Treball de recerca

L'ARTISTA INSENSIBLE

Creació d'una intel·ligència artificial per a la composició d'una cançó

Pau Hidalgo Pujol
Tutor: Narcís Bartis
2n de batxillerat A
INS Pere Alsius
Banyoles, 4 d'octubre de 2021

Agraïments

En primer lloc, m’agradaria donar les gràcies al meu tutor Narcís Bartis per fer-me de guia a l’hora de crear aquest Treball de Recerca (TDR), i suportar totes les explicacions que li anava fent sobre el codi. Agrair també la seva col·laboració a en Sergi Pérez (professor de l’institut) i a en Ramon Gallench (membre d’AImCAT, l’equip català que va participar en l’*AI Song Contest*). Tots dos van escoltar les idees que tenia sobre com crear el codi en un principi, i em van donar les seves opinions al respecte.

Moltes gràcies.

Abstract

We have always been told that computers don’t have human qualities, cannot express feelings, have no originality... But what would happen if we gave a computer the task of composing a song? This work seeks to solve this doubt using neural networks, watching how they work, and programming one from scratch. The artificial intelligence model, which tries to resemble the functioning of the human brain, will be trained using various compositions, learning from them to answer the question raised above. At the end of this work, we will see if, despite not hearing, this program is capable of creating a melody, and ultimately, art.

Siempre se ha dicho que los ordenadores no tienen cualidades humanas, no pueden expresar sentimientos, no tienen originalidad... Pero, ¿qué pasaría si le diéramos a un ordenador la tarea de componer una canción? Este trabajo busca resolver esta duda utilizando las redes neuronales, viendo como funcionan y programando una desde cero. El modelo de inteligencia artificial, que busca asemejarse al funcionamiento del cerebro humano, será entrenado utilizando varias composiciones, aprendiendo de ellas para dar respuesta a la pregunta planteada anteriormente. Al final de este trabajo, veremos si a pesar de no sentir, este programa es capaz de crear una melodía, y en definitiva, arte.

Índex

Índex **3**
Índex d'imatges 4
Llista d'abreviacions 6
Glossari 7
Introducció **8**
Objectius **9**
Marc teòric **10**
La intel·ligència artificial 10
Symbolic Artificial Intelligence 11
Algoritmes evolucionaris 11
Machine Learning 14
Neural Networks (xarxes neuronals) 17
Deep learning 25
Exemples d'AI 26
Representacions del contingut musical 27
Representacions dels senyals d'àudio 27
Representacions simbòliques 28
Piano roll 28
Notació ABC 28
MusicXML 29
Fitxers MIDI 30
Missatges de sistema (System Messages) 30
Missatges de canal (Channel Messages) 31
Llibreria MIDO 32
Intel·ligència artificial aplicada a la música i composició automatitzada 33
Història de la composició automatitzada 34
Projectes actuals 36
Magenta 36
Musenet 36
AIVA 37
AI song contest 37
Part pràctica **38**
Disseny del producte 38
Fases de l'elaboració del codi i evolució d'aquest 42
Funcionament del codi 43
Valoració del producte 60
Millora del producte 62
Resultats i interpretació **65**
Resultats de la primera versió 65

Resultats de la segona versió 66
Resultats tercera versió 68
Comparacions entre versions 69
Resultats versions de dotze notes 71

Conclusions **72**
Propostes de millora 73
Conclusions personals 74
Conclusions finals 75

Bibliografia **76**

Annexos **85**
Annex 1 : primera versió de la xarxa 85
Codi 85
Resultats 85
Annex 2 : segona versió de la xarxa 86
Codi 86
Resultats 86
Annex 3 : tercera versió de la xarxa 87
Codi 87
Resultats 87
Annex 4 : versions de dotze notes 88
Annex 5 89

Índex d’imatges

Figura 1: Schematic overview of the main branches of artificial intelligence (AI), including machine learning (ML) methods which are having an impact on spine research 10
Figura 2: Les relacions agent-medi en el RL 15
Figura 3: Esquema general d’una xarxa neuronal 18
Figura 4: Esquema del funcionament d’una neurona 19
Figura 5: Funcionament del GD 22
Figura 6: Capes d’una xarxa neuronal 24
Figura 7: Espectrograma i chroma feature 27
Figura 8: Inside a MusicXML file… 29
Figura 9: Mathematical definition of the softmax function 54
Figura 10: Categorical Cross-entropy fórmula 55
Figura 11: Unfolded representation of the implemented RNN structure 62
Figura 12: Gràfica segona versió 67
Figura 13: Gràfica tercera versió 69
Figura 14: Gràfiques comparació primera versió 69
Figura 15: Gràfiques comparació segona versió 70
Figura 16: Gràfiques comparació tercera versió 70
Figura 17: Gràfica cançó generada per TDR-B-12 71

Llista d’abreviacions

ANN: Artificial Neural Network

CNN: convolutional neural network

EA: Evolutionary Algorithms

EMI: Experiments in Musical Intelligence

EP: Evolutionary Programming

ES: Evolution Strategies

FSM: Finite State Machine

GA: Genetic Algorithms

GD: Gradient Descent

IA: Intel·ligència Artificial

LSTM: Long-Short Term Memory network

ML: Machine Learning

MLP: Multi-Layer Perceptron

NLP: Natural Language Processing

NN: Neural Network

RL: Reinforcement Learning

RNN: Recurrent Neural Network

SGD: Stochastic Gradient Descent

Glossari

Si no s’està familiaritzat amb el tema, alguns conceptes de IA poden ser complicats d’entendre. A continuació, intentarem conceptualitzar-ne alguns. Destacar que molts termes estan en anglès, ja que en el món de la programació és com s’utilitzen més.

Array: matriu, consisteix en una estructura de dades identificades amb un índex. Poden tenir tantes dimensions com faci falta, i permeten realitzar operacions matemàtiques.
Backpropagation: procés pel qual el loss d’una xarxa neuronal es va transferint en direcció contrària a l’habitual, actualitzant els pesos
Input: valor que es dona d’entrada
Llista: molt similar a un array d’una sola dimensió, la principal diferència és que fer operacions amb aquestes donarà error.
Loss: funció encarregada de calcular l’error en una NN. També s’anomena Cost.
Machine Learning: Branca de la IA que es centra en l’aprenentatge automàtic.
Multi-Layer Perceptron: un dels models més simples de NN, també anomenat feedforward neural network. Les seves connexions no formen cap mena de bucle, i les seves neurones tenen connexions directament a la següent capa.
Neural Network: xarxa neuronal, forma de ML caracteritzada per tenir diversos nodes (neurones) connectats entre si, imitant el cervell humà.
Output: valor de sortida
Weights: pesos interns de la xarxa, són els valors que les NN van ajustant a mesura que aprenen. Existeix un pes per cada connexió entre neurones.

Introducció

La música és un art complex, ple de normes, i que molts cops necessita anys d’estudi previ per poder comprendre-la. Les relacions entre les notes no sempre segueixen unes regles establertes, i té un gran paper la creativitat del compositor. Sembla complicat que un ordinador sol sigui capaç de crear una cançó, però existeix una branca de la informàtica, la intel·ligència artificial, que busca imitar el funcionament del nostre cervell, i que per tant, podria ser capaç de compondre música. En aquest treball, intentarem simplificar el complicat món de la intel·ligència artificial i de les xarxes neuronals, observant i aprenent com funcionen diferents tipus de models. Per tal d’aplicar aquests coneixements, es crearà una xarxa neuronal des de zero, sense utilitzar cap dels models ja disponibles a internet, amb l’objectiu de generar una melodia basada en diverses composicions per a piano.

Aquest és un treball fruit de la combinació entre la curiositat que em provocava el camp de la IA i l’amor per la música que tinc, havent-la practicat des que tenia 3 anys. L’objectiu principal serà, com he dit, desenvolupar una xarxa neuronal capaç de crear una melodia, però també tindrà com a objectius secundaris entendre realment com és capaç d’aprendre una xarxa neuronal, analitzar els diferents models ja existents capaços de compondre cançons, i descobrir si una intel·ligència artificial és capaç d’aprendre a crear música sense saber res de teoria musical.

L’estructura d’aquest treball es pot dividir en tres grans fases. En la primera, buscarem familiaritzar-nos amb els conceptes relacionats amb el tema i obtenir els coneixements que aplicarem més tard en l’elaboració del codi de la xarxa, elaborant un marc teòric. En aquest, veurem les diferents representacions informàtiques de la música, els diferents models d’IA i les aplicacions d’aquest camp en la música.

Seguidament, explicarem els processos seguits per tal de poder elaborar el producte, i diferenciarem les diferents modificacions que ha anat patint. Per últim, analitzarem els diferents resultats produïts i veurem quines millores es podrien aplicar a la nostra xarxa.

Partint pràcticament des del desconeixement sobre el funcionament de les xarxes neuronals, serà tot un repte crear-ne una sense pràcticament ajuda. Ja existeixen models capaços d’elaborar cançons, com veurem més endavant, però realment aquest treball busca respondre a una pregunta: pot una sola persona, partint des de zero i amb uns recursos limitats, crear una intel·ligència artificial funcional?

Objectius

Com hem vist abans, aquest treball gira entorn un objectiu principal:

- Crear una xarxa neuronal capaç de compondre una melodia.

Si bé aquest objectiu ha estat assolit per altres investigadors, la idiosincràsia d’aquest treball rau en el fet que s’utilitzen recursos bàsics, a l’abast de qualsevol usuari d’internet, i sense una formació específica en l’àmbit de la intel·ligència artificial.

Així doncs, per aconseguir aquest propòsit, es plantegen una sèrie d'objectius específics:

- Entendre com funciona el procés d’aprenentatge de les xarxes neuronals.
- Analitzar els models ja existents per a la creació de cançons.
- Comprovar si és possible que una IA creï música.

En definitiva, aquest treball no busca crear una gran melodia utilitzant models ja existents, sinó veure si és factible concebre una xarxa neuronal simple, les seves limitacions i els resultats que es poden obtenir.

Marc teòric

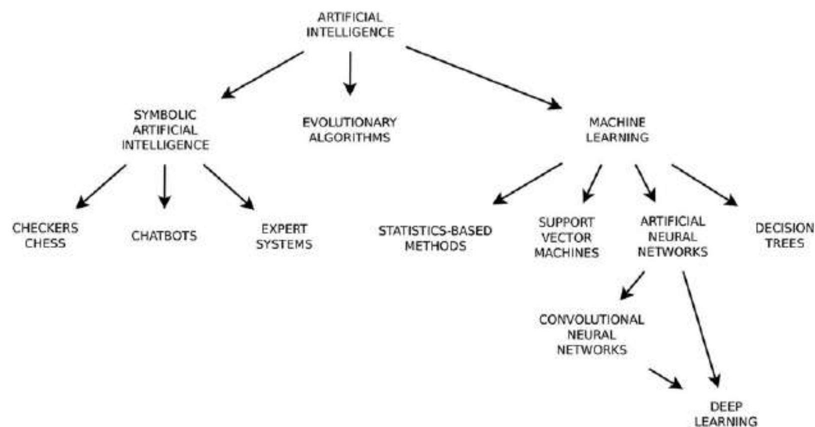
Per entendre com un ordinador és capaç d’aprendre per generar música, és important saber en què consisteix la intel·ligència artificial. A continuació, explicarem els diferents tipus de IA existents, la seva història i en veurem alguns exemples.

La intel·ligència artificial

En l’enciclopedia Britannica, Copeland (2020) defineix intel·ligència artificial com l’habilitat d’un ordinador o d’un robot controlat per ordinador de fer tasques associades amb la intel·ligència humana. Aquesta branca de la informàtica dedicada al perfeccionament d’algoritmes capaços d’imitar als humans ha evolucionat molt en els darrers anys. Dintre aquesta especialitat, en podem trobar diferents formes, cadascuna amb propòsits i funcionaments diferents. Aquestes es poden catalogar segons els processos que segueixen en diferents tipus.

Figura 1

Schematic overview of the main branches of artificial intelligence (AI), including machine learning (ML) methods which are having an impact on spine research



Nota. Per F. Galbusera, G. Casaroli & T. Bassani, 2019, https://www.researchgate.net/figure/Schematic-overview-of-the-main-branches-of-artificial-intelligence-AI-includin g_fig1_330878118

Francament, les diferents branques de la IA són complicades de catalogar i organitzar. Depenent del lloc web on busquis, et poden sortir classificacions completament diferents, però a continuació intentarem fer-ne una versió simplificada.

Symbolic Artificial Intelligence

La Intel·ligència artificial simbòlica va ser una de les primeres branques a sorgir. De fet, se la coneix també com a IA clàssica. És una intel·ligència artificial basada en símbols i regles, i aquest fet comporta bastants problemes, i és per això que es va deixar d’utilitzar.

El principal problema de la Symbolic Artificial Intelligence és que només funciona en uns entorns molt concrets, ja que es basa en expressar el coneixement humà seguint una sèrie de regles. Intenta imitar la forma que tenim els humans de representar el món utilitzant símbols (d’aquí el nom), i aplicant un seguit de lleis i regles. Està molt basada en la lògica, fet que fa que sigui fàcil d’interpretar per una persona. Però si es volgués utilitzar un programa d’aquests en un entorn molt generalitzat, la quantitat de símbols i regles que el creador hauria de subministrar-li seria molt gran (Dickson, 2019).

Tot i això, hi ha algunes situacions en les quals sí que s’utilitza aquest tipus d’AI: els problemes amb un gran nombre de restriccions, alguns programes de NLP (Natural Language Processing), problemes lògics i Expert Systems (Sistemes que intenten imitar a un expert en una àrea determinada, basats en una sèrie de coneixements) (TECHSLANG, 2020).

Un exemple de Symbolic AI que es fa servir actualment és la IA conversacional de l’empresa Inbenta. Aquesta permet a les empreses comunicar-se amb els seus clients per mitjà de chatbots basats en Symbolic AI i NLP.

Finalment, afegir que tot i que va ser el camp principal en el desenvolupament de les IA des dels anys 50 fins a aproximadament els anys 80, actualment s’utilitza molt poc i està pràcticament en desús.

Algoritmes evolucionaris

Els algoritmes evolucionaris (EA) són una branca de la IA que es va començar a desenvolupar a mitjans dels anys 50. El seu nom deriva del fet que es basen en el procés evolutiu (com el que els humans, per exemple, experimentem) per generar determinats algoritmes o solucions (De Jong, Fogel, Schwefel,1997).

Durant els anys 60 es van diferenciar tres camps principals dins d’aquesta branca: Evolutionary Programming, Genetic Algorithms i Evolution Strategies.

El primer programa d’EP va ser descrit per Fogel l’any 1966 en el seu llibre “Artificial Intelligence Through Simulated Evolution“. Alguns consideren aquesta branca com la base de l’Evolutionary Computation (camp dins la informàtica en el qual podem situar els EA). Cal tenir en compte que en aquells moments, la IA es centrava simplement en dos camps: la Symbolic Artificial Intelligence, i les NN (Neural Networks o xarxes neuronals).

El que feia el programa de Fogel, que ens serveix per entendre l’EP d’una forma molt bàsica, era crear autòmats finits (FSM en anglès) com si fossin organismes d’una població que intenta resoldre un problema. Aquests autòmats es situen en un medi (la seqüència de símbols que hi ha hagut fins a aquell moment) i reben una entrada. Aleshores, generen una sortida i la comparen amb el següent símbol d’entrada. Un cop han arribat al final, es comproven tots els símbols que cada autòmat ha donat i es puntuen. Els que tenen puntuacions més altes serviran com a base per la següent generació d’organismes, que sofreix una mutació. Aquest procés es va repetint tants cops com faci falta (De Jong, Fogel, Schwefel,1997).

Durant els anys 70 aquests algoritmes es van utilitzar sobretot en el reconeixement de patrons. Actualment, tot i que són útils en certs camps, no són dels més utilitzats.

D’altra banda, trobem els algoritmes genètics. Aquests són molt similars als algoritmes d’EP, però l’estructura del seu programa no és fixa. Els GA són molt similars al procés de selecció natural que trobem en la vida real, ja que es basen en aquest, igual que la resta d’EA. Per començar, tenim una població de diferents individus amb gens diferents. Se’ls proposa una tasca, i en funció de com la fan se’ls assigna un valor basat en l’habilitat que tenen per competir amb els altres. Aleshores, “s’aparellen” dos organismes basant-se en aquests valors i en un punt aleatori s’encreuen els seus gens. A més, també hi ha una possibilitat de que hi hagi mutacions, per garantir la variabilitat. Arriba un punt on tota la població té més o menys els mateixos gens, i aleshores és quan es dona per solucionat el problema (Mallawaarachchi, 2017).

Un exemple molt fàcil de visualitzar d'un programa utilitzant un GA el podem trobar a la següent pàgina web :

(*Evolution by Keiwan*, 2019)

Aquest petit joc, que utilitza tant un GA com una Neural Network, ens permet crear les nostres pròpies criatures i observar com van evolucionant per poder arribar el màxim lluny possible.

Finalment, l'últim dels tres primers camps dins d'aquesta branca és el de les Evolution Strategies. Aquest es centra a utilitzar processos similars als que hem vist en la resta de camps, imitant la selecció natural, per un vector de nombres reals. Molts cops es presenta com a una tècnica d'optimització de caixa negra (intenta optimitzar una funció per a una sortida determinada, sense realment fer cap suposició sobre la funció). En general és molt similar a la resta d'algoritmes evolutius, però té una particularitat que ha fet que rebés certa atenció en els últims anys (Weng, 2019).

En l'ES, els "pares" s'escullen aleatòriament, i llavors es fa una tria entre els descendents, que són els que passen a formar part de la següent generació.

Els algoritmes de ES s'estaven deixant d'utilitzar perquè es creia que no era possible implementar-los en problemes de grans dimensions. Però l'any 2017, l'equip d'OpenAi va trobar que l'ES era una alternativa, en alguns casos millor, al RL (del qual parlarem més endavant). En definitiva, molts casos era més ràpid, simple i eficaç utilitzar aquests algoritmes, que estaven pràcticament en desús. En l'article penjat a la seva pàgina web expliquen, a més, cada un dels avantatges i inconvenients d'utilitzar aquests programes (Karpathy, 2020).

És important saber que dintre els EA existeixen molts més camps, però no són tan rellevants com els tres presentats. Un exemple és l’evolució diferencial (DE), la neuroevolució o la programació genètica (GP). Tot i això, molts cops és complicat fer distincions entre programes que utilitzin EA, ja que les diferències entre els diferents camps en alguns casos són mínimes.

Machine Learning

Tot i que molts cops es confonen els conceptes de IA i ML, el Machine Learning és simplement un tipus d'intel·ligència artificial.

Machine Learning, de forma simplificada, permet a les màquines “aprendre” automàticament a interpretar, processar i analitzar diverses dades. En definitiva, els permet detectar paterns que es van repetint, i a partir d’ells fer prediccions. Arriba un punt on aquests algoritmes permeten actuar de formes per les quals no han estat programats (Alameda, 2019).

Sense ser-ne del tot conscients, trobem algoritmes de Machine Learning en el nostre dia a dia: des del feed d’Instagram o YouTube, fins als assistents virtuals o les respostes automàtiques. S’utilitzen molts cops per ensenyar-nos anuncis personalitzats. Per exemple, Google Ads ofereix una estratègia basada en Machine Learning per optimitzar al màxim el cost dels anuncis.

Certament, aquesta branca de la intel·ligència artificial és la que està més desenvolupada, i també la que promet més. Actualment s’està investigant i provant l’ajuda d’algoritmes per millorar el diagnòstic en certes malalties i en la interpretació de radiografies. Hi ha estudis que demostren com una IA, utilitzant un algoritme de Machine Learning, és capaç de detectar fractures traumàtiques toracolumbars a partir de radiografies sagitals (Rosenberg et al., 2021). Tot i que evidentment la paraula final l’ha de tenir un metge expert, aquests codis poden ajudar a fer el veredicte adequat.

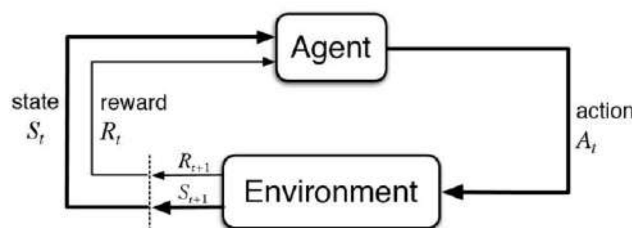
Dins la branca del Machine Learning, trobem tres formes d’aplicar-la. Es diferencien en la manera com s’entrena l’algoritme.

La primera és l’entrenament reforçat (Reinforcement Learning). En aquest mètode, l’ordinador aprèn a base de prova i error. No se li dona cap directriu de quin és el camí correcte que ha de seguir: l’algoritme sol és capaç de descobrir-lo. Es pot entendre com el procés d’entrenament d’un gos, que rep càstigs o llaminadures en funció del seu comportament.

En l’entrenament reforçat, ens trobem un agent que du a terme una acció, i seguidament mira l’estat en el qual es troba i quina recompensa li dona. Aquestes recompenses s’acaben “guardant” dins l’entorn on es troba l’agent. Quan arriba al resultat final, si ho aconsegueix, la màquina avalua aquestes recompenses. Aquest procés tindrà lloc múltiples vegades, i el codi anirà entenent quines són les accions que li donen millors recompenses (Osiński & Budek, 2021).

Figura 2

Les relacions agent-medi en el RL



Nota. En el temps t, l’agent rep informació de l’estat del medi (St), i tria una acció (At). L’impacte d’aquesta acció sobre el medi es veu reflexat en la recompensa (Rt). De Modeling Biological Agents Beyond the Reinforcement-learning Paradigm, per O. L. Georgeon, R. C. Casado & L. A. Matignon, 2015, https://www.researchgate.net/publication/289991380_Modeling_Biological_Agents_Beyond_the_Reinforcement-learning_Paradigm

Aquest procés de Machine Learning és el que s’utilitza per crear bots capaços de jugar a videojocs o jocs de taula, per exemple. Un bot és un programa informàtic capaç d’efectuar tasques. Si se li afegeix una intel·ligència artificial d’aquest tipus, pot arribar a ser capaç de guanyar en jocs com els escacs, go, o d’obtenir les màximes puntuacions a videojocs com Mario. Evidentment, hi ha diverses formes d’aplicar el RL, amb molts noms diferents. Però possiblement el més bàsic és el Q-Learning, una forma d’entrenament reforçat on els valors de recompensa es van emmagatzemant en una taula. Una variant d’aquest s’utilitza en els cotxes de conducció autònoma a l’hora de canviar de carril.

Dos exemples de Reinforcement Learning són AlphaGo Zero, un algoritme pel joc de taula go que en tan sols 40 dies va ser capaç de guanyar a la seva versió anterior, que ja havia superat al campió del món (Silver et al., 2017); i AWS DeepRacer, un cotxe autònom a escala creat específicament per provar models de RL i desenvolupat per Amazon.

La segona és l'entrenament supervisat. En aquest cas, la màquina no és totalment independent, a diferència del cas anterior, i les dades estan etiquetades. En definitiva, els algoritmes d'entrenament supervisat, que és la forma més fàcil i comuna de ML, aprenen a partir d'exemples.

Les dades que haurà d'agafar un codi d'aquest estil seran dades d'entrada, juntament amb les seves corresponents sortides correctes. A partir d'aquí anirà entrenant per poder generar el resultat d'una entrada nova. Existeixen dos tipus d'algoritmes d'aquest tipus: els de classificació i els de regressió (Wilson, 2019).

Si hi ha una relació entre les dades d'entrada i les de sortida, s'utilitzen algoritmes de regressió. Aquests normalment es dediquen a predir una nova sortida, utilitzant alguns dels algoritmes següents:

- Linear regression
- Regression trees
- Non-Linear Regression
- Bayesian Linear Regression
- Polynomial Regression

Els altres algoritmes d'entrenament supervisat són els de classificació. Aquests es fan servir per dividir les dades en grups. Els algoritmes utilitzats són:

- Logistic regression
- K-nearest neighbors
- Support Vector Machines
- Kernel SVM
- Naïve Bayes
- Decision Tree Classification
- Random Forest Classification

Per últim, trobem l'entrenament no supervisat. En aquest, els algoritmes no es dediquen a buscar patrons i etiquetes, sinó que busquen similituds. Intenten agrupar les dades en grups que s'assemblin, però a diferència de l'aprenentatge supervisat, desconeix els grups. Es basa en la idea que la majoria de la informació que arriba al nostre cervell no està classificada, i busca crear una intel·ligència artificial capaç d'autoorganitzar-se (Barlow,

1989). Els objectius acostumen a ser o bé clustering (agrupar les dades en grups) o density estimation (mostrar com estan distribuïdes les dades).

El clustering és una gran part de l’entrenament no supervisat. Es basa a agrupar les dades en grups, segons, per exemple, la distància a la qual es troben. Existeixen 4 tipus principals d’algoritmes de clustering:

- Exclusive clustering (K-means)
- Overlapping clustering (Fuzzy K-means)
- Hierarchical clustering
- Probabilistic clustering (Mixture of Gaussians)

D’altra banda, el density estimation, com el seu nom indica, intenta agrupar les dades observant en quins punts s’agrupen més dades o menys. Un exemple seria un programa que es dediqués a crear histogrames.

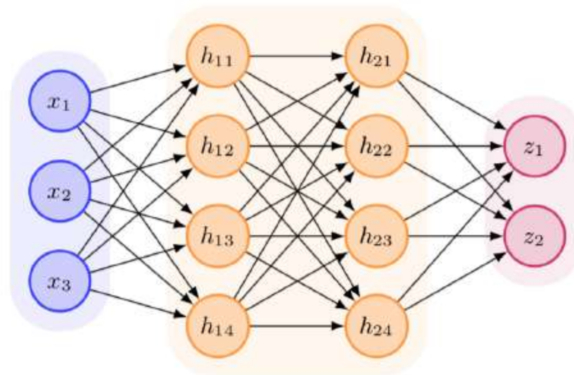
L’entrenament no supervisat és molt més complicat que el supervisat, però hi ha algunes situacions en les quals és necessari i molt útil. Aquestes són principalment, aquelles on no sabem quantes “etiquetes” hi ha entre les dades que tenim, aquelles en les que hi ha massa etiquetes com per a introduir-les totes (com el reconeixement de veu) o simplement volem saber com s’estructuren les dades de les quals disposem (S. Mishra, 2018).

Neural Networks (xarxes neuronals)

Les Neural Networks (NN o ANN, d’Artificial Neural Network) són models d’intel·ligència artificial pensats per imitar com funciona el cervell humà (d'aquí el nom de xarxes neuronals). Funcionen utilitzant “neurones” interconnectades, que reben dades (inputs) i en produeixen de noves (outputs). Les xarxes neuronals també contenen un sistema intern de pesos i biases, que són variables que es regulen per alterar la sortida de les neurones.

Figura 3

Esquema general d'una xarxa neuronal



Nota. Estructura general d'una xarxa neuronal, amb les diferents capes i neurones. De *How Neural Network Process Your Input (Trained Neural Network)*, per M. Ryan, 2018, (https://medium.datadriveninvestor.com/how-neural-network-process-your-input-trained-neural-network-fd48f1bf310)

Els inicis de la investigació en ANN es remunten a l'any 1943, quan McCulloch i Pitts van proposar un sistema lògic per representar l'activitat que duien a terme les neurones en el cervell (McCulloch & Pitts, 1943). Més tard, l'any 1958, Rosenblatt va crear un model computacional ideat per simular la capacitat del cervell per discriminar, anomenat Perceptron (Rosenblatt, 1958). Aquest es pot considerar el primer model d'ANN, encara que es tracti tan sols d'un model binari (la seva predicció era o bé 0 o 1) i d'una sola neurona. Aquesta neurona té un funcionament molt senzill: si la suma dels inputs multiplicats pels seus pesos corresponents és major a cert nombre, el resultat és 1. Si no, el resultat es queda en 0 (Ibañez, 2020).

Malgrat aquests primers èxits, ja cap als anys 60 l'ús de les xarxes neuronals va disminuir. Per començar, es creia que no es podien crear xarxes neuronals de capes múltiples. A més, els èxits inicials havien fet que s'exagerés el potencial de les màquines, les qüestions filosòfiques sobre les "màquines pensants" i l'ús de funcions defectuoses van provocar que el finançament en aquest àmbit es reduís (Neural Networks - History, 2000) .

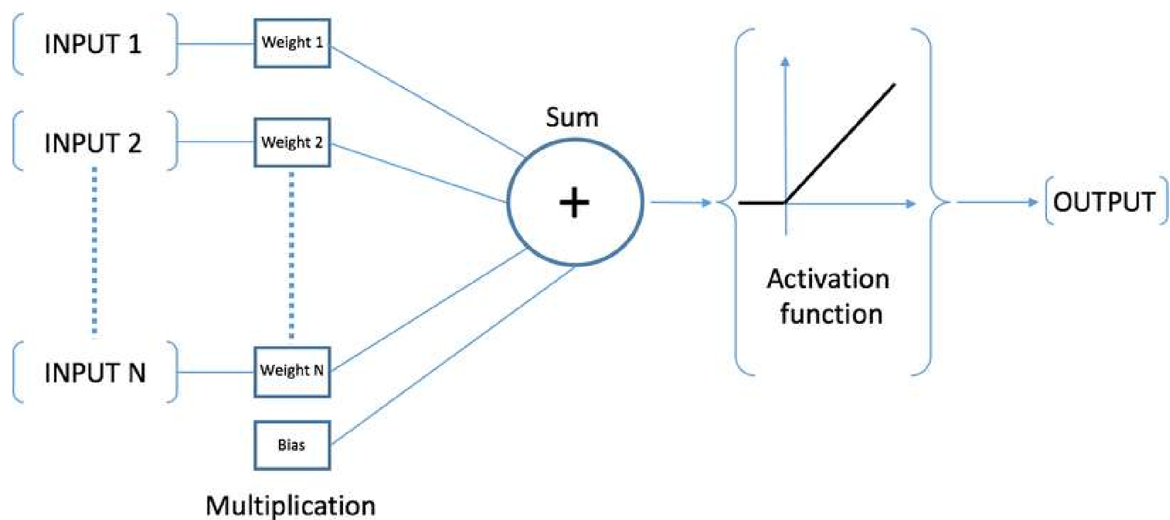
L'any 1975 es va aconseguir crear, finalment, la primera xarxa multicapa. L'any 1982, el camp de les NN va tornar a rebre més finançament, gràcies a certs avenços i a una espècie de competitivitat entre els Estats Units i el Japó. L'any 1986 va sorgir el terme Back-Propagation, del qual parlarem més endavant.

En els últims anys, les xarxes neuronals han guanyat molt protagonisme. Cada cop es publiquen més articles respecte a aquest tema, i són moltes les empreses que es dediquen a investigar aquest camp. De fet, sembla ser un dels camps més prometedors dins la Intel·ligència Artificial (Pons, 2018).

Les ANN més senzilles consten de tres capes: una d'entrada, una oculta, i una de sortida. Cada capa conté les seves neurones, que estan connectades amb totes les de la següent capa. Cada connexió entre les neurones té un pes associat.

Figura 4

Esquema del funcionament d'una neurona



Nota. De "Machine Learning Approach for Prediction of Hematic Parameters in Hemodialysis Patients", per C. Decaro, G. B. Montanari, R. Molinariz & A. Gilberti, 2019, *(https://www.researchgate.net/figure/Working-principle-of-an-artificial-neuron-Artificial-neural-network-ANN-22-is-a_fig1_335867733)*

Com hem dit abans, les neurones reben i produeixen dades. Cada una neurona (que no sigui de la primera capa) rep tots els outputs de les neurones de la capa anterior multiplicats pels pesos específics de cada connexió. Un cop tots arriben dintre la neurona, es sumen juntament amb el bias. Llavors entren en joc les funcions d'activació (Decaro et al., 2019).

Les funcions d'activació simulen l'activació de la neurona en el cervell. Normalment, si un nombre és més gran que un llindar determinat, la neurona "s'activa", passant un valor relativament gran. En canvi, si el nombre és més petit, la neurona passa un nombre petit, molts cops proper a zero. És molt útil en les xarxes neuronals, ja que és la base que permet que es puguin resoldre problemes no lineals (Wood, 2020). Existeixen diverses funcions per

dur a terme aquesta tasca, a més senzilla de les quals, i més fàcil d’explicar, és la Step Function. En aquesta funció, si els valors ja sumats són més petits de zero, la neurona no s’activa, passant un 0. Si no, passa un 1 (Kinsley & Kukieła, 2020).

Altres funcions d’activació són:

- Linear
- Logistic Activation Function / Sigmoid
- Tanh
- ReLu
- Leaky ReLu
- ELu
- Softplus
- Softmax

Cal dir que existeixen dos tipus de funcions d’activació: les que s’utilitzen en les capes intermèdies, i les que s’utilitzen en l’última capa. Normalment, les funcions utilitzades en la última capa són la Sigmoid (per a classificació binària) o la Softmax (classificació multiclasse) (Ronaghan, 2019).

Les funcions d’activació poden variar bastant depenent del problema al qual vulguem fer front. Això provoca que molts cops no existeix una sola funció que sigui millor que la resta, ja que totes presenten diferents punts positius i negatius. La majoria dels cops, reben els outputs anteriors ja multiplicats i sumats, i la sortida que donen és un valor entre 0 i 1, o entre -1 i 1.

Una altra funció molt important en les NN, és la Loss function (també anomenada Cost function). Aquesta serveix per determinar com d’equivocada està la xarxa neuronal en la seva predicció, per poder ajustar els pesos posteriorment. Per tant, s’utilitza després de l’última funció d'activació (que realitza la predicció). Com més baix és el nombre de sortida de la Loss function, menor serà l’error que tindrà en les seves prediccions (Pere, 2020).

Abans d’explicar el Loss, però, cal entendre per què no s’utilitza simplement el percentatge d’encert que té la màquina. Tot i que això seria possible, no és gaire útil, ja que les NN prediuen un valor segons la “confiança” que tenen en què sigui correcte. Si ens baséssim només en l’encert que tenen, estaríem donant la mateixa importància als valors que ha predit amb una confiança del 70%, per exemple, als que ha predit amb una confiança del 30%.

Existeixen tres tipus de Loss functions principals: les de regressió, les de classificació binària i les de classificació multiclasse. Depenent de l’objectiu de la nostra NN, utilitzarem una funció d’un grup o d’un altre. Les principals funcions per cada problema són les següents (Brownlee, 2020a):

Regressió:
- MSE
- MSLE
- MAE

Classificació binària:
- Binary Cross-entropy
- Hinge Loss
- Squared Hinge Loss

Classificació multiclasse:
- Multi-Class Cross-Entropy
- Sparse Multiclass Cross-Entropy
- KL Divergence

Un cop entès el Loss, cal veure com aprenen realment les xarxes neuronals. Aquestes, en definitiva, van ajustant certs paràmetres per obtenir un resultat cada cop millor. Aquest pas és el principal problema al qual s'enfronten les NN. Es tracta de mirar de quina forma es poden modificar els pesos i biases per tal de reduir el Loss. Aquest procés s’anomena optimització (Optimitzation).

Abans de parlar sobre els diferents Optimizers, ens falta veure un altre dels paràmetres que conformen les xarxes neuronals: el learning rate. Aquest valor ens indica quant canviem els pesos en funció de l’error que ha tingut la nostra predicció. Ajustat correctament, aquest valor pot ajudar molt al rendiment de la NN, però si el valor és massa gran, o massa petit, pot dificultar arribar a una bona predicció. En definitiva, el learning rate pot tenir una gran influència en els resultats de les xarxes neuronals (Brownlee, 2020b).

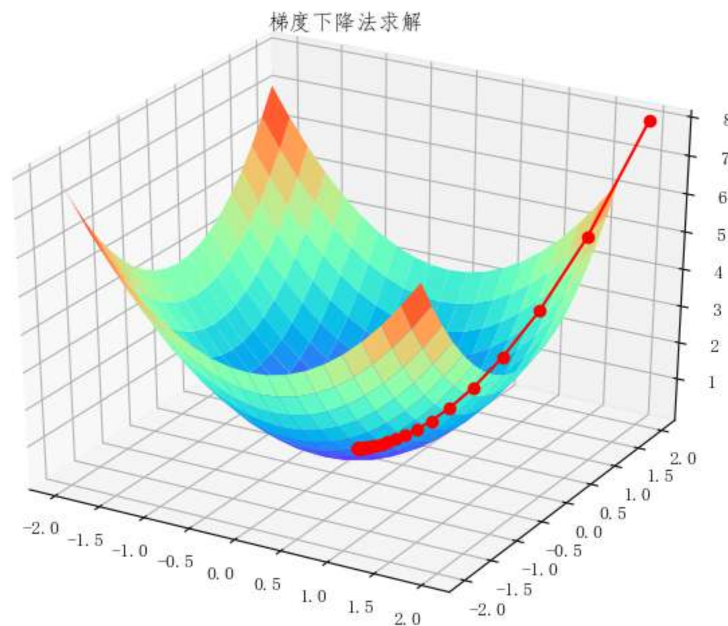
Ara sí, parlem dels optimizers. Abans hem dit que un optimizer, en el context de les xarxes neuronals (ja que realment existeixen optimizers per a molts casos), serveix per mirar de quina forma podem modificar els paràmetres per tal d’obtenir millors resultats (reduir el

loss). Existeixen diversos tipus d’algoritmes capaços de fer aquesta feina i que s’utilitzen en NN, cadascun amb els seus propis avantatges i inconvenients.

El més bàsic de tots, i també un dels més utilitzats és el Gradient Descent. En general, es tracta d’un algoritme que serveix per trobar el mínim d’una funció. Per aconseguir això, busca el gradient, i “avança” en la direcció contrària modificant els pesos i els biases (el learning rate afecta simplement a si avança molt o poc) (Ruder, 2016).

Figura 5

Funcionament del GD



Nota. De *Machine Learning Training Method: Gradient Descent Method*, per O. Zhao, 2021, (https://forum.huawei.com/enterprise/en/machine-learning-training-method-gradient-descent-method/thread/708303-895?page=1&authorid=2976461)

Per entendre el Gradient Descent cal entendre primer què és el gradient. Aquest és un vector compost per les diferents derivades parcials d’una funció calculades respecte cadascun dels inputs. En matemàtiques, es denota utilitzant el símbol ∇. Vindria a ser similar al pendent d’una funció, però és vectorial i no escalar (Difference between Slope and Gradient, 2012).

El gradient serveix perquè apunta cap al punt on la funció augmenta més. Per tant, si retrocedim en aquesta mateixa direcció repetidament, arribarem en el punt on la funció té el seu mínim. Aquest és el principi en el qual es basa el Gradient Descent. El primer a utilitzar aquest mètode va ser Cauchy l’any 1847, molt abans de l’existència de les intel·ligències

artificials (tot i que ell utilitzava les derivades parcials i no el gradient, ja que aquest últim no es va inventar fins més tard) (Cauchy, 1847).

Per poder calcular aquest gradient en les xarxes neuronals, s’han de resoldre un gran nombre d’operacions, ja que s’han de calcular les derivades parcials de totes les funcions respecte als inputs que aquestes reben, i molts cops aquests són matrius, fet que encara dificulta més la tasca. Aquest procés s’anomena backpropagation, ja que consisteix a passar el loss per la xarxa neuronal des de la capa d’output fins a la capa d’input (anant cap endarrere, d’aquí el nom). Cal saber també, que és molt important la velocitat a la qual els programes de xarxes neuronals són capaços d’executar-se, ja que més velocitat significarà un major entrenament en menys temps. És per això que en alguns casos, en lloc d’utilitzar el Gradient Descent (GD) s’utilitza una altra tècnica: la Stochastic Gradient Descent (Robbins & Monro, 1951).

L’algoritme de Stochastic Gradient Descent és molt similar, de fet, al Gradient Descent. La principal diferència que té respecte a l’anterior és que, mentre que el GD itera per totes les mostres d’entrada, l’Stochastic s’utilitza una sola mostra. En definitiva, el GDés més precís, però més lent, mentre que SGD és més ràpid, però no minimitza tant l’error (Ruder, 2016).

Alguns optimitzadors són variants d’aquests dos, com per exemple el Mini-Batch Gradient Descent, que divideix les mostres en grups i calcula una aproximació. Altres optimizers, més complexos, són:

- Adam
- AdaGrad
- RMSProp
- Adadelta
- Nadam
- AdaMax

Però les xarxes neuronals, en alguns casos, segueixen sense funcionar del tot correctament, per molt que l’optimizer sigui adequat i tot funcioni correctament. Aquests problemes, molts cops, es deuen a overfitting o a underfitting. En aquests casos, s’utilitzen tècniques de regularització (Regularization), com introduir nous paràmetres que penalitzin l’overfitting (L2 Regularization) (M. Mishra, 2018).

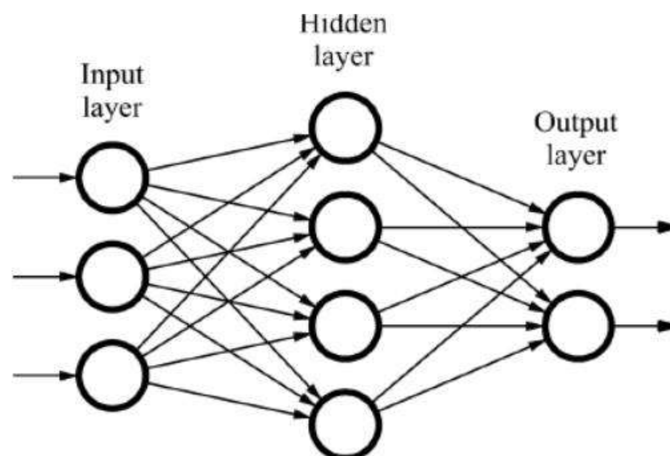
Un cop vistes les diferents funcions que conformen les xarxes neuronals, podem tornar a explicar més o menys com funciona una xarxa neuronal simple amb 3 capes: una d'entrada, una "hidden layer" (on tenen lloc la majoria dels càlculs) i una de sortida.

Les NN reben una sèrie de dades a la capa d'entrada. Cada una d'aquestes dades es multiplica pels pesos (que són valors associats a les connexions entre neurones), i es sumen als biases. A aquesta suma li direm z. Dins la neurona, la funció d'activació rep z, i passa un output. Els outputs d'aquestes neurones seran els inputs de la següent capa, que en aquest cas ja és l'última, i es tornaran a multiplicar per uns altres pesos i biases. En aquesta capa, la funció d'activació serà diferent, ja que intentarà predir una sortida.

Un cop la xarxa neuronal faci la seva predicció, es calcularà l'error amb la Loss function (que com a paràmetre rep la sortida de l'última capa, i els valors "correctes"). Quan tingui l'error, la xarxa neuronal s'ajustarà (simplement modificant els pesos aleatòriament, o utilitzant un optimizer i backpropagation per reduir-lo). Aquest procés es va repetint (cada repetició s'anomena epoch), entrenant la xarxa neuronal. Un cop està entrenada, s'utilitza per predir utilitzant dades noves per les quals no ha estat entrenada.

Figura 6

Capes d'una xarxa neuronal



Nota. Graph of a feed-forward neural network, de "Computational modeling of machining systems", per R. Quiza, J. P. Davim, 2009, (https://www.researchgate.net/figure/Graph-of-a-feed-forward-neural-network_fig2_234055140)

En funció del nombre de neurones, capes i mecanismes que utilitzen les xarxes neuronals es poden classificar de moltes formes diferents. Els tipus principals de xarxes neuronals són tres (Pai, 2020).

- Les Feedforward Neural Networks, també anomenades Multi Layer Perceptron, que són la forma més senzilla de xarxa neuronal. El seu funcionament és bàsicament idèntic a l'explicat anteriorment.

- Les Convolutional Neural Networks, que estan adaptades al reconeixement d'imatges. La forma en la qual funcionen és utilitzant una capa de convolució, que aplica un "filtre" a les dades d'entrada. Aquest filtre es pot entendre com una graella de valors que es situa sobre els píxels de la imatge, i es multipliquen els valors dels píxels pels del filtre per generar un output. A partir d'aquest punt, funciona com una xarxa neuronal normal.

- Les Recurrent Neural Networks, que incorporen un loop que permet que els valors de la iteració anterior també es tinguin en compte. S'utilitzen sobretot en NLP.

Deep learning

El Deep Learning, en català Aprenentatge Profund, és una forma de Machine Learning basada en les Neural Networks. La principal diferència és que perquè es consideri deep learning, la xarxa ha de tenir mínim 3 hidden layers (AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?, 2021).

Aquesta forma de IA permet trobar les complicades formes en les quals s'estructuren grans quantitats de dades. Molts cops, els algoritmes de Deep Learning utilitzen backpropagation per ajustar l'enorme quantitat de paràmetres que tenen.

El Deep Learning també minimitza més el treball humà que les xarxes neuronals més simples, ja que molts cops automatitza directament la classificació de les dades. Això permet que sigui escalable (que es puguin utilitzar grans quantitats de dades, a gran escala) i que no necessiti sempre dades etiquetades. En definitiva el deep learning és una versió més automatitzada de les xarxes neuronals que en molts cops és millor, però que necessita més dades per poder funcionar correctament.

Exemples d’AI

El camp de la intel·ligència artificial promet molt. Cada cop apareixen programes més impactants i més intel·ligents per dir-ho d’alguna manera. És per això que m’agradaria mostrar alguns exemples de fins a quin punt han arribat les IA en el present.

El primer exemple del qual m’agradaria parlar és GPT-3 (Brown et al., 2020). Aquest és un model de NLP creat per l’empresa Open AI i actualment és el més potent del món, amb una capacitat de 175.000 milions de paràmetres. En definitiva, és un programa capaç d’entendre el llenguatge semànticament i de produir textos molt similars als creats per un humà. Algunes de les seves facultats són generar articles de notícies, traduir textos i respondre a preguntes, entre d’altres.

Aquest model de NPT s’ha utilitzat per crear algunes aplicacions verdaderament increïbles. Una d’aquestes aplicacions s’anomena GitHub copilot, que es basa en un model derivat de GPT-3 anomenat Codex, que està entrenat utilitzant més exemples de codi que el model original (Zaremba et al., 2021). Aquesta aplicació utilitza la capacitat de GPT-3 per entendre el llenguatge i la gran quantitat de codi guardat a la web GitHub per generar codi en un context. Per exemple, és capaç de crear un programa senzill a partir d’una simple frase, o bé entendre part del codi que has escrit per autocompletar-lo, pot crear funcions noves simplement amb el nom… En definitiva, és una gran eina que pot ajudar als programadors a ser més eficients a l’hora de treballar i de trobar errors en el seu codi.

Una altra app basada en aquest programa és Dall-e (OpenAi, 2021). Aquesta aplicació és capaç de generar una imatge a partir d’una frase. Tot i que de moment no ha sortit el codi, i simplement es poden veure els exemples que hi ha a la seva web, té pinta a ser molt interessant, i és molt possible que sigui la base per a il·lustracions generades per ordinador en un futur.

Un cop resolts els dubtes pel que fa als conceptes d’IA, se’ns planteja un dubte per tal d’arribar a la composició musical per ordinador. Com és capaç l’ordinador d’interpretar la música?

Representacions del contingut musical

Per tal que un ordinador sigui capaç de generar música, primer ha de ser capaç d’interpretar-la. És per això que existeixen diferents formes de representar el contingut musical per tal que sigui llegible per un ordinador.

Dins aquestes representacions, en trobem dos grans grups: les representacions dels senyals d'àudio i les representacions simbòliques (Briot et al., 2019).

Representacions dels senyals d'àudio

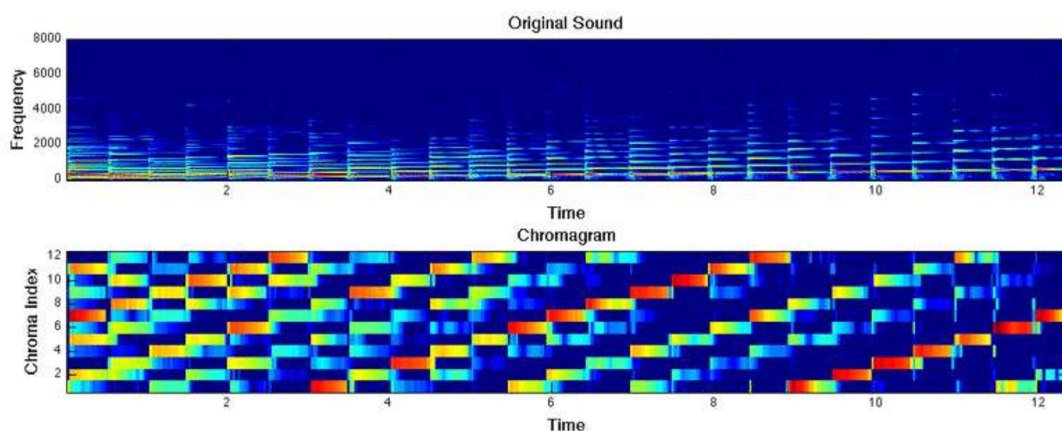
Les representacions de senyals d’àudio s’utilitzen per tal que l’ordinador sigui capaç d’entendre tota mena de sons, no només música. Aquests arxius contenen bàsicament les vibracions que produeixen els sons.

Existeixen dos tipus de representació d’aquestes vibracions: en la seva forma pura (utilitzant directament les formes d’ona dels sons), o amb alguna transformació.

Les transformacions més típiques d’aquestes ones són utilitzant les transformacions de Fourier (que crea un espectrograma). En la música, també existeixen unes representacions anomenades Chroma feature, que “classifiquen” els sons en les 12 notes de l’escala.

Figura 7

Espectrograma i chroma feature



Nota. Espectrograma (part superior) i cromagrama (part inferior) d’una escala ascendent. De “An Introduction to Fourier Analysis with Applications to Music”, per N. Lenssen & D. Needell, 2014, (https://www.researchgate.net/figure/The-spectrogram-top-and-chromagram-bottom-of-an-ascending-scale_fig3_314918556)

Representacions simbòliques

Les representacions simbòliques de la música agafen directament els conceptes clau de la música, com les notes, el ritme, el tempo… i els representen utilitzant diferents formats, que permeten que l'ordinador els entengui (Briot et al., 2019). Els principals són:

- Piano roll
- Text (ex. Notació ABC)
- MusicXML
- MIDI

Piano roll

El piano roll té els seus inicis a principis del segle XX. En aquella època, s'utilitzaven rotlles de paper amb forats per a operar automàticament els pianos. Basant-se en aquests papers, es va crear la representació de piano roll, que permet visualitzar la música de forma geomètrica en una gràfica nota-temps. Cada nota emmagatzemada en aquest tipus de representació té tres valors: el to, el temps i la durada de la nota (Müller, 2015).

Notació ABC

La notació ABC permet representar la música en un fitxer de text. En els arxius d'aquesta notació es diferencien dues parts: la capçalera i les notes.

La capçalera conté informació sobre l'autor de la peça, el to en el qual està, el compàs…

L'altra part dels fitxers conté les notes representades utilitzant lletres i l'alfabet musical. Si la lletra està en majúscula, és de l'octava més baixa. Si és en minúscula, de l'octava intermèdia, i si conté un apòstrof, forma part de l'octava alta. La durada de la nota es representa amb el número que la segueix (si no té cap número, s'interpreta que és 1) i una variable de la capçalera. Els compassos es representen amb un símbol | (Walshaw, 2010). Un exemple d'aquest tipus de fitxer seria:

```
X:1
T:Paddy O'Rafferty
C:Trad.
M:6/8
K:D
dff cee|def gfe|dff cee|dfe dBA|dff cee|def gfe|faf gfe|1 dfe dBA:|2 dfe dcB|]
~A3 B3|gfe fdB|AFA B2c|dfe dcB|~A3 ~B3|efe efg|faf gfe|1 dfe dcB:|2 dfe dBA|]
fAA eAA|def gfe|fAA eAA|dfe dBA|fAA eAA|def gfe|faf gfe|dfe dBA:|
```

MusicXML

MusicXML és un llenguatge basat en el metallenguatge Extensible Markup Language. Aquest format és l'estàndard utilitzat pels programes de creació de partitures. Existeixen més de 250 aplicacions que admeten aquest format (MusicXML for Exchanging Digital Sheet Music, 2020).

Un exemple d’un fitxer d’aquest tipus és el següent:

Figura 8

Inside a MusicXML file...

```
      <unpitched>
        <display-step>E</display-step>
        <display-octave>5</display-octave>
      </unpitched>
      <duration>2</duration>
      <instrument id="P15-X8"/>
      <voice>2</voice>
      <type>16th</type>
      <stem>down</stem>
      <beam number="1">end</beam>
      <beam number="2">end</beam>
    </note>
  </measure>
  <!--
```

Nota. De *Importing MusicXML scores*, per Bol Processor, 2021, (https://bolprocessor.org/importing-musicxml/

Fitxers MIDI

La representació simbòlica més utilitzada per a programes d'intel·ligència artificial és el MIDI. Aquestes sigles signifiquen Musical Instrument Digital Interface, i es tracta bàsicament, d'un llenguatge creat per poder comunicar els instruments amb els ordinadors.

La primera versió estàndard de MIDI va ser creada l'any 1982 per Dave Smith i Ikutaru Kakehash (The MIDI Association, 2021). Originàriament, i durant molts anys, va ser la principal forma de comunicació entre diversos equipaments musicals, especialment sintetitzadors. La part interessant d'aquest llenguatge és que funciona enviant codi, i no ones de so com un podria originàriament pensar. Mitjançant això, és possible utilitzar aquests fitxers per tal que un ordinador els llegeixi, i els converteixi a números.

Feta aquesta petita introducció sobre els fitxers en si, ens endinsarem una mica més en els seus paràmetres, i com funcionen. Els fitxers MIDI s'organitzen en Tracks, que tenen diversos canals que contenen missatges escrits en codi binari, que es van enviant successivament. Essencialment existeixen dos tipus de missatges, els missatges de canal, i els de sistema (Sionsoft, 2020).

Missatges de sistema (System Messages)

Els missatges de sistema contenen informació sobre tot el fitxer, i n'hi ha de tres tipus:

Els System Common Messages contenen informacions diverses que hagin d'arribar a tots els dispositius que estan sent controlats. Els principals són:

- MTC: S'utilitza en la sincronització entre l'equipament MIDI i, per exemple, imatges o vídeos.
- Song Select Message: s'utilitza en dispositius que continguin més d'una cançó, permet escollir-ne una en concret.
- Song Positioner Pointer: permet començar una cançó des d'un punt específic.
- Tuning Request, EOX… (menys importància).

Els System Real Time Messages s'utilitzen per sincronitzar tots els dispositius que necessitin tenir un temps controlat, com poden ser les caixes de ritmes. Els principals

missatges d'aquest tipus són Timing Clock, Start, Continue, Stop, Active Sensing, i el System Reset message. Tots ells tenen funcions bàsiques que es poden deduir directament a partir del seu nom.

Finalment, els System Exclusive Messages són, possiblement, els missatges més estranys de tots. Són aquells missatges que permeten als fabricants de dispositius "crear" els seus propis missatges, que permeten editar paràmetres inaccessibles en el codi normal, fer accessible qualsevol funcionalitat dels equips… (The MIDI Association, 2021b)

Missatges de canal (Channel Messages)

Els missatges de canal en els MIDI són els encarregats de la part musical. Contenen informacions tant per les notes, com per com s'han de tocar aquestes, els ritmes, el tempo, com reacciona el dispositiu a les notes… Tenen la característica que només afecten el canal en el qual es troben.

De la mateixa forma que en els missatges de sistema, en trobem diferents tipus. En aquest cas, però, només n'hi ha dos: els Channel Voice Messages i els Mode Messages.

Els Channel Voice Messages constitueixen la part més àmplia d'aquests tipus de missatges. Són els encarregats de tota la part interpretativa, i n'hi ha de molts tipus diferents. A continuació, us exposem un llistat dels més importants.

- Note On: missatge que apunta que una nota inicia. Ha d'anar seguit de bytes de data més: l'indicador de la nota (en un teclat, quina tecla s'ha premut) i velocitat (com de fort s'ha tocat aquella tecla).
- Note Off: indica que una nota acaba.
- Aftertouch: indica la pressió que s'ha fet sobre la nota en els dispositius compatibles.
- Pitch Bend: permet alterar el to en els instruments compatibles, ha d'anar seguit de dos bytes que permeten que els canvis siguin fluids.
- Program Change: especifica quin instrument ha de tocar les diferents notes del canal.
- Control Change: permet controlar les diverses funcions que té el sintetitzador. Varien bastant en funció del fabricant, però n'hi ha una gran varietat. (The MIDI Association, 2021b).

Els Channel Mode Messages, permeten controlar bàsicament la forma en la qual el sintetitzador interpreta els missatges. Poden modificar aspectes com la polifonia, els canals…

Coneixent els diferents tipus de missatges, ens podem fer una idea de què és el que conforma un fitxer MIDI. Ara ja coneixem com l’ordinador pot interpretar la música, però encara ens falta un pont entre aquest tipus de representació i el codi.

Llibreria MIDO

Per fer aquest pas, existeixen diverses llibreries de Python (el llenguatge de programació que utilitzarem). Una llibreria es pot entendre com una espècie de programa, que ens permet tenir més eines dins d’un mateix llenguatge de programació.

Una d’aquestes llibreries, la llibreria MIDO, ens permet llegir i modificar els fitxers MIDI. Però per tal d’organitzar la gran quantitat de missatges diferents que poden contenir aquests tipus d’arxius, dins la llibreria estan classificats d’una altra forma.

Dins la llibreria MIDO hi ha els missatges, i els metamissages. El primer grup comprèn els Channel Voice Messages i els System Messages. Per tal d’escriure’ls en codi Python, es pot posar simplement el següent:

mido.Message(tipus, paràmetres)

Com podeu observar, a part del tipus de missatge, ens demana uns paràmetres. Aquests varien en funció del tipus de missatge que sigui, i poden ser els següents seguint la documentació de la llibreria (Message Types — Mido 1.2.10 Documentation, 2020):

- Channel
- Frame_type
- Frame_value
- Control
- Note
- Program
- Song
- Value

- Velocity
- Data
- Pitch
- Pos
- Time

Els metamissatges (Meta Messages) són missatges que contenen, normalment, diferents metadates, que permeten tant modificar diferents aspectes de com sonarà la cançó, com el tempo, com introduir els noms dels diferents canals i text dins el fitxer MIDI. A l’haver-n’hi de tantes formes diverses, a continuació us explicarem una mica per sobre els més comuns.

- Text: emmagatzema un text qualsevol.
- Track_name: permet assignar a cada pista un nom concret.
- Instrument_name: guarda el nom de l’instrument que està sonant. És útil per programes informàtics que simulen el so dels diferents instruments.
- End_of_track: marca el final de la pista.
- Set_tempo: indica el tempo de la cançó, o el modifica.
- Time_signature: indica el compàs.

De la mateixa forma que els Messages, contenen paràmetres, però aquests són molt específics de cada missatge.

Vistes les formes en les quals un ordinador és capaç de representar el contingut musical, toca parlar una mica de la història de la composició automatitzada, i veure alguns exemples actuals que utilitzen IA.

Intel·ligència artificial aplicada a la música i composició automatitzada

Hem vist com són capaços els ordinadors d’aprendre i generar dades que mai han vist mitjançant Machine Learning, i també hem repassat les diferents formes en les quals poden les màquines interpretar la música. Ara ens toca veure com es poden unir aquests dos camps per tal d’assolir un objectiu: la generació de música automatitzada.

És realment complicat definir què és la música, però es podria dir que es tracta de la ciència o art d'ordenar tons o sons en successió, combinació i en relacions temporals per tal de produir una composició que tingui unitat i continuïtat (Merriam-Webster, 2021).

Aquesta definició ens porta a una pregunta: podria ser un ordinador capaç d'ordenar sons i relacionar-los entre ells creant una composició?

De la mateixa manera com abans hem vist programes informàtics capaços d'escriure textos, o de reconèixer imatges, existeixen algoritmes d'Intel·ligència Artificial amb l'objectiu de generar música. Aquests acostumen a ser xarxes neuronals que són entrenades per composicions ja existents, i observant els paterns que defineixen són capaces de generar nova música.

Història de la composició automatitzada

L'objectiu de generar música de forma automatitzada, i en definitiva, de buscar patrons en la música, apareix anys abans de la invenció de les xarxes neuronals.

Pitàgores ja relacionava la música amb els nombres, i creia en una relació entre les harmonies de la música i el cosmos. Ptolemeu relacionava la música amb l'astronomia, ja que creia que les matemàtiques eren la base tant dels intervals musicals com dels cossos celestes. Aquí neix la idea que les lleis matemàtiques s'apliquen a la música (Maurer, 1999).

En certa forma, la teoria musical es pot entendre com uns algoritmes. Per exemple, en la creació d'una peça, la primera veu és ideada pel compositor, però molts cops les veus d'acompanyament es creen seguint una sèrie de regles. A partir d'aquí, l'any 1957 un matemàtic anomenat Leonard Isaacson i un compositor de nom Lejaren Hiller van crear la primera composició musical creada per ordinador: la Illiac Suite For String Quartet (Hiller & Isaacson, 1959).

Per crear-la van generar nombres enters aleatòriament que es corresponien a diferents elements musicals (ritmes, notes, o dinàmiques). Aquests eren acceptats o rebutjats a través de regles de selecció basades en teoria musical.

L'any 1960, Zaripov va publicar el primer article on parlava sobre la composició de música per ordinador basant-se en algoritmes (Zaripov, 1960). L'any 1963, Iannis Xenakis va publicar un llibre sobre el seu programa, que "ajudava" a crear composicions basades en

matemàtiques a partir de dades estadístiques, probabilitats i atzar, anomenat Formalized Music (Xenakis, 1992).

El primer ús de xarxes neuronals en la generació de música no va tenir lloc fins anys més tard, quan el 1988, Lewis va publicar un article on presentava un perceptró de múltiples capes (una variant molt senzilla d'ANN) que creava una cançó utilitzant Gradient Descent (Lewis, 1988). Gairebé simultàniament, Peter Todd va publicar un altre article on utilitzava una RNN per crear música, que va ser publicat un any després (Todd, 1989).

L'any 1996, David Cope va publicar un llibre titulat Experiments in Musical Intelligence (Cope, 1996). El seu programa funcionava amb 3 passos: primer descomponia la música en parts, llavors determinava els paterns que definien l'estil de la cançó i finalment recombinava diverses peces en nous paterns. Aquesta intel·ligència artificial es va posar a prova quan enfront d'una audiència, una pianista va tocar tres peces (una original de Bach, una creada pel compositor Steve Larson i una creada pel programa EMI). Els oients no van ser capaços de distingir les cançons, ja que el seu veredicte va ser que la imitació creada pel compositor era feta per ordinador, i que la que havia creat el programa era la cançó original de Bach (Johnson, 1997).

El següent avenç en la creació de música utilitzant xarxes neuronals es va produir quan es van començar a implementar un tipus especial de RNN: les LSTM (Long Short Term Memory) Networks. Les xarxes recurrents convencionals són capaces de "recordar" valors a curt termini, mentre que les LSTM poden aprendre dependències més llargues. Aquest tipus de RNN va ser creat l'any 1997 (Hochreiter & Schmidhuber, 1997), i es segueix utilitzant actualment en traduccions, síntesi de veu i reconeixament de veu . 5 anys després, Eck i Schmidhuber van publicar un article sobre la utilització d'una d'aquestes LSTM per la creació de música inspirada en el blues (Eck & Schmidhuber, 2002). L'objectiu de l'ús d'una d'aquestes xarxes era que la cançó final tingués una certa estructura.

El mateix any, Marolt va publicar un article on utilitzava espectrogrames (en lloc de representacions simbòliques de la música, com s'havia fet fins aquell moment) per detectar l'inici de les notes (Marolt et al., 2002). A partir d'aquest punt, van començar a aparèixer més xarxes neuronals que utilitzaven aquest altre tipus de dades per crear les cançons.

En els següents anys, les xarxes neuronals es van aplicar en la música per classificar-la en gèneres o per detectar acords. Pel que fa a la generació de música, actualment s'utilitzen

models més nous, com GAN (Generative Adversarial Network), GRU (Gated Recurrent Units) o VAE (Variational Auto Encoders) (Pons, 2018).

Projectes actuals

Existeixen molts projectes actuals de diferents empreses amb xarxes neuronals aplicades a la música. Es poden diferenciar vàries categories, com la classificació de cançons per gèneres o algoritmes de recomanació com el de Spotify, però la que ens interessa és la de composició de música. A continuació, us mostrem 3 exemples de projectes enfocats en aquest àmbit.

Magenta

Magenta, que forma part de l'equip de Google Brain, té com a objectiu, més que crear nova música, proporcionar eines als compositors perquè puguin crear les seves cançons. Cal dir que tenen vàries xarxes neuronals que serveixen propòsits diferents, i que és un projecte open source, o sigui que el codi està disponible a tothom. Alguns dels models que tenen són:

- Drums RNN (crea una línia de bateria)
- Melody RNN (crea una melodia utilitzant LSTM)
- Performance RNN (imita una actuació real amb expressivitat)
- Coconet (utilitzant una CNN completa cançons incompletes
- Improv RNN (similar a Melody RNN, però segueix una successió d'acords)
- Music Transformer (similar a Performance RNN, però genera automàticament la cançó).(Magenta, 2016).

Cal destacar que molts d'aquests programes són capaços de generar no només melodies, sinó també acompanyaments, continuacions, ritmes nous… A més, alguns d'aquests models estan també disponibles com a plugins del programa Ableton live.

Musenet

Musenet és una xarxa neuronal de Deep Learning basada en el model de NLP GPT-2 (antecessor de GPT-3). Funciona amb 10 instruments diferents, i és fins i tot capaç de generar músiques barrejant diferents gèneres/compositors. Aquesta NN, creada per l'equip d'Open Ai, té un total de 72 capes i utilitza un tipus de model anomenat Transformer. Aquest

tipus de model està basat en el concepte de l’atenció, i quan rep dades no les processa necessàriament en ordre, sinò que dona més atenció a unes que d’altres basant-se en el context (Payne, 2021).

En la seva pàgina web mostren també les relacions que va trobar aquesta xarxa neuronal entre els diferents estils de música i compositors, i es poden provar diferents combinacions d’estils.

AIVA

AIVA és un programa especialitzat en la creació de música clàssica i simfònica. És interessant perquè l’any 2017 es va convertir en la primera intel·ligència artificial a rebre la condició de compositor, i per tant, té drets d’autor propis. Utilitza un algoritme de Deep Learning i Reinforcment Learning, tot i que en alguns casos també utilitza Genetic Algorithms. Tot i això, utilitza humans per enregistrar les composicions i utilitzen software d’edició de música per millorar els seus resultats (*About AIVA*, 2016; Borgos, 2021).

AI song contest

L’AI song contest és una competició internacional que es va celebrar per primer cop l’any 2020, on diferents equips presenten cançons creades amb l’ajuda d’intel·ligència artificial. Està molt inspirat en el festival d’Eurovisió, i de fet, té un sistema de puntuació molt similar.

El jurat no només avalua la qualitat de la cançó, sinó també la de la IA utilitzada. Als participants se’ls permetia combinar melodies generades automàticament amb intervencions humanes, creant així cançons molt similars a les que podria crear un grup de música humà (*FAQs*, 2021).

En general, aquest concurs és una bona forma de veure els límits de la intel·ligència artificial en la música, el procés de cocreació amb humans i quins resultats es poden obtenir. És interessant també veure els processos que els diferents equips van dur a terme, utilitzant IA en processos com la generació de la lletra, de l’acompanyament o directament de la melodia.

Part pràctica

Disseny del producte

Per tal de dissenyar i crear el meu projecte per tal d’assolir els objectius proposats he seguit una sèrie de passos que es poden resumir en els següents:

- Anàlisis
- Definició de l’estructura bàsica
- Programari i altres recursos
- Elaboració del codi
- Explicació del codi
- Valoració del producte
- Refinament del producte
- Validació d’experts

A continuació, aprofundiré més en cadascun dels apartats, explicant quina feina vaig haver de fer, el funcionament del meu codi, els problemes que he anat trobant i com els he afrontat.

Anàlisis

El primer que vaig fer per poder assolir els meus objectius va ser una àmplia recerca d’informació en diverses pàgines web, vídeos de YouTube i llibres, com Neural Networks from Scratch (Kinsley & Kukieła, 2020). Part dels coneixements obtinguts els vaig destinar a la creació del marc teòric vist anteriorment, tot i que en general van servir-me per poder determinar les característiques del producte que volia generar i conèixer com es podia programar una intel·ligència artificial.

En un inici no tenia ni tan sols clar quin tipus d’intel·ligència artificial volia utilitzar per a poder generar música, però una breu recerca inicial em va mostrar que la gran majoria de problemes similars, com la predicció de temperatures, es resolien utilitzant xarxes neuronals. A partir d’aquest punt vaig buscar exemples de xarxes neuronals que estiguessin programades des de zero, ja que eren els models que més s’assemblaven al que volia fer jo.

El llibre Neural Networks from Scratch (Kinsley & Kukieła, 2020) em va ser de gran ajuda, ja que explicava les diferents parts de les xarxes neuronals de manera senzilla i clara, i estava acompanyat per animacions que es poden observar en la seva pàgina web. Addicionalment, diversos tutorials de com crear xarxes neuronals, articles en pàgines webs i cerques en diferents fòrums també m’han servit com a font d’informació.

En aquesta fase del projecte, el problema principal que vaig tenir va ser trobar xarxes neuronals que estiguessin realment creades des de zero, ja que la majoria d’exemples utilitzaven llibreries com Keras, que faciliten el procés de creació de les NN.

Estructura bàsica del producte

Un cop feta la recerca d’informació, era hora de començar el procés de creació del codi. Per determinar l’estructura general de la meva xarxa neuronal vaig utilitzar una metodologia proposada en el llibre Deep Learning Techniques for Music Generation – A Survey (Briot et al., 2017), que es basa en 5 apartats.

El primer de tots és l’objectiu de la xarxa neuronal, que és en definitiva la resposta a dues preguntes: quin és el contingut musical que es vol generar, i la finalitat d’aquesta música. Abans de fer l’anàlisi d’informació, el meu objectiu era generar una cançó polifònica que representés l’ordinador mateix. Però més endavant em vaig adonar que era un objectiu massa ambiciós per a mi a causa de la complexitat que suposa generar melodies polifòniques, i vaig acabar determinant que l’objectiu de la meva xarxa neuronal fos crear una melodia que jo posteriorment pogués tocar amb el violoncel.

El segon apartat és el de representació, que bàsicament es refereix a la forma de representar la música que s’utilitza. En el marc teòric hem vist que existeixen diverses representacions possibles, però finalment em vaig acabar decantant cap als fitxers MIDI. Els meus motius van ser els següents:

- La gran quantitat de bases de dades disponibles
- L’existència de llibreries que em permetien modificar els fitxers directament des del codi
- La facilitat per transformar aquests arxius a números que la xarxa neuronal pugui agafar d’inputs

Aquesta representació, per tal que es pugui interpretar directament per la NN, s'ha de codificar. En el meu cas, transformo els fitxers MIDI a llistes de valors enters. A més, per no complicar massa les coses, simplement utilitzaré les notes, sense tenir en compte els ritmes, les dinàmiques…

El tercer punt és definir quin tipus de xarxa neuronal s'utilitzarà. En el meu cas, he utilitzat una xarxa neuronal directa (en anglès Feedforward Neural Network) que és simplement un tipus d'ANN que no té cap mena de retroalimentació. És la forma més pura d'ANN, i la més senzilla de programar.

Els desafiaments i els límits als quals s'haurà d'afrontar la xarxa neuronal formen part del quart punt. Aquests estan molt lligats amb els objectius, i es poden definir com les qualitats que busquem en la generació de la música. El meu objectiu en aquest sentit era que la xarxa neuronal creés una cançó original, és a dir, que tingués creativitat.

Finalment, l'últim punt és el de l'estratègia, que respon a la pregunta "Com modelem i controlem el procés de generació?". En el meu cas, he utilitzat un mètode anomenat Iterative Feedforward que funciona de la següent manera:

- Un inici es passa per la xarxa neuronal i es prediu la següent nota
- S'utilitza la predicció per crear uns nous valors
- Es passen els nous valors i es va repetint el procés

El principal avantatge que té utilitzar aquest mètode és que es pot escollir la llargada de la música generada, mentre que en la majoria d'alternatives aquesta ve determinada per la xarxa neuronal directament.

Programari i altres recursos utilitzats

Per tal de dur a terme el meu projecte he utilitzat diversos programes i recursos diferents, que em permetien fer diferents tasques. Però l'eina més bàsica que he utilitzat per crear el meu programa és el llenguatge de programació.

En el meu cas he utilitzat el llenguatge de programació Python, ja que és el més utilitzat en la creació de IA i és senzill i clar d'entendre. A més, ja disposava d'experiència prèvia utilitzant aquest llenguatge, fet que ha estat molt útil pel meu projecte.

Per tal de facilitar el treball amb el codi, molts cops s'utilitza un IDE (Integrated Development Environment), una aplicació que consisteix bàsicament en un editor de codi, un compilador, eines d'automatització com l'autocompletat, eines de depuració del codi i ressaltador de sintaxi. En el meu cas, per elaborar la primera part del codi vaig utilitzar un programa anomenat PyCharm en la seva versió Community (que té funcions limitades, però és gratuïta).

Però per la segona part del meu codi necessitava un altre tipus de programa, un que em permetés treballar amb més potència de càlcul i compartir el codi de forma fàcil. Per això, vaig passar d'utilitzar PyCharm a utilitzar Google Colab. Aquesta aplicació de Google permet escriure i executar codi de Python al navegador. Està pensada per a estudiants, científics de dades i investigadors en el camp de la IA. Permet l'accés remot a ordinadors de google, compartir les notebooks (els fitxers que emmagatzemen el codi), veure l'historial de versions i combinar text amb codi, entre altres funcions. A més, conté llibreries ja instal·lades que permeten treballar amb IA (tot i que moltes no les he fet servir perquè volia crear el meu codi des de 0) (*Google Colaboratory*, n.d.).

Un cop definits els entorns de treball, em tocava començar a programar. Però per fer algunes tasques determinades, les comandes bàsiques de Python no em servien i si les hagués programat jo haguessin augmentat molt més les hores invertides en el treball i no les podria realitzar de forma eficient. És per això que he utilitzat algunes llibreries (extensions del llenguatge de programació), com la mencionada en el marc teòric MIDO.

A part de la llibreria MIDO, una de les llibreries que he utilitzat és la llibreria numpy. Aquesta, que molts cops ja ve instal·lada directament, afegeix comandes que permeten operacions matemàtiques més complexes, nous tipus de variables i generadors de nombres aleatoris, entre d'altres. El seu ús ofereix diversos avantatges, com una millor llegibilitat del codi, una major velocitat d'execució… En el meu cas, l'he utilitzat per a crear arrays (matrius) i per a realitzar operacions amb aquestes matrius sobretot, ja que aquesta és la forma en la qual s'estructuren les dades que es transmeten per la xarxa neuronal.

Una altra llibreria utilitzada en aquest treball és la llibreria matplotlib. Aquesta ja venia preinstal·lada en el Google Colab, i la seva funció principal és visualitzar dades. Permet crear gràfics de barres, histogrames, gràfiques, trames tridimensionals… Personalment, l'he fet servir per a representar els fitxers MIDI en una gràfica nota-temps, ja que feia molt més visual la representació tant de dades com de la música generada.

Addicionalment he utilitzat altres llibreries, com la llibreria Os, que permet accedir a les carpetes i als fitxers de l’ordinador, o la llibreria google.colab que permet interactuar amb l’entorn de google colab (en el meu cas, descarregar fitxers automàticament).

Un cop vist l’apartat del programari a utilitzar, em faltava trobar una font de fitxers MIDI.

Inicialment vaig simplement cercar bases de dades de MIDI en diferents webs. Les primeres que vaig trobar eren de música clàssica, i tot i que en un inici van ser les que utilitzava, em generava problemes perquè els arxius estaven confeccionats de diferents maneres.

Com que no m’acabava de convèncer, vaig tornar a buscar altres bases dades, i vaig acabar arribant a una anomenada Lakh MIDI dataset, que conté milers de fitxers. Aquesta base de dades et permet descarregar diverses carpetes, algunes de MIDI emparellats amb àudios, i d’altres simplement amb MIDI. En el meu cas, no necessitava els àudios, o sigui que em vaig descarregar la versió de MIDI. Un cop baixada, vaig començar a provar-la en el meu programa.

Vaig estar uns quants dies utilitzant aquesta base de dades, i no em va generar problemes mentre creava la primera part del codi. Però quan vaig acabar de programar la primera xarxa neuronal inicial, que encara era molt rudimentària, em vaig adonar que no em serviria per al meu propòsit. Les cançons de la base de dades, contenien tots els diferents instruments de les cançons, i jo simplement volia generar una melodia (utilitzant un sol instrument), ja que generar una melodia polifònica és molt més complicat. És per això que em vaig posar a buscar una tercera base de dades que simplement tingués fitxers MIDI de piano (un sol instrument).

Aquesta base de dades la vaig trobar en un projecte de Github que utilitzava una LSTM per crear composicions de piano. Tot i que el nombre de cançons era molt reduït en comparació a les que havia estat utilitzant, s’adaptava perfectament al projecte que estava creant i a la melodia que volia crear.

Fases de l’elaboració del codi i evolució d’aquest

En aquest apartat, explicaré com va ser el procés de creació del codi, i com va anar evolucionant aquest.

El primer que vaig fer van ser tres funcions orientades al processament de dades: una que transformava els fitxers MIDI a arrays, una que transformava els arrays a fitxers MIDI i una que creava gràfiques. En el següent apartat explicaré com funcionen aquestes funcions, i en definitiva, tot el que he anat creant.

Aquesta primera fase de l’elaboració del codi la vaig dur a terme utilitzant el programa Pycharm i agafant de referències diversos fitxers MIDI de la base de dades de música clàssica. Vaig estar uns quants dies intentant perfeccionar el codi perquè funcionés amb els diferents tipus de missatge, afegint condicionals cada cop que em trobava amb missatges MIDI nous. Va tenir una duració d’uns 5 mesos, però no tant per la complexitat del codi sinó per les poques hores que hi invertia, ja que encara estava estudiant i no tenia massa temps.

La següent fase va ser l’inici de la xarxa neuronal, que va constar sobretot en observar codis de diferents llocs web, provar-los i analitzar com funcionaven. A partir d’aquí, vaig anar creant el meu codi, basant-me en exemples que trobava a internet i el que anava aprenent.

Vaig arribar a un punt on funcionava, però volia intentar crear una xarxa una mica més potent. Mentre creava la primera versió, vaig evitar implementar-hi la retropropagació, ja que m’era complicat entendre com funcionava i com implementar-la. Però un cop vaig tenir una xarxa que ja era capaç d’entrenar, vaig pensar que seria una bona millora, i per tant, vaig crear una còpia del document i em vaig posar a implementar-la.

Va ser un procés una mica complicat, ja que no trobava les derivades que necessitava per a realitzar el Gradient Descent, i posteriorment no sabia del tot com aplicar-les. Però tot i això, vaig aconseguir fer-ho, creant així la segona versió del meu codi (Annex 2).

Finalment, després de valorar el projecte vaig acabar creant una tercera versió del codi. Els fets que em van portar a crear-la, i les diferències a respecte les altres versions les explicaré en un altre apartat més endavant. De moment, em limitaré a explicar el funcionament del codi de les dues primeres versions, ja que és molt similar.

Funcionament del codi

En l’apartat següent explicaré una mica com funciona el codi que he creat, però penso que abans és important explicar unes nocions bàsiques de programació per facilitar l’entesa de les següents pàgines.

Per fer aquesta explicació em basaré en el llenguatge Python, ja que és el que he utilitzat. Aquest va ser creat l’any 1991 per Guido van Rossum, i destaca sobretot per la seva senzillesa i llegibilitat (GeeksforGeeks, 2019).

Començarem per entendre que un statement (declaració) és una instrucció que l’interpretador pot executar. A partir d’aquella, l’ordinador interpreta el que ha de fer. Un exemple d’una declaració bàsica és el següent:

```
print(“Hello world”)
```

Aquesta simple declaració el que fa és mostrar per la pantalla (dit d’altra forma, imprimir) el text que li hem posat, en aquest cas, “Hello world”. No ens endinsarem molt en la sintaxi en si del llenguatge, ja que tampoc es tracta de fer un tutorial, sinó simplement explicar el bàsic per poder entendre almenys una part del codi. D’aquests statements n’hi ha de molt tipus diferents que realitzen accions diferents. La part positiva de Python és que amb unes nocions bàsiques d’anglès es poden entendre perfectament. A continuació veurem els statements més bàsics que s’utilitzen en pràcticament tots els codis.

Una de les característiques més útils dels llenguatges de programació són les variables. Aquestes es poden entendre com una espècie de contenidors, que emmagatzemen diferents tipus de dades. Cada contenidor ha de tenir un nom diferent, ja que si no es solaparan. Un dels avantatges que té Python és que és molt fàcil declarar aquestes variables, ja que automàticament interpreta de quin tipus són. Un exemple d’una declaració d’una variable és:

```
x = 0
```

Amb aquesta simple línia li estem dient al programa que creï una variable de nom x i li assigni el valor numèric 0. A partir d’aquí, aquesta variable pot patir canvis. Per exemple:

```
x = 0
x = x + 2
x = x - 1
```

Ara, la variable x ha passat de valer 0, a valer 2 i finalment, 1. Per fer-ho, hem utilitzat un operador, el +. Els altres operadors que trobem en Python són - (resta), / (divisió), * (multiplicació), ** (potència), % (mòdul), // (divisió entera).
Les variables poden ser de molts tipus, com per exemple str (cadenes de text), bool (booleans, emmagatzemen True o False), float (números amb decimals), int (nombres enters)...

Un cop més o menys explicades les variables, toca parlar de la que és la part més important de la programació, almenys des del meu punt de vista: els condicionals i els bucles.

El condicional principal és l’if. Aquest en si, més que un condicional, és una declaració que retorna cert si els arguments que se li donen són certs. Aquests arguments són els que contenen els condicionals. Els principals són:

- == (retorna cert si és exactament igual) -
- >= (retorna cert si el número és més gran o igual)
- <= (retorna cert si el número és més petit o igual)
- > (retorna cert si el número és més gran)
- < (retorna cert si el número és més petit)
- != (retorna cert si els dos valors són diferents)

Aquests condicionals, alhora, poden estar en combinació amb operadors lògics com l’and (si els dos valors son certs) i l’or (si un dels valors és cert).

Altres declaracions són l’Elif (si el primer if no es compleix, però és necessari un altre condicional), i l’Else (si cap dels if o elif que hi ha es compleixen).

A part dels condicionals, també trobem els bucles. Aquests es poden entendre com declaracions que tenen lloc mentre una sèrie de requisits es compleixin. N’hi ha dos tipus principals.

El while executa el codi que té dins mentre es compleixi una funció. Quan arriba al final, torna a comprovar si es segueix complint. Es pot entendre com un if que es va repetint diverses vegades.

D’altra banda, el for serveix per iterar pels diversos elements d’una cadena, llista… És per això que no utilitza realment una condició que es va complint. Si per algun motiu es

necessita un for que es repeteixi un nombre determinat de vegades, es pot inicialitzar un int i escriure el següent, utilitzant la funció range():

```
for(i in range(nombrederepeticions)):
```

Una seqüència de statements amb una única sortida és una funció. D’aquestes n’hi ha de dos tipus: les que venen predefinides per Python (funció int() per exemple) i les funcions que declarem nosaltres en el nostre codi. Aquestes les hem d’inicialitzar primer, utilitzant la següent estructura:

```
def nomdelafunció (paràmetres):
    statements
```

Un cop inicialitzades, escrivint el nom de la funció amb els paràmetres que necessita executarà automàticament tots els statements.

Finalment, cal parlar sobre els objectes i les classes. Els objectes són la base del llenguatge Python en si, i es poden definir com un grup conformat per dades (variables) i funcions que actuen sobre aquestes dades. Pel que fa a les classes, es poden definir com els plànols en els quals es basen els objectes. D’una sola classe, es poden definir diversos objectes. De la mateixa manera que amb les funcions, existeixen classes ja definides per Python, i d’altres s’han d’inicialitzar. En aquest cas, l’estructura és:

```
class nomdelaclasse:
    funcions de la classe
```

I per crear l’objecte:

```
nomdelobjecte = nomdelaclasse()
```

En el codi, a part de tots els statements que he mencionat anteriorment, també trobarem textos amb un # a l’inici de la línia. Aquests són comentaris, i a l’hora de compilar el programa no es tenen en compte. Són útils per introduir explicacions dintre del mateix codi.
En alguns casos es necessiten comentaris en múltiples línies. Per a realitzar-los, es crea una cadena (String) multilineal sense assignar-la a cap variable de la següent manera:

```
"""
Comentari
escrit
en
diferents
línies
"""
```

Un cop vista la forma general en la qual funciona el llenguatge de programació que he utilitzat, és el torn d'explicar com funciona el codi del meu projecte i com l'he creat.

Per simplificar l'explicació, es pot separar el codi que he creat en dues parts: la part de preparació de dades, i la xarxa neuronal en si (que corresponen amb les fases principals de la creació del codi).

La primera part, com he dit abans, la vaig escriure inicialment en el programa PyCharm. L'objectiu d'aquesta part del codi és transformar de diverses maneres els fitxers MIDI i els arrays amb els que treballo. En definitiva, aquest procés vindria a ser una espècie de preprocessament de dades, que a grans trets consisteix a transformar dades en brut a dades ben formatades, tot i que també he afegit algunes funcions per visualitzar aquestes dades. Per realitzar aquesta part del codi, l'única informació que vaig consultar va ser com s'estructuraven els fitxers MIDI en la llibreria MIDO, ja que amb els coneixements que tenia de programació ja era capaç d'assolir l'objectiu d'aquesta part.

La primera funció d'aquesta part és l'encarregada de transformar els fitxers MIDI a arrays de números. Rep com a paràmetres un fitxer MIDI. El que fa és inicialitzar una llista buida i el fitxer MIDI utilitzant la llibreria MIDO, i iterar per cadascun dels seus missatges. Si el missatge conté metadades o controls de programa, els afegeix a la llista directament, sense modificar-los. Però si es tracta d'un missatge de note_on o note_off, que són els que ens interessen, afegeix els valors numèrics dels seus paràmetres. Per exemple:

Message('note_on', note = 64, velocity = 50, time = 20)

es transforma en:

[1, 64, 50, 20]

Una visió simplificada del codi (una espècie de pseudocodi) que conforma aquesta funció seria la següent:

```
def miditoarray(midi):
        llista = [] #inicialitza una llista buida
        midi = MIDO.midi
        for missatge in midi: #itera per tots els missatges del fitxer
                if missatge = note_on:
variable = separa les parts del missatge on hi ha '=' per obtenir els nombres
                        afegeix [1, nombres] a llista
                elif missatge = note_off:
variable = separa les parts del missatge on hi ha '=' per obtenir els nombres
                        afegeix [0, nombres] a llista
                else:
                        afegeix missatge a llista
        return llista
```

La sortida d'aquesta funció és una llista de llistes. Un exemple d'aquesta sortida és el següent:

[['program_change', 'channel=0', 'program=0', 'time=0'], [0, 0, 66, 0, 512], [1, 0, 72, 88, 0], [1, 0, 53, 88, 0], ... [0, 0, 53, 0, 512], [1, 0, 53, 88, 0], [0, 0, 53, 0, 512], [0, 0, 72, 0, 0]]

La següent funció és molt similar, però funciona a la iinversa. És l'encarregada de transformar els arrays de números a fitxers MIDI, que es puguin escoltar posteriorment. En el meu cas, era necessària per a poder guardar i sentir la música creada per la IA.

Bàsicament treballa amb els mateixos conceptes que la funció anterior, i utilitza els recursos de la llibreria MIDO per crear el fitxer MIDI. A més, també afegeix un ritme aleatori als arrays que no tenen durada predefinida de les notes (com el que genera la nostra IA, ja que no es centra en el ritme).

La versió en pseudocodi d'aquesta funció, que he anomenat arraytomidi, és la següent:

```
def arraytomidi(array):
        outfile = inicialitza un fitxer midi buit
        for element in array:
                if primer element = 1:
                        afegeix Message(“note_on”, resta de paràmetres) a outfile
                elif primer element = 0:
                        afegeix Message(“note_off”, resta de paràmetres) a outfile
                elif primer element = “program_change”:
                        afegeix Message(element separat per parts) a outfile
                                        ...
                (repeteix el mateix pels diferents tipus d’elements)
        guarda outfile
```

La tercera de les funcions d’aquesta part ja no està tan enfocada en el format de les dades, sinó en la seva representació. Es tracta de la funció graficador, que rep com a paràmetre un array. Com el seu nom indica, és la funció utilitzada per crear un gràfic.

El seu funcionament és bastant senzill. Va itinerant per tots els elements de l’array, i si detecta que és una “note_on”, afegeix el valor de la nota en una llista, i el temps d’aquesta nota en una altra. Posteriorment, utilitza aquestes dues llistes per crear un gràfic nota-temps utilitzant la funció plot de la llibreria matplotlib. Finalment, retorna la llista que conté les notes.

Cal dir que si observem el codi, veurem que hi ha moltes variables que no s’utilitzen. Això es deu al fet que en un inici, aquesta funció retornava un altre array similar al que rebia però amb algunes modificacions. Més endavant em vaig adonar que la majoria dels valors no em servien , i que només necessitaria les notes, i és per això que vaig canviar el valor que retorna la funció.

Una versió simplificada d’aquest codi seria la següent:

```
def graficador(array):
        notes = [] #inicialitza una llista buida
        temps = []
        for llista in array:
                if llista[0] és igual a 1: #[0] és el primer element de la llista
                        afegeix llista[nota] a notes
```

```
                    afegeix llista[temps] a temps

#en el codi existeixen alguns elifs, però  com hem dit abans no ens interessen)
        matplotlib.plot(temps, notes)
        mostra plot
        return notes
```

En una part del codi més endavant, m'era útil visualitzar dues gràfiques en la mateixa imatge. És per això que vaig crear una altra funció, molt similar a la funció graficador, de nom grafcomparacio. El seu funcionament és molt similar: bàsicament rep dos arrays en lloc d'un, i crea dues llistes de notes i dues de temps. La funció plot permet automàticament mostrar dues gràfiques en la mateixa imatge, i utilitzant diferents colors per poder-les diferenciar.

El pseudocodi, molt similar al del graficador, seria:

```
def grafcomparacio(array1, array2):
        notes1 = [] #inicialitza una llista buida
        notes2 = []
        temps1 = []
        temps2 = []
        for llista in array1:
                if llista[0] és igual a 1: #[0] és el primer element de la llista
                        afegeix llista[nota] a notes1
                        afegeix llista[temps] a temps1

#en el codi existeixen alguns elifs, però  com hem dit abans no ens interessen)
        for llista in array2:
                if llista[0] és igual a 1: #[0] és el primer element de la llista
                        afegeix llista[nota] a notes2
                        afegeix llista[temps] a temps2


        matplotlib.plot(temps1, notes1)
matplotlib.plot(temps2, notes2)
        mostra plot
```

Finalment, ens queda una última funció d'aquesta part, i és la funció dades. Aquesta representa l'última transformació de les dades abans d'arribar a la xarxa neuronal. Rep com a paràmetre una llista de notes, i retorna dos arrays: un amb els inputs de la xarxa neuronal, i un altre amb els outputs corresponents a aquests inputs. El codi vist de forma simplificada seria el següent:

```
def dades(notes):
    arrayinputs = []
    prediccions = []
    iteració = 0
    for element in notes: #en el codi realment he utilitzat un while però funciona igual
        inputs = []
        variable = 0
        while (variable més petita que el número d'inputs):
            afegeix  notes[iteració + variable] a inputs
            variable += 1
        afegeix inputs a arrayinputs
        afegeix  notes[iteració + número d'inputs] a prediccions
        iteració += 1
    return inputstotals, prediccions
```

Un exemple del que retorna aquesta funció és el següent:

Arrayinputs:

[[76, 76, 72, 72], [76, 72, 72, 69], … … [69, 72, 81, 69], [72, 81, 69, 72], [81, 69, 72, 81], [69, 72, 81, 76], [72, 81, 76, 79], [81, 76, 79, 88]]

Prediccions:

[69, 69, … … 72, 81, 76, 79, 88, 76]

La segona part del codi és la que conforma la xarxa neuronal. Cal destacar que el funcionament d'aquesta és molt similar a l'explicat durant el marc teòric.

La part principal d'aquesta la trobem en la classe capa. Aquesta, com el seu nom indica, és l'encarregada d'inicialitzar les capes de la xarxa neuronal. Consta de dues funcions: la funció anomenada __init__ i una altra anomenada forward.

La primera d'aquestes funcions és el constructor de la classe. S'executa quan es crea un objecte d'aquesta classe, i serveix per inicialitzar els seus atributs. En aquest cas, genera aleatòriament dues variables: una consistent en un array de pesos de dimensions [nombre d'inputs, nombre de neurones] i una altra de biases, amb dimensions [1, nombre de neurones].

Els pesos són els valors assignats a cada connexió entre neurones. És per això que existeix un valor per cada input que rep la neurona i per cada neurona que existeix. Per exemple, si la nostra xarxa neuronal tingués 3 inputs i 4 neurones, els pesos de la primera capa serien similars als següents:

[[23.0, 53.2, 21.5, 76.4],
[32.5, 65.1, 11.4, 31.6],
[95.2, 12.5, 55.6, 91.5]]

L'altra funció és l'encarregada de multiplicar els inputs pels pesos i sumar els biases. Bàsicament utilitza la funció de numpy.dot per multiplicar arrays entre si. S'anomena forward, ja que és la que s'utilitza en la propagació cap endavant de les dades. En definitiva, és l'encarregada de crear les dades que passaran per la funció d'activació.

La següent classe que trobem és la ReLu. Aquesta és la funció d'activació que utilitza la nostra xarxa neuronal. El seu nom prové de Rectified Linear, i va començar a aplicar-se a les xarxes neuronals entorn l'any 2010. Aquesta és molt útil, ja que sembla i actua de manera molt similar a una funció lineal sense ser-ho, fet que facilita molt els processos com la retropropagació per GD. A més, un dels avantatges que té és que no activa totes les neurones alhora, fent-la més eficient computacionalment.

La seva funció expressada matemàticament és:

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

Bàsicament, si un número és més petit que 0 el valor resultant d'aquesta funció serà 0. Si en canvi és més gran, serà el mateix valor que se li ha passat. Modificant els pesos indirectament es canvia el pendent d'aquesta funció, i modificant els biases es canvia el punt on comença el pendent. Jugant amb aquests valors, la xarxa neuronal pot adaptar aquesta funció a les dades que se li donen.

Aquesta és la funció forward de la classe. La funció backward consisteix en la seva derivada, ja que és el que s’utilitza en el gradient descent. Aquesta derivada es pot expressar com a:

$$f'(x) = 0 \; if \; x < 0$$

$$f'(x) = 1 \; if \; x > 0$$

Per realitzar aquesta funció, el codi utilitza una eina de numpy que permet accedir als elements d’un array mitjançant un condicional. Per tant, busca tots els elements que siguin més petits que 0 i els canvia per 0, i els que siguin més grans els canvia per 1.

Les següents classes del codi corresponen a altres funcions d’activació. Tot i que en la versió final del codi utilitza la funció ReLu, també vaig programar-ne 3 més:

La funció Leaky ReLu, que és bàsicament la ReLu, però enlloc de 0 passa un valor molt petit (0.01) multiplicat per l’input. Serveix per evitar el problema de Dying ReLu, que provoca que quan les neurones es deixen d’activar, com que el seu gradient és 0 i els pesos no es canvien, no es tornin a activar cap altre cop.

La funció ELU, que no és lineal i modifica el pendent de la part negativa de la funció, permetent que es passin valors negatius. La seva fórmula és:

$$f(x) = x \; if \; x > 0$$
$$f(x) = \alpha(e^{x} - 1) \; if \; x <= 0$$

Aquesta introdueix un altre paràmetre, α, que normalment és un valor entre 0.1 i 0.3. Si es vol, es pot anar ajustant aquest valor, modificant el pendent de la part negativa de la funció. El problema principal d’aquesta funció d’activació és que tarda més a calcular-se i que és complicat que la xarxa aprengui el valor α.

La funció sigmoide, una de les més típiques en les xarxes neuronals, que transforma els valors entre 0 i 1. La seva funció és:

$$f(x) \ = \ 1/(1 \ + e^{-x})$$

A continuació d’aquestes funcions, trobem la funció softmax. Aquesta és la funció utilitzada per a la classificació multiclasse (la funció d’activació de l’última capa), i retorna un vector amb les probabilitats de cada classe. La seva fórmula és:

Figura 9

Mathematical definition of the softmax function

$$\sigma(\vec{z})_i \ = \ \frac{e^{z_i}}{\sum_{j=1}^{K} \ e^{z_j}}$$

Nota. De *Softmax function definition*, per T. Wood, 2020, (https://deepai.org/machine-learning-glossary-and-terms/softmax-layer)

Bàsicament, cada element del vector que retorna es calcula de la següent forma: s’eleva e al valor en aquella posició del vector d’entrada, i es divideix per la suma de e elevat a tots els valors. Addicionalment, destacar que en el codi primer es resta el valor més gran del vector a tots els elements, ja que evita possibles problemes en dividir per 0. En el nostre cas acaba generant una possibilitat per a cadascuna de les 127 classes (notes) possibles.

La següent classe només existeix en la versió del codi que no implementa la retropropagació (Annex 1), ja que en l’altra versió vaig simplificar la seva funció per facilitar la backpropagation. Es tracta de la classe Loss_CategoricalCrossentropy, que com el seu nom indica, és l’encarregada de calcular el Loss utilitzant la funció categorical cross-entropy, que és la més utilitzada per les xarxes en les quals s’utilitza Softmax a l’última capa. La seva expressió matemàticament és:

Figura 10

Categorical Cross-entropy fórmula

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output} \\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Nota. De *How to choose cross-entropy loss function in Keras?*, per Keras, 2021, (https://androidkt.com/choose-cross-entropy-loss-function-in-keras/)

En el codi, trobem aquesta funció simplificada, ja que no ens fa falta calcular-la pels valors que no són certs. En el codi, la funció seria més similar a -log(confiança_en_la_classe_correcte). Bàsicament, rep les probabilitats de cada classe i la classe correcta, i busca la probabilitat d'aquella classe en concret. Llavors aplica el logaritme (en base e) en aquest valor. El loss és el negatiu d'aquest logaritme. Addicionalment, com que treballa amb diferents prediccions a la vegada, el Loss acaba sent la mitjana entre tots els losses.

A partir d'aquest punt, totes les classes i funcions ja estan definides. Ara falta declarar-les i passar les dades per la xarxa neuronal, entrenar-la un nombre determinat de vegades i acabar utilitzant els millors pesos per generar una predicció.

Aquesta part del codi, que és l'última, comença aplicant les funcions que s'encarreguen de crear les dades.

El primer bucle que trobem és un for que itera per tots els fitxers de la carpeta on estan els fitxers MIDI. Aplica la funció miditoarray, i ajunta tots els arrays en un de sol. D'aquest array s'agafen els valors fins a un punt determinat per una variable anomenada notesgeneral. Aquesta variable la introdueix l'usuari, i permet determinar quantes notes es faran servir en el procés d'entrenament. Seguidament, es passa aquest array per la funció graficador (visualitzant les dades de la cançó) i per la funció dades, que acaba generant dos arrays (X, que són les dades d'entrada, i y, que són les classes correctes pels valors corresponents de X).

El que fa a continuació és crear 4 objectes: dense1, que és la hidden layer (amb dimensions nombre d'inputs, nombre de neurones), activation1, que és la funció d'activació d'aquesta capa (ReLu), dense2, que és la capa de sortida (amb dimensions nombre de neurones, 127) i activation2, que és l'activació de l'última capa (Softmax).

En la primera versió del codi, també es crea l'objecte loss_function, basat en la classe Loss_CategoricalCrossentropy.

Seguidament, es creen unes quantes variables. Una és el lowest_loss, que comença amb un valor molt i es va actualitzant quan entrena la xarxa. La resta, que també es van actualitzant, són els millors pesos possibles. En la versió B, també es crea una variable anomenada one_hot_y. Aquesta conté un array d'arrays basat en la variable y. Els valors d'aquests arrays són tots 0 menys en la posició de la classe correcta. Un exemple d'un d'aquests arrays seria:

[0, 0, 1, 0, 0, 0, 0, 0, … 0, 0, 0, 0, 0, 0]

En aquest cas, la classe correcta és la classe 2 (en programació la primera posició rep el valor 0).

A continuació trobem un bucle, que es va repetint un número de vegades determinat per la variable repsgeneral. Determina la quantitat de repeticions que fa la xarxa neuronal per entrenar.

La versió sense retropropagació comença actualitzant els pesos aleatòriament, sumant als pesos el learning rate multiplicat per un valor aleatori.

A continuació, els dos codis realitzen una passada cap endavant de les dades, utilitzant les funcions forward dels objectes. El funcionament és el següent:

Es passen les dades per la primera capa: dense1.forward(X)
Es passen les dades per la primera funció d'activació: activation1.forward(dense1.output)
Els resultats de l'activació es passen per la segona capa: dense2.forward(activation1.output)
Es passen aquests valors per la funció softmax: activation2.forward(dense2.output)

Un cop realitzades aquestes funcions, es calcula el loss. En la primera versió, es crida l'objecte loss_function, calculant el loss utilitzant l'output de l'activation2 i la variable y. A més, es creen dues altres variables: la de predictions (la posició on hi ha una probabilitat més alta) i la d'accuracy (la mitjana entre els valors totals i els valors que coincideixen amb la predicció). Un com en aquest punt, es mira si el loss és més petit que el loss més petit, i si ho és, s'actualitzen els millors pesos i el loss més petit.

En l’altra versió del codi, les coses funcionen una mica diferent. Per començar, es calcula el Loss utilitzant la funció matemàtica sense simplificar. També es creen les variables predictions i accuracy de la mateixa forma, i es mira si el loss és més petit i es guarden els pesos.

A continuació, comença el procés de Backpropagation. Aquest fa referència a propagar l’error cap endarrere (en direcció als pesos de la primera capa). He utilitzat el mètode de Gradient Descent, i per tant, el programa calcula les derivades de les funcions respecte als paràmetres de la xara utilitzant la regla de la cadena. A continuació veurem una mica per sobre com funciona, sense explicar tot el procés de derivació que té lloc fins a arribar a les derivades que necessitem.

La primera derivada que calcula és la de la funció del loss respecte als inputs de la funció softmax. Matemàticament es representaria de la següent forma:

$$\frac{dloss}{dsoftmax} \cdot \frac{dsoftmax}{dzo} = softmax.output \ - \ y$$

On zo són els inputs que rep la funció softmax (dense2.output) i y la variable one_hot_y. La següent derivada és la de la funció zo respecte als pesos (wo):

$$\frac{dzo}{dwo} = activation1.output$$

La unió d’aquestes derivades és la variable dloss_pes (derivada del loss respecte als pesos de capa de sortida), que consisteix en la multiplicació entre si de les dues. La dloss_bias (derivada del loss respecte als biases de la capa de sortida) és la mateixa que la primera derivada que hem calculat. Aquestes dues derivades fan referència als paràmetres de la capa de sortida de la xarxa, i ara toca veure les que fan referència a la capa intermèdia (hidden layer).

La primera d’aquestes és la derivada dels inputs de la funció softmax (zo) respecte a la sortida de la funció ReLu (l’activació de la hidden layer, ah). Aquesta correspon als pesos de la capa de sortida (wo).

$$\frac{dzo}{dah} = dense2.weights$$

A partir d'aquí i de la primera derivada de totes que hem vist, determinem que:

$$\frac{dloss}{dah} = \frac{dloss}{dzo} \cdot \frac{dzo}{dah}$$

Aquesta és la variable dloss_dah. A continuació trobem la derivada de la funció d'activació respecte als seus inputs (dah respecte dzh). Aquesta és la derivada de la funció ReLu que hem vist abans.

$$\frac{dah}{dzh} = 0 \; if \; zh \; < \; 0$$

$$\frac{dah}{dzh} = 1 \; if \; zh \; > \; 0$$

La derivada dels inputs d'aquesta funció respecte als pesos és simplement els inputs de la xarxa neuronal.

$$\frac{dzh}{dwh} = X$$

I respecte als biases és:

$$\frac{dzh}{dbh} = 1$$

A partir d'aquí, utilitzant la regla de la cadena, determinem que la derivada de la funció de Loss respecte als pesos de la hidden layer és:

$$\frac{dloss}{dwh} = \frac{dloss}{dah} \cdot \frac{dah}{dzh} \cdot \frac{dzh}{dwh}$$

I respecte als biases, podem deduir que és:

$$\frac{dloss}{dbh} = \frac{dloss}{dah} \cdot \frac{dah}{dzh} \cdot \frac{dzh}{dbh}$$

Un cop calculades totes les derivades, actualitza els pesos utilitzant la següent fórmula:

$$pesos \ = \ pesos \ - \ learningrate \ * \ derivada\ del\ loss\ respecte\ als\ pesos$$

Que en codi de Python és:

```
dense1.weights -= learningrategeneral * dloss_pesh
dense1.biases -= learningrategeneral * dloss_biash.sum(axis=0)
dense2.weights -= learningrategeneral * dloss_pes
dense2.biases -= learningrategeneral * dloss_bias.sum(axis=0)
(pesh = wh, biash = bh, pes = wo, bias = bo)
```

Aquests pesos actualitzats són els que s'utilitzen en la següent iteració. Un cop realitzades totes les iteracions, els dos codis (tant Annex 1 com Annex 2) realitzen un forward pass amb els millors pesos i biases. Les prediccions d'aquesta passada de dades s'utilitzen per visualitzar una gràfica comparant-la amb els valors de y, utilitzant la funció de grafcomparador. En aquest punt, el procés d'entrenament ha acabat, i toca passar a la generació de música.

El que es fa en aquesta part és el mateix en els dos codis. Per començar, es crea un array X amb l'inici de la cançó tradicional "Cada dia al dematí". A continuació, hi ha un for que es va repetint fins a assolir la llargada desitjada de la cançó generada.

El que fa el codi dins aquest bucle és molt simple: passa l'array X per la xarxa, prediu el següent valor, i afegeix aquest valor al final de X i al final d'un altre array, que és el que serveix per guardar la cançó. Per tal que segueixi tenint la mateixa llargada, també s'elimina el primer valor de l'array X. Això es va repetint tants cops com el for indiqui.

Si observem el codi, veiem que hi ha més condicionals i statements. Aquests simplement serveixen per acabar de formatar bé l'array per tal que al final, es pugui utilitzar la funció arraytomidi per crear un fitxer amb la música generada. A més, aquesta funció també és l'encarregada d'afegir el ritme aleatòriament a la cançó, ja que la xarxa només és capaç de predir notes.

L'array de la cançó generada també es visualitza amb la funció graficador, ja que així és més fàcil veure si ha generat alguna cosa interessant, o si hi ha hagut algun problema.

Valoració del producte

Un cop vaig tenir una xarxa neuronal creada, vaig dedicar-me a entrenar-la en diverses ocasions, modificant paràmetres com el nombre de neurones, el learning rate o el nombre de notes que la NN agafava d'input. Durant la creació d'aquesta ja havia estat realitzant execucions del codi i observant com es comportava.

Quan vaig començar a fer proves amb la xarxa neuronal, encara no havia creat l'última part del codi, que és la que generava la cançó final. El que feia bàsicament era entrenar la màquina amb una cançó, i mirar com intentava replicar-la. Un cop vaig veure que més o menys replicava la cançó correctament, amb un error no massa gran, va ser quan vaig començar a programar la part de la generació de la cançó.

Un cop la xarxa neuronal acabava d'entrenar i de generar la nova música, em dedicava a observar les diferents gràfiques i a escoltar el fitxer MIDI que generava. En aquest apartat em va ser molt útil la funció que havia creat abans de grafcomparador, ja que em permetia saber com era de similar la cançó que predeia mentre entrenava comparada amb les dades que li eren subministrades. Addicionalment, era capaç de veure com anava millorant el loss i l'encert, ja que quan detectava uns pesos millors mostrava per la pantalla aquests valors i el número de la iteració que era.

El fet de disposar de tots aquests valors em va fer veure que, tot i que la xarxa neuronal teòricament funcionava, i era capaç d'aprendre, encara tenia alguns problemes.

El problema més comú amb el que em vaig trobar, era que la xarxa neuronal es quedés encallada en el procés d'entrenament i/o que la seva predicció fos una sola nota. Després d'una cerca a internet, vaig trobar que aquest fet podia ser degut a diferents causes.

Els factors principals que causen aquests errors en les xarxes neuronals són algoritmes d'entrenament amb errors, dades del tipus incorrecte, problemes en la inicialització de les dades i l'optimització, learning rate equívoc, preprocessament diferent a la hora d'entrenar i

generar les prediccions, el problema de Dying ReLu, que les dades estiguin molt desbalancejades o que simplement les dades siguin massa complexes per la xarxa neuronal.

A partir dels resultats d’aquesta petita recerca, vaig decidir anar mirant un per un els possibles problemes per veure si eren els que causaven problemes en la meva xarxa. El primer pas que vaig fer va ser canviar les dades per unes més senzilles de classificar i una quantitat de neurones més baixa, i vaig executar el codi. Això em va servir per descartar el fet que la xarxa neuronal tingués problemes en els algoritmes que utilitzava per entrenar, ja que va ser capaç de classificar aquestes dades amb bastant precisió.

El segon possible problema també va quedar descartat, ja que simplement imprimint per pantalla els valors d’entrada ja em mostrava que estaven tots bé. El factor del learning rate tampoc vaig pensar que fos el causant del problema, ja que vaig realitzar repeticions amb LR diferents i el problema seguia persistint en la majoria dels casos. També vaig descartar el preprocessament com a problema, ja que utilitzava les mateixes funcions tant en l’entrenament com en la generació de noves cançons.

El problema de Dying ReLu tampoc era possible, ja que abans d’implementar la retropropagació, quan els pesos s’inicialitzen aleatòriament en cada repetició, ja em trobava amb el problema.

La resta de problemes m’eren més complicats de demostrar. Per començar, sabia que les meves dades estaven desequilibrades, ja que la música també ho està, però no creia que fos el causant del problema. És cert que en un inici els pesos de les notes extremes (com 0, 1… o 125, 126…), que són molt poc comunes, tenen valors similars a la resta de notes, però pel mateix procés d’entrenament aquests pesos es van ajustant.

Tot i que tots aquests problemes podrien ser causants d’un mal funcionament de la xarxa neuronal, vaig arribar a la conclusió que simplement el model de xarxa neuronal que estava utilitzant no em serviria per generar música. Aquest fet es deu a diversos motius, el principal dels quals és que l’estructura de la cançó en aquest tipus de model hauria d’estar predefinida per l’arquitectura de la xarxa, i que en la música les dades són dependents de l’anterior i tenen una relació lineal. Tot i això, em va sorprendre la capacitat d’aquest tipus de xarxa neuronal per adaptar-se a les dades durant l’entrenament.

En definitiva, vaig proposar-me crear un altre model de xarxa neuronal, aquest cop en lloc d’una feedforward neural network, una RNN. Aquesta nova versió del codi la repassaré en el següent apartat. En aquest punt del projecte ja havia assolit alguns dels meus objectius, ja que comprenia el funcionament general de la xarxa neuronal i l’havia aconseguit programar des de zero. A més, la xarxa era capaç d’entrenar, arribant a assolir un 40% de precisió en alguns dels casos. L’únic que em faltava era realitzar el pas final de generar una melodia original.

Millora del producte

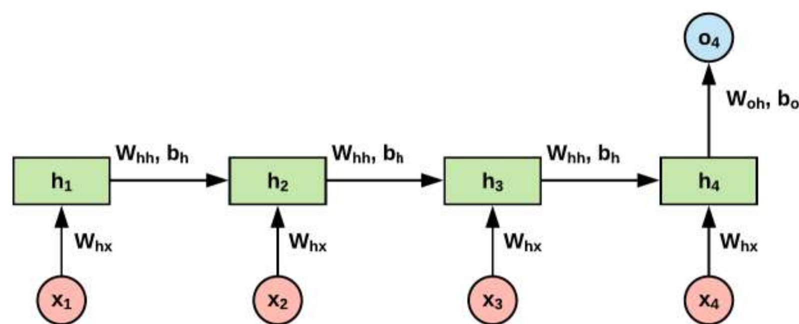
Les Xarxes Neuronals Recurrents (RNN) són un tipus de xarxa neuronal que incorpora bucles que permeten que la informació s’emmagatzemi. En certa manera, es poden entendre com una sèrie de xarxes neuronals normals que estan relacionades. Aquestes poden generar els outputs de diferents maneres.

Per exemple, una RNN One-to-Many genera diversos outputs a partir d’una sola entrada. Una RNN Many-to-one, en canvi, genera un sol output a partir de diversos inputs.

En el meu cas he utilitzat una xarxa Many-to-one, i l'estructura de les dades ha seguit sent la mateixa que en l'altra versió del codi (un array de 5 notes d'entrada correspon a una sortida).

Figura 11

Unfolded representation of the implemented RNN structure



Nota. De *Many to One RNN with Variable Sequence Length*, per Easy TensorFlow, 2018, (https://www.easy-tensorflow.com/tf-tutorials/recurrent-neural-networks/many-to-one-with-variable-sequence-length)

En aquest codi, la classe encarregada de crear la xarxa neuronal és la classe RNN. La funció d’inicialització d’aquesta classe és molt similar a la utilitzada en la classe capa de les

altres versions, ja que d'igual manera inicialitzem uns pesos. L'única diferència és que en lloc de crear-ne només uns, en creem tres de cop: els input to hidden (U), que són els que transformen els inputs que rep la xarxa; els hidden to hidden (W), que transformen la sortida que utilitzarà la xarxa següent; i els hidden to output (V), que són els utilitzats finalment en l'output.

La funció forward simplement el que fa és multiplicar primer els inputs pels pesos U i l'activació de la xarxa anterior (h) pels pesos W, i ho suma. Aplica la funció d'activació (en aquest cas la tanh, per evitar que els números siguin massa grans) i retorna aquest valor, que ara passarà a ser h.

La resta de funcions són idèntiques a les de les altres versions, ja que l'estructura de les dades que fa servir és la mateixa.

La part que torna a canviar és la part final del codi, en la que crida totes les funcions i classes i realitza l'entrenament.

L'inici del bucle de l'entrenament és molt similar al de la primera versió del codi (Annex 1), ja que actualitza els pesos aleatòriament (el GD és diferent en aquest tipus de xarxa neuronal, i m'era complicat implementar-lo). Seguidament inicialitza les variables del loss i del nombre d'encerts a 0, i crea un array ple de zeros que serà l'h que utilitzarà la primera xarxa. A continuació, hi ha un for que itera per tots els números dins del primer element de X (o sigui que fa tantes repeticions com notes hi ha per cada seqüència). Cal destacar que en un inici, hi havia un bucle que passava per tots els elements de totes les seqüències un a un, però fer això és molt ineficient. Per això, el que fem a partir d'aquest punt és cridar la xarxa i passar-li la variable h i tots els primers elements de les seqüències, llavors h i cada segon… Un cop hem passat per tots els valors, realitzem una multiplicació entre la sortida h de la xarxa i els weights_V, hi sumem els biases by i ho passem per la funció softmax.

A partir d'aquest punt, la xarxa ja és com la primera versió de totes: calcula l'error i actualitza els pesos en funció de si el loss és menor que el loss més petit.

Com hem pogut veure, la principal diferència entre aquesta xarxa i la resta és que les altres feien les operacions capa a capa, mentre que aquest tipus de xarxa es podria entendre com un grup de xarxes més simplificades que estan relacionades. Normalment s'utilitzen xarxes d'aquest estil en problemes com el llenguatge, ja que els valors anteriors influeixen en els resultats. Tot i això, com hem vist en el marc teòric, aquest models pateixen una ràpida

pèrdua de memòria, i per això quan s’utilitza una RNN la majoria dels cops es fan servir altres variants com LSTM.

Més endavant veurem que aquesta xarxa tampoc va acabar donant els resultats esperats, o sigui que els codis van acabar patint una última millora. Aquesta és molt senzilla, ja que simplement consisteix en utilitzar enlloc de les 127 notes, només 12 (les que conformen l’escala musical). Aquest fet va ajudar sobretot amb els models que tendien a predir notes molt extremes, ja que limita el rang de la cançó a una sola octava.

La forma de transformar les notes és molt simple, ja que bàsicament el que fa aquest codi és dividir cada nota per 12 i agafar-ne el residu, utilitzant el símbol % (operació mòdul). Per la generació final, hi suma 60 ja que sinó queden notes molt greus. Aquesta millora ha estat provada en una còpia de la versió 2 i una altra de la versió 3

Resultats i interpretació

Un cop creat el codi, va ser executat diverses vegades utilitzant paràmetres diferents, com el nombre de notes que utilitzava en l’entrenament, el nombre de neurones, el learning rate i el nombre de repeticions que realitzava. Aquest fet ens ha permès realitzar execucions del codi que han arribat a durar unes 12 hores (el màxim que normalment et permet la versió gratuïta del Colab) i d’altres que han tardat tan sols 5 minuts a completar-se. A continuació, explicarem més o menys els resultats que han anat donant les diferents versions del codi i els paràmetres utilitzats, juntament amb dades com l’encert.

Resultats de la primera versió

La primera versió del codi és la més senzilla de totes, i segurament amb la que s’han realitzat més proves que no han acabat de funcionar. Però tot i això, és interessant conèixer les primeres notes que van ser generades per la IA.

Cal dir, però, que en un inici la xarxa no comptava amb la part final del codi, encarregada de generar la cançó, i simplement entrenava. Com a cançó generada utilitzava les prediccions que la màquina realitzava durant l’entrenament. Aquesta versió, al estar teòricament inacabada, ha estat anomenada versió 0. Generava resultats com els següents:

Resultats versió 0

Aquests resultats van ser obtinguts utilitzant seqüències de 4 o 5 notes, 32 neurones i 100.000 repeticions. El learning rate que normalment utilitzava en un principi era de només 0.1. Una mica més tard, coincidint amb el moment en el qual es va crear la segona versió, va ser implementada la part encarregada de generar música. Amb aquesta part, els resultats van ser similars a aquests:

Resultats versió 1

Un fet important a destacar és que al ser un MLP senzill, aquesta versió és la més ràpida de les tres a l’hora de fer les repeticions. Més tard veurem les diferències entre les versions, i podrem veure clarament aquest fet.

Resultats de la segona versió

La segona versió del codi és la que més s’ha utilitzat a l’hora de generar música. Va ser durant bastant de temps la versió més “avançada” i la que obtenia millors resultats. De fet, al final, tot i haver creat la tercera versió utilitzant una xarxa diferent, els resultats d’aquesta eren millors. En general, vam adonar-nos que si la deixàvem entrenar poca estona els resultats no eren satisfactoris, però a mesura que provàvem més neurones i més repeticions van sortir altres melodies més interessants.

Cal dir que moltes de les sortides que generava eren descartades directament, sense ni tan sols escoltar la versió, ja que en la gràfica ja es veia que la xarxa havia generat una sola nota. Aquest és un problema mencionat en l’apartat de valoració del producte, i un dels que va impulsar a crear una tercera versió del codi per mirar si hi havia alguna possibilitat d’arreglar-lo.

També afegir que a vegades, tot i no quedar-se generant una sola nota, la xarxa acabava repetint les mateixes notes en bucle, provocant una melodia molt repetitiva. Tot i això, en algunes ocasions (sobretot quan s’estava bastants hores entrenant) acabava generant melodies més o menys originals.

Els paràmetres que normalment utilitzava aquest model acostumaven a ser 1000 notes d’entrada, agrupades en seqüències de 8, amb 1024 o 2048 neurones i un learning rate de 0.0000001. El nombre total de repeticions que realitzava variava entre 10.000 i 50.000.

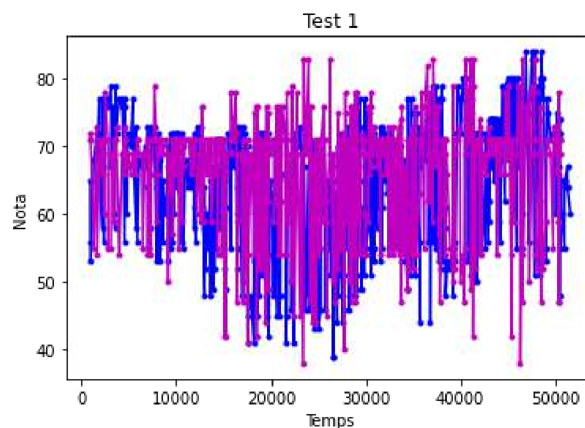
Alguns exemples dels resultats d’aquesta versió del codi són els següents:

Resultats segona versió

Aquesta versió generava gràfiques durant l’entrenament com la següent:

Figura 12

Gràfica segona versió



Nota. Elaboració pròpia

A partir d’aquesta gràfica, podem veure com la NN és capaç d’identificar les parts principals de la cançó, i ajustar-se a elles utilitzant el GD. És important destacar que, tot i que s’ajusta a les zones generals de la cançó, l’encert molts cops no supera el 6%.

Resultats tercera versió

La tercera versió creada, que utilitza una RNN, va donar resultats pitjors als esperats. En un inici confiava en el fet que utilitzant aquest altre tipus de xarxa neuronal es solucionarien els problemes que tenien les altres versions, però no sempre era així i fins i tot, en va provocar alguns de nous.

Aquesta versió del codi utilitzava una forma d'actualitzar els pesos molt ineficient, actualitzant-los aleatòriament, ja que la retropropagació funciona una mica diferent en aquest tipus de xarxes i tampoc tenia clar del tot com implementar-la. A més, al haver-hi reiteracions dintre el codi, aquesta versió és bastant més lenta que les altres.

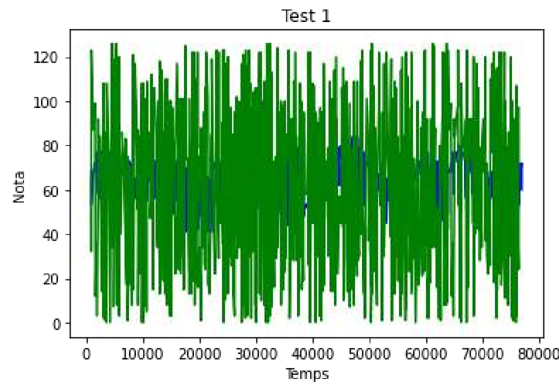
Els resultats generats per aquesta versió no contenen tants de bucles com la resta, però utilitzen notes molt més extremes. Si veiem les gràfiques, per exemple, de la versió 2, veiem que la nota més aguda no acostuma a ser més de 100, i la més greu no acostuma a baixar de 50. En canvi, amb aquesta versió hi ha bastants notes per sobre de 120 i per sota de 30.

Resultats tercera versió

Si ens fixem en les gràfiques, observem que no s'ajusta tant a les dades d'entrenament com la versió 2, però tot i així, segueix identificant l'estructura general. Veiem també com, en alguns punts, apareixen de cop i volta salts molt grans, però es recupera al cap de poc. Aquest fet podria estar degut a una mala inicialització dels pesos o un entrenament massa curt (ja que el Colab molts cops es desconecta automàticament a partir de les 10 hores).

Figura 13

Gràfica tercera versió



Nota. Elaboració pròpia

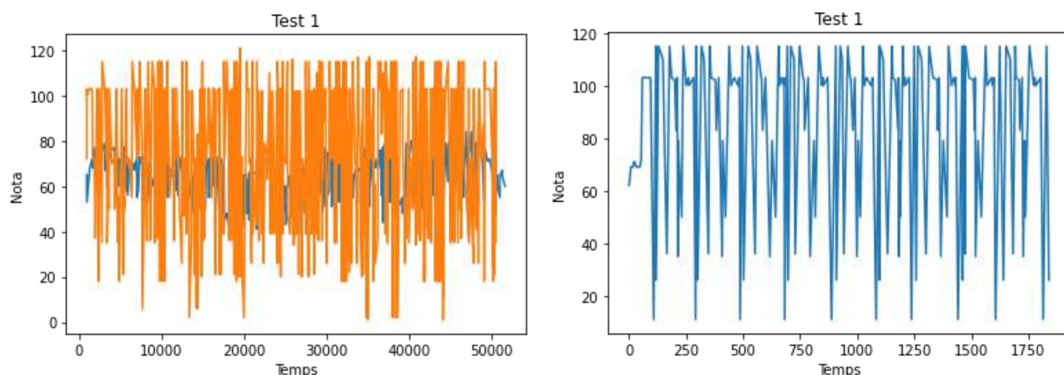
Comparacions entre versions

Per veure les diferències entre les xarxes, també vam realitzar algunes repeticions utilitzant els mateixos paràmetres per totes 3: 1000 notes d’entrada, seqüències de 8, 512 neurones i 10.000 repeticions. El learning rate va ser variat depenent de les versions, ja que la versió de RNN n’utilitza un molt més alt en comparació amb les altres. A continuació, mostrarem les 3 cançons generades i les gràfiques del procés d’entrenament, per tal de poder comparar els diferents codis.

La versió 1 va tardar tan sols 4:35 minuts a realitzar les 10000 repeticions. Va tenir un encert del 2%. Les gràfiques d’aquesta execució són les següents:

Figura 14

Gràfiques comparació primera versió

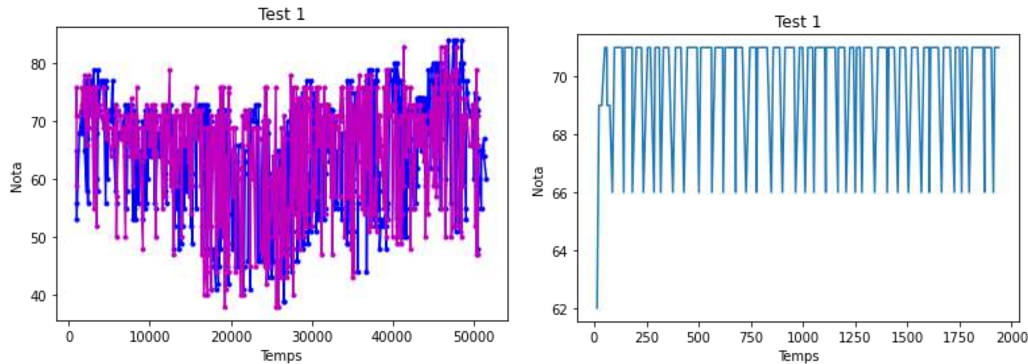


Nota. Elaboració pròpia

La versió que implementa la retropropagació va tardar lleugerament més, 5:47 minuts, i va obtenir un encert del 2.3% . Les seves gràfiques són:

Figura 15

Gràfiques comparació segona versió



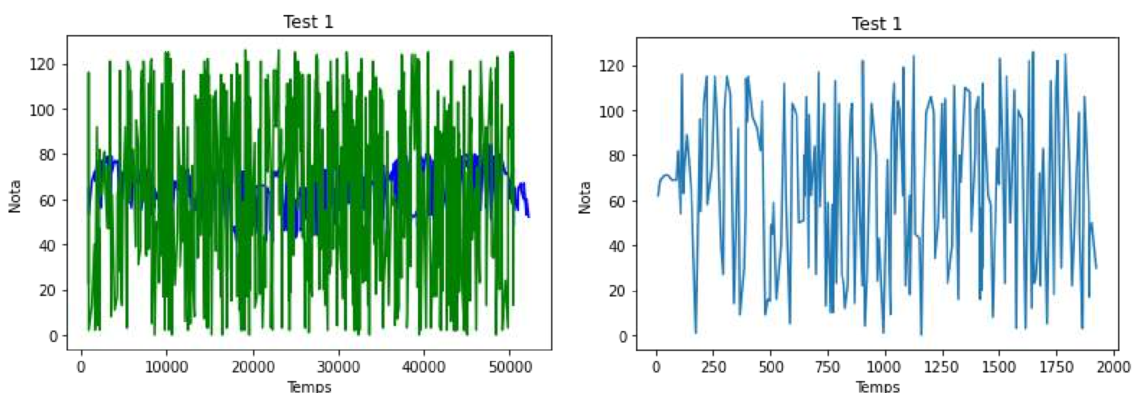
Nota. Elaboració pròpia

Podem observar com, tot i entrenar millor que les altres dues versions, a la hora de generar la nova cançó es queda encallada en un bucle.

La tercera versió, a diferència de les altres, va tardar 46:30 minuts i va tenir un encert de tan sols l'1%.. Com podem veure, és aproximadament 8 cops més lenta aquesta versió que les altres. Aquest fet es deu bàsicament, als 8 inputs que rep la xarxa, ja que ha de calcular totes les funcions d'una xarxa neuronal per cadascun d'aquests. A més a més, a la gràfica es pot observar el problema descrit anteriorment de les notes extremes. Tot i això, veiem que en la cançó generada no es troben bucles, a diferència de les altres dues versions.

Figura 16

Gràfiques comparació tercera versió



Nota. Elaboració pròpia

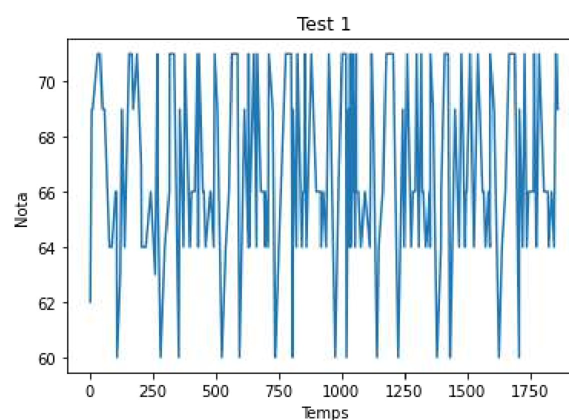
Les conclusions generals que podem treure d'aquestes proves són que cada una de les versions té un punt fort. La primera destaca per ser ràpida, però no entrena del tot bé i crea bucles a la hora de generar la cançó. La segona versió és la que més s'acosta a les dades d'entrenament, però la seva cançó també conté repeticions. La tercera en canvi, no acaba d'entrenar del tot bé, però tampoc presenta el problema de les reiteracions.

Resultats versions de dotze notes

Com hem comentat abans, l'última millora d'aquest projecte va ser la limitació de les notes a tan sols 12. Cal destacar que aquest fet permet que les cançons puguin ser tocades per un instrument de forma més senzilla, ja que els salts entre les notes no són massa grans. Addicionalment, és més ràpida i més precisa que la resta de versions, ja que el nombre de possibles outputs es redueix dràsticament. Les dues versions (TDR-RNN-12 i TDR-B-12) que implementen aquests canvis estan a l'annex 4, juntament amb alguns dels seus resultats. Tampoc ens centrarem gaire en aquests, ja que són molt similars a la resta de xarxes amb la diferència que el seu rang és més petit. Tot i això, personalment les melodies que m'agraden més són fruit d'aquests models, i són les que més s'assemblen a les cançons "normals".

Figura 17

Gràfica cançó generada per TDR-B-12



Nota. Elaboració pròpia

Conclusions

L’objectiu principal d’aquest treball era crear una xarxa neuronal, i aquest ha sigut no tan sols assolit, sinó superat, ja que hem acabat desenvolupant un total de cinc models funcionals. Vist això, també podem determinar que hem aconseguit comprendre com funcionen aquestes xarxes, i com aprenen.

La recerca que hem dut a terme durant tot el procés de creació del treball, des de la definició d’intel·ligència artificial fins als possibles problemes de la nostra xarxa, ens ha permès establir una base sòlida de coneixements que ha facilitat tot el procés de programació i anàlisis dels resultats. Hem vist els diferents exemples d’IA, hem establert de quina forma els ordinadors poden concebre la música, hem fet un resum de la història de la composició automatitzada, i després de tot això, hem après a programar una xarxa neuronal i l’hem aplicat al nostre objectiu. També hem analitzat les diferents parts que conformen aquest tipus d’IA, des dels pesos fins a les funcions d’activació.

El fet d’haver creat des de zero una xarxa ha facilitat molt la comprensió del funcionament d’aquestes en comparació a si haguéssim utilitzat un model ja existent. Cal destacar que el fet de programar la IA ha estat basat en l’assaig i l’error, i que sense anar provant el codi per parts és molt complicat fer-se una idea precisa sobre què fa realment.

Durant aquest treball, hem conegut alguns dels models més potents capaços de generar cançons, i hem descobert que molts d’ells utilitzen tècniques com LSTM, GAN, GRU… Hem detectat que la majoria utilitzen xarxes neuronals recurrents, i hem aplicat aquesta arquitectura a la nostra xarxa en la tercera versió. Ens hem adonat, també, que classificar en 127 notes diferents és una tasca molt complicada, i per això hem creat altres versions amb menys classes possibles.

El fet d’haver creat totes aquestes versions ens ha permès comparar-les entre elles. Hem vist que les xarxes neuronals aprenen molt millor utilitzant tècniques d’optimitzation, com el gradient descent, hem detectat que els MLP no acostumen a ser gaire bons utilitzant dades seqüencials i ens hem adonat de la gran importància que té la velocitat d’execució d’aquests programes.

Les xarxes neuronals tenen un munt de paràmetres que es poden modificar. Alguns els modifica la mateixa IA per tal d’entrenar, com els pesos i els biases, però d’altres li han de

ser introduïts per l’usuari. El fet de disposar de tantes combinacions possibles diferents ha provocat que hi hagi una gran quantitat de resultats molt diversos. Alguns no tenen gaire contingut musical, i es queden encallats en una nota, però en d’altres la xarxa ha acabat generant melodies diferents. Algunes d’aquestes realment semblen notes aleatòries, i podria ser perfectament una cançó creada per un nen de tres anys que desconeix la música, però com hem vist al principi, el nostre objectiu no era crear una gran obra sinó veure si es podia crear una NN amb recursos limitats.

Propostes de millora

A partir del que hem anat veient de xarxes neuronals i havent parlat amb persones del meu entorn, he vist que hi ha unes quantes possibles millores a les xarxes que he creat. Aquestes no les he pogut dur a terme ja sigui per temps, per recursos o per falta de coneixement.

La primera millora possible seria, bàsicament, utilitzar un model més avançat, amb més capes i que utilitzés altres estratègies (com un model de deep-learning que emprés una LSTM). Això hauria permès poder treballar amb quantitats de dades més grans, de formes més eficients i amb millors resultats, però hagués suposat una inversió d’hores i esforç massa grans per una sola persona. Cal tenir en compte que molts cops aquests models els desenvolupen equips formats per bastants persones expertes en la matèria. També hauria millorat els resultats del producte haver pogut disposar d’ordinadors potents de forma il·limitada, ja que els períodes d’entrenament haguessin pogut ser més llargs i les repeticions tardarien menys a realitzar-se.

Un altre fet que hagués pogut ser interessant d’implementar hagués estat que la xarxa fos capaç de generar no només melodies, sinó cançons polifòniques (diverses veus), i fins i tot, diferents instruments. Tot i que en un principi, quan encara no coneixia pràcticament res del tema, vaig arribar a pensar que seria possible, em vaig adonar ràpidament que era molt complicat.

Una altra millora, bastant evident, seria el fet que el programa generés no només notes, sinó també ritme (sense estar determinat aleatòriament). De fet, tenia pensat mirar de transformar les dades d’una altra forma, fent que la durada de cada nota es correspongui amb la quantitat de cops que apareixia. Per exemple, si la nota 32 durés 5 temps, a la llista d’entrada de notes hi hauria [32, 32, 32, 32, 32]. Encara que la dificultat per implementar

una funció que agrupés les dades d’aquesta manera no seria gaire elevada, sí que necessitaria una xarxa més potent, ja que el nombre d’inputs augmenta molt.

En definitiva, hi ha bastants coses que es podrien millorar, però tot i això, el codi creat permet assolir els objectius proposats a l’inici del treball. Tenint en compte que és un model senzill, creat per una sola persona, i que estem treballant amb unes dades molt complicades de classificar, alguns resultats són bastant sorprenents. A més, si comparem els 100.000 milions de neurones que conté el cervell humà amb les 4000 de la xarxa (en les repeticions que n’he utilitzat més), podem adonar-nos de la gran capacitat d’adaptació de les xarxes neuronals.

Conclusions personals

Tot i que en un inici tenia unes idees molt ambicioses sobre què faria el meu codi, i més endavant em vaig adonar que no seria possible assolir-les, he quedat satisfet amb el producte final. Com hem vist, encara hi ha moltes possibles millores, però moltes d’aquestes no serien compatibles amb els objectius del treball.

Personalment, la recerca realitzada i el fet de crear el codi des de zero m’han servit per adquirir molts coneixements sobre un tema del qual simplement havia sentit a parlar. És un tema que de primeres sembla molt complicat, i pot arribar a costar d’entendre, però penso que el fet d’haver programat una xarxa des de zero m’ha facilitat molt comprendre com funciona. També m’ha sigut útil el fet que en els últims anys sigui un camp que ha rebut molt protagonisme, ja que no m’ha sigut tan complicat trobar articles i llibres que parlessin sobre el tema.

Pel que fa a les cançons generades, trobo que n’hi ha alguna que té patrons de notes interessants. L’únic problema que els hi trobaria és que el ritme aleatori no acaba de quedar del tot bé, i fa que semblin cançons molt abstractes. Si bé és cert que en alguns moments queda millor amb aquests ritmes que si fos el mateix tota l’estona, a vegades provoca que la melodia sembli pitjor del que és. Tot i això, en general estic content amb el que he assolit.

Conclusions finals

Aquest treball ens ha permès entendre com és capaç de funcionar una intel·ligència artificial, comprendre què és realment una xarxa neuronal i com funciona, i els problemes més comuns amb els que ens podem trobar utilitzant aquestes xarxes. Hem vist també que, al ritme que progressen les NN, d’aquí a pocs anys és possible que veiem cançons parcialment o totalment compostes per una IA entre les més populars. En general, han estat assolits tots els objectius proposats, tot i que alguns models no han donat els resultats esperats. En definitiva, encara que la música tingui un component artístic molt important, aquest fet no impedeix que les xarxes siguin teòricament capaces de crear cançons originals i creatives.

És per tot això que podem concloure que, encara que sembli contradictori, es pot arribar a crear art sense tenir idees, sensacions, percepcions o emocions.

Bibliografia

About AIVA. (2016). AIVA. https://www.aiva.ai/about

AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference? (2021, June 18). IBM. https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks

Alameda, T. (2019, November 11). *Machine learning: What is it and how does it work?* NEWS BBVA. https://www.bbva.com/en/machine-learning-what-is-it-and-how-does-it-work/

Barlow, H. (1989). Unsupervised Learning. *Neural Computation*, *1*(3), 295–311. https://doi.org/10.1162/neco.1989.1.3.295

Borgos, E. (2021, May 7). *How I Made One of the World's First 100% AI Songs - Towards Data Science*. Medium. https://towardsdatascience.com/how-i-made-one-of-the-worlds-first-100-ai-songs-45da7297075c

Briot, J. P., Hadjeres, G., & Pachet, F. D. (2017). Deep Learning Techniques for Music Generation. *Computational Synthesis and Creative Systems*. Published. https://arxiv.org/pdf/1709.01620.pdf

Briot, J. P., Hadjeres, G., & Pachet, F. D. (2019). *Deep Learning Techniques for Music Generation* [E-book]. Springer Publishing.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplant, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D.

(2020, May 28). *Language Models are Few-Shot Learners*. ArXiv.Org. https://arxiv.org/abs/2005.14165

Brownlee, J. (2020a, August 25). *How to Choose Loss Functions When Training Deep Learning Neural Networks*. Machine Learning Mastery. https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Brownlee, J. (2020b, September 11). *Understand the Impact of Learning Rate on Neural Network Performance*. Machine Learning Mastery. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Cauchy, A. (1847). Methode generale pour la resolution des systemes d’equations simultanees. *C.R. Acad. Sci. Paris*, *25*, 536–538. https://cs.uwaterloo.ca/~y328yu/classics/cauchy-fr.pdf

Cope, D. (1996). *Experiments in Musical Intelligence*. A-R Editions.

Copeland, B. J. (2020, August 11). *Artificial intelligence*. Britannica. https://www.britannica.com/technology/artificial-intelligence

de Jong, K., Fogel, D. B., & Schwefel, H.-P. (1997). A history of evolutionary computation. In *A history of evolutionary computation.* (p. A2.3:1–12). Oxford University Press. https://www.researchgate.net/publication/216300863_A_history_of_evolutionary_computation

Decaro, C., Montanari, G. B., Molinari, R., Gilberti, A., Bagnoli, D., Bianconi, M., & Bellanca, G. (2019). Machine Learning Approach for Prediction of Hematic Parameters in Hemodialysis Patients. *IEEE Journal of Translational Engineering in Health and Medicine*, *7*, 1–8. https://doi.org/10.1109/jtehm.2019.2938951

Dickson, B. (2019, November 18). *What is symbolic artificial intelligence?* TechTalks. https://bdtechtalks.com/2019/11/18/what-is-symbolic-artificial-intelligence/

Difference between Slope and Gradient. (2012, September 4). Mathematics Stack Exchange. https://math.stackexchange.com/questions/190756/difference-between-slope-and-gradient/190760

Eck, D., & Schmidhuber, J. (2002). Finding temporal structure in music: blues improvisation with LSTM recurrent networks. *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*. Published. https://doi.org/10.1109/nnsp.2002.1030094

Evolution by Keiwan. (2019). Itch.Io. https://keiwan.itch.io/evolution

FAQs. (2021). AI Song Contest 2021. https://www.aisongcontest.com/faqs

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence Through Simulated Evolution*. Wiley. https://www.semanticscholar.org/paper/Artificial-Intelligence-through-Simulated-Evolution-Fogel-Owens/a9e41a611b3b57b828775a45a7d74a1c75ed3f20

Galbusera, F., Casaroli, G., & Bassani, T. (2019). *Schematic overview of the main branches of artificial intelligence (AI), including machine learning (ML) methods which are having an impact on spine research* [Illustration]. https://www.researchgate.net/figure/Schematic-overview-of-the-main-branches-of-artificial-intelligence-AI-including_fig1_330878118

GeeksforGeeks. (2019, May 6). *History of Python*. https://www.geeksforgeeks.org/history-of-python/

Georgeon, O. L., Casado, R. C., & Matignon, L. A. (2015). *Les relacions agent-medi en el RL* [Diagram]. https://www.researchgate.net/figure/The-agent-environment-interaction-in-reinforcement-learning-1-Figure-31-On-time-t_fig1_289991380

Google Colaboratory. (n.d.). Google Colab. Retrieved August 27, 2021, from https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index

Hiller, L. A., & Isaacson, L. M. (1959). *Experimental Music*. McGraw-Hill. https://archive.org/details/experimentalmusi00hill/page/182/mode/2up

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Ibañez, D. (2020, March 3). *Artificial Neural Networks – The Rosenblatt Perceptron*. Neuroelectrics Blog - Latest News about EEG & Brain Stimulation. https://www.neuroelectrics.com/blog/2016/08/02/artificial-neural-networks-the-rosenblatt-perceptron/

Johnson, G. (1997, November 11). Undiscovered Bach? No, a Computer Wrote It. *The New York Times*. https://www.nytimes.com/1997/11/11/science/undiscovered-bach-no-a-computer-wrote-it.html

Karpathy, A. (2020, August 29). *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. OpenAI. https://openai.com/blog/evolution-strategies/

Kinsley, H., & Kukieła, D. (2020). *Neural Networks from Scratch in Python*. https://books.google.es/books/about/Neural_Networks_from_Scratch_in_Python.html?id=Ll1CzgEACAAJ&redir_esc=y

Lenssen, N., & Needell, D. (2014). *Espectrograma i chroma feature* [Graph]. https://www.researchgate.net/figure/The-spectrogram-top-and-chromagram-bottom-of-an-ascending-scale_fig3_314918556

Lewis. (1988). Creation by refinement: a creativity paradigm for gradient descent learning networks. *IEEE International Conference on Neural Networks*. Published. https://doi.org/10.1109/icnn.1988.23933

Magenta. (2016). *GitHub - Magenta: Music and Art Generation with Machine Intelligence*. GitHub. https://github.com/magenta/magenta

Mallawaarachchi, V. (2020, March 1). *Introduction to Genetic Algorithms — Including Example Code*. Medium. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

Marolt, M., Kavcic, A., & Privosnik, M. (2002). Neural Networks for Note Onset Detection in Piano Music. *Proceedings of the International Computer Music Conference (ICMC)*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2277

Maurer, J. A. (1999). *The History of Algorithmic Composition*. CCRMA. https://ccrma.stanford.edu/%7Eblackrse/algorithm.html

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133. https://doi.org/10.1007/bf02478259

Merriam-Webster. (2021). Music. In *Merriam-Webster*. https://www.merriam-webster.com/dictionary/music

Message Types — Mido 1.2.10 documentation. (2020). Mido Documentation. https://mido.readthedocs.io/en/latest/message_types.html

Mishra, M. (2018, July 2). *REGULARIZATION: An important concept in Machine Learning*. Medium. https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea

Mishra, S. (2018, June 21). *Unsupervised Learning and Data Clustering - Towards Data Science*. Medium. https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a

Müller, M. (2015). *PianoRoll*. Audiolabs. https://www.audiolabs-erlangen.de/resources/MIR/FMP/C1/C1S2_PianoRoll.html

MusicXML for Exchanging Digital Sheet Music. (2020, October 1). MusicXML. https://www.musicxml.com/home

Neural Networks - History. (2000). Neural Networks Eric Roberts. https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

OpenAi. (2021, June 22). *DALL·E: Creating Images from Text*. https://openai.com/blog/dall-e/

Osiński, B., & Budek, K. (2021, January 5). *What is reinforcement learning? The complete guide*. Deepsense.Ai. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/

Pai, A. (2020, October 19). *ANN vs CNN vs RNN | Types of Neural Networks*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

Payne, C. M. (2021, June 21). *MuseNet*. OpenAI. https://openai.com/blog/musenet/

Pere, C. (2020, June 18). *What are Loss Functions? - Towards Data Science*. Medium. https://towardsdatascience.com/what-is-loss-function-1e2605aeb904

Pons, J. (2018, November 5). *Neural Networks For Music: A Journey Through Its History*. Medium. https://towardsdatascience.com/neural-networks-for-music-a-journey-through-its-history-91f93c3459fb

Quiza, R., & Davim, J. P. (2009). *Capes d’una xarxa neuronal* [Graph]. https://www.researchgate.net/figure/Graph-of-a-feed-forward-neural-network_fig2_234055140

Robbins, H., & Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, *22*(3), 400–407. https://doi.org/10.1214/aoms/1177729586

Ronaghan, S. (2019, August 1). *Deep Learning: Which Loss and Activation Functions should I use?* Medium. https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8

Rosenberg, G. S., Cina, A., Schirò, G. R., Giorgi, P. D., Gueorguiev, B., Alini, M., Varga, P., Galbusera, F., & Gallazzi, E. (2021). ARTIFICIAL INTELLIGENCE ACCURATELY DETECTS TRAUMATIC THORACOLUMBAR FRACTURES ON SAGITTAL RADIOGRAPHS. *MedRxiv*. Published. https://doi.org/10.1101/2021.05.09.21256762

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

Ruder, S. (2016, September 15). *An overview of gradient descent optimization algorithms*. ArXiv.Org. https://arxiv.org/abs/1609.04747

Ryan, M. (2018). *Esquema general d'una xarxa neuronal* [Diagram]. https://medium.datadriveninvestor.com/how-neural-network-process-your-input-trained-neural-network-fd48f1bf310

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Sionsoft. (2020, January 21). *MIDI Messages*. https://www.sionsoft.com/MIDI/MIDIMessages.html

TECHSLANG. (2020, November 16). *What is Symbolic AI: Examining Its Successes and Failures*. Techslang — Tech Explained in Simple Terms. https://www.techslang.com/what-is-symbolic-ai-examining-its-successes-and-failures/

The MIDI Association. (2021a, April 4). *MIDI History:Chapter 6-MIDI Is Born 1980–1983*. https://www.midi.org/articles/midi-history-chapter-6-midi-is-born-1980-1983

The MIDI Association. (2021b, May 24). *About MIDI-Part 3:MIDI Messages*. https://www.midi.org/midi-articles/about-midi-part-3-midi-messages

Todd, P. M. (1989). A Connectionist Approach to Algorithmic Composition. *Computer Music Journal*, *13*(4), 27. https://doi.org/10.2307/3679551

Walshaw, C. (2010, January 31). *How to understand abc . . . the basics « abc notation blog*. Abc Notation Blog. https://abcnotation.com/blog/2010/01/31/how-to-understand-abc-the-basics/

Weng, L. (2019, September 5). *Evolution Strategies*. Lil’Log. https://lilianweng.github.io/lil-log/2019/09/05/evolution-strategies.html#what-are-evolution-strategies

Wilson, A. (2019, October 1). *A Brief Introduction to Supervised Learning - Towards Data Science*. Medium. https://towardsdatascience.com/a-brief-introduction-to-supervised-learning-54a3e3932590

Wood, T. (2020, September 27). *Activation Function*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/activation-function

Xenakis, I. (1992). *Formalized Music: Thought and Mathematics in Composition (Harmonologia Series)* (2nd ed.). Pendragon Press. https://www.amazon.com/Formalized-Music-Mathematics-Composition-Harmonologia/dp/1576470792

Zaremba, W., Brockman, G., & OpenAi. (2021, August 10). *OpenAI Codex*. OpenAI. https://openai.com/blog/openai-codex/

Zaripov, R. K. (1960). An algorithmic description of a process of musical composition. *Dokl. Akad. Nauk SSSR*, *132*(6), 1283–1286. http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=dan&paperid=23732&option_lang=eng

Annexos

Annex 1 : primera versió de la xarxa

Codi

TDR-Pau

Resultats

Resultats versió 1[1]

[1] Les melodies generades estan en format MIDI. Per escoltar-les es pot utilitzar el Windows Media Player o altres aplicacions de tercers.

Annex 2 : segona versió de la xarxa

Codi

TDR-B

Resultats

Resultats versió 2

Annex 3 : tercera versió de la xarxa

Codi

TDR-RNN

Resultats

Resultats versió 3

Annex 4 : versions de dotze notes

TDR-B-12

TDR-RNN-12

Resultats dotze notes

Annex 5

Aprofitant que sé tocar el violoncel, em semblava interessant tocar una de les cançons que havia generat la meva IA a la presentació del treball. Per convertir el fitxer MIDI en una partitura que pugui llegir, vaig utilitzar un programa de creació de partitures anomenat MuseScore, ja que és capaç de llegir aquests tipus de fitxer automàticament. Com que el violoncel és un instrument més aviat greu, també vaig haver de transposar la cançó una octava avall.

Per tal que les persones que no estiguin a la presentació d’aquest treball puguin escoltar aquesta melodia, a continuació adjunto un MP3 generat també pel programa MuseScore perquè es facin a la idea de com sona.

Cançó generada per la versió RNN-12

♩ = 84

6

10

15

21

27

34

40

44

48