Języki i narzędzia programowania III
Machine Learning w praktyce

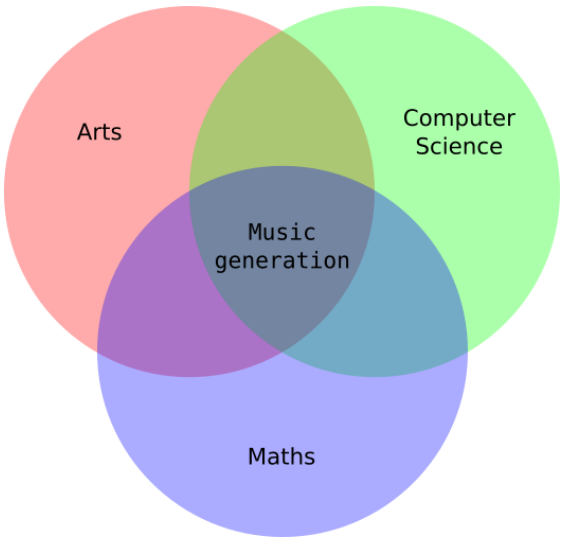
Generowanie muzyki do gier

Dawid Borys, Zofia Partyka

Semestr zimowy 2019

1 Wizja

Główną motywacją do stworzenia projektu była chęć samodzielnego eksperymentowania z szeroko znanym tematem generowania muzyki, korzystając z melodii, które znamy i lubimy. Stwierdziliśmy również, że wykorzystując podstawową wiedzę z teorii muzyki, jesteśmy w stanie osiągnąć ciekawe wyniki. Projekt ten może stać się również solidną bazą do realizacji bardziej dalekosiężnej wizji generowania muzyki do gier na podstawie otoczenia, liczby przeciwników, nastroju, etc. W sytuacji idealnej mielibyśmy cechy odpowiadające za na przykład dramatyczność, tempo, natężenie dźwięku w zależności od sytuacji na ekranie. Analogiczny model uczony na innej bazie danych mógłby generować muzykę pod filmy nieme!



Rysunek 1: Diagram Venna reprezentujący schemat myślowy autorów.

2 Baza danych

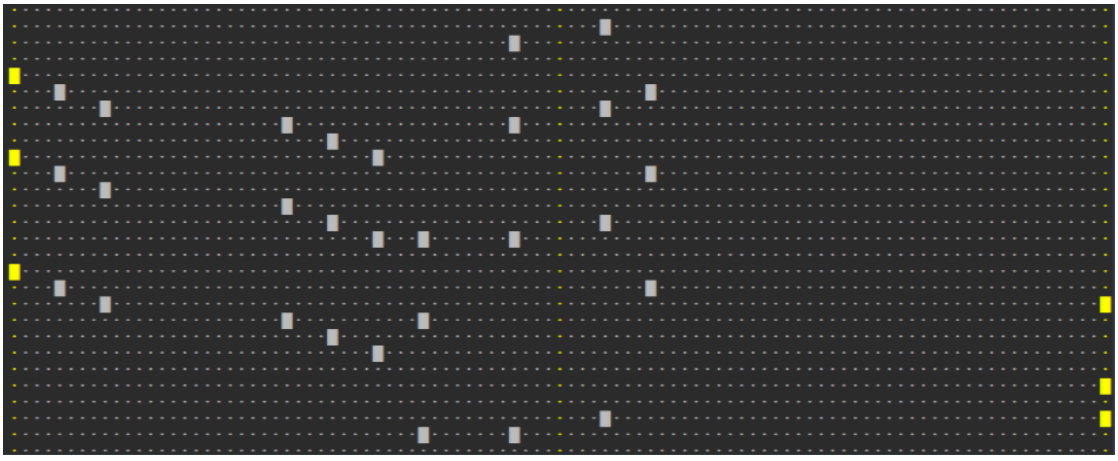
Pobraliśmy **4000** plików MIDI z https://www.ninsheetmusic.org/. Strona kolekcjonuje muzykę z gier komputerowych kultowej marki Nintendo. Wszystkie utwory na stronie są aranżacjami na pianino. Dzięki temu, że są one aranżacjami na jeden instrument, dane są stosunkowo jednolite.

Kolejnym problemem była różna tonacja utworów. Nie wgłębiając się zanadto w teorię muzyki, każdy utwór może być napisany w jednej z 12 tzw. tonacji. Początkowo baza danych byłą nieprzetworzona, co generowało melodie w których różne tonacje „gryzły" się ze sobą. W późniejszej fazie eksperymentu rozwiązaliśmy ten problem poprzez przetransponowanie wszystkich utworów do jednej tonacji (tj. zmianę wysokości wszystkich dźwięków w danym utworze o ten sam skalar). Jakość generowanych utworów istotnie się poprawiła.

2.1 Reprezentacja danych

Jedna piosenka (w naszym programie reprezentowana przez klasę `Song`) jest trójwymiarową tablicą zer i jedynek: **16** taktów × **48** jednostek czasu × **88** wysokości dźwięku. A zatem np. `song[1][15][50] == 1` oznacza, że w pierwszym[1] takcie, w 15. jednostce czasu, piosenka zawiera dźwięk o wysokości 50.

Na rysunku 2 zwizualizowany został początek pewnej piosenki z bazy danych. Prostokąt to 1 — dźwięk, a kropka 0 — brak dźwięku. Oś pozioma reprezentuje czas, a oś pionowa (tutaj: tylko fragment) wysokość dźwięku. Muzyka gra więc od lewej do prawej, a wyżej umieszczone prostokąty odpowiadają wyższym dźwiękom. Dźwięki w jednej kolumnie grają jednocześnie. Żółte linie wyznaczają początki taktów: widzimy zatem dwa pierwsze takty i pierwszą jednostkę czasu trzeciego taktu.



Rysunek 2: Wizualizacja pewnej piosenki z bazy danych.

[1] Zakładając, że takty liczymy od zera.

Wysokość dźwięku w naszym programie P_* (z ang. *pitch*) to ilość półtonów[2] od najniższego klawisza w pianinie. Jednoznacznie odpowiada to wysokości dźwięku w standardzie MIDI P_{MIDI} równaniem:

$$P_* = P_{MIDI} - 21$$

Jedną jednostkę czasu definiujemy jako 1/48 taktu. Nie zawiera zatem w sobie prędkości odtwarzania. Wynika to z reprezentacji danych w standardzie MIDI. Jako tempo przyjmujemy zatem domyślne w muzyce $\text{♩} = 120$, co oznacza, że każdy takt trwa dokładnie 2 sekundy.

3 Architektura programu

Wybraliśmy znany w świecie uczenia maszynowego język `Python`, z wykorzystaniem biblioteki `PyTorch`. Pełny kod programu dostępny jest w repozytorium pod adresem: https://github.com/rotifyld/music-generation.

3.1 Architektura sieci neuronowej

Po przeczytaniu kilku artykułów w sieci na temat generowania muzyki przy pomocy uczenia maszynowego, przed szereg wybiły się dwa modele: sieci *RNN* i *Autoencoder*. Te dwa podejścia różnią się od siebie sposobem generacji danych. W naszym przypadku *Autoencoder* z danych wejściowych generuje od razu cały utwór o stałej długości. *RNN* dynamicznie generuje kolejne fragmenty wyjścia (w naszym przypadku rozważaliśmy, by były to takty).

Na potrzeby naszego projektu *Autoencoder* będzie w generować melodię, która domyślnie się nie zmienia (16 taktów, które zapętlają się), ale też w razie zapotrzebowania (zmiany parametrów) może **stopniowo** się zmieniać. *RNN* nie daje takiej możliwości i przez cały czas generuje nowy materiał muzyczny. Dzięki tej cesze *RNN* jest często wykorzystywany do tworzenia muzyki o wolnej strukturze jak np. jazz. W naszym przypadku jednak zależało nam na powtarzalności motywów i „chwytności" muzyki — wybraliśmy zatem architekturę *Autoencoder*.

3.1.1 Autoencoder

Autoencoder to model sieci jednokierunkowej mającej na celu zakodowanie sygnału wejściowego, a następnie jego zdekodowanie z jak najmniejszą stratą i różnicą w wynikach. Jest to uczenie bez nadzoru, ponieważ dane nie są w żaden sposób etykietowane. *Autoencoder* redukuje wymiar sygnału wejściowego, a następnie stara się ze zredukowanej postaci go odtworzyć. *Autoencoder* składa się z dwóch funkcji, *encodera* E i *decodera* D, które kolejno przeprowadzają sygnał z wymiaru wejściowego do (mniejszego) wymiaru cech i z powrotem:

[2] Najmniejsza jednostka wysokości dźwięku.

$$E : \mathcal{X} \to \mathcal{F}$$
$$D : \mathcal{F} \to \mathcal{X},$$

gdzie $\mathcal{X}$ to wymiar sygnału wejściowego, a $\mathcal{F}$ to wymiar wektora cech. Wektor wejściowy X jest przeprowadzony w wektor wyjściowy $X' = (D \circ E)(X)$. Sieć uczona jest tak, aby zminimalizować różnicę pomiędzy X, a X'.

3.1.2 Generowanie danych

Stanem pośrednim pomiędzy wejściem a wyjściem jest wektor cech $f = E(X)$. W późniejszych iteracjach eksperymentu dodaliśmy w *encoderze* normalizację *batchów*, dzięki czemu wektor cech jest postaci: $f \in \mathcal{F} = [-1; 1]^{128}$. Dzięki temu można wykorzystać *Autoencoder* do wygenerowania losowych danych poprzez podanie mu losowego wektora wejściowego: $f_{rand} \in \mathcal{F}$. Po zastosowaniu (wytrenowanej) funkcji *decodera* na takim losowym wektorze, otrzymamy losową piosenkę: $X_{rand} = D(f_{rand})$. W praktyce utwór wygenerowany na wytrenowanym *Autoencoderze* będzie w przybliżeniu pewną kombinacją liniową wszystkich utworów.

3.1.3 Podwójny autoencoder

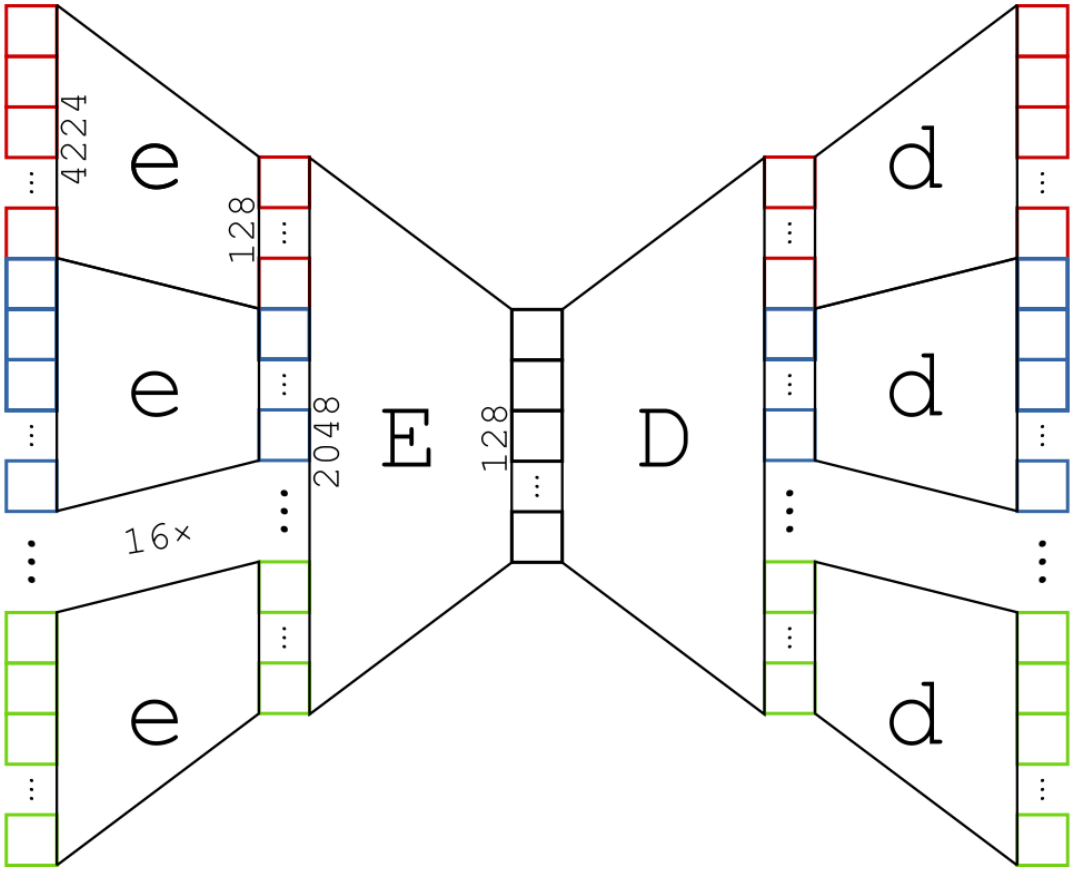
Dane dajemy do *Autoencodera* w dwóch turach: najpierw dzielimy piosenkę na takty i enkodujemy takty, a następnie znowu je konkatenujemy i enkodujemy całą piosenkę. Dekoder działa symetrycznie. Czyli mamy dwie funkcje enkodujące, do taktów i piosenki i dwie dekodujące analogiczne. Oznaczmy je kolejno e, E, D, d:

$$e : \mathbb{R}^{4224} \to \mathbb{R}^{128}$$
$$E : \mathbb{R}^{2048} \to \mathbb{R}^{128}$$
$$D : \mathbb{R}^{128} \to \mathbb{R}^{2048}$$
$$d : \mathbb{R}^{128} \to \mathbb{R}^{4224}$$

czyli bierzemy piosenkę $S = M_0 \cdot M_1 \ldots M_{15}$ i przeprowadzamy ją przez funkcje (schemat poniżej):

$$F = E\Big(e(M_0) \cdot e(M_1) \cdot \ldots \cdot e(M_{15})\Big)$$
$$\text{i dzieląc teraz } F = f_0 \cdot f_1 \cdot \ldots \cdot f_{15}$$
$$S' = D\Big((d(f_0) \cdot d(f_1) \cdot \ldots \cdot d(f_{15})\Big)$$

gdzie F nazywamy wektorem cech (ang. *feature vector*).



Rysunek 3: Schemat architektury sieci.

Jako funkcję straty przyjęliśmy początkowo MSE. Nie dawała ona jednak zadowalających wyników, więc zmieniliśmy ją po pierwszej iteracji na BCE (*Binary Cross Entropy*). Opis matematyczny w naszym projekcie:

$$BCE = \text{avg}\Big(- S \cdot log(S') - (1 - S) \cdot log(1 - S') \Big)$$

$S, S' \in \mathbb{R}^{4224 \cdot 16}$ — wektory odpowiednio wejściowe i wyjściowe,
$\text{avg} : \mathbb{R}^{4224 \cdot 16} \to \mathbb{R}$ — średnia arytmetyczna elementów w wektorze,
a wszystkie pozostałe operacje $(-, \cdot, log)$ są operacjami na *elementach* wektorów.

4 Wyniki

Wykonaliśmy 4 iteracje eksperymentów z niemalejącym poziomem skomplikowania modelu. Poniżej opisane są kolejne zmiany i otrzymane efekty.

4.1 Model Bazowy

```
# measure encoder
e = nn.Sequential(
    nn.Linear(4224, 128),
    nn.ReLU(True)
)

# song encoder
E = nn.Sequential(
    nn.Linear(2048, 128),
    nn.Tanh()
)
```

```
# measure decoder
d = nn.Sequential(
    nn.Linear(128, 4224),
    nn.Sigmoid()
)

# song decoder
D = nn.Sequential(
    nn.Linear(128, 2048),
    nn.ReLU(True),
)
```

50 piosenek, 200 epok `1.1.mp3`[3]

Początkowo, aby przetestować poprawność programu, uczyliśmy model na zmniejszonej bazie danych. Pierwszymi obiecującymi wynikami było otrzymanie dla wszystkich losowych danych wejściowych niemal identycznego utworu będącego przemieszaniem pewnych dwóch utworów z bazy danych. Wygenerowane utwory między sobą nie były jednak identyczne — zmiana parametrów nieznacznie wpływała na wynik końcowy.

4017 piosenek, 200 epok `1.2.mp3`

Wytrenowanie modelu na pełnej bazie danych nie poprawiło rezultatów, ale powieliło błędy poprzednika: wszystkie wygenerowane piosenki są identyczne, a dodatkowo wszystkie takty w każdej piosence są niemalże identyczne. Niemniej, brzmi dość chwytliwie.

[3]Wszystkie piosenki wspomniane w tym raporcie są dostępne pod adresem: (z możliwością odtworzenia w przeglądarce)
https://1drv.ms/u/s!Akwt6Uz7ay76ziKqNfCkM1bH8HiB?e=p1pRx0.

4.2 Warstwy ukryte

Pogrubieniem zaznaczono fragmenty kodu zmienione względem poprzedniej iteracji.

```
 # measure encoder
e = nn.Sequential(
    nn.Linear(4224, 1024),
    nn.ReLU(True),
    nn.Linear(1024, 128),
 )

# song encoder
E = nn.Sequential(
    nn.Linear(2048, 512),
    nn.ReLU(True),
    nn.Linear(512, 128),
)
```

```
# measure decoder
d = nn.Sequential(
    nn.Linear(128, 1024),
    nn.ReLU(True),
    nn.Linear(1024, 4224),
    nn.Sigmoid()
)

# song decoder
D = nn.Sequential(
    nn.Linear(128, 512),
    nn.ReLU(True),
    nn.Linear(512, 2048),
)
```

4017 piosenek, 50 epok `2.1.mp3`

Żadne z powyższych problemów nie zostały rozwiązane: wszystkie (identyczne) generowane piosenki składały się z identycznych taktów. Zaszło jednak inne, ciekawe zjawisko. Ostatnią funkcją aktywującą *decodera* jest sigmoid, zatem generowane piosenki składają się z nut, każdej o wartości od 0 do 1. Nazwijmy je roboczo „pewnością" nuty. Dla wszystkich pozostałych modeli pewność nut Dla tego modelu jednak pewność każdej z nut była równa albo 0, albo pewna stała (ok. 0.29666), przez co nie da się manipulować ilością nut w utworze. Generowane piosenki albo składają się z żadnych nut, albo bardzo dużej (ale nie losowej) ich ilości. Tę drugą opcję reprezentuje przykładowy utwór.

4.3 Batch normalization

Pogrubieniem zaznaczono fragmenty kodu zmienione względem poprzedniej iteracji.

```
# measure encoder
e = nn.Sequential(
    nn.Linear(4224, 1024),
    nn.ReLU(True),
    nn.Linear(1024, 128),
    nn.ReLU(True),
)
```

```
# measure decoder
d = nn.Sequential(
    nn.BatchNorm1d(128),
    nn.ReLU(True),
    nn.Linear(128, 1024),
    nn.BatchNorm1d(1024),
    nn.ReLU(True),
    nn.Linear(1024, 4224),
    nn.Sigmoid()
)
```

```
# song encoder
E = nn.Sequential(
    nn.Linear(2048, 512),
    nn.ReLU(True),
    nn.Linear(512, 128),
    nn.BatchNorm1d(128),
)
```

```
# song decoder
D = nn.Sequential(
    nn.Linear(128, 512),
    nn.BatchNorm1d(512),
    nn.ReLU(True),
    nn.Linear(512, 2048),
)
```

4017 piosenek, 500 epok	3.1.mp3
4017 piosenek, 2000 epok	3.2.mp3
4017 piosenek, 16000 epok	3.3.mp3

Model, z którymi wiązaliśmy duże nadzieje dał efekty niestety tylko częściowo zachwycające — mają jednak swoje momenty. Dużym plusem jest różnorodność utworów. Ciekawą obserwacją jest rosnąca niespójność utworów między taktami (dwusekundowe fragmenty utworów). Przykładowo, dla 2000 epok praktycznie żaden motyw muzyczny nie jest później powtórzony, chociaż indywidualnie brzmią one nienajgorzej.

4.4 Transpozycja bazy danych

Ostatnia iteracja nie różniła się modelem — edytowaliśmy wyłącznie bazę danych. Dokonaliśmy transpozycji wszystkich utworów do gamy C.

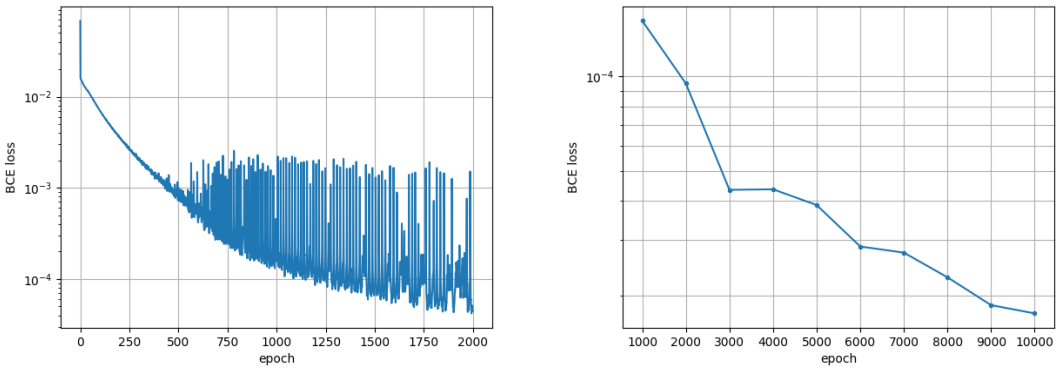
	4.1.mp3
4017 piosenek, 10000 epok	4.2.mp3

Otrzymane utwory są bardziej spójne pod względem tonalności. Efekty są nieco lepsze, da się usłyszeć pewną (acz niewielką) powtarzalność motywów. Dalej jednak zdecydowanie odbiega to od obranego przez nas celu.

5 Ewaluacja wyników

Muzyka jest pojmowana bardzo subiektywne, dlatego jednym z wyznaczników tego, czy wygenerowane piosenki są dobre, była ocenia własna. Subiektywnym wyznacznikiem jakości generowanej przez nas piosenki jest fakt, czy chcielibyśmy grać w grę z taką muzyką. Pod tym względem wyniki, które osiągnęliśmy można nazwać mniej niż zadowalającymi.

Warto jednak w takim razie kontrolować, czy sieć w ogóle poprawnie się uczy. Najprostszym wyznacznikiem jest wykres funkcji straty:



(a) dla każdej epoki w przedziale $[0; 2000]$. (b) co 1000 epok w przedziale $[10^3; 10^4]$.

Rysunek 4: Wykres funkcji straty ostatniej iteracji eksperymentu.

z których można wywnioskować, że sieć uczy się poprawnie. Jako uzasadnienie dla dużej ilości szpicy na wykresie straty znaleźliśmy potencjalne przyczyny: podawanie danych porcjami (batching), używanie ReLU jako funkcji aktywującej. Nie znaleźliśmy jednak definitywnej odpowiedzi.

Jeżeli chodzi o subiektywne cechy muzyczne, wygląda na to, że sieć nauczyła się rozmieszczać nuty rytmicznie (tj. wzdłuż w czasie), ale niekoniecznie harmonicznie (wysokość dźwięków i ich relacje między sobą): nie są powtarzane prawie żadne motywy muzyczne, nie widać żadnego następstwa w progresjach akordowych.

Z naszych obserwacji wysunęliśmy wniosek, że architektura *autoencodera* może mieć duży potencjał w dziedzinie generowania muzyki. Do dojścia do bardziej definitywnych wniosków konieczne są jednak dalsze próby dostrajania architektury sieci neuronowych przybliżających funkcje enkodujące i dekodujące, korekty hiperparametrów, oraz dłuższy czas treningów.