplexový šum 3. varianta



Obrázek 6.12: Ukázka natrénované sítě na Simplexovém šumu a negenerované sítě na náhodném šumu

Závěr

Na základě mého hodnocení a experimentů s generováním šumu pomocí GAN modelu jsem dospěl k několika závěrům. Zkoumal jsem tři různé typy šumu: náhodný, Perlinův a Simplexový, a snažil se najít optimální konfiguraci sítě pro každý z těchto typů. Z mého pohledu a na základě výsledků, které jsem dosáhl, lze říct, že nejlepší výsledky jsem dosáhl s náhodným šumem. Tyto výsledky velmi odpovídaly vstupnímu datasetu. Nicméně je důležité zdůraznit, že proces optimalizace sítě je složitý a nemusely se zvážit veškeré proměnné, které by mohly jiným způsobem ovlivnit výsledek.

Perlinův a Simplexový šum se chovaly lépe po vyšším počtu epoch, přičemž v mé práci Perlinův šum vykazoval lepší výsledky než Simplexový. Simplexový šum se zdál být náročnější na generování optimálních výsledků, s výraznějšími artefakty v podobě malých čtverečků a obtížnějším dosažením požadovaných modrých ploch.

Experimentace a ladění generativního modelu si vyžádaly mnoho opakovaných pokusů a testování různých přístupů. Každou z metod jsem zkoušel do té doby, než se výsledky začaly zhoršovat nebo se výstupní obrázky zacyklily.

Pro další výzkum a zdokonalení by bylo vhodné zvážit použití modifikovaných GAN modelů, jako jsou například WGAN nebo DCGAN. Tyto modely by mohly poskytnout detailnější výsledky i při menším počtu epoch. Rovněž by bylo možné provést další úpravy a experimenty s architekturou sítě, například přidání nebo odebrání vrstev, úpravu počtu neuronů nebo změnu aktivačních funkcí. Celkově jsem přesvědčen, že pokračující výzkum a zdokonalení těchto metod by mohly vést k dosažení ještě lepších výsledků a širšímu využití generativních modelů v oblasti vizualizace a umělé inteligence.

Literatura

[1] RADFORD, Alec, Luke METZ a Soumith CHINTALA. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* [online]. USA, 2016 [cit. 2023-08-01]. Dostupné z: https://arxiv.org/abs/1511.06434

[2] *Neural Network* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Neural_network&oldid=1102194149

[3] *Generative adversarial network* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Generative_adversarial_network

[4] BAELDUNG, Eugen. *Epoch in Neural Networks* [online]. Rumunsko, 2023 [cit. 2023-08-01]. Dostupné z: https://www.baeldung.com/cs/author/baeldung

[5] BAHETI, Pragati. *Activation Functions in Neural Networks [12 Types & Use Cases]*. V7 Labs Blog, 2021. Dostupné také z: https://www.v7labs.com/blog/neural-networks-activation-functions

[6] SCHEIBENPFLUG, Andreas, Johannes KARDER, Susanne SCHALLER, Stefan WAGNER a Michael AFFENZELLER. *Evolutionary Procedural 2D Map Generation using Novelty Search.* Rakousko, 2016.

[7] GOODFELLOW, Ian J., Jean POUGET-ABADIE, Mehdi MIRZA, Bing XU, David WARDE-FARLEY, Sherjil OZAIR, Aaron COURVILLE a Yoshua BENGIO. *Generative Adversarial Nets* [online]. 2014 [cit. 2023-08-01]. Dostupné z: https://arxiv.org/pdf/1406.2661.pdf

[8] TISOVČÍK, Rastislav. *Generation and Visualization of Terrain in Virtual Environment.* Brno, 2012. Dostupné také z: https://is.muni.cz/th/hb7zk/Thesis.pdf. Bakalářské práce. Masarykova Univerzita- Fakulta Informatiky. Vedoucí práce Mgr. Jiří Chmelík.

[9] VEEN, Fjodor van. *THE NEURAL NETWORK ZOO* [online]. USA: The Asimov Institute, 2016 [cit. 2023-08-01]. Dostupné z: https://www.asimovinstitute.org/neural-network- zoo/

[10] BEALE, -Russell a Tom JACKSON. *Neural Computing - An Introduction.* 1. USA: Institute of Physics Publishing, 1990. ISBN 0852742622.

[11] BIAGIOLI, Adrian. *Understanding perlin noise* [online]. adrians soapbox, 2014 [cit. 2023-08-01]. Dostupné z: https://adrianb.io/2014/08/09/perlinnoise.html

[12] KVASNIČKA, Vladimír, Ľubica BEŇUŠKOVÁ, Jiří POSPÍCHAL, Igor FARKAŠ, Peter TIŇO a Andrej KRÁĽ. *Úvod do teórie neurónových sietí*. Bratislava: IRIS, 1997. Dostupné také z: http://www2.fiit.stuba.sk/ kvasnicka-/Free%20books/Uvod%20do%20teorie%20neuronovych%20sieti_all.pdf

[13] HERTZ, John A., Anders FLISBERG a . *Introduction To The Theory Of Neural Computation.* Physics Today. 1991. Dostupné také z: https://www.researchgate.net/publication/200033871_Introduction_To_The_Theory_Of_Neural_Computation

[14] HOPFIELD, John J. *Neural networks and physical systems with emergent collective computational abilities.* USA: Proc Natl Acad Sci, 1982. Dostupné také z: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC346238/

[15] *Artificial neural network* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Artificial_neural_network

[16] *15 Generative Adversarial Networks (GAN) Based Project Ideas* [online]. 2023 [cit. 2023-08-01]. Dostupné z: https://www.projectpro.io/article/generative-adversarial-networks-gan-based-projects-to-work-on/530

[17] *Convolutional neural network* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Convolutional_neural_network

[18] *Convolutional Neural Networks (CNNs / ConvNets)* [online]. USA, 2022 [cit. 2023-08-01]. Dostupné z: https://cs231n.github.io/convolutional-networks/

[19] KUKAL, Jaromír. *Úvod do neuronových sítí* [online]. Děčín: Automa - časopis pro automatizační techniku, 2005, (11) [cit. 2023-08-01]. Dostupné z: https://automa.cz/cz/casopis-clanky/uvod-do-neuronovych-siti-2005_01_30255_1330/

[20] BEVINS, Jason. *Libnoise* [online]. USA, 2009 [cit. 2023-08-01]. Dostupné z: https://libnoise.sourceforge.net/glossary/

[21] *Simplex Noise* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Simplex_noise

[22] MOUNT, Dave a Roger EASTMAN. *Procedural Generation: 2D Perlin Noise* [online]. 2018 [cit. 2023-08-01]. Dostupné z: https://www.cs.umd.edu/class/spring2018/cmsc425/Lects/lect13-2d-perlin.pdf

[23] SILVEIRA, Gabriel. *How I Developed My Own 2D Procedural World Generation.* [online]. Španělsko: Ateliware, 2023 [cit. 2023-08-01]. Dostupné z: https://ateliware.com/blog/how-i-developed-my-own-2d-procedural- world-generation

[24] *Perlin noise* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Perlin_noise

[25] ARCHER, Travis. *Procedurally Generating Terrain.* USA: Morningside College, 2011. Dostupné také z: https://micsymposium.org/mics_2011_proceedings/mics2011_ submission_30.pdf

[26] CULURCIELLO, Eugenio. *Neural Network Architectures.* Towards Data Science, 2017. Dostupné také z: https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

[27] SCHEIBENPFLUG, Andreas, Johannes KARDER, Susanne SCHALLER, Stefan WAGNER a Michael AFFENZELLER. *Evolutionary Procedural 2D Map Generation using Novelty Search.* Rakousko, 2016. Dostupné také z: https://doi.org/10.1145/2908961.2909047

[28] LIU, Jialin, Sam SNODGRASS, Ahmed KHALIFA, Sebastian RISI, georgios N. YANNAKAKIS a Julius TOGELIUS. *Neural Computing and Applications: Deep learning for procedural content generation.* 2020, (33). Dostupné také z: https://link.springer.com/article/10.1007/s00521-020-05383-8

[29] CHENG, Wei-Chien, Wen-Chieh LIN a Yi-Jheng HUANG. *Smart Graphics: Controllable and Real-Time Reproducible Perlin Noise.* Taiwan, 2014. ISBN 9783319116495. Dostupné také z: https://doi.org/10.1007/978-3-319-11650-1_8

[30] *Voronoi diagram* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-02]. Dostupné z: https://en.wikipedia.org/wiki/Voronoi_diagram

[31] WORLEY, Steven. *A Cellular Texture Basis Function.* USA, 1996. Dostupné také z: https://doi.org/10.1145/237170.237267

[32] PETRŽELKOVÁ, Nela. *Face Image Editing in Latent Space of Generative Adversarial Networks.* Praha, 2021. Bakalářská práce. ČVUT. Vedoucí práce Jan Čech.

[33] *Biological neuron model* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Biological_neuron_model

[34] SANTELL, Jordan. *L-systems* [online]. USA, 2019 [cit. 2023-08-01]. Dostupné z: https://jsantell.com/l-systems/

[35] JOHNSON, Lawrence, Georgios YANNAKAKIS a Julian TOGELIUS. *Cellular automata for real-time generation.* USA: PCGames, 2010. ISBN 9781450300230. Dostupné také z: https://www.researchgate.net/publication/228919622_Cellular_automata _for_real-time_generation_of

[36] *Mandelbrot set* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Mandelbrot_set

[37] PABST, Joost Lawick van a Hans JENSE. *High Performance Computing for Computer Graphics and Visualisation: Dynamic Terrain Generation Based on Multifractal Techniques*. Anglie, 1996. ISBN 9783540760160. Dostupné také z: https://doi.org/10.1007/978-1-4471-1011-8_13

[38] BROWN, Adam. The Maths of Fractal Landscapes and Procedural Landscape Generation. *Fractal Landscapes* [online]. Velká Británie, 2020 [cit. 2023-08-01]. Dostupné z: https://www.fractal-landscapes.co.uk/maths.html

[39] *Value Noise* [online]. USA: Wikimedia Foundation, 2005 [cit. 2023-08-01]. Dostupné z: https://en.wikipedia.org/wiki/Value_noise

[40] FLICK, Jasper. *Value Noise* [online]. 2021 [cit. 2023-08-02]. Dostupné z: https://catlikecoding.com/unity/tutorials/pseudorandom-noise/value-noise/

[41] *Voronoi Noise* [online]. 2021 [cit. 2023-08-02]. Dostupné z: https://www.ronja-tutorials.com/post/028-voronoi-noise/#: :text=Another%20form%20of,easy%20to%20repeat.

Příloha A

Externí příloha

Externí příloha mé bakalářské práce je umístěna na adrese:

`https://github.com/mirek163/bakalarka`.

Obsah git repozitáře je následující:

```
main.py               Skript pro ruční generování/trénování sítě
UI.py                 Skript pro vizualizaci generování sítě v okně
augmentation.py       Skript pro vytvoření datasetu
png_to_gif.py         Skript pro převádění vygenerovaných obrázků
                       do gif souboru k zlepšení vizualizace změn
requirements.txt      Textový soubor s požadavky využitých knihoven
region.tif            Obrázek, ze kterého generuju můj dataset
```