FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Kiss Andor

ZENESZERZÉS GÉPI TANULÁS HASZNÁLATÁVAL

KONZULENS

Dr. Szegletes Luca

BUDAPEST, 2021

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Kiss Andor**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 10. 04.

...…….
Kiss Andor

Összefoglaló

abstract magyarul

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül. Ez a magyar nyelvű összefoglaló angolra fordított változata.

1 Bevezetés

A zeneszerzés az őskor óta jelen van az emberiség életében. Ahogy múlik az idő, úgy jelennek meg ott is egyre modernebb technológiák. Kezdetben megjelentek akusztikus hangszerek, például a hárfa, később a lant. Az elektronika megjelenését nem sokkal később követték az elektromos hangszerek is. Egy ilyen híres, még ma is használt elektromos hangszer a szintetizátor volt, ami alapból zenei hullámformákat volt képes generálni különböző célhardverek, áramkörök segítségével. Később a szintetizátorok már egymással, és a számítógépekkel is képesek voltak kommunikálni, a Musical Instrument Digital Interface (MIDI) protokoll segítségével. Ahogy észrevehető ebből a rövid történelmi áttekintőből, a technológia fejlődése, a gépesítés a hangszerekre is nagy hatással volt. Ma már számos olyan számítógépes programot lehet használni, ami zeneszerzésre alkalmas, különféle hangszerek hangjait képes lejátszani, és a felhasználó kezébe adja a kontrollt, aki kreativitása segítségével alkothat, viszont itt még mindig a felhasználó, az ember a gép mögött az igazi készítő, ő választja ki a megfelelő beállításokat, a hangokat és időtartamokat, amikből összeáll a zene.

A gépi tanulás már az 1950-es években is létezett, viszont zeneszerzésre akkor még nem használták. Akkor kezdtek el ezzel foglalkozni, amikor a 2010-es években a mély tanulás (deep learning) technológia robbanásszerű fejlődésen ment keresztül. Ma már lehetséges zenét létrehozni úgy, hogy nem egy ember által szerkesztett zenét módosítanak szoftverek segítségével, hanem az alapot is teljes mértékben a szoftver adja. Természetesen egyéb szoftverek vagy az emberi tényező is megjelenik a procedúrában, például a zene utólagos feldolgozásánál, effektezésénél. Emellett azt sem szabad elfelejteni, hogy a tanító adathalmazt is emberek hozták létre, hiszen a gépi tanuló algoritmus se a semmiből szerzi ismereteit.

A zene, a hangszerek és a gépi tanulás fejlődésének érdekes párhuzama mellett személyes motivációm is van, ami miatt ezt a szakdolgozat témát választottam. Több éve gitározom, elektromos gitáron is, foglalkoztam már zenei szoftverekkel is. Mind a zene, mind a gépi tanulás világát nagyon érdekesnek tartom, így azt tűztem ki magamnak feladatként, hogy körüljárjam a témában rejlő lehetőségeket, feltérképezzem a zene számítógépes formátumait, és természetesen azt, hogy hogyan lehet generatív modellekkel létrehozni azokat.

Egy másik személyes kötődésem teszi érdekessé még ezt a projektet. A legtöbb korábbi gépi tanulás alapú zeneszerzési megoldás klasszikus zenékkel foglalkozik, azon belül is zongoraművekkel. Én nem ezzel szeretnék, hanem az egyik kedvenc előadómmal, az Iron Maidennel. Az ő zenéjük főleg gitáron és basszusgitáron van játszva, természetesen a ritmust adó dobokkal, és ritkán egyéb hangszerekkel, például szintetizátorral kiegészítve. Vokál is van szinte minden zeneszámuknál, viszont én azzal ebben a projektben nem foglalkozom, inkább a hangszerekre helyezem a fókuszt. Egy másik dolog, ami az Iron Maiden mellett szól, hogy jellegzetes, kicsit repetitív stílusuk van, ami szerintem egy gépi tanuló algoritmusnak egy könnyedén megérthető, tanulható.

1.1 A dolgozat felépítése

A következő fejezetben a szakdolgozatom elméleti háttere olvasható. Ez az alkalmazott gépi tanuló modelleim alapjait írja le, hogy milyen tudásból építkeztem azok megalkotásakor, illetve, hogy milyen korábbi megvalósítások voltak a gépi tanulás alapú zeneszerzés világában.

A technikai háttér fejezetben azokat fájlformátumokat, szoftvercsomagokat írom le, amik használata lehetővé tette azt, hogy az elméletben megalkotott, megtervezett ötleteimet a gyakorlatban meg is tudjam valósítani.

A megvalósítás fejezetben írom le részletesen ezt a gyakorlati megvalósítást, lépésekre bontva. A tanítóadathalmaz beszerzésétől kezdve, a generált zenéim képernyőre kerüléséig végigmegyek a lépéseken.

Az eredmények fejezetben összevetem különféle gépi tanuló modelleim generált zenéit. A zenéket több módon is kiértékelem, próbálok objektív metrikákat is megfogalmazni a zene jóságára, és szubjektíven, emberi fülek megítélése alapján is végzek értékelést.

Az utolsó fejezetben pedig összefoglalom, hogy miket értem el a projektem során, és hogy a jövőben hogyan lehetne még fejleszteni azon.

2 Elméleti háttér

2.1 Korábbi megoldások

2.1.1 Markov-lánc

A modern deep learning technológiába való betekintés előtt megnéztem, hogyan lehetne egyszerűbb, valószínűség alapú gépi tanulás segítségével zenét szerezni. Ehhez az elsőrendű Markov-láncokat használtam, aminek definíciója így hangzik:

Legyen S egy állapottér, amire igaz, hogy:

$$X_1, X_2, X_3, \dots, X_n, X_{n+1} \epsilon S$$

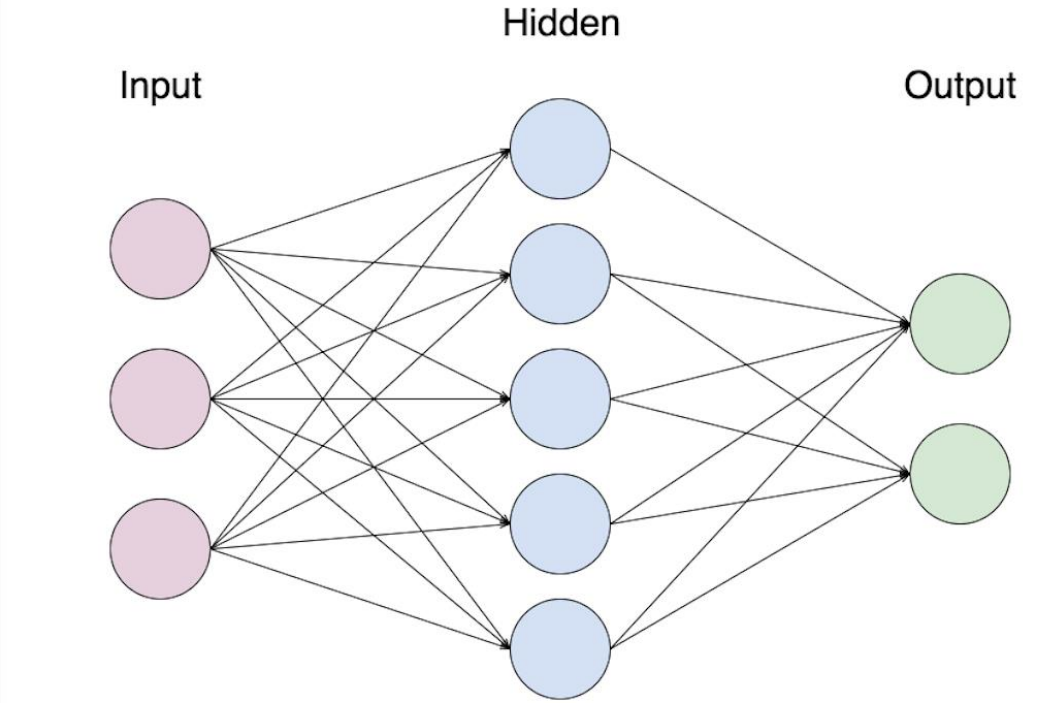
$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = P(X_{n+1} = x | X_n = x_n)$$

Ekkor egy elsőrendű Markov-láncot definiáltunk. Az elsőrendű Markov lánc egy olyan generatív modell, ahol minden állapot csak az azt megelőzőtől függ. Az átmenetek valószínűségét az úgynevezett átmenetvalószínűség mátrix határozza meg. Ez a mátrix meghatározza, hogy egy adott állapotból egy másikba mekkora valószínűséggel léphetünk át. Ha eszerint a mátrix szerint az állapotokon sorban lépkedünk, akkor létre tudunk hozni egy állapotsorozatot.

Ezt zenei alkalmazásra úgy tudnám lefordítani, hogy minden hangot egy állapot jelképez, és minden i-edik hang csak az őt megelőző, i-1-edik hangtól függ. Amikor állapotsorozatot generálok, az a zene hangjait fogja tartalmazni.

2.1.2 Az elemi neuron és a neurális hálózat

A gépi tanulás mai legnépszerűbb algoritmusának, a deep learningnek az alapvető építőeleme az elemi neuron. A neuron egy n hosszú vektort vár bemenetként, emellett egy n hosszú súlyvektorral, egy aktivációs függvénnyel és egy skalár kimenettel rendelkezik. A bemenetén kapott n hosszú vektort a súlyaival skalárisan összeszorozza, majd kimeneteként kiadja ezen skalárszorzat aktivációs függvényén átvezetett, „aktivált” értékét. Ezeket a neuronokat ritkán használják önmagukban, neurális hálózatokat szoktak belőlük alkotni, amik több, rétegbe szedett, összekapcsolt neuronból állnak.

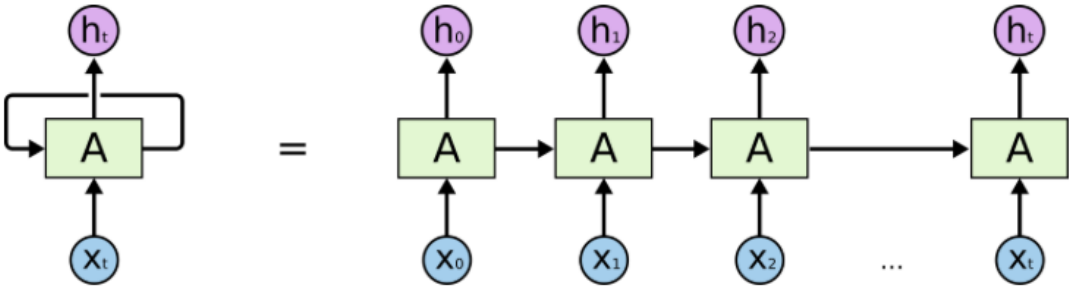


2.1. ábra A mesterséges neurális hálózat [1]

A neurális hálózatokban a bemeneti rétegbe kerül a bemenet, majd továbbításra kerül a következő rétegbe, a réteg súlyaival összeszorozva. Mivel itt már nem csak 1 neuron kapja a bemenetet, hanem több, ezért a súlyvektor helyett már súlymátrixról beszélhetünk, jelen esetben egy 3×5 -ös mátrixról van szó, mivel a bemeneti rétegben 2 neuronból mennek az adatok a következő réteg 5 neuronjába. A bemeneti adatok összeszorzódnak az első súlymátrixszal, majd az ennek eredményeként kialakult mátrix értékei a réteg aktivációs függvényével aktiválva lesznek, és ez az érték megy tovább a következő rétegekbe. A háló ezután úgy tanul, hogy a kimeneti rétegén lévő értéket tanítás során összehasonlítják az elvárt értékkel, majd az ezek között lévő különbség függvényében a hibavisszaterjesztés (backpropagation) algoritmus módosítja a rétegek súlyait, keresve az optimális értékeket. Ahogy ez az érték egyre optimálisabb lesz, úgy fog egyre jobb, az elvárthoz közelebbi értékeket adni a háló.

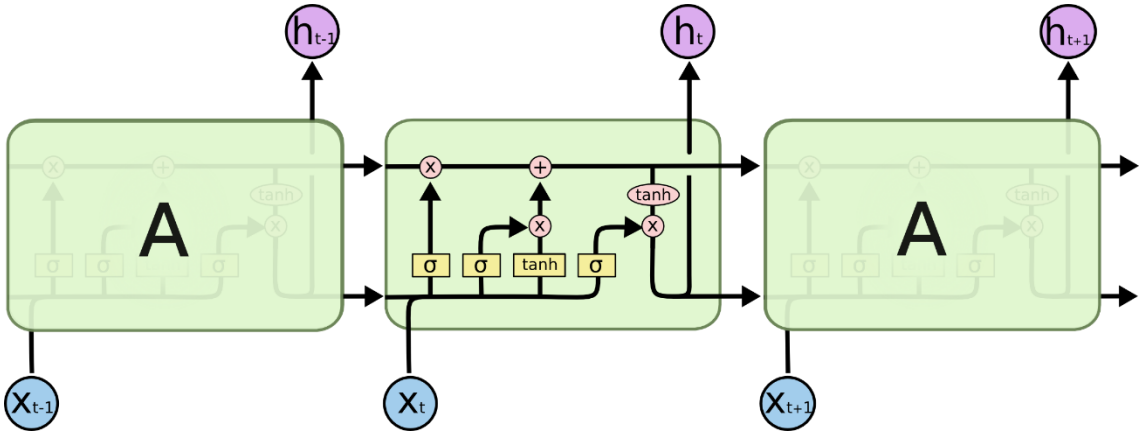
2.1.3 Rekurrens neurális hálók

A sima neurális hálók kimenetei csak az adott pillanatbeli bemenettől függnek, így időbeli szekvenciák, például zenék modellezéséhez nem valami jók. Ezt a problémát orvosolják a rekurrens neurális hálók (RNN), amik olyan konstrukciók, hogy kimenetük a korábbi bementeiktől is függ. [2]



2.2. ábra A kibontott RNN

Ha kibontunk egy ilyen rekurrens hálót, akkor egy Markov-lánchoz hasonló objektumot láthatunk. x_t időbeli kiterjedéssel rendelkező bemeneti vektor feldolgozásakor az i időpillanatbeli kimenet nem csak az i időpillanatbeli bemenettől függ, hanem a korábbi kimenetektől is. A probléma ezekkel, hogy nem tudnak hoszútávú függőségeket fedolgozni, mivel a Markov-lánchoz hasonlóan csak az eggyel korábbi időpillanatból kapnak plusz információt, így hosszú távon elenyészik a jóval korábbi információ. Ezt a problémát hivatott orvosolni a Long Short-Term Memory (LSTM) [3] hálózat.



2.3. ábra Az LSTM cella

Az LSTM fölső vonalán a sima RNN-hez hasonlóan továbbmennek az adatok, viszont történnek rajtuk változások, amik elősegítik a hosszútávú függőségek feldolgozását. Az LSTM több kapuból áll, amik ezekért a változásokért felelősek. Az első kapu, az input gate egy szigmoid függvénnyel aktivált neuron, ami azt határozza meg, hogy a korábbi LSTM cellából érkező információ mennyire maradjon meg. A szigmoid 0 és 1 közötti kimenettel rendelkezik, amivel összeszorozva az információt, dől el, hogy mennyi maradjon abból meg. A 0 jelenti, hogy teljesen dobjuk el azt, az 1 pedig, hogy teljesen tartsuk meg. A cella belső állapotában lévő információt a súlyozott bemenet szigmoiddal

és tangens hiperbolikusszal (tanh) aktivált szorzata adja, ami aztán a fölső vonalon lévő adathoz adódik hozzá. A szigmoid előtti súlyok a forget gate súlyai, a tanh előtti súlyok pedig az LSTM cella saját súlyai. Végül a jelenlegi kimenetet kell eldöntenünk. Ehhez szintén egy kaput használunk, az output gatet, ami egy újabb szigmoid segítségével dönti el, hogy a fölső vonalon lévő adatból mennyi maradjon meg. Természetesen ennek a kapunak is vannak súlyai, így az LSTM-ről elmondható, hogy 4 súlymátrixszal rendelkezik, amik mind más-más cél érdekében tanulnak, így nehezebben taníthatók, több számítási kapacitást igényelnek, mint a sima RNN-ek, viszont jobban tudják modellezni a hosszú távú függőségeket. A rekurrens hálók tanítása során egy módosított hibavisszaterjesztés algoritmust szokás használni. Ez az időbeli hibavisszaterjesztés (backpropagation through time), ami a korábbi időpillanatbeli állapotokra úgy számolja ki a súlyváltoztatást, mintha azok sima rétegek lennének.

2.1.4 A figyelem mechanizmusa

Az LSTM ugyan képes már hosszabb időbeli kapcsolatokat értelmezni, viszont az sem volt elég erős ahhoz, hogy például egy hosszú szövegben meg tudja mondani, hogy egyes szavak fontosabbak másikaknál. Nyelvfordítási feladatoknál kezdetben seq2seq modellekkel dolgoztak. Ezek úgy működtek, hogy egy rekurrens enkóder hálózat létrehozott egy kontextusvektort, amivel inicializálták a kimenetet generáló dekóder hálózatot. Hosszabb szövegeknél mivel az LSTM nem tudott hosszú szövegekből elég jó kontextusvektort létrehozni, az attention (figyelem) volt a megoldás. [4]

A bemenetet egy kétoldalú LSTM dolgozta fel. Minden szóhoz az LSTM mindkét oldala egy értéket rendelt, ez volt az adott szóhoz tartozó LSTM cella kimeneti értéke. A két értéket összeadták, ez adott a szónak egy alap értéket. Ezt az értéket aztán az attention segítségével súlyozták, és ez a súlyozott érték volt a kontextusvektor, amivel a dekóder LSTM celláit inicializálni lehetett. A súlyt kiszámoló attention pedig nem más, mint egy újabb neurális háló. Ez a neurális háló sima, teljesen összekötött rétegekből áll, és szoftmax kimenettel rendelkezik. Ez a szoftmax az enkóder LSTM összes állapotának, és a dekóder LSTM jelenlegi állapotának függvényében számolja ki az enkóder jelenlegi állapotának valószínűségét. Ez a valószínűség gyakorlatilag azt mondja meg, hogy mennyire korrelál egymással a bemeneti és kimeneti szöveg 1-1 szava.

Az attention tehát egy olyan mechanizmus, amivel nyelvi fordításra tanított modellek tudják jobban megtanulni, hogy melyik bemeneti szó melyik kimeneti szóval

hozható leginkább kapcsolatba, így könnyítve a tanítást. Ez viszont csak a kezdet, később arra is rájöttek, hogy az attentionhoz nem kell egy dekóder LSTM, amit figyelembe veszünk. Használható szimplán arra is, hogy megtanítsuk neki, hogy egy mondatban egy szó mennyire korrelál az azt megelőzőekkel, ezt hívják self-attentionnek. [5] Így nem csak fordításra használható ez a mechanizmus, hanem a nyelv megértésére is. Ez már önmagában is hasonlít a rekurrens hálókra, ahol szintén a korábbi állapotok függvényében van a jelenlegi időpillanatbeli bemenet feldolgozva. Ez inspirálta a transformer [6] architektúrát. Ez az architektúra rekurrens hálók alkalmazása nélkül valósította meg a seq2seq modellezést, kizárólag a self-attentiont alkalmazva. A transformerek alapvető építőkövei a Multi-Head Attention rétegek, amik a bemenetükön többszörösen hajtják végre az attention mechanizmust. Ezekből vannak felépítve az enkóder és dekóder hálózatok, amik végül elvégzik a fordítást.

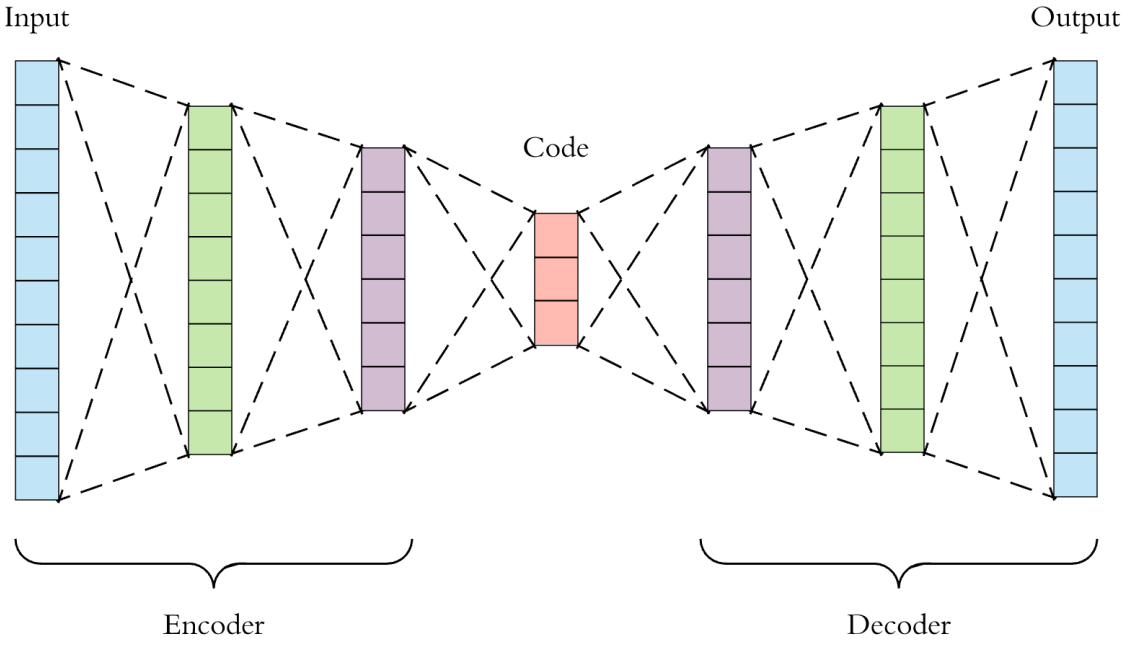
2.1.5 GPT-2

Egy transformer modell nem csak fordításra használható. A self-attention rétegek képesek a szöveg feldolgozására, és megértésére. Ezért egy ilyen modell tanítható úgy, hogy először megtanítunk neki egy adott nyelvet felügyelet nélkül, amit úgy érünk el, hogy az attention rétegei megpróbálják a bemeneti szövegkorpusz minden szavát megbecsülni az azt megelőzőek ismeretében. Ezzel a modell megérti magát a nyelvet, és utána használható lesz célzott feladatokra, például nyelvezet alapján érzelemdetekcióra. Az érzelemdetekcióhoz, vagy egyéb feladatokhoz címkézett adatok állnak rendelkezésre, így a modell finomhangolni tudja a súlyait az elvárt kimenet függvényében. Mivel itt a célnak megfelelő címkézett adataink vannak, ezért ezt felügyelt tanulásnak hívják. Az előtanításnál viszont csak a szövegkorpusz állt rendelkezésre, ott címkék, és külön kiszemelt cél nem volt, csak a nyelv megértése, az egy felügyelet nélküli tanulás volt. Ennek az az előnye, hogy a címkézetlen adatokhoz sokkal egyszerűbb hozzáférni, mint a címkézettekhez, így az előtanításhoz nagyon egyszerű sok adatot felhalmozni, majd a nyelvet már ismerő modellt kevés címkézett adattal is lehet finomhangolni. Ezt a felügyelet nélküli előtanítási technikát alkalmazta az OpenAI csapata 2018-ban egy transformer architektúrájú modellen, így alkották meg a Generative Pre-Trained Transformer (GPT) [7] architektúrát, amivel számos state-of-the-art modellnél jobb eredményeket tudtak elérni különböző célzott feladatokon. Ennek utódjaként készítették el 2019-ben a GPT-2-t [8], ami a sima GPT sokkal nagyobb változata, több, mint 10-szer annyi tanítható paraméterrel, és több, mint 10-szer akkora tanító adathalmazzal. Főleg

szöveggenerálási képességei miatt volt nagy mérföldkő ez a modell. Azóta egyébként a következő iteráció, a GPT-3 [9] is elkészült, még nagyobb adat és paramétermennyiséggel, és még jobb, értékelők szerint emberi szintű szöveggenerálási képességekkel, viszont az még nem érhető el teljes mértékben az interneten, csak egy Application Programming Interface-en (API) keresztül hívható, megadott feladatokra.

2.1.6 Autoencoder hálózatok

Az autoencoder egy olyan neurális hálózat, amely három fontos részből, egy encoderből, egy bottleneckből és egy decoderből áll, és a tanítása során a bemenet és a kimenet megegyezik.



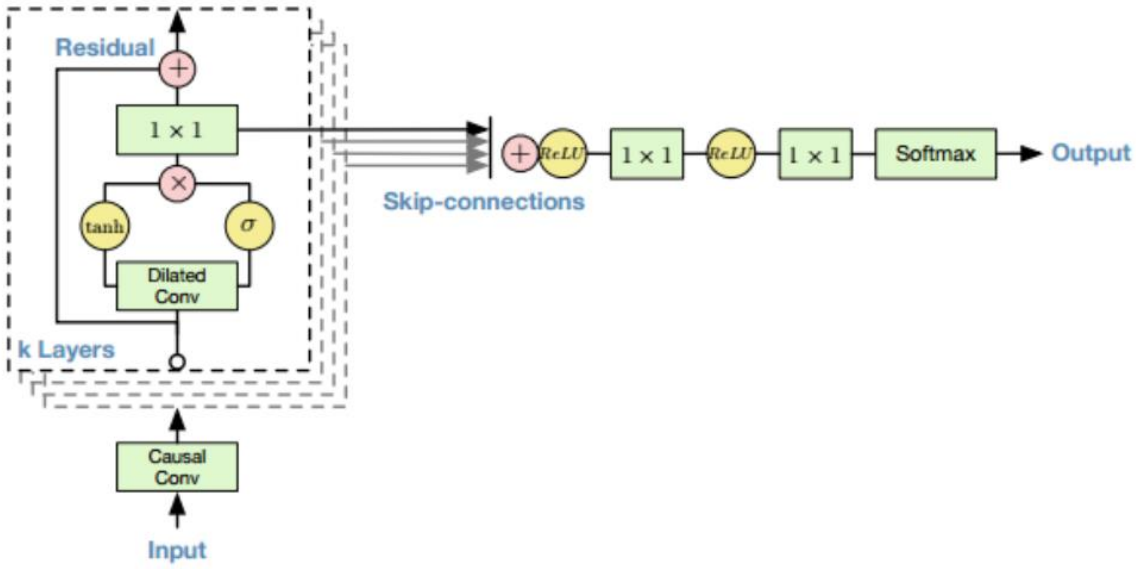
2.4. ábra Az autoencoder [10]

Az alap autencoder háló úgy működik, hogy tanulása során a bemenetre és a kimenetre adott adat megegyezik. A lényeg itt a bottleneck rétegen van, ami a kódot hozza létre. A háló egy olyan kódot próbál generálni, és ehhez úgy optimalizálja súlyait, hogy aztán a kódból vissza tudja állítani az inputján kapott eredeti adatokat. Ezzel lehet például klasszifikációt végezni. Ha a bottleneck rétegre annyi neuront helyezünk el, amennyi osztályba szeretnénk rakni a klasszifikálandó adatokat, akkor a betanult hálót tudjuk úgy használni, hogy az osztályozandó adatot az inputra kötjük, akkor a generált kódból már el is dönthető az adat osztálya.

Az autoencodereknek létezik egy generatív változata is, a Variational AutoEncoder (VAE). Ez annyiban különbözik a sima változattól, hogy a kód itt egy normáleloszlású zaj. Az inputból az enkóder létrehozza ezt a zajt, majd abból a dekóder megpróbálja visszaállítani tanulás során az eredeti adatokat. Generáláskor pedig csak a dekóder részt használjuk, amibe inputként zajt helyezünk el, és az abból kiadott outputja lesz a generált adatunk.

2.1.7 Wavenet

A Wavenet [11] a Google DeepMind csapata által 2016-ban alkotott generatív modell, ami folytonos hullámformájú hangot képes generálni. Ez volt talán az első nagy áttörés a deep learning alapú zenélésben. Text-to-speechre használták nagyrészt, viszont zene generálására is alkalmas.



2.5. ábra A Wavenet

A Wavenet bemenetként egy szeletet vár a folytonos beszédjelből, vagy zenéből, és azt egy 1D konvolúciós rétegekből álló blokkon vezeti át, amik különböző eltolás értékekkel rendelkeznek, így más-más részletességgel néznek rá az bemenetre. Itt a konvolúciós rétegek reziduális összeköttetéseket tartalmaznak, azaz nem a kimenetüket, hanem a kimenetük és bemenetük összegét kötik hozzá a következő réteghez. Ez azért

fontos, mert nagy modelleknél képes előjönni a vanishing gradient probléma, ami miatt a mélyebb rétegek nem tanulnak, ez az összeköttetés viszont ezt képes megoldani. Minden ilyen konvolúciós réteg kimenetét kikötik ebből a nagy blokkból egy összeadó rétegbe. Ezen a szummázott, konvolúciókkal feldolgozott hangjelen még néhány 1D konvolúciós szűrő dolgozik, ReLU aktivációkkal. Végül egy szoftmax aktivációval rendelkező réteg adja a kimenetet. A generált hangokat autoregresszíven a következő inputhoz hozzárakjuk, és így generálja sorban egymás után a hangokat a Wavenet.

2.1.8 SampleRNN

A SampleRNN [12] a Wavenethez hasonlóan diszkrét szoftmax kimenetet használ, viszont lecseréli a konvolúciós rétegeket RNN-ekre, amik jobban tudnak szekvenciákat modellezni, viszont túl sokáig tanulnának, mivel lassabban dolgozzák fel a bemenetet, mint a konvolúciók. Ezt úgy oldják meg, hogy hierarchikus RNN-eket használnak, amik más-más hosszúságú szekvenciákra tekintenek rá, kicsit a Wavenet különböző dilation értékkel rendelkező konvolúcióihoz hasonlóan. Méréseik szerint ez sokkal jobb zenék generálására alkalmas.

2.1.9 Magenta NSynth

Az NSynth [13] neurális háló ötlete újra a Wavenethez nyúl vissza, viszont azt egy még nagyobb, autoencoder struktúrára cseréli le.

2.1.10 Performance RNN

A Performance RNN [14] MIDI-ken dolgozik folytonos zene helyett. Lényegében csak egy többrétegű (stacked) LSTM hálózat, amit a MAESTRO dataseten tanítottak. Ez volt a Google főleg mesterséges zenei kutatással foglalkozó Magenta csapatának első publikált deep learning modellje, ami ezen a fájlformátumon dolgozott.

2.1.11 MusicVAE

A MusicVAE [15] a Google Magenta csapatának MIDI-n tanuló Variational Autoencoder alapú megoldása, amit én is megvalósítottam a paper alapján.

2.1.12 MuseGAN

A MuseGAN [16] egy generatív versengő hálózat [17] (GAN) alapú, MIDI generátor megoldás. Egy érdekes gondolat itt az, hogy többhangszeres zenén tanul, és

mindegyik hangszer saját generátor modellt kap, így egymástól függetlenül módosíthatóak a paramétereik.

2.1.13 Wave2MIDI2Wave

A Wave2MIDI2Wave [18] egy komplex modell, ami folytonos hullámformájú zene létrehozására képes, viszont a generálást MIDI-n végzi. Először a folytonos hullámformájú inputból MIDI-t csinál, majd generál új hangot a MIDI sávhoz, amit visszaalakít folytonos zenévé, és azt adja outputként.

2.1.14 Music Transformer

A Music Transformer [19] a modernebb nyelvi modellekhez hasonlóan, ahogy a nevéből is következik, transformer alapú, azaz attention mechanizmust használ a generáláshoz. Ahogy a nyelvi modelleknél, úgy a MIDI generálásban is a transformer alapú megoldások jelentik a jövőt.

2.1.15 MuseNet

Ahogy a Music Transformer esetében is lehetett látni, a MIDI generálásban átvették az uralmat a transformer alapú megoldások. A MuseNet [20] a GPT-2-höz hasonlóan előtanítást használó transformer architektúrájú modell. Több százezer MIDI fájl alapján ismerte meg a zene különböző műfajait, így célfeladatként képes különféle előadók, műfajok alapján zenét alkotni, akár kombinálni is azokat.

2.1.16 JukeBox

A JukeBox [21] talán a mai legmodernebb, legjobb teljesítményre képes zenegenerátor a deep learning világában. Hatalmas modell, kombinálja a Wavenet 1D konvolúcióit a transformerek attention mechanizmusával.

3 Technológiai háttér

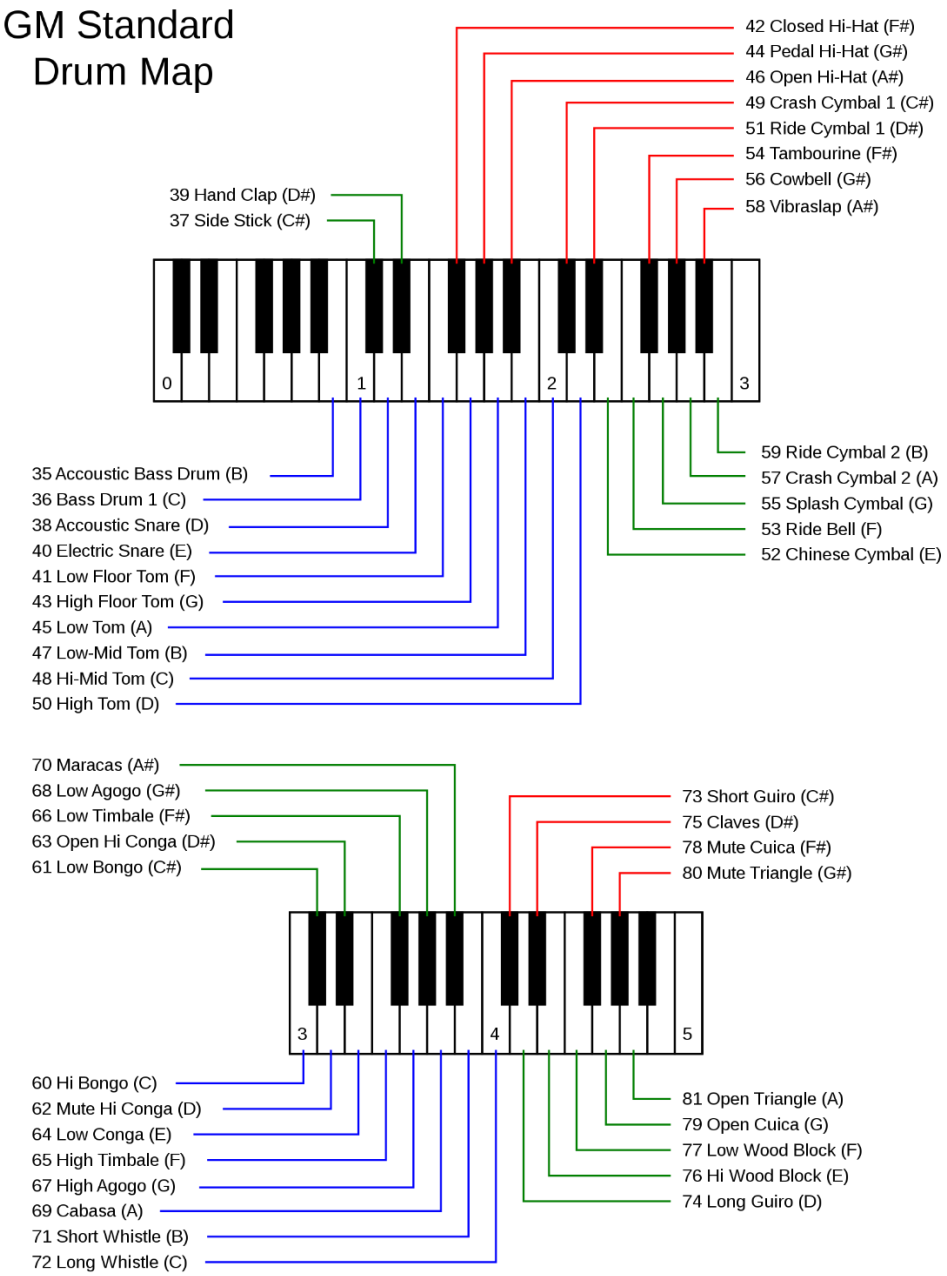
3.1 Musical Instrument Digital Interface

A MIDI [22] egy szabvány, ami több dolgot is definiál számítógépek és különböző elektronikus hangszerek összekapcsolásához. Egy kommunikációs protokollt, egy digitális interfészt és univerzális csatlakozókat, amik segítségével különféle gyártók eszközei is összekapcsolhatók. Emellett egy számomra még fontosabb dolgot definiál, egy zenei fájlformátumot. Ezek a fájlok binárisan tárolják a zenét, viszont nem közvetlenül a hangok amplitúdóját, hanem csak információkat tartalmaznak róluk, ezért például kisebbek is méretben, mint egy folytonos zenei hullámormát tároló fájl. Minden zenei hanghoz le van írva annak magassága, hangereje, lejátszási ideje, és az is, hogy milyen hangszeren kéne azt lejátszani. Minden hang magasságát egy 0 és 127 közötti szám jelöli, az alábbi képen látható módon:

Note	Octave										
	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	**60**	72	84	96	108	120
C#	1	13	25	37	49	**61**	73	85	97	109	121
D	2	14	26	38	50	**62**	74	86	98	110	122
D#	3	15	27	39	51	**63**	75	87	99	111	123
E	4	16	28	40	52	**64**	76	88	100	112	124
F	5	17	29	41	53	**65**	77	89	101	113	125
F#	6	18	30	42	54	**66**	78	90	102	114	126
G	7	19	31	43	55	**67**	79	91	103	115	127
G#	8	20	32	44	56	**68**	80	92	104	116	
A	9	21	33	45	57	**69**	81	93	105	117	
A#	10	22	34	46	58	**70**	82	94	106	118	
B	11	23	35	47	59	**71**	83	95	107	119	

3.1. ábra Hangok és MIDI értékeik

Egy hangszer viszont kilóg a sorból a MIDI fájlok esetén is, méghozzá az ütős hangszerek. Ott nem hangmagassághoz vannak kötve a számok, hanem magukhoz a különböző ütős hangszerekhez.



3.2. ábra A MIDI ütőhangszerek értékei

3.2 Zenei hullámforma

A folytonos hullámformájú zenéket többféle fileformátum is leírhatja. [23] Közülük talán a két legelterjedtebb a .wav és a .mp3 kiterjesztés. A wav egy tömörítetlen bináris formátum, ami pulse code modulation (PCM) algoritmussal feldolgozott analóg hanghullámokat tárol digitálisan. A PCM mintavételezési frekvenciája, és a kvantálásának bitszáma megválasztható természetesen, azok növelése a minőség

javulásával és a méret növekedésével, azok csökkentése pedig minőségromlással, és méretcsökkenéssel jár. A CD minőség 44.1 kHz-es mintavételezési frekvenciát jelent 16 biten ábrázolva. Az mp3 viszont egy veszteségesen tömörített fájlformátum. Nagyon elterjedt, mivel ugyan némi minőségromlás figyelhető meg a tömörítetlen zenéhez képest, 75-95%-os tömörítésre is képes. Egy fontos mérőszám nála a bitráta, ami azt jelenti, hogy másodpercenként hány biten van leírva a zene, ebből legelterjedtebb a 128kbit/s.

3.3 Python

Szakdolgozatom során implementációhoz a Python nyelvet használtam. A Python egy egyszerűen használható, magas szintű, interpretált szkriptnyelv. Dinamikusan típusos, ami azt jelenti, hogy egy adott típusú, például integer változó bármikor kaphat új, más típusú értéket, például stringet, és akkor hiba nélkül fog futni tovább a program, a változónk pedig már string típusú. Ez nagyon kedvező rövidebb, egyszerűbb scriptek megírásához, viszont egy nagyobb kiterjedésű programnál nagyon figyelmesnek kell lenni, hogy ne legyen követhetetlen a változók típusának megváltozása. A nyelv szintaxisa nagyon egyszerű, sok helyen inkább hasonlít az angol nyelvre, mint egy komplex programozási nyelvre. Nem ezek miatt választottam viszont, hanem azért, mert számos remek csomag (package) található hozzá, amiket egy egyszerű csomagtelepítő szoftver, a pip segítségével lehet letölteni.

Projektem során az adatok feldolgozásához a folytonos zenék beolvasására a librosa nevű csomagot használom, ami egy hangelemzéssel foglalkozó csomag. A MIDI formátumú zenékhez ezzel szemben a music21-et használtam, ami az MIT professzorai és hallgatói által fejlesztett csomag, ami hatalmas segítséget ad a felhasználó kezébe a MIDI mellett más hasonló formátumokkal (például MusicXML) leírt zenék elemzéséhez. Egy idő után az adatok elemzésénél eljutok oda, hogy több dimenziós tömbjeim, mátrixaim vannak, amik feldolgozásához, ábrázolásához két nagyon híres csomagot használok, a numpy-t és a matplotlibet. Használok kisebb csomagokat is, például a tqdm-et, ami csak annyit csinál, hogy a ciklusok futásánál a standard outputra egy progressbart jelenít meg, ami jó optimalizációjának köszönhetően kevés overheadet jelent, viszont nagyon hasznos, hogy láthatom a hosszú adatfeldolgozási műveletek jelenlegi állását.

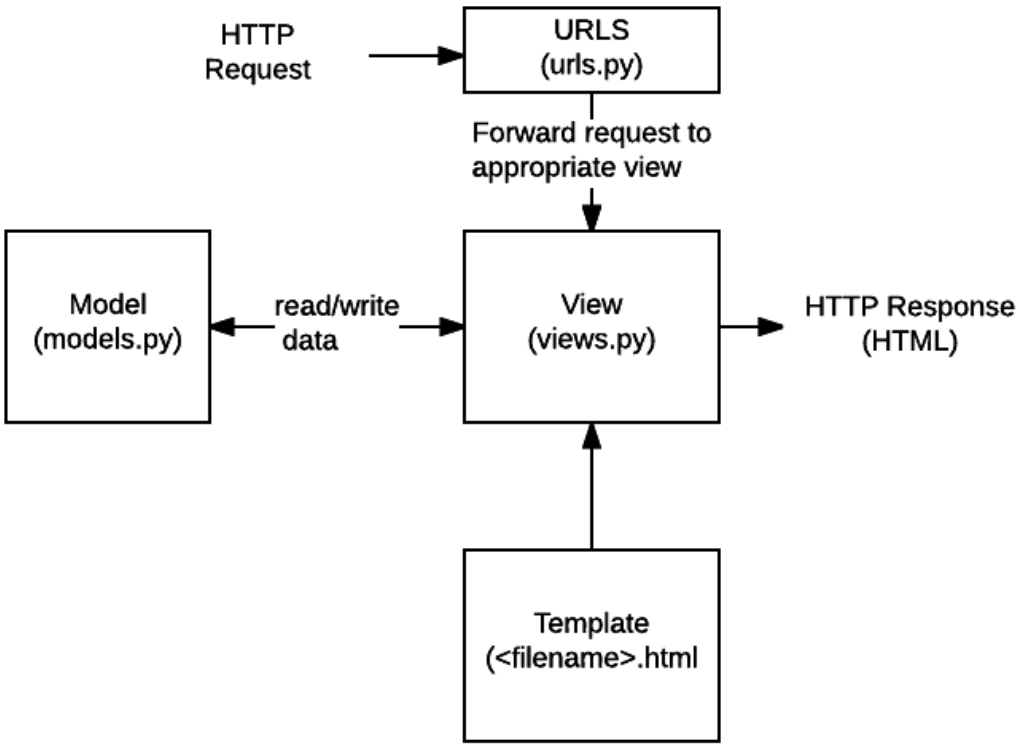
3.4 Keras/TensorFlow

Projektem legfontosabb Python csomagja a Keras, hiszen annak segítségével készítem el a neurális hálóimat. Egy Google által fejlesztett deep learning API, ami lehetővé teszi neurális hálók gyors és egyszerű építését. A Keras korábban egy magasabb szintű wrapper volt többféle alacsonyabb szintű backend fölött, Theano, CNTK, TensorFlow is működhetett alatta, viszont a 2.4-es verzió óta jobban összekapcsolódott a TensorFlow 2-vel, már csak afölött képes működni. Az összekapcsolódás olyan mértékű, hogy az alacsonyabb szintű TensorFlow is már azt tanácsolja, hogy használjuk a Keras beépített magasszintű függvényeit, amennyiben nincs feltétlenül szükségünk az alacsonyabb szintű programozás komolyabb testreszabhatósági lehetőségeire. Számos, fejlesztés során felparaméterezhető előre elkészített réteg és aktivációs függvény található benne, amiket egymás után pakolva lehet neurális hálózatokat létrehozni. A létrehozott hálókat ezután szintén sok előre elkészített költségfüggvény (loss function), optimalizáló algoritmus segítségével lehet tanítani, amik természetesen szintén paraméterezhetőek. Ha még ez sem lenne elég, akkor alaposztályokból leszármazva lehet saját magunknak készíteni a fent említett dolgokat, ezzel elérve a TensorFlow alacsonyabb szintjére. Mivel az API jól és konzisztensen lett megírva, ezek problémamentesen integrálhatók az előre elkészített komponensekből felépített neurális hálókba. A tanítás során callbackeket is használhatunk, amikből szintén lehet készíteni sajátot. Az alap callbackek között van például TensorBoard callback, ami számos hasznos információt jelenít meg a TensorBoard szoftverben az éppen futó tanításról, de ide tartozik az EarlyStopping is, ami egy beépített eszköz a túltanulás elkerülésére, ami azért fontos, hogy maximalizáljuk a modell általánosító képességét. Összességében szerintem az a Keras legnagyobb előnye, hogy amellett, hogy nagyon sok ezközt ad a fejlesztők kezébe, minden lehetőséget megad arra, hogy komplexebb modellek építéséhez belenyúljunk az alacsonyabb szintű TensorFlowba.

3.5 Django

A Django egy Python nyelvű webes keretrendszer, ami szabályrendszerével gyors és skálázható webalkalmazások fejlesztését teszi lehetővé. A keretrendszer nagyon népszerű, és skálázhatóságára bizonyítékot ad, hogy óriás weboldalak, például az Instagram, a Spotify és a YouTube is használják. [24] Mivel az ilyen nagyméretű webalkalmazások adatbázisok használatával szoktak működni, a Django ad ezekhez egy

objektumrelációs leképzést (ORM), hogy megkönnyítse az adatbázishoz való hozzáférést Python kódból. Csak definiálnunk kell a fejlesztés során, hogy milyen motorú adatbázist akarunk használni, például sqliteot, vagy PostgreSQL-t, az ORM egységessé teszi azt számunkra. Az alkalmazások a Model View Template (MVT) architektúrát követik, ami így néz ki:



3.3. ábra Az MVT architektúra [25]

Az alkalmazás először egy HTTP kérést kap, amit a routing modul, az urls.py eloszt a megfelelő nézet számára. A nézet megkapja a hívást, amennyiben voltak paraméterei, azok is átadásra kerültek. A nézet ezután kiszolgálja a kérést úgy, hogy kirenderel egy HyperText Markup Language (html) oldalt, ami egy templateből épül fel, amire adatkötéssel lehet dinamikus adatokat elhelyezni, amik az adatbázisból, vagy amennyiben van, az üzleti logikai rétegből érkeznek.

A modellek reprezentálják az adatbázist. A benne lévő rekordok az ORM-en keresztül Pythonos objektumokként jelennek meg, amiknek az adatait a program futása során változtathatjuk.

3.6 Google Colab

2014-ben a Project Jupyter keretein belül létrehozták a jupyter notebookokat, amik többféle programozási nyelvben, eredetileg Juliában, Pythonban és R-ben történő fejlesztést tesznek lehetővé. Mára már kiegészült például a Haskell és a Ruby nyelvekkel is a rendszer. A név egyébként a JUlia, PYThon, R szavakból is ered. Ezekbe a notebookokba lehet szöveget, és futtatható programkódot is írni. Különlegességük, hogy nem szorosan egymás után fut az összes kód, hanem cellákba lehet rendezni azokat, amik külön futtathatók, és két futás között a memória nem törlődik, a változók értékei megmaradnak. Ez különösen hasznos akkor, amikor egyik cellában lassan lefutó dolgok vannak, például egy adatbetöltés, vagy egy neurális háló tanítás, így nem kell azt a kód többi részének módosítása esetén többször is lefuttatni, így jelentős idő spórolható meg a fejlesztéskor. Ezek a notebookok jsonként mentik el a beléjük írt szöveget és programkódot.

A Google Colaboratory, vagy röviden Colab egy olyan felhőszolgáltatás, aminek segítségével lehet a böngészőben, Colab notebookokban (lényegében kicsit átdolgozott jupyter notebookok) Python kódot írni, és azokat előre konfigurált, GPU-val rendelkező virtuális gépeken futtatni. Minden bejelentkezéskor egy másik virtuális géphez kerülünk, ezért azokon nem maradnak meg az adatok, mint például a Google Cloud Platformnál, viszont van Google Drive integrációja, amivel a saját Drive fiókunkból lehet felhőben tárolt adatokat betölteni a fejlesztési igényeinknek megfelelően. A megírt Colab notebookok is a Drive fiókunkon tárolódnak, így azok nem vesznek el, és könnyen megoszthatók másokkal.

A neurális hálók szinte minden folyamata mátrixműveleteket igényel. Ezekhez van szükség a GPU-kra, mivel sokkal jobb célhardvereként funkcionálnak a sima CPU-nál. A GPU-k eredeti célja a számítógépes grafikában a megjelenítéshez szükséges mátrixműveletek elvégzése volt, így nagyon jól vannak optimalizálva arra. Ezt a funkcionalitást tudják a neurális hálók mögött meghúzódó libraryk is kihasználni a sebesség növelése érdekében.

Projektem során az erős GPU-t igénylő feladatokat így ezen a felhőplatformon végeztem el. Neurális hálóimat itt írtam meg, itt tanítottam, és teszteltem. A betanított modelleket aztán lementettem a Drive fiókomba, hogy onnan bármikor letölthessem, és ne vesszenek el a virtuális gép lekapcsolásakor.

3.7 GitHub

A GitHub-ot azt hiszem minden informatikus ismeri. A Git verziókezelő rendszerre épít, és ahhoz nyújt egy online szolgáltatást, ahol tárolhatók a fájlok, így azok nem csak saját számítógépünkön vannak verziókezelve, hanem más emberekkel is könnyedén megoszthatók egy közös munka során. A verziókezelés egy nagyon fontos dolog számos informatikai projekt során, hiszen esetlegesen elrontott kódok pillanatok alatt visszaállíthatók korábbi állapotukhoz. Az online tárolás egy plusz védelmet is nyújt, hiszen ez a szakdolgozat nem fog elveszni, ha írás közben a számítógépem elromlik, a saját GitHub tárolómból letölthető lesz. További előny még, hogy számos eszköz integrálható a GitHubhoz, amik a szoftverfejlesztést elősegítik. Például, ha egy szoftver rendelkezik tesztekkel, azok a GitHub Actions nevű folytonos integrációt támogató rendszerrel minden pusholáskor (a repositoryba történő mentéskor) lefutnak, és kijelzik az eredményeiket, hogy keletkezett-e hiba, ami miatt nem tudtak lefutni. A verziókezelést elősegíti továbbá az is, hogy a legtöbb fejlesztőkörnyezet ismeri, és használja a Git rendszert, egyszerűen követhető bennük a fejlesztés folyamata.

3.8 Cím2

4 Megvalósítás

4.1 Adatbeszerzés

Kétféle adattal dolgoztam a projektem során, amik más-más reprezentációi a zenének. Az adatbeszerzéshez két megoldást használtam, amelyből az egyik az, hogy a Spotify for developers API segítségével Representational State Transfer (REST) kéréseken keresztül letölthetek 30 másodperces előnézeteket a Spotifyon meghallgatható különféle zenékből. Ezeket a kéréseket Python kódból tudom küldeni, majd válaszként kapok egy mp3 fájlt, amit majd feldolgozhatok később. Így tetszőleges stílusú zenét letölthetek a Spotify hatalmas adatbázisából.

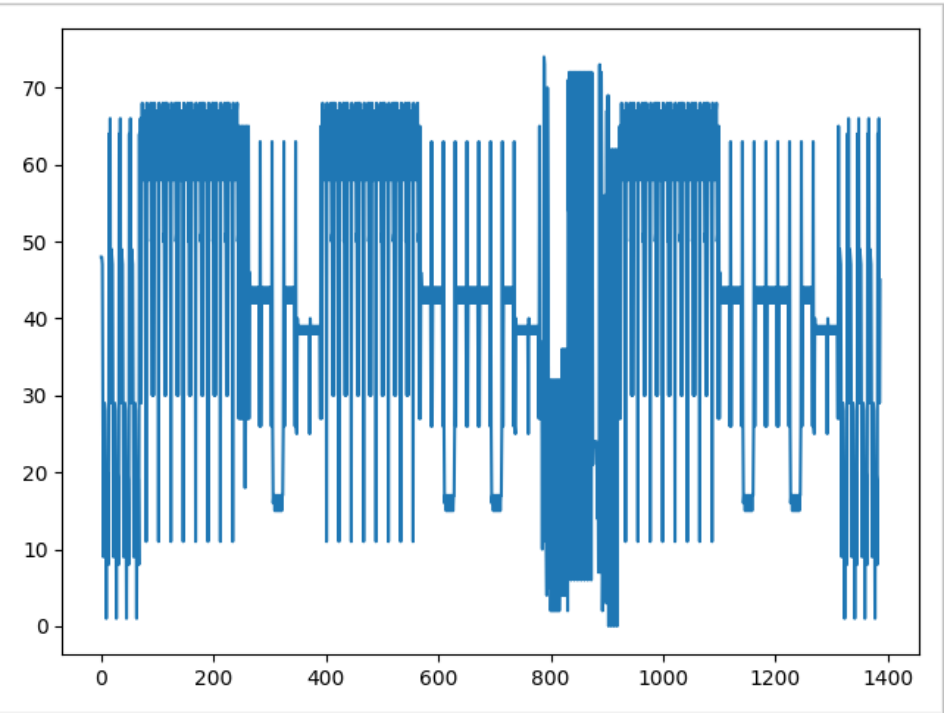
Egy másik megoldás a MAESTRO dataset használata. Ez a Google Magenta csapatának egy adatbázisa, ami ingyenesen letölthető és több, mint 100 GB-nyi klasszikus zenét tartalmaz, zongorán eljátszva. Ennek az adathalmaznak szépsége, hogy nem csak folytonos hullámformaként tartalmazza ezeket a zenéket, hanem MIDI formátumban is, így fel tudom használni projektem másik felén is, ahol MIDI formátumú zenéket használok.

A MAESTRO adathalmazon kívül egy másik MIDI beszerzési módot is találtam, amit viszont az Iron Maiden zenékhez tudok használni. Az Ultimate Guitar oldaláról előfizetők legálisan tölthetnek le Guitar Pro tabokat. A gitártabok egyszerűsített kották, amiket gitárjátékosok szoktak használni, mivel a gitár húrjain mutatják, hogy miket kell lefogni, így egyszerűbben olvashatók, mint a mindenki által ismert zenei kották. A Guitar Pro tabok ezek elektronikus változatai, amik interaktívan lejátszhatók, így még jobban segítik a gitárost az olvasásban és a játékban. Ezeket a fájlokat nem a gitártudásom növelése érdekében töltöttem le, hanem azért, mert ezek egyszerűen konvertálhatók MIDI-vé. A TuxGuitar nevű open-source programot használtam erre a konverzióra.

4.2 Adat előfeldolgozás

4.2.1 MIDI adatok

A MIDI reprezentációjú zenéket a music21 Python csomaggal olvastam be. Különbontottam a különböző hangszerek MIDI sávjait, és úgy mentem végig rajtuk, beolvasva az elemeiket. A beolvasás során Pythonos objektumokat kaptam, amik a music21 csomagban definiált osztályok példányai. Ilyenek például a Chord, Note, Rest, Duration osztályok, amik egy akkordot, egy hangot, egy szünetet, és egy hanghosszt reprezentálnak. A MIDI-k hangerősségével nem foglalkoztam, egyrészt azért, mert az már túl sokféle adat lenne, másrészt azért, mert nem is minden MIDI zenénél kapnak a hangok külön hangerősséget, mivel annyira nem releváns ez az adat. Beolvasás után ezeket az objektumokat számokká alakítottam, mivel a gépi tanuló algoritmusok számokon működnek, nem objektumokon. Pythonos dictionaryk segítségével hoztam létre a beolvasott hangok alapján mappereket, amik átalakítják az objektumokat számokká. Itt eljutottam arra a pontra, hogy egész számokkal leírt MIDI sávjaim vannak. Ezeket már tudtam ábrázolni.

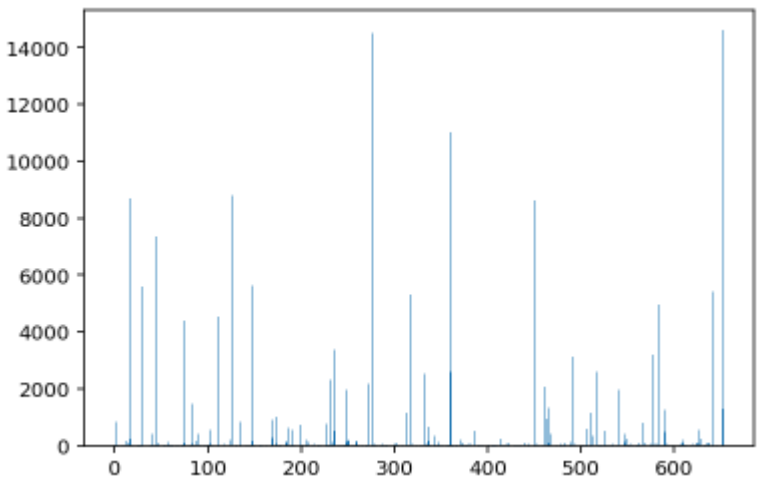


4.1. ábra A feldolgozott MIDI sáv

Így nézett ki egy beolvasott, és számokká alakított MIDI sáv. Ezután több különböző számmá kódolási megoldást is számításba vettem. Ezek között vannak olyan megoldások, amik azért vannak, hogy könnyítsenek a gépi tanuló modell feladatán, csökkentve a kimeneti osztályok számát.

- Egyszerűsítés kedvéért az akkordoknak csak az alaphangját (root note) veszem figyelembe, mivel azok többi hangja általában csak annak szinesítéseként van.
- Foglalkozok akkordokkal és különálló hangokkal is.
- Nem foglalkozok a hangok hosszával, feltételezek egy konstans tempót. Popzenében általában 4/4-es ütem van, viszont az Iron Maiden sajátossága, hogy másfajta ritmikákat (például tripletek) használ.
- Külön kódolom a hangmagasságot és a hosszt, így minden MIDI sávból két egész számokból álló tömböt kapok, egyik a hangok magasságát tartalmazza, másik a hangok hosszát.
- Egyben kódolom a kettőt, így lényegében embeddingeket képzek. Ezáltal egy tömböm lesz, amiben sokféle szám lesz, mivel más számértéket kap például egy negyedes hosszal rendelkező E3 hang, mint egy nyolcados hosszal rendelkező E3.

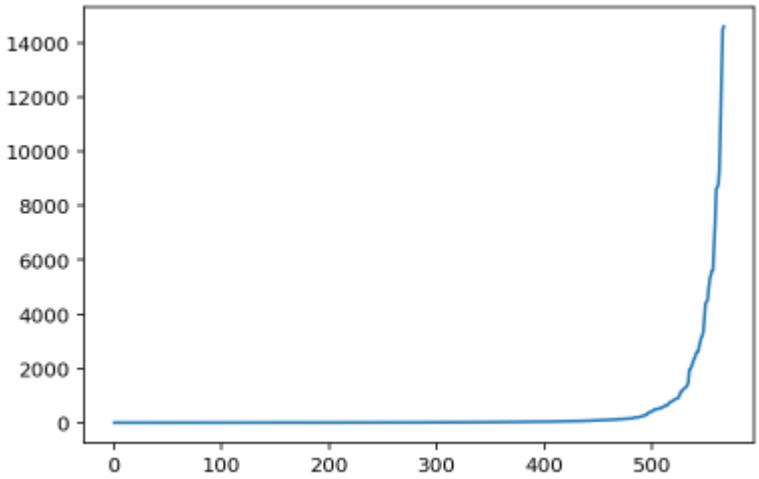
A kódolás után azt vizsgáltam, hogy az adott számértékek hányszor fordulnak elő. Akkordokkal együtt beolvasott MIDI gitársávoknál például így ilyen oszlopdiagramon tudtam ábrázolni ezt.



4.2. ábra A MIDI hangok

Látható, hogy vannak nagyon kiemelkedő vonalak, akár 14000 előfordulási számmal, viszont a 600-nál több adatból soknak szinte nem is látható az előfordulási

számát jelző oszlop, olyan alacsony az. Növekvő sorrendbe rendezve ezeket az értékeket, meg tudtam tekinteni, hogy milyen függvényhez hasonló ezen értékek eloszlása.



4.3. ábra Sorbarakott MIDI hangok

A képen egy exponenciálishoz hasonló függvény képe jelenik meg. Ez azt jelenti, hogy az értékek nagy részének előfordulási száma a zeneszámokban elenyésző. Azt állapítottam meg, hogy ezek kiugró értékek, outlierek, ezeket el lehet dobni a tanítás egyszerűsítése érdekében. Természetesen a zenét minden hang érdekesen tudja színesíteni, viszont, ha egy adott akkord a 93 feldolgozott dal során egy adott értéknél kevesebbszer (például 5-nél kevesebbszer) fordul elő, akkor szerintem tekinthető irrelevánsnak, kivehető a feldolgozott adatok halmazából, így csökkentve a kimeneti lehetőségek számát, segítve a modell tanítását, a lehető legkisebb áldozat árán.

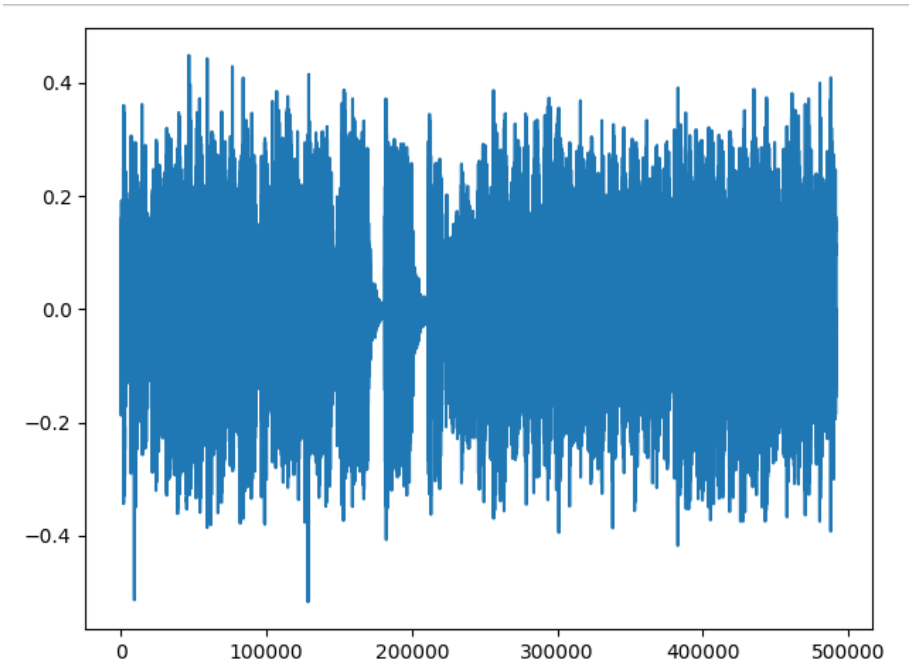
Végül a tanítás további segítése érdekében az adatokat a [0; 1] intervallumba normalizáltam.

A Long-short term memory (LSTM) alapú neurális hálózataim olyanok, hogy egy megadott hosszúságú input zeneszeletből próbálja a háló megjósolni a következő hangot. Például 20 hangból mondja meg a 21.-et. Ehhez létre kellett hoznom egy függvényt, ami felszeleteli az adathalmazomat ilyen párokra. Ez a függvény bemenetként az adathalmazt várja, és a szeletek hosszát, kimenetként pedig olyan input-output párokat ad, ahol az input egy “szelethossz” hosszúságú tömb, ami számokká kódolt MIDI objektumokat tárol, az output pedig az adott szelet után következő, számmá kódolt objektum.

Mivel az LSTM architektúráimban szoftmax kimenetet használok, az output értékeket one-hot kódolom.

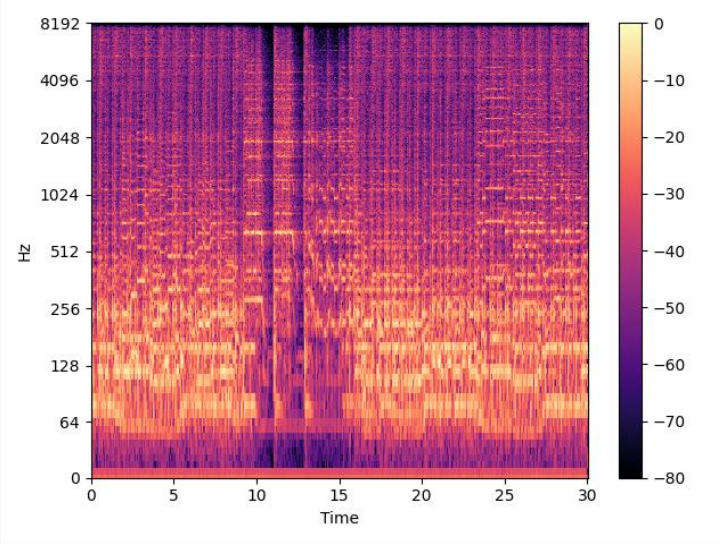
4.2.2 Zenei hullámformák

A folytonos zenéket tároló mp3 fájlokat a librosa nevű Python csomaggal olvastam be. A beolvasásért felelős függvény egy numpy tömbbel tért vissza, ami float típusú valós számokként tárolta az adott időpillanatbeli amplitúdóját a zenének. Egy ilyen tömb elemeit matplotlib segítségével ábrázolva az alábbi időtartománybeli reprezentációt kaptam:



4.4. ábra Zenei hullámforma időtartományban

Ahogy az ábra x tengelyén látható, egy ilyen zeneszám nagyon sok float értékből áll. Ezt megpróbáltam a lehető legalacsonyabb értéken tartani úgy, hogy ne is veszítsek túl sokat a dal minőségéből, alulmintavételezés miatt, viszont ne kelljen a tanításhoz túl sok adattal dolgoznom. Ezért ábrázoltam frekvenciatartományon is a folytonos zenéket, hogy a spektrális tulajdonságaik alapján találhassak egy megfelelő mintavételezési frekvenciát. Az alábbi képen a kiválasztott mintavételezési frekvenciám ábrája található, 16 kHz-t választottam ezen értéknek.



4.5. ábra Zenei hullámforma frekvenciatartományban

Végül ezeket az folytonos értékeket a [-1; 1] intervallumba normalizáltam, mivel a preprocessing pipelineom következő algoritmusa ezen a tartományon várja az értékeket. Ezaz algoritmus pedig a 8 bites μ-law kódolás. Ennek segítségével a végtelen lehetséges valós értéket véges számú értékké alakítom (a 8 bit miatt 255-té), ezáltal kis veszteség árán tudom diszkrét értékekkel reprezentálni a folytonos zenét. Mivel a gépi tanuló algoritmusok a folytonos számokat, és a normalizált értékeket jobban szeretik, mint az egész értékeket, ezért a kódolás után leosztom a maximum értékkel a tömb minden értékét, így az adatok a [0; 1] intervallumba normalizálódnak.

4.3 Adat utófeldolgozás

4.3.1 A szoftmax hőmérséklet

Néhány gépi tanuló algoritmusom önmagában nem képes elég jó zenét generálni, ezért a kimenetét még egy kicsit át kell alakítanom egy másik algoritmussal, ami az utófeldolgozásom első lépéseként szerepel. Ez az algoritmus az LSTM alapú hálózatoknál megjelenő, “beakadás” jelenségét próbálja ellensúlyozni. A “beakadás” azt jelenti, hogy egy input zeneszelet például kizárólag E3 hangokból áll, 20 darabból, ezért a neurális hálózat a következő hangnak is egy E3-mat fog jósolni. Ezután viszont, amikor a következő 20 hangot kapja inputként, ami szintén csak E3-makból áll, mivel a legutolsó

output is az volt, ezért szintén E3-mat fog outputként adni a hálózat. Ez mehetne így a végtelenségig, viszont ebből élvezhető zene nem lenne, mivel azért 1 vég nélkül ismételt hangot nem neveznék annak. Ennek elkerülése végett egy kicsit meg kell ismerni a szoftmax aktivációs függvényt.

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

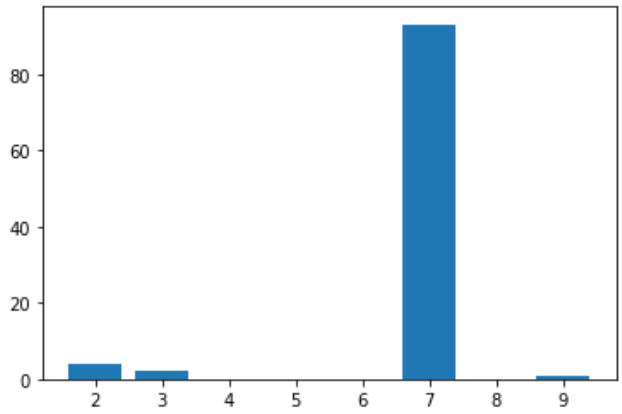
A szoftmax egy exponenciális alapú valószínűségi eloszlás. Amikor a neurális hálózat egy szoftmax aktivációval ellátott rétege outputot ad ki magából, akkor egy ilyesmi tömböt jelent ez:

```
[2.82518557e-30 0.00000000e+00 3.41315882e-07 1.00777954e-11
 1.34826385e-20 1.40432097e-22 0.00000000e+00 9.99999642e-01
 0.00000000e+00 1.88994518e-13]
```

Ha összeadjuk ezeket az értékeket, kijön az 1, tehát ez tényleg egy teljes eseménytér. Amit egy ilyen outputtal gyakran csinálni szoktak, az az, hogy veszik az argmax értékét, ami jelen esetben a 7, mivel a tömböket 0-tól indexeljük, és a 8. értéke majdnem 1. Ez okozza a beakadás jelenségét, mivel fixen mindig a maximum értéket vesszük outputnak ebből. Ehelyett van erre egy megoldás, ami egy úgy nevezett hőmérsékleti tényezőt alkalmaz, aminek segítségével újraszámolja ezt az eloszlást. Ez a tényező minél nagyobb, annál véletlenszerűbb kimenete lesz, minél kisebb, annál jobban igazodik az eredetihez. Ezután az újraszámlálás után pedig a valószínűség értékekből nem az argmax értéket vesszük, hanem a valószínűségeik alapján választunk egyet. Így az érték nagy eséllyel az argmax lesz, de nem mindig, és ez az, ami miatt nem lesz megfigyelhető a beakadás jelensége. Az előző példa kimenethez visszatérve, ha lefuttatom ezt a hőmérséklet alapú újraszámolást (nagy hőmérséklet értékkel, hogy látványos legyen), a kimenet így néz ki:

```
[1.16095838e-06 0.00000000e+00 4.79997034e-02 5.96046801e-03
 1.00129543e-04 4.01883828e-05 0.00000000e+00 9.43207453e-01
 0.00000000e+00 2.69089713e-03]
```

Itt is megfigyelhető az, ami az előbb, hogy az argmax hívás értéke 7 lenne, viszont a hozzá tartozó valószínűség jelentősen csökkent. A valószínűségekből történő véletlenszerű mintavételt pedig 100-szor lefuttattam, és eredményüket egy oszlopdiagramon ábrázoltam:



4.6. ábra A mintaavétel eredményei

Látható, hogy így is a 7-es értéket sorsolja legtöbbször a gép, viszont a 2-es, a 3-mas és a 9-es is kap esélyt. Ez pont elég arra, hogy a beakadás problémáját elkerülje a zeneszerző program. A hőmérséklet érték pedig zeneszerzésnél jelentheti azt, hogy "mennyire engedje szabadon a fantáziáját" a zeneszerző algoritmus. Így például repetitívebb ritmushangszeres részekhez elég alacsonyabb hőmérséklet értékkel foglalkozni, a vadabb szólókhoz pedig mehet a magasabb érték.

4.3.2 MIDI adatok

Utófeldolgozási algoritmusom lényegében az előfeldolgozás inverze. Az opcionális hőmérséklet alapú újraskálázás után megvan a végleges számsorozatom, amiből majd zenét fogok csinálni. Ehhez először előveszem az előfeldolgozás során definiált mapper dictionarym, amivel a számokat visszaalakítom hangokká. Egy ciklusban megyek végig a számsorozaton, minden számot visszalakítok hanggá, és megnézem, hogy az csak egy sima hang, egy akkord vagy egy szünet. A megfelelő Python objektumot ezután példányosítom, és az aktuális hanghosszt is megadom neki. Ezeket az objektumokat egy listába rakom, majd ebből a listából egy MIDI fájlt készítek a music21 segítségével, amit elmentek. Ebben a ciklusban a hangokat ellátom hangszerinformációval is, azaz minden hang tárolja, hogy milyen hangszeren játszódik le a zenében. Ezzel az a probléma, hogy így nem működik, mindig zongorán játszódik le a hang, és nem tudom miért. Ezt úgy tudtam orvosolni, hogy a MIDI fileok elejére szintén a music21 csomaggal, a 0. időpillanatba beszúrom a hangszerinformációt, így minden hang azon a hangszeren fog lejátszódni. Ez a megoldás tökéletesen működött akkor, amikor egy hangsávon egy hangszer volt, például gitár esetében. A dobokat viszont nem

tudtam sajnos működésre bírni így, mivel ahogy korábban kifejtettem, a MIDI dobsávok esetén minden ütőhangszer külön hangszerként van megjelenítve, így mégis hangonként kéne átadnom a hangszerinformációt, ami sajnos nem működött. A zenéket ezután mivel egy weboldalra szeretném kihelyezni, olyan formátumban kell lementenem, amit az alap html szabvány kezelni tud. A MIDI sajnos nem ilyen, ezért a wav-ra esett a választásom. Természetesen mivel JavaScriptben minden meg van írva, ezért html oldalakra beszúrható MIDI lejátszó is van, viszont én jobban szeretném ezt a konverziót szerveroldalon csinálni, Python kódból. Ehhez a midi2audio csomagot használtam, aminek a FluidSynth osztálya, ami a parancssorból használható FluidSynthet MIDI szintetizátort hívja, képes volt elvégezni a wavvá konvertálást.

4.3.3 Zenei hullámformák

Hullámformák esetében egy kicsit egyszerűbb volt a hanggá történő visszakonvertálás. Az elkészült zenét μ-law dekódoltam, így visszakerültem a folytonos hangtartományba. Természetesen a dekódolt értékeim is diszkrétek maradtak, viszont így már vissza tudtam alakítani folytonos zenévé azokat.

4.4 Markov-lánc

Jelen esetben nem mentem el odáig, hogy valós értékekké alakítsam, a [0; 1] intervallumba a számokat, meghagytam azokat egész számoknak, amiket stringgé alakítok, és azokat használom az állapotok neveinek. Ezután az mchmm csomag MarkovChain osztálya elkészíti nekem az átmenetmátrixot. Ezen mátrix alapján már ki tudtam indulni egy véletlenszerűen felvett állapotból (első hangja a zenének), és a valószínűségek segítségével tudtam lépkedni az újabb állapotokba. Ezekből a bejárt állapotokból egy tömböt csinálok, aminek elemeit a korábban készített mapperem, és egy kis string manipuláció segítségével visszaalakítok music21 MIDI objektumokká. Ezekből az objektumokból már könnyedén tudtam MIDI fájlokat létrehozni.

A Markov-lánc segítségével Iron Maiden zenéket próbáltam generálni, és először egyszerűbb, majd egyre komplexebb problémákat adtam a gépi tanuló algoritmusnak.

Először csak basszusgitár MIDI részeket dolgoztam fel, mivel az általában egyszerűbb, és kevesebb hangból áll, mint a többi hangszer. Emellett feltételeztem egy konstans tempót és egyszerűsítés végett az akkordoknak csak a fő hangját (root note) vettem figyelembe. Következő futtatáskor maradtam a basszusgitárnál, viszont a teljes

akkordokkal dolgoztam. Ezután bekapcsoltam a változó hanghosszokat is, külön kódolva, egy másik Markov-lánc segítségével generálva azokat.

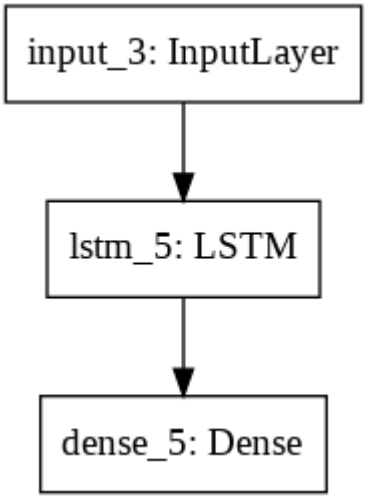
A basszusgitárt lecseréltem rendes gitárra, de az akkordok helyett maradtak csak a root noteok. Bekapcsoltam az akkordokat is, viszont itt falba ütköztem, mivel az állapotvalószínűség mátrix létrehozásának ideje elszállt, mivel túl sok fajta hang lehetőség volt, így az nem futott le csak percek alatt, ezért nem mentem tovább, mivel ennél többféle hangból álló zene esetén még sokkal hosszabb lenne a mátrix létrehozásának ideje, így az egybekódolt hossz+hang kombóval nem próbálkoztam.

A MIDI sávokon működő algoritmust ezután megpróbáltam folytonos zenékre is alkalmazni. Ehhez először diszkrét értékekké alakítottam azokat a μ-law algoritmussal. Sajnos az algoritmus alap részletessége, a 8 bit túl sok volt, mivel a 256 diszkrét értékre kódolt hangok is túl sok kombinációt eredményeztek, hogy a mátrix belátható időn belül létrejöjjön. Emiatt csökkentettem a kódolás bitszámát 7-re, 128 diszkrét értékem lett. Így néhány perc alatt létrejött a mátrix, viszont a szimulációval akadtak gondok. MIDI esetben egy generált hang sokkal hosszabb időt jelent, mint folytonos hang esetén, mivel folytonos hangnál magas a sampling rate, így ahhoz, hogy akár néhány másodpercnyi audiót is létre lehessen hozni, több tízezer értéket kéne létrehoznom az állapotátmenetek segítségével.

4.5 Deep learning megoldások

4.5.1 LSTM alapú neurális hálózat

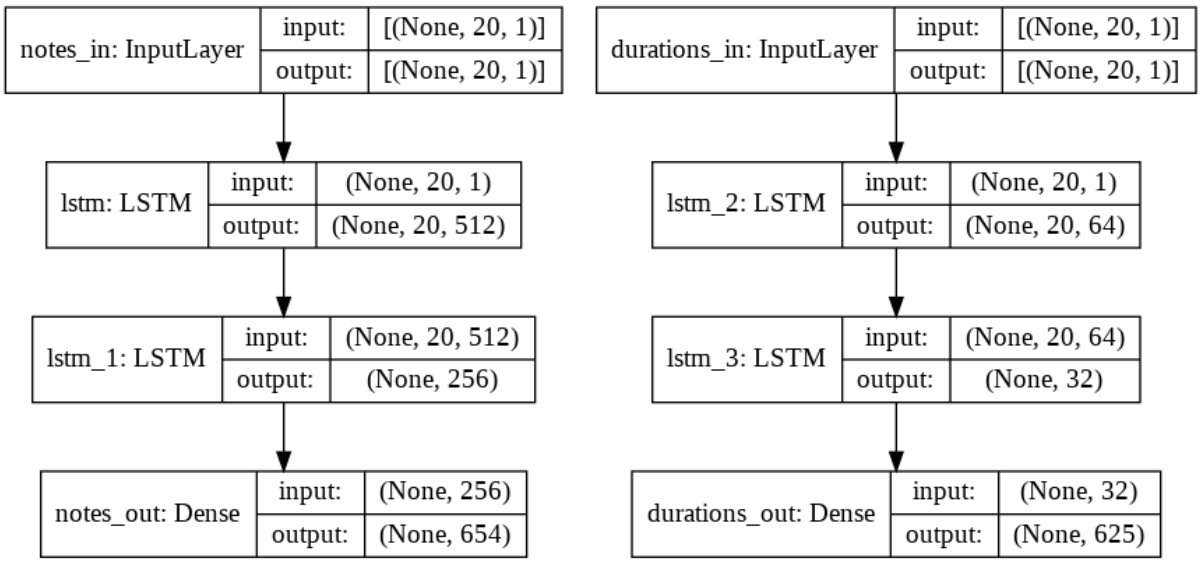
A deep learninges megoldásaimat egy egyszerű LSTM alapú neurális hálózattal kezdtem. Az architektúra így nézett ki:



4.7. ábra Az alap LSTM architektúra

Az input rétegbe belemegy egy szeletnyi számmá kódolt MIDI objektum, az LSTM réteg feldolgozza ezt a szekvenciát, majd a Dense réteg ad egy szoftmax kimenetet, ami a következő hangra jósolt érték. Egyszerűbb szekvenciákon, például basszusgitár MIDI sávokon ez is képes volt kellemes eredményt elérni, viszont komplexebb zenékhez nem tudott elég jól tanulni. Ezt a problémát a háló mélyítésével orvosoltam, beleraktam még egy LSTM réteget, nagyobb neuronszámmal. Így már képes volt komplexebb zenék struktúráit is megtanulni, basszusgitárzenéket már gondok nélkül tudott generálni, és már a gitársávokkal is megbírkózott. Ebben a megoldásomban egyben kellett kódolnom a hangokat a hanghosszaikkal, és a modell együtt adott a kettőre predikciót.

Egy másik LSTM megoldásom is volt, ami külön kódolt hangmagasság és hanghossz értékeken dolgozott. Két bemeneti és két kimeneti rétege volt. Az egyik oldalán inputként beletettem a hanghosszok tömbjét, másik oldalán a hangmagasságokat. Az egyik output réteg jósolta a következő hang magasságát, a másik pedig annak a hosszát. Az architektúra így nézett ki:

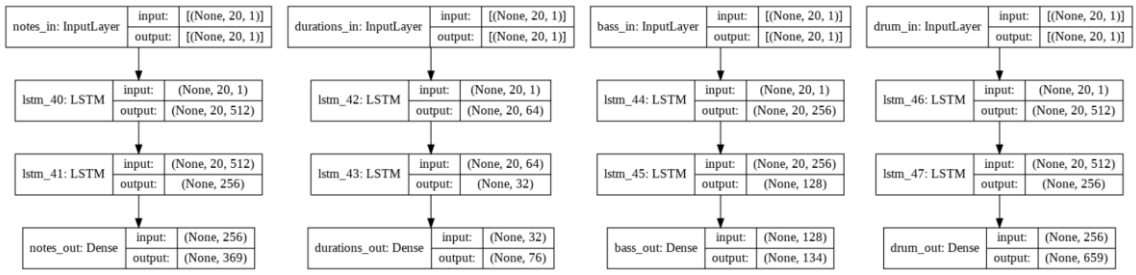


4.8. ábra A kétoldalú stacked LSTM architektúra

Ez a megoldás jónak bizonyult, az előző LSTM architektúrához hasonló eredményt tudott produkálni. Ennek a megoldásnak az az előnye viszont, hogy szét van választva a két generátor, így megadhatók neki más-más paraméterek. Így, ha mondjuk a hangok magasságainak feldolgozásához optimálisabb egy nagyobb neuronszámot választani, mint a hanghosszoknak, akkor itt megtehetem azt, az előző megoldásnál pedig nem.

Próbáltam úgy továbbfejleszteni a modellt, hogy a két, LSTM rétegek által feldolgozott szekvencia egy Merging layer segítségével összefut egy közös részbe, és onnan megy kifele a két output irányába a predikció. Ez viszont nem tudta hozni az eddigi eredményeket, rosszabbul hangzó outputokat tudtam vele generálni, és a tanulás során a loss sem ment le kellően jó értékig.

4.5.2 Többhangszeres stacked LSTM



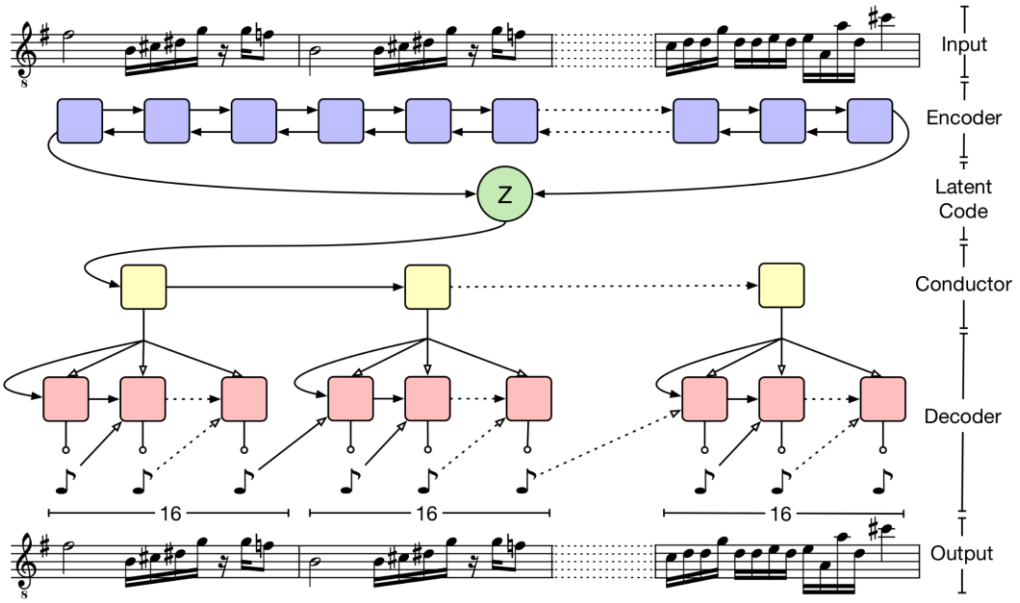
4.9. ábra A többhnagszeres stacked LSTM

Eddig csak egyféle zenei sávon tanítottam a modellem, így külön-külön tudtam gitár, basszusgitár zenéket generálni, viszont együtt a kettőt nem. Az előző modellemből kiindulva oldottam meg ezt a problémát. Most nem csak két összekapcsolt modellt csináltam, hanem négyet. Az egyik diktálja a tempót, az predikálja a következő hangnak a hosszát, így garantálva, hogy nem esnek ki a ritmusból egymáshoz képest a "zenészek". A másik három ága a modellnek gitár, basszusgitár és dob hangokat predikált, így szimulálva egy három tagú rockzenekar működését. Itt például nagyon jól jött, hogy szabadon választhatom meg a neuronszámokat egyes ágakban, így ki tudtam használni, hogy a gitársávokon történő tanuláshoz komplexebb, több cellából álló LSTM-ek szükségesek, a basszusgitárhoz képest. Ez is jó eredményeket produkált, kellemesen hangzó, együttműködő többhangszeres zenéket sikerült létrehoznom.

Egy problélmája volt ennek a megoldásnak, hogy a dob nem működött. A dobok egy más MIDI sávban vannak megoldva, és ugyan bájtszinten sikerült jó predikciókat létrehozni a modellemnek, azt nem tudtam megoldani, hogy a dobos MIDI sáv rendesen működjön, és ütőhangszeres hangja legyen. Ezt viszont nem tekintettem prioritásnak, ezért ezekből a generált zenékből végül kivettem a dobos MIDI sávot.

4.5.3 MusicVAE

Eddig LSTM alapú, autoregresszív generatív modellekkel foglalkoztam. Ez a megoldás viszont attól teljesen eltér. Itt a generátor egy teljes szeletnyi zenét ad ki magából outputként, nem csak egy hangot. A MusicVAE egy Variational Autoencoder alapú generatív megoldás. Az architektúra így néz ki:



4.10. ábra A MusicVAE architektúra

Az autoencoder encoder része Bidirectional LSTM-ekből áll, ezek tanítás során értelmezik az input zenét, és egy látens kódot hoznak belőle létre. Ezt a látens kódot dolgozza fel a decoder, ami először egy Conductornak nevezett LSTM rétegen küldi át azt. Ezek a rétegek egy kis feldolgozást végeznek az inputjukon, majd továbbküldik a feldolgozott inputjukat a Decoder LSTM blokkoknak, amik ezekből a szekvenciákból 1-1 output hangot állítanak elő. Ezeket az output hangokat egymás után konkatenálom, így kapom meg az outputot.

Tanítás során az történik, hogy az input és output szelet ugyanaz, az encoder által tanult zajból az eredeti szeletet próbálja visszaállítani a modell, így módosulnak eközben a decoder súlyai. Generáláskor viszont a decoder kap normáleloszlású zajt inputként, és a behangolt súlyai segítségével abból egy teljesen új zenei szekvenciát állít elő. Ezeken a szekvenciákon is végrehajtom a szoftmax kimenet hőmérséklet alapú újraskálázását, azt figyeltem meg, hogy úgy jobb eredményeket kapok.

4.5.4 Attention alapú neurális hálózat

Az LSTM hálózatomból kiindulva kezdtem el megtervezni ezt az architektúrát. Először lecseréltem az LSTM réteget egy Attention rétegre, a háló többi része változatlan maradt. Ez a réteg tanulta a hangok közötti összefüggéseket, viszont alapból nem hozott jó eredményt. Úgy tudtam az eredményen javítani, hogy a Transformerekből megismert

MultiHeadAttention réteget használtam, ami több „fejének” köszönhetően más-más dolgokat tud megtanulni az input hangokról, így jobban tudta a függőségeket modellezni. Ezután két újabb dolgot vettem át a Transformerektől, az embeddinget és a helyzeti kódolást (positional encoding). Az embedding a bemeneti rétegként szolgál, ahol a modell normált értékek helyett egész számokat kap, majd magának tanulja meg, hogy hogyan érdemes azokat valós számokként reprezentálni. Nem is csak egyetlen számként, hanem egy 256 hosszú, valós számokból álló vektorként voltak így ábrázolva a hangok. Ezután következett a positional encoding. A valós számokat az input hangsorban, és az embeddelt vektorban elfoglalt helyzetük szerint az alábbi függvénnyel módosítottam úgy, hogy a kiszámolt függvényértékeket hozzáadtam a számokhoz:

$$PE(pos,\ i) = \begin{cases} \sin\left(pos * e^{\frac{-log(10000)*i}{256}}\right), & i\ =\ 2k, k \in N \\ cos\left(pos * e^{\frac{-log(10000)*i+log(10000)}{256}}\right), & i\ =\ 2k\ +\ 1, k \in N \end{cases}$$

Ez egy kicsit eltér az eredeti, Transformereknél alkalmazott enkódoló függvénytől, viszont a Music Transformer implementációban ez volt. Az enkódolással és a MultiHeadAttentionnel már sikerült a modellnek tanulnia, és jó eredményeket elérni. A végső architektúrában a kódolt bemenet átmegy az Attention rétegen, majd egy szoftmaxszal ellátott teljesen összekötött réteg adja a háló kimenetét.

Ezután kipróbáltam azt, hogy az enkódoláshoz nem embedding és positional encoding rétegeket használok, hanem egy LSTM-et, így kombinálom a rekurrens, és az attention alapú megoldásomat. Ezzel viszont sajnos nem tudtam eredményt elérni, nem tanult rendesen a hálózat.

4.5.5 Transformer hálózat

Ahogy az Attentionnél, a Transformernél is a Music Transformer megvalósítása volt segítségemre. Az architektúra annyiban változott, hogy miután a positional encoding megtörtént, nem rögtön az Attention rétegbe megy az adat, hanem enkóder és dekóder blokkokba. Az enkóder blokk egy Attention rétegből, egy rétegnormalizációból és két teljesen összekötött rétegből áll. Ezek a blokkok egymás után dolgozzák fel a beléjük helyezett adatot, az egyik blokk kimenete a másikéhoz van kötve. A dekóder blokkoknál viszont nem csak az előző blokk kimenete jelenti a bemenetet, hanem az utolsó enkóder blokk kimenete is. A dekóder egy Attention réteggel és egy rétegnormalizációval dolgozza fel az előző blokk bemenetét, majd egy újabb Attention réteggel veti össze ezt

az értéket az enkóder kimenetével, majd azt is normalizálja, és bevezeti a végső két teljesen összekötött rétegbe. Az utolsó dekóder blokk kimenete az eddig már megszokott szoftmaxos kimeneti rétegbe vezeti az adatot.

4.5.6 GPT-2

A GPT-2 architektúrát nem valósítottam meg, mivel egy hatalmas modellről van szó, így valószínűleg nem tudtam volna rendesen betanítani. Helyette egy Colab notebookot használtam, ami a GPT-2 Simple modell használatát könnyíti meg. [26] A GPT-2 Simple egy kicsit eltér az alapvető modelltől, de a projekthemhez teljesen megfelelt. Először egy szövegkorpuszt hoztam létre a hangjaimból, amihez a sima előfeldolgozó eljárásomat használtam, csak mielőtt számokká alakítottam volna a beolvasott hangjaimat, összekonkatenáltam azokat egy hosszú stringbe, szóközökkel elválasztva, és elmentettem egy .txt fájlba. Ezt a fájlt ezután feltöltöttem a GPT-2-es Colab notebookot futtató virtuális gépre, majd miután letöltöttem a GPT-2 Simplet a notebook utasításai alapján, finomhangoltam a modellt a megfelelő metódusával ezen a fileon. Lényegében transfer learninget hajtottam végre. Ezután a modellel generáltam szöveget, ami így a saját szövegkorpuszom elemeit, azaz a hangokat tárolta. Ezeket a generált szövegeket a meglévő utófeldolgozó algoritmusaimmal nagyon egyszerű volt visszaalakítani zenévé. Az egyetlen problémát az jelentette, hogy képes volt néha érvénytelen hangot generálni a modell, viszont ez kizárólag a generált szöveg elején és végén fordult elő. Ezt úgy orvosoltam, hogy levágtam minden dal első és utolsó kettő hangját. Amennyiben még így is érvénytelen hangra futna a program, kivételkezeléssel elkapom a hibát, hogy a teljes alkalmazás ne álljon le, csak az adott generált zene vesszen el.

4.5.7 Wavenet

Az architektúrát az eredeti leírás szerint valósítottam meg. A reziduális blokkokban a bemenetet egy konvolúciós szűrő dolgozza fel, majd kimenetének szigmoiddal és tanh-val aktivált szorzatát adom hozzá a szűrő bemenetéhez. Ez az összeg egy ilyen blokk kimenete, és ezt a kimenetet a következő blokk mellett a blokkok után elhelyezkedő összeadó rétegbe kötöm be. Az összeadó réteg kimenete ismét átmegy konvolúciós szűrőkön, majd egy szoftmax aktivációval rendelkező teljesen összekötött kimeneti réteg adja a végső predikciót.

4.6 Webalkalmazás Django frameworkkel

A deep learning modellek által generált zene könnyebb elérhetőségének céljából terveztem egy webalkalmazást, aminek segítségével meghallgathatók a létrehozott zenék. Az alkalmazást Python nyelven készítettem el, mivel a neurális hálóim elő és utófeldolgozó algoritmusait is abban írtam meg, így sok kód átemelhető volt onnan. A Python egyik legnépszerűbb webes keretrendszerét, a Djangot használtam.

4.6.1 A szoftver tervezése

A szoftver tervezésekor a Django alapvető MVT architektúrája mellett a háromrétegű architektúrát alkalmaztam. Az üzleti logikát lényegében a megvalósított gépi tanuló modellek adják. A felhasználói kéréseket a nézetek dolgozzák fel, és attól függően, hogy azok milyen kérések, nyúlnak az adatelérési réteghez, vagy a gépi tanuló modellekhez. Az alkalmazásban lehetőség van már korábban legenerált zenék meghallgatására, vagy újak generálására. Már meglévő zenék meghallgatásakor fölösleges a gépi tanuló modellekhez hozzányúlni, ilyenkor a nézet közvetlenül az adatbázis elérését biztosító szolgáltatásoktól kéri le a megfelelő paraméterekkel rendelkező zenét. Az új zenék generálása viszont másképp történik, ott vannak használva a gépi tanuló modellek. A nézet ilyenkor szól a megfelelő modellnek, hogy készítenie kell megadott számú új zenét. A modell létrehozza a zenéket, majd azokat elmenti az adatbázisba, így majd hallgathatók lesznek.

A fejlesztés során az objektumorientált szemléletmódot igyekszem követni. Annyi különbséget viszek a dologba, hogy mivel a Python nyelv nem annyira erősen objektumorientált, mint például a Java, hogy mindennek osztálynak kell lenni, csak azokat az objektumokat kezelem osztályként, amiknek osztályszintű attribútumaik vannak, vagy fontos használnom náluk az öröklést. Ilyenek például az adatbázis modelleim, amik a Django ORM helyes működése érdekében egy, a keretrendszer által megadott osztályból kell leszármazzanak. A gépi tanuló modelleimnél is fontos az öröklés, hiszen ott egy ősosztály fogja definiálni az alapvető működésüket, néhány attribútumot és közös metódust, viszont minden leszármazott felül fogja definiálni magának a megfelelő függvényeket, mivel például más előfeldolgozási eljárást igényelnek. Vannak viszont funkciók a szoftveremben, amiket csak külön fájlban helyezek el, a felelősségeiknek megfelelően, viszont a korábban említett okokból nem

zárom őket egységbe osztályként. Ilyen például az adatbázisműveleteket definiáló szolgáltatás.

4.6.2 A szoftver implementációja

Fejlesztés során figyelembe kellett vennem a keretrendszer sajátosságait, és a saját terveimet is. A projekt létrehozása után beállítottam a settings.py fájlban, hogy sqlite adatbázist szeretnék használni a szoftveremhez. Ezután létre is hoztam az ORM-hez a modelleimet. Két adatbázis entitásom van, egyik a zenéket, másik a lementett gépi tanuló modelleimet reprezentálja. A gépi tanuló modelleimnek van egy neve, és egy elérési útja, ami azt az elérési utat jelenti, ahova az adott betanított modellt lementettem. A zenéknek címe van, elérési útja, és egy előadója, ami egy külső kulcs, amivel hozzákapcsolom az őket letrehozó gépi tanuló modellhez a zenéket. Mindkét modellnél felüldefiniáltam a stringgé alakító metódust, így, ha egy objektumot közvetlen szeretnék a standard kimeneten megjeleníteni, akkor nem csak a memóriaterületüket jelző hexadecimális kód, és az osztály neve jelenik meg, hanem magáról az objektumról információ, zenéknél a címe, gépi tanuló modelleknél pedig a neve. Készítettem egy adatbázisszolgáltatás fájlt is, amiben különböző műveleteklet definiálok. Itt lehet lekérni az adatbázisból az összes dalt, vagy éppen beszúrni abba egy újat. Az entitások és a szolgáltatások adják az alkalmazás adatrétegét.

Az üzleti logikát a gépi tanuló modellek viselkedését megvalósító osztályok adják. Először definiáltam hozzájuk egy alaposztályt, ami tárolja a minden leszármazottra jellemző attribútumokat, méghozzá a nevet és az elérési utat, amik az adott osztályt jellemző adatbázis rekordban is megjelennek. Az összes modellem MIDI adatokon dolgozik, így a MIDI adathalmaz betöltése is egységesíthető volt itt. A modell betöltő függvény is itt helyezkedik el, hiszen minden Kerasban írt neurális hálózatomnak ez a folyamata egyformán történik. Természetesen eltérés esetén a leszármazott felül is definiálhatja a közös viselkedést. Ezek mellett a generált zenék adatbázisba való mentése is ki van az ősbe szervezve, ez is közös viselkedés. Minden ilyen osztályom rendelkezik egy előfeldolgozó, és egy zenegeneráló függvénnyel is, amiket nem szerveztem ki az ősbe, mivel mindegyik osztálynál teljesen másképp történik ez a folyamat. A leszármazott osztályokban az előfeldolgozás, és a zenegenerálás a modelltanításokhoz használt jupyter notebookjaimban megírt megfelelő metódusok átemelésével keletkezett.

Mivel természetesen a modellek belső működésében, például az előfeldolgozásnál is vannak hasonló dolgok, ezeket kategóriákra lebontva kiszerveztem utility fájlokba. Ezekben a fájlokban csak segítő függvények találhatók, amiket a modellek használnak, így ebben az architekturális rétegben kaptak azok is helyet, más rétegből nincsenek hívva.

Továbbá van az execute_models.py fájl, ami a gépi tanuló modellek végrehajtásáért felel. Ebben olyan függvények találhatók, amik a megfelelő modelleket felparaméterezve példányosítják, majd meghívják a zene létrehozásához szükséges függvényeket.

A megjelenítési réteg összerakásánál fontos szerepet kapott a Django alap architektúrája. Először az urls.py fájlban definiálnom kellett a routingot, hogy milyen Uniform Resource Locator (url) paramétert hova küldjön tovább a rendszer. Ezekhez megfelelő nézeteket kellett írnom a views.py-ban. Mindegyik url-t a nézet egy külön függvénye szolgál ki. Amikor url paraméter nélkül hívjuk a szervert, akkor egy véletlenszerű dalt kér le az adatbázisból, és az jelenik majd meg a weboldalon. Ha az első paraméter a song, akkor megadható további paraméterként egy adott modell neve, és a kívánt hangszer is, és akkor egy azáltal a modell által generált, az adott hangszeren játszott dal fog megjelenni. Ha az első paraméter a generate, utána a kívánt modell neve, utána a hangszer, majd opcionálisan egy szám van az url-ben, akkor pedig generál a kívánt modell egy zenét, a kívánt hangszeren, ha szám is van megadva, akkor nem egyet, hanem annyit, amennyit a felhasználó megadott. Egy utolsó opcionális paraméter a hőmérsékleti tényező, ami a modell kimenetének szoftmax függvényének újraskálázásáért felel. Ez a generálás egy hosszú folyamat, és nem is elvárható, hogy az eredményt megvárja a felhasználó, így nem is fog az oldalon megjelenni az. A generálás kezdetéről egy üzenet megjelenik a weboldalon, majd egy háttérszálon elkezd futni a megfelelő gépi tanuló modell, és legenerálja a megfelelő zenét, amit az adatbázisban eltárol.

A help url paraméterrel egy olyan oldalra kerülhetünk, ami elmagyarázza, hogyan működik a weboldal, és ott egy egyszerű menü segítségével kiválaszthatók a korábban említett paraméterek, nem kell a böngészősávban az url-t szerkesztgetni.

Az about url paraméterrel egy kis bemutató oldalra kerülünk, ahol egy rövid leírás található a weboldalról. Ennek jelenleg nincs sok jelentősége, viszont amennyiben később ezt a weboldalt publikussá tenném, egy jó irányadó lehet a felhasználóknak.

A weboldalon történő megjelenítéshez szükséges az architektúra T betűje, a template. A templatek html fileok, amik az adott weboldal struktúráját írják le, adatkötés segítségével plusz információkkal kiegészítve. A projektem templatein található egy navigációs sáv fölül, amivel egyszerűen el lehet más oldalakra navigálni. Ezalatt a zenehallgatós oldalakon csak a zene adatai szerepelnek, és maga az audio. Mivel ezek minden zenéhez mások, ezért ezek az információk a nézetek adatkötésein keresztül kerülnek ki a weboldalra. Természetesen stílusozást is kaptak a weboldalak. A navigációs sávhoz a bootstrapet használom, amivel nagyon egyszerűen létre tudtam hozni a kinézetét, egy kis saját Cascading Style Sheets (css) leírása után. A help oldalhoz a kliensoldali webfejlesztés harmadik elengedhetetlen eszközét, a JavaScriptet is használnom kellett, mivel az ott kiválasztott paraméterek függvényében egy eseménykezelő fogja betölteni a kívánt oldalt. Ezeknél a fájloknál is követtem a függőségek szétválasztásának elvét. Ugyan a htmlbe lehetne közvetlenül is beleírni a css és JavaScript kódot, az áttekinthetőség kedvéért mindegyik külön fájlt kapott magának.

5 Eredmények

5.1 A Wavenet, és a folytonos hullámformák problémái

Ugyan az architektúrát sikeresen meg tudtam valósítani, viszont a tanításom már nem volt sikeres, erőforrás korlátok miatt. 12 darab, körülbelül egyenként félórás, folytonos hullámformájú dalt olvastam be a librosa csomaggal, ami olyan hatalmas adatamennyiséget eredményezett, hogy a Wavenetet a teljes adathalmazon tanítva, egy NVIDIA Tesla T4-es GPU-n 85 óra lett volna egy epoch. Ezért le kellett csökkentenem a tanító adathalmaz méretét a századára, hogy végig tudjam követni a tanítást. Sajnos nem tudott rendesen tanulni a modell, fluktuáltak az eredmények az epochok során, valamikor jobb lett a loss, valamikor rosszabb. Néhány epoch után abbahagytam, és megnéztem, mit tud generálás során a modell. A generálás is hosszú ideig tartott, 2 másodpercnyi 16kHz mintavételezésű hangot 20 perc alatt sikerült generálnia. Ugyan sikerült hangot generálnom, viszont az csak egy rövid recsegés volt, ami zenének nem nevezhető, valószínűleg a kevés tanítási idő és a lecsökkentett adatmennyiség miatt. Így a Wavenetet és a folytonos hullámformák alkalmazását sajnos nem tudtam rendesen körüljárni ebben a projektben.

5.2 A MIDI generátor modellek értékelése

5.2.1 A hosszú távú függőségek kérdése

A rekurrens neurális hálózatoknál kellemetlen probléma volt, hogy hosszútávú függőségeket nehezen tudtak modellezni, nem tudtak rendesen tanulni. Ennél a projektnél le is teszteltem ezt. Létrehoztam egy egyszerű LSTM és egy Attention alapú neurális hálózatot, és 200 hangból álló bemeneti vektorokon tanítottam. Az első szembetűnő dolog a sebesség volt. A rekurrens hálók nagyon lassan tanulnak, egy epoch az Attention alapúnak 134 másodperc volt, míg az LSTM-nek 548, ez kb 4-szeres sebességkülönbséget jelent. További sebességkülönbséget jelentett az, hogy az Attention alapú előbb kezdett el konvergálni és kisebb loss értékeket elérni. Miután betanultak a hálók, a teszthalmazon elért loss az Attention esetében 1.4452 volt, az LSTM-nél 1.8911. A teszt alapján kijelenthetem, hogy hosszútávú függőségek modellezésében tényleg az Attention alapú hálózatok jelentik a jövőt.

5.2.2 Személyes értékelés

Eleinte minden betanított modellel generáltattam zenéket, és meghallgattam azokat, amiből rájöttem, hogy a zenék minősége valamilyen szinten korrelál az elért test loss értékkel. Innentől kezdve csak azon modellek zenéit hallgattam meg, amik egy adott értéknél jobban tudtak a teszten teljesíteni. A modellekkel gitár, basszusgitár, és két architektúrával a kettő kombinációját is létrehoztam. Basszusgitárzenét szinte mindegyik architektúra elég jót tudott létrehozni, ez annak volt köszönhető, hogy alapvetően azon nem szoktak komplex szólókat játszani a zenészek. Természetesen léteznek basszusigtár virtuózok, akik igen, de az Iron Maiden zenéjében, amiből a tanító adathalmaz állt, nem ez a jellemző. Emiatt az nem is igazi metrikája szerintem a modellek összehasonlításának, mivel elmondhatom, hogy mindegyik jól teljesített. Magasabb hőmérsékleti értékeknél szólók is megjelentek, néha elég komplexek is, amik viszont már az Iron Maidenre nem voltak jellemzők, de alacsonyan tartva ezt az értéket, megkaphattuk az elvárt, jó hangzású és ritmusú basszusgitárzenét. A gitársávok létrehozása már komolyabb kihívás volt, a kettő összekombinálása pedig egy még bonyolultabb feladat, amit csak a legjobban teljesítő modelleimmel próbáltam meg.

Első kiértékelendő modellem a Markov-lánc volt. Ez a megoldás gyorsan volt implementálható, és a tanulása, azaz az állapotátmenet valószínűség mátrix létrehozásának az ideje is gyors volt. Túl sok hangkombináció esetén ez az idő viszont exponenciálisan elszállt, szóval nagyon komplex zenékhez már emiatt is rossz választás lenne. Alapvetően kicsit virtuóz módon generálta a zenéket ez a modell, mivel nem igazán értette meg a bemeneteket, csak az előfordulásuk valószínűsége alapján találgatott valamit, és mivel minden hang létrehozásához csak az előző hangból következtetett, nem volt a zenében semmi hosszabb távú koherencia. Néha ugyan képes volt elfogadható kimenetet produkálni, összességében nem tetszett.

Ezután az LSTM következett, ami már sokkal jobb eredményt volt képes elérni. Látszott benne a hosszabb távú koherencia, alacsony hőmérsékleti értékkel kellemes ritmusgitár dallamokat tudott létrehozni, kicsit magasabb értékkel viszont a szólógitár részei is jók voltak. A szólóknál ugyan nem tudta rendesen eltalálni az Iron Maiden stílusát, elég virtuózok voltak azok, és néhány résznél túltanulás is megfigyelhető volt, amikor a generált zenében meg volt ismételve egy már létező Iron Maiden riff, viszont ez volt az egyik legjobb eredményt elérő modellem. Megpróbáltam ezzel az architektúrával gitárt és basszusgitárt is tartalmazó zenéket generálni, amik meglepően jók voltak.

Természetesen nem voltak összetéveszthetetlenek az eredeti Iron Maidennel, de szerintem együtt jól hangzó kombinációkat sikerült a modellnek generálni.

Az Autoencoder alapú modellem eredménye nem tetszett, a generált gitársávok túlságosan randomok voltak. Valószínűleg azért volt ez, mert a komplex LSTM struktúra nagyon lassan és nehezen tanult, és a korlátozott erőforrásmennyiségemmel nem tudtam rendesen végigcsinálni a tanítási folyamatot.

Az Attention alapú modellek ugyan a hosszútávú függőségeket az LSTM-nél jobban tudták modellezni, viszont ezen az adathalmazon a rövidtávú függőségeket megtanuló LSTM jobb eredményt volt képes elérni. Ugyan képes volt a modell elég jó zenéket generálni, a Markov-láncnál jobb volt, de az LSTM-nél nem.

A komplexebb Attention alapú architektúra, a Transformer már jobban tudott tanulni. Az LSTM-mel azonos szintre tenném a generált zenéit. A zenék élvezetesek voltak, volt köztük jó ritmusgitáros és szólógitáros eredmény is. A gitárt és basszust is tartalmazó zenéknél elért eredményei nem voltak annyira jók szerintem, a két hangszer nem hangzott annyira jól együtt, mint az LSTM esetében. Mivel ez egy elég komplex architektúra, a sok paraméterének rendes tanításához szerintem egy nagyobb méretű adathalmazra lenne szükség, amivel az LSTM-nél is jobb eredményt érhetne el.

A finetuneolt GPT-2 képes volt jó zenék, főleg akkordmenetek generálására, szólói ritkán voltak, azok sem elég jók. Akkordmenetei viszont meglepően jól hangzottak, ritmusgitárt generálva jó alapot tudnának adni egy dalhoz. Látszik, hogy nem a GPT-2 architektúra nagyon erős, nagyon sok lehetőség van benne, viszont egy ilyen finomhangolás nem hoz ki belőle mindent, főleg egy ilyen kis adathalmaznál. Ha nulláról, sok idő alatt be lehetne tanítani egy nagyméretű metálzenei adathalmazon, akkor kiváló eredményeket tudna elérni, hasonlóan az OpenAI módosított zeneszerző GPT-2 módosulatához, a MuseNethez.

5.2.3 Szubjektív értékelés

A zenéket nem csak saját hallgatásom alapján értékeltem, hanem készítettem egy kérdőívet, amit kiküldve embereknek, több értékelést is kaphatok, amivel össze tudom hasonlítani eredményeimet. Mivel a kérdőív nem lehet túl hosszú, sajnos csak 1, maximum 2 zenét tudtam beletenni minden architektúra kínálatából. Basszusgitárzenéket nem tettem bele, mivel azok szinte minden architektúránál jók voltak, így a gitárzenére fektettem a hangsúlyt, minden architektúrámnál beletettem egy generált gitárzenét, és két

modellem, az LSTM és a Transformer gitárt és basszust is tartalmazó zenéjét is beleraktam. Ez nem egy reprezentatív adatmennyiség, mivel 1-2 zene alapján nem biztos, hogy jól meg lehet ítélni egy architektúra teljesítményét, viszont így legalább több ember is tudta értékelni a zenéket. Minden zenénél három kérdést tettem fel, és egy opcionális szöveges értékelésre is lehetőséget adtam.

1. Nevezhető zenének?
2. Mennyire tartozik bele a metál műfajába?
3. Mennyire hasonlít az Iron Maidenhez?

Az első kérdésre igennel/nemmel lehetett válaszolni, az utolsó 2 kérdésre a választ pedig egy 0-5-ig terjedő skálán kellett megadni.

5.2.4 Objektív értékelés

A szubjektív, emberi fülre alapozott zeneértékelés mellett egy olyan módszerre is szükségem volt, amivel nagymennyiségű zenét ki lehet gyorsan értékelni, viszonylag objektíven. A test loss nem egy jó metrika erre, mert az összességében nem díjazza a neurális háló kreativitását, egy kisméretű adathalmazon való teljesítményét vizsgálja csak, nem a generált műveit. Egy korábban már említett cikkben, a MuseGAN-éban megfogalmaztak néhány zenekiértékelési metrikát, amiket alapul vettem a saját metrikáimnál.

Nézték, hogy hány üres ütem található a zenében, én ehelyett azt néztem, hogy összesen mennyi szünet van a zenében és azt is, hogy a leghosszabb szünet mennyire hosszú. Minél kevesebb a szünet, annál jobb, mivel annál többet tudunk zenét hallgatni, és nem csak a csendet.

Nézték a használt hangmagasságok számát is, ezeket én is nézem, továbbá a hangokat úgy is nézem, hogy azonos hang más oktávban nem számít külön értéknek, így például tudom azt is vizsgálni, hogy egy generált zene teljes egésze mennyire követ egy skálát. Emiatt néztem azt is, hogy hány hang lóg ki egy adott hétfokú skálából a generált zenében.

Nézték a zene ritmusát is, az ő cikkükben a popzene leggyakoribb ütemét, a 4/4-et keresték, én viszont csak összességében nézem az ütemeket, hogy miből mennyi van, mivel az Iron Maiden zenéje tartalmaz sokszor érdekesebb ritmikákat is a sima 4/4-nél.

Emellett saját metrikaként gondoltam arra is, hogy milyen hangközök szerepelnek

a zenében. Minél jobban tudja a hangközöket is modellezni egy adott generatív modell, annál közelebb állhat az eredetihez.

Ezen metrikák kiértékeléséhez egy python scriptet írtam, amit az eredeti Iron Maiden MIDI-kre is lefuttattam, így ahhoz is tudom hasonlítani a generált zenéimet ezzel a kiértékelési módszerrel.

6 Összefoglalás

6.1 Továbbfejlesztési lehetőségek

6.1.1 A gépi tanuló megoldások továbbfejlesztése

Mivel a folytonos hullámformák generálása nem volt rendesen körüljárva a projektben, ezért az mindenképp egy érdekes továbbfejlesztési lehetőség lenne. A Google Colab időkorlátos, ezért az ottani GPU-kon nem tudnék egy olyan hosszú tanítást végigcsinálni, ami elég jó eredményeket tudna produkálni folytonos zenéken. Egy rendes felhőplatformon hardvert bérelve, nagyobb adathalmazon, erős GPU-kon végig lehetne csinálni a Wavenet tanítását, megnézni, hogy milyen eredményt adna.

6.1.2 A webalkalmazás továbbfejlesztése

Projektem során a szerveroldali renderelés és a Django csomag használata mellett döntöttem, hogy ezt a technológiát is megismerjem, és ilyen környezetben is tudjak fejleszteni. Azonban ma a webfejlesztésben nem ez a legelterjedtebb, hanem REST interfészek használata. Az adatot nem közvetlenül szerveroldalról renderelik ki egy html fájlba, hanem a szerver csak json formátumban küldi ki azokat, és így többfajta kliens is tud kapcsolódni hozzá. A weboldalhoz külön készül egy kliensalkalmazás, ami ezt az API-t hívja, és az jeleníti meg az onnan lehívott adatokat. Az előnye ennek az, hogy így például egy mobil kliens is készülhet, ami szintén az API-t hívja, nem csak webböngészőre lesz korlátozva a működés.

Jelen alkalmazásomnál úgy képzeltem el a REST interface működését, hogy egy olyan jsont adjon vissza, ami tárolja a zenék címét, szerzőjét, és emellett a tartalmát is. A tartalma ugyan egy bináris formátum, viszont az API válasz jsonbe bele lehet tenni azt stringgé alakítva, amennyiben a mérete nem túl nagy. Hogyha pedig mégis túl nagy a méret, akkor a wav formátum helyett egy kisebb méretű formátumot érdemes használni. Vagy mp3-má tömörítve lehet küldeni a zenét, vagy rögtön MIDI-ként, ami még kisebb, viszont akkor figyelni kell arra, hogy a kliens le tudja azt játszani, vagy vissza tudja konvertálni egy általa lejátszható formátumba.

7 Irodalomjegyzék

[1] M. C. Chun Hei, „Classical Neural Network: What really are Nodes and Layers?,” Medium, [Online]. Available: https://towardsdatascience.com/classical-neural-network-what-really-are-nodes-and-layers-ec51c6122e09.

[2] C. Olah, „Understanding LSTM Networks,” [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[3] S. Hochreiter és J. Schmidthuber, „LONG SHORT-TERM MEMORY”.

[4] L. Weng, „Attention? Attention!,” [Online]. Available: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html.

[5] D. Bahdanau, C. KyungHyun és Y. Bengio, „NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE”.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser és I. Polosukhin, „Attention Is All You Need”.

[7] A. Radford, K. Narasimhan, T. Salimans és I. Sitskever, „Improving Language Understanding by Generative Pre-Training”.

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei és I. Sutskever, „Language Models are Unsupervised Multitask Learners”.

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah és J. Kaplan, „Language Models are Few-Shot Learners”.

[10] A. Dertat, „Applied Deep Learning - Part 3: Autoencoders,” [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[11] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior és K. Kavukcuoglu, „WAVENET: A GENERATIVE MODEL FOR RAW AUDIO,” 2016.

[12] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville és Y. Bengio, „SAMPLERNN: AN UNCONDITIONAL END-TO-END NEURAL AUDIO GENERATION MODEL," 2017.

[13] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan és M. Norouzi, „Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders," 2017.

[14] S. Oore, I. Simon, S. Dieleman és D. Eck, „Learning to Create Piano Performances," 2017.

[15] A. Roberts, J. Engel, C. Raffel, C. Hawthorne és D. Eck, „A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music," 2018.

[16] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang és Y.-H. Yang, „MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," 2017.

[17] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, X. Bing, D. Warde-Farley, S. Ozair, A. Courville és Y. Bengio, „Generative Adversarial Networks".

[18] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel és D. Eck, „ENABLING FACTORIZED PIANO MUSIC MODELING AND GENERATION WITH THE MAESTRO DATASET," 2019.

[19] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu és D. Eck, „MUSIC TRANSFORMER: GENERATING MUSIC WITH LONG-TERM STRUCTURE," 2018.

[20] C. M. Payne, „MuseNet," 2019.

[21] P. Dhariwal, J. Heewoo, C. Payne, J. W. Kim, A. Radford és I. Sutskever, „Jukebox: A Generative Model for Music," 2020.

[22] „What is a MID file?," [Online]. Available: https://docs.fileformat.com/audio/mid/.

[23] „WAV vs. MP3 Files: A Guide to Audio File Formats," [Online]. Available: https://www.masterclass.com/articles/a-guide-to-audio-file-formats.

[24] J. Korsun, „10 Popular Websites Built With Django," [Online]. Available: https://djangostars.com/blog/10-popular-sites-made-on-django/.

[25] „Django introduction,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction.

[26] M. Woolf, „Train a GPT-2 Text-Generating Model w/ GPU For Free,” [Online]. Available: https://colab.research.google.com/drive/1VLG8e7YSEwypxU-noRNhsv5dW4NfTGce.

[27] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue és A. Roberts, „GANSYNTH: ADVERSARIAL NEURAL AUDIO SYNTHESIS,” 2019.

Függelék