VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV BIOMEDICÍNSKÉHO INŽENÝRSTVÍ

DEPARTMENT OF BIOMEDICAL ENGINEERING

NÁVRH GENERATIVNÍ KOMPETITIVNÍ NEURONOVÉ SÍTĚ PRO GENEROVÁNÍ UMĚLÝCH EKG ZÁZNAMŮ

GENERATIVE ADVERSARIAL NETWORK FOR ARTIFICIAL ECG GENERATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Martin Šagát**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Hejč**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Biomedicínské inženýrství a bioinformatika**

Ústav biomedicínského inženýrství

Student: Bc. Martin Šagát ***ID:*** 186199
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Návrh generativní kompetitivní neuronové sítě pro generování umělých EKG záznamů

POKYNY PRO VYPRACOVÁNÍ:

1) Seznamte se s principy generativních kompetitivních sítí (GAN) a vytvořte literární rešerši na toto téma. Dále se seznamte se základními principy arytmogeneze a specifickými projevy arytmií v EKG signálu. 2) Z dostupných dat sestavte vlastní databázi arytmických EKG záznamů vhodnou pro učení GAN. 3) Navrhněte architekturu GAN pro generování umělých arytmických EKG záznamů. Model implementujte a ověřte jeho funkci. 4) Na základě dosažených výsledků proveďte optimalizaci architektury a hyperparametrů modelu. 5) Vyhodnoťte a podrobně diskutujte dosažené výsledky.

DOPORUČENÁ LITERATURA:

[1] ZHU, F., YE, F. and FU, Y. Electrocardiogram Generation With a Bidirectional LSTM-CNN Generative Adversial Network. Scientific Reports, 2019, vol. 9:6734, pp 242:253

[2] HOCHREITER, S. and SCHMIDHUBER, S. Long Short-Term Memory. Neural Computation, 1997, vol. 9:, pp 1735–1780.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 29.5.2020

Vedoucí práce: Ing. Jakub Hejč

prof. Ing. Ivo Provazník, Ph.D.
předseda rady studijního programu

Abstrakt

Práca sa zaoberá generovaním EKG signálov pomocou generatívnych kompetitívnych sietí (GAN). Podrobne skúma základy umelých neurónových sietí a princípy ich fungovania. Teoreticky popisuje využitie a fungovanie a najčastejšie typy zlyhaní generatívnych kompetitívnych sietí. V práci bol odvodený všeobecný postup predspracovania signálov vhodných pre trénovanie GAN, ktorý bol využitý na zostavenie databázy. V tejto práci boli navrhnuté a implementované celkovo 3 rôzne modely GAN. Výsledky modelov boli vizuálne zobrazené a podrobne analyzované. Na záver práca komentuje dosiahnuté výsledky a navrhuje ďalšie smerovanie výskumu metód zaoberajúcich sa generovaním EKG signálov.

Klíčová slova

Generatívne kompetitívne siete, GAN, umelé neurónové siete, neurón, EKG, arytmie, generovanie signálov, Python, CNN, BiLSTM

Abstract

The work deals with the generation of ECG signals using generative adversarial networks (GAN). It examines in detail the basics of artificial neural networks and the principles of their operation. It theoretically describes the use and operation and the most common types of failures of generative adversarial networks. In this work, a general procedure of signal preprocessing suitable for GAN training was derived, which was used to compile a database. In this work, a total of 3 different GAN models were designed and implemented. The results of the models were visually displayed and analyzed in detail. Finally, the work comments on the achieved results and suggests further research direction of methods dealing with the generation of ECG signals.

Keywords

Generative adversarial networks, GAN, artificial neural networks, neuron, ECG, arrhythmias, signal generation, Python, CNN, BiLSTM

Bibliografická citace:

ŠAGÁT, Martin. *Návrh generativní kompetitivní neuronové sítě pro generování umělých EKG záznamů* [online]. Brno, 2020 [cit. 2020-05-29]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/126849. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav biomedicínského inženýrství. 74s. Vedoucí práce Jakub Hejč.

Prohlášení

Prohlašuji, že svou diplomovou práci na téma „Návrh generativní kompetitivní neuronové sítě pro generování umělých EKG záznamů“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ……

podpis autora

Poděkování

Děkuji vedoucímu diplomové práce Ing. Jakubovi Hejčovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne

…….
podpis autora

Obsah

Zoznam obrázkov a tabuliek

ÚVOD

V dnešnej pokročilej dobe je mnoho riešení ako detekovať a analyzovať signály EKG. Boli vyvinuté rôzne algoritmy na diagnostiku kardiovaskulárnych chorôb. Hodnotenie a analýza EKG kriviek, je dnes bežne realizovaná pomocou pokročilých počítačových metód, avšak v praxi stále pretrváva problém s nedostatkom signálov v dostupných databázach.

Tento problém rieši práve naša práca, ktorá sa zaoberá generovaním umelých EKG záznamov za účelom rozšírenia databázy dostupných signálov. Generovanie umelých EKG záznamov bolo realizované pomocou generatívnych kompetitívnych sietí (GAN), ktoré sú veľmi perspektívnou oblasťou umelých neurónových sietí.

V práci je podrobne vysvetlený okruh umelých neurónových sietí, ktorý je nutným krokom pre pochopenie funkcie GAN. Následne práca teoreticky rozoberá využitie GAN a základné princípy ich fungovania ale aj najčastejšie dôvody ich zlyhania. Teoreticky popisuje vznik, prejav, delenie a typy arytmií v signáloch EKG. Ďalej je v práci zostavená databáza signálov, ktorých použiteľnosť pre tréning GAN je následne analyzovaná. Na základe tejto analýzy práca poukazuje na vhodné predspracovanie signálov a postup pri výbere trénovacích signálov zo zostavenej databázy.

Celkovo práca navrhuje a priamo implementuje tri rôzne modely GAN. Začína jednoduchšími modelmi a postupne zvyšuje komplexnosť riešených problémov a modelov GAN. Popisuje architektúry použitých modelov a spôsoby ich trénovania. Dosiahnuté výsledky generovania signálov a priebehov učenia sú dôkladne okomentované spolu s vizuálnymi ukážkami výstupov. Výstupy sú ďalej odôvodnené a práca sa snaží pochopiť dôvody správneho aj nesprávneho fungovania modelov GAN.

Diskusia rozoberá nadobudnuté výsledky a slovne komentuje príčiny ich správania. Dosiahnuté výsledky sú porovnané s výsledkami odborných prác. Práca taktiež opravuje niektoré z tvrdení odborných článkov a vyvodzuje nové tvrdenia a riešenia. Práca poskytuje niekoľko tipov pre trénovanie modelov GAN pre generovanie EKG signálov na základe nadobudnutých vedomostí. Na záver práca poukazuje na smerovanie vývoja metód generovania EKG signálov pomocou GAN a navrhuje ďalšie postupy, ktoré by mohli byť kľúčové pre úspešné generovanie reálne vyzerajúcich EKG záznamov.

1 UMELÉ NEURÓNOVÉ SIETE

V súčasnej dobe rýchleho technologického pokroku a obrovského množstva dát, ktoré máme k dispozícií, vzniká problém tieto dáta spracovávať. Tento problém, dokážu veľmi dobre riešiť umelé neurónové siete. Pomocou počítaču môžeme analyzovať komplexné dáta a získať z nich cenné informácie, ktoré nám pomôžu vyriešiť rôzne problémy. Neurónové siete sú dnes preto často aplikované v rôznych vedeckých odvetviach, zasahujúcich oblasť matematiky, fyziky, medicíny či financií. [1]

Správne použitie neurónových sietí avšak vyžaduje podrobnú znalosť teoretických základov. Aby sme dosiahli optimálnych výsledkov, je za potreby poznať ich vlastnosti – klady aj zápory. Z dôvodu obrovskej komplexnosti neurónových sietí, sa v ďalších podkapitolách zameriame na základné pojmy a definície, ktoré postupne rozvinieme do špecifickejších oblastí, týkajúcich sa tejto práce.

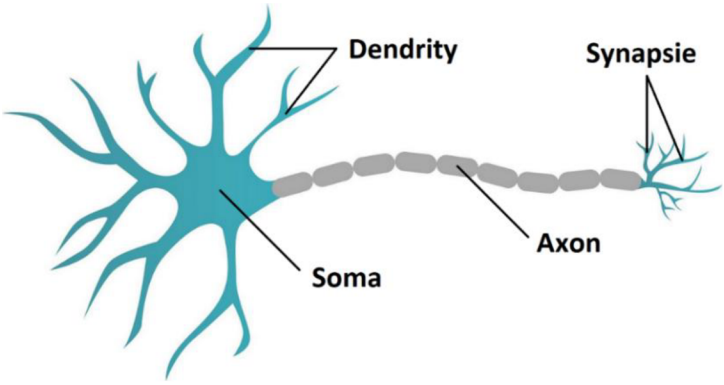
1.1 Biologický neurón

Pri pojme neurónové siete je nutné si zadefinovať, či máme na mysli biologické alebo umelé neurónové siete. Umelé neurónové siete boli inšpirované funkciou ľudského mozgu, ktorý sa skladá z približne 20 až 100 miliárd neurónov. Neuróny sú živé bunky, ktoré sú stavebným základom komplikovanej štruktúry mozgu a sú charakterizované svojimi mimoriadne zložitými spojeniami – biologické neurónové siete. [2]

Neurón (viď. Obr. 1.1) sa špecializuje na príjmanie, ukladanie alebo prenos jednotlivých informácií. Bez tejto funkcie by biologické systémy neboli schopné prežiť a rozvíjať sa. Biologický neurón pozostáva z 4 rôznych častí : [1]

- Soma – tzv. telo neurónu (obsahuje bunkové jadro)
- Dendrity – vstupy neurónu (sú zodpovedné za príjmanie informácií v bunke)
- Axon – výstup neurónu (vždy len 1), rôzne dlhý a silnejší ako dendrity
- Synapsie – sú zakončenia axónov a slúžia na prenos informácií medzi jednotlivými neurónmi. Môžeme ich teda prirovnať k akýmsi mostom, ktoré sa vyznačujú veľkou plasticitou. To znamená, že priechodnosť informácií cez tieto „mosty“ závisí na danej situácií (potrebe) - synapsie sa dokážu adaptovať (učiť) a ich priechodnosť sa v časovom priebehu môže meniť. [1]

Neuróny sú však omnoho komplikovanejšie bunky, ktoré komunikujú navzájom. Ak vstupné impulzy z ostatných neurónov (prichádzajúce do dendritov) prekročia určitý prah, neurón vyšle signál – tieto matematické operácie prebiehajúce v neurónových sietiach vytvárajú obrovskú variabilitu možných stavov. Snaha popísať spomínané mechanizmy viedla k vytvoreniu matematického modelu neurónu. [1, 2]



Obr. 1.1: Hlavné časti biologického neurónu, prevzaté z [9].

1.2 Matematický model neurónu

Základnou stavebnou jednotkou umelých neurónových sietí (ďalej už len UNS) je matematický model neurónu. Použitím odlišných typov matematických funkcií neurónov a ich vzájomnými kombináciami dokážeme vytvárať špecifické architektúry UNS. [1]

Prvý matematický model neurónu bol popísaný v roku 1943 [10] a nazývame ho McCulloch-Pittsov neurón (viď. Obr. 1.2). Hodnoty vstupného vektoru x_i simulujú impulzy prichádzajúce do dendritov, vektor synaptických váh w_{ji} j-tého neurónu odpovedá synaptickým zakončeniam a ich premenlivému toku informácií (neuróny majú schopnosť sa učiť – adaptovať), prah j-tého neurónu značíme θ_j a vyjadruje nám, kedy je neurón aktívny (výsledná suma vstupov násobených váhami musí byť väčšia ako hodnota prahu), u_j predstavuje tzv. potenciál j-tého neurónu, aktivačnú funkciu definuje blok AF a nakoniec dostávame výstup neurónu y, ktorý môžeme chápať ako výsledný signál nervovej bunky, ktorý sa pomocou axónu posiela do ďalších buniek, kde sa tento proces znova opakuje. [1, 4]

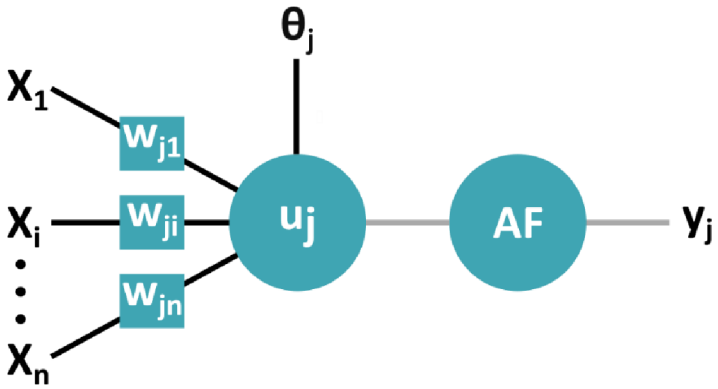
Pre lepšie pochopenie funkcie tohto neurónu sa hlbšie pozrieme na matematické operácie, ktoré v ňom prebiehajú. Potenciál j-tého neurónu vypočítame ako váženú sumu vektoru vstupných hodnôt (rovnica 1.1): [1]

$$u_j = \sum_{i}^{n} w_{ji}\, x_i + \theta_j \tag{1.1}$$

Vypočítaná hodnota potenciálu u_j je vstupom pre aktivačnú funkciu AF, ktorá nám určuje spôsob transformácie vstupných dát na výstupné hodnoty. Výstupnú hodnotu y_j, teda dostávame po dosadení do rovnice 1.2: [1]

$$y_j = f_{AF}(u_j) \tag{1.2}$$

Z tohto dôvodu ju často nazývame aj ako transformačnú funkciu neurónu. [1]



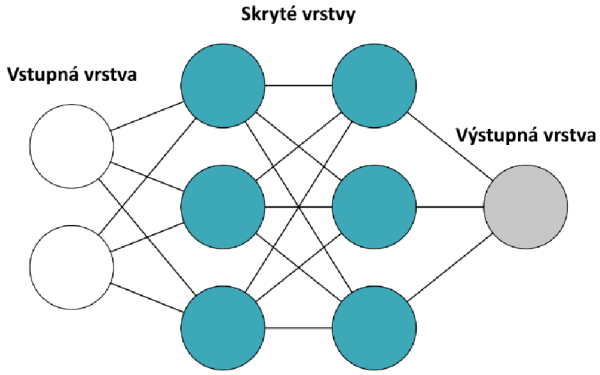
Obr. 1.2: Model McCulloch-Pittsovho neurónu.

1.3 Základné princípy návrhu neurónovej siete

Po zadefinovaní matematického modelu neurónu sa ďalej pozrieme ako takéto neuróny spojiť a vytvoriť tak jednoduchú neurónovú sieť. Každá umelá neurónová sieť je daná svojou architektúrou - počtom neurónov, spôsobom ich prepojenia a metódou učenia siete. [1]

Na Obrázku 1.3 môžeme vidieť jednoduchú architektúru UNS, ktorá sa skladá z troch častí : [1]

- Vstupná vrstva – jedná sa o distribučnú vrstvu (vstupných dát) a neprebiehajú na nej žiadne výpočty, vstupné dáta sú posielané ďalej do skrytých vrstiev
- Skryté vrstvy – príjmajú dáta zo vstupných vrstiev, ktoré pomocou rôznych matematických operácií transformujú a ďalej posielajú do výstupnej vrstvy
- Výstupná vrstva – obsahuje finálne výsledky (odozvy modelu na vstup), výstupy siete záležia od typu riešenej úlohy



Obr. 1.3: Zjednodušná architektúra viacvrstvovej doprednej neurónovej siete.

Počet neurónov a vrstiev volíme podľa povahy riešeného problému. Pri malom počte neurónov sieť nebude schopná aproximovať (pokryť) všetky vlastnosti riešeného

problému. Naopak priveľký počet neurónov môže spôsobiť tzv. preučenie siete - sieť stráca schopnosť generalizovať a učí sa zbytočne dlho. [3]

Ako sme uviedli v predchádzajúcej podkapitole, potenciál každého neurónu je počítaný ako lineárna kombinácia vstupov (viď. Rovnica 1.1). Každý neurón teda môžeme chápať ako rovnicu nadroviny, ktorej stupeň určuje počet vstupov – napríklad pre dve vstupné hodnoty dostávame rovnicu priamky, pre tri vstupné hodnoty zas rovnicu roviny. Zvyšovaním počtu neurónov zvyšujeme počet nadrovín, ktorými sme schopní rozdelovať dáta do jednotlivých tried. Do ďalšej vrstvy vstupuje výstup z predchádzajúcej vrstvy – týmto spôsobom je možné kombinovať nadroviny do otvorených alebo zatvorených konvexných usporiadaní. Tento proces sa opakuje, a končí až na výstupnej vrstve – zjednodušene môžeme tvrdiť, že čím viac neurónov a vrstiev použijeme, tým komplexnejší priestor dokáže sieť vytvoriť (simulovať). [3]

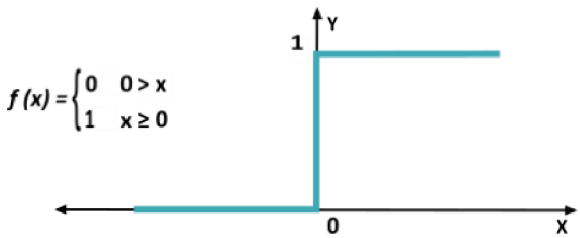
Predstavili sme si základné časti UNS – jej tri hlavné vrstvy a princípy správneho výberu počtu jej neurónov a vrstiev. Tieto znalosti nám ale ešte nebudú stačiť na pochopenie procesu vytvárania optimálnej architektúry UNS a preto sa v nasledujúcej podkapitole pozrieme podrobnejšie na už spomínané aktivačné funkcie.

1.4 Aktivačné funkcie UNS

Hlavnou úlohou aktivačnej funkcie (ďalej už len AF) je vhodná transformácia vstupných hodnôt na výstupné hodnoty, ktoré tvoria vstup do ďalších vrstiev siete. Je veľmi podstatné akú AF pri tvorbe UNS zvolíme. Výber samotnej AF záleží na riešenom probléme a definuje nám zložitosť výpočtov a teda priamo ovplyvňuje čas trénovania UNS. Aktivačné funkcie môžeme rozdeliť na tri hlavné skupiny – skokové, lineárne a nelineárne AF.[1, 11]

1.4.1 Skokové aktivačné funkcie

Tento typ AF transformuje vstupné dáta na základe zvoleného prahu. Ako príklad si uvedieme binárnu skokovú funkciu. Ako vidíme na Obr. 1.4, hodnoty väčšie ako prah ktorý je rovný 0, sa transformujú na 1, naopak hodnoty menšie ako prah budú rovné nule.[11]

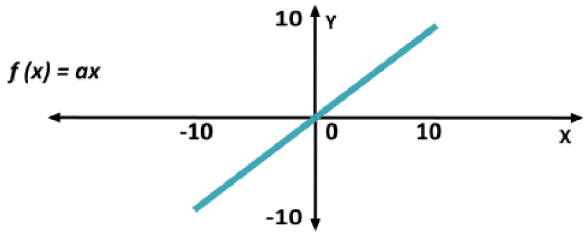


Obr. 1.4: Binárna skoková funkcia.

Jedná sa o veľmi jednoduchú AF, je výpočtovo nenáročná ale používa sa len zriedka. Nevýhodou tejto AF je neschopnosť klasifikácie vstupných dát do viacerých tried (dokáže klasifikovať len do dvoch tried).[11]

1.4.2 Lineárne aktivačné funkcie

Pre lineárne problémy ako aproximácia (regresia), je vhodné používať práve lineárne AF. Lineárne AF sú taktiež veľmi jednoduché a ľahko spočítateľné a používajú sa vo výstupných vrstvách siete.[13]



Obr. 1.5: Lineárna aktivačná funkcia.

Nevýhodou týchto AF je, že ich deriváciou dostávame konštantu – gradient tejto funkcie sa nebude meniť. Použitím lineárnych AF eventuálne „meníme“ všetky vrstvy (nezáležiac na ich počte) na výstupnú vrstvu, ktorá je lineárnou kombináciou prvej vrstvy. Môžeme teda povedať, že lineárne AF nám „transformujú“ všetky použité vrstvy do jednej výstupnej vrstvy a teda strácame možnosť tieto vrstvy kombinovať.[13, 12]

1.4.3 Nelineárne aktivačné funkcie

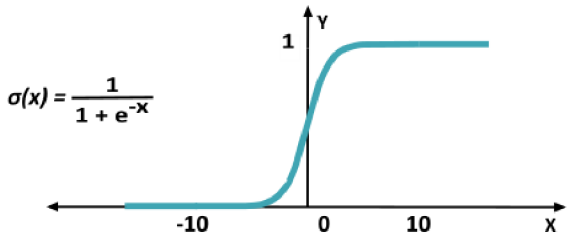
Medzi najpoužívanejšie AF patria práve nelineárne AF a to z dôvodu, že zavádzajú do výstupu neurónu nelinearitu. To nám umožňuje použiť algoritmus spätného šírenia chyby (pretože ich derivácia je viazaná na vstupné hodnoty) – sieť sa dokáže učiť. Taktiež nám táto vlastnosť dovoľuje implementovať viaceré vrstvy, na rozdiel od lineárnych AF. [13]

Nelineárne AF sú schopné lepšie mapovať vzťahy vstupu a výstupu – model obecne lepšie generalizuje a je vhodnejší pre reprezentáciu komplexných dát, ktoré sú vo väčšine prípadov nelineárne a majú viacdimenzionálny charakter (videá, obrazy, signály). [13]

Sigmoidálna (logistická) AF (viď Obr. 1.6) je používaná najmä v prípadoch, kedy chceme na výstupe neurónu predikovať pravdepodobnosť – jej výstup je vždy v intervale < 0,1 >. Má hladký gradient (výstupné hodnoty nemajú tendenciu náhle sa meniť, ako pri skokovej binárnej funkcií) a normalizuje výstup neurónu. Je vhodná na klasifikáciu dát do viacerých tried, avšak často je nahradzovaná AF typu Softmax. [13] [25]

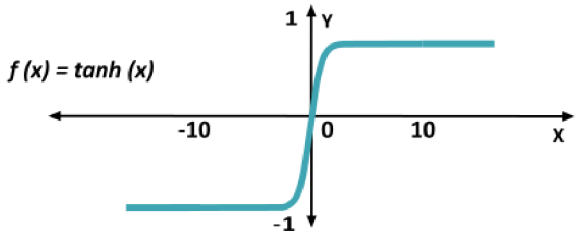
Nevýhodou logistickej AF je problém miznúceho gradientu (gradient vanishing problem). Funkcia sa pri veľmi vysokých alebo veľmi nízkych vstupných hodnotách

saturuje (dosiahne maxima alebo minima - 0/1) a gradient tak postupne vymizne alebo dosiahne nulových hodnôt. Ďalšou z nevýhod je, že výstupy tejto AF nie sú centrované okolo nuly – výstup bude vždy pozitívny. Zjednodušene to spôsobí, že gradient všetkých váh napájaných na jeden konkrétny neurón bude vždy pozitívny alebo negatívny – optimalizácia bude vždy prebiehať len jedným „smerom“ a je tým pádom výpočtovo náročnejšia. [13, 25]



Obr. 1.6: Sigmoidálna (logistická) aktivačná funkcia.

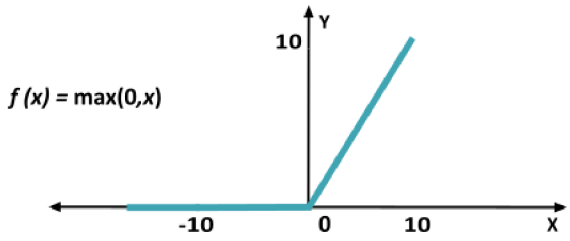
Druhou, veľmi podobnou nelineárnou AF je hyperbolický tangens. V porovnaní so sigmoidálnou (logistickou) AF sa používa častejšie (hlavne v rekurentných neurónových sieťach), má strmší gradient (viď Obr. 1.7) a jej výstupy sú centrované okolo nuly – výstup sa nachádza v intervale < -1,1 >. Práve z tohto dôvodu je optimalizácia siete pri jej použití rýchlejšia a jednoduchšia. Negatíva pri jej použití zostávajú rovnaké ako pri logistickej funkcií. [26]



Obr. 1.7: Hyperbolický tangens - aktivačná funkcia.

Medzi najpoužívanejšie AF radíme ReLU (viď. Obr. 1.8) a jej modifikáciu Leaky ReLU (viď. Obr. 1.9). Výhodou týchto AF je ich jednoduchosť – sú výpočtovo nenáročné a sieťam umožňujú rýchlu konvergenciu. Prvá spomenutá AF, je rovná nule pre všetky negatívne vstupné hodnoty a pre všetky pozítivne vstupné hodnoty sa chová rovnako ako lineárna AF – výstupné hodnoty sa teda pohybujú v intervale $< 0,\infty >$. [26]

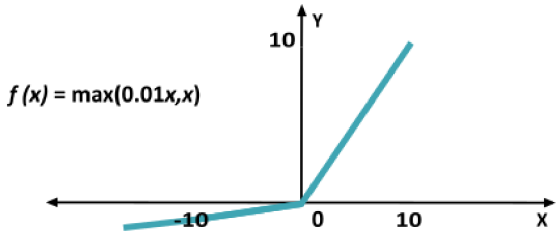
Na rozdiel od hyperbolického tangensu a logistickej AF nemá problém s miznutím gradientu a taktiež nedochádza k jej saturácií pri kladných vstupných hodnotách. Pri jej využití je aktivácia siete riedkejšia – niektoré neuróny (napr. 50% z celkového počtu) tak nebudú aktivované (pre $x < 0$) a celá sieť bude omnoho efektívnejšia. [12]



Obr. 1.8: ReLU aktivačná funkcia.

Táto výhoda je ale zároveň aj nevýhodou tejto AF. Gradient je nulový v oblasti kde x < 0, váhy sa teda nebudú aktualizovať – neuróny ďalej nebudú reagovať na vstupné hodnoty, hovoríme o tzv. dying ReLU probléme. Tento problém rieši práve Leaky ReLU – záporná časť tejto AF je narozdiel od ReLU mierne naklonená (nelineárna) – gradient je nenulový. Tento sklon môžeme meniť podľa potreby. [12, 25]

Obidve AF sa všeobecne používajú v skrytých vrstvách sietí, v praxi sa jedná najmä o konvolučné neurónové siete. [26]



Obr. 1.9: Leaky ReLU aktivačná funkcia.

1.5 Princíp trénovania UNS

Teraz, keď rozumieme funkcií matematického modelu neurónu, základným prvkom UNS a aktivačným funkciám, ktoré v nich môžeme použiť sa pozrieme na učenie siete.

1.5.1 Inicializácia váh

Ako prvé, musíme najrpv v sieti správne nastaviť váhy – hovoríme o tzv. inicializácií váh. Cieľom inicializácie váh je ochrániť výstupy z aktivovaných vrstiev pred ich nekontrolovateľným narastaním alebo pred problémom miznúceho gradientu, ktorý sme si popísali v podkapitole 1.4.3. [28, 29]

Existuje viacero typov inicializácie váh, uvedieme si ale len niektoré. Medzi najpoužívanejšie techniky inicializácií váh patria: [28, 29]

- Xavier inicializácia váh – používa sa pri tanh aktivačných funkciách.
- He inicializácia váh – používa sa pri ReLU aktivačných funkciách

.

1.5.2 Vybavovacia fáza UNS

Učenie siete začína vybavovacou fázou – vstupné hodnoty dopredne šírime sieťou, naprieč skrytými vrstvami, kde (ako bolo popísane v podkapitole 1.3) postupne prebiehajú matematické výpočty potenciálov všetkých neurónov (viď. Rovnica 1.1) aktuálnej vrstvy. Následne sú tieto vypočítané hodnoty transformované AF, ktoré sme popísali v podkapitole 1.4. Týmto spôsobom prechádzame všetky skryté vrstvy UNS a dostávame sa až k vrstve výstupnej, v ktorej dostávame výsledné výstupy siete.

Tieto výstupy (zväčsa sa jedná o reálne čísla) nakoniec ešte transformuje AF výstupnej vrstvy do požadovanej podoby. Napríklad v prípade, keď chceme klasifikovať výstupné dáta do dvoch tried, použijeme logistickú funkciu (alebo Softmax), ktorá nám tieto hodnoty transformuje do intervalu < 0,1 >.

1.5.3 Výpočet chyby siete

Základnou myšlienkou učenia siete, je upráva jej váh za cieľom minimalizácie chyby. Výpočet chyby je realizovaný tzv. chybovou funkciou siete. Výber chybovej funkcie závisí od riešeného problému a zvoleného spôsobu učenia siete. Chybové funkcie môžeme rozdeliť na 3 skupiny: [30]

- Regresné chybové funkcie – používame pri regresných modeloch, kde sa snažíme odhadnúť, predpovedať výstupnú hodnotu (reálne číslo)
    - MSE – využíva strednú kvadratickú chybu.
    - MSLE – využíva strednú kvadratickú logaritmickú chybu.
    - MAE – využíva strednú absolútnu chybu.
- Binárne klasifikačné chybové funkcie - používame pri modeloch, kde potrebujeme na výstupe určiť jednu z dvoch možností (binárne), napríklad áno/nie, trieda1/trieda2 a podobne.
    - Binárna krížová entropia – štandartne používaná pre binárne problémy (klasifikácia do tried 0 a 1).
    - Hinge chyba – alternatívna náhrada binárnej krížovej entropie, používaná hlavne v metódach podporných vektorov (SVM) s klasifikáciou do tried s hodnotou -1 a 1.
    - Kvadratická hinge chyba – modifikácia Hinge chyby, taktiež používaná v SVM, vyhladzuje priebeh chybovej funkcie a rovnako klasifikuje do tried s hodnotou -1 a 1.
- Chybové funkcie pre klasifikáciu do viacerých tried – používame pri modeloch, kde predikujeme pravdepodobnosť zaradenia výstupu do každej z viacerých tried (viac ako dvoch).
    - Multi-class krížová entropia – štandartne používaná pre zaradenie do viacerých tried (pre každú z tried dostávame pravdepodobnosť zaradenia).

- Riedka multi-class krížová entropia – modifikácia predošlej chybovej funkcie, používaná v prípadoch, kde máme väčší počet klasifikačných tried.
- Divergentná Kullback Leiblerova chyba – využívaná pri porovnávaní chýb distribúcií pravdepodobností.

Chybu siete môžeme zjednodušene chápať ako rozdiel požadovaných výstupov a výstupov aktuálnej iterácie – jej minimalizáciou sa UNS učí. Po vypočítaní aktuálnej chyby modelu nám jej minimalizáciu zaručí algoritmus spätného šírenia chyby.

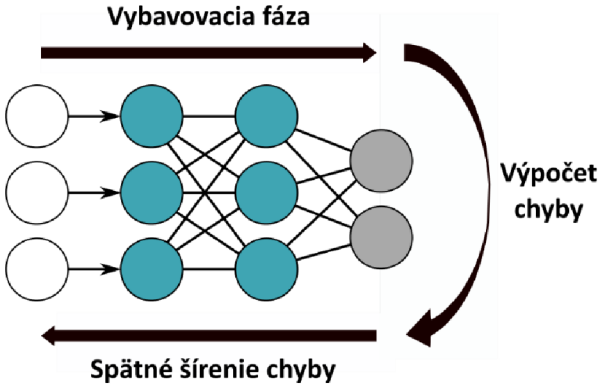
1.5.4 Algoritmus spätného šírenia chyby

Úlohou algoritmu spätného šírenia chyby je meniť váhy neurónov (predstavujú akúsi pamäť siete) tak, aby sa aktuálna chyba siete minimalizovala a aby sme sa priblížili bližšie k správnym výsledkom siete. Jedná sa o iteračný proces - po vybavovacej faze UNS dostávame výstup, ktorý podľa chybovej funkcií porovnáme s požadovaným výstupom, vypočítame chybu, ktorú pomocou spätného šírenia chyby minimalizujeme (viď. Obr. 1.10) až k vstupnej vrstve, kde sa tento proces opakuje, až kým sa nesplní podmienka ukončenia (počet iterácií, požadovaná minimálna chyba atď.).

Jeden takýto proces (vybavovanie siete + spätné šírenie chyby) pre všetky trénovacie data nazývame epochou, počet iterácií v jednej epoche definujeme ako podiel počtu všetkých trénovacích dát a veľkosťou skupiny (*batch_size*) spracovávaných dát. [1, 29]

Jednoduchým riešením by bolo nájsť všetky možné nastavenia váh, vypočítať chybu pre každé nastavenie a vybrať prípad s najmenšou chybou. Tento spôsob je ale viacmenej nerealizovateľný (výpočtovo náročný) pri použití hlbokých neurónových sietí, ktoré majú viacero skrytých vrstiev a veľký počet neurónov. [31, 1]

Práve z tohto dôvodu bolo nutné nájsť lepší spôsob hľadania optimálneho nastavenia váh – to vyriešil algoritmus spätného šírenia chyby. Tento algoritmus minimalizuje chybovú funkciu pomocou gradientných metód. Základné verzie využívajúce gradientného zostupu menia hodnoty váh a prahu, parametry AF však nemenia. Jediný optimalizačný parameter, ktoré je v týchto verziách možné meniť je parameter rýchlosti učenia η. [1]



Obr. 1.10: Grafické znázornenie algoritmu spätného šírenia chyby, prevzaté z [11]. Po vybavovacej fáze UNS dostávame výstupné hodnoty (sivé kruhy), z ktorých sa vypočíta chyba a tá je následne spätne šírená naprieč všetkými skrytými vrstvami siete (tyrkysové kruhy) až k vstupnej vrstve (biele kruhy).

Medzi základné optimalizačné učebné algoritmy, využívajúce gradientného zostupu patria nasledujúce metódy: [29]

- Metóda gradientného zostupu (GD) – váhy sa aktualizujú každú epochu, nepoužíva sa, je veľmi pomalá.
- Metóda náhodného gradientného zostupu (SGD) – váhy sú aktualizované každú iteráciu, čo je výpočtovo náročné ale metóda je ajtak rýchlejšia ako GD. Hodnoty sa rýchlo menia, čo umožňuje sieti "skočiť" do lepšieho globálneho minima.
- Mini-batch metóda gradientného zostupu – váhy sú aktualizované po určitom počte iterácií, redukuje rýchle zmeny váh, jedná sa o akýsi kompromis predchádzajúcich dvoch metód.

Veľkosť parametru rýchlosti učenia η, je veľmi dôležitá pre konvergenciu k správnemu výsledku siete. Pri vysokej hodnote η je na jednej strane učenie siete rýchle, ale môže ľahko „preskočiť" správne riešenie a divergovať. Naopak, pri nastavení η na veľmi nízke hodnoty, bude chyba klesať veľmi pomaly a sieť sa bude učiť príliš dlho. Tieto problémy riešia dnes už dostupné optimalizačné algoritmy. [4]

Modifikované metódy algoritmu spätného šírenia chyby využívajú navyše aj adaptáciu prahov a sklonov AF, celkovo využívajú viac optimalizačných parametrov (napríklad parameter momentu), sú výpočtovo náročnejšie ale dosahujú omnoho lepších výsledkov. Medzi najpoužívanejšie optimalizačné algoritmy patria hlavne: [29]

- SGD s momentom – obsahuje parameter momentu, ktorý akceleruje gradient do smeru predchádzajej iterácie a zároveň redukuje jeho pohyb v opačnom smere.
- Nestrovov akcelerovaný gradient (NAG) – modifikácia predošlej metódy, ktorá predchádza rýchlemu pohybu gradientu nesprávnym smerom.
- Adagrad – mení parameter učenia osobite, pre každú z váh (parametrov).

- Adadelta/RMSprop – modifikácia Adagradu, ktorá rieši problém rýchleho zmenšovania parametru učenia.
- Adaptívne určovanie momentu (ADAM) – štandartne používaný, kombinuje exponenciálny úpadok priemeru predošlých kvadratických gradientov momentov.
- Nestrovove akcelerované určovanie momentu (NADAM) – modifikácia ADAM, používa Nestrovovo momentum.

Po nastavení typu inicializácií váh, výberu chybovej funkcie a zvolení učebného algoritmu model môžeme začať učiť. Po skončení učenia, je model pripravený predikovať neznáme vstupné dáta.

1.6 Vrstvy neurónových sietí

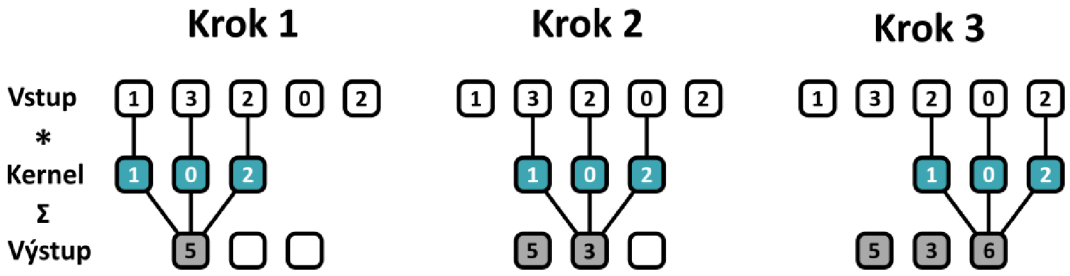
Teraz keď chápeme ako UNS fungujú, pozrieme sa na typy vrstiev UNS. Neurónové siete využívajú mnohé typy vrstiev, my sa však v tejto práci zameriame na vrstvy, ktoré budú ďalej implementované v praktickej časti práce.

1.6.1 Konvolučná vrstva 1D

Dimenzia použitej konvolučnej vrstvy záleží na riešenom probléme. V tejto práci sa zaoberáme generovaním EKG signálov a náš vstup je jednodimenzionálny (1D). V prípade spracovania obrazov používame 2D konvolučné vrstvy, alebo v prípade videa je možné použiť 3D konvolučné vrstvy.

Konvolučné vrstvy dokážu dobre zachytiť lokálne vlastnosti vstupných dát. Pre použitie 1D konvolučnej vrstvy, v neurónovej sieti, si najprv ale musíme zadefinovať niekoľko parametrov. Ak je táto vrstva úplne prvou použitou vrstvou v UNS, je nutné zadefinovať veľkosť a tvar vstupných dát. Následne si môžeme zvoliť počet použitých filtrov (určujú veľkosť dimenzie výstupu), veľkosť posuvného okna (kernel) a nakoniec veľkost kroku, s ktorým sa okno posúva. V prípade, že chceme zachovať veľkosť vstupných dát, musíme výstupnú sekvenciu doplniť nulami (pretože konvolučné vrstvy redukujú dlžku vstupných dát) – tzv. “padding”. [29]

Na Obrázku 1.11 vidíme, princíp fungovania 1D konvolučnej vrstvy, kde bol pre zjednodušenie použitý iba jeden filter, s posuvným oknom o veľkosti 3, ktoré sa posúvalo zakaždým o jeden krok. Výstupná sekvencia v tomto prípade nebola doplnená nulami.



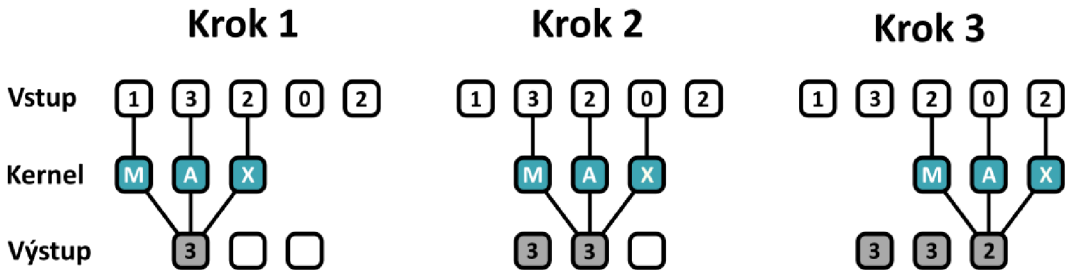
Obr. 1.11: Zjednodušený princíp fungovania 1D konvolučenej vrstvy. Hodnoty vstupného vektora sú postupne násobené s posuvným oknom (kernel) a ich suma sa ukladá na výstup.

Po konvolučných vrstvách sa v praxi často používa nelineárna aktivácia (napr. ReLU aktivačná funkcia – popísaná v podkapitole 1.4.3) nasledovaná „poolingovou" vrstvou. Na poolingové vrstvy sa pozrieme bližšie v nasledujúcej podkapitole.

1.6.2 MaxPooling vrstva 1D

Poolingové vrstvy možno rozdeliť na dve základné skupiny – Maximum Pooling a Average Pooling vrstvy. Cieľom týchto vrstiev je zmenšiť dimenzionalitu (v našom 1D prípade – podvzorkovať signál) a extrahovať charakteristické vlastnosti zo vstupných dát. Tieto nové abstraktné reprezentácie dát pomáhajú sieti riešiť problém preučenia zároveň redukujú počet parametrov siete a znižujú tak jej výpočtovú náročnosť. [29]

Opäť ako aj pri konvolučných vrstvách si musíme zvoliť parametre tejto vrstvy. V poolingových vrstvách si volíme veľkosť poolingového okna a veľkosť jeho kroku. Tieto vrstvy aplikujú na vstupné data príslušný filter (v prípade MaxPooling vyberáme maximum a v prípade AveragePooling vyberáme priemernú hodnotu v danom okne). Táto hodnota sa následne ukladá do výstupného vektoru (viď. Obr. 1.12 – sivé štvorce).



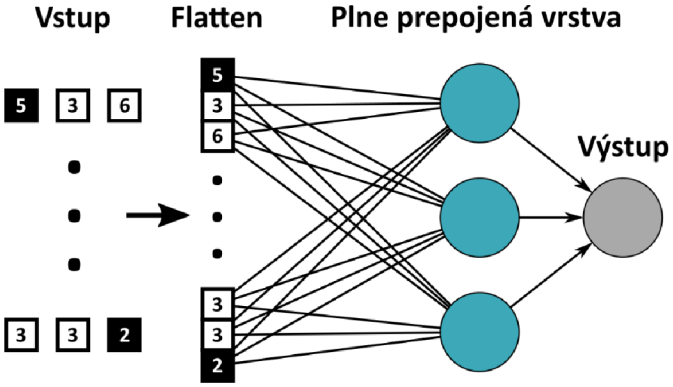
Obr. 1.12: Zjednodušený princíp fungovania MaxPooling 1D vrstvy. Na hodnoty vstupného vektoru je aplikovaný max filter, ktorý vyberá maximálnu hodnotu v príslušnom okne (kernel), ktorá je následne uložená do výstupného vektoru. V prípade AveragePoolingu sa namiesto maximálnej hodnoty vyberá priemerná hodnota v príslušnom okne.

1.6.3 Plne prepojené vrstvy

Tento typ vrstiev používame väčšinou vo finálnej (výstupnej) časti UNS. Plne prepojené vrstvy sa dokážu naučiť vzťahy medzi extrahovanými vlastnosťami vstupných dát z predchádzajúcich vrstiev (napr. po extrakcií príznakov pomocou

konvolučných a poolingových vrstiev) a tak pomôcť pri výslednej klasifikácií/regresií. [33]

Pred vstupom do plne prepojenej vrstvy sa využíva tzv. „flatten“ vrstva, ktorá nám transformuje dáta do stĺpcového vektoru ako môžeme vidieť na Obrázku 1.13. Všetky vstupné hodnoty predchádzajúcej vrstvy sú kompletne prepojené (každý s každým) s neurónmi plne prepojenej vrstvy, ale neobsahujú žiadne spojenia vrámci svojej (rovnakej) vrstvy. Výstupnú vrstvu navrhujeme podľa riešeného problému. Ak chceme na výstupe hodnotu pravdepodobnosti zaradenia do jednej z klasifikačných tried - použijeme napríklad sigmoidálnu aktivačnú funkciu (použitie AF opäť závisí na riešenom probléme). [33]



Obr. 1.13: Ukážka použitia plne prepojenej vrstvy s 3 neurónmi vo výstupnej časti siete. Všetky vstupné hodnoty predchádzajúcej vrstvy sú najprv transformované na stĺpcový vektor pomocou Flatten vrstvy, a následne plne prepojené (každý s každým) s neurónmi plne prepojenej vrstvy.

1.6.4 LSTM vrstva

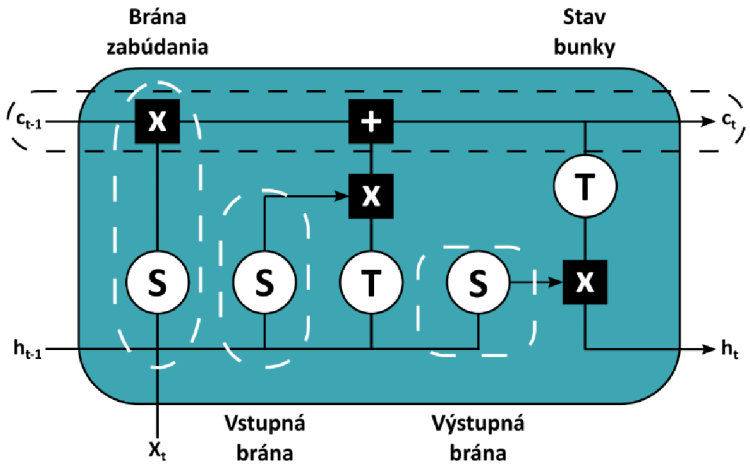
Keď pracujeme s dátami, ktoré majú určitú závislosť (náväznosť) medzi ich jednotlivými časťami je vhodné použiť rekurentné neurónové siete. Tie sú schopné, zapamätať si vzťahy sekvenčných (napr. časovo závislých) dát do pamäťových buniek, ktoré pomáhaju sieti pochopiť dočasné vzťahy a závislosti spracovávaných dát. [29]

Medzi najpopulárnejšie použitia rekurentných neurónových sietí patrí spracovávanie ľudskej reči – tieto siete dokážu pochopiť kontext viet, prekladať text do iných jazykov a podobne. Nevýhodou týchto sietí je ale problém miznúceho gradientu. Výstupné neuróny týchto sietí sú omnoho menej náchylné na vstupné dáta – nedokážu si dobre pamätať informácie zo začiatku sekvencií (nemajú dlhodobú pamäť) – majú krátkodobú pamať.

Tento problém rieši tzv. LSTM (long short-term memory) vrstvy. Tie boli prvý krát popísané v roku 1997 [34] a ako plynie z ich názvu – dokážu si dlhodobo pamätať krátkodobé závislosti spracovávaných dát a čiastočne riešia problém miznúceho gradientu.

Jedna bunka LSTM vrstvy pozostáva zo hlavných 4 častí [34]:

- Stav bunky – reprezentuje zdieľanú pamäť siete
- Brána zabúdania – rozhoduje o zachovaní informácie
- Vstupná brána – reguluje skrytý aj výsledný stav bunky
- Výstupná brána – definuje nasledujúci skrytý stav bunky



Obr. 1.14: Bunka LSTM a jej jednotlivé časti, prevzaté z [29]. Jedna LSTM bunka pozostáva zo 4 hlavných častí - stavom bunky (čierne prerušovaná čiara), a troch brán (biele prerušované čiary). Sigmoidálnu aktiváciu značí blok *S*, hyperbolický tangens naopak blok *T*. Do bunky vstupuje predchádzajúci stav bunky c_{t-1}, predchádzajúci skrytý stav h_{t-1} a aktuálny vstup x_t. Výstupom z LSTM bunky je nový stav bunky c_t a nový skrytý stav h_t.

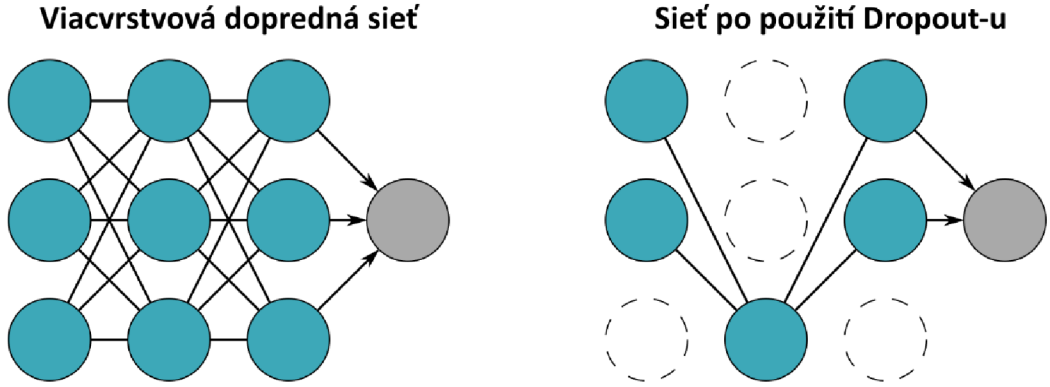
Na začiatku do bunky vstupujú tri premenné (viď. Obr. 1.14) - predchádzajúci stav bunky c_{t-1}, predchádzajúci skrytý stav h_{t-1} a aktuálny vstup x_t. Prvým krokom, je výpočet výstupu brány zabúdania, do ktorej vstupuje h_{t-1} a x_t . Pomocou sigmoidálnej aktivácie *S* je určená miera relavancie týchto dát (0 – zabudnúť, 1 – ponechať). Následne rovnakým postupom vypočítame po sigmoidálnej aktivácií výstup vstupnej brány, ktorý je následne násobený s výstupom z aktivácie h_{t-1} pomocou hyperbolického tangensu *T*. Týmto spôsobom regulujeme skrytý aj výsledný stav bunky. Tretím krokom je výpočet aktuálneho stavu bunky c_t , ktorý dostaneme sčítaním (prvok po prvku) regulovaného výstupu vstupnej brány a výstupu z brány zabúdania. Posledný krok vykonáva výstupná brána, ktorej výstup h_t získame vynásobením predchájúceho skrytého stavu h_{t-1} po sigmoidálnej aktivácií *S* s aktuálnym regulovaným (po aktivácií hyperbolickým tangensom - *T*) stavom bunky. [34]

1.6.5 Dropout vrstva

Problém preučenia UNS, kedy sieť produkuje dobré výsledky na trénovacích dátach ale zlé na testovacích, rieši tzv. “Dropout” vrstva. Jedná sa v podstate o formu regularizácie UNS, kedy sieti môžeme nastaviť pravdepodobnosť “vypnutia” každého z neurónov. [29]

V trénovacej fázi siete sa každý jeden z neurónov môže s pravdepodobnosťou p (ktorú si môžeme nastaviť, napríklad $p = 0.5$) náhodne vypnúť, a teda jeho váhy nevstupujú (sú ignorované) do učebného procesu siete. Pre zachovanie energie, je každý výstup v ďalšej vrstve podelený parametrom p z predchádzajúcej vrstvy. Následne sa algoritmom spätného šírenia chyby (podkapitola 1.5.4) aktualizujú váhy neurónov, ktoré neboli vypnuté a proces sa takto opakuje znova. [29]

Týmto spôsobom sa UNS učí na rôznych sub-architektúrach siete a dokáže sa naučiť robustnejšie vlastnosti, avšak za cenu väčšej výpočtovej náročnosti. Ako vyzerá sieť po použití dropoutu môžeme vidieť na Obrázku 1.15. [29]

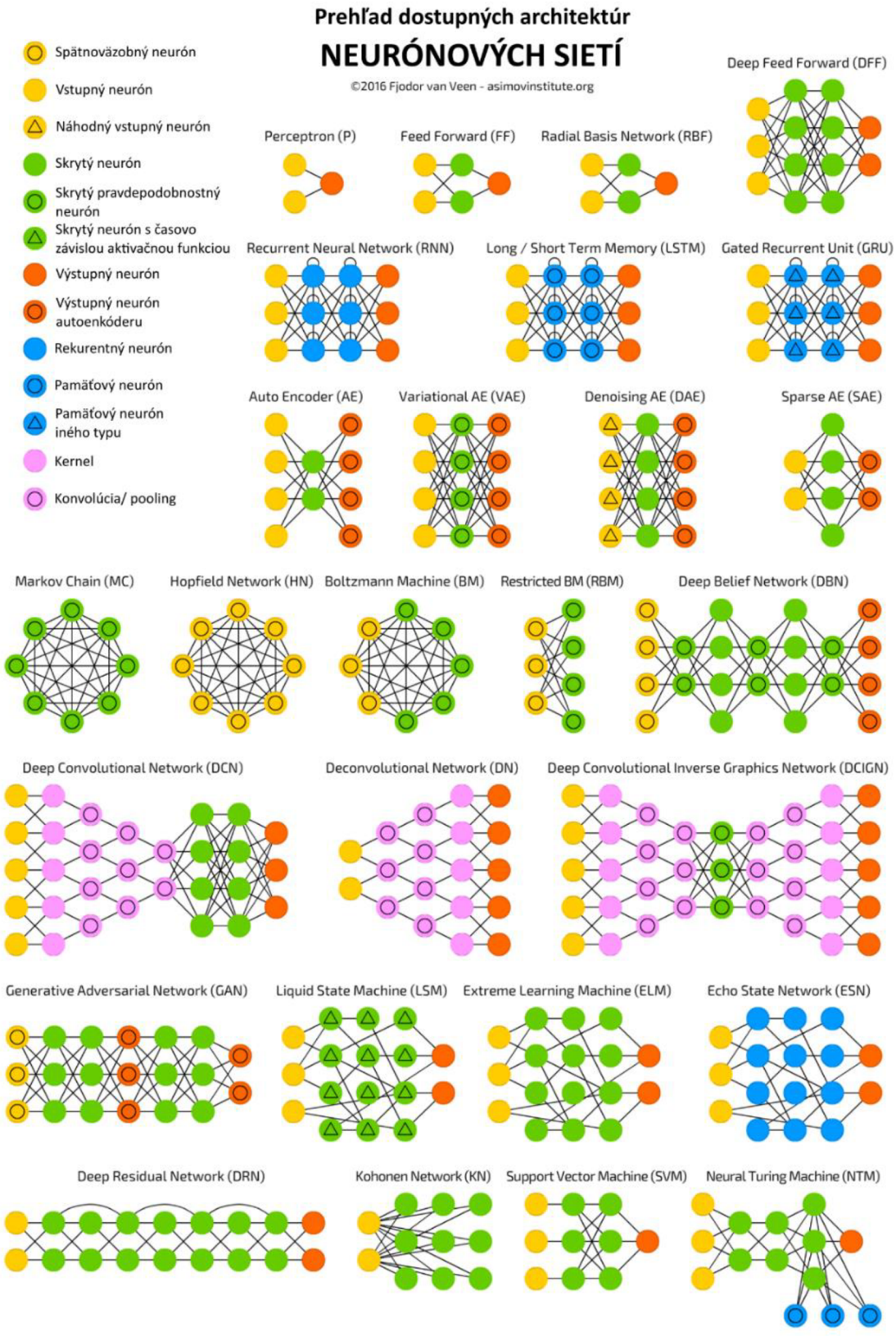


Obr. 1.15: Jednoduché znázornenie použitia Dropout vrstvy na viacvrstvovú doprednú sieť, prevzaté z [29]. Vidíme, že po použití Dropout vrstvy sieť vypla 4 neuróny a na aktualizácií váh sa podieľa len zvyšných 5 neurónov.

1.7 Architektúry neurónových sietí

V predchádzajúcej podkapitole sme si popísali niekoľko typov vrstiev používaných v UNS. Týchto vrstiev je však omnoho viac, a každá z vrstiev má svoje výhody aj nevýhody. Skladaním jednotlivých typov vrstiev a volením ich parametrov môžeme budovať tzv. architektúry neurónových sietí.

Ako môžeme vidieť na Obrázku 1.16, dostupných architektúr neurónových sietí je v dnešnej dobe nespočetne veľa. V tejto práci budeme na generovanie umelých signálov EKG využívať práve generatívne kompetitívne siete (GAN), na ktoré sa pozrieme v nasledujúcej kapitole.

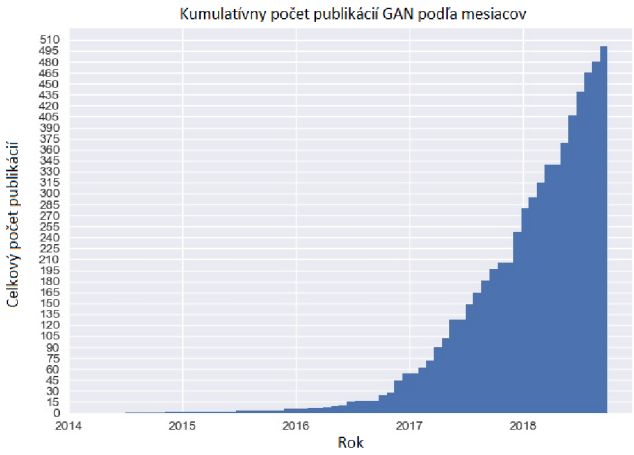


Obr. 1.16: Prehľad dostupných architektúr neurónových sietí, prevzaté z [27].

2 GENERATÍVNE KOMPETITÍVNE NEURÓNOVÉ SIETE (GAN)

Generatívne kompetitívne siete nazývané aj ako „Generative Adversarial Networks“ (ďalej už len GAN) sú pomerne novou oblasťou v strojovom učení, prvý článok na túto tématiku bol publikovaný v roku 2014 [5]. Typologicky ich radíme do metód učenia UNS bez učiteľa, GAN dokážu nájsť vlastnosti a charakteristiky vstupných dát – týmto spôsobom dokážu generovať nové dáta, ktoré sú veľmi podobné dátam trénovacím. [5]

Ich využitie teda spočíva v náhodnom (výstup je zakaždým iný) generovaní nových dát, ktoré ale stále pripomínajú vstupné dáta, na ktorých boli trénované. Použitie GAN má veľký potenciál v budúcnosti, jedná sa o veľmi rýchlo rozvíjajúci sa obor (viď. Obr. 2.1) – schopnosť generovať realistické dáta je možné využiť v rôznych oblastiach. Pôvodný článok z roku 2014 [5], bol aktuálne (máj 2020) citovaný na portáli *Web of Science* už celkovo 5409 krát. [8]



Obr. 2.1: Kumulatívny počet publikácií GAN podľa mesiacov, do konca roku 2018, prevzaté z [35].

2.1 Využitie GAN

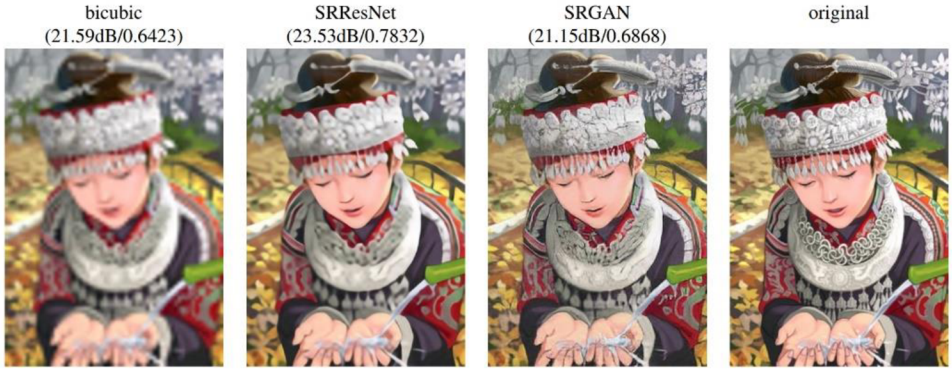
Informácií a dát je v dnešnej technologickej dobe až priveľa, a preto sa môže zdať, že generovanie nových bude zbytočné. Predstavíme si teda hlavné dôvody študovania a využitia GAN : [8]

- Aplikovanie v modeloch s posilneným učením [14, 15, 16] – GAN sú schopné vygenerovať množstvo možných výsledkov (rozšíriť trénovacie dáta) a tak pomôcť s učením a predikovaním modelu (podrobnejšie v [8] - sekcia 5.6).
- Možnosť trénovať GAN aj s chýbajúcimi dátami – použitie v modeloch s čiastočne riadeným učením[17, 18], kde sa používa malý počet trénovacích dát zaradených do klasifikačných tried (podrobnejšie v [8] - sekcia 5.4).

- Použitie v grafike [20, 21] – interaktívne programy, ktoré dokážu „kresliť“ alebo editovať obrázky na základe vstupov (úprav) používateľa.
- Prevod obrazu na obraz [22] – široké spektrum aplikácií ako napríklad prevod satelitných snímkov na mapy, prevod skečov ľudských tvárí na reálne fotografie (kriminalistika) a rôzne iné kreatívne využitia (viď. Obr. 2.2).

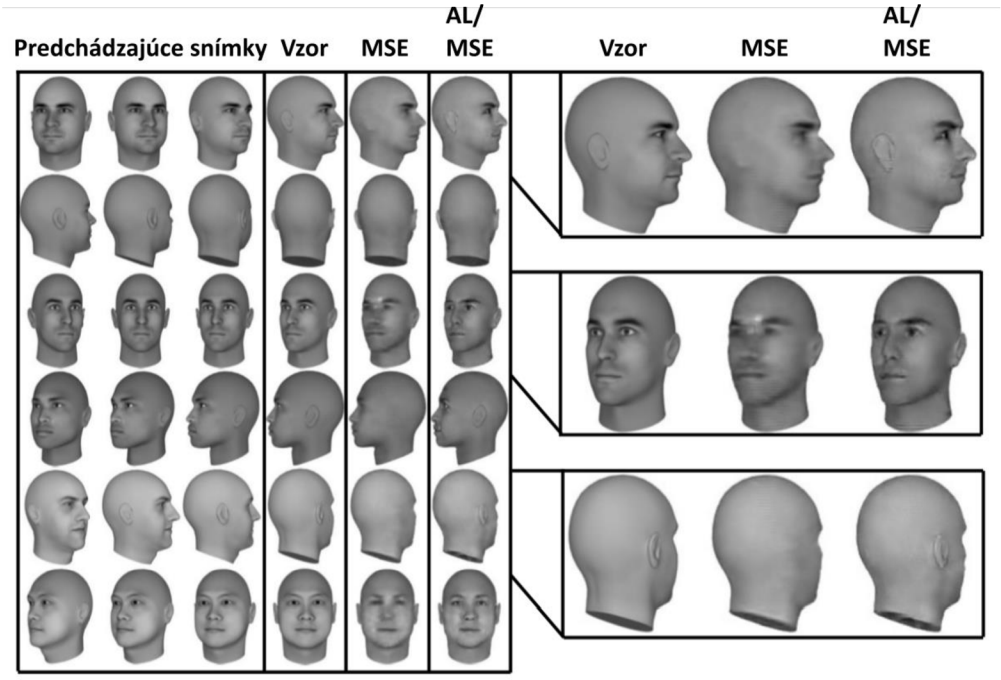
Obr. 2.2: Ukážka použitia GAN na prevod obrazu na obraz, prevzaté z [22].

- Super-resolution techniky [19] – v porovnaní s metódami využívajúcimi minimalizáciu MSE medzi vygenerovaným obrázkom a vzorom, GAN dokážu lepšie zachytiť podrobné detaily a textúry obrázkov (viď Obr. 2.3).



Obr. 2.3: Zľava do prava: Výsledok bikubickej interpolácie, hlboká reziduálna sieť optimalizovaná pomocou MSE, reziduálny GAN optimalizovaný na základe kritéria definovaného na základe ľudského vnímania, prevzaté z [19].

- Schopnosť GAN generovať viacero unikátnych výstupov [24] – na rozdiel od väčšiny klasických prístupov strojového učenia, kde sa snažíme minimalizovať MSE (stredná kvadradická chyba) medzi požadovaným (správnym) výstupom a modelom vygenerovaným výstupom (dostávame len jeden výsledok), GAN dokáže vygenerovať viacero rozlišných výstupov. Tento rozdiel je dobre vidieť na Obrázku 2.4.
- Generovanie (neexistujúcich) ľudských tvárí [5, 37, 38, 39, 40] – praktický význam tohto využitia GAN môžeme chápať ako tzv. „benchmark“ (test) výkonnosti GAN. Na Obrázku 2.5 môžeme vidieť vývoj schopnosti GAN generovať reálne vyzerajúce ľudské tváre.



Obr. 2.4: Ukážka predikcie snímku z videosekvencie, prevzaté z [24]. Úplne vľavo vidíme pohybujúce sa (predchádzajúce) 3D snímky modelu hlavy. Model bol trénovaný predikovať nasledujúci snímok.

Obr. 2.5: Ukážka postupného vývoju schopnosti GAN generovať reálne vyzerajúce ľudské tváre (všetky uvedené tváre sú vygenerované), prevzaté postupne zľava do prava [5, 37, 38, 39, 40]

- V tejto práci sa budeme podrobnejšie zaoberať využitím GAN na generovanie umelých EKG signálov za účelom rozšírenia databázy dostupných signálov podobne ako v [7, 23, 41, 42].

2.2 Princíp fungovania GAN

Princíp GAN spočíva vo vzájomnom "súboji" dvoch neurónových sietí – preto názov kompetitívne siete. Prvú zo sietí nazývame generátor a druhú diskriminátor. [5]

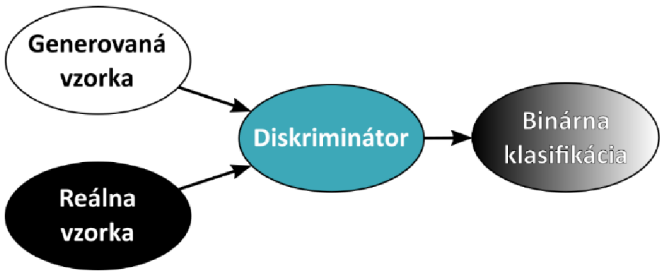
Pre jednoduchšie pochopenie môžeme generátor prirovnať k falšovateľovi bankoviek, ktorý sa snaží vyrobiť čo najdôveryhodnejšie falzifikáty. Diskriminátor naopak prirovnávame k polícii a teda jeho úlohou je odhaliť, či sa jedná o falzifikát alebo pravé peniaze. Zo začiatku polícia (diskriminátor) veľmi ľahko odhalí, že sa jedná

o falzifikát a s vysokým sebavedomím (pravdepodobnosťou) tieto bankovky určí ako falošné. Naopak falšovateľ, sa na svojich chybách rýchlo naučí, čo bude potreba zmeniť, aby vyrobil lepší falzifikát. [5]

Tento proces sa opakuje, až do bodu, kedy polícia nedokáže rozoznať falzifikát od originálu – vtedy sme dosiahli optimum a považujeme model za naučený. Model už od tejto chvíle netrénujeme a generátor (falšovateľ) už len využívame na generovanie nových falzifikátov, ktoré diskriminátor (polícia) nedokáže odhaliť. [5]

2.2.1 Diskriminátor

Diskriminátor je prvá z neurónových sietí použitá v GAN. Ako sme už spomenuli v predchádzajúcej podkapitole, jej úlohou je rozhodnúť či sú vstupné dáta reálne (z trénovacej množiny) alebo falošné (vygenerované generátorom). Jedná sa teda o binárny klasifikátor.



Obr. 2.6: Bloková schéma diskriminátoru v GAN. Do diskriminátoru vstupuje generovaná alebo reálna vzorka (vždy však len jedna, nikdy nie dve naraz) a úlohou diskriminátoru je rozhodnúť či sa jedná o reálnu alebo falošnú (generovanú) vzorku.

Architektúra diskriminátoru sa väčšinou skladá zo série blokov konvolučných vrstiev v kombinácií s aktivačnými a poolingovými vrstvami. Úlohou týchto vrstiev je extrahovať zo vstupných dát čo najviac príznakov a charakteristických vlastností (tak ako bolo popísané v podkapitole 1.6) aby bol diskriminátor schopný správne klasifikovať. Počet týchto blokov (hĺbku siete) sa snažíme zvoliť čo najoptimálnejšie – dostatočne hlbokú aby dokázala správne klasifikovať dáta, ale zároveň s čo najmenším počtom parametrov z dôvodu výpočtovej náročnosti siete.

Toto však nie je pravidlom, a v architekúrach diskriminátorov sa vyskytujú rôzne kombinácie typov použitých vrstiev. Každý riešený problém si vyžaduje individuálny prístup a navrhnutie správnej architektúry je veľmi dôležité, avšak komplexnosť funkcionality GAN spočíva vo vzájomne vyváženom súboji dvoch UNS. V ďalšej podkapitole sa preto pozrieme na hlavného súpera diskriminátoru – generátor.

2.2.2 Generátor

GAN zaraďujeme do generatívnych modelov hlbokého učenia. Tieto modely riešia omnoho komplikovanejší problém ako diskriminačné modely (napr. klasifikátory). Na rozdiel od diskriminačných modelov, kde hľadáme hranicu, ktorá nám dáta rozdelí do

skupín, v generatívnych modeloch sa snažíme nájsť rozloženie pravdepodobnosti, ktoré charakterizuje dáta, ktoré chceme generovať. [43]

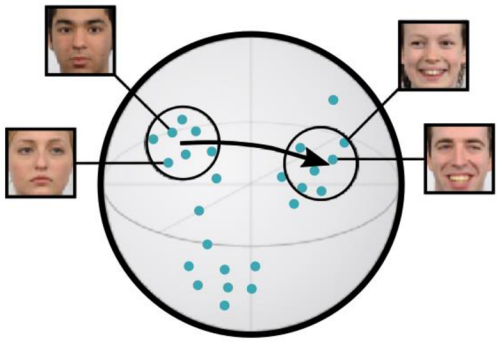
Generátor je druhou použitou UNS v GAN. Jeho úlohou je naučiť sa generovať prijateľné vzorky dát na základe spätnej väzby diskriminátora. Podrobnejší popis ako generátor využíva váhy diskriminátora pre svoj tréning, si popíšeme v nasledujúcich podkapitolách. Teraz je ale dôležité uvedomiť si, ako generátor principiálne funguje.

Vstupom do generátoru je náhodný vektor čísel z tzv. latentného priestoru, ktorého dimenziu si môžeme nastaviť. Generátor tento náhodne vygenerovaný vektor (šum) transformuje na zmysluplný výstup (viď. Obr. 2.7). Generátor sa týmto spôsobom snaží "namapovať" vzorky z latentného priestoru, na čo najreálnejšie vyzerajúci výstup.

Obr. 2.7: Bloková schéma generátoru v GAN. Do generátoru vstupuje náhodný vektor (napríklad z Gaussovského rozloženia) a úlohou generátoru je vygenerovať novú vzorku, ktorá bude čo najviac pripomínať reálnu vzorku (z trénovacích dát).

2.2.3 Latentný priestor

Latentný priestor je multidimenzionálny priestor, z ktorého si generátor náhodne vyberá vzorky (šum), ktoré následne použije na vygenerovanie dôveryhodného výstupu. Tento vstupný šum je volený z určitého typu rozloženia – väčšinou sa používa Gaussovské alebo uniformné rozloženie pravdepodobnosti. [44]



Obr. 2.8: Ukážka latentného priestoru generatívneho modelu, prevzaté z [44]. Generatívny model (v našom prípade GAN) sa počas tréningu učí mapovať výstupné vzorky (ľudské tváre) do multidimenzionálneho latentného priestoru.

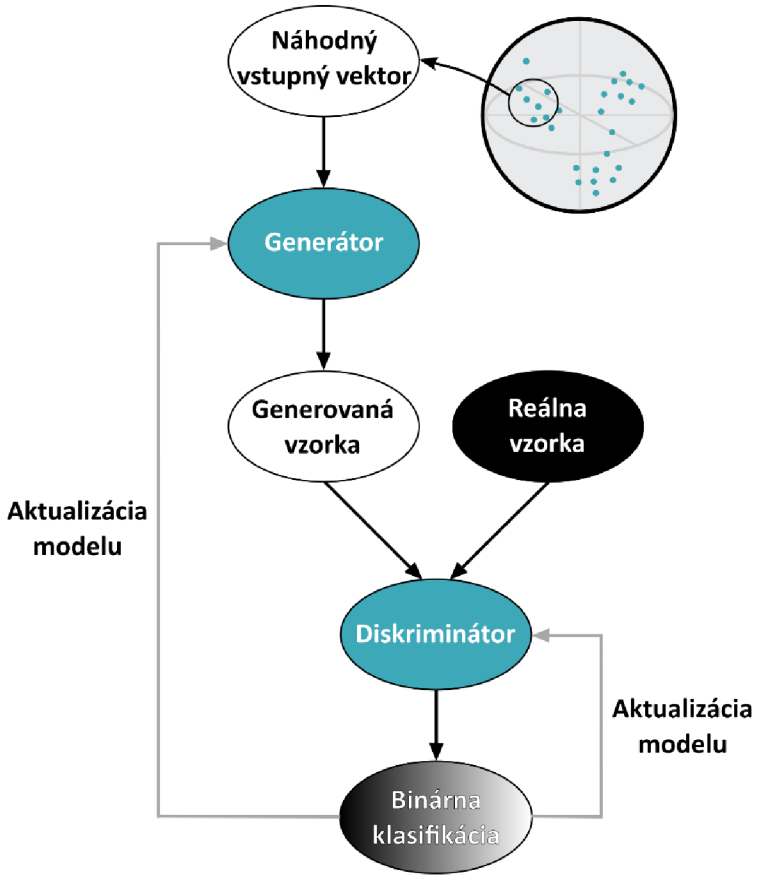
Povedzme, že si zvolíme celkovo 5 dimenzií a teda dostávame abstraktný 5D priestor. Každý bod v tomto priestore je reprezentovaný 5 číslami (súradnicami). Po dostatočne dlhom tréningu, budú body v latentnom priestore zodpovedať bodom z výstupnej oblasti a týmto spôsobom reprezentovať rozloženie vstupných dát. Zjednodušene môžeme povedať, že generátor sa snaží týmto bodom dať zmysel – na

Obrázku 2.8, vidíme, že fotografie usmievajúcich sa ľudí sa nachádzajú v inej časti latentného priestoru ako fotografie ľudí s neutrálnym výrazom. [44]

Po naučení modelu, môžeme generátor využiť na generovanie nových, náhodne vyzerajúcich ľudských tvárí. Pokročilejšie techniky dokonca dokážu zistiť vzťahy medzi polohou v latentnom priestore a výstupným obrazom – a týmto spôsobom vykonávajú aritmetické operácie. Ak poznáme, kde sa v latentnom priestore nachádzajú fotografie mužov s neutrálnym výrazom tváre a budeme sa pohybovať smerom k priestoru, kde sa vyskytujú fotografie mužov s úsmevom, postupne sa bude aktuálny neutrálny výraz výstupu tváre meniť na usmiatú mužskú tvár (šípka na Obr. 2.8). [44]

2.2.4 Zostavenie modelu GAN

Teraz, keď sme si predstavili všetky dôležité súčasti GAN, môžeme zostaviť celkový model. Model je zostavený prepojením diskriminátoru, generátoru a latentného priestoru ako môžeme vidieť na Obrázku 2.9.



Obr. 2.9: Ukážka zjednodušenej architektúry GAN.

Vstupom do generátoru je vektor náhodne vybraných hodnôt z latentného priestoru. Generovaná vzorka a reálna vzorka tvoria dve skupiny vstupných dát pre diskriminátor. Jeho úlohou je určiť pravdepodobnosť, či sa jedná o falošnú alebo reálnu vzorku (binárna klasifikácia). Diskriminátor je trénovaný postupne v skupinách, ktorých veľkosť definuje parameter *batch_size*, obsahujúcich ako falošné (trieda 0), tak i reálne

vzorky (trieda 1). Trénovanie generátoru je omnoho komplikovanejšie – generátor využíva "zmrazené" váhy diskriminátoru pre aktualizovanie svojich váh. Tento proces si ale podrobnejšie vysvetlíme v praktickej časti práce.

2.2.5 Definovanie matematického modelu GAN

Diskriminátor si môžeme zadefinovať ako funkciu D, do ktorej vstupujú hodnoty x a táto funkcia používa parametre $\theta^{(D)}$. Generátor si zadefinujeme ako funkciu G, do ktorého vstupujú hodnoty z a jej parametre označíme $\theta^{(G)}$. Úloha diskriminátoru spočíva v minimalizácií svojej chybovej funkcie $J^{(D)}(\theta^{(D)}, \theta^{(G)})$. K jej minimalizácií však, môže využiť len svoje parametre $\theta^{(D)}$, rovnako ako generátor môže využiť svoje parametre $\theta^{(G)}$ k minimalizácií vlastnej chybovej funkcie $J^{(G)}(\theta^{(D)}, \theta^{(G)})$. [8]

Obidve neurónové siete (diskriminátor aj generátor) majú teda vlastné chybové funkcie a vlastné optimalizačné parametre. [8]

2.2.6 Spôsob trénovania GAN

Trénovanie GAN môže znieť ako komplikovaná úloha, ale môžeme ho popísať nasledovne. Najprv vyberieme reálne vzorky x (z trénovacích dát) o zvolenej veľkosti (*batch_size*). Druhým krokom je vygenerovanie falošných vzoriek z (z latentného priestoru, ktorého dimenziu si volíme) o rovnakej veľkosti. [8]

Samotné trénovanie využíva metódy náhodného gradientného zostupu (popísané v podkapitole 1.5.4) a pozostáva z dvoch simultálne prebiehajúcich krokov. Prvý z krokov aktualizuje parametry $\theta^{(D)}$, za účelom minimalizácie $J^{(D)}$ a zároveň druhý z krokov aktualizuje parametre $\theta^{(G)}$ a minimalizuje $J^{(G)}$. Optimalizačný učebný algoritmus je možné voliť, často sa používa SGD alebo ADAM. [8]

2.2.7 Typy chybových funkcií GAN

Ako sme už spomenuli v podkapitole 2.2.5, diskriminátor aj generátor majú vlastné chybové funkcie. Vo všetkých modeloch GAN sa pre diskriminátor používa rovnaká chybová funkcia, ktorú môžeme popísať ako: [8]

$$J^{(D)}\left(\theta^{(D)}, \theta^{(G)}\right) = -\frac{1}{2} E_{x \sim p_{data}} \log D(x) - \frac{1}{2} E_z \log\left(1 - D\left(G(z)\right)\right), \tag{2.1}$$

kde E_x odpovedá očakávanej hodnote všetkých reálnych (trénovacích) dát – teda 1, a naopak E_z odpovedá očakávanej hodnote všetkých generovaných dát – teda 0. Všetky modely GAN sa snažia o minimalizáciu tejto funkcie, ktorá je v podstate modifikáciou binárnej krížovej entropie (popísanej v podkapitole 1.5.3).

Pre generátor GAN môžeme voliť tri rôzne chybové funkcie - v prvom prípade ide o hru s nulovým súčtom čo znamená, že súčet ziskov a strát oboch hráčov je rovný nule. Akékoľvek zlepšenie jedného z hráčov vedie k úmernému zhoršeniu druhého hráča : [8]

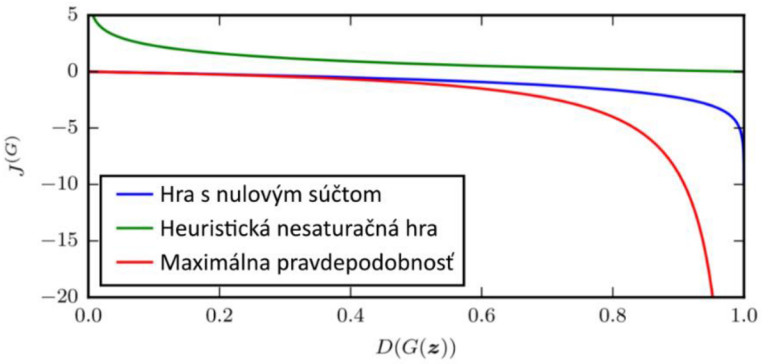
$$J^{(G)} = -J^{(D)} \tag{2.2}$$

Hra s nulovým súčtom sa často nazýva aj ako min-max hra, pretože jej riešenie spočíva v minimalizácií vonkajšej funkcie a maximalizácií vnútornej funkcie. Min-max hru definuje rovnica:

$$\min_G \max_D V(D,G) = E_{x\sim p_{data(x)}}[\log D(x)] + E_{z\sim p_z(z)}[\log\left(1 - D\big(G(z)\big)\right)]. \tag{2.3}$$

Diskriminátor *D* trénujeme tak, aby maximalizoval pravdepodobnosť správneho zaradenia dát do reálnych aj falošných tried. Zároveň, trénujeme generátor *G* tak, aby minimalizoval pravdepodobnosť chyby diskriminátoru ([log (*1 - D(G(z))*)]).

V praxi, sa ale využíva modifikácia Rovnice 2.3, pretože tento spôsob učenia neprináša dostatočný gradient pre učenie generátoru (viď. Obr. 2.10 – ľavá plochá časť modrej krivky). V počiatočných fázach tréningu, kedy sa generátor len začína učiť a podáva veľmi slabý výkon (generované dáta nepripomínajú reálne dáta), diskriminátor dokáže s vysokým sebavedomím (pravdepodobnosťou) tieto dáta odlíšiť od reálnych. V tomto prípade *log (1 - D(G(z)))* rýchlo saturuje.



Obr. 2.10: Porovnanie troch chybových funkcií generátoru GAN, prevzaté z [8]. Na y-ovej osi vidíme hodnotu chyby generátoru $J^{(G)}$, na x-ovej osi pravdepodobnosť zaradenia falošne vygenerovanej vzorky do jednej z tried (0 – falošná, 1 – reálna), určenú diskriminátorom.

Tento saturačný problém rieši práve heuristická nesaturačná hra, kde tiež minimalizujeme chybu binárnej krížovej entropie, tentoraz ale generátor neminimalizuje pravdepodobnosť chyby diskriminátoru (viď. Rovnica 2.3) ale maximalizuje pravdepodobnosť, že sa diskriminátor pomýli: [8]

$$J^{(G)} = -\frac{1}{2}E_z \log D\big(G(z)\big). \tag{2.4}$$

To v konečnom dôsledku spôsobí, že každý z hráčov bude mať silný gradient keď bude aktuálne „prehrávať“. [8]

Poslednou z hier je maximálna pravdepodobnosť, kde sa snažíme minimalizovať Kullback-Leiblerovú divergenciu, medzi dátami a modelom. Existuje mnoho spôsobov, ako tento princíp využiť v GAN modeloch, avšak táto metóda je ešte vo fázi výskumu, a podrobnejší popis tejto metódy je zbytočne nad rámec tejto práce. [8]

2.3 Najčastejšie dôvody zlyhania GAN

Trénovanie dvoch neurónových sietí je veľmi náročná úloha. Modely GAN sú často nestabilné a dochádza u nich k zlyhaniam. V tejto podkapitole si popíšeme najčastejšie príčiny zlyhania GAN modelov a ako tieto problémy riešiť.

Z dôvodu komplexnosti GAN, sú avšak tieto riešenia len akési heuristické tipy a triky, ktoré sa osvedčili v pomerne mladom obore modelov GAN a stále sú predmetom ďalšieho skúmania. Medzi 3 hlavné dôvody zlyhania modelov GAN patria: [36]

- Problém miznúceho gradientu
- Strata unikátnosti výstupu modelu
- Neschopnosť konvergovať do lokálneho optima

2.3.1 Problém miznúceho gradientu

V prípade, že diskriminátor ľahko dokáže rozlíšiť medzi falošnými a reálnymi vzorkami, je gradient generátoru malý a trénovanie generátoru môže zlyhať z dôvodu miznúceho gradientu. Tento problém bol spomenutý aj v podkapitole 2.2.7, a môžeme ho riešiť nasledovnými krokmi: [45]

- Použitie Wassersteinovej chybovej funkcie [46] – neminimalizuje vzdialenosť medzi distribúciou reálnych a falošných dát, ale minimalizuje "úsilie" zmeny potrebnej k vzájomnému priblíženiu týchto distribúcií.
- Použitie heuristickej nesaturačnej hry [8] – podrobne popísané v podkapitole 2.2.7.

2.3.2 Strata unikátnosti výstupu modelu

Jednou z hlavných výhod modelov GAN, je schopnosť generovať unikátne výstupy. Keďže do generátoru vstupuje zakaždým nový náhodne vygenerovaný šum, dostávame zakaždým iný výstup, ktorý ale stále pripomína objekt nášho záujmu. Prakticky to znamená, že ak chceme generovať ľudské tváre, očakávame na výstupe zakaždým novú (jedinečnú) tvár. [45]

Bežným problémom modelov GAN je ale strata unikátnosti výstupu. Zlyhanie modelu je spôsobené tým, že generátor sa naučí generovať len zopár vzoriek, ktorými dokáže diskriminátor dokonale "oklamať". Najlepšou stratégiou diskriminátoru je tieto nemenné vzorky z generátoru odmietať. Ak ale diskriminátor uviazne v lokálnom minime, nedokáže sa z neho v tomto prípade dostať, a generátor opakovane generuje to

isté – hovoríme o strate unikátnosti výstupu modelu. Aby sme tomutu zlyhaniu predišli je môžné použiť: [45]

- Wassersteinovú chybovú funckiu [46] – umožnuje trénovať diskriminátor bez rizika spojeného s miznúcim gradientom. To predchádza uviaznutiu diskriminátora v lokálnom minime – generátor sa znova musí snažiť vygenerovať nové výstupy.
- Rozbalené modely GAN [47] – chybová funkcia generátoru obsahuje nielen aktuálne ale aj budúce predikcie diskriminátoru a tak predchádza, preučeniu generátoru na aktuálny model diskriminátoru.

2.3.3 Neschopnosť konvergovať do lokálneho minima

Konvergencia u GAN modelov je ťažko dosiahnuteľná a často nestála. Na začiatku tréningu, by mala byť chyba generátoru omnoho väčšia ako chyba diskriminátoru – ten by mal jasne dokázať rozlišovať medzi falošnými a reálnymi dátami. Počas tréningu by sa chyba oboch UNS (generátoru a diskriminátoru) mala postupne znižovať. [45]

V prípade generátoru môžeme očakávať značné kolísanie chybových hodnôt (veľkú odchýlku), naopak chyba diskriminátoru by mala klesať pomalšie (stabilnejšie) s malou odchýlkou. V ideálnom prípade chceme, aby obe UNS dosiahli stabilný stav, kedy diskriminátor nedokáže rozlíšiť, či sa jedná o falošnú alebo reálnu vzorku a teda na výstupe dostávame 50% presnosť zaradenia do tried. [45]

Nájsť tento optimálny stav môže byť ale veľmi náročné. Ako sme si popísali v predchádzajúcom odstavci, chyba diskriminátoru s uplynulým časom tréningu klesá a generátor je aktualizovaný menej. Ak tréning GAN pokračuje aj po nájdení tohto optimámalneho stavu, môže dôjsť k zlyhaniu modelu. Generátor sa začne učiť na zmätených predikciách diskriminátoru (50/50) a kvalita generovaných vzoriek bude klesať. [45]

Z tohto dôvodu sa pri navrhovaní a trénovaní GAN architektúr snažíme o čo najstabilnejší tréning, počas ktorého si zároveň zálohujeme aktuálne modely (v prípade zlyhania modelu sa môžeme vrátiť k ešte fungujúcemu modelu). Ďalej si uvedieme niekoľko heuristických tipov, ktoré sa v rozvíjajúcom obore GAN osvedčili a mali by pomôcť čiastočne stabilizovať trénovací proces: [48]

- Normalizácia vstupných dát – dáta je vhodné normalizovať do intervalu <-1,1> z dôvodu následného použitia aktivačnej funkcie hyperbolického tangensu v generátore.
- Použitie heuristickej nesaturačnej hry pri tréningu GAN [5, 8] – tým predchádzame problému s miznúcim gradientom.
- Použitie Gaussovského rozdelenia v latentnom priestore. [44]

- Osobité trénovanie reálnych a falošných dát [48] – pri trénovaní diskriminátoru využívať osobitné trénovacie skupiny pre reálne a falošné vzorky (nemiešať ich dokopy).
- Použitie LeakyReLU aktivačnej vrstvy [48] – pre zamedzenie problému miznúceho gradientu.
- Transformácia hodnoty klasifikačných tried [17] – nahradenie „tvrdých" označení tried (0,1), hodnotami v určitom intervale (napr. 0-0.3, 0.7-1). Táto zmena hodnoty klasifikačných tried zvýši robustnosť nášho modelu a pomôže s problémom miznúceho gradientu.
- Pridanie šumu do diskriminátora [48] – pri trénovaní diskriminátora, prevrátime označenie klasifikačných tried pre určitý počet (napr. 5 %) vzoriek, a týmto spôsobom zavedieme šum do diskriminátoru.
- Použitie optimalizačného algoritmu ADAM [37] – kombinácia ADAM v generátore a SGD v diskriminátore dosahovala dobrých výsledkov.
- Sledovanie chyby generátoru aj diskriminátoru [48] – analýza a sledovanie aktuálnych chýb oboch UNS pred zlyhaním modelu.
- Použitie Dropout vrstvy v generátore. [22]

Je potrebné poznamenať, že väčšina modelov GAN je využívaná na generovanie obrazových dát (2D). My sa v tejto práci avšak zameriavame na generovanie časovo závislých dát (1D), v podobe EKG signálov. Nemôžeme teda predpokladať, že všetky spomenuté tipy a triky budú rovnako dobre aplikovateľné aj na náš konkrétny problém.

3 ZOSTAVENIE DATABÁZY SIGNÁLOV PRE TRÉNOVANIE GAN

Pred generovaním EKG signálov pomocou GAN sa najprv pozrieme bližšie na arytmické záznamy EKG. Po teoretickom rozdelení arytmií a následnej analýze poskytnutých signálov bude zostavená databáza vhodná na generovanie záznamov EKG.

Za normálnych podmienok srdce bije fyziologickým (sinusovým) rytmom. Môžu však nastať prípady, kedy toto neplatí – hovoríme o tzv. arytmiách srdca. Arytmie sú charakterizované nepravideľným srdcovým rytmom. [32]

3.1 Základné delenie arytmií podľa tepovej frekvencie

Pravideľný srdcový rytmus sa pohybuje v hodnotách medzi 60-100 úderov srdca za minútu. Pri arytmiách však srdce bije abnormálne, nastávajú 4 rôzne prípady: [32]

- Fibrilácie – majú nepravideľný rytmus srdca, charakterizovaný asynchrónnym šírením vzruchu
- Extrasystoly – predčasné, nepravideľné sťahy srdca, charakterizované abnormálne veľkým QRS komplexom
- Tachykardie – charakterizované zrýchlením srdcového rytmu (viac ako 100 úderov za minútu)
- Bradykardie – vystihujúce sa spomalením srdcového rytmu (menej ako 60 úderov za minútu)

Arytmie sa však líšia aj miestom svojho vzniku – jednotlivé miesta vzniku si ďalej popíšeme v nasledujúcej podkapitole.

3.2 Základné delenie arytmií podľa miesta vzniku

Arytmie sa môžu tvoriť na rôznych miestach v srdci. Jedná sa o poruchy srdcového rytmu, a teda môžu vznikať pri miestach zlyhania akéhokoľvek článku prevodného systému srdca (fyziologické šírenie vzruchu). Podľa miestu vzniku môžeme arytmie rozdeliť na 3 skupiny: [32]

- Sieňové arytmie – vznikajú vo svalovine predsiení
- Komorové arytmie – vznikajú vo svalovine komôr
- Junkčné arytmie – vznikajú v oblasti AV uzlu

Arytmie môžeme deliť na rôzne sub-kategórie a podľa rôznych kritérií, pre účely tejto práce si však vystačíme s jednoduchým rozdelením. V ďalšej kapitole sa pozrieme na ukážky konkrétnych typov arytmií a ich prejavov v EKG signáloch.

3.3 Vytvorenie databázy arytmických EKG signálov

Pre vytvorenie databázy arytmických signálov bolo použitých celkovo 6884 signálov. Jedná sa o krátke (10 sekundové) záznamy, ktoré boli poskytnuté vedúcim tejto práce. Vzorkovacia frekvencia týchto signálov bola 500 Hz a boli použité filtrované dáta. Celkovo sme mali k dispozícií 12 rôznych zvodov, ďalej sme ale pracovali už len so zvodom II.

Každý zo signálov obsahoval aj príslušný vektor anotácií, ktorý popisoval aké typy rytmov (poprípade artefaktov) sa v signále nachádzali. Celkovo tento vektor obsahoval 10 typov anotácií. Databáza bola rozdelená do 8 skupín (viď. Tabuľka 3.1), vynechané boli len dva typy anotácií. Jednalo sa o anotácie „noisy“ a „other“, ktoré anotujú zašumené signály s rôznymi artefaktami – pre tieto anotácie nebola vytvorená samostatná skupina signálov, ale stále sa môžu nachádzať v ostatných signáloch.

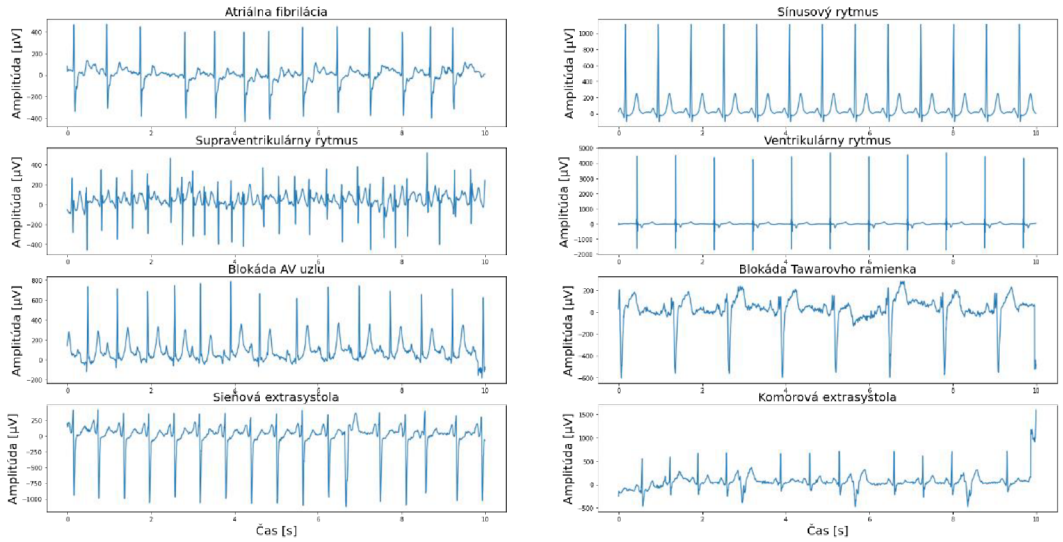
Tabuľka 3.1 : Databáza arytmických EKG signálov pre trénovanie GAN

Typ skupiny signálov	Počet multi-label signálov	Počet single-label signálov
Atriálna fibrilácia	553	267
Sinusový rytmus	5942	3350
Supraventrikulárny rytmus	194	89
Ventrikulárny rytmus	102	69
Blokáda AV uzlu	250	0
Blokáda Tawarovho ramienka	434	0
Sieňová extrasystola	418	0
Komorová extrasystola	413	0

Jednotlivé skupiny signálov boli ďalej rozdelené na dve podskupiny a to na signály obsahujúce viacero anotácií (multi-label) a signály obsahujúce len jediný typ anotácie (single-label). Prvé z nich, obsahujú všetky signály, ktoré obsahovali anotáciu svojej skupiny, ale môžu obsahovať aj iné typy anotácií (šum, pohyb pacienta, iný typ arytmie). Naopak v druhej skupine sa vyskytujú len signály, ktoré obsahovali jedine anotáciu svojej skupiny. Ako príklad, sa pozrieme na atriálne fibrilácie – existuje 553

signálov obsahujúcich atriálnu fibriláciu ale len 267 z nich obsahuje samostatné (bez ostatných anotácií) atriálne fibrilácie.

Signály boli spracovávané v jazykovom prostredí Python. Importovanie signálov (vo formáte .mat) bolo realizované pomocou dostupnej knižnice *scipy.io* a po nahratí dát bola vytvorená databáza arytmických signálov pre trénovanie GAN (viď. Obr. 3.1).



Obr. 3.1: Ukážka signálov vytvorenej databázy EKG. Databáza obsahuje 8 skupín signálov (Tabuľka 3.1), z každej skupiny bol vykreslený práve jeden signál - názov skupiny je uvedený v titulke grafu. Zobrazené signály sú zo zvodu II.

3.4 Analýza použiteľnosti poskytnutých signálov

Po vytvorení databázy bolo potrebné zvoliť signály vhodné pre trénovanie GAN. Ako prvé sme sa pozreli na odborné články, ktoré sa zaoberali generovaním medicínskych signálov pomocou GAN. Pri študovaní týchto článkov, sme sa snažili nájsť spoločné znaky a odvodiť obecný prístup predspracovania EKG signálov.

3.4.1 Predspracovanie signálov publikovaných metód

V štúdií [7], bolo pre trénovanie GAN na generovanie EKG záznamov použitých 48 signálov EKG z databázy MIT-BIH. Každý zo signálov trval 30 minút a vzorkovacia frekvencia týchto signálov bola 360Hz. Pre ďalšie spracovanie bol použitý len zvod II a dĺžka generovaných signálov bola rozdelená na úseky o dĺžke 3120 vzoriek, čo odpovedá približne 8,67s záznamu. Celkovo bolo teda pre trénovanie GAN použitých približne 9970 signálov.

Druhá zo štúdií [23] využila pre svoj tréning databázu z jednotky intenzívnej starostlivosti (Philips eICU), ktorá obsahovala záznamy približne 17 000 pacientov. Nejednalo sa však o záznamy EKG, ale o štyri rôzne hodnoty monitorovaných životných funkcií – saturácia kyslíkom, pulz, dychová frekvencia a stredný arteriálny tlak. Na základe týchto parametrov, sa snažili vyvinúť varovný systém, ktorý by dokázal vopred predpovedať kritický stav pacienta.

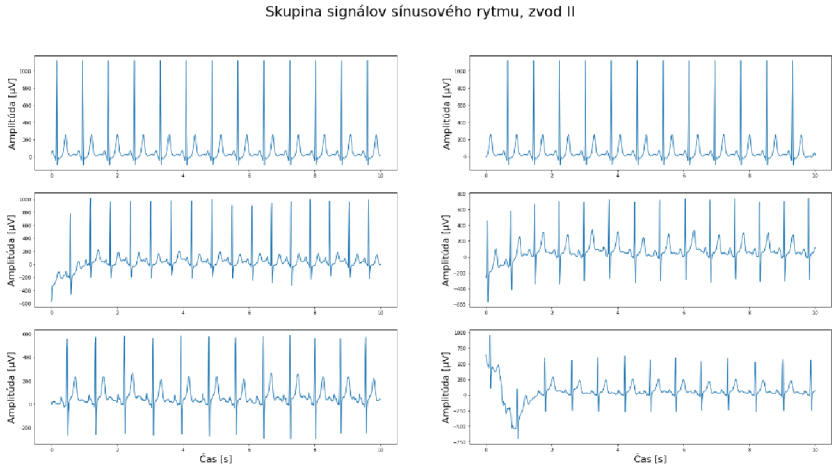
Ďalší z článkov [41] generoval syntetické EKG záznamy o dĺžke 187 vzoriek. Na tréning bola opäť použitá databáza MIT-BIH so signálmi o dĺžke 30 minút a vzorkovacou frekvenciou 360Hz. Signály boli následne prevzorkované na 125Hz a rozdelené na 10 sekundové úseky. Každý z úsekov bol normalizovaný samostatne. Následne boli signály rozdelené na úseky obsahujúce dve R vlny, každý úsek o dĺžke 187 vzoriek signálu. Nakoniec bolo 90 589 takýchto úsekov rozdelených na trénovaciu (72471) a testovaciu skupinu (18118).

V poslednej štúdií [42], boli použité experimentálne EKG signály. Jednalo sa o signály s vysokou vzorkovacou frekvenciou – 5000 Hz. Ďalej boli signály rozdelené na sekundové úseky, ktoré obsahovali jeden QRS komplex. Amplitúdy signálov boli transformované na šedotónové hodnoty v intervale <0, 255> a následne pretvorené na obrazy o veľkosti 64x64 pixelov. Jednalo sa teda o sekvencie o dĺžke 4096 vzoriek. Celkovo bolo pre tréning GAN použitých 4880 takýchto transformovaných signálov.

3.4.2 Predspracovanie poskytnutých signálov EKG

Vidíme, že prístupy a spôsoby predspracovania signálov odborných štúdií sa často líšia. Avšak, každa z štúdií používa pre trénovanie GAN len jeden zvod EKG. Preto sme sa rozhodli v našej práci ďalej pracovať už len so zvodom II.

Ďalším krokom bol výber skupiny signálov, ktorú sa budeme snažiť generovať. Vidíme, že žiadny z článkov, ktoré generovali EKG signály sa nezameriaval na konkrétny typ arytmie. Ďalej pozorujeme, že každý z článkov potreboval pre tréning dostatočne veľkú databázu signálov. Z tohto dôvodu, sme sa rozhodli v práci ďalej pracovať už len so signálmi sínusového rytmu. Týchto signálov máme najviac (viď. Tabuľka 3.1), a považujeme ich teda za najvhodnejšiu skupinu signálov pre tréning GAN. Na obrázku 3.2 môžeme vidieť ukážku prvých šiestich EKG signálov zo skupiny sínusového rytmu. Rozhodli sme sa pracovať so skupinou single-label signálov, ktorá obsahovala 3350 signálov EKG, nakoľko skupina multi-label signálov mohla obsahovať aj iné typy anotácií.



Obr. 3.2: Ukážka prvých šiestich EKG signálov zo skupiny sínusového rytmu.

4 NÁVRH A IMPLEMENTÁCIA MODELOV GAN

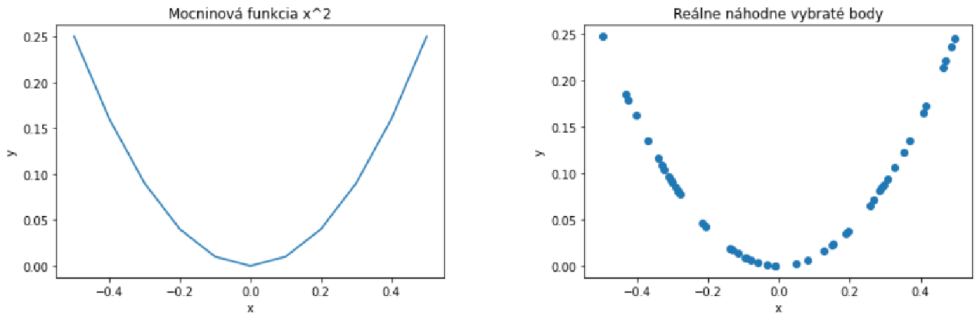
Pred tým ako sa pokúsime generovať reálne vyzerajúce EKG signály pomocou GAN, sme sa najprv rozhodli otestovať funkcionalitu GAN modelov pre jednodimenzionálne dáta. Ako sme spomenuli v podkapitole 2.3.3, väčšina GAN modelov sa zaoberá generovaním obrazových (2D) dát. Práve z tohto dôvodu, sme si najprv otestovali správanie GAN modelov na jednoduchých 1D dátach.

Všetky použité modely boli navrhované v programovom prostredí Python 3.6.9, pomocou služby Google Colab (Jupyter notebook). Pre implementáciu modelov bola využitá knižnica hlbokého učenia Keras s verziou 2.3.1.

4.1 GAN pre generovanie mocninovej funkcie x^2

Ako prvé, sme sa rozhodli generovať jednoduchú 1D funkciu, podobne ako v [49]. Modely GAN sú veľmi komplexné a práve z tohto dôvodu, sme zvolili zo začiatku veľmi jednoduchú úlohu, na ktorej si dobre vysvetlíme ako GAN fungujú. Výsledky generovaných aj vzorových dát sú ľahko zobraziteľné a jednoduchá 1D funkcia nevyžaduje komplikovanú neurónovú sieť na jej realizáciu.

Konkrétne budeme generovať kvadratickú funkciu $f(x)=x^2$, ktorú môžeme vidieť na Obrázku 4.1. Vstupy do tejto funkcie boli vyberané z intervalu <-0.5, 0.5>.



Obr. 4.1 : Ukážka mocninovej funkcie x^2 (vľavo) a 50 náhodne dosadených bodov do tejto funkcie (vpravo). Naším cieľom je vytvoriť model GAN schopný generovať podobné výstupy.

Ako sme si popísali v podkapitole 2.2.4, na vytvorenie GAN potrebujeme zostaviť model pozostávajúci z generátoru a diskriminátoru. Následne tento model natrénovať a nakoniec generovať požadované výstupy.

4.1.1 Zostavenie diskriminátoru

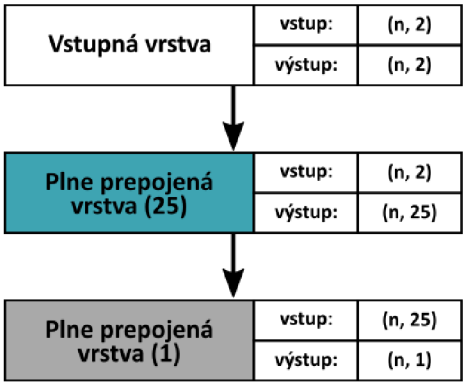
Vstupom do diskriminátoru sú dva vektory vstupných hodnôt (súradnice bodov) o dĺžke *n*. Model obsahuje jednu plne prepojenú skrytú vrstvu s 25 neurónmi a používa He inicializáciu váh, ktorú je vhodné kombinovať spolu s ReLu aktivačnou funkciou (podkapitola 1.5.1). Vo výstupnej vrstve je realizovaná binárna klasifikácia pomocou

sigmoidálnej AF – pretože na výstupe chceme dostať pravdepodobnosť zaradenia do jednej z tried. Ako chybová funkcia bola použitá binárna krížová entropia, ktorá sa využíva vo všetkých modeloch GAN (viď. podkapitola 2.2.7) a ako učebný algoritmus bol zvolený ADAM s predvolenými hodnotami parametru rýchlosti učenia η a parametru zabúdania *beta_1* (η*=0.001 / beta_1=0.9)* – rovnako ako v [49]. Nakoniec bola ešte zvolená metrika – presnosť, podľa ktorej môžeme hodnotiť správanie diskriminátoru.

Zostavenie diskriminátoru je v kóde realizované pomocou vytvorenej funkcie *define_discriminator*, ktorá je znázornená na Obrázku 4.2 spolu s blokovým diagramom architektúry navrhnutého diskriminátoru.

```
def define_discriminator(n_inputs=2):
  discriminator_model = Sequential()
  discriminator_model.add(Dense(25, activation='relu', kernel_initializer='he_uniform', input_dim=n_inputs))
  discriminator_model.add(Dense(1, activation='sigmoid'))
  discriminator_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return discriminator_model

# vytvorenie modelu
discriminator_model = define_discriminator()
```



Obr. 4.2: Implementácia (hore) architektúry (dole) diskriminátoru pre generovanie mocninovej funkcie x^2. V pravej časti každého z troch blokov, ktoré znázorňujú jednotlivé vrstvy diskriminátoru, je uvedený rozmer vstupných aj výstupných dát. Počet neurónov vrstvy je uvedený v zátvorke za jej názvom.

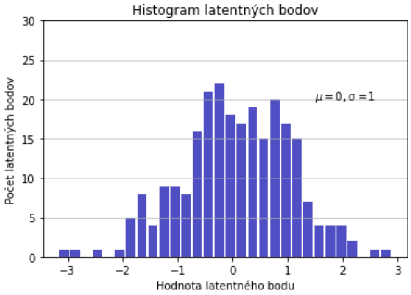
4.1.2 Zostavenie generátoru

Vstupom do generátoru sú náhodne vygenerované vzorky z latentného priestoru s veľkosťou dimenzie *d*. Vygenerované latentné body majú teda veľkosť $d \times n$, kde n odpovedá počtu generovaných bodov. Na Obrázku 4.3 sme vygenerovali 50 bodov v 5 dimenziách, tieto body sú náhodne vyberané z Gaussovského rozloženia (μ=0, σ=1).

Nakoniec boli vygenerované body ešte transformované na rozmer $n \times d$, aby zodpovedali požiadavkám pre rozmer vstupných dát generátoru. Pri každej predikcií generátoru sú najprv vygenerované latentné body, ktoré sú následne transformované na cieľové výstupné dáta. Generovanie latentných bodov zabezpečuje funkcia *generate_latent_points* (Obr. 4.3).

```python
def generate_latent_points(latent_dim, n):
  x_input = randn(latent_dim * n)
  x_input = x_input.reshape(n, latent_dim)
  return x_input

latent_points=generate_latent_points(5,50)
```

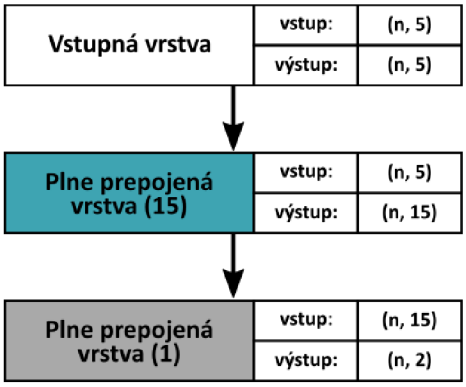


Obr. 4.3: Funkcia na generovanie latentných bodov (hore) a histogram 250 bodov (50x5) latentného priestoru (dole). Vidíme, že vygenerované latentné body zodpovedajú krivke Gaussovského rozloženia.

Generátor [49], obsahuje jednu plne prepojenú vrstvu s 15 neurónmi, používa He inicializáciu váh a ReLU aktivačnú funkciu, z rovnakého dôvodu ako pri zostavovaní diskriminátoru. Výstupom sú súradnice bodov (x, y) – dva samostatné vektory a využíva sa lineárna aktivačná funkcia. Lineárna aktivačná funkcia bola použitá, z dôvodu, že očakávame reálne výstupné hodnoty generátoru v intervale <-0.5, 0.5> pre súradnicu x a v intervale <0, 0.25> pre súradnicu y. Generátor je zostavené pomocou funkcie *define_generator,* ako môžeme vidieť na Obrázku 4.4.

```python
def define_generator(latent_dim, n_outputs=2):
  generator_model = Sequential()
  generator_model.add(Dense(15, activation='relu', kernel_initializer='he_uniform', input_dim=latent_dim))
  generator_model.add(Dense(n_outputs, activation='linear'))
  return generator_model

# velkosť latentného priestoru
latent_dim = 5
# vytvorenie generátoru
generator_model = define_generator(latent_dim)
```



Obr. 4.4: Implementácia (hore) architektúry (dole) generátoru pre generovanie mocninovej funkcie x^2. V pravej časti každého z troch blokov, ktoré znázorňujú jednotlivé vrstvy generátoru, je uvedený rozmer vstupných aj výstupných dát. Počet neurónov vrstvy je uvedený v zátvorke za jej názvom.

4.1.3 Zostavenie submodelu pre trénovanie generátoru

Pre zostavenie finálneho modelu GAN, je ešte potreba vytvoriť submodel pomocou, ktorého budeme trénovať generátor. Trénovanie generátoru je odlišné od trénovania

diskriminátoru – generátor využíva pre svoj tréning váhy diskriminátora (viď. Obr. 2.9). Submodel je vytvorený pomocou funkcie *define_gan* (viď. Obr. 4.5).

```
def define_gan(generator_model, discriminator_model):
  # zmrazenie váh diskriminátoru
  discriminator_model.trainable = False
  # spojenie generátoru a diskriminátoru
  gan_model = Sequential()
  gan_model.add(generator_model)
  gan_model.add(discriminator_model)
  gan_model.compile(loss='binary_crossentropy', optimizer='adam')
  return gan_model

# vytvorenie submodelu pre trénovanie generátoru
gan_model = define_gan(generator_model, discriminator_model)
```

Obr. 4.5: Funkcia na vytvorenie submodelu GAN pre trénovanie generátoru.

Ako chybová funkcia bola použitá binárna krížová entropia a ako učebný algoritmus bol zvolený ADAM s predvolenými hodnotami parametru rýchlosti učenia η a parametru zabúdania *beta_1* (η*=0.001 / beta_1=0.9)* – rovnako ako [49].

4.1.4 Trénovanie modelu GAN

Trénovanie modelu GAN začína nastavením trénovacích parametrov:

- Dimenzia latentného priestoru *latent_dim=5*, rovnako ako v [7, 49]
- Počet iterácií *n_iter* bol experimentálne stanovený na 100 000
- Veľkosť trénovacej skupiny (*batch_size)* bola stanovená na hodnotu 512. Vyššia hodnota bola vybratá z dôvodu urýchlenia tréningu GAN.
- Krok evaluácie modelu *n_eval* bol nastavený na 500 – každú 500.tú iteráciu dostávame vizuálne vyhodnotenie modelu.

Tréning modelu prebieha v dvoch striedavých krokoch. Väčšina prístupov strojového učenia zvyčajne UNS trénuje iba raz – spoločne s falošnými aj reálnymi dátami, ktoré ešte zvyknú byť náhodne zamiešané. My sme sa rozhodli použiť tip pre stabilné trénovanie GAN (popísaný v podkapitole 2.3.3) – osobité trénovanie reálnych a falošných dát. Použili sme identické chybové funkcie a optimalizačné algoritmy pre generátor aj diskriminátor, avšak toto nemusí byť pravidlom. Ladenie modelov GAN podľa pomeru frekvencie trénovania jednotlivých modelov sa neodporúča. [48]

Prvým z krokov je tréning diskriminátoru. V každej iterácií, sa náhodne vygeneruje polovica počtu trénovacej skupiny (512/2=256) reálnych (*x_real*) aj generovaných (*x_fake*) dát spolu s príslušnými označeniami skupiny (*y_real, y*_fake) pomocou funkcií *generate_real_samples* a *generate_fake_samples* (Obr. 4.6). Reálne (trénovacie) dáta sú v tomto prípade náhodne vybraté čísla v intervale <-0.5, 0.5> pre vektor *x* a vo vektore *y* sú tieto čísla umocnené na druhú. Dostávame teda dva vektory

o rovnakej dĺžke, ktoré ďalej vstupujú do diskriminátoru. Diskriminátor je trénovaný dva krát – osobite na obidvoch skupinách.

```python
def generate_fake_samples(generator, latent_dim, n):
  # generovanie latentných bodov
  x_input = generate_latent_points(latent_dim, n)
  # predikcia generátoru
  X = generator.predict(x_input)
  # označenie skupiny
  y = zeros((n, 1))
  return X, y
```

```python
def generate_real_samples(n):
  # generovanie náhodných čísel v intervale [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # umocnenie
  X2 = X1 * X1
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # označenie skupiny
  y = ones((n, 1))
  return X, y
```

Obr. 4.6: Funkcie pre generovanie falošných (vľavo) a reálnych (vpravo) dát pre tréning modelu GAN.

Druhý krok začína vygenerovaním latentných bodov (ktoré sú vstupom pre generátor) pomocou funkcie *generate_latent_points*. Generátor trénujeme použitím zmrazených váh aktuálneho modelu diskriminátoru (generátor je prepojený s diskriminátorom – submodel v podkapitole 4.1.3). Pri trénovaní diskriminátoru, sú prichádzajúce vstupy označené príslušnou triedou (0 – falošné, 1 – reálne). Pri trénovaní generátoru avšak používame trik, kedy prevrátime označenie triedy falošných dát na reálne – premenná *y_gan*. Tento prístup má stabilnejšie a lepšie výsledky (podkapitola 2.3.3) a hovoríme o použití heuristickej nesaturačnej hry pri tréningu GAN. Generátor je počas jednej iterácie trénovaný iba raz.

```python
def train(generator_model, discriminator_model, gan_model, latent_dim=5, n_iter=100000, batch_size=512,n_eval=500):
  # polovica veľkosti skupiny
  half_batch = int(batch_size / 2)
  for i in range(n_iter):
    # vygenerovanie reálnych vzoriek
    x_real, y_real = generate_real_samples(half_batch)
    # vygenerovanie falošných vzoriek
    x_fake, y_fake = generate_fake_samples(generator_model, latent_dim, half_batch)
    # tréning diskriminátoru
    d_real_loss,d_real_acc=discriminator_model.train_on_batch(x_real, y_real)
    d_fake_loss,d_fake_acc=discriminator_model.train_on_batch(x_fake, y_fake)
    # generovanie latentných bodov
    x_gan = generate_latent_points(latent_dim, batch_size)
    # prevrátenie označenia triedy falošných vzoriek na reálne
    y_gan = ones((batch_size, 1))
    # trénovanie generátoru
    g_loss=gan_model.train_on_batch(x_gan, y_gan)
    # uloženie presností a chýb oboch modelov
    d_real_loss_graph.append(d_real_loss)
    d_real_acc_graph.append(d_real_acc)
    d_fake_loss_graph.append(d_fake_loss)
    d_fake_acc_graph.append(d_fake_acc)
    g_loss_graph.append(g_loss)
  # krok vizuálnej evaluácie modelu
    if (i+1) % n_eval == 0:
      summarize_performance(i, generator_model, discriminator_model, latent_dim)

train(generator, discriminator, gan_model)
```

Obr. 4.7: Implementácia trénovacieho algoritmu GAN.

Na konci každej iterácie si ešte uložíme chyby diskriminátoru na reálnych aj falošných dátach do premenných *d_real_loss_graph* a *d_fake_loss_graph*, spolu s aktuálnou metrikou presnosti diskriminátoru *d_real_acc_graph* a *d_fake_acc_graph*.

Chyba generátoru bola uložená v premennej *g_loss_graph*. Tieto premenné budú neskôr použité pre vykreslenie grafov a následnú analýzu modelu GAN v ďalších kapitolách.

4.2 GAN pre generovanie funkcie sínus

Druhý z modelov bol veľmi podobný prvému modelu. Tentoraz sme sa snažili generovať funkciu sínus. Generovanie tejto funkcie je o niečo náročnejšie ale princíp zostáva rovnaký. Architektúru druhého modelu GAN sme sa pokúsili vylepšiť – nakoľko sa jedná o komplexnejší problém.

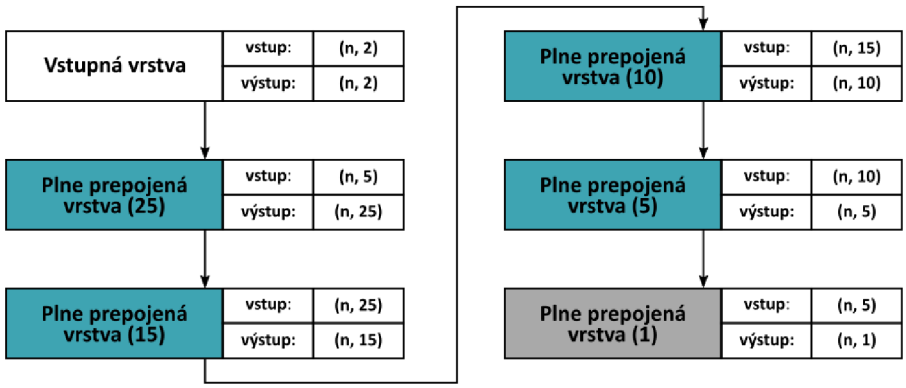
Celkovo budeme GAN trénovať na dvoch rôznych skupinách reálnych dát. Prvá skupina obsahuje funkcie sínus s rovnakou frekvenciou aj amplitúdou. V druhej skupine dát sme trénovaciu skupinu zľahka augmentovali – amplitúdy jednotlivých sínusoviek sa mohli meniť. Týmto spôsobom môžeme lepšie analyzovať a porovnať jednotlivé prístupy – čo nám pomôže pochopiť správanie GAN pri komplexnejších úlohách.

4.2.1 Zostavenie diskriminátoru

Tentoraz sme zvolili viac skrytých vrstiev v diskriminátore, ktoré by mali byť schopné lepšie zachytiť charakter vstupných dát.

```python
def define_discriminator(n_inputs=2):
    model = Sequential()
    model.add(Dense(25,  kernel_initializer='he_uniform', input_dim=n_inputs))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(15))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(10))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(5))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

discriminator2 = define_discriminator()
```



Obr. 4.9: Implementácia (hore) architektúry diskriminátoru (dole) pre generovanie funkcie sínus. V pravej časti každého z šiestich blokov, ktoré znázorňujú jednotlivé vrstvy diskriminátoru, je uvedený rozmer vstupných aj výstupných dát. Počet neurónov vrstvy je uvedený v zátvorke za jej názvom.

Diskrimininátor pozostával z celkovo 4 plne prepojených skrytých vrstiev (25,15,10 a 5 neurónov) využíval rovnakú inicializáciu váh (He) a namiesto AF ReLU bola použitá AF Leaky ReLU. Toto vylepšenie by malo sieti pomôcť predchádzať problému miznúceho gradientu. Parameter *alpha* v AF LeakyReLU vyjadruje sklon tejto funkcie (viď. Obr. 1.9). Vytvorenie diskriminátoru je realizované pomocou funkcie *define_discriminator* (viď. Obr. 4.9). [48]

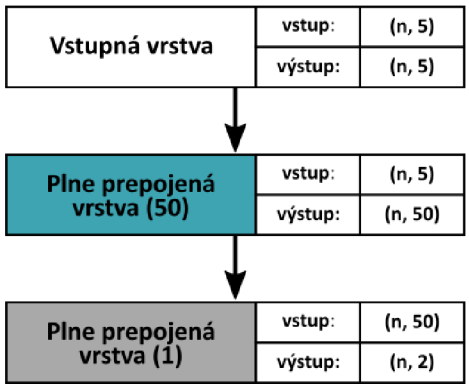
Aktivačná funkcia vo výstupnej vrstve diskriminátoru zostala rovnaká – sigmoidálna AF. Ako chybovú funkciu sme opäť použili binárnu krížovú entropiu a parametre optimalizačného algoritmu ADAM sme ponechali taktiež nezmenené – rovnako ako v podkapitole 4.1. [49]

4.2.2 Zostavenie generátoru

Vstupom do generátoru sú latentné body – ich generovanie bolo zabezpečné funkciou *generate_latent_points* (popísanou v podkapitole 4.1.2). Veľkosť dimenzie latentného priestoru sme ponechali rovnakú ako v [49] *latent_dim=5*, jedinou zmenou v generátore bolo zvýšenie počtu neurónov použitých v skrytej vrstve na 50 – nakoľko očakávame, že generovanie funkcie sínus bude komplexnejší problém (viď. podkapitola 1.3).

Generátor znova používa He inicializáciu váh, ktorú je vhodné kombinovať spolu s ReLu aktivačnou funkciou (podkapitola 1.5.1). Lineárna aktivačná funkcia bola použitá, pretože očakávame opäť reálne výstupné hodnoty generátoru v určitom intervale. Architektúra generátoru druhého modelu spolu s funkciou *define_generator* je zobrazená na Obrázku 4.10.

```
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(50, activation='relu', kernel_initializer='he_uniform', input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
  return model
```



Obr. 4.10: Architektúra generátoru pre generovanie funkcie sínus. V pravej časti každého z troch blokov, ktoré znázorňujú jednotlivé vrstvy generátoru, je uvedený rozmer vstupných aj výstupných dát. Počet neurónov vrstvy je uvedený v zátvorke za jej názvom.

4.2.3 Zostavenie submodelu pre trénovanie generátoru

Submodel pre trénovanie generátoru druhého modelu bol zostavený rovnako ako v podkapitole 4.1.3 (viď. Obrázok 4.5) – použitím funkcie *define_gan*. Chybová funkcia zostala nezmenená – binárna krížová entropia, a ako učebný algoritmus bol nastavený ADAM s predvolenými hodnotami parametru rýchlosti učenia η a parametru zabúdania *beta_1* (η*=0.001 / beta_1=0.9)*. [49]

4.2.4 Trénovanie modelu GAN

Ako bolo spomenuté v podkapitole 4.2, model bol trénovaný na dvoch rôznych trénovacích skupinách. Prvá zo skupín, obsahovala funkcie sínus, ktoré mali rovnakú amplitúdu aj frekvenciu. Naopak v druhej skupine, bolo pre každú z reálnych vzoriek vygenerované náhodné číslo v intervale <1, 2>, ktorým bola funkcia sínus vynásobená. Týmto spôsobom sme augmentovali trénovacie dáta na funkcie sínus s variabilnou amplitúdou.

```
def generate_real_samples(n):
  X1 = np.linspace(0,10,n)
  # sinus
  X2 = np.sin(X1)
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # označenie skupiny
  y = ones((n, 1))
  return X, y
```

```
def generate_real_samples2(n):
  X1 = np.linspace(0,10,n)
  # sinus s premenlivou amplitúdou
  X2 = (np.random.random(1)+1)*np.sin(X1)
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # označenie skupiny
  y = ones((n, 1))
  return X, y
```

Obr. 4.11: Funkcie pre generovanie reálnych sínusových signálov s nemennou amplitúdou (vľavo) a reálnych sínusových signálov s premenlivou amplitúdou (vpravo) pre tréning modelu GAN.

Našim predpokladom bolo, že generovať funkciu sínus bude náročnejšie, a preto bolo v tréningu druhého modelu zmenených niekoľko parametrov:

- Počet iterácií *n_iter* bol zvýšený na 500 000 (pre prvú skupinu) a 1 000 000 (pre druhú skupinu s premenlivou amplitúdou funkcií sínus).
- Krok evaluácie modelu *n_eval* bol nastavený na 5 000 (pre prvú skupinu) a 10 000 (pre druhú skupinu) – vizuálne vyhodnotenie modelu, tentoraz nebudeme potrebovať tak často – model sa učí pomalšie.
- Dimenzia latentného priestoru *latent_dim=5*, zostala rovnaká ako v [7, 49].
- Veľkosť trénovacej skupiny (*batch_size)* bola ponechaná na hodnotu 512. Táto hodnota nám prišla v predchádzajúcom modeli ako dostatočne veľká.

Spôsob trénovania GAN bol podrobne popísaný v podkapitole 4.1.4. Implementácia je aj v druhom modeli rovnaká a nebudeme ju znova popisovať – jediným rozdielom je spôsob generovania reálnych dát, ktoré zabezpečujú funkcie zobrazené na Obrázku 4.11.

4.3 GAN pre generovanie signálov EKG

Generovanie EKG signálov, je pomerne nová oblasť použitia GAN – podarilo sa nám nájsť celkovo 4 odborné články, ktoré sa zaoberali touto tématikou [7, 23, 41, 42]. Všetky články generovali sínusové EKG signály – okrem článku [23], ktorý sa snažil predikovať kritický stav pacientov na základe hodnôt monitorovaných životných funkcií (generoval predpokladaný vývoj stavu pacienta).

Po preštudovaní všetkých článkov, sme zistili, že iba články [7, 41] zverejnili použité architektúry a parametre trénovania. Preto sme sa rozhodli, začať s prvým z článkov a pokúsiť sa implementovať daný model GAN. Tento článok bol doporučený aj v zadaní tejto diplomovej práce [7].

4.3.1 Prepojenie s Google Drive

Konkrétne dáta (poskytnuté vedúcim tejto práce), na ktorých sme model trénovali nemôžu byť v tejto práci zverejnené. Napriek tomu, si popíšeme postup akým je akékoľvek dáta (ak by niekto chcel tento model implementovať s vlastnými dátami) možné nahrať na Google Disk.

Užívateľ musí byť prihlásený na svojom Google účte – prepojenie pracovného zošitu (Jupyter notebook) s diskom je realizované pomocou nasledovného bloku (viď. Obr. 4.12). Užívateľ musí špecifikovať názov pracovného priečienku, s ktorým bude ďalej pracovať v premennej *WORKDIR*.

```
from google.colab import drive
drive.mount('/content/gdrive')

GOOGLE_DRIVE_PATH = Path('/content/gdrive/My Drive')
WORKDIR = 'ECG_DATA' # nazov pracovneho priecinku
WORKDIR_PATH = GOOGLE_DRIVE_PATH / WORKDIR

if not os.path.exists(WORKDIR_PATH):
        os.mkdir(WORKDIR_PATH)
```

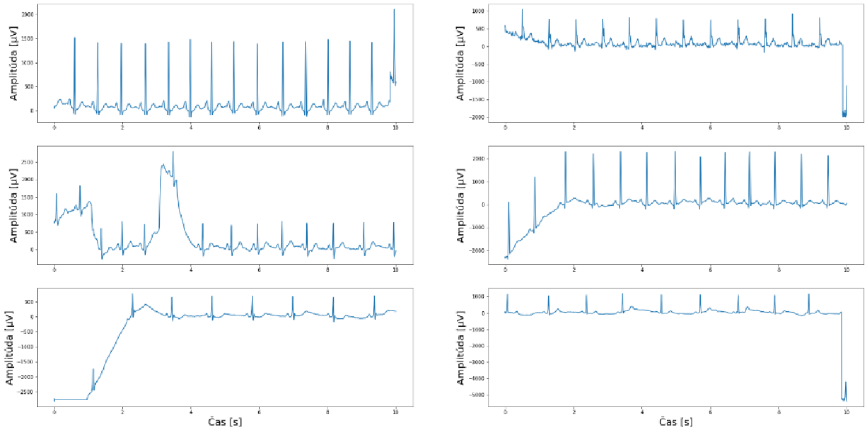
Obr. 4.12: Nastavenie prepojenia služby Google Disk s pracovným zošitom (Jupyter notebook).

4.3.2 Predspracovanie signálov

Pred tým, ako zostavíme model GAN a začneme ho trénovať – musíme predspracovať použité signály. Správne vstupné dáta sú veľmi dôležité, pre správnu funkciu neurónových sietí. Ako sme popísali v podkapitole 3.4.2 – v tejto práci bol použitý zvod II a na testovanie modelu bola zvolená skupina signálov obsahujúca sínusové signály.

Tie boli následne prevzorkované z pôvodných 500Hz na 360Hz – túto vzorkovaciu frekvenciu použili aj [7]. Podvzorkovanie signálu bolo realizované pomocou funkcie *scipy.signal.resample*, ktorá využíva Fourierovú transformáciu. Počet vzoriek nových 10 sekundových signálov sa teda zmenil z pôvodných 5000 na 3600. V práci [7], generovali signály o dĺžke 3120 vzoriek – práve z tohto dôvodu sme sa rozhodili ešte signály skrátiť (prvých 480 vzoriek bolo odstránených).

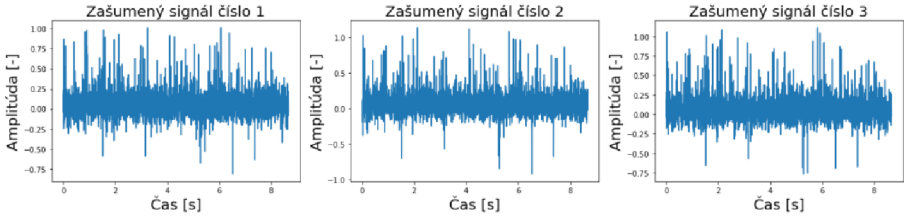
Druhým krokom bolo odstránenie nevhodných signálov. Všetky signály, ktorých amplitúda, sa nenechádzala v intervale <-1500μV, 2000μV> boli odstránené. Celkový počet týchto signálov bol 55 a ukážku takýchto signálov vidíme na Obr. 4.13. Skupina sínusových signálov po tomto kroku obsahovala už len 3295 signálov.



Obr. 4.13: Ukážka odstránených (nevhodných) sínusových signálov.

Ďalším krokom bolo orezanie signálov – tento krok bol vykonaný z dôvodu, centrovania signálov okolo 0. Všetky body signálov, ktorých amplitúda prekročila jeden z limitov intervalu <-1200μV, 1200μV> boli nahradené hraničnou hodnotou spomenutého intervalu. Vďaka orezaniu, sme signály jednoducho transformovali na rozsah hodnôt <-1, 1>, podelením každého signálu hodnotou 1200. Táto transformácia bola dôležitá, nakoľko budeme v generátore modelu GAN používať AF hyperbolického tangensu (viď. podkapitola 2.3.3 – normalizácia vstupných dát).

Posledným krokom bolo vytvorenie zašumených signálov, ktoré využijeme neskôr, pri testovaní diskriminátoru. Takto vytvorené signály, boli zašumené Gaussovským šumom s parametrami μ=0, σ=0.1. Šum bol postupne pričítaný ku každému zo signálov. Ukážku zašumených signálov môžeme vidieť na Obrázku 4.14.

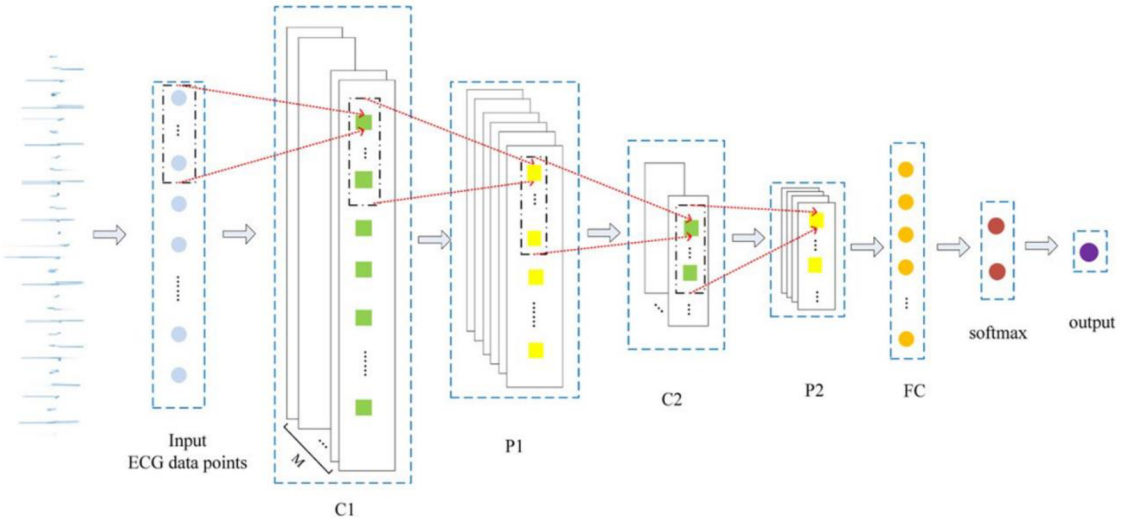


Obr. 4.14: Ukážka zašumených signálov pre testovanie diskriminátoru.

4.3.3 Zostavanie diskriminátoru

Architektúra diskriminátoru bola prevzatá z článku [7]. Do diskriminátoru vstupuje EKG signál o dĺžke 3120 vzoriek, nasledujú dvojice konvolučných 1D vrstiev a MaxPoolingových vrstiev (popísané v podkapitole 1.6). Diskriminátor používa He inicializáciu váh, pretože používame Leaky ReLU AF (podkapitola 1.5.1).

Po dvoch takýchto dvojiciach, nasledovala plne prepojená vrstva s 25 neurónmi. Pred použtím tejto vrstvy sme ešte použili Flatten vrstvu, popísanú v Obrázku 1.13. Vo výstupnej vrstve je realizovaná binárna klasifikácia pomocou sigmoidálnej AF – pretože na výstupe chceme dostať pravdepodobnosť zaradenia do jednej z tried. V tomto modeli bola vyskúšaná aj softmax AF – ktorú používali v [7]. Táto AF avšak v našej implementácií nedosahovala dobrých výsledkov, a rozhodli sme sa ďalej používať už len sigmoidálnu AF, ktorá sa osvedčila aj v modeloch pre generovanie 1D signálov.




```
def define_discriminator(input_shape=(3120,1)):
  model = Sequential()
  model.add(Conv1D(input_shape=(3120,1),filters=10, kernel_size=120, strides=5, padding='valid',kernel_initializer='he_normal'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(MaxPooling1D(pool_size=46, strides=3, padding='valid', data_format='channels_last'))
  model.add(Conv1D(filters=5, kernel_size=36, strides=3, padding='valid'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(MaxPooling1D(pool_size=24, strides=3, padding='valid', data_format='channels_last'))
  model.add(Flatten())
  model.add(Dense(25))
  model.add(Dense(1,activation='sigmoid'))
  # optimalizacny algoritmus
  #adam_D = Adam(lr=0.0002, beta_1=0.5)
  #sgd_D = SGD(lr=0.001)
  #sgd_D = SGD(lr=0.0001)
  sgd_D = SGD(lr=0.00001)
  model.compile(loss='binary_crossentropy', optimizer=sgd_D, metrics=['accuracy'])
  return model

# zostavenie diskriminátoru
discriminator_model = define_discriminator()
```

Obr. 4.15: Ukážka implementácie (dole) prevzatej architektúry (hore) z odborného článku [7].

Porovnanie architektúry s našou implementáciou pomocou funkcie *define_discriminator* vidíme na Obr. 4.15. Architektúru sme ešte doplnili o vrstvy AF LeakyReLU, ktoré by mali pomôcť s problémom miznúceho gradientu (podkapitola 2.3.3 – použitie Leaky ReLU AF). Táto vrstva taktiež dosahovala dobrých výsledkov pri generovaní sínusových signálov v podkapitole 4.2. Parameter *alpha* vyjadruje sklon Leaky ReLU AF, zvolili sme hodnotu 0.2, ktorá bola experimentálne stanovená.

Parametre konvolučných a poolingových vrstiev boli prevzaté z [7], počet filtrov určuje parameter *filters*, veľkosť posuvných okien parametre *kernel_size* a *pool_size*, krok posuvného okna definuje parameter *strides*. Ako posledné bolo potrebné nastaviť typ optimalizačného algoritmu – v práci [7] používali SGD s paramatrom rýchlosti učenia o veľkosti $1x10^{-5}$. Táto práca otestovala niekoľko optimalizačných algoritmov s rôznymi rýchlosťami učenia, ktoré sú popísané v ďalších kapitolách práce. Metrika vyhodnocovania diskriminátoru bola nastavená na presnosť a chybová funkcia bola binárna krížová entropia.

4.3.4 Zostavenie generátoru

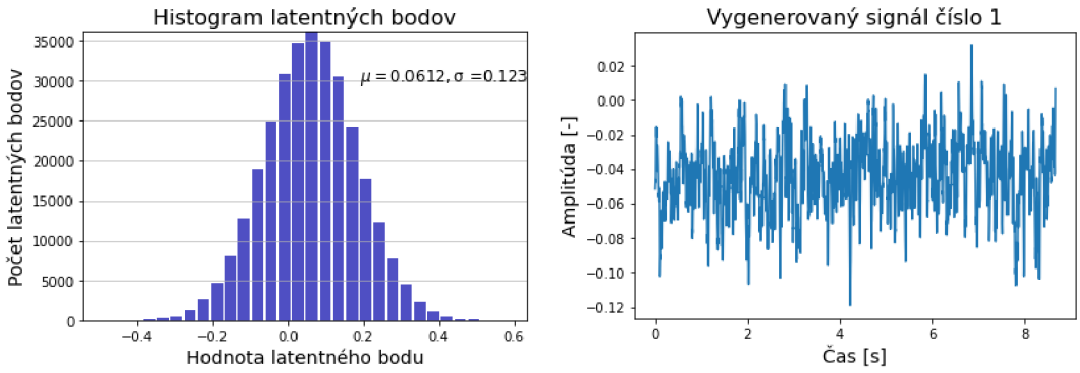
Vstupom do generátoru sú náhodne vygenerované vzorky z latentného priestoru o rozmeroch *batch_size*×*length_of_signal*×*latent_dim*. Parameter *batch_size* udáva veľkosť skupiny (počet signálov), parameter *length_of_signal* udáva dĺžku signálu a parameter *latent_dim* určuje veľkosť dimenzie latentného priestoru.

Generovanie latentných bodov zabezpečuje funkcia *generate_latent_points* (Obr. 4.16), ktorá generuje latentné body, vhodné pre vstup generátoru, pomocou Gaussovského rozloženia s parametrami *mu* a *sigma*. Tieto parametre aproximujú rozloženie nami spracovávaných signálov – parameter *mu* odpovedá priemernej hodnote databázy signálov (izolínia signálu) a parameter *sigma* odpovedá smerodajnej odchýlke týchto signálov. Týmto spôsobom generátoru čiastočne „pomôžeme“ – takto náhodne vybraté vzorky budú dobre popisovať vlastnosti trénovacích signálov (nejedná sa teda o náhodný šum).

Ukážku vygenerovania jedného signálu môžeme vidieť na Obrázku 4.17. Ako prvé si zistíme parametre *mu* a *sigma* z databázy trénovaích signálov – na Obrázku 4.17 vidíme, že *mu*=0.0612 a *sigma*=0.123. Keďže generujeme jeden signál, parameter *batch_size* má hodnotu 1 a veľkosť parametru latentnej dimenzie sme v tomto príklade nastavili na *latent_dim*=100. Ako bolo spomenuté v podkapitole 3.4.1, naše signály majú dĺžku 3120 vzoriek. Dostávame latentné body o rozmere 1×3120×100, celkovo teda hovoríme o 312000 bodoch. V histograme na Obrázku 4.17, vidíme, že vygenerované latentné body sú Gaussovsky rozložené – avšak s parametrami použitých signálov. Vygenerovaný signál samozrejme nepripomína EKG signál, nakoľko model generátoru ešte nebol natrénovaný.

```
def generate_latent_points(latent_dim,batch_size,mu,sigma):
  latent_points = np.random.normal(mu,sigma, size=(3120 * latent_dim * batch_size))
  latent_points=latent_points.reshape(batch_size,length_of_signal,latent_dim) #batch x 3120 x 5
  return latent_points
```

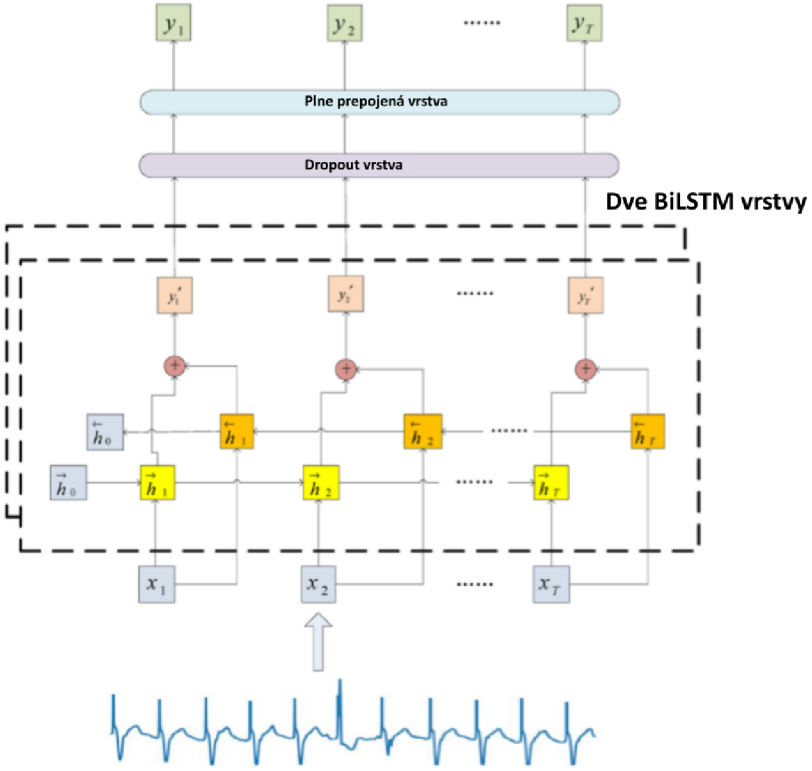
Obr. 4.16: Funkcia pre generovanie latentných bodov.



Obr. 4.17: Ukážka generovania signálu z latentných bodov

Model generátoru pozostáva z dvoch BiLSTM vrstiev [7], ktoré by mali dobre zachytávať krátkodobé vlastnosti (závislosti) spracovávaných signálov. Narozdiel od klasických LSTM vrstiev, ktoré sme si popísali v podkapitole 1.6.4, BiLSTM dokážu lepšie zachytiť kontext dát – výstup závisí na minulých aj budúcich stavoch bunky. Zjednodušene môžeme povedať, že LSTM sa snažia predikovať nasledujúci prvok na základe informácií z „minulosti“ na rozdiel od BiLSTM vrstiev, ktoré svoju predikciu zakladajú na „minulosti“ aj „budúcnosti“. Obidve BiLSTM vrstvy obsahujú 100 neúronov [7] a v prvej vrstve sme sa rozhodli použiť Xavier inicializáciu váh – tú je vhodnú kombinovať s AF hyperbolického tangensu (podkapitola 1.5.1), ktorú v generátore používame. Veľmi dôležitým parametrom v BiLSTM vrstvách je spôsob spracovania (spojenia) minulých a budúcich stavov bunky – v tejto práci sú tieto stavy sčítané, rovnako ako v [7], čo nám udáva parameter *merge_mode*=‘sum‘. Posledným z parametrov je *return_sequences*, ktorý rozhoduje, či chceme na výstupe posledný stav bunky alebo všetky stavy bunky – keďže chceme vygenerovať celý signál, nie len jeho časť nastavili sme parameter na hodnotu *True*.

Architektúru generátora prevzatého z článku [7], vidíme na Obrázku 4.18. Po spomínaných BiLSTM vrstvách je pridaná Dropout vrstva (podkapitola 1.6.5), ktorá má zamedziť preučeniu siete. Jej hodnota bola nastavená na hodnotu 0.5 [7]. Poslednou vrstvou je plne prepojená vrstva s jedným neurónom, ktorá nám vygeneruje hodnoty výstupného signálu.




```
def define_generator(latent_dim):
  model = Sequential()
  model.add(Bidirectional(LSTM(100, return_sequences=True,init='glorot_normal',activation='tanh'),
                          input_shape=(3120,latent_dim), merge_mode='sum'))
  #model.add(LeakyReLU(alpha=0.2))
  model.add(Bidirectional(LSTM(100, return_sequences=True,activation='tanh'),merge_mode='sum'))
  #model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.5))
  model.add(Dense(1, activation='tanh'))
  return model

# veľkosť latentného priestoru
latent_dim =5
# zostavenie generátoru
generator_model = define_generator(latent_dim)
```

Obr. 4.18: Ukážka implementácie (dole) prevzatej architektúry (hore) z odborného článku [7].

4.3.5 Zostavenie submodelu pre trénovanie GAN

Trénovanie generátoru bolo realizované pomocou vytvoreného submodelu, ktorý spojuje generátor a diskriminátor. Generátor pre svoj tréning využíva zmrazené váhy diskriminátoru. Chybová funkcia spolu s typom optimalizačného algoritmu pre tréning generátora sa nastavuje práve v tomto submodeli, ktorý je vytvorený pomocou funkcie *define_gan*, ktorú môžeme vidieť na Obrázku 4.19.

```
def define_gan(generator, discriminator):
  # zmrazenie vah diskriminatoru
  discriminator.trainable = False
  # prepojenie modelov
  model = Sequential()
  model.add(generator)
  model.add(discriminator)
  # optimalizacny algoritmus generatoru
  #adam = Adam(lr=0.0002, beta_1=0.5)
  sgd = SGD(lr=0.001)
  #sgd = SGD(lr=0.0001)
  #sgd = SGD(lr=0.00001)
  model.compile(loss='binary_crossentropy', optimizer=sgd)
  return model

# vytvorenie submodelu
gan_model = define_gan(generator_model, discriminator_model)
```

Obr. 4.19: Funkcia na vytvorenie submodelu GAN pre trénovanie generátoru.

4.3.6 Tréning modelu GAN

Najzložitejšou časťou implementácie modelov GAN pre generovanie EKG signálov je algoritmus učenia GAN. Ten pozostáva z troch hlavných častí. Hovoríme o tréningu diskriminátoru, generátoru a nakoniec o uložení evaluácie výsledkov modelu.

Prvým krokom trénovania modelu GAN, je nastavenie parametrov trénovania spolu s pripravením vstupných dát – predspracovanie signálov bolo podrobne popísané v podkapitole 4.3.2. Po pripravení signálov, je vytvorená databáza trénovacích a testovacích signálov. Dostupných 3295 signálov sme rozdelili na 2636 testovacích a 659 trénovacích signálov (v pomere 80/20). Následne sú vypočítané parametre *mu* a *sigma* – ktoré charakterizujú štatistické rozloženie hodnôt signálu (podkapitola 4.3.4). Ďalej je potrebné nastaviť parametre učenia siete:

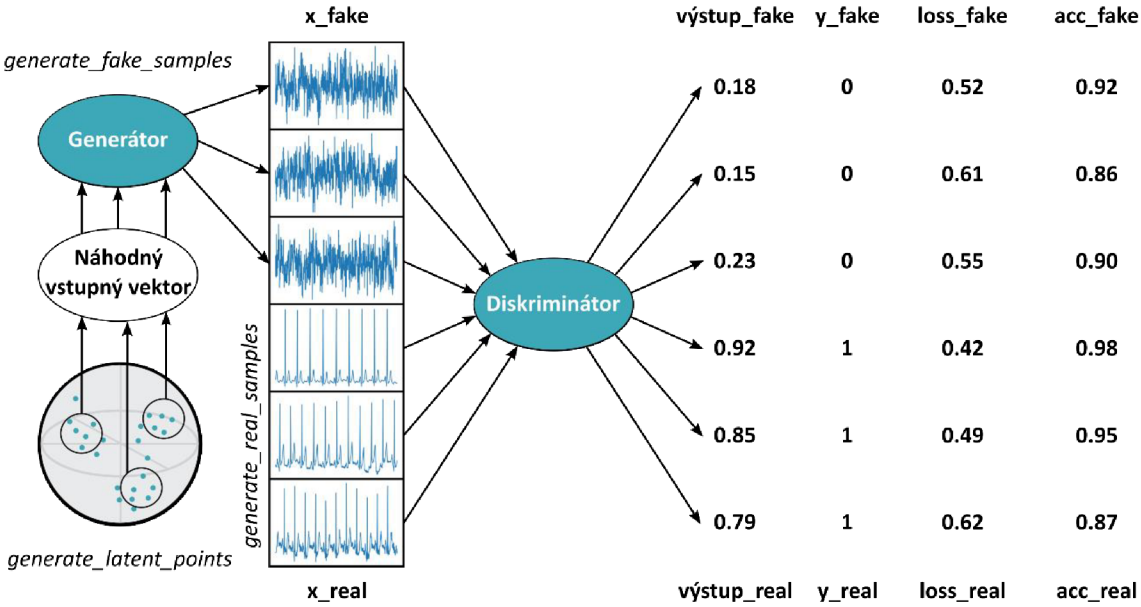
- Počet epôch trénovania – parameter *n_epochs*.
- Veľkosť trénovacej skupiny – parameter *batch_size*.
- Veľkosť dimenzie latentného priestoru – parameter *latent_dim*.
- Optimalizačný algoritmus diskriminátoru spolu s parametrom rýchlosti učenia
- Optimalizačný algoritmus generátoru spolu s parametrom rýchlosti učenia
- Krok evaluácie modelu – parameter *n_eval*.

Teraz, keď sme si pripravili databázu signálov a nastavili parametre trénovania GAN môžeme začať s tréningom modelu GAN. Trénovanie je realizované pomocou osobitného trénovania reálnych aj falošných signálov (podkapitola 2.3.3) a samotný algoritmus tréningu je principiálne podobný algoritmu použitému v generovaní 1D funkcií. Algoritmus sme samozrejme museli ale značne pomeniť a prispôsobiť generovaniu EKG signálov.

Tréning začína trénovaním diskriminátoru (viď. Obrázok 4.20) – pomocou funkcie *generate_real_samples* sa ako prvé vygenerujú reálne trénovacie signály z vytvorenej trénovacej databázy o spomenutej veľkosti (2636). Náhodne vyberieme

presne polovicu signálov z trénovacej skupiny (*batch_size)*. Vybraté signály uložíme do premennej *x_real* a označenia tejto skupiny (1) si vygenerujeme do premennej y_real. Diskriminátor na týchto dátach trénujeme prvýkrát – aktuálnu chybu a presnosť si ukladáme do premenných *loss_real* a *acc_real*.

Druhýkrát trénujeme diskriminátor na falošných (vygenerovaných) signáloch, ktoré si pripravíme spolu s príslušnými označeniami skupiny (0) pomocou funkcie *generate_fake_samples* do premenných *x_fake* a *y_fake*. Vstupom do tejto funkcie je aktuálny model generátoru – spôsom generovania dát je popísaný v podkapitole 4.3.4. Chybu a presnosť diskriminátoru na falošných dátach si znovu ukladáme do premenných *loss_fake* a *acc_fake*.

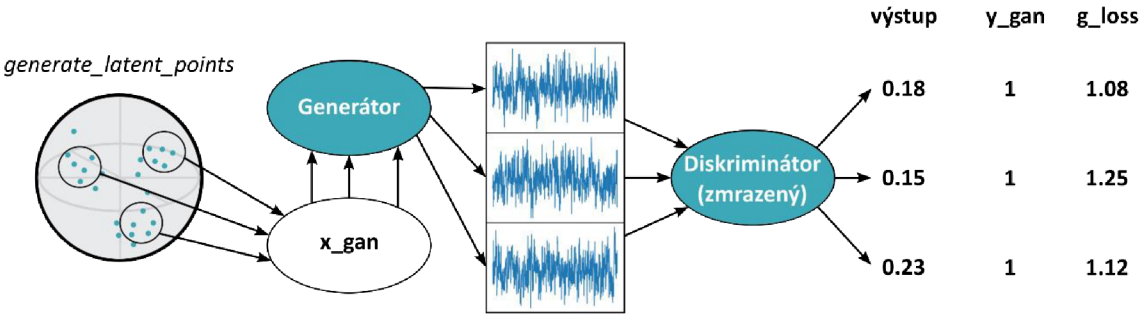


Obr. 4.20: Ukážka spôsobu osobitného trénovania diskriminátora na EKG signáloch. Všetky použité obrázky EKG signálov, sú z databázy signálov poskytnutých vedúcim tejto práce.

Tréning pokračuje vygenerovaním latentných bodov pomocou funkcie *generate_latent_points* do premennej *x_gan*. Tieto body sú následne vstupom pre generátor. Nasledujúci krok, je veľmi dôležitý – označenie skupiny pre latentné body, meníme na 1 a ukladáme do premennej *y_gan*. Týmto trikom, použijeme princíp heuristickej nesaturačnej hry pri tréningu GAN, ktorú sme popísali v podkapitole 2.2.7. Jej použitie stabilizuje tréning učenia GAN (podkapitoly 2.3.1 a 2.3.3) a predchádza problému miznúceho gradientu. Diskriminátor, samozrejme tieto vzorky odmietne s veľkou pravdepodobnosťou (určí ich za falošné). Proces spätného šírenia chyby to však „uvidí“ ako veľkú chybu a aktualizuje váhy omnoho viac – generátor bude lepší v generovaní (klamaní) hodnoverných dát.

Ako sme už niekoľkokrát spomenuli – trénovanie generátoru je odlišné od trénovania diskriminátoru. Spôsobom akým sa generátor trénuje vidíme na Obrázku 4.21, po vygenerovaní latentných bodov a signálov sa generátor aktualizuje pomocou

zmrazených váh diskriminátoru – ten sa v tento moment neučí. Chybu generátoru si ukladáme do premenej *g_loss*.



Obr. 4.21: Spôsob trénovania generátoru na EKG signáloch. Všetky použité obrázky EKG signálov, sú z databázy signálov poskytnutých vedúcim tejto práce.

Kompletnú implementáciu algoritmu pre trénovanie tohto modelu GAN, môžeme vidieť na Obr. 4.22. Zvyšné časti trénovacej funkcie *train*, ktoré pomáhajú s evaluáciou a ukladaním aktuálnych modelov si popíšeme v nasledujúcej podkapitole.

```
def train(g_model, d_model, gan_model, latent_dim, signal_train, signal_test,
          mu, sigma,length_of_signal, n_epochs=100, batch_size=200, n_eval=2):
  #polovica velkosti skupiny a pocet iteracii jednej epochy
  batch_per_epoch = int( SI2_II_11.shape[1] / batch_size) #16
  half_batch = int(batch_size / 2) #100
  #epocha
  for i in range(n_epochs):
    # iteracia
    for j in range(batch_per_epoch):
      # vygenerovanie realnych signalov a trening diskriminatoru
      x_real,y_real = generate_real_samples(half_batch,signal_train)
      loss_real,acc_real = d_model.train_on_batch(x_real, y_real)
      # vygenerovanie falosnych signalov a trening diskriminatoru
      x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch, mu, sigma)
      loss_fake, acc_fake = d_model.train_on_batch(x_fake, y_fake)
      # vygenerovanie latentnych bodov
      x_gan = generate_latent_points(latent_dim, batch_size,mu,sigma)
      # prevratenie oznacenia triedy falosnych signalov na realne
      label=[]
      for k in range(batch_size):
        y = np.round(random.uniform(0.9, 1),2)
        label.append(y)
      label=np.asarray(label)
      y_gan=label.reshape(batch_size,1)
      # trening generatoru
      g_loss = gan_model.train_on_batch(x_gan, y_gan)
      # ulozenie presnosti a chyb modelov
      d_real_loss_graph_sgd3_latent.append(loss_real)
      d_real_acc_graph_sgd3_latent.append(acc_real)
      d_fake_loss_graph_sgd3_latent.append(loss_fake)
      d_fake_acc_graph_sgd3_latent.append(acc_fake)
      g_loss_graph_sgd3_latent.append(g_loss)
      #ukladanie chyb na disk v pripade straty pripojenia/zlyhania
      np.save(WORKDIR_PATH / "d_real_loss_graph_sgd3_latent",d_real_loss_graph_sgd3_latent)
      np.save(WORKDIR_PATH / "d_real_acc_graph_sgd3_latent",d_real_acc_graph_sgd3_latent)
      np.save(WORKDIR_PATH / "d_fake_loss_graph_sgd3_latent",d_fake_loss_graph_sgd3_latent)
      np.save(WORKDIR_PATH / "d_fake_acc_graph_sgd3_latent",d_fake_acc_graph_sgd3_latent)
      np.save(WORKDIR_PATH / "g_loss_graph_sgd3_latent",g_loss_graph_sgd3_latent)
      # priebezne vysledky modelu
      print('>%d, %d/%d, d_real=%.0f%% d_fake=%.0f%%        d_loss=%f  g_loss=%f'
            % (i+1,j+1,batch_per_epoch, acc_real*100, acc_fake*100, (loss_real+loss_fake), g_loss))
    # krok evaluacie modelu a ukladania priebehu ucenia GAN
    if (i+1) % n_eval == 0:
      summarize_performance(i+1, g_model, d_model, signal_test, latent_dim, batch_size,
                            SI2_II_11_train_mean, SI2_II_11_train_std, length_of_signal)
```

Obr. 4.22: Implementácia algoritmu trénovania GAN pre generovanie EKG signálov.

4.3.7 Automatické ukladanie výsledkov a aktuálneho modelu GAN

Zostavenie stabilného modelu GAN je veľmi náročná úloha. Modely sú často nestabilné a ich optimalizovanie trvá dlhú dobu – trvanie tréningu je dlhé, každá zmena hyperparametrov a následné testovanie zaberá dlhú dobu. Z tohto dôvodu je ukladanie aktuálnych modelov a sledovanie výsledkou nutnosťou.

Vyhodnotenie modelu spolu s uložením vizuálneho výstupu a aktuálnych modelov generátoru a diskriminátoru zabezpečuje funkcia *summarize_performance*, ktorú vidíme na Obrázku 4.2.3. Funkcia je volaná na základe parametru *n_eval,* ktorý nám určuje krok evaluácie modelu (napríklad každú 5-tu epochu).

Diskriminátor vyhodnocujeme na testovacích signáloch (659), ktoré si vygenerujeme pomocou funkcie *generate_real_samples*. Následne rovnako evaluujeme diskriminátor aj na falošných signáloch, generovaných aktuálnym modelom generátoru. Funkcia *save_plot* ukladá aktuálne výstupy generátoru spolu s reálnymi signálmi (v jednom okne) v mriežke obrázkov o zvolenej veľkosti. Tieto výstupy boli použité pre analýzu modelov GAN, ktorú si popíšeme v ďalšej kapitole. Funkcia nakoniec všetky údaje spolu s aktuálnymi modelmi generátoru a diskriminátoru ukladá na Google Disk, do konkrétneho priečinku, ktorý si zadá užívateľ.

```
def summarize_performance(epoch, g_model, d_model, SI2_II_11_test,
                          latent_dim, batch_size, mu, sigma,length_of_signal):
  # evaluacia testovacich signalov
  x_real, y_real = generate_real_samples(batch_size,SI2_II_11_test)
  _, acc_real = d_model.evaluate(x_real, y_real, verbose=0)
  # evaluacia falosnych vygenerovanych signalov
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, batch_size, mu, sigma)
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # vyhodnotenie
  print('>Acc Test Real: %.0f%%, Acc Fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # ulozenie obrazkov priebehu ucenia GAN
  save_plot(acc_real, acc_fake, x_real,x_fake,length_of_signal,epoch, n=5)
  # ulozenie modelu generatoru
  g_model.save(str(WORKDIR_PATH / "GAN/train6/generatormodel") + str(epoch))
  d_model.save(str(WORKDIR_PATH / "GAN/train6/discriminatormodel") + str(epoch))
```

Obr. 4.23: Funkcia automatického ukladania výsledkov a priebehu učenia GAN

5 VÝSLEDKY MODELOV GAN

V tejto kapitole sa bližšie pozrieme na vizuálne výstupy modelov GAN. Začneme jednoduchšími modelmi pre generovanie 1D funkcií – x^2 a sínus. Následne sa vrhneme na komplexný model pre generovanie EKG záznamov. Každý z modelov si budeme podrobne analyzovať, a snažiť sa pochopiť dôvody jeho správania.

Analýza modelov je realizovaná pomocou porovnania kriviek chýb generátora a diskriminátora spolu s vyhodnotením priebežnej presnosti diskriminátora. Na základe týchto kriviek, dokážeme dobre zdôvodniť vizuálne výstupy modelov GAN. Pochopenie správania modelov GAN je veľmi dôležité pre následnú optimalizáciu modelu – najmä analýza zlyhania modelov, ktorú sme si popísali v podkapitole 2.3.

5.1 Výsledky modelu GAN pre generovanie funkcie x^2

Začneme výsledkami z najjednoduchšieho modelu – GAN pre generovanie funkcie x^2. Implementáciu, architektúru a podrobný popis fungovania modelu sme popísali v podkapitole 4.1. Trénovanie celého modelu trvalo s použitím GPU (Google Colab) približne 20 minút.

5.1.1 Vizualizácia dosiahnutých výsledkov modelu GAN pre generovanie funkcie x^2

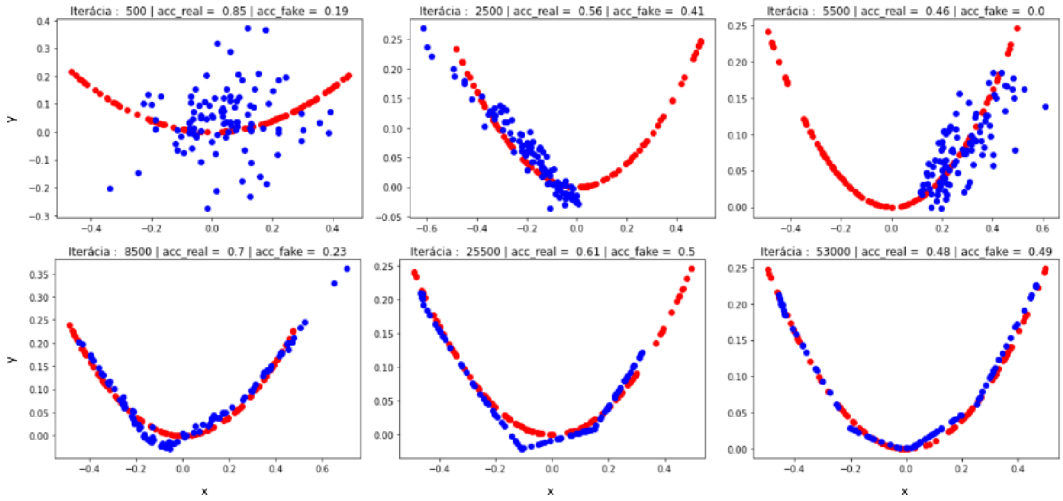
Trénovacie parametre modelu boli nasledovné:

- Počet iterácií *n_iter* bol nastavený na 100 000.
- Veľkosť trénovacej skupiny *n_batch* bola zvolená 512.
- Veľkosť dimenzie latentného priestoru *latent_dim* nastavená na 5.
- Optimalizačný algoritmus pre diskiminátor – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9*.
- Optimalizačný algoritmus pre generátor – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9*.
- Krok evaluácie modelu *n_eval* bol zvolený 500.

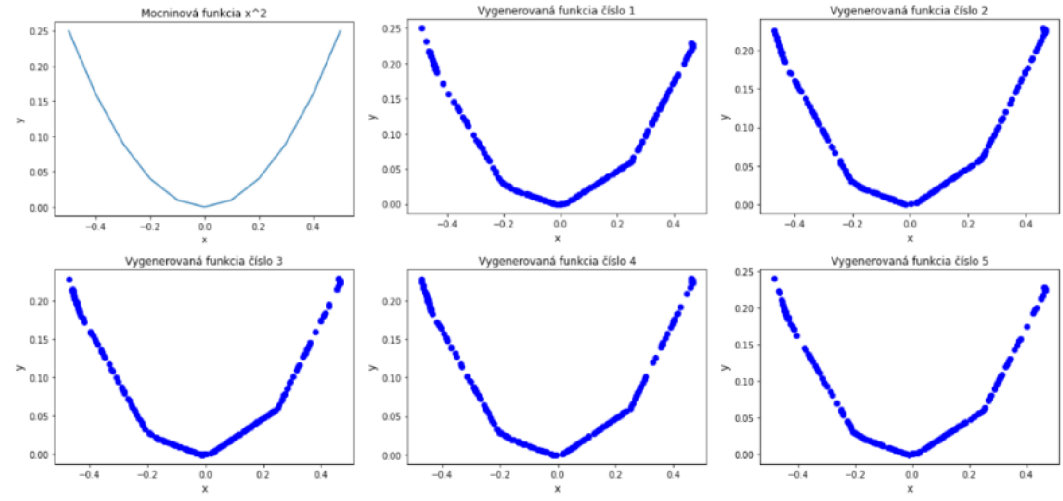
Tréning tohto konkrétneho modelu GAN bol ukončený v prípade, že počet iterácií bol väčší ako 50 000, a presnosť diskriminátoru sa pre reálne aj falošné vzorky nachádzala v intervale <0.475, 0.525>. Cieľom trénovania GAN, je naučiť generátor generovať vzorky, ktoré diskriminátor nebude schopný odlíšiť od reálnych (podkapitola 2.2). Preto sme nastavili tento interval – bolo očakávané, že po dostatočnom počte iterácií (50 000) diskriminátor dospeje do bodu, kedy priradí obom vzorkám pravdepodobnosť ½ a teda nebude schopný rozoznať reálne vzorky od falošných.

Táto podmienka bola splnená v iterácií číslo 53 000. Na Obrázku 5.1 vidíme, že generátor generuje dostatočne reálne vyzerajúce vzorky. Teraz keď máme tréning

zastavený, môžeme naučený model generátoru využiť na generovanie nových dát – pripomínajúcich kvadratickú funkciu $f(x)=x^2$ (podkapitola 4.1). Generované výsledky (viď. Obr. 5.2) považujeme za dostatočné a cieľ úlohy považujeme za splnený.



Obr. 5.1: Zobrazenie postupného učenia GAN pre generovanie funkcie x^2. Vidíme, že s narastajúcim počtom iterácií sa GAN učí generovať (modré body) podobné vzorky ako reálne dáta (červené body). V názve grafu vidíme v akej iterácií sa GAN aktuálne nachádza a s akou pravdepodobnosťou si diskriminátor myslí, že sa jedná o reálnu (acc_real) a falošnú (acc_fake) vzorku.



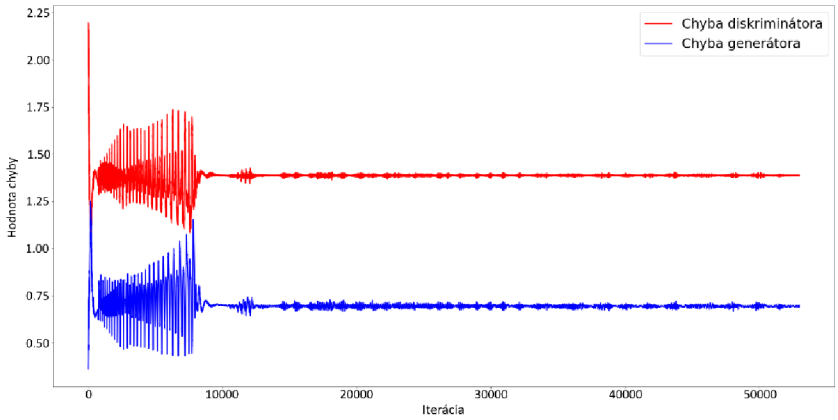
Obr. 5.2: Porovnanie výstupov naučeného generátora s cieľovou mocninovou funkciou x^2. Celkovo bolo generovaných 250 bodov pre každú z funkcií.

V ďalšej podkapitole sa pozrieme na podrobnejšiu analýzu tohto modelu GAN, kde si zobrazíme chyby diskriminátora aj generátora a budeme komentovať dosiahnuté výsledky.

5.1.2 Analýza modelu GAN pre generovanie funkcie x^2

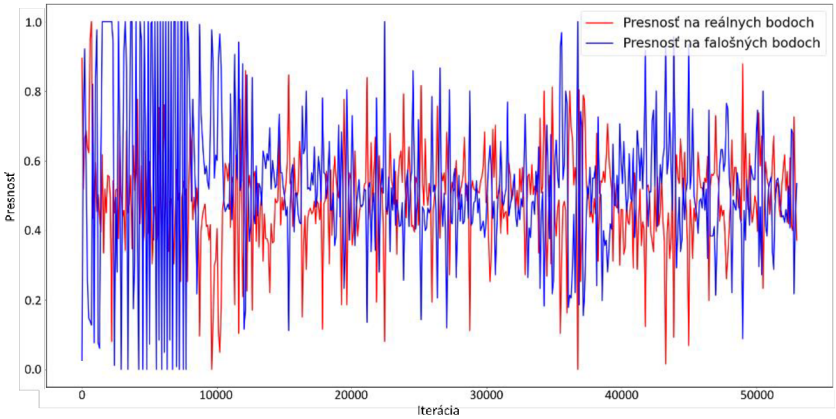
Výsledky alebo zlyhania modelov GAN môžeme analyzovať na základe vývoja chýb diskriminátora a generátora. Taktiež sa môžeme pozrieť na presnosti, s akými diskriminátor zaraďoval falošné aj reálne vzorky v priebehu trénovania.

Ako sme spomenuli v podkapitole 2.3.3, naším cieľom je aby model GAN dosiahol stabilného stavu – chyby generátoru a diskriminátoru sa nebudú značne meniť a ustália sa na určitej hodnote. Na Obrázku 5.3 vidíme, že na začiatku tréningu chyby diskriminátoru aj generátoru oscilovali s veľkou odchýlkou – model sa učil. Približne v iterácií číslo 8000 sa chyby oboch UNS ustálili – bolo dosiahnuté stabilného stavu. To môžeme vidieť aj v iterácií číslo 8500 (na Obr. 5.1 - vľavo dole) – generované výsledky začali pripomínať tvar funkcie x^2.



Obr. 5.3: Vývoj chýb diskriminátoru a generátoru počas učenia modelu GAN pre generovanie funkcie x^2. Chyba diskriminátoru bola počítaná ako súčet chýb diskriminátoru na reálnych aj falošných dátach.

Ďalej sa pozrieme na analýzu presnosti diskriminátora počas učenia modelu GAN. Na Obrázku 5.4 vidíme, že priebeh presností diskriminátoru má podobný charakter ako priebeh chýb. Zo začiatku si diskriminátor nebol istý, čo sú reálne a čo falošné dáta – priebeh presností značne osciloval. Následne rovnako ako v predošlom prípade, približne pri 8000-tej iterácií oscilácie mierne poklesli. Vidíme, že od tejto chvíle, generátor dokonale „klamal“ diskriminátor, ktorý v celom ďalšom procese tréningu osciloval okolo hodnoty 0.5 pre reálne aj falošné vzorky. Pôvodný graf vývoja presností diskriminátoru bol neprehľadný – z tohto dôvodu sme na zobrazenie presností, vyberali len každú 100-tú hodnotu presnosti.



Obr. 5.4: Vývoj presností diskriminátoru na reálnych a falošných dátach počas trénovania modelu GAN pre generovanie funkcie x^2. Pre lepšiu názornosť je zobrazená len každá 100-tá hodnota presnosti.

5.2 Výsledky modelu GAN pre generovanie funkcie sínus

Pokračujeme výsledkami z mierne pokročilého modelu – GAN pre generovanie funkcie sínus. Implementáciu, architektúru a podrobný popis fungovania modelu sme popísali v podkapitole 4.2.

Model GAN budeme trénovať na dvoch rôznych skupinách reálnych dát. Prvá skupina pozostáva z funkcií sínus s rovnakou frekvenciou aj amplitúdou. V druhej skupine dát bola trénovacia skupina mierne augmentovaná – amplitúdy jednotlivých sínusoviek mali rovnakú frekveniciu ale ich amplitúda sa náhodne menila. Týmto spôsobom môžeme podrobnejšie analyzovať a porovnať jednotlivé prístupy – čo nám pomôže pochopiť správanie GAN pri mierne pokročilých problémoch.

Trénovanie prvého z modelov trvalo s použitím GPU (Google Colab) približne 165 minút, tréning druhého modelu (GPU) trval 340 minút.

5.2.1 Vizualizácia dosiahnutých výsledkov modelu GAN s nemennou amplitúdou

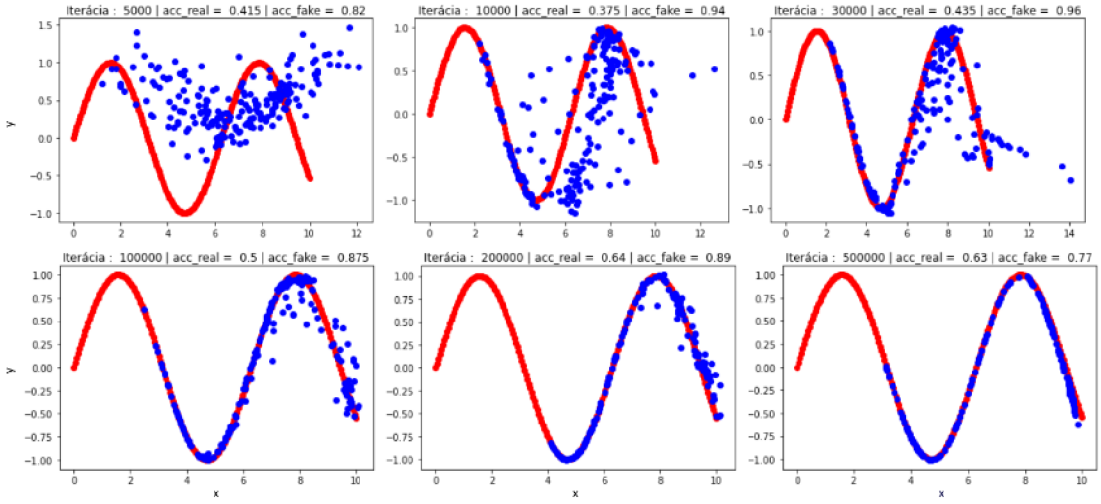
Trénovacie parametre prvého modelu (s nemennou amplitúdou) boli nasledovné:

- Počet iterácií *n_iter* bol nastavený na 500 000.
- Veľkosť trénovacej skupiny *n_batch* bola zvolená 512.
- Veľkosť dimenzie latentného priestoru *latent_dim* nastavená na 5 [7, 49].
- Optimalizačný algoritmus pre diskiminátor – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9* [49].
- Optimalizačný algoritmus pre generátor – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9* [49].
- Krok evaluácie modelu *n_eval* bol zvolený 5000.

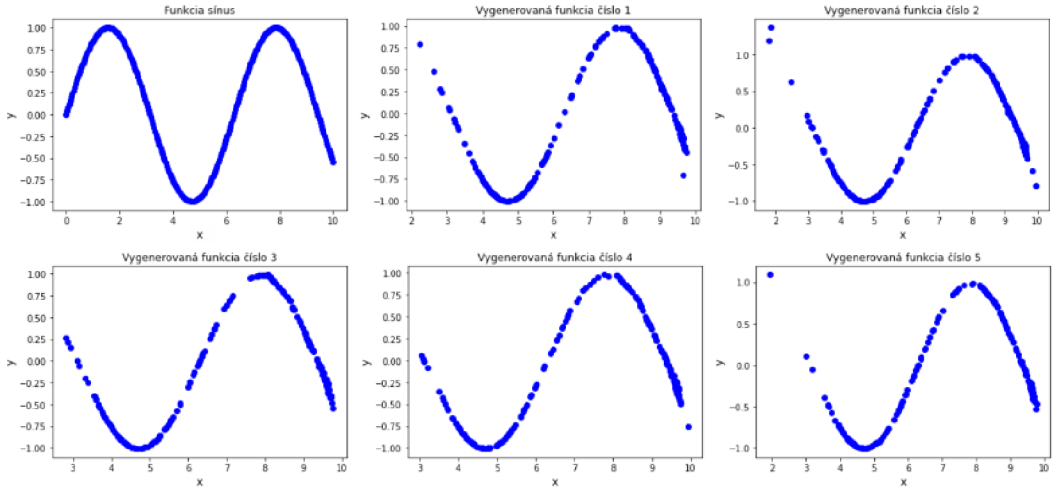
Z postupného priebehu trénovania (s nemennou amplitúdou) GAN na Obrázku 5.5 vidíme, že zo začiatku GAN dobre rozoznáva extrémy funkcie sínus. Generované body sú zväčša rozptýlené okolo týchto extrémov a následne sa učia kopírovať priebeh funkcie. V poslednej iterácií však vidíme, že GAN nedokázal pokryť celú dĺžku funkcie sínus. Tento jav sa opakoval aj v iných testovaniach modelu, niekedy však na opačnej strane funkcie. Príčinou mohla byť malá veľkosť skupiny trénovacích aj generovaných dát, alebo spôsob generovania latentných bodov.

Po dosiahnutí maximálneho počtu iterácií, v tomto prípade hovoríme o 500 000 iteráciach, bolo trénovanie modelu ukončené. Výsledný model sa dobre naučil kopírovať funkciu sínus, avšak zachytil len približne dve tretiny dĺžky tejto funkcie. Výstupy z naučeného generátoru sú uvedené na Obrázku 5.6. Vidíme, že približne v 100 000-tej iterácií začínajú vygenerované body pripomínať priebeh funkcie sínus. Na základe tejto informácie predpokladáme, že v rovnakej iterácií nájdeme pri krivkách

chyby diskriminátoru a generátoru kľúčové zmeny, ktoré si budeme analyzovať v ďalšej podkapitole.



Obr. 5.5: Zobrazenie postupného učenia GAN pre generovanie funkcie sínus s nemennou amplitúdou. Vidíme, že s narastajúcim počtom iterácií sa GAN učí generovať (modré body) podobné vzorky ako reálne dáta (červené body). V názve grafu vidíme v akej iterácií sa GAN aktuálne nachádza a s akou pravdepodobnosťou si diskriminátor myslí, že sa jedná o reálnu (acc_real) a falošnú (acc_fake) vzorku.

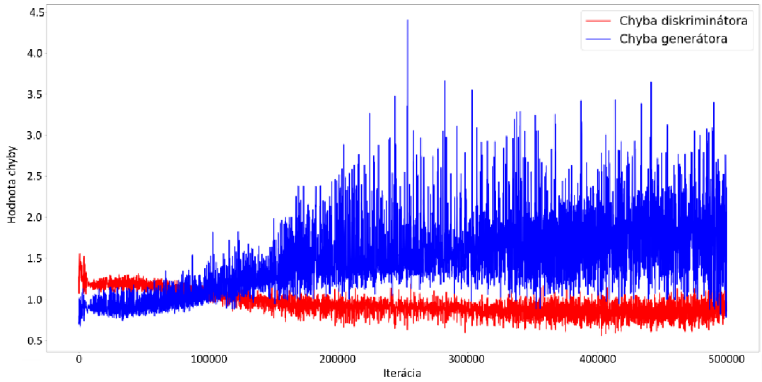


Obr. 5.6: Porovnanie výstupov naučeného generátora s cieľovou funkciou sínus. Celkovo bolo generovaných 250 bodov pre každú z funkcií.

5.2.2 Analýza modelu GAN s nemennou amplitúdou

Prvý z modelov, sa dokonale naučil kopírovať priebeh dvoch tretín funkcie sínus (viď. Obr. 5.1). Tento model bol trénovaný vždy na rovnakej skupine nemenných funkcií sínus – ich amplitúda ani frekvencia sa nemenila. Preto bolo predpokladané, že v tomto prípade sa môže GAN veľmi ľahko preučiť a nebude generovať unikátne výstupy. Vidíme, že "naučený" (preučený) generátor tohto modelu, generoval viacmenej rovnaké výstupy (Obr. 5.2) – hovoríme o strate unikátnosti výstupu GAN (podkapitola 2.3.3).

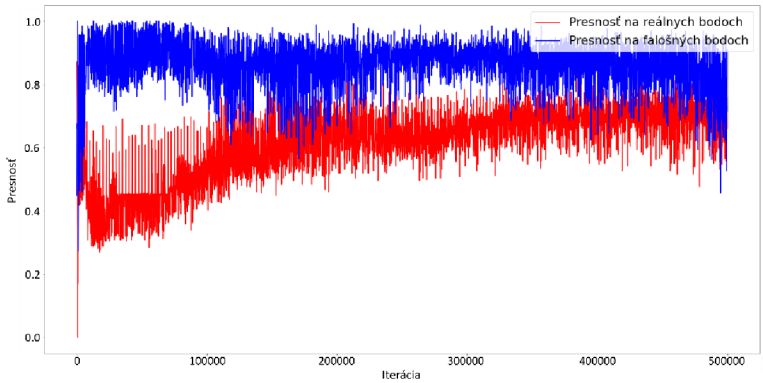
Začiatok tohto zlyhania môžeme vidieť približne v iterácií číslo 100 000, kedy sa grafy chýb začínajú prekrižovať. Chyba generátoru osciluje s vysokou odchýlkou – generátor skáče z jednej hodnoty chyby na druhú a naopak. Tento typ zlyhania je u GAN modelov bežný – avšak v tomto prípade bol spôsobený použitou databázou trénovacích dát. Generátor sa naučil generovať jeden výstup, a naopak diskriminátor sa ho naučil jednoducho odlíšiť od reálnych dát. To môžeme vidieť na Obrázku 5.7. Na obrázku, vidíme, že chyba diskriminátora je malá a stabilná (červená farba) – diskriminátor sa už ďalej neučí, respektíve sa učí minimálne.



Obr. 5.7: Vývoj chýb diskriminátoru a generátoru počas učenia modelu GAN pre generovanie funkcie sínus s nemennou amplitúdou. Chyba diskriminátoru bola počítaná ako súčet chýb diskriminátoru na reálnych aj falošných dátach.

Táto situácia vývoja chyby diskriminátoru je ideálna – chceme, aby sa diskriminátor najrpv naučil rozoznávať vstupné dáta a následne chceme aby jeho chyba pomaly klesala. Lokálne oscilácie chyby diskriminátoru sú znakom učenia modelu.

Naučil sa teda dobre rozoznávať falošné a reálne vzorky, čo nám potvrdzuje aj Obrázok 5.8 – diskriminátor dosiahol vysokých hodnôt presností pre obe triedy dát. Tento fakt, však hodnotíme negatívne – chceme aby presnosť oscilovala okolo ½, čo by značilo že generátor generuje vzorky, ktoré diskriminátor nedokáže rozoznať od reálnych.



Obr. 5.8: Vývoj presností diskriminátoru na reálnych a falošných dátach počas trénovania modelu GAN pre generovanie funkcie sínus nemennou amplitúdou. Pre lepšiu názornosť je zobrazená len každá 100-tá hodnota presnosti.

Model sa teda naučil generovať funkciu sínus, ale nakoľko tieto body zachytávali len určitý rozsah (2/3 osi x) cieľovej funkcie– diskriminátor pravdepodobne vedel, že sa jedná o falošné dáta – viď. Obr. 5.5.

5.2.3 Vizualizácia dosiahnutých výsledkov modelu GAN s premenlivou amplitúdou

Druhý z modelov GAN, sa trénoval na sínusových funkciách s premenlivou amplitúdou. Týmto sme sa viac priblížili k cieľu tejto diplomovej práce – generovaniu signálov EKG. Databáza, ktorú sme vytvorili v kapitole 3, obsahuje množstvo signálov, ktoré majú síce rovnaké vlastnosti, ale samotné signály sa líšia. Práve z tohto dôvodu, sme chceli zistiť ako bude reagovať GAN na takúto zmenu trénovacích dát.

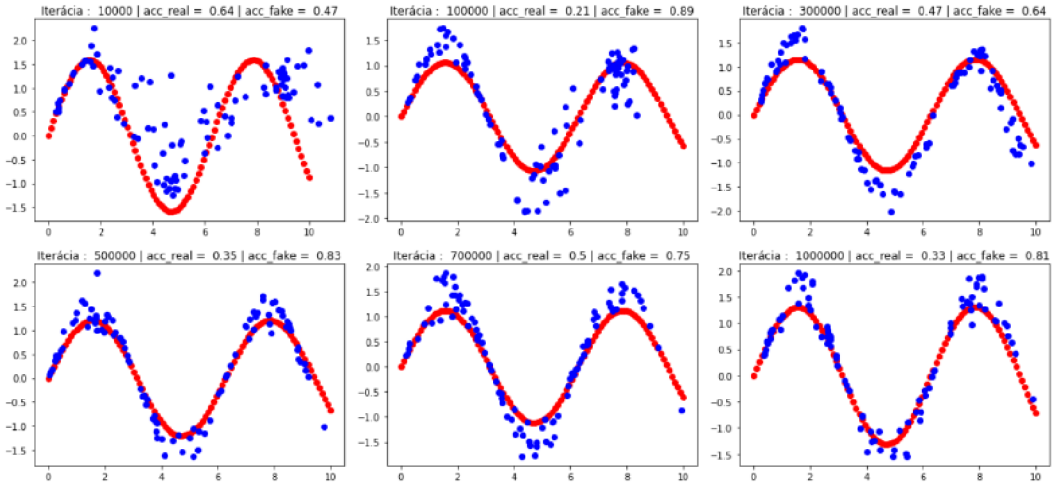
Nastavenie trénovacích parametrov druhého modelu (s premenlivou amplitúdou):

- Počet iterácií *n_iter* bol zvýšený na 1 000 000 – z dôvodu väčšej komplexnosti riešeného problému.
- Veľkosť trénovacej skupiny *n_batch* bola ponechaná na hodnotu 512. Táto hodnota nám prišla v predchádzajúcom modeli ako dostatočne veľká.
- Dimenzia latentného priestoru *latent_dim=5*, zostala rovnaká [7, 49].
- Optimalizačný algoritmus pre diskiminátor bol ten istý – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9* [49].
- Optimalizačný algoritmus pre generátor zostal rovnaký – ADAM s nastavenými parametrami rýchlosti učenia $\eta=0.001$ a zabúdania *beta_1=0.9* [49].
- Krok evaluácie modelu *n_eval* bol zvýšený na 10 000– vizuálne vyhodnotenie modelu, tentoraz nebudeme potrebovať tak často – model sa učí pomalšie.

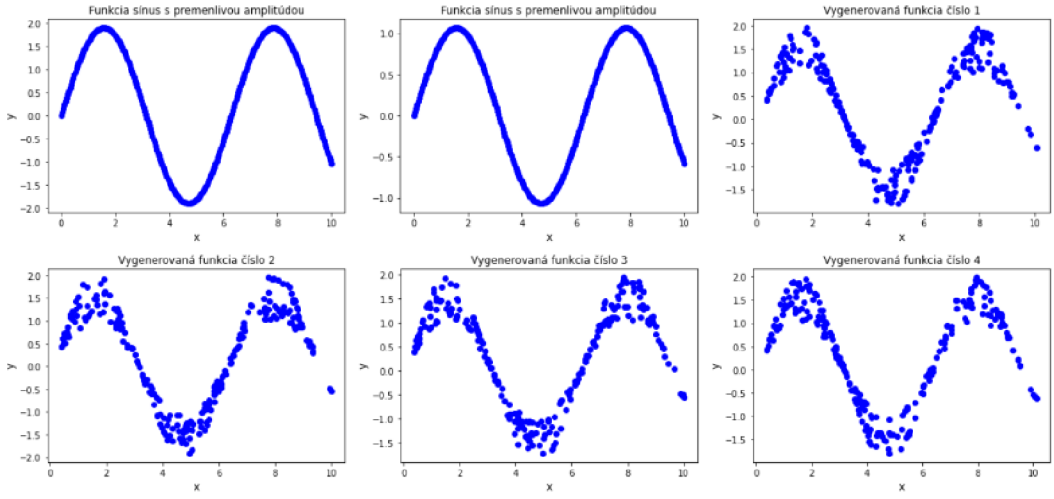
Priebeh trénovania tohto modelu vidíme na Obrázku 5.9. Už po 100 000 iteráciách pozorujeme podobnosť medzi generovanými a reálnymi dátami. Na rozdiel od prvého spôsobu trénovania GAN, kde sme GAN trénovali na rovnakej funkcií sínus, v tomto prípade augmentácia dát pomohla robustnosti tohto modelu.

Vidíme, že model sa nepretrénoval na určitej časti funkcie, a lepšie pochopil ako má funkcia „vyzerať“. Kvalita generovaných dát ale nie je tak dobrá a body sú veľmi rozptýlené. To mohlo byť spôsobené nedostatočným tréningom siete, alebo jednoduchosťou použitej architektúry na riešený problém.

Výstupy z generátoru tohto modelu, sú uvedené na Obrázku 5.10. Pozorujeme, že GAN znovu pridelil extrémom funkcie viac vzoriek – môžeme teda tvrdiť, že GAN zo začiatku najlepšie „chápe“ práve tieto oblasti funkcie. Predpokladáme, že pri dlhšom tréningu a ladení siete, by sme dosiahli lepších výsledkov. Naše výsledky avšak považujeme za dostačujúce. V nasledujúcej podkapitole sa pozrieme na analýzu tohto modelu.



Obr. 5.9: Zobrazenie postupného učenia GAN pre generovanie funkcie sínus s premenlivou amplitúdou. Vidíme, že s narastajúcim počtom iterácií sa GAN učí generovať (modré body) podobné vzorky ako reálne dáta (červené body). V názve grafu vidíme v akej iterácií sa GAN aktuálne nachádza a s akou pravdepodobnosťou si diskriminátor myslí, že sa jedná o reálnu (acc_real) a falošnú (acc_fake) vzorku.



Obr. 5.10: Porovnanie výstupov naučeného generátora s cieľovou funkciou sínus s premenlivou amplitúdou. Na prvých dvoch obrázkoch vidíme, že trénovacie dáta mali rozdielne amplitúdy. Celkovo bolo generovaných 250 bodov pre každú z funkcií.

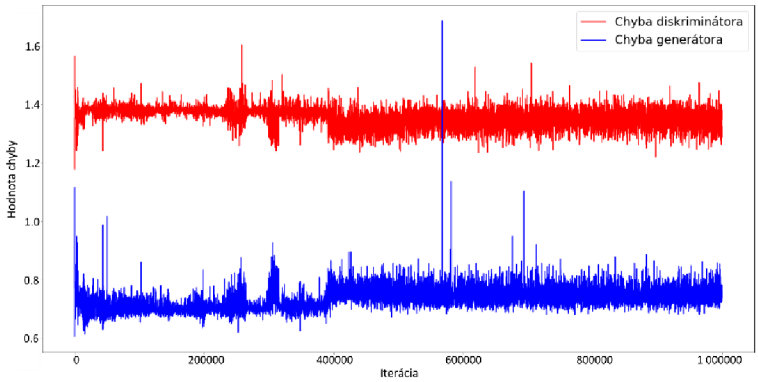
5.2.4 Analýza modelu GAN s premenlivou amplitúdou

Tento model síce nedosiahol dokonalých výsledkov, ale jeho priebeh tréningu pôsobil stabilne. Keď sa pozrieme na vývoj chyby modelu GAN pre generovanie funkcie x^2 (viď. Obr. 5.3), vidíme medzi nimi určitú podobnosť. Hodnoty chýb generátora aj diskriminátora sú si zo začiatku veľmi podobné, druhý z modelov má však omnoho väčšie oscilácie – model sa ešte stále učil (viď. Obrázok 5.11).

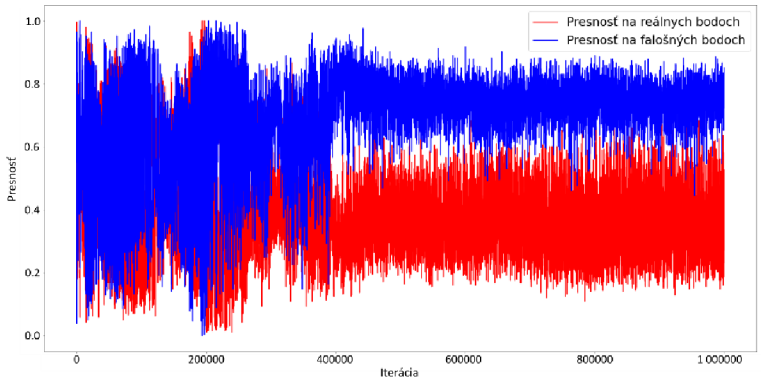
Na Obrázku 5.12, je jasne vidieť, že diskriminátor si bol časom menej istý presnosťou falošných bodov – krivka postupne klesala. Taktiež vidíme, že zo začiatku tréningu diskriminátor nevedel dobre rozlíšiť jednotlivé skupiny signálov – odchýlka bola priveľká. Postupne, približne v 400 000 iterácií, sa presnosť na reálnych aj

falošných dátach ustálila. Predpokladáme, že model sa ešte stále učil, a s dostatočne dlhým časom tréningu, alebo ladením siete by dosiahol lepších výsledkov.

Konštatujeme, že proces učenia druhého modelu GAN pre generovanie funkcií sínus s premenlivou amplitúdou bol stabilný – augmentácia trénovacích dát mala pozitívny vplyv na stabilitu a robustnosť modelu. Na rozdiel od prvého modelu s nemennou amplitúdou sinusových funkcií sa tento model nepreučil a pokračoval v efektívnom tréningu.



Obr. 5.11: Vývoj chýb diskriminátoru a generátoru počas učenia modelu GAN pre generovanie funkcie sínus s premenlivou amplitúdou. Chyba diskriminátoru bola počítaná ako súčet chýb diskriminátoru na reálnych aj falošných dátach.



Obr. 5.12: Vývoj presností diskriminátoru na reálnych a falošných dátach počas trénovania modelu GAN pre generovanie funkcie sínus premenlivou amplitúdou. Pre lepšiu názornosť je zobrazená len každá 100-tá hodnota presnosti.

Vidíme, že trénovanie GAN modelov je zdĺhavé a konvergencia do lokálneho minima môže byť náročná. Modely GAN sú často nestabilné, avšak po hlbšej analýze, sa dá pochopiť dôvodom ich zlyhania – riešenie týchto problémov a optimalizácia je ale tou ťažšou časťou trénovania GAN.

Uvedené modely sme sa ďalej už nesnažili vylepšiť, nakoľko cieľom práce bolo vygenerovať EKG signály. Ako bolo spomenuté v podkapitole 5.2, prvý z modelov sa učil 165 minút a druhý z modelov 340 minút. Každý pokus o zmenu parametrov (ladenia siete) zaberal dlhý čas – modely GAN sú taktiež „stochastické“, pri každom ďalšom spustení môžeme dostať značne rozdielne výsledky.

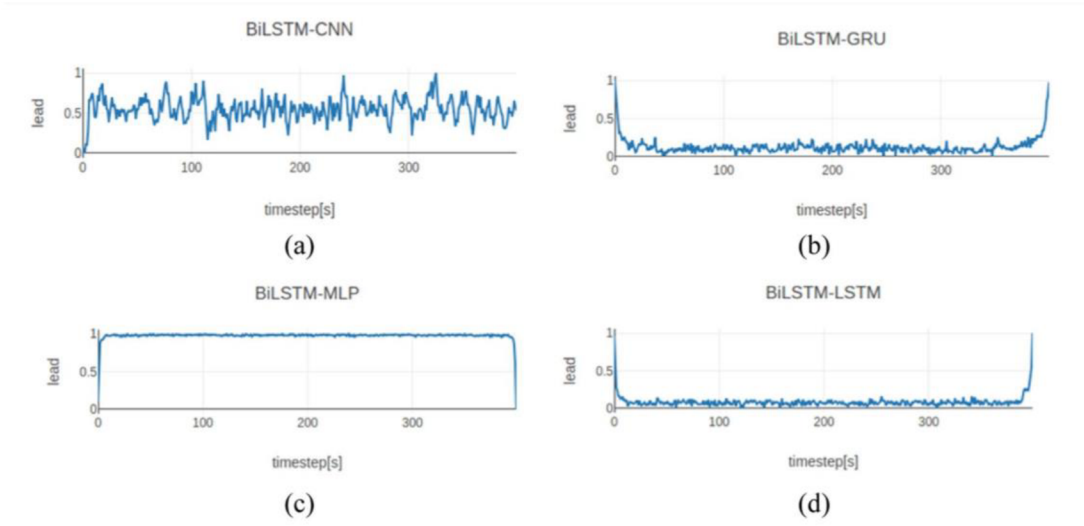
5.3 Výsledky modelu GAN pre generovanie EKG signálov

Finálny model GAN pre generovanie EKG signálov bol inšpirovaný odborným článkom [7], ktorý využíval tzv. BiLSTM-CNN model GAN. Ten pozostával z diskriminátora využívajúceho 1D konvolučné vrstvy, a z generátoru, ktorý používal BiLSTM vrstvy. Predspracovanie signálov, potrebných na trénovanie tohto modelu bolo popísané v kapitole 3, implementácia a ukážka architektúr sa nachádza v podkapitole 4.3.

5.3.1 Vizualizácia dosiahnutých výsledkov odborného článku

Ako prvé sa pozrieme na výsledky z článku [7], ktoré sme sa v tejto práci snažili napodobniť a následne vylepšiť – hovoríme o optimalizácií modelu. Článok [7] využíval pre tréning počítač s procesorom Intel i7-7820X (8 jadier) CPU, s primárnou pamäťou o veľkosti 16GB. Ako programovací jazyk autori použili Python 2.7, na zostavenie sietí boli použité balíčky PyTorch a NumPy.

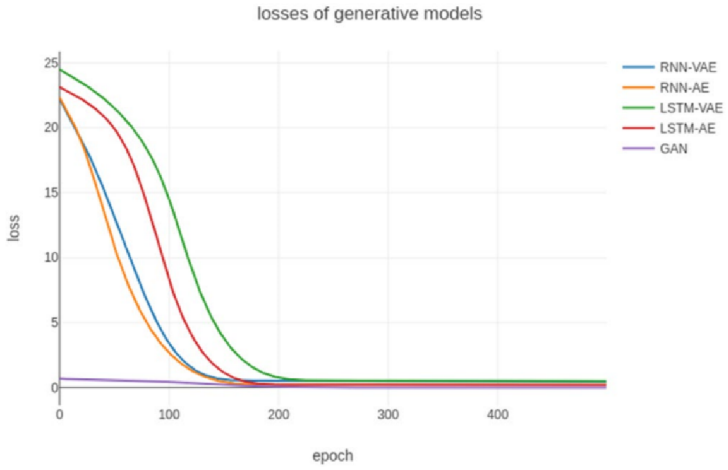
Autori tejto práce používali na tréning 30 minútové signály z MIT-BIH databázy, detailnejší popis sa nachádza v podkapitole 3.4.1. Práca porovnávala 4 rôzne použité architektúry na generovanie EKG signálov. Všetky modely boli trénované po dobu 500 epôch, so signálmi o dĺžke 3120. Veľkosť trénovacej skupiny zvolili na hodnotu 100 a používali optimalizačný algoritmus SGD s parametrom rýchlosti učenia o veľkosti 10^{-5}. Dĺžka výsledných generovaných signálov bola 400 – výsledné generované signály 4 rôznych architektúr môžeme vidieť na Obr. 5.13.



Obr. 5.13: Vygenerované EKG signály pomocou 4 rôznych architektúr – BiLSTM-CNN (a), BiLSTM-GRU (b), BiLSTM-MLP (c) a BiLSTM-LSTM (d). Na horizontálnej osi sú zobrazené časové kroky signálu, na vertikálnej osi hodnoty zvodu signálu, prevzaté z [7].

V záveri práce autori tvrdia, že ich vygenerované signály boli morfologicky podobné s EKG signálmi a mali podobnú štatistickú distribúciu ako signály EKG. Toto tvrdenie nám avšak nepríde pravdivé (viď. Obr. 5.13). Signály mohli odpovedať štatistickému rozloženiu EKG signálov ale rozhodne neboli EKG signálom podobné.

Ďalej práca ukazuje krivky chýb generatívnych modelov (viď. Obr. 5.14) – presnejší popis kriviek sme v práci nenašli. Autori argumentujú, že ich model konvergoval k nule zo všetkých modelov najrýchlejšie (epocha 200) a začiatočná chyba modelu GAN bola značne menšia ako chyba ostatných modelov. Toto tvrdenie avšak považujeme za nesprávne – ak jeden z "hráčov" GAN (generátor, diskriminátor) dosiahne hodnotu chyby blízku 0, váhy sa prestanú aktualizovať a model začína zlyhávať. Ako sme videli na Obrázku 5.3 – vývoj chýb oboch hráčov by sa mal síce ustáliť, ale nikdy by nemal dosiahnúť nuly.



Obr. 5.14: Krivka chýb niekoľkých generatívnych modelov, prevzatá z [7].

Práci ale nemôžeme poprieť tvrdenie, že ich model dosahoval najlepších výsledkov spomedzi všetkých porovnávaných metód.

5.3.2 Vizualizácia výsledkov modelu GAN pre generovanie EKG signálov

Architektúra finálneho modelu GAN pre generovanie EKG signálov je vysvetlená a dopodrobne popísaná spolu s implementáciou a ukážkami kódu. Kým sme dospeli k finálnemu modelu prevádzali sme v samotnej architektúre niekoľko zmien. V nasledujúcom odstavci si stručne popíšeme zistené informácie, ktoré sme nadobudli pri trénovaní a navrhovaní architektúry nášho modelu GAN.

V architektúre diskriminátoru boli pridané AF LeakyReLU (autori článku [7] nespomenuli ich použitie). Tieto vrstvy mali zamedziť problému miznúceho gradientu.

Leaky ReLU vrstvy sme skúsili aplikovať aj do generátoru, ten však po ich aplikácií znížil amplitúdy vygenerovaných signálov a celkovo generátor fungoval lepšie bez použitia Leaky ReLU vrstiev medzi jednotlivými BiLSTM vrstvami. Parameter Dropout vrstvy sme nemenili – bol ponechaný rovnako ako v [7].

Ďalším vylepšením modelu bolo tzv. transformácia hodnoty klasifikačných tried [17], ktorá spočívala v nahradení „tvrdých" označení tried (0,1), hodnotami v určitom intervale (napr. 0-0.1, 0.9-1). Táto zmena hodnoty klasifikačných tried mala zvýšiť robustnosť nášho modelu a taktiež pomôcť s robustnosťou modelu.

Narozdiel od klasických spôsobov generovania latentných bodov – sme generovanie latentných bodov prispôsobili štatististickému rozloženiu trénovacíh dát (signálov). Toto vylepšenie malo pozitívny vplyv na generované dáta.

Ďalším z parametrov, ktoré sme sa snažili otestovať bol použitý optimalizačný algoritmus. Celkovo bolo vyskúšané dva tipy algoritmov – ADAM, ktorý sme používali aj v generovaní 1D signálov a algoritmus SGD. Algoritmus SGD používali aj v práci [7], z tohto dôvodu sme ho používali aj pri našom modely na generovanie EKG signálov.

Parameter rýchlosti učenia sa ukázal ako jeden z najdôležitejších parametrov ladenia siete. Testovali sme celkovo 3 rýchlosti učenia – $1x10^{-3}$, $1x10^{-4}$, $1x10^{-5}$. V práci [7] používali algoritmus SGD s rýchlosťou učenia $1x10^{-5}$, avšak nám sa viac osvedčili rýchlosti $1x10^{-3}$, $1x10^{-4}$ pretože sme kvôli výpočtovým obmedzeniam nedokázali model trénovať dlho – Google Colab poskytuje GPU len na 12 hodín. Z tohto dôvodu, sme vopred vedeli, že nebude model možné dostatočne dlho trénovať a zvolili väčšie rýchlosti trénovania.

Veľkosť trénovacej skupiny sme zvolili na 200 – aby sme naplno využili pamäť GPU, nakoľko sme vedeli, že tréning bude prebiehať maximálne 12 hodín.

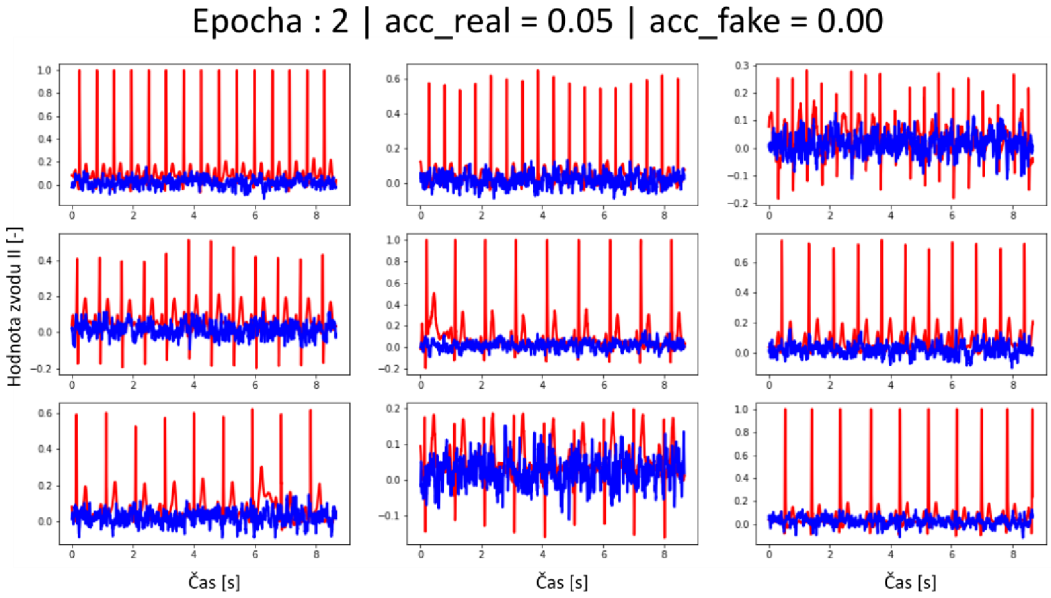
Posledným z parametrov bola veľkost latentnej dimenzie *latent_dim*. Boli vyskúšané 2 veľkosti – veľkosť 5, používaná v [7,49] a veľkosť 100. Pri použití väčšej dimenzie, sme zaznamenali väčšiu amplitúdu vygenerovaných signálov, celkovo s lepšími vizuálnymi výstupmi.

Počet epôch trénovania a krok evaluácie sme volili na základe dostupného času (12h), ktorý bol spolu s výpočtovou silou najväčším problémom pri ladení siete. Každá zmena modelu potrebovala následnú analýzu – tréning modelu však väčšinou trval 8-10 hodín. Ďalším faktorom bol aj „stochastický charakter“ GAN modelov, ktorých správanie je často nestabilné a pri každom novom spustení modelu, môžeme dosiahnuť veľmi rozdielnych výsledkov.

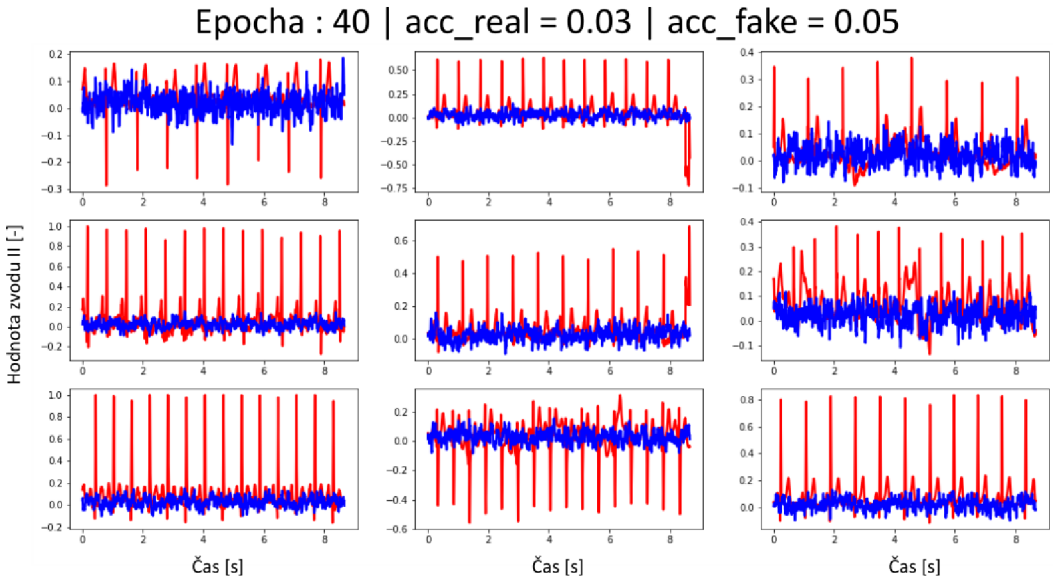
Náš najlepší model využíval nasledujúce parametre a vylepšenia:

- Počet epôch trénovania *n_epochs*=50
- Veľkosť trénovacej skupiny – parameter *batch_size = 200*
- Veľkosť dimenzie latentného priestoru – parameter *latent_dim* = 100
- Optimalizačný algoritmus diskriminátoru spolu s parametrom rýchlosti učenia – SGD s rýchlosťou učenia 10^{-3}.
- Optimalizačný algoritmus generátoru spolu s parametrom rýchlosti učenia – SGD s rýchlosťou učenia 10^{-3}.
- Krok evaluácie modelu – parameter *n_eval* = 2
- Generátor bez použitia Leaky ReLU vrstiev.
- Použitie transformovaných označení klasifikačných tried
- Použitie Leaky ReLU vrstvy v diskriminátore

Dosiahnuté výsledky môžeme vidieť na Obrázkoch 5.16 a 5.17. Vidíme, že charakter vygenerovaných signálov sa počas trvania 40 epôch nezmenil. Vygenerované signály dokázali dobre zachytiť izolíniu signálu, a vidíme, že sa periodicky opakujú – amplitúda vygenerovaných signálov bola ale nízka – žiadny zo signálov nedokázal simulovať R vlnu.



Obr. 5.16: Zobrazenie generovaných signálov spolu s trénovacími signálmi (červené) pomocou modelu GAN pre generovanie EKG signálov. Hodnota zvodu signálov bola transformovaná na interval <-1, 1>. V názve grafu vidíme v akej iterácii sa GAN aktuálne nachádza a s akou pravdepodobnosťou si diskriminátor myslí, že sa jedná o reálnu (acc_real) a falošnú (acc_fake) vzorku.

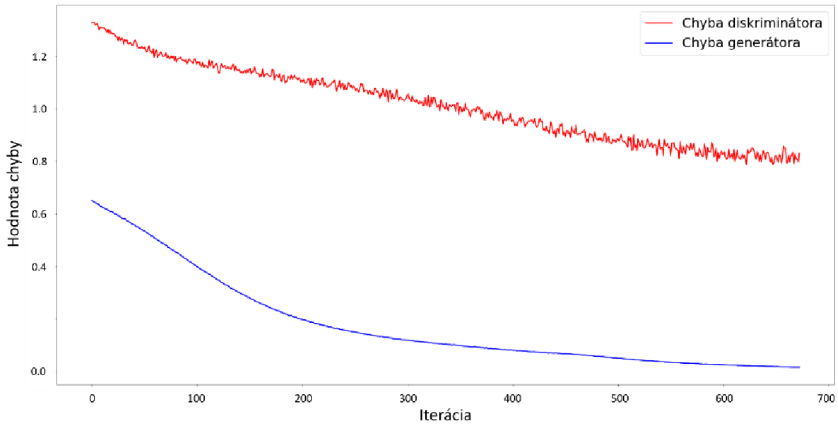


Obr. 5.17: Zobrazenie generovaných signálov spolu s trénovacími signálmi (červené) pomocou modelu GAN pre generovanie EKG signálov. Hodnota zvodu signálov bola transformovaná na interval <-1, 1>. V názve grafu vidíme v akej iterácii sa GAN aktuálne nachádza a s akou pravdepodobnosťou si diskriminátor myslí, že sa jedná o reálnu (acc_real) a falošnú (acc_fake) vzorku.

5.3.3 Analýza modelu GAN pre generovanie EKG signálov

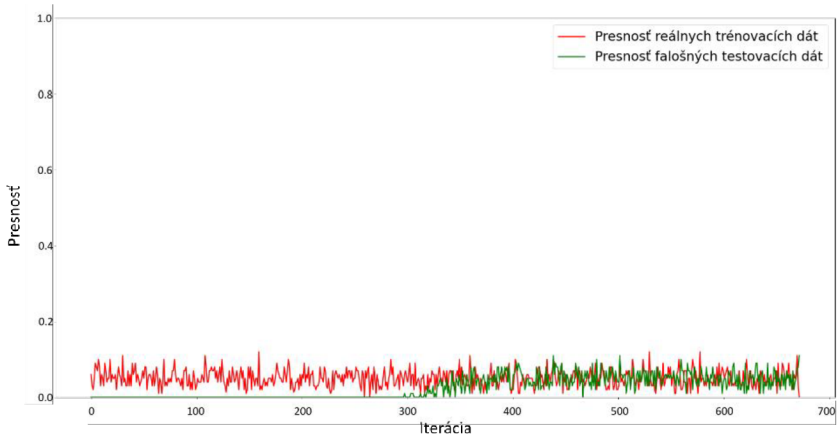
V tejto kapitole sa pozrieme na možné dôvody prečo náš model GAN zlyhal. Vidíme, že kvalita generovaných signálov je slabá – generátor sa nedokázal naučiť generovať vieruhodné dáta. Taktiež vidíme, že presnosť diskriminátoru je veľmi nízka – diskriminátor nedokázal dobre rozlišovať medzi falošnými a reálnymi dátami.

Na Obrázku 5.17, vidíme že chyba generátoru aj diskriminátoru postupne klesala. Oscilácie oboch "hráčov" sú minimálne – trénovanie neprebiehalo optimálne. Podľa skúseností získaných v tejto diplomovej práci odhadujeme, že hlavnou príčinou chyby by mohla byť nedostatočne hlboká architektúra sietí. Výsledky taktiež nemôžme dostatočne analyzovať, pretože sme boli limitovaný výpočtovou náročnosťou použitej platformy a nedokázali model trénovať po dostatočne dlhú dobu.



Obr. 5.17: Vývoj chýb diskriminátoru a generátoru počas učenia modelu GAN pre generovanie EKG signálov. Chyba diskriminátoru bola počítaná ako súčet chýb diskriminátoru na reálnych aj falošných dátach.

Na druhom Obrázku 5.18, vidíme, že diskriminátor dosahoval veľmi nízkych hodnôt presností – pravdepodobne kvôli nedostatočnej hĺbke použitej architektúry, poprípade parametrov konvolučných či Max Pooling vrstiev.



Obr. 5.18: Vývoj presností diskriminátoru na reálnych a falošných dátach počas trénovania modelu GAN pre generovanie EKG signálov.

Pre porovnanie v článku [7], model trénovali celkovo po dobu 500 epôch, nám sa podarilo modeli trénovať len 50 epôch a z tohto dôvodu sme používali vyššie hodnoty parametru rýchlosti učenia. Napriek tomu predpokladáme, že architektúra v [7] nie je vhodná na generovanie realisticky vyzerajúcich EKG signálov.

6 DISKUSIA

Generovanie EKG záznamov pomocou GAN je veľmi mladá a perspektívna metóda. Pretrvávajúci problém nedostatku EKG signálov pre trénovanie rôznych algoritmov je riešiteľný – metóda použitia GAN ukazuje nádejné výsledky. Tento problém si však ešte vyžaduje ďalší výskum, nakoľko nedosahuje prakticky použiteľné výsledky.

Pre generovanie arytmických signálov potrebujeme veľkú databázu signálov, ktorá zatiaľ nie je k dispozícií. Preto sa metóda generovania signálov EKG zameriava na sínusové signály, ktoré sú dostupné vo väčších počtoch ako arytmické signály. Zistili sme, že trénovanie GAN je časovo náročné a vyžaduje si veľú výpočetnú náročnosť. Z tohto dôvodu je optimalizácia GAN modelov veľmi náročnou úlohou. Napriek tomu predpokladáme, že výskum tejto metódy má zmysel.

Na jednoduchých 1D signáloch sme si ukázali, že modely GAN sú schopné generovať reálne vyzerajúce signály. Taktiež sme zistili, že pri trénovaní 1D signálov, GAN najrýchlejšie rozpoznával extrémy funkcií. Ukázali sme, že diverznosť vstupných dát má pozitívny vplyv na stabilizáciu trénovacieho procesu GAN. Dokázali sme, že pomocou kriviek chýb generátoru a diskriminátoru dokážeme analyzovať správanie GAN. Ukázalo sa, že pochopenie a správna interpretácia zlyhaní modelov je kľúčovým faktorom pre optimalizáciu daného modelu.

Vytvorili sme niekoľko použiteľných architektúr pre trénovanie modelov GAN, začínajúc s jednoduchými modelmi až po model pre generovanie EKG signálov. Pri navrhovaní architektúr je vhodné začať s menším modelom a postupne ho zväčšovať, aby sme dokázali lepšie pochopiť a analyzovať tak komplexnú architektúru akou je GAN.

Implementovali sme kompletnú štruktúru algoritmov pre trénovanie GAN. Zistili sme, že najťažšou časťou návrhu modelov je samotná realizácia architektúr a trénovacieho procesu GAN. Po zostavení funkčnej štruktúry trénovania a vyhodnocovania GAN dokážeme modely správne a vedecky analyzovať. Dôležitým faktorom je sledovanie vývoja chýb modelu a presnosti diskriminátoru. Následne sme dospeli k záveru, že na efektívne optimalizovanie modelov GAN je nutnosťou disponovať veľkou výpočetnou silou a prehľadným ukladaním parametrov.

Opravili sme tvrdenia autorov [7] - konvergencia chýb modelu k nule nie je pozitívum ale znak zlyhania modelu. V práci sme podrobne popísali dôvody zlyhania modelov a v praktickej časti navrhli a ukázali spôsoby ich detekcie. Tvrdíme, že architektúra spolu s použitými parametrami v práci [7], nie je vhodná na generovanie EKG signálov. Navrhujeme použiť hlbší model siete – napríklad väčší počet konvolučných vrstiev.

Použitie Leaky ReLU vrstvy v diskriminátore modelov malo dobré výsledky, naopak použitie Leaky ReLU vrstvy v generátore znižovalo amplitúdu generovaných signálov – konštatujeme, že použitie Leaky ReLU nie je vhodné pre architektúry

generátorov pre generovanie EKG signálov. Zvýšenie dimenzie latentného priestoru na hodnotu 100 dosahovalo lepších vizálnych výsledkov ako pri hodnote 5, avšak toto tvrdenie by si vyžadovalo väčší počet realizácií takýchto modelov. Preto toto trvdenie považujeme za heuristický typ, zistený pri učení malého počtu modelov GAN.

Jadrom našej práce bolo zostavenie funkčných modelov GAN. Tieto modely spolu s trénovacím, zálohovacím a evaluačným algoritmom sú kľúčom k začiatku optimalizácie akéhokoľvek modelu GAN. Našu implementovanú štruktúru modelov GAN je možné jednoducho využiť a týmto spôsobom sme prispeli k ďalšiemu výskumu tématiky generovania EKG signálov pomocou GAN. Konštatujeme, že naša práca môže byť po použití iných architektúr pomocným krokom k vyriešeniu problému generovania EKG signálov.

Nakoniec by sme sa chceli zmieniť o zaujímavých postupoch pri generovaní EKG signálov, ktoré využili práce [23, 41, 42]. Tieto postupy by mohli byť ďalším predmetom skúmania tohto problému. V prílohe práce poskytujeme všetky navrhnuté architektúry, spolu s rešeršom modelov GAN pre generovanie časovo závislých dát.

ZÁVER

Začiatok práce teoreticky popisuje základné princípy a vysvetľuje funkciu umelých neurónových sietí. Popisuje aktivačné funkcie, princípy trénovania siete a vysvetľuje fungovanie niekoľkých typov vrstiev neurónových sietí. Ďalej boli vypracované literárne rešerše na tému generatívnych neurónových kompetitívnych sietí, kde bolo popísané ich využitie a princípy fungovania spolu s hlavnými dôvodmi ich zlyhania.

Bola zostavená vlastná databáza EKG signálov vhodných na trénovanie GAN, pomocou signálov poskytnutých vedúcim práce. Na základe odborných publikácií sme navrhli všeobecný postup predspracovania signálov EKG pre trénovanie modelov GAN.

Práca ďalej implementovala tri funkčné modely GAN. Dva modely sa zaoberali generovaním jednoduchých 1D funkcií. Dokázali sme, že modely GAN sú vhodné na generovanie jednoduchších signálov. Augmentácia vstupných dát mala pozitívny vplyv na stabilný tréning modelu GAN. Tretí z modelov sa zaoberal generovaním EKG signálov. Bol popísaný podrobný postup spôsobu a implementácie tréningu modelu GAN. Tento model modifikoval spôsob generovania latentných bodov a tak pomohol generátoru generovať kvalitnejšie výstupy. Zároveň sme v tomto modeli vytvorili funkciu na zálohovanie aktuálnych modelov a ukladanie priebežných chýb modelu spolu s presnosťou diskriminátoru. Ukázali sme, že ukladanie a následné vizualizovanie týchto dát bolo kľúčovým krokom k analyzovaniu správania modelov GAN.

Vyhodnetenie kriviek chýb generátoru a diskriminátoru značne pomohlo s pochopením úspešnej aj neúspešnej funkcie modelov GAN. Potvrdili sme, že analýza chýb modelu je prvým krokom k jeho optimalizácií. Navrhli sme zvýšenie parametru rýchlosti učenia, nakoľko sme boli limitovaný výpočtovou silou.

Poukázali sme na chybné tvrdenia v práci [7], ktoré sme odôvodnene vysvetlili na našich výsledkoch. Architektúra použitá v [7], nie je dostatočne hlboká na generovanie reálne vyzerajúcich EKG signálov. Navrhli sme zvýšiť počet konvolučných vrstiev a uviedli sme niekoľko tipov, ktoré sme nadobudli počas vytvárania tejto práce.

Hlavným prínosom práce boli tri funkčné architektúry modelov GAN, ktorých implemtácie ja podrobne vysvetlená. Tieto modely by mohli byť v budúcnosti použité na implementáciu iných architektúr, ktoré by problém generovania nových syntetických EKG signálov mohli vyriešiť. Dosiahnuté výsledky sme slovne okomentovali a v diskusií, sme načrtli ďalšie smerovanie výskumu metód zaoberajúcich sa generovaním EKG signálov.

LITERATÚRA

[1] TUČKOVÁ, Jana. *Vybrané aplikace umělých neuronových sítí při zpracování signálů*. Praha: České vysoké učení technické v Praze, 2009. ISBN 978-80-01-04229-8.

[2] NOVÁK, Mirko, Josef FABER a Olga KUFUDAKI. *Neuronové sítě a informační systémy živých organismů*. Praha: Grada, 1993. ISBN 80-854-2495-9.

[3] MAŘÍK, Vladimír, Olga ŠTĚPÁNKOVÁ a Jiří LAŽANSKÝ. *Umělá inteligence*. Praha: Academia, 2001. ISBN 80-200-0472-6.

[4] MAŘÍK, Vladimír, Olga ŠTĚPÁNKOVÁ a Jiří LAŽANSKÝ. *Umělá inteligence*. Praha: Academia, 2003. ISBN 80-200-1044-0.

[5] GOODFELLOW, Ian J., et al. Generative Adversarial Nets. *27th Conference on Neural Information Processing Systems (NIPS 2014)* [online]. Dostupné z: https://arxiv.org/pdf/1406.2661.pdf

[7] ZHU, Fei, Fei YE, Yuchen FU, Quan LIU a Bairong SHEN. Electrocardiogram generation with a bidirectional LSTM-CNN generative adversarial network. *Scientific Reports* [online]. 2019, **9**(1) [cit. 2020-01-03]. DOI: 10.1038/s41598-019-42516-z. ISSN 2045-2322. Dostupné z: http://www.nature.com/articles/s41598-019-42516-z

[8] GOODFELLOW, Ian J. NIPS 2016 Tutorial: Generative Adversarial Networks. *30th Conference on Neural Information Processing Systems (NIPS 2016)* [online]. Dostupné z: https://arxiv.org/pdf/1701.00160.pdf

[9] Neuron Design. *PNGio* [online]. Dostupné z: https://pngio.com/

[10] MCCULLOCH, Warren S. a Walter PITTS. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics* [online]. 1943, **5**(4), 115-133 [cit. 2020-01-03]. DOI: 10.1007/BF02478259. ISSN 0007-4985. Dostupné z: https://link.springer.com/article/10.1007/BF02478259#citeas

[11] CHAUHAN, N.S. Introduction to Artificial Neural Networks(ANN). *Towards data science* [online]. 2019. Dostupné z: https://towardsdatascience.com/introduction-to-artificial-neural-networks-ann-1aea15775ef9

[12] SHARMA, Avinash. Understanding Activation Functions in Neural Networks. *Medium* [online]. 2017. Dostupné z: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[13] 7 Types of Neural Network Activation Functions: How to Choose? *MissingLink* [online]. Dostupné z: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

[14] Pfau, David and Oriol VINYALS. *Connecting Generative Adversarial Networks and Actor-Critic Methods* [online]. Cornell University, 2016. Dostupné z: https://arxiv.org/abs/1610.01945

[15] FINN, Chelsea, Paul CHRISTIANO, Pieter ABBEEL and Sergey LEVINE. *A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models* [online]. Cornell University, 2016. Dostupné z: https://arxiv.org/abs/1611.03852

[16] HO, Jonathan and Stefano ERMON. Generative adversarial imitation learning. In Advances in Neural Information Processing Systems. *30th Conference on Neural Information Processing Systems (NIPS 2017)* [online]. 2017, 4565–4573. Dostupné z: https://papers.nips.cc/paper/6391-generative-adversarial-imitation-learning.pdf

[17] SALIMANS, Tim, et al. Improved techniques for training gans. In Advances in Neural Information Processing Systems, pages 2226–2234. *30th Conference on Neural Information Processing Systems (NIPS 2016)* [online]. 2016, 2226–2234. Dostupné z: https://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.pdf

[18] ODENA, Augustus. Semi-supervised learning with generative adversarial networks. *Data Efficient Machine Learning workshop at International Conference on Machine Learning (ICML 2016)* [online]. 2016. Dostupné z: https://arxiv.org/pdf/1606.01583.pdf

[19] LEDIG, Christian, et al. Photo-realistic single image super-resolution using a generative adversarial network. *Computer Vision and Pattern Recognition* [online]. 2016. Dostupné z: https://arxiv.org/pdf/1609.04802.pdf

[20] ZHU, Jun-Yan, Philipp KRÄHENBÜHL, Eli SHECHTMAN a Alexei A. EFROS. Generative Visual Manipulation on the Natural Image Manifold. *Computer Vision – ECCV 2016* [online]. Cham: Springer International Publishing, 2016, 597-613. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-46454-1_36. ISBN 978-3-319-

46453-4. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-319-46454-1_36#citeas

[21] BROCK, A., T. LIM, J.M. RITCHIE, and N. WESTON. Neural photo editing with introspective adversarial networks. *The International Conference on Learning Representations (ICLR 2017)* [online]. 2016. Dostupné z: https://arxiv.org/pdf/1609.07093.pdf

[22] ISOLA, Philip, J.-Y. ZHU, T. ZHOU, and A.A EFROS. Image-to-image translation with conditional adversarial networks. *Computer Vision – ECCV 2016* [online]. 2016. Dostupné z: https://arxiv.org/pdf/1611.07004.pdf

[23] ESTEBAN, Crisóbal, Stephanie L. HYLAND and Gunnar RATSCH. *Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs* [online]. 2017. Dostupné z: https://arxiv.org/pdf/1706.02633.pdf

[24] LOTTER, William, G. KREIMAN, and D. COX. Unsupervised learning of visual structure using predictive generative networks. *The International Conference on Learning Representations (ICLR 2016)* [online]. 2015. Dostupné z: https://arxiv.org/pdf/1511.06380.pdf

[25] LUHANIWAL, Vikashraj. Analyzing different types of activation functions in neural networks — which one to prefer? *Towards data science* [online]. 2019. Dostupné z: https://towardsdatascience.com/analyzing-different-types-of-activation-functions-in-neural-networks-which-one-to-prefer-e11649256209

[26] JAIN, Pawan. Complete Guide of Activation Functions: A practical guide comparing advantages, problems, and solution of activation functions. *Towards data science* [online]. 2019. Dostupné z: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

[27] TCH, Andrew. The mostly complete chart of Neural Networks, explained. *Towards data science* [online]. 2017. Dostupné z: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

[28] DELLINGER, James. Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming. *Towards data science* [online]. 2019. Dostupné z: https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

[29] KOLÁŘ, Radim, Jakub CHMELÍK, Roman JAKUBÍČEK. *Machine learning lectures*. Brno: Ústav Biomedicínskeho Inžinierstva, FEKT, Vysoké učení technické v Brne. 2019. Prezentácia.

[30] BROWNLEE, Jason. How to Choose Loss Functions When Training Deep Learning Neural Networks. *Machine Learning Mastery* [online]. 2019. Dostupné z: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

[31] Backpropagation in Neural Networks: Process, Example & Code. *MissingLink* [online]. Dostupné z: https://missinglink.ai/guides/neural-network-concepts/backpropagation-neural-networks-process-examples-code-minus-math/

[32] HAMPTON, John R. *EKG stručně, jasně, přehledně*. Praha: Grada, 2013. ISBN 978-80-247-4246-5.

[33] Fully Connected Layers in Convolutional Neural Networks: The Complete Guide. *MissingLink* [online]. Dostupné z: https://missinglink.ai/guides/convolutional-neural-networks/fully-connected-layers-convolutional-neural-networks-complete-guide/

[34] HOCHREITER, Sepp a Jürgen SCHMIDHUBER. Long Short-Term Memory. *Neural Computation* [online]. 1997, **9**(8), 1735-1780 [cit. 2020-05-18]. DOI: 10.1162/neco.1997.9.8.1735. ISSN 0899-7667. Dostupné z: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[35] HINDUPUR, Avinash. The GAN Zoo. *GitHub* [online]. 2017. Dostupné z: https://github.com/hindupuravinash/the-gan-zoo

[36] Machine Learning Crash Course: GANs. *Developers Google* [online]. 2018. Dostupné z: https://developers.google.com/machine-learning/gan/problems

[37] MEHRALIAN, Mehran, Babak KARASFI and Donglai XU. RDCGAN: Unsupervised Representation Learning With Regularized Deep Convolutional Generative Adversarial Networks. *2018 9th Conference on Artificial Intelligence and Robotics and 2nd Asia-Pacific International Symposium* [online].. IEEE, 2018, 31-38. DOI: 10.1109/AIAR.2018.8769811. ISBN 978-1-7281-2842-9. Dostupné také z: https://ieeexplore.ieee.org/document/8769811/

[38] LIU, Ming-Yu and Oncel TUZEL. Coupled Generative Adversarial Networks. *29th Conference on Neural Information Processing Systems (NIPS 2016)* [online]. 2016. Dostupné z: https://arxiv.org/abs/1606.07536.

[39] KARRAS, Tero, Timo Aila, Samuli Laine, Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *International Conference on Learning Representations (ICLR 2018)* [online]. 2017. Dostupné z: https://arxiv.org/abs/1710.10196

[40] KARRAS, Tero, Samuli LAINE a Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks: Unsupervised Representation Learning With Regularized Deep Convolutional Generative Adversarial Networks. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* [online]. IEEE, 2019, 4396-4405.
DOI: 10.1109/CVPR.2019.00453. ISBN 978-1-7281-3293-8. Dostupné z: https://ieeexplore.ieee.org/document/8953766/

[41] DELANEY, Anne-Marie, Eion BROPHY and Tomás E. WARD. *Synthesis of realistic ECG using generative adversarial networks* [online]. 2019. Dostupné z: https://arxiv.org/abs/1909.09150

[42] BROPHY, Eion, Zhengwei WANG and Tomás E. WARD. *Quick and Easy Time Series Generation with Established Image-based GANs* [online]. 2019. Dostupné z: https://arxiv.org/abs/1902.05624v3

[43] GOODFELLOW, Ian, Yoshua BENGIO and Aaron Courville. *Deep learning* [online]. Cambridge: MIT press, 2016. Dostupné z: https://www.deeplearningbook.org/

[44] WHITE Tom. Sampling Generative Networks. *29th Conference on Neural Information Processing Systems (NIPS 2016)* [online]. 2016. Dostupné z: https://arxiv.org/abs/1609.04468

[45] ARJOVSKY, Martin and Léon Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *International Conference on Learning Representations (ICLR 2017)* [online]. 2017. Dostupné z: https://arxiv.org/abs/1701.04862

[46] ARJOVSKY, Martin, Soumith Chintala a Léon Bottou. *Wasserstein GAN* [online]. 2017. Dostupné z: https://arxiv.org/abs/1701.07875

[47] METZ, Luke, Ben POOLE, David PFAU, Jascha SOHL-DICKSTEIN. Unrolled Generative Adversarial Networks. *International Conference on Learning Representations (ICLR 2017)* [online]. 2017. Dostupné z: https://arxiv.org/abs/1611.02163

[48] CHINTALA, Soumith, Emily DENTON, Martin Arjovsky, Michael MATHIEU. How to Train a GAN? Tips and tricks to make GANs work. *Github.com* [online]. 2020. Dostupné z: https://github.com/soumith/ganhacks

[49] BROWNLEE, Jason. How to Develop a 1D Generative Adversarial Network From Scratch in Keras. *Machine Learning Mastery* [online]. 2019. Dostupné z: https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-1-dimensional-function-from-scratch-in-keras/