

Załącznik nr 4 do zarządzenia Nr 45
Rektora UMK z dnia 18 kwietnia 2016 r.

Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Łukasz Ogan
nr albumu: 260196

Praca magisterska
na kierunku informatyka

Algorytmiczne metody generowania muzyki

Opiekun pracy dyplomowej
dr Andrzej Kurpiel

Toruń 2018

Spis treści

1 Łańcuchy Markowa	7
1.1 Podstawowe pojęcia i definicje	7
1.1.1 Przestrzeń i funkcja mieralna	7
1.1.2 Zmienna losowa	8
1.2 Cechy Łańcuchów Markowa	9
1.2.1 Pojęcia i własności łańcuchów	15
1.2.2 Zastosowania Łańcuchów Markowa	20
1.2.3 Łańcuchy Markowa w muzyce	21
1.2.4 Przykłady	25
2 Gramatyki	29
2.1 Teoria języków i gramatyk formalnych	29
2.1.1 Języki regularne	30
2.1.2 Języki bezkontekstowe - gramatyki bezkontekstowe	32
2.1.3 Hierarchia Chomsky'ego	33
2.2 Gramatyki i muzyka	34
2.2.1 Przykład wykorzystania gramatyk w muzyce	34
2.2.2 Programy wykorzystujące gramatyki	38
3 Sieci Neuronowe	45
3.1 Wprowadzenie	45
3.2 Model sztucznego neuronu	46
3.3 Metoda spadku gradientowego	49
3.4 Wielowarstwowe sieci neuronowe	51
3.5 Funkcje aktywacji	53
3.5.1 Funkcja liniowa	54
3.5.2 Funkcja sigmoidalna i tanh	55
3.5.3 Funkcja ReLu	56
3.6 Wsteczna propagacja błędu	57
3.7 Rekurencyjne sieci neuronowe	60
3.7.1 Pamięć Long Short-Term	64
3.7.2 Rodzaj sieci "Sequence to sequence"	68
3.7.3 Rodzaj sieci "Character RNN"	69
3.7.4 Metody regularyzacji	70

4 Sieci neuronowe w muzyce	73
4.1 Klasyfikacja narzędzi do tworzenia muzyki	74
4.1.1 Klasyfikacja ze względu na dane wejściowe	75
4.1.2 Klasyfikacja ze względu na architekturę	76
4.1.3 Klasyfikacja ze względu na użyte narzędzia	76
4.1.4 Klasyfikacja ze względu na źródła finansowania	76
4.2 Własny model sieci	76
4.2.1 Konfiguracja maszyny w chmurze	77
4.3 Schemat działania modelu	79
4.3.1 Architektura modelu	84
4.4 Szczegółowy opis programu	84
4.4.1 Użyte narzędzia	87
4.4.2 Wyniki	87
4.4.3 Wnioski	90
Appendices	95
A Elementy muzyki	97
A.1 Wysokość dźwięku	97
A.2 Wartości nut	98
A.3 Metrum	98
A.4 Melodia i harmonia	99
A.5 Transpozycja	99
A.6 Akordy	100
A.7 Progresje	101

Wstęp

O algorytmicznym komponowaniu muzyki myślano na długo przed powstaniem szybkich komputerów. Jako pionierów generowania sztucznej muzyki uważa się Hillera i Isaacsona [1]. Obaj autorzy jako pierwsi w 1957 roku użyli komputera Uniwersytetu w Illinois do wygenerowania kompozycji dla kwartetu smyczkowego. Kolejnym ważnym krokiem w historii był rok 1991, w którym Horner i Goldberg opracowali algorytm genetyczny do generowania muzyki [2]. Wcześniej, w 1981 roku David Cope rozpoczął pracę nad algorytmiczną kompozycją. Połączył łańcuchy Markowa z gramatykami i elementami kombinatoryki. Stworzył pół automatyczny system, który nazwał "Experiments in Musical Intelligence". David Cope w swoich pracach cytuję Xenakisa i Lejarena Hillera jako swoje inspiracje. łańcuchy Markowa mogą jedynie generować podsekwensje, które pochodzą z utworów, na podstawie których łańcuch stworzył macierz przejścia. Inne podejście do problematyki oferują Rekurencyjne Sieci Neuronowe (RNN) wychodzą poza te ograniczenia. W 1989 roku odbyły się pierwsze próby generowania muzyki za pomocą rekurencyjnych sieci neuronowych. Badania zostały opracowane przez Petera M. Dodda, Michaela C. Mozera[3] i kilku innych naukowców. W roku 2002 dokonano przełomu przechodząc ze standardowych komórek rekurencyjnej sieci neuronowej do komórki pamięci długotrwałej - LSTM (*Long Short Term Memory*). Takie podejście wykorzystał Doug Eck do improwizacji bluesa w oparciu o krótkie nagranie[4]. Aktualnie Doug Eck kieruje zespołem projektu Magenta w Google Brain. Projekt Magenta wykorzystuje zastosowania sieci LSTM do generowania fraz perkusyjnych, melodii oraz do generowania muzyki polifonicznej.

Celem pracy magisterskiej jest przeanalizowanie dostępnych metod za pomocą których można wygenerować frazy muzyczne. Analizie zostaną poddane: łańcuchy Markowa, gramatyki oraz sztuczne sieci neuronowe. W każdym przypadku przedstawiono konieczne elementy teorii matematycznych, na których oparte są konkretne przykłady generowania fraz muzycznych.

Celem praktycznym pracy jest prezentacja interesujących przykładów zastosowania powyższych metod wraz ze szczegółową ich analizą oraz krytyką. Głównym celem praktycznym jest stworzenie własnego projektu, który wykorzysta rekurencyjne sieci neuronowe do wygenerowania nowej melodii na podstawie zadanego zbioru danych (zbiór plików MIDI).

Praca złożona jest z czterech rozdziałów. Pierwszy rozdział omawia łańcuchy Markowa z uwzględnieniem ukrytych łańcuchów Markowa. Przedstawione zostały zastosowania łańcuchów oraz praktyczne ich wykorzystanie przy generowaniu muzyki na podstawie analizy plików MIDI. Ukryte łańcuchy Markowa zostały przedstawione w kontekście generowania akordów jazzowych.

Drugi rozdział "Gramatyki" omawia teorię języków i gramatyk formalnych ze szczególnym uwzględnieniem języków regularnych i bezkontekstowych. W rozdziale tym przedstawiono szereg aplikacji, które wykorzystują gramatyki bezkontekstowe przy generowaniu prostych melodii.

Rozdział trzeci "Sieci neuronowe" zawiera szczegółowe omówienie wielu elementów teorii sztucznych sieci neuronowych. W rozdziale zostały przedstawione najważniejsze rodzaje sieci neuronowych. Opisane zostały zasady działania rekurencyjnych sieci neuronowych oraz sieci *Long Short Term Memory*.

Rozdział czwarty zawiera praktyczne przykłady wykorzystania sztucznych sieci neuronowych na polu generowania muzyki. Są to w większości rozbudowane projekty, które służą do generowania muzyki. Rozdział ten opisuje własny projekt praktyczny. Projekt zawiera konstrukcje modelu rozbudowanej sieci neuronowej, której celem jest wygenerowanie melodii na podstawie zadanego zbioru danych.

Do pracy pracy dołączono DVD-ROM, na którym znajduje się:

- Źródła pracy w języku LaTeX
- Plik PDF pracy
- Kody źródłowe do poszczególnych rozdziałów
- Pliki MIDI

Rozdział 1

Łańcuchy Markowa

1.1 Podstawowe pojęcia i definicje

W tym podrozdziale dokonamy przeglądu kilku podstawowych definicji z teorii prawdopodobieństwa, które będą potrzebne przy rozważaniach opisanych w kolejnych częściach rozdziału pierwszego.

1.1.1 Przestrzeń i funkcja mierzalna

Definicja 1.1.1. σ -algebrą podzbiorów zbioru Ω nazywamy rodzinę $\mathcal{F} \subset 2^\Omega$ spełniającą następujące warunki.

- $\emptyset, \Omega \in \mathcal{F}$.
- Jeżeli $\mathcal{A} \in \mathcal{F}$, to również $\mathcal{A}^C \in \mathcal{F}$.
- Jeżeli $\mathcal{A}_1, \mathcal{A}_2, \dots \in \mathcal{F}$, to $\bigcap_{j=1}^{\infty} A_j \in \mathcal{F}$

Przykład 1.1.1. Przykłady σ -algebr:

- Rodzina wszystkich podzbiorów zbioru Ω tworzy σ -algebrę: $\mathcal{F} = 2^\Omega$,
- Rodzina \mathcal{F} złożona z $\{\emptyset, \Omega\}$ tworzy σ -algebrę,
- Niech $\mathcal{R} = \{A_1, A_2, \dots, A_n\}$ skończone rozbicie przestrzeni Ω , tj. $\Omega = \bigcup_{j=1}^n A_i$ i zbiory A_j są parami rozłączne: $A_i \cap A_j = \emptyset$, jeśli $i \neq j$. Wtedy \mathcal{F} złożona z sum rozłącznych elementów rozbicia \mathcal{R} jest σ -algebrą.

Definicja 1.1.2. Przestrzenią mierzalną nazywamy parę (Ω, \mathcal{F}) , gdzie Ω jest niepustym zbiorem, a \mathcal{F} jest σ -algebrą podzbiorów zbioru Ω .

Przykład 1.1.2. Niech Ω będzie niepustym zbiorem. Następujące pary (Ω, \mathcal{F}) stanowią przykłady przestrzeni mierzalnych:

- $(\Omega, 2^\Omega)$ - tworzy σ -algebrę
- $(\Omega, \{\emptyset, \Omega\})$ - tworzy σ -algebrę

- $(\Omega, \{\emptyset, \Omega, A, \Omega \setminus A\})$ dla dowolnego $A \subseteq \Omega$ - tworzy σ -algebrę

Przykład 1.1.3. Rodzina wszystkich podzbiorów zbioru Ω jest σ -algebrą w Ω

Symbolom \mathbb{R}^1 oznaczmy zbiór liczb rzeczywistych dodatnich.

Definicja 1.1.3. Funkcję $f : (\Omega, \mathcal{F}) \rightarrow \mathbb{R}^1$ nazywamy mierzalną, jeśli dla każdego $a \in \mathbb{R}^1$

$$\{f \leq a\} = \{\omega; f(\omega) \leq a\} \in \mathcal{F}.$$

Definicja 1.1.4. Podzbiory borełowskie \mathbb{R}^1 to elementy σ -algebry generowanej przez podzbiory otwarte (równoważnie: domknięte) zbioru \mathbb{R}^1 . σ -algebrę zbiorów borełowskich oznaczamy symbolem \mathcal{B}^1 .

Przykład 1.1.4. Z definicji wiemy, że każdy zbiór otwarty jest zbiorem borełowskim, a także zbiór domknięty, jako uzupełnienie zbioru otwartego. Zbiorami borełowskimi są również zbiory 1-punktowe. Zbiór liczb wymiernych $Q \subset R$ jest borełowski jako przeliczalna suma zbiorów 1-punktowych, zatem liczby niewymiernie $R \setminus Q$ także stanowią zbiór borełowski.

1.1.2 Zmienna losowa

Definicja 1.1.5. Przestrzenią probabilistyczną nazywamy trójkę $(\Omega, \mathcal{F}, \mathcal{P})$, gdzie

- Ω jest zbiorem "zdarzeń elementarnych" (elementy ω zbioru Ω nazywamy *zdarzeniami elementarnymi*).
- \mathcal{F} jest σ -algebrą podzbiorów zbioru Ω . Elementy \mathcal{F} nazywane są *zdarzeniami*.
- $P : \mathcal{F} \rightarrow [0, 1]$ jest prawdopodobieństwem na (Ω, \mathcal{F}) .

Przykład 1.1.5. Aby obliczyć szansę dowolnego zdarzenia A trzeba określić liczbę zdarzeń sprzyjających oraz liczbę wszystkich możliwych zdarzeń. Do obliczenia prawdopodobieństwa korzysta się z wzoru:

$$P(A) = \frac{\#A}{\#\Omega} = \frac{\text{ilosc elementow w zbiorze } A}{\text{ilosc elementow w zbiorze } \Omega}$$

gdzie:

- $\#A$ to liczba zdarzeń sprzyjających
- $\#\Omega$ to liczba wszystkich możliwych zdarzeń

Rozważmy przykład, w którym można zastosować powyższe rozważania. Należy obliczyć prawdopodobieństwo, że w rzucie kostką wypadnie liczba oczek mniejsza od 5.

Dane:

- zdarzeniem losowym jest rzut kostką
- Ω - zbiór wszystkich możliwych zdarzeń. $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- A - zbiór wyników, których liczba oczek jest mniejsza od 5. $A = \{1, 2, 3, 4\}$.
- $\#\Omega = 6$ - zbiór Ω liczy 6 elementów
- $\#A = 4$ - zbiór A liczy 4 elementy

Zatem prawdopodobieństwo zdarzenia A jest liczone według wzoru:

$$P(A) = \frac{A}{\Omega} = \frac{4}{6} = \frac{2}{3}$$

Definicja 1.1.6. Zmienną losową na przestrzeni $(\Omega, \mathcal{F}, \mathcal{P})$ nazywamy funkcję $X : (\Omega, \mathcal{F}) \rightarrow \mathbb{R}^1$ o własności

$$X^{-1}(-\infty, u]) \in \mathcal{F}, u \in \mathbb{R}^1.$$

Przykład 1.1.6. Niech Ω będzie zbiorem wszystkich możliwych wyników przy pojedynczym rzucie dwiema kostkami do gry. Zbiór Ω składa się z 36 możliwych wyników i oznaczmy go: $\Omega = \{(i, j); 1 \leq i, j \leq 6\}$. Rozważmy funkcję $X : \Omega \rightarrow \mathbb{R}^1$. Określona wzorem:

$$\forall_{(i,j) \in \Omega} X(i, j) = i + j$$

Symbol \mathcal{F} oznaczmy wszystkie podzbiory zbioru Ω . Wówczas trójka $(\Omega, \mathcal{F}, \mathcal{P})$ gdzie $P : \mathcal{F} \rightarrow [0, 1]$ określona wzorem:

$$P(A) = \frac{\#A}{36}$$

jest przestrzenią probabilistyczną, a funkcja X jest zmienną losową.

Przykład 1.1.7. Rozważmy przestrzeń $(\Omega, \mathcal{F}, \mathcal{P})$ w której:

- Ω - jest niepustym zbiorem,
- $P : \mathcal{F} \rightarrow [0, 1]$ jest prawdopodobieństwem na (Ω, \mathcal{F})
- $P(\emptyset) = 0$
- $P(\Omega) = 1$
- $P(A) = p$ dla $0 \leq p \leq 1$

Wówczas:

- $P(\Omega \setminus A) = 1 - p$

1.2 Cechy Łańcuchów Markowa

Procesem Markowa to ciąg zmiennych losowych (ciąg zdarzeń), w którym prawdopodobieństwo każdego zdarzenia zależy jedynie od wyniku poprzedniego. Procesem stochastycznym nazywamy proces, w którym teoria prawdopodobieństwa używana jest do modelowania losowych zjawisk. Łańcuch Markowa to proces Markowa, który zdefiniowany jest na dyskretnej przestrzeni stanów, jest to jeden z procesów stochastycznych.

Definicja 1.2.1. Łańcuchem Markowa o zbiorze stanów $S \subseteq \mathbb{R}$ nazywamy ciąg zmiennych losowych X_0, X_1, X_2, \dots taki, że:

$$P(X_n = x_n | X_{n-1} = x_{n-1} \wedge X_{n-2} = x_{n-2} \wedge \dots \wedge X_0 = x_0) = P(X_n = x_n | X_{n-1} = x_{n-1}) = p_{x_{n-1}, x_n}$$

dla każdego $n > 0$ i ciągu stanów $x_0, x_1, \dots, x_n \in S$.

Dziedzina zmiennych losowych nazywana jest przestrzenią stanów. Pojedynczy stan opisywany jest przez X_t w chwili t . Stany w Łańcuchu Markowa wzajemnie od siebie zależą, możemy powiedzieć, że zmienna X_t zależy tylko od zmiennej X_{t-1} . Nie oznacza to, że X_t jest niezależna od X_0, X_1, \dots, x_{t-2}

Definicja 1.2.2. Macierzą przejścia w jednym kroku łańcucha Markowa nazywamy macierz $M = (p_{i,j})_{i,j \in S}$. Wiersze macierzy przejścia łańcucha Markowa sumują się do 1, czyli $\sum_j p_{i,j} = 1$.

Liczby $p_{i,j}$, które występują w definicji łańcucha Markowa to prawdopodobieństwa przejścia w jednym kroku. Jeżeli w danej chwili łańcuch Markowa jest w stanie i , to z prawdopodobieństwem $p_{i,j}$ znajdzie się w następnej chwili w stanie j .

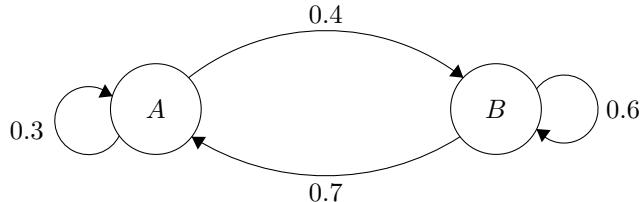
$$M = \begin{bmatrix} p_{0,0}(t) & p_{0,1}(t) & p_{0,2}(t) & \dots & p_{0,j}(t) \\ p_{1,0}(t) & p_{1,1}(t) & p_{1,2}(t) & \dots & p_{1,j}(t) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{i,j}(t) & x_{i,j}(t) & p_{i,j}(t) & \dots & p_{i,j}(t) \end{bmatrix}$$

Macierz przejścia

Przykład 1.2.1. Dany jest zbiór stanów $S = \{A, B\}$, w tej przestrzeni stanów można zdefiniować 4 możliwe przejścia. Macierz przejścia M wygląda następująco:

$$M = \begin{bmatrix} 0.3 & 0.7 \\ 0.4 & 0.6 \end{bmatrix}$$

Liczba 0.4 z pierwszej kolumny drugiego wiersza mówi, że istnieje czterdziesto procentowe prawdopodobieństwo, że ze stanu A dojdziemy do stanu B . Następnie, że stanu B istnieje sześćdziesięcio procentowe prawdopodobieństwo, że zostanie osiągnięty z powrotem stan B . Macierzy M odpowiada automat z przejściami przedstawiony poniżej.



Inną równoważną definicję jest trójka $\langle Q, \pi, p \rangle$, którą definiuje się w sposób następujący:

Definicja 1.2.3. łańcuchem Markowa nazywamy trójkę $\langle Q, \pi, p \rangle$, gdzie:

- Q jest przestrzenią stanów,
- $p : Q \times Q \rightarrow [0, 1]$ - prawdopodobieństwo osiągnięcia stanu w momencie przejścia,
- $\pi : Q \rightarrow [0, 1]$ - prawdopodobieństwo osiągnięcia stanu inicjalizującego.

Ponadto muszą zachodzić warunki:

- dla $q_i \in Q$, $\sum_{q_j \in Q} a(q_i, q_j) = 1$,
- $\sum_{q \in Q} \pi(q) = 1$.

Przykład 1.2.2. Student raz w tygodni bierze udział w zajęciach z rachunku prawdopodobieństwa. Na każde zajęcia przychodzi przygotowany bądź nie. Jeśli w danym tygodniu jest przygotowany, to w następnym jest przygotowany z prawdopodobieństwem 0.7. Jeśli natomiast w danym tygodniu nie jest przygotowany, to w następnym jest przygotowany z prawdopodobieństwem 0.2. Interesujące są pytania na odpowiedzi:

1. Jeśli student jest w tym tygodniu nieprzygotowany, to ile tygodni musimy średnio czekać aż będzie przygotowany?
2. Na dłuższą metę, jak często student jest przygotowany?

Wyróżniamy tutaj dwa stany: 1 - student jest przygotowany, oraz 2 - student nie jest przygotowany. Na podstawie danych można skonstruować macierz przejścia:

$$M = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix}$$

Dane: zbiór stanów $S \subseteq \mathcal{R}$, $S = \{0, 1\}$. Prawdopodobieństwa zdarzeń przedstawiają się w sposób następujący:

$$\begin{aligned} P(X_{t_2} = 0 | X_{t_2} = 1) &= 0.3 = p_{1,0} \\ P(X_{t_2} = 1 | X_{t_2} = 1) &= 0.7 = p_{1,1} \\ P(X_{t_2} = 1 | X_{t_2} = 0) &= 0.2 = p_{0,1} \\ P(X_{t_2} = 0 | X_{t_2} = 0) &= 0.8 = p_{0,0} \end{aligned}$$

Wiersze macierzy M zgodnie z definicją macierzy przejścia sumują się do 1. Przypuśćmy, że znany jest rozkład zmiennej X_t czyli prawdopodobieństwo tego, że w chwili t znajdujemy się w poszczególnych stanach. Trzeba znaleźć rozkład zmiennej X_{t+1} oraz później rozkład zmiennej X_{t+s} . Taki rozkład można łatwo znaleźć korzystając z macierzy M .

Ze wzoru na prawdopodobieństwo całkowite wynika:

$$P(X_{t+1} = a) = \sum_{b \in S} P(X_t = b)P(X_{t+1} = a | X_t = b) = \sum_{b \in S} \pi(t)_b M_{b,a}$$

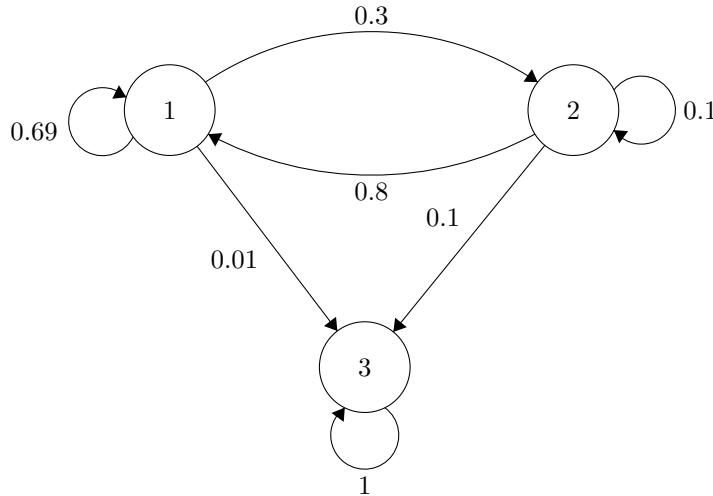
Mówiąc prościej, mamy tutaj do czynienia z mnożeniem macierzy przez wektor, co można zapisać: $\pi(t) \cdot M = \pi(t+1)$

Przykład 1.2.3. Firmy ubezpieczeniowe wykorzystują łańcuchy Markowa do obliczenia ile rzeczywiście pieniędzy pobierają od swoich klientów. Przykładowym modelem, na którym można to zbadać jest model podsumowujący stan zdrowia człowieka w cyklu miesięcznym.

Dane: zbiór stanów $S \subseteq \mathcal{R}$, $S = \{1, 2, 3\}$. gdzie 1 - oznacza osobę zdrową, 2 - oznacza osobę chorą, 3 - oznacza śmierć. Prawdopodobieństwa zdarzeń przedstawiają się w następujący sposób.

$$\begin{aligned} P(X_n = 1 | X_n = 2) &= 0.3 = p_{2,1} \\ P(X_n = 1 | X_n = 1) &= 0.6 = p_{1,1} \\ P(X_n = 2 | X_n = 1) &= 0.8 = p_{1,2} \\ P(X_n = 2 | X_n = 2) &= 0.1 = p_{2,2} \end{aligned}$$

Prawdopodobieństwa można zobrazować w postaci automatu z przejściami:

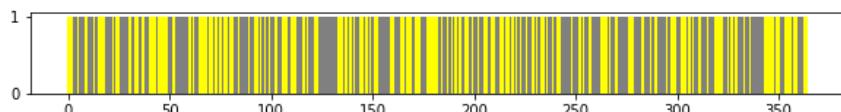


Brakujące prawdopodobieństwa zostały wyliczone biorąc pod uwagę własność macierzy przejścia, w której wiersze sumują się do 1.

$p_{1,1} = 1 - p_{1,2} - p_{1,3}$ oraz $p_{2,2} = 1 - p_{2,1} - p_{2,3}$. Na podstawie powyższego łańcuchy Markowa można próbować odpowiedzieć na pytania:

- Jaka jest długość życia obecnie zdrowej osoby?
- Jaka jest długość życia obecnie chorej osoby?

Przykład 1.2.4. Niech dane będą dwie zmienne losowe X_0, X_1 , gdzie X_0 oznaczać będzie pogodę słoneczną, a X_1 oznaczać pogodę pochmurną. Dla ułatwienia wprowadźmy oznaczenia: $X_0 - S$, $X_1 - C$. Następnie przypiszmy prawdopodobieństwo wystąpienia pogody słonecznej równe $\frac{1}{2}$ oraz prawdopodobieństwo wystąpienia pogody pochmurnej równe $\frac{1}{2}$. Następnym krokiem będzie wygenerowanie losowej pogody na 356, gdzie taki rozkład można zaprezentować za pomocą symboli np. $S, S, C, C, S, S, S, C, S, C, S, C, \dots, n$. Mając rozkład w postaci symboli można go zobrazować w postaci wykresu.

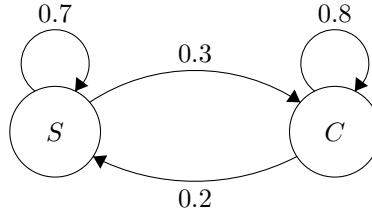


Rysunek 1.1: Wykres prawdopodobieństwa

Powyższa ilustracja obrazuje działanie modelu, pogoda każdego dnia jest niezależna od poprzednich dni, pobierana jest również z tego samego rozkładu. W kolejnym kroku wykorzystamy łańcuch Markowa i macierz przejścia aby uzależnić dany stan od poprzedniego. łańcuch Markowa rozpocznie się w określonym stanie, a następnie pozostanie w tym samym stanie lub przejdzie do innego stanu na podstawie macierzy prawdopodobieństw:

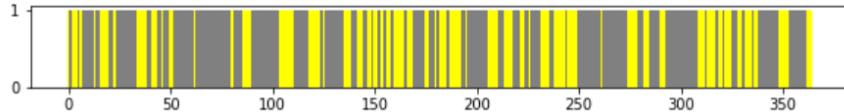
$$P = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix}$$

Z macierzy możemy odczytać, że prawdopodobieństwo zajścia dnia słonecznego pod warunkiem, że po-predniego dnia był słoneczny dzień wynosi 0.7



Rysunek 1.2: Graf prawdopodobieństw równoważny z macierzą przejścia

Należy stworzyć zmienną losową dla każdego rzędu macierzy przejścia, zmienna losowa odpowiada prawdopodobieństwom przejść od stanu do stanu.



Rysunek 1.3: Wykres prawdopodobieństwa po zastosowaniu łańcucha Markowa

Jak widać na ilustracji powyżej, po zastosowaniu łańcucha Markowa z zadaną macierzą przejścia wykres wygląda trochę inaczej. Za pomocą macierzy jesteśmy w stanie kontrolować długości danego stanu. Poniżej znajduje się pełen listing kodu programu, który realizuje powyższe rozważania.

```

1 # coding: utf-8
2
3 import matplotlib.pyplot as plt
4 from scipy import stats
5 import numpy as np
6
7 class RandomVar:
8     def __init__(self, name, values, probability):
9         self.name = name
10        self.values = values
11        self.probability = probability
12        if all(type(item) is np.int64 for item in values):
13            self.var_type = 1
14            self.rv = stats.rv_discrete(name=name, values=(values, probability))
15        elif all(type(item) is str for item in values):
16            self.var_type = 2
17            self.rv = stats.rv_discrete(name=name, values=(np.arange(len(values)), probability))
18        self.symbolic_values = values
19        print self.rv
20
21    def sample(self, size):
22        numeric = self.rv.rvs(size=size)
  
```

```

24     mapped = [ self.values[x] for x in numeric]
25     return mapped
26
27 def prob(self):
28     return self.probability
29
30 def vals(self):
31     print self.type
32     return self.values
33
34
35 values = [ 'S' , 'C' ]
36 probabilities = [0.5 , 0.5]
37 weather = RandomVar( 'weater' , values , probabilities )
38
39 samples = weather.sample(365)
40
41 state2color = {}
42 state2color[ 'S' ] = 'yellow'
43 state2color[ 'C' ] = 'grey'
44
45
46 def plot(samples , state2color):
47     colors = [state2color[x] for x in samples]
48     x = np.arange(0 , len(colors))
49     y = np.ones(len(colors))
50     plt.figure(figsize=(10, 1))
51     plt.bar(x , y , color=colors , width=1)
52     plt.show()
53
54
55 samples = weather.sample(365)
56 plot(samples , state2color)
57
58 def markov_chain(transition_matrix , state , state_names , samples):
59     (row , cols) = transition_matrix.shape
60     rvs = []
61     values = list(np.arange(0 , row))
62     print "Values {}".format(values)
63
64     # stworz losowa wartosc dla kazdego wiersza macierzy przejscia
65     for r in xrange(row):
66         rv = RandomVar("row" + str(r) , values , transition_matrix[r])
67         print "rv {} r {}".format(rv , transition_matrix[r])
68         rvs.append(rv)
69
70     # zacznij ze stanu poczatkowego
71     states = []
72     for n in range(samples):
73         state = rvs[state].sample(1)[0]
74         states.append(state_names[state])
75     return states
76
77 transition_matrix = np.array([[0.7 , 0.3] ,
78                             [0.2 , 0.8]])
79 samples = weather.sample(365)
80 plot(samples , state2color)
81 samples_markov = markov_chain(transition_matrix , 0 , [ 'S' , 'C' ] , 365)

```

```
82 plot(samples_markov, state2color)
```

1.2.1 Pojęcia i własności łańcuchów

W teorii prawdopodobieństwa oraz statystyce rozważa się inne modele matematyczne, które bazują na probabilistycznych i są trochę bardziej rozbudowane niż klasyczny łańcuch Markowa. Należy tutaj wymienić takie modele jak: łańcuchy Markowa wyższego rzędu, ukryte modele Markowa, łańcuchy Markowa mieszanego rzędu, łańcuchy Markowa wyższego rzędu, probabilistyczne skończone automaty. Poniżej zostaną przedstawione niektóre z nich.

Rozpoczniemy od opisu ukrytych modeli Markowa. Ukryte modele Markowa (HMM, ang. hidden Markov models) są rozszerzeniem standardowej definicji łańcucha Markowa. Używane są głównie do modelowania związku między ukrytymi, a obserwowanymi sekwencjami. Ukrytym modelem Markowa jest łańcuch Markowa z dyskretnym rozkładem prawdopodobieństwa w każdym stanie. Dyskretne rozkłady prawdopodobieństwa definiują prawdopodobieństwa emisji określonego symbolu alfabetu w danym stanie.

Definicja 1.2.4. Ukrytym modelem Markowa nazywamy piątkę $\langle \sum, Q, a, b, \pi \rangle$ gdzie:

- \sum jest skończonym alfabetem widocznych symboli,
- Q jest skończonym zbiorem ukrytych stanów,
- $a : Q \times Q \rightarrow [0, 1]$ - prawdopodobieństwo osiągnięcia stanu pomiędzy stanami ukrytymi,
- $b : Q \times \sum \rightarrow [0, 1]$ - prawdopodobieństwo emisji określonego symbolu alfabetu w danym ukrytym stanie,
- $\pi : Q \rightarrow [0, 1]$ - początkowe prawdopodobieństwo ukrytych stanów.

Ponadto muszą zachodzić warunki:

- dla $q_i \in Q$, $\sum_{q_j \in Q} a(q_i, q_j) = 1$,
- $\sum_{q \in Q} \pi(q) = 1$.

Oznaczmy sekwencję obserwacji i stanu ukrytego przez $X = X_1, \dots, X_n$ i $s = s_1, \dots, s_n$, gdzie $X_i \in \sum$ oraz $s_i \in Q$. W tym przypadku używamy następującej notacji:

$$\begin{aligned}\pi(q_i) &= P(s_1 = q_i) \\ a(q_i, q_j) &= P(s_t = q_j | s_{t-1} = q_i) \\ b(q_j, X_t) &= P(X_t | s_t = q_j)\end{aligned}$$

$P(x)$ jest prawdopodobieństwem zdarzenia, natomiast $P(x|y)$ jest prawdopodobieństwem zdarzenia x pod warunkiem y .

Przykład 1.2.5. Klasycznym przykładem ukrytego łańcucha Markowa jest nieuczciwe kasyno do gry, które ma dwa rodzaje kości: uczciwą kostkę do gry (z prawdopodobieństwem $\frac{1}{6}$ wyrzuca się każdą z sześciu możliwych wartości) oraz nieuczciwą kostkę (dla której prawdopodobieństwo wyrzucenia szóstki wynosi $\frac{1}{2}$, a dla pozostałych liczb $\frac{1}{10}$). Mamy do dyspozycji dwa stany: F (uczciwa kostka) i L (nieuczciwa kostka). Układ może zmieniać swój stan z pewnym prawdopodobieństwem, ale my stanu nie możemy zaobserwować (np. krupier zmienia kostki pod stołem). Jedynie widzimy ciąg liczb będących wynikiem rzutów kostką. To, którą

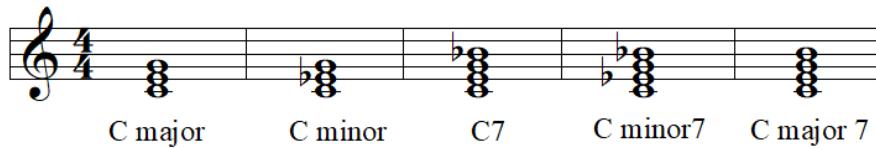
kostką rzucamy zależy tylko i wyłącznie od stanu ukrytego łańcucha Markowa. Macierz przejścia zdefiniujemy następująco:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.45 & 0.55 \end{bmatrix}$$

gdzie $p_{F,F} = 0.90$, $p_{L,L} = 0.55$, $p_{F,L} = 0.10$, $p_{L,F} = 0.45$. Średni czas nieprzerwanego rzucania kostką uczciwą jest równy $\frac{1}{1-0.9} = 10$ okresów, a kostką fałszywą $\frac{1}{0.55} = 2$. Chodź nie wiemy, którą kostką rzucamy to mamy o tym jednak jakąś informacje. Obserwujemy bowiem ciąg liczb będących wynikiem rzutów kostką. Macierz emisji dla tego przykładu wygląda następująco:

$$\Pi = \begin{bmatrix} \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{2} \end{bmatrix}$$

Przykład 1.2.6. Ukryty łańcuch Markowa można również wykorzystać w muzyce do generowania akordów na podstawie progresji II V I. Progresja II V I jako następstwo akordów, jest wykorzystywana w szerokim zakresie gatunków muzycznych oraz jest podstawą harmonii jazzowej. W przykładzie będziemy rozważać progresje II V I dla akordów tonacji C-dur. Poszczególne akordy będą stanami w łańcuchu Markowa. Stany ukryte będą odpowiadające za typ akordu (minor7, major7, dominant7).



Rysunek 1.4: Podstawowe rodzaje akordów dla gamy C-dur

W pierwszym kroku należy zdefiniować macierz przejścia dla stanów łańcucha Markowa. W przykładzie będą wykorzystane trzy stany, które będą odpowiadały nutą D, G, C - progresja II V I.

$$P = \begin{bmatrix} 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \\ 0.0 & 0.3 & 0.7 \end{bmatrix}$$

Następnie musi zostać stworzona macierz emisji dla stanów ukrytych. Warstwa ukryta będzie odpowiadająca za rodzaj akordu. Do dyspozycji będą trzy rodzaje akordów: minor7, major7, dominant7.

$$M = \begin{bmatrix} 0.4 & 0.0 & 0.4 \\ 0.3 & 0.3 & 0.3 \\ 0.2 & 0.8 & 0.0 \end{bmatrix}$$

Dla danego stanu z macierzy P będzie wybierany dany stan z macierzy M z uwzględnieniem reguł łańcucha Markowa. Praktycznie zostanie to zrealizowane z użyciem języka Python. Pakiet o nazwie *hmmlearn* zwolni nas z implementowania logiki Ukrytego Modelu Markowa, a za pomocą pakietu *music21* będzie możliwa wygenerować akordy na pięciolinii.

```

1 import numpy as np
2 from hmmlearn import hmm
3 from music21 import *

```

```

4
5 environment.set("musescoreDirectPNGPath",      "/usr/bin/musescore")
6 environment.set("musicxmlPath",                "/usr/bin/musescore")
7 environment.set("midiPath",                   "/usr/bin/lilypond")
8
9 conv = converter.subConverters.ConverterLilypond()
10
11
12 transmat = np.array([[0.4,  0.4,  0.2],
13                      [0.1,  0.1,  0.8],
14                      [0.0,  0.3,  0.7]])
15
16 start_prob = np.array([1.0,  0.0,  0.0])
17
18 emission_probs = np.array([[0.4,  0.0,  0.4],
19                            [0.3,  0.3,  0.3],
20                            [0.2,  0.8,  0.0]])
21
22 chord_model = hmm.MultinomialHMM(n_components=2)
23
24 chord_model.startprob_ = start_prob
25
26 chord_model.transmat_ = transmat
27 chord_model.emissionprob_ = emission_probs
28 X, Z = chord_model.sample(10)
29
30 state2name = {}
31 state2name[0] = 'D'
32 state2name[1] = 'G'
33 state2name[2] = 'C'
34 chords = [state2name[state] for state in Z]
35
36 obj2name = {}
37 obj2name[0] = 'min7'
38 obj2name[1] = 'maj7'
39 obj2name[2] = '7'
40
41 observations = [obj2name[item] for sublist in X for item in sublist]
42 chords = [','.join(chord) for chord in zip(chords, observations)]
43
44
45 # create some chords for II, V, I
46 d7 = chord.Chord(['D4', 'F4', 'A4', 'C5'])
47 dmin7 = chord.Chord(['D4', 'F-4', 'A4', 'C5'])
48 dmaj7 = chord.Chord(['D4', 'F#4', 'A4', 'C#5'])
49
50 c7 = d7.transpose(-2)
51 cmin7 = dmin7.transpose(-2)
52 cmaj7 = dmaj7.transpose(-2)
53
54 g7 = d7.transpose(5)
55 gmin7 = dmin7.transpose(5)
56 gmaj7 = dmaj7.transpose(5)
57 print(g7.pitches)
58
59 stream1 = stream.Stream()
60 stream1.repeatAppend(dmin7, 1)
61 stream1.repeatAppend(g7, 1)

```

```

62 stream1.repeatAppend(cmaj7, 1)
63 stream1.repeatAppend(cmaj7, 1)
64 print(stream1)
65
66 name2chord = {}
67 name2chord[ 'C7' ] = c7
68 name2chord[ 'Cmin7' ] = cmin7
69 name2chord[ 'Cmaj7' ] = cmaj7
70
71 name2chord[ 'D7' ] = d7
72 name2chord[ 'Dmin7' ] = dmin7
73 name2chord[ 'Dmaj7' ] = dmaj7
74
75 name2chord[ 'G7' ] = g7
76 name2chord[ 'Gmin7' ] = gmin7
77 name2chord[ 'Gmaj7' ] = gmaj7
78 hmm_chords = stream.Stream()
79
80 for c in chords:
81     hmm_chords.repeatAppend(name2chord[ c ], 1)
82
83 hmm_chords.show()

```

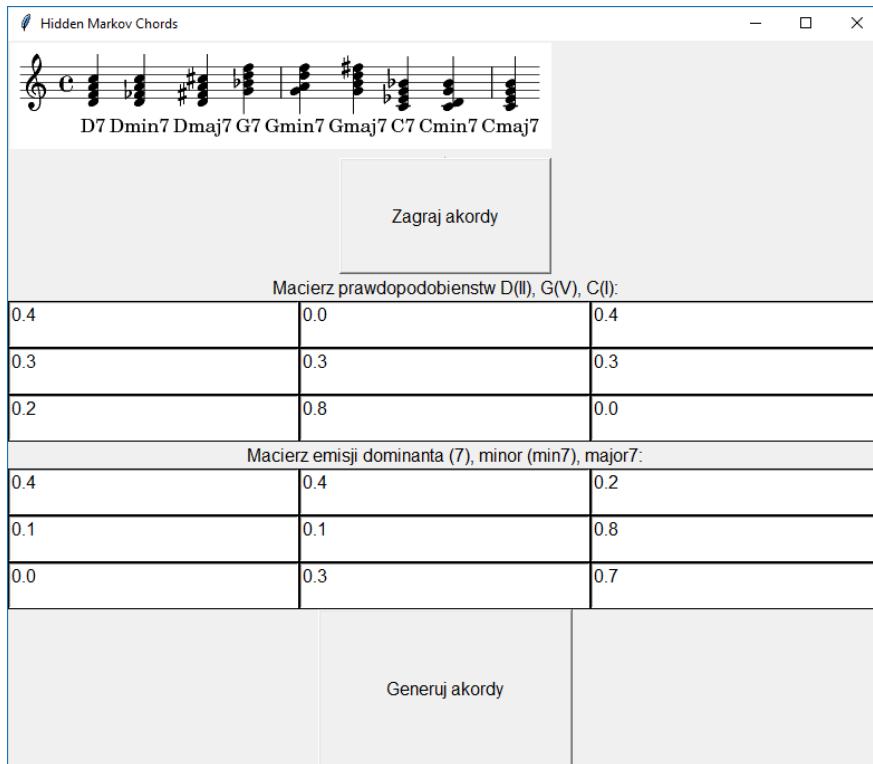
Listing 1.1: Ukryty Model Markowa z użyciem Pythona

W przykładzie użyto 10 iteracji. Wynikiem działania programu jest sekwencja akordów przedstawiona poniżej:

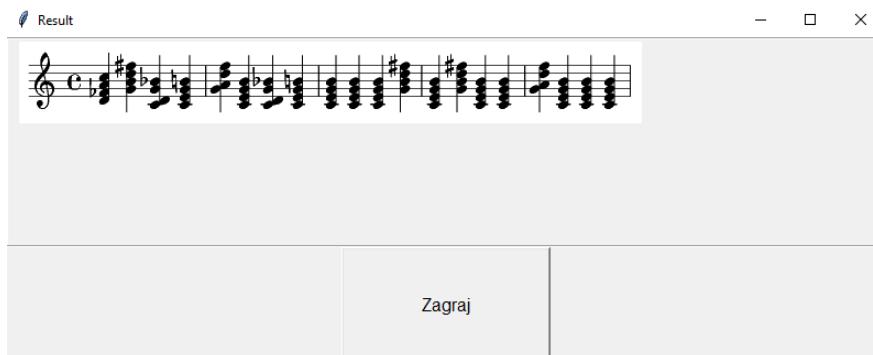


Rysunek 1.5: Wygenerowane akordy z użyciem ukrytego modelu Markowa

Powyższy kod został wzbogacony o interfejs użytkownika, w którym można definiować wartości macierzy prawdopodobieństw i emisji.



Rysunek 1.6: Interfejs aplikacji, która wykorzystuje ukryte modele Markowa do generowania akordów



Rysunek 1.7: Wygenerowane akordy na podstawie zadanych macierzy

W programie wykorzystano bibliotekę music21, która posłużyła do zapisu nutowego oraz zapisu nut do pliku. Algorytm ukrytego modelu Markowa pochodzi z biblioteki hmmlearn.

Możemy powiedzieć, że standardowe Łańcuchy Markowa mają pamięć długości 1 ponieważ n-ty stan zależy tylko i wyłącznie od stanu poprzedniego. Przeciwieństwo stanowią Łańcuchy Markowa wyższego rzędu (ang. Higher Order Markov Chains), które mają pamięć większą niż jeden.

Definicja 1.2.5. Łańcuch Markowa N-tego rzędu definiujemy w następujący sposób:

$$P(q_t|q_{t-1}, q_{t-2}, \dots, q_1) = P(q_t|q_{t-1}, \dots, q_{t-\min(t-1, N)})$$

Jeżeli $t \leq N$ to q_t jest stanem startowym oraz nadmierne stany q_t gdzie $t < 0$ są reprezentowane przez puste słowo λ . W przypadku Łańcucha Markowa N-tego rzędu będziemy używać zapisu $a(q_{t-N}, \dots, q_{t-1}, q_t)$ zamiast $a(q_{t-1}, q_t)$ aby wskazać prawdopodobieństwa przejścia.

1.2.2 Zastosowania Łańcuchów Markowa

Łańcuchy Markowa mają bardzo szerokie zastosowanie, możemy tutaj wskazać takie dyscypliny nauki jak: fizyka, genetyka, meteorologia, gospodarka. Przydatność łanuchów Markowa uwidacznia się w przypadku, gdy nie można przyjąć założenia o niezależności zdarzeń i zmiennych losowych. Do zalet prognozowania na podstawie łanuchów Markowa można zaliczyć:

- możliwość predykcji w przypadku, gdy nie są znane przyczyny występowania badanego zjawiska lub gdy jest ich zbyt wiele,
- możliwość konstruowania prognoz dla zjawisk mierzalnych i niemierzalnych,
- możliwość budowy prognoz krótko, średnio, oraz długoterminowych,
- możliwość prognozowania strukturalnych zjawisk ekonomicznych o wzajemnie zależnych w czasie elementach składowych

Jednym z ciekawszych zastosowań łanuchów Markowa jest użycie ich w popularnym algorytmie wyszukiwarki **Google - PageRank**. Algorytm można interpretować jako znajdowanie ustalonego stanu z łanucha Markowa. W tym przypadku łanuch Markowa jest modelem procesu poruszania się użytkownika po zbiorze wszystkich stron www. Każda strona jest stanem, a powiązania między stronami są prawdopodobieństwami. Niezależnie od tego na jakiej stronie się znajdujemy szansa na znalezienie się na innej stronie X określone jest stałym prawdopodobieństwem. Formalnie: jeżeli N jest liczbą wszystkich znanych stron internetowych oraz strona i posiada k_i linków prowadzących do niej wówczas można ustalić prawdopodobieństwo przejścia $\frac{\alpha}{k_i} + \frac{1-\alpha}{N}$ dla wszystkich stron, które są połączone z daną stroną i $\frac{1-\alpha}{N}$, które nie są połączone, gdzie α jest stałą równą 0.85[5]. Łanuch Markowa pozwala również na analizę zachowania użytkownika podczas nawigacji na stronie internetowej. Na podstawie takiej analizy można np. spersonalizować nawigację pod danego użytkownika.

Większość popularnych klawiatur na systemy Android umożliwia szybsze pisanie wiadomości poprzez **podpowiadanie następnych słów** na podstawie poprzednich. Słowa, które piszemy są analizowane i włączane do prawdopodobieństwa w łanuchu Markowa. Czasami aplikacje proszą o możliwość dostępu np. do emaili aby na ich podstawie zbudować bazę prawdopodobieństw. Mając bazę prawdopodobieństw, czyli w tym wypadku łanuch Markowa, aplikacje wykorzystują model probabilistyczny n-gram, który jest wykorzystywany do pobierania kolejnego elementu z sekwencji łanucha Markowa.

Ukryte modele Markowa mają swoje zastosowanie w systemach do **automatycznego rozpoznawania mowy**. Za ich pomocą tworzy się modele akustyczne, które odpowiadają danemu wyrazowi. Najbardziej popularnym algorytmem, który jest wykorzystywany przy rozpoznawaniu mowy jest algorytm Bauma-Welcha. Systemy do rozpoznawania mowy możemy podzielić na dwa rodzaje: systemy rozpoznawania mowy ciągłej oraz systemy rozpoznawania wystąpień izolowanych słów. Główną ideą ukrytych modeli Markowa jest traktowanie sygnału mowy jako sekwencji wektorów obserwacji, które z jednej strony stanowią ciąg uczący w

procesie uczenia, podczas gdy tworzony jest model akustyczny mówcy, a z drugiej strony są wyjściem modeli w tworzonym procesie weryfikacji [6].

Łańcuchy Markowa stosuje się również do **generowania sekwencji liczb losowych** w celu dokładnego odzwierciedlenia bardzo skomplikowanych rozkładów prawdopodobieństw. Metoda, która wykorzystuje łańcuch Markowa do generowania sekwencji liczb nosi nazwę łańcuch Markowa Monte Carlo. W ostatnich latach wymieniona wcześniej metoda zrewolucjonizowała metody wnioskowania bayesowskiego, pozwalając na symulacje szerokiego zakresu rozkładów w odcinku bocznym i ich parametrów.

Łańcuchy Markowa mają również swoje zastosowanie w **modelowaniu biologicznym** szczególnie w procesach populacyjnych. Stan łańcucha Markowa w procesie populacyjnym jest analogiczny do liczby osóbników w danej populacji (0, 1, 2, 3, ..., n) zmiany stanu są analogiczne do dodawania lub usuwania osób z danej populacji. Jednym z takich przykładów jest macierz Lesli, która opisuje dynamikę populacji wielu gatunków. Innym przykładem jest modelowanie kształtu komórek w dzielących się segmentach komórek nabłonka. łańcuchy Markowa są również wykorzystywane w symulacjach funkcji mózgu takich jak symulacja kory nowe, która jest odpowiedzialna za odbieranie i przetwarzanie wrażeń zmysłowych, planowanie i wykonywanie ruchów dowolnych oraz procesy poznawcze (pamięć, myślenie, funkcje językowe).

Łańcuchy Markowa są używane w celu **przetwarzania informacji**. Teoria informacji, która wprowadza pojęcie entropii wykorzystuje łańcuchy Markowa w celu efektywnej kompresji danych za pomocą technik kodowania entropijnego - kodowanie arytmetyczne. Przykładem algorytmu do kompresji danych jest bezstratny algorytm LZMA[7], który łączy łańcuchy Markowa z kompresją Lempel-Ziv[8] aby osiągnąć bardzo wysokie współczynniki kompresji. Jako prekursora teorii informacji uważa się Claudia Shannona i jego dzieło z 1948 roku "A Mathematical Theory of Communication". Ukryte Modele Markowa, które mają swoją podstawę w łańcuchach Markowa są ważnym narzędziem w sieciach telefonicznych. Dużą rolę odgrywa tutaj algorytm Viterbiego, który jest wykorzystywany do korekcji błędów.

Łańcuchy Markowa wykorzystywane są również w **finansach i ekonomii**. Za ich pomocą modeluje się różne zjawiska, między innymi ceny aktywów i krachy rynkowe. Dynamiczna makroekonomia w dużym stopniu wykorzystuje łańcuchy Markowa. Przykładem jest wykorzystanie łańcuchów Markowa do egzogenicznego modelowania cen kapitału własnego w ogólnym ustawnieniu równowagi. Innym przykładem jest sporządzenie przez agencje ratingowe rocznych tabel prawdopodobieństw przejścia dla obligacji o różnych ratingach kredytowych.

Łańcuchy Markowa są szeroko stosowane w **termodynamice i mechanice statystycznej**. Prawdopodobieństwa są używane do reprezentowania nieznanych lub niezmodyfikowanych szczegółów systemu, jeśli można założyć, że dynamika jest niezmienna w czasie i nie należy brać pod uwagę odpowiedniej historii, która nie została jeszcze uwzględniona w opisie stanu. łańcuchy Markowa są również wykorzystywane praktycznie do symulacji sieci chromodynamiki kwantowej, która to ma za zadanie opisanie oddziaływań silnych (kwantowa teoria pola).

1.2.3 Łańcuchy Markowa w muzyce

Pierwszą osobą, która użyła łańcuchów Markowa do kompozycji muzyki był kompozytor rumuńsko - francuskiego pochodzenia Iannis Xenakis. Muzyk w 1958 roku użył łańcuchów Markowa w kompozycji Analogique[9]. Swoją pracę opisał w książce "Formalized Music: Thought and Mathematics in Composition [10]

Utwór muzyczny jest reprezentowany przez sekwencję zdarzeń, które są obiektami muzycznymi. Każdy obiekt muzyczny czyli nuta posiada dwie wartości: częstotliwość dźwięku, która reprezentowana jest przez literkę nuty oraz długość trwania. Linearna budowa utworu muzycznego pozwala na wyodrębnienie struktury fraz i polifonii. Biorąc pod uwagę statystyczny model w kontekście utworu muzycznego, możemy przypisać prawdopodobieństwo wystąpienia danej nuty w sekwencji. Dlatego też łańcuchy Markowa mają swoje za-

stosowanie w dziedzinie jaką jest algorytmiczne komponowanie muzyki. Można tutaj wymienić przykłady programowania odwołując się do programów CSound, Max, SuperCollider.

Istnieje wiele formatów, które umożliwiają zapis utworu muzycznego w formacie cyfrowym, który umożliwia jego odtworzenie na komputerze. Możemy wymieć tu powszechnie używane formaty plików takie jak mp3, flac czy wav. Jednak za pomocą tych formatów nie jesteśmy w prosty sposób wyodrębnić poszczególnych nut utworu muzycznego. W tym celu można się posłużyć innym sposobem zapisu muzyki. Formaty MIDI oraz MusicXML pozwalają zapisać utwór muzyczny nuta po nucie, dzięki temu też istnieje możliwość odczytania pełnego zapisu nutowego danego utworu zapisanego w jednym z tych formatów.

Standard MIDI (Musical Instrument Interface) został opracowany z myślą o komunikacji elektronicznych instrumentów muzycznych. Plik MIDI może się składać z wielu ścieżek, z których każda odpowiada brzmieniu określonego instrumentu. Należy mieć na uwadze, że pliki MIDI nie są plikami muzycznymi tak jak np. popularny format MP3 czy FLAC. W plikach MIDI znajdują się tylko instrukcje mówiące o tym jak dana nuta ma być zagrana. Następnie z wykorzystaniem samplera można odczytać taki plik co spowoduje w konsekwencji odegranie pewnego kawałka muzyki, który to jest reprezentowany przez dane tekstowe. Format MIDI może przechowywać tylko ograniczoną ilość informacji, nie można w nim zawrzeć chociażby słów utworu. Informacje o dźwięku zapisywane są w postaci tak zwanych zdarzeń. Przykładowe dwa zdarzenia MIDI zostały zaprezentowane w tabeli poniżej.

typ zdarzenia	time	channel	note	velocity
note_on	0	0	67	127
note_off	400	0	67	0

Tabela 1.1: Dwa zdarzenia MIDI

Natomiast odczyt zdarzenia wygląda następująco:

```

1 Track 0:
2 note_on channel=0 note=60 velocity=127 time=192
3 note_off channel=0 note=60 velocity=64 time=192
4 <meta message end_of_track time=0>

```

Listing 1.2: Odczyt pliku MIDI

Aby skonstruować prosty plik MIDI, którego zadaniem będzie zapisanie trzech nut: C, E, G posłużymy się językiem Python i pakietem mido.

```

1 import mido
2 filename = "outfile.mid"
3 def note_to_message(note):
4     return [
5         mido.Message('note_on', note=note, velocity=127, time=192),
6         mido.Message('note_off', note=note, velocity=64, time=192)
7     ]
8 def read_midi(filename):
9     mid = mido.MidiFile(filename)
10    for i, track in enumerate(mid.tracks):
11        print 'Track {}: {}'.format(i, track.name)
12        for message in track:
13            print message
14    with mido.midifiles.MidiFile() as midi:
15        track = mido.MidiTrack()
16        notes = [60, 64, 67]

```

```

17     for i in notes:
18         track.extend(note_to_message(i))
19         midi.tracks.append(track)
20     midi.save(filename)

```

Listing 1.3: Skrypt w Pythonie realizujący zapis określonych nut do pliku MIDI

Następnie strukturę stworzonego pliku można odczytać za pomocą funkcji `read_midi`. Jej wynik jest zaprezentowany poniżej:

```

1  Track 0:
2  note_on channel=0 note=60 velocity=127 time=192
3  note_off channel=0 note=60 velocity=64 time=192
4  note_on channel=0 note=64 velocity=127 time=192
5  note_off channel=0 note=64 velocity=64 time=192
6  note_on channel=0 note=67 velocity=127 time=192
7  note_off channel=0 note=67 velocity=64 time=192
8  <meta message end_of_track time=0>

```

Jedno zdarzenie jest reprezentowane przez cztery zmienne: `time`, `channel`, `note`, `velocity`. Zmienna `time` określa czas danego zdarzenia. `Channel` wskazuje jeden z 16 kanałów (0-15) do którego dane zdarzenie ma należeć. Poszczególne nuty nie są reprezentowane w postaci symboli A, H, G tylko w postaci liczb z zakresu od 0 do 127. Tabela poniżej przedstawia reprezentacje wszystkich możliwych nut w postaci liczbowej.

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

Rysunek 1.8: Reprezentacja nut w formacie MIDI

Nawiązując do powyżej tabeli, która reprezentuje dwa zdarzenia MIDI numery 67 oznaczają nutę G4. Kolejna zmienna o nazwie `velocity` przyjmuje wartości z zakresu (0-127). Jej zadaniem jest określenie siły danego dźwięku, inaczej mówiąc im mniejsza wartość zmiennej `velocity` tym dźwięk jest bardziej cichy. Pliki MIDI zawierają wszystkie istotne dane muzyczne, możliwe jest więc określenie prawdopodobieństwa przejścia z jednej nuty na drugą poprzez odpowiednią analizę pliku.

Innym dostępnym formatem służącym do zapisu informacji o utworze muzycznym jest standard MusicXML[11]. Jest to znacznikowy format prezentacji graficznej notacji muzycznej, który oparty jest na wzorach dokumentowych DTD[12]. Aby zrozumieć notację zapisu MusicXML wystarczy podstawa znajomość języka XML, który jest dość powszechnie wykorzystywany. Format MusicXML jest wspierany przez ponad 230 programów służących do zapisu notacji muzycznej takich jak Finale[13], Sibelius[14] czy MusicScore[15]. Poniżej zostanie przedstawiona przykładowa struktura pliku MusicXML.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
3  "http://www.musicxml.org/dtds/partwise.dtd">
4  <score-partwise>
5    <identification>
6      <rights></rights>
7      <encoding>
8        <software></software>
9        <encoding-date></encoding-date>
10       </encoding>
11     </identification>
12   <part-list>
13     <score-part id='P1'>
14       <part-name>Track 1</part-name>
15     </score-part>
16   </part-list>
17   <part id="P1">
18     <measure number='1'>
19       <attributes>atrybut miary</attributes>
20       <note>
21         <pitch>
22           <step>C</step>
23           <octave>4</octave>
24         </pitch>
25         <duration>96</duration>
26         <type>whole</type>
27       </note>
28     </measure>
29   </part>
30 </score-partwise>
```

Pierwsze trzy linijki definiują standard dokumentu XML. Plik partwise.dtd definiuje reprezentacje nut, plik timewise.dtd definiuje długości trwania nut. Następnie występuje główny tag o nazwie **<score-partwise>**, którego dzieckiem jest tag **<identification>**, którego zadaniem jest przechowywanie meta informacji o pliku. Tag **<score-part>** reprezentuje ścieżkę utworu. Przykład powyżej posiada jedną ścieżkę, która jest opisana w tagu **<measure>**. Poszczególna ścieżka posiada takie atrybuty jak tempo, rozmiar, tonację, metrum, klucz.

Bardzo ciekawym zastosowaniem łańcucha Markowa jest wykorzystanie go do generowania muzyki. Weźmy dla przykładu dwie melodie ($c, d, e, c, d, c, d, e, c, d$) oraz ($d, e, d, e, c, d, c, d, e, d$). Podane melodie tworzą przestrzeń stanów $Q = \{c, d, e\}$. Macierz prawdopodobieństw jest obliczana według wzoru:

$$a(q_i, q_j) = \frac{\#(q_i \rightarrow q_j)}{\sum_{q_k \in Q} \#(q_i \rightarrow q_k)}$$

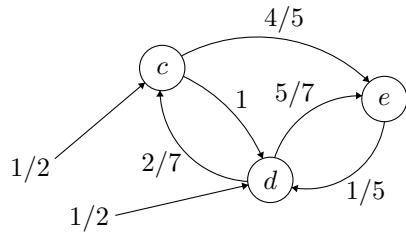
gdzie $\#(q_i \rightarrow q_j)$ jest ilością możliwych przejść ze stanu q_i do stanu q_j . Macierz prawdopodobieństw wygląda następująco:

$$a = \begin{bmatrix} a(c, c) & a(c, d) & a(c, e) \\ a(d, c) & a(d, d) & a(d, e) \\ a(e, c) & a(e, d) & a(e, e) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{2}{7} & 0 & \frac{5}{7} \\ \frac{4}{5} & \frac{1}{5} & 0 \end{bmatrix}$$

oraz:

$$\pi = (\pi(c), \pi(d), \pi(e)) = \left(\frac{1}{2}, \frac{1}{2}, 0\right)$$

Graf prawdopodobieństw dla powyższej macierzy przedstawia się w sposób następujący:



Rysunek 1.9: Graf prawdopodobieństw równoważny z macierzą przejść

1.2.4 Przykłady

Alvin Lin w artykule pod tytułem: "Generating Musik Using Markov Chains" [16] opisał proces generowania muzyki z wykorzystaniem łańcucha Markowa na podstawie pliku MIDI. Wyniki jego rozważań można zobaczyć w praktyce uruchamiając przygotowane przez niego skrypty[17] napisane w języku Python. Rozważmy fragment utworu Dla Elizy kompozycji Ludwiga Van Beethovena, który zamieszczony jest poniżej. Na podstawie tego fragmentu będziemy chcieli wygenerować nowy fragment. Poniższy schemat nutowy zapisywany jest do pliku MIDI.



Rysunek 1.10: Przykład melodii na podstawie której zostanie wygenerowana nowa.

Poniżej dokonano analizy pliku MIDI pod kontem poprawności. Jak widać poniżej plik MIDI zaczyna się z charakterystycznymi dla swojego formatu początkowymi nagłówkami.

```
$ python inspect.py ./fur_elise.mid
track 0:
<meta message smpte_offset frame_rate=25 hours=32 minutes=0 seconds=0 frames=0 sub_frames=0 time=0>
<meta message time_signature numerator=1 denominator=8 clocks_per_click=12 notated_32nd_notes_per_beat=8 time=0>
<meta message key_signature key='C' time=0>
<meta message set_tempo tempo=625001 time=0>
<meta message set_tempo tempo=800001 time=0>
<meta message time_signature numerator=3 denominator=8 clocks_per_click=26 notated_32nd_notes_per_beat=8 time=512>
<meta message key_signature key='C' time=0>
<meta message end_of_track time=972>
track 1: Piano
<meta message device name=name='SmartMusic SoftSynth 1' time=0>
<meta message track_name=name='Piano' time=0>
program_change channel=0 program=0 time=0
control_change channel=0 control=7 value=101 time=0
control_change channel=0 control=10 value=64 time=0
control_change channel=0 control=11 value=85 time=0
control_change channel=0 control=10 value=101 time=0
control_change channel=0 control=64 value=0 time=0
control_change channel=0 control=121 value=0 time=0
control_change channel=0 control=10 value=76 time=0
control_change channel=0 control=64 value=0 time=0
control_change channel=0 control=121 value=0 time=0
note_on channel=0 note=76 velocity=36 time=0
note_on channel=0 note=75 velocity=33 time=256
note_off channel=0 note=76 velocity=0 time=4
note_off channel=0 note=75 velocity=0 time=252
note_on channel=0 note=76 velocity=45 time=0
note_on channel=0 note=75 velocity=43 time=256
note_off channel=0 note=76 velocity=0 time=4
note_on channel=0 note=76 velocity=50 time=252
note_off channel=0 note=75 velocity=0 time=4
note_on channel=0 note=71 velocity=39 time=252
note_off channel=0 note=76 velocity=0 time=4
note_on channel=0 note=74 velocity=46 time=252
note_off channel=0 note=71 velocity=0 time=4
note_on channel=0 note=72 velocity=44 time=252
note_off channel=0 note=74 velocity=0 time=4
```

Rysunek 1.11: Fragment pliku MIDI po dokonaniu analizy

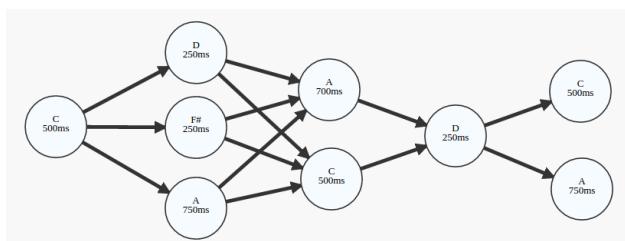
W dalszej części widać już poszczególne wartości nut razem z czasem oraz głośnością.

```
note_on channel=0 note=76 velocity=36 time=0
note_on channel=0 note=75 velocity=33 time=256
note_off channel=0 note=76 velocity=0 time=4
note_off channel=0 note=75 velocity=0 time=252
note_on channel=0 note=76 velocity=45 time=0
note_on channel=0 note=75 velocity=43 time=256
note_off channel=0 note=76 velocity=0 time=4
note_on channel=0 note=76 velocity=50 time=252
note_off channel=0 note=75 velocity=0 time=4
note_on channel=0 note=71 velocity=39 time=252
note_off channel=0 note=76 velocity=0 time=4
note_on channel=0 note=74 velocity=46 time=252
note_off channel=0 note=71 velocity=0 time=4
note_on channel=0 note=72 velocity=44 time=252
note_off channel=0 note=74 velocity=0 time=4
```

Rysunek 1.12: Wartości nut

Do zbudowania łańcucha Markowa potrzebne będą informacje zawarte w liniach zaczynających się od *note_on* informacje tam zawarte mówią o numerze nuty i czasie. Należy więc z pliku MIDI wyodrębnić wszystkie dane zawierające wartość *note_on*.

Dla każdej nuty, która gra z inną w tym samym czasie zaokrąglany jest jej czas do najbliższych 250 milisekund. Zostało to przedstawione na grafie poniżej.



Rysunek 1.13: Nuty przedstawione w formie grafu skierowanego

Reprezentacją takiego grafu może być macierz sąsiedztwa. Między węzłami poszczególna nuta przechodzi w kolejną nutę. Dla uproszczenia uwzględniony został tylko czas trwania danej nuty i liczba przypadków w

których została zmieniona na inną nutę. Macierz sąsiedztwa dla powyższego grafu przedstawia się w sposób następujący:

1	C (500 ms)		D (250 ms)		F# (250 ms)		A (750 ms)
2	C	0		2	1		1
3	D	2		0	0		2
4	F#	1		0	0		1
5	A	1		1	0		1

Listing 1.4: Macierz przejścia

Ilustracja poniżej przedstawia macierz sąsiedztwa dla fragmentu utworu "Dla Elizy"

	40:250	71:250	45:250	72:250	76:250	69:250
64	2	2	0	0	0	0
68	0	0	1	1	0	0
76	0	0	0	0	2	0
72	0	0	1	0	0	1

Rysunek 1.14: Macierz sąsiedztwa dla fragmentu utworu "Dla Elizy"

Numery komórek reprezentują nuty i czas trwania nuty do której nastąpiło przejście. Numery wierszy oznaczają nuty, z który odbyło się przejście. Każda pozycja w macierzy reprezentuje liczbę przypadków, gdy dana nuta została przeniesiona do innej. Do wygenerowania nowego utworu wybierana jest losowa nuta, następnie jest ona odnajdywana w macierzy. Kolejno z grupy wszystkich wyprowadzeń losowana jest następna nuta, jednak priorytet ma tutaj częstość przejść. Do wygenerowania nowych melodii posłużono się 50 iteracjami.

Rysunek 1.15: Wyniki - przykład 1



Rysunek 1.16: Wyniki - przykład 2

Niestety, ale melodie nie są przyjazne dla ucha. Chociaż, nie są też abstrakcyjne. Widocznie widać, że brakuje tutaj formy.

Trochę inne, ale bardzo podobne podejście zaprezentował Justin Bozonier w artykule "Algorithm of the Week: Generate Music Algorithmically" [18]. W przykładzie wykorzystywana jest biblioteka PySynth[19] za pomocą, której autor generuje plik wav powstały z wcześniej wygenerowanych symboli nut w postaci A, B, C. W tym przykładzie nie jest wykorzystywany plik MIDI tylko zadaniem użytkownika jest ręczne wprowadzenie melodii na podstawie, której zostanie skonstruowana macierz przejść, a następnie nowa sekwencja nut. Poniżej został przedstawiony sposób definiowania sekwencji nut. Jak widać pierwszym elementem tablicy jest symbol nuty, a drugim czas trwania przy czym 1 oznacza całą nutę, 2 półnuttę, 4 ćwierć nutę, 8 ósemkę i 16 szesnastkę. Cały kod dostępny jest w serwisie GitHub[20].

```

1  [...]
2  musicLearner.add(["c", 4])
3  musicLearner.add(["c", 4])
4  musicLearner.add(["d", 8])
5  musicLearner.add(["e", 4])
6  musicLearner.add(["f", 8])
7  musicLearner.add(["g", 2])
8  [...]

```

Listing 1.5: Sposób zapisu sekwencji nut

Jeszcze inne podejście zaprezentował użytkownik johmryan[21], którego aplikacja składa się z dwóch części. Pierwsza część aplikacji to strona www, która zbiera od użytkownika ciąg nut w postaci symboli np. DBAGA oraz informacje o tym czy generowanie muzyki ma opierać się na łańcuchu Markowa pierwszego rzędu czy drugiego. Następnie dane wprowadzone przez użytkownika są wysyłane do skryptu napisanego w Pythonie. Skrypt na podstawie otrzymanej sekwencji tworzy nowe sekwencje i wysyła do witryny internetowej, która prezentuje wyniki.

Rozdział 2

Gramatyki

2.1 Teoria języków i gramatyk formalnych

Przed zdefiniowaniem języka formalnego należy zdefiniować pojęcie alfabetu.

Definicja 2.1.1. Przez alfabet rozumiemy dowolny skończony zbiór symboli $V = \{a_1, \dots, a_n\}$. Skończone ciągi (nazywane słowami) z powtórzeniami elementów zbioru V zapisywane są w postaci $a_1a_2a_3a_1$. Dla każdego $a \in V$ słowo jednoliterowe składające się z pojedynczego symbolu a jest identyfikowane z tym symbolem i oznaczane przez a . Zbiór wszystkich słów nad alfabetem V oznaczany jest przez V^* .

Definicja 2.1.2. Niech V będzie dowolnym alfabetem. Każdy podzbiór L zbioru V^* nazywamy językiem nad alfabetem V . Operacje wyprowadzenia można podzielić na dwie grupy. Do pierwszej z nich należą operacje teoriomnogościowe: operacje sumy, przecięcia, różnicę. Do drugiej grupy należą operacje złożenia i odbicia zwierciadlanego.

Definicja 2.1.3. **Złożeniem** języków L_1 oraz L_2 nad alfabetem Σ nazywamy język $X : L_1 \cdot L_2 = \{x \cdot y | x \in L_1 \wedge y \in L_2\}$

Definicja 2.1.4. W podobny sposób definiujemy potęgę języka L :

- $L^\emptyset = \{\epsilon\}$
- $L^{n+1} = L^n \cdot L$
- $L^0 = \{\emptyset\}$

Definicja 2.1.5. **Odbiciem zwierciadlanym** słowa $w = a_1 \dots a_n \in A^*$ nazywamy słowo $\bar{w} = a_n \dots a_1$. Odbiciem zwierciadlanym języka $L \subset A^*$ nazywamy język $\bar{L} = \{\bar{w} \in A^* : w \in L\}$

Przez gramatykę należy rozumieć systematyczny opis wybranego języka naturalnego. Opis musi obejmować jego składnię (syntaktykę), znaczenie (semantykę) i fonologię, czyli dźwiękowy system języka. Reguły składni określają regularności rządzące kombinacjami słów, semantyka bada znaczenie słów i zdań, a fonologia wyróżnia dźwięki i ich dopuszczalne zestawienia w opisywanym języku. Teoria języków formalnych bada tylko syntaktyczne własności języków.

Definicja 2.1.6. Gramatyką nazywamy czwórkę $G = \langle \Sigma, V, P, S \rangle$ gdzie:

- Σ jest alfabetem,
- V jest skończonym zbiorem zmiennych (symboli nieterminalnych) rozłączonym z Σ ,
- $S \in V$ jest wyróżnionym symbolem generującym (symbol startowy),
- $P \subseteq (\Sigma \cup V)^+ \times (\Sigma \cup V)^*$ - jest skończonym zbiorem reguł (produkcji)

Definicja 2.1.7. Produkce gramatyki określają relację **bezpośredniego wyprowadzenia** \Rightarrow na słowach nad alfabetem $(\Sigma \cup V)$ następująco:

$x \Rightarrow y$ wtedy i tylko wtedy, gdy istnieją słowa x_0 i x_1 oraz reguła $\alpha \rightarrow \beta$ gramatyki G takie, że $x = x_0 \alpha x_1$ i $y = x_0 \beta x_1$.

Definicja 2.1.8. Wyprowadzeniem słowa y ze słowa x w gramatyce G ($x \Rightarrow^* y$) nazwa się każdy ciąg słów x_0, \dots, x_n , $n \geq 0$, taki że $x_i \Rightarrow x_{i+1}$ dla $i = 0, \dots, n-1$ oraz $x_0 = x$ i $x_0 = x$ i $x_n = y$. Relacja wyprowadzenia jest zwrotnym i przechodnim domknięciem (\Rightarrow^*) relacji bezpośredniego wyprowadzenia (\Rightarrow).

Definicja 2.1.9. Język $L(G)$ generowany przez gramatykę G definiowany jest jako zbiór wszystkich słów nad alfabetem Σ , dla których istnieje wyprowadzenie z symbolu startowego S , czyli: $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$.

2.1.1 Języki regularne

Dla języka regularnego musi istnieć automat o skończonej liczbie stanów, który potrafi zdecydować czy dane słowo należy do języka.

Pojęcie 2.1.1. Gramatyka prawostronnie liniowa, to taka, w której wszystkie produkcje mają postać: $A \rightarrow \alpha B$ lub $A \rightarrow \alpha$ dla $A, B \in V$ i $\alpha \in \Sigma^*$

Definicja 2.1.10. Niech Σ będzie skończonym alfabetem. Rodzina $REG(\Sigma^*)$ języków regularnych nad alfabetem Σ to najmniejsza, w sensie inkluzji, rodzina R języków taka, że:

1. $\emptyset \in R$, dla każdego $a \in \Sigma$, $\{a\} \in R$ - są to tak zwane języki atomowe,
2. jeśli $X, Y \in R$ to $X \cup Y \in R$, $X \cdot Y \in R$
3. jeśli $X \in R$, to $X^* = \bigcup_{n=0}^{\infty} X^n \in R$

Z definicji wynika, że $\{\epsilon\} = \emptyset^* \in R$.

Języki regularne tworzone są z języka pustego i języków jednoelementowych złożonych z jednej litery za pomocą skończonej liczby operacji sumy, konkatenacji i gwiazdki. Konkatenacja i suma języków skończonych to języki skończone, natomiast operacja gwiazdki na dowolnym języku różnym od \emptyset i $\{\epsilon\}$ powoduje powstanie języka nieskończonego.

Przykład 2.1.1. Przykłady języków regularnych:

- $\{a, b\} \cdot \{a, b\} = \{aa, ab, ba, bb\}$ jest regularny, ponieważ jest zbudowany ze złożenia dwóch języków jednoelementowych
- $\{a, aa\} \cdot \{a, aa\} = \{aa, aaa, aaaa\} = \{a^2, a^3, a^4\}$ jest regularny, ponieważ jest złożony ze złożenia dwóch języków złożonych z konkatenacją języków skończonych

- $X = \{a\}, X^* = \{\epsilon, a, a^2, a^3, \dots\}$
- $\{a\} \cup \{aa\} = \{a, aa\}$ jest regularny, ponieważ jest zbudowany z sumy dwóch języków złożonych z języków jednoelementowych

Pojęcie 2.1.2. Deterministycznym i skończonym automatem A nazywamy piątkę $\langle \Sigma, Q, F, \delta, q_0 \rangle$, w której:

- Σ jest zbiorem skończonym nazywanym alfabetem
- Q jest zbiorem skończonym nazywanym zbiorem stanów (rozłącznym z Σ)
- $F \subseteq Q$ jest podzbiorem stanów końcowych (akceptujących, terminalnych)
- $q_0 \in Q$ jest wyróżnionym stanem początkowym
- funkcja δ zwana jest funkcją przejścia lub zmiany stanu, $\delta : Q \times \Sigma \rightarrow Q$

Funkcję δ łatwo można rozszerzyć do funkcji $\delta^* : Q \times \Sigma^* \rightarrow Q$ w następujący sposób:

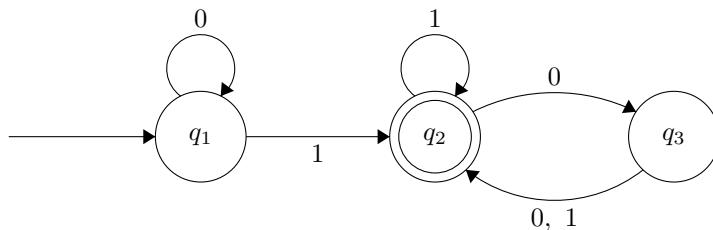
- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, Xw) = \delta^*(\delta(q, x), w)$

Definicja 2.1.11. Język $L \subset \Sigma^*$ jest rozpoznawalny (akceptowalny) wtedy i tylko wtedy, gdy istnieje automat skończony $A = \langle \Sigma, Q, F, \delta, q_0 \rangle$ taki, że: $L = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$.

Twierdzenie 2.1.1. Dla dowolnego języka $L \subset \Sigma^*$ następujące warunki są równoważne:

- Język L jest regularny
- Istnieje gramatyka G prawostronnie liniowa taka, że $L = L(G)$ - język L jest generowany przez gramatykę G
- Istnieje automat skończony $A = \langle \Sigma, Q, F, \delta, q_0 \rangle$, taki że $L = L(A)$ - to znaczy, że język L jest akceptowany przez A

Automat jest urządzeniem, który posiada nieskończoną taśmę wejściową, służącą tylko do czytania słów nad alfabetem Σ . Pod wpływem przeczytanej ostatnio informacji (litery słowa na wejściu) automat zmienia swój stan. Automat skończony wczytuje dane słowo znaku po znaku i zgodnie z funkcją przejścia podejmuje decyzję o zmianie swojego stanu. Po wczytaniu całego słowa sprawdza, czy znajduje się w jednym ze stanów akceptujących. Jeżeli automat znajdzie się w stanie akceptującym to powiem, że automat akceptuje słowo. W przeciwnym wypadku mówimy, że słowo nie jest zaakceptowane przez automat. Schemat automatu może zostać zaprezentowany jako graf skierowany, którego wierzchołkami są stany, a krawędzie są etykietowane literami alfabetu wyjściowego.



Powyższy automat akceptuje język $L = \{w \in \{0,1\}^* | w \text{ zawiera przynajmniej jedną jedynkę i parzystą liczbę zer na końcu}\}$.

2.1.2 Języki bezkontekstowe - gramatyki bezkontekstowe

W języku naturalnym jakim jest język polski gramatyka ustala zasady poprawnego budowania zdań. Dzięki temu rozmawiające ze sobą osoby są w stanie się zrozumieć. Języki bezkontekstowe są rodziną szerszą niż omówione wcześniej języki regularne. Język bezkontekstowy to taki język formalny, dla którego istnieje niedeterministyczny automat ze stosem, który potrafi zdecydować czy dany łańcuch należy do języka. Również dla takiego łańcucha musi istnieć gramatyka bezkontekstowa. Gramatyki bezkontekstowe generują napisy poprzez sekwencję przepisywać, która ma strukturę drzewa. Ze względu na strukturę drzewiastą gramatyki bezkontekstowe nadają się bardzo dobrze do opisu syntaktyki języków programowania.

Definicja 2.1.12. Gramatyka jest bezkontekstowa, gdy wszystkie jej produkcje są postaci $A \rightarrow \beta$, gdzie $A \in V, \beta \in (\sum \cup V)^*$

Przykład 2.1.2. Gramatyka $G = (\{a, b\}, S, S \rightarrow aSb|\epsilon, S)$ generuje język $\{a^n b^n\}$. Język wygenerowany przez gramatykę nie jest regularny.

Tak jak wyrażenia regularne mają równoważny z nimi automat - automat skończony, tak gramatyki bezkontekstowe mają swój odpowiednik maszynowy, jest to automat ze stosem. Automat ze stosem to automat skończony, który został wyposażony w dodatkową pamięć w postaci stosu - jest to lista działająca na zasadzie first in, last out. Nowa symbole mogą być dopisywana lub czytane jedynie na wierzchołku listy. Automaty skończone dysponują skońzoną pamięcią, niezależną od długości danych wejściowych. W przypadku automatów ze stosem, dysponujemy także nieskońzoną pamięcią, dzięki czemu można rozpoznawać szerszą klasę języków. Tą klasą są języki bezkontekstowe.

Definicja 2.1.13. Automatem ze stosem nazywamy system $A_s = \langle \sum, Q, F, \Gamma, \delta, q_0, Z_0 \rangle$, gdzie:

- \sum - skończony zbiór symboli wejściowych (alfabet)
- Q - skończony zbiór stanów
- Γ - skończony zbiór symboli na stosie (alfabet stosowy)
- $q_0 \in Q$ - wyróżniony stan początkowy
- $Z_0 \in \Gamma$ - symbol początkowy na stosie - wyróżniony symbol stosowy
- $F \subseteq Q$ - jest podzbiorem stanów końcowych
- δ - funkcja przejścia

Maszyna będąc w określonym stanie czyta literę słowa wejściowego oraz sprawdza, jaki symbol znajduje się na wierzchołku stosu. Na tej podstawie podejmowana jest decyzja o zmianie stanu, następnie zdejmowany jest element z wierzchołka stosu. Na zdjętym elemencie umieszczane jest słowo złożone z symboli stosowych.

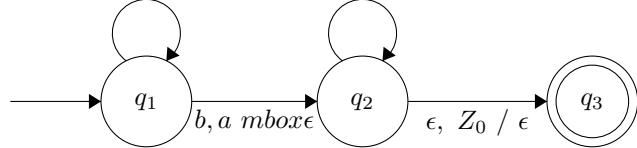
Twierdzenie 2.1.2. Dla dowolnego języka $L \subset \sum^*$ następujące warunki są równoważne:

- Język L jest bezkontekstowy
- Istnieje gramatyka G prawostronnie liniowa taka, że $L = L(G)$ - język L jest generowany przez gramatykę bezkontekstową G
- Istnieje automat ze stosem $\underline{A}_s = \langle \sum, Q, F, \Gamma, \delta, q_0, Z_0 \rangle$, taki że $L = L(\underline{A})$ - to znaczy, że język L jest akceptowany przez \underline{A}

Twierdzenie 2.1.3. Język $L \subset \Sigma^*$ jest generowany przez gramatykę bezkontekstową wtedy i tylko wtedy gdy L jest akceptowalny przez automat ze stosem.

Niech będzie dany automat ze stosem $A = \langle \Sigma, Q, F, \Gamma, \delta, q_1, Z_0 \rangle$, który rozpoznaje język $L = \{a^n b^n \mid n \geq 1\}$

$$a, Z_0 \rightarrow aZ_0; a, a \rightarrow aa \quad b, a \rightarrow b, a \text{ mbox } \epsilon$$



Zapis $a, b \rightarrow c$ oznacza, że maszyna czytając ze słowa wejściowego literę a , może zastąpić literę b na wierzchołku ciągiem symboli c .

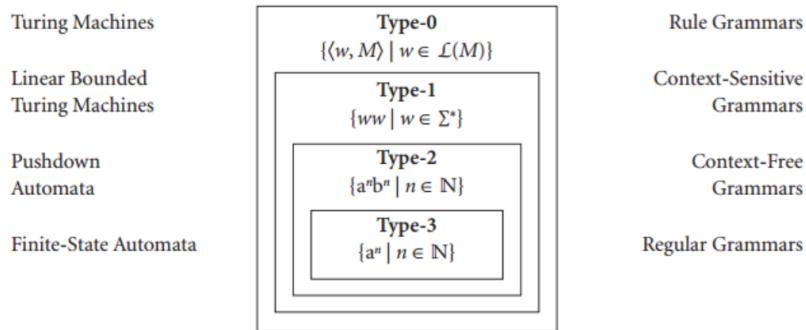
2.1.3 Hierarchia Chomsky'ego

Amerykański językoznawca Noam Chomsky zaproponował cztery typy gramatyk, które mają swoją hierarchię. Hierarchia Chomsky'ego to hierarchia czterech klas języków. Są to języki: regularne, bezkontekstowe, kontekstowe i częściowo obliczalne. Każdej z tych klas przypada jeden rodzaj gramatyki. Odpowiadają im gramatyki: liniowe, bezkontekstowe, kontekstowe i rekurencyjnie przeliczalne. Tym samym klasom odpowiadają cztery różne modele obliczeniowe: automaty skończone, automaty ze stosem, maszyny Turinga (ograniczone liniowo i dowolne).

Definicja 2.1.14. Niech $G = \langle \Sigma, V, P, S \rangle$ będzie dowolną gramatyką.

1. G jest typu 1 lub jest gramatyką kontekstową wtedy i tylko wtedy, gdy wszystkie produkcje $\alpha \rightarrow \beta$ spełniają warunek $|\alpha| \leq |\beta|$
2. G jest typu 2 lub jest gramatyką bezkontekstową wtedy i tylko wtedy, gdy wszystkie produkcje są postaci $A \rightarrow \beta$, gdzie $A \in V, \beta \in (\Sigma \cup V)^*$
3. G jest typu 3 lub jest gramatyką regularną wtedy i tylko wtedy gdy jest prawostronnie lub lewostronnie liniowa.

Poniższy diagram przedstawia graficzną reprezentację języków formalnych.



Rysunek 2.1: Hierarchia języków formalnych

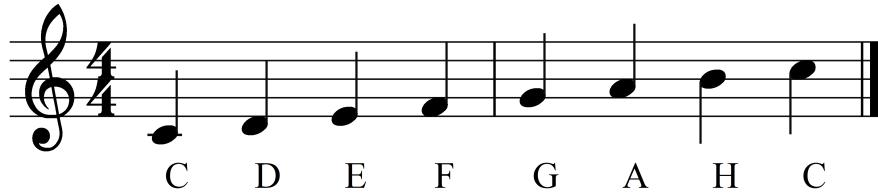
Definicja 2.1.15. Język należy do danej klasy wtedy i tylko wtedy, gdy jest możliwe zbudowanie gramatyki formalnej, która generuje dany język, a której reguły przestrzegają ograniczeń dla danej klasy.

2.2 Gramatyki i muzyka

Wykorzystanie gramatyki do generowania muzyki charakteryzuje się kilkoma tendencjami. Pierwszą z nich jest określenie jakim rodzajem muzyki ma dana gramatyka się zajmować. Na podstawie rodzaju muzyki należy dokonać charakteryzacji według rodzajów akordów, tempa, linii melodycznej itp. Przykładowo dla muzyki jazzowej można stworzyć produkcje, które będą tworzyły harmoniczne pasujące do siebie akordy bazujące na teorii jazzu i następstwie akordów.

2.2.1 Przykład wykorzystania gramatyk w muzyce

Jako przykład zostanie przygotowana gramatyka, której celem będzie wygenerowanie prostej melodii bazującej na tonacji C-dur. Na gamę C-dur składają się następujące dźwięki: C, D, E, F, G, A, H.



Rysunek 2.2: Gama C-dur

Zapis tonacji C-dur podobnie jak A-moll (A, H, C, D, E, F, G) nie posiada znaków chromatycznych. Tonacje są pokrewne ze sobą.

Melodia nie powinna przekraczać dwóch oktaf (dwie oktawy: C, D, E, F, G, A, H, C, D, E, F, G, A, H). Metrum melodii czyli czynnik porządkujący ugrupowania rytmiczne za pomocą regularnie powtarzających się akcentów metrycznych wyniesie 4/4 (cztery czwarte). Góra liczba metrum oznacza ile ma być w takcie jednostek miarowych oznaczonych przez liczbę dolną. Przykładowo określenie 2/4 wskazuje, że w jednym taktie mają być dwie ćwierćnoty lub wartości dające w sumie dwie ćwierćnoty. Dolna liczba wskazuje, jaka wartość nutowa jest podstawą miary taktowej. Jednostką miarową może być cała nuta, półnuta, ósemka, szesnastka i trzydziestodwójka. Aby utworzyć melodię kolejność nut musi być ulożona w taki sposób aby utwór był przyjemny dla ucha. Jeżeli nuty zostaną użyte w sposób losowy melodia nie będzie brzmiała dobrze.

Przykład 2.2.1. Rozważmy gramatykę $G = \langle \Sigma, V, P, S \rangle$ gdzie:

- $\Sigma = \{n.k; n \in \langle 0, 87 \rangle, k \in \langle 1, 3 \rangle\}$ - wyrażenie $n.k$ będzie rozpatrywane jako nuta gdzie n oznaczać będzie wysokość nuty, a k długość trwania dźwięku 1 - cała nuta, 2 - półnuta, 3 - ćwierć nuta.
- $V = \{S, Beg, Mid, End\}$
- Zbiór produkcji P jest następujący:
 $S \rightarrow Beg \ Mid \ End$
 $Beg \rightarrow 47'1 \ 49'1 \ 51'1 \ 52'1$

$Beg \rightarrow 52'2 56'2 59'2 57'2$
 $Beg \rightarrow Beg 52'2 54'2 56'2 54'2 Mid$
 $Beg \rightarrow 47'1 49'1 51'1 52'1 54'3 End$
 $Mid \rightarrow Beg 52'2 56'2 End$
 $Mid \rightarrow 54'2 56'1 54'1 52'2 47'2$
 $Mid \rightarrow Mid 47'2 52'2 56'1 57'1 56'1 52'1 Beg$
 $Mid \rightarrow 56'2 52'1 54'1 56'2$
 $End \rightarrow Mid Beg 54'2 51'2 52'3$
 $End \rightarrow 52'2 52'2 51'2 52'3$
 $End \rightarrow Beg 52'2 54'2 52'3 End$
 $End \rightarrow 52'4$
 $End \rightarrow 42'2 40'3$
 $End \rightarrow 49'1 47'1 49'1 51'1 52'3 End$

Za pomocą języka Python można zautomatyzować proces wyprowadzeń. W tym celu będzie pomocna biblioteka nltk, która służy do pracy z przetwarzaniem języka naturalnego.

```

1  from nltk import CFG
2  from nltk.parse.generate import generate
3  import re
4  from music21 import *
5  notes = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
6  num_notes = 127
7  num_octaves = 10
8  octave_count = 0
9  num_to_note = dict()
10
11 for i in range(0, num_notes+1):
12     # print "{} {} {}".format(i, octave_count, notes[i%len(notes)])
13     num_to_note.update({str(i): notes[i % len(notes)]+str(octave_count)})
14     if i % 12 == 11:
15         octave_count += 1
16
17 gramma = """
18 S -> Beg Mid End
19 Beg -> '47.1 49.1 51.1 52.1'
20 Beg -> '52.2 56.2 59.2 57.2'
21 Beg -> Beg '52.2 54.2 56.2 54.2' Mid
22 Beg -> '47.1 49.1 51.1 52.1 54.3' End
23 Mid -> Beg '52.2 56.2' End
24 Mid -> '54.2 56.1 54.1 52.2 47.2'
25 Mid -> Mid '47.2 52.2 56.1 57.1 56.1 52.1' Beg
26 Mid -> '56.2 52.1 54.1 56.2'
27 End -> Mid Beg '54.2 51.2 52.3'
28 End -> 52'2 52'2 51'2 52'3
29 End -> Beg '52.2 54.2 52.3' End
30 End -> '52.4'
31 End -> '42.2 40.3'
32 End -> '49.1 47.1 49.1 51.1 52.3' End
33 """
34 environment.set("musescoreDirectPNGPath",      "/usr/bin/musescore")
35 environment.set("musicxmlPath",      "/usr/bin/musescore")
36 environment.set("midiPath",      "/usr/bin/lilypond")
37

```

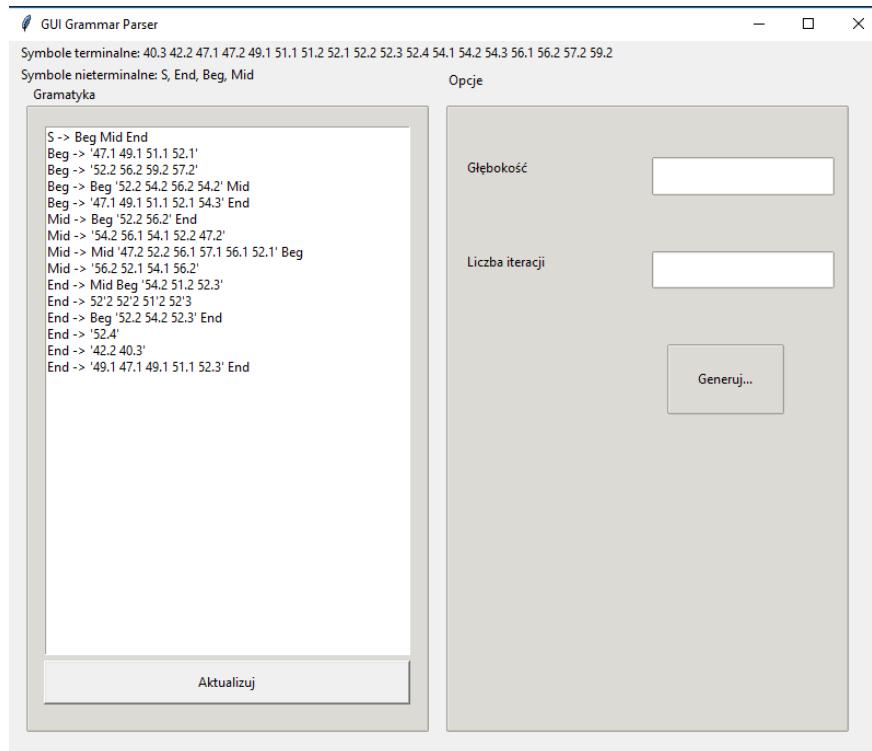
```

38 conv = converter.subConverters.ConverterLilypond()
39
40 grammar = CFG.fromstring(grammar)
41 track_array = []
42 for track in generate(grammar, n=5, depth=3):
43     track_array.append(''.join(track))
44     print(track)
45
46 tracks_notes = []
47
48 for tr in track_array:
49     s1 = stream.Stream()
50     for n in re.findall(r'\S+', tr):
51         s1.append(note.Note(num_to_note.get(str(n.split('.')[0])), quarterLength=int(n.split('.')[1])))
52     tracks_notes.append(s1)
53
54
55 for i, k in enumerate(tracks_notes):
56     print(k)
57     conv.write(k, fmt='lilypond', fp='/mnt/c/Users/Lukasz/MGR_Code/Grammar/examples/' +
58                 str(i), subformats=['png', 'midi'])

```

Listing 2.1: Generowanie wyrowadzeń z gramatyki bezkontekstowej

Do powyższego przykładu został stworzony interfejs graficzny. Za pomocą interfejsu można wygenerować przykładowe melodie.



Rysunek 2.3: Interfejs aplikacji

Interfejs pozwala na wprowadzenie ilości wyprowadzeń oraz głębokości, która jest istotna przy procesie wyprowadzania słów. Zbyt duża głębokość prowadzi do zapętlenia. Po ustaleniu odpowiednich liczb należy wcisnąć przycisk "Generuj".



Rysunek 2.4: Wyniki dla głębokości równej 4 oraz liczby iteracji równej 3

Po pomyślnym wyprowadzeniu słów program wyświetla wyniki w nowym oknie.

2.2.2 Programy wykorzystujące gramatyki

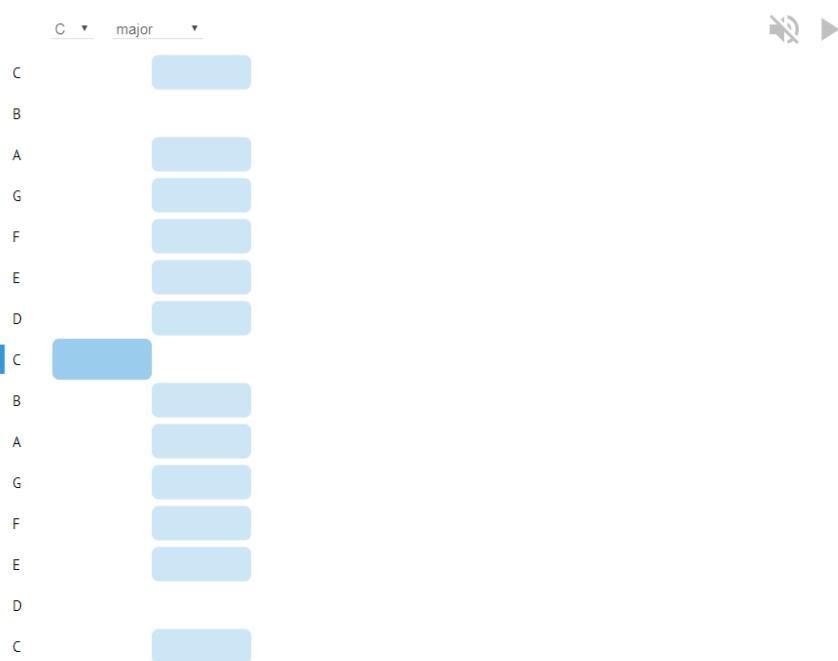
MusicMachine.io - system stworzony przez Johna Leszczyńskiego. Jest to przewodnik, którego celem jest akceptowanie sekwencji nut, które będą spełniały zasady stylu Cantus Firmus. Zasady, które muszą być spełnione to:

- sekwencja musi się składać z co najmniej 8 nut
- w sekwencji musi wystąpić punk kulminacyjny - jedna nuta musi być wyższa niż pozostałe
- koniec sekwencji musi skończyć się na tonice (nuta zaczynająca)
- akceptowane interwały to: sekundy wielkie, sekundy małe, tercje, seksty oraz kwarty czyste, kwinty i oktawy

Autor oparł swój system na bibliotece napisanej w JavaScript counterpoint (John Jeszczynski jest również autorem tej biblioteki). Interfejs aplikacji jest dostępny pod adresem <http://musicmachine.io>.

Cantus Firmus Guide

"A well-constructed cantus reveals in embryo many of the characteristics of more highly developed musical organisms."
—Felix Salzer and Carl Schachter, *Counterpoint in Composition*



Rysunek 2.5: Interfejs aplikacji musicmachine

Interfejs musicmachine podaje listę możliwych nut po każdym kroku. Jeżeli sekwencja wybranych nut jest zgodna z zasadami Cantus Firmus to jest akceptowana przez gramatykę umieszczoną w bibliotece. Biblioteka counterpoint wykorzystuje bibliotekę GrammarGraph. Biblioteka GrammarGraph pozwala na stworzenie gramatyki bezkonieckowej, a następnie pozwala na wyprowadzanie słów z wcześniej zadanej gramatyki oraz potrafi zdecydować czy jakieś wyprowadzenie rzeczywiście pochodzi z gramatyki, która została zadana.

Przykład 2.2.2. Przykład gramatyki zbudowanej w oparciu o temat z Sonaty Księżyckowej:

```

1  var jupiterGrammar = {
2      InfinitePhrase: [ 'JupiterTheme InfinitePhrase',
3          'SecondMotive InfinitePhrase' ],
4      JupiterTheme: [ '2 3 -2' ],
5      SecondMotive: [ '4 StepDown' ],
6      StepDown: [ '-2',
7                  '-2 StepDown' ]
8  }

```

W powyższej gramatyce można wyróżnić symbole terminalne na które składają się: 2, 3, -2, 4. Symbole terminalne oznaczają odległość między nutami - interwały. W skład symboli nie-terminalnych wchodzą:

InfinitePhrase, JupiterTheme, SecondMotive, StepDown. Definicje rekurencyjne w gramatyce sprawiają, że długość wyprowadzanego słowa może być nieskończona.

Gramatyka użyta w aplikacji musicmachine.io wygląda następująco:

```

1 var upOnly = {
2
3   // designed to be an infinite phrase
4   UpPhrase: [ 'UpLeap DownStepPhrase UpPhrase' ,
5   'UpStepPhrase DownPhrase' ],
6
7   // all choices must be prepared for a potential down leap in downphrase
8   UpStepPhrase: [ 'Up2Phrase' ,
9   'Up3Phrase' ,
10  'UpLeapForwardPhrase' ],
11
12  // after 2, can reverse direction or continue up 2 or 3
13  Up2Phrase: [ '2' ,
14  '2 Up2Phrase' ,
15  '2 Up3Phrase' ],
16
17  // after 3, can reverse direction or continue up 2
18  Up3Phrase: [ '3' ,
19  '3 Up2Phrase' ],
20
21  // up leap must be recovered with down step
22  // prepare for a potential downward leap by adding another UpPhrase
23  UpLeapForwardPhrase: [ '2 UpLeapForward DownStepPhrase UpPhrase' ],
24
25  // allowed up leaps after already moving a second up
26  UpLeapForward: [ '4' , '5' ],
27
28  // allowed leaps at the beginning or after a direction change
29  UpLeap: [ '4' , '5' , '6' , '8' ]
30 }
```

Symboly terminalne: 2, 3, 4, 5, 6, 8, oznaczają odległości od nut - interwały. Symbole nie-terminalne: UpPhrase, UpStepPhrase, Up2Phrase, Up3Phrase, UpLeapForwardPhrase, UpLearnForward, UpLeap, DownStepPhrase, DownPhrase. Reguły obowiązują w dwóch kierunkach, dolnym i górnym. Gramatyka została zdefiniowana tylko w jednym kierunku, ponieważ kierunek przeciwny można uzyskać zamieniając symbol terminalny na przeciwny. Po każdym wyborze nuty sprawdzane jest czy sekwencja jest prawidłowa z wszystkimi zasadami Cantus Firmus. Za sprawdzanie odpowiedzialna jest poniższa funkcja:

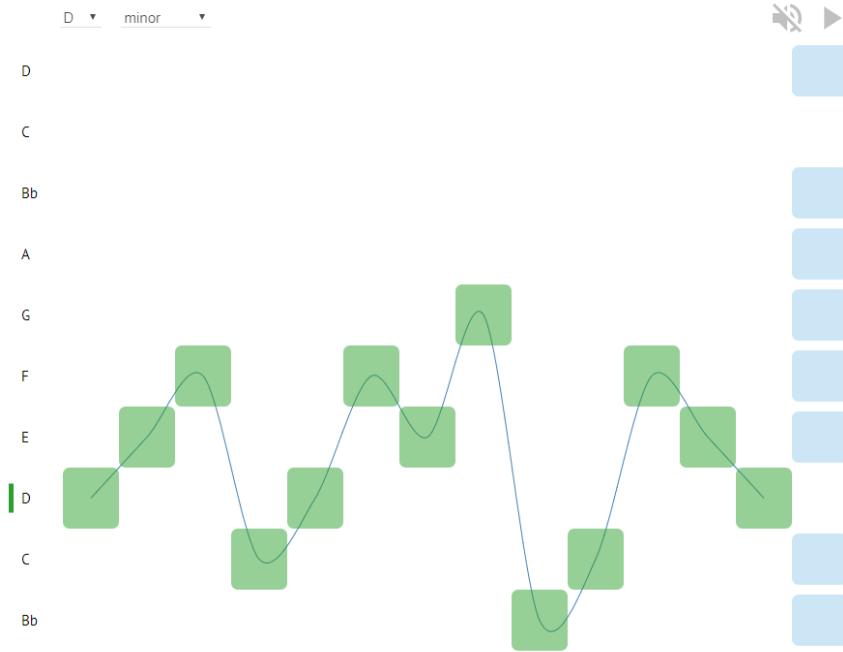
```

1 this.isValid = function () {
2   var cf = this.cf()
3
4   // is it long enough
5   if (cf.length < MIN_CFLLENGTH || cf.length > MAX_CFLLENGTH) {
6     return false
7   }
8
9   // is last note tonic?
10  if (Pitch(cf[cf.length - 1]).pitchClass() !== guide.tonic()) {
11    return false
12  } else if (Pitch(cf[0]).pitchClass() === guide.tonic()) {
13    // if first note is tonic, last note should end in the same octave
14    // if first note is not tonic, this is probably a first species counterpoint
15    if (cf[0] !== cf[cf.length - 1]) {
```

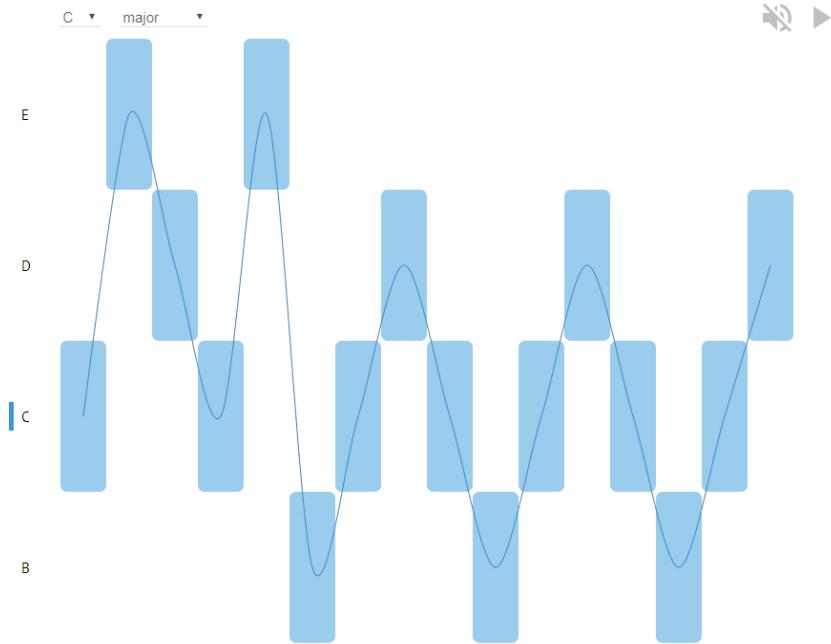
```

16     return false
17   }
18 }
19
20 // is the penultimate note scale degree 2 or possible 7?
21 if (intervalSize(cf[cf.length - 2], cf[cf.length - 1]) !== 2) {
22   return false
23 }
24
25 // is there a unique climax (highest note is not repeated)?
26 var sorted = sortPitches(cf)
27 if (sorted[sorted.length - 1] === sorted[sorted.length - 2]) {
28   return false
29 }
30
31 return true
32 }
```

Poniżej pokazane zostały dwa przykłady sekwencji, w których jeden jest sekwencją, która spełnia zasady Cantus Firmus, a drugi nie. Prawidłowa sekwencja jest podświetlona na zielono.



Rysunek 2.6: Prawidłowa sekwencja Cantus Firmus



Rysunek 2.7: Nieprawidlowa sekwencja

Przykład wyprowadzenia melodii, która jest zgodna z zasadami stylu Cantus Firmus bezpośrednio z wykorzystaniem biblioteki:

```

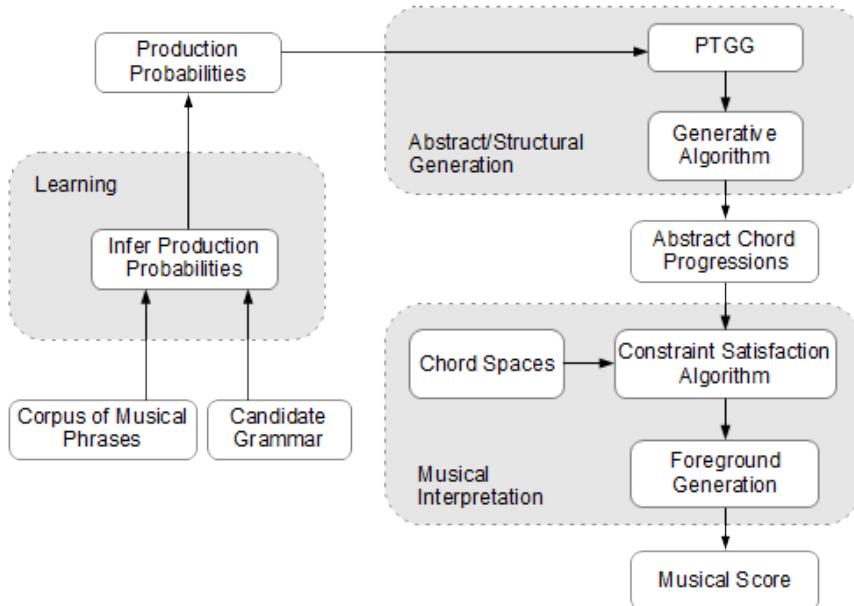
1 var CantusFirmus = require('counterpoint').CantusFirmus
2 var cantus = new CantusFirmus('G major')
3 cantus.choices() => [ 'G' ]
4 cantus.addNote('G4')
5
6 cantus.choices() => [ 'A4', 'B4', 'C5', 'D5', 'E5', 'G5',
7 'F#4', 'E4', 'G3', 'B3', 'C4', 'D4' ]
8
9 cantus.addNote('E5')
10 cantus.choices() => [ 'D5', 'C5' ]
11
12 cantus.addNote('D5')
13 cantus.choices() => [ 'E5', 'F#5', 'G5', 'A5', 'B5',
14 'C5', 'B4', 'G4', 'A4' ]
15 cantus.addNote('F#5')
16 cantus.choices() => [ 'G5', 'E5', 'F#4', 'A4', 'B4' ]
17
18 cantus.addNote('G5')
19 cantus.choices() => [ 'A5', 'B5', 'F#5', 'E5', 'G4', 'B4', 'C5', 'D5' ]
20
21 cantus.addNote('B4')
22 cantus.choices() => [ 'C5', 'D5' ]
23 cantus.addNote('C5')
24 cantus.addNote('A4')
```

```

25
26 cantus.choices()    => [ 'B4', 'D5', 'E5', 'F#5', 'A5', 'G4' ]
27 cantus.addNote('G4')
28
29 console.log(cantus.print())
30
31 G5          o
32 F#5         o
33 E5          o
34 D5          o
35 C5          o
36 B4          o
37 A4          o
38 G4          o
39     G4  E5  D5  F#5  G5  B4  C5  A4  G4

```

Kulitta - framework napisany w Haskellu przez Donya Quick przeznaczony do automatycznego i algorytmicznego komponowania muzyki. System używa generatywnych gramatyk do tworzenia abstrakcyjnej struktury muzycznej, która jest następnie stopniowo ulepszana za pomocą matematycznych modeli harmonii. Kolejno specyficzne dla stylu algorytmy zamieniają harmonie w konkretny rodzaj muzyki np. chorał, jazz czy bossa nova.



Rysunek 2.8: Struktura framerowka Kulitta

Kulitta korzysta ze specjalnego rodzaju gramatyk jakim są Probabilistic Temporal Graph Grammars (PTGG). PTGG są podobne do gramatyk bezkontekstowych, umożliwiają równoczesne generowanie harmonicznej i metrycznej struktury za pomocą sparametryzowanego alfabetu.

0.41	$T^t \rightarrow T^t$	0.64	$D^t \rightarrow D^t$
0.30	$T^t \rightarrow T^{t/2}T^{t/2}$	0.09	$D^t \rightarrow D^{t/2}D^{t/2}$
0.16	$T^t \rightarrow D^{t/2}T^{t/2}$	0.27	$D^t \rightarrow S^{t/2}D^{t/2}$
0.12	$T^t \rightarrow T^{t/2}D^{t/2}$	0.95	$S^t \rightarrow S^t$
		0.05	$S^t \rightarrow S^{t/2}S^{t/2}$

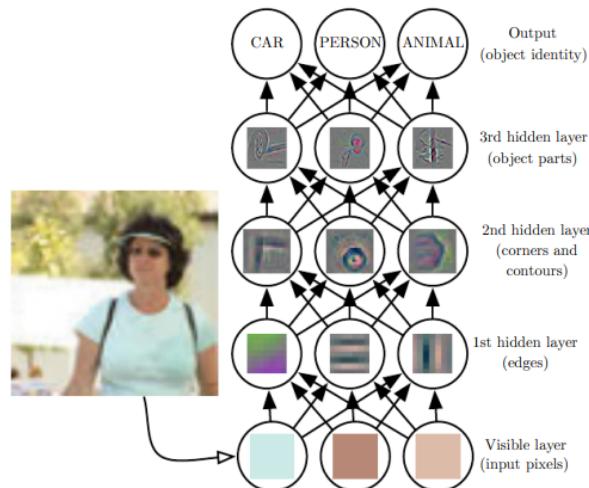
Rysunek 2.9: Przykład PTGG nad symbolami funkcji akordowych, tonika (T), dominanta (D) i subdominanta (S), sparametryzowana przez czas trwania jako indeks górnny. Przedstawione prawdopodobieństwa produkcyjne pochodzą z głównej części potrzebnej do generowania chorałów

Rozdział 3

Sieci Neuronowe

3.1 Wprowadzenie

Temat sieci neuronowych oraz uczenia maszynowego obecnie stanowi przedmiot zainteresowania wielu osób. Współczesne sieci neuronowe są kamieniem milowym w dziedzinie sztucznej inteligencji. Dzięki uczeniu maszynowemu mamy dostęp do solidnych filtrów anty spamowych, programów do rozpoznawania tekstu i glosu, szybkich i niezawodnych wyszukiwarek internetowych oraz w niedalekiej przyszłości bezpiecznymi i wydajnymi samochodami autonomicznymi. Aby poznać intuicję uczenia maszynowego postawmy problem, który polega na zbudowaniu klasyfikatora, który to na podstawie rysunku lub filmu będzie potrafił rozpoznawać obiekty. Sieci wielowarstwowe służą do budowania warstw interpretacyjnych. Każda warstwa wykorzystuje dane z poprzedniej warstwy.

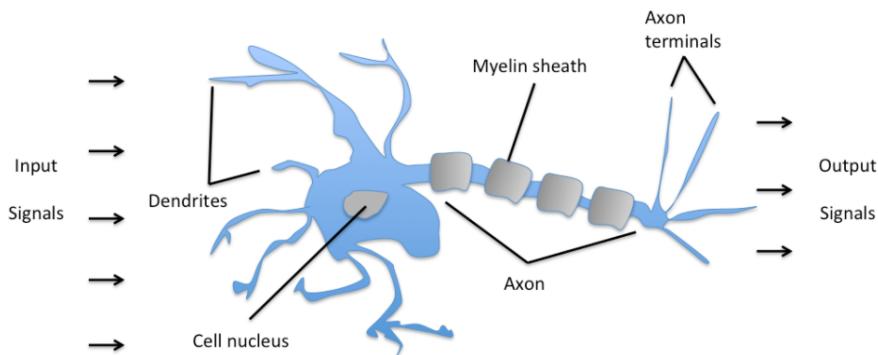


Rysunek 3.1: Schemat wielowarstwowej sieci neuronowej [22]

Taka architektura została zaproponowana, ponieważ komputer nie potrafi zrozumieć znaczenia surowych danych reprezentowanych jako zbiór pikseli. Początkowe dane trafiają do widocznej warstwy początkowej, następnie warstwy ukryte wyciągają z obrazu cechy. Mając zbiór pikseli pierwsza warstwa ukryta może zidentyfikować krawędzie na podstawie porównania jasności sąsiednich pikseli. Kolejna warstwa, która zna krawędzie występujące na obrazku może zidentyfikować narożniki i rozszerzone kontury. Kolejna warstwa mając opisy z poprzednich warstw może wykryć całe fragmenty określonych obiektów. Opis obrazu w postaci kategorii: krawędzie, narożniki, obiekty i części może być wykorzystany do rozpoznania obiektów znajdujących się na obrazie. Podobny model można zastosować w przypadku generowania muzyki bądź wykrywania gatunku muzycznego na podstawie utworu. Utwór muzyczny posiada wiele cech: metrum, wysokość dźwięku, długość dźwięku. W takim przypadku każda warstwa sieci neuronowej może być odpowiedzialna za charakteryzację każdej cechy.

3.2 Model sztucznego neuronu

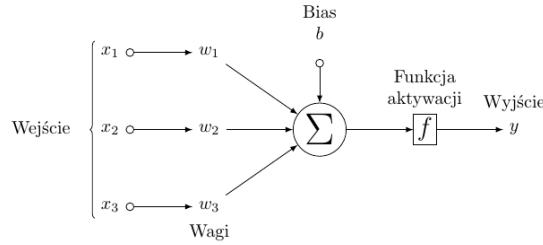
Sieci neuronowe nie są nową technologią. Pierwsze pomysły aby wykorzystać zasadę działania ludzkich neuronów w modelach matematycznych pojawiły się w latach 40 ubiegłego wieku. Biologiczny neuron jest aktywowany na podstawie danych wejściowych. Dane pochodzą z kilku powiązanych neuronów wejściowych. Neurony za pomocą dendrytów rejestrują pozytywne i negatywne informacje wyjściowe z innych neuronów i kodują je za pomocą impulsów elektrycznych przesyłanych przez akson. Następnie akson rozdziela się i dociera do setek tysięcy dendrytów innych neuronów. Pomiędzy aksonem, a wejściowymi dendrytami kolejnych neuronów znajduje się synapsa. Synapsa odpowiada za przekształcanie impulsów elektrycznych na chemiczne sygnały wpływające na dendryt następnego neuronu. Wynik uczenia jest kodowany przez same neurony. Neurony przesyłają wiadomości aksonami wtedy, gdy poziom pobudzenia jest wystarczająco wysoki.



Rysunek 3.2: Schemat biologicznego neuronu [23]

Warren McCulloch i Walter Pitts w roku 1943 opracowali model sztucznego neuronu, nazwali go **MCP** od McCulloch-Pitts[24] swoje wyniki opisali w dziele o nazwie *A Logical Calculus of the Ideas Immanent in Nervous Activity*. Naukowcy przedstawili neuron w postaci prostej bramki logicznej z binarnym wyjściem. Za pomocą neuronów MCP można zbudować prostą sieć neuronową, która realizuje funkcje logiczne AND i OR. Kilkanaście lat później w roku 1953 amerykański uczony Frank Rosenblatt zaproponował sztuczną sieć neuronową[25], którą nazwał perceptronem. Sieć zaprojektowana przez Rosenblatta mogła nauczyć się rozpoznawania ograniczonej klasy wzorców. Algorytm uczenia perceptronu polega na doborze wag dla sygnałów

wejściowych w celu narysowania liniowej granicy decyzji, która pozwala nam rozróżnić dwie liniowo rozłączne klasy. Dane wejściowe są otrzymywane za pośrednictwem odpowiedników dendrytów, następnie obliczana jest suma z uwzględnieniem wag. Jeżeli dane wyjściowe przekroczą określony próg, a wejście hamujące nie jest aktywne to neuron wygeneruje wartość pozytywną. Jeżeli wejście hamujące jest aktywne, dane wyjściowe są hamowane, co oznacza to że dane są niepoprawne. Model sztucznego neuronu jest modelem liniowym w przestrzeni n-wymiarowej gdzie n to liczba wejść neuronu, a wejścia powiązane są ze współczynnikami.



Rysunek 3.3: Model perceptronu

Wejście a można opisać za pomocą równania $a = w_1x_1 + \dots + w_nx_n$, gdzie:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \dots \\ w_n \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix}$$

Mając zdefiniowany wektor wejść oraz wag powyższe równanie można rozszerzyć do:

$$a = w_0x_0 + w_1x_1 + \dots + w_nx_n = \mathbf{w}^T \mathbf{x}$$

Opis modelu:

- Każde wejście x_n posiada wagę w_n . Wartość wejścia jest mnożona przez wartość wagi.
- Obliczana jest suma $\sum w_i x_i$
- Funkcja aktywacji $\phi(a)$ pobudza neuron w zależności od zwróconej wartości (większej od granicy θ lub mniejszej).

Prościej, model sztucznego neuronu można opisać za pomocą dwóch równań:

- $a = \sum_{j=1}^n w_j x_j$
- $y = \phi(u + w_0)$

Funkcja aktywacji ϕ jest funkcją progową, to znaczy, że wyjście funkcji jest równe 1 dla dowolnej wartości wejściowej nie większej niż θ .

$$\phi(a) = \begin{cases} 1, & \text{if } a \geq \theta \\ -1, & \text{in other case} \end{cases}$$

Model perceptronu ma trzy ważne cechy:

- Do sumy dodawany jest błąd systematyczny (ang. bias) uwzględniany przy sprawdzaniu progu. Ma to kilka celów. Po pierwsze, pozwala uwzględnić błąd statystyczny występujący w neuronach wejściowych. Po drugie, umożliwia standaryzację progów z użyciem wartości (na przykład zera) bez utraty ogólności.
- W perceptronie wagie wartości wejściowych mogą być niezależne od siebie i ujemne. Oznacza to dwie ważne rzeczy. Po pierwsze, neuronu nie trzeba wielokrotnie wiązać z wejściem, aby zwiększyć znaczenie danego wejścia. Po drugie, wejście z dendrytu może mieć wpływ hamujący, jeśli przypisana jest mu ujemna waga.
- Opracowanie perceptronu dało początek algorytmom uczącym się optymalnych wag na podstawie zbioru danych wejściowych i wyjściowych

Algorytm uczenia się Perceptronu

1. Stwórz wektor wag składający się z liczb z przedziału 0, 1
2. Dla każdej wartości treningowej $x^{(i)}$:
 - (a) Oblicz wartość wyjściową \hat{y}
 - (b) Zaktualizuj wag

Wartość wyjściowa \hat{y} to etykieta klasy do której zostało przyporządkowane dane wejście. Aktualizację każdej wagi w_j w wektorze wag \mathbf{w} można zapisać jako:

$$w_j := w_j + \Delta w_j$$

Wartość Δw_j , która służy do aktualizacji wag obliczana jest według tak zwanej reguły uczenia perceptronu:

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

Gdzie η to pewna stała uczenia, zwykle wybierana z przedziału pomiędzy 0, 1. $y^{(i)}$ jest etykietą i-tej próbki szkoleniowej, a $\hat{y}^{(i)}$ jest przewidywaną etykietą klasy. Wszystkie wagi w wektorze wagowym są aktualizowane jednocześnie.

Definicja 3.2.1. Proces uczenia neuronu wymaga zdefiniowania hipotezy h_θ . Dla danych wejściowych $x^{(i)}$ predykcje można opisać za pomocą $h_\theta(x^{(i)})$

Definicja 3.2.2. Funkcja straty:

$$L : (z, y) \in \mathbb{R} \times Y \rightarrow L(z, y) \in \mathbb{R} \quad (3.1)$$

Funkcja przyjmuje dane wejściowe z oraz ich przewidywaną wartość y . Funkcja zwraca liczbę mówiącą o tym jak bardzo dane z oraz y różnią się między sobą.

Definicja 3.2.3. Funkcja kosztu J jest używana do zebrania informacji na temat efektywności procesu uczenia się. Do jej zdefiniowana używana jest funkcja straty L .

$$J(\theta) = \sum_{i=1}^m L(h_\theta(x^{(i)}), y^{(i)}) \quad (3.2)$$

W uczeniu nadzorowanym występują różne funkcje kosztu. Zostały one pokazane w tabeli poniżej.

Tabela 3.1: Zestawienie funkcji kosztu

MSE	Strata logistyczna	Utrata zawiasów	Entropia krzyżowa
$\frac{1}{2}(y - z)^2$	$\log(1 + \exp(-yz))$	$\max(0, 1 - yz)$	$[-y\log(z) + (1 - y)\log(1 - z)]$
Regresja liniowa	Regresja logistyczna	SVM	Sieci neuronowe

Jeżeli funkcja kosztu J jest różniczkowalna do jej minimalizacji można użyć metody największego spadku gradientowego. Celem metody spadku gradientowego jest znalezienie takich wag, które minimalizują funkcję kosztu. Wartość $\frac{1}{2}$ na początku wzoru dodana jest dla ułatwienia obliczeń.

Twierdzenie 3.2.1. Uczenie sieci neuronowej polega na minimalizacji funkcji kosztu.

Twierdzenie 3.2.2. Jeśli zbiór danych jest liniowo separowalny, a współczynnik szybkości uczenia η jest wystarczająco mały to algorytm uczenia perceptronu jest zbieżny.

Twierdzenie 3.2.3. Jeśli zbiór danych nie jest liniowo separowalny to algorytm zbiega lokalnie do minimalnego błędu średniokwadratowego.

3.3 Metoda spadku gradientowego

Celem algorytmu spadku gradientowego jest minimalizacja funkcji kosztu $J(w)$. Minimalizacja funkcji kosztu pozwala na odpowiedni dobór wag. Ogólnie mówiąc minimalizacja funkcji polega na schodzeniu po jej powierzchni w oparciu o jej nachylenie. Aktualizacja wag polega na wykonaniu kroku w kierunku przeciwnym do gradientu $\nabla J(w)$ funkcji kosztu $J(w)$.

$$w := w + \Delta w$$

Gdzie zmiana wagi Δw zdefiniowana jest jako:

$$\Delta w = -\eta \nabla J(w)$$

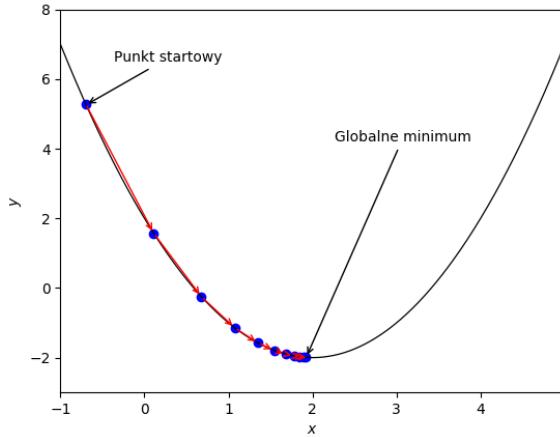
gdzie η to stała uczenia - szybkość schodzenia w dół. Aby obliczyć gradient funkcji kosztu potrzeba wyliczyć pochodną cząstkową funkcji kosztu dla danej wagi w_j :

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \quad (3.3)$$

Zatem reguła aktualizacji wag może wyglądać w następujący sposób:

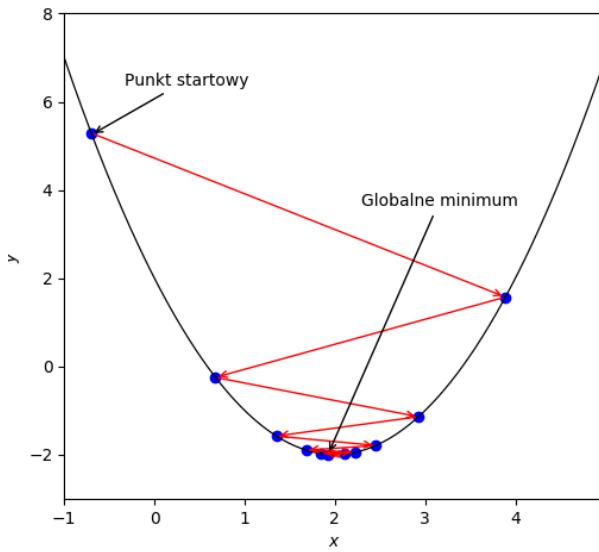
$$\delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \quad (3.4)$$

Przykład 3.3.1. Dana jest funkcja jednej zmiennej $y = f(x) = x^2 - 4x + 2$, funkcja jest ciągła więc do znalezienia jej globalnego minimum można użyć algorytmu spadku gradientowego, rezultat został pokazany poniżej.



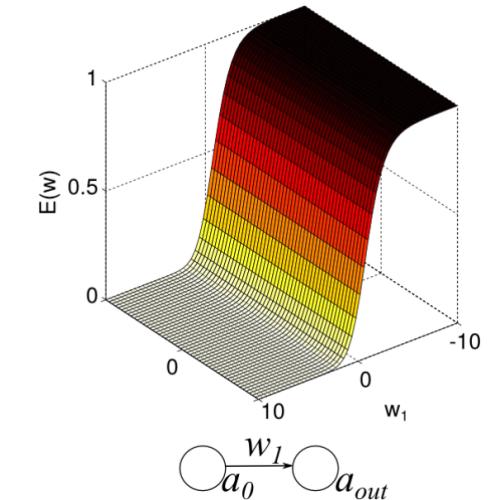
Rysunek 3.4: Metoda spadku gradientowego

Poniżej wynik działania algorytmu za dużą stałą uczenia η :

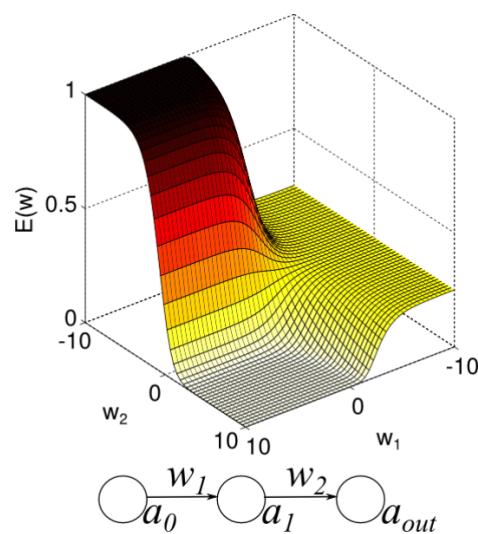


Rysunek 3.5: Za duży współczynnik uczenia

Przykład 3.3.2. Funkcję wielu zmiennych można również minimalizować za pomocą spadku gradientowego, pod warunkiem, że taka funkcja jest ciągła. Wyniki można obserwować na płaszczyźnie 3D.



Rysunek 3.6: Przestrzeń błędów dla sieci jednowarstwowej



Rysunek 3.7: Przestrzeń błędów dla sieci dwuwarstwowej

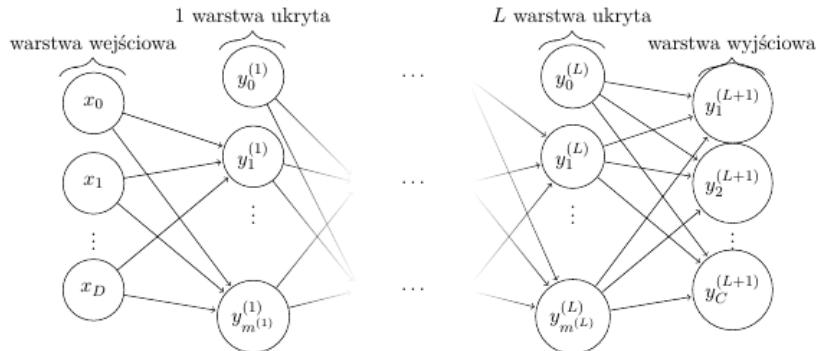
3.4 Wielowarstwowe sieci neuronowe

Sieci, które składają się przynajmniej z dwóch neuronów muszą mieć określoną architekturę, ponieważ wyniki zwracane przez te neurony mogą być wejściami dla innych neuronów. Takie sieci można podzielić na dwa rodzaje:

- skierowane (ang. feed-forward)

- rekurencyjne (ang. recurrent)

W sieciach wielowarstwowych wyróżnia się podzbiór neuronów akceptujących dane wejściowe, nazywa się je jednostkami wejściowymi oraz podzbiór neuronów, których wyjście jest również wyjściem całej sieci, są to jednostki wyjściowe. Pozostałe neurony nazywa się ukrytymi. Sieci wielowarstwowe są w stanie przeprowadzić nieliniowy podział danych. Dane wejściowe każdej warstwy pochodzą z warstwy poprzedniej, a dane wejściowe poszczególnych warstw są danymi wejściowymi dla warstw następnych. Sieci skierowane są nazywane również sieciami jednokierunkowymi ponieważ związane to jest z tym, że dane płyną tylko w jedną stronę sieci, nie występują w niej cykle. Sieć wielowarstwową można powiązać ze skierowanym grafem acyklicznym opisującym, jak funkcje są ze sobą powiązane. Założymy sytuację, w której mamy trzy funkcje $f^{(1)}, f^{(2)}, f^{(3)}$ połączone w łańcuch, tworząc $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. Sieci wielowarstwowe najczęściej reprezentują się właśnie w postaci takich struktur łańcuchowych. W takiej strukturze mówimy, że $f^{(1)}$ jest pierwszą warstwą sieci, $f^{(2)}$ to druga warstwa sieci. Całkowita długość łańcucha funkcji daje w wyniku głębokość modelu. Poniżej został przedstawiony model wielowarstwowej sieci neuronowej, który zawiera $(L+1)$ -warstw z D wejściami oraz C wyjściami.



Rysunek 3.8: Model wielowarstwowej sieci neuronowej

Wagi wielowarstwowych sieci neuronowych pokazują następujące ważne twierdzenia:

Twierdzenie 3.4.1. Każda funkcja boolowska może być reprezentowana przez sieć z jedną warstwą ukrytą, ale może wymagać wykładniczej liczby jednostek ukrytych.

Twierdzenie 3.4.2. Każda ograniczona funkcja ciągła może być aproksymowana z dowolnie małym błędem przez sieć z jedną warstwą ukrytą [Cybenko 1989; Hornik et al. 1989]

Twierdzenie 3.4.3. Dowolna funkcja może być aproksymowana z dowolną dokładnością przez sieć z dwoma warstwami ukrytymi [Cybenko 1988]

Sieć neuronowa złożona z jednej warstwy ma ograniczenia, ponieważ może tylko realizować problemy liniowo separowalne. Bardzo klasycznym nieliniowym problemem jest funkcja XOR (alternatywa wykluczająca). Funkcja działa na dwóch wartościach bitowych x_1 i x_2 . Funkcja zwraca 1 wtedy gdy dokładnie jedna z dwóch wartości jest równa 1. Ograniczenia sieci jednowarstwowych opisali Marvin Minsky i Seymour Papert

w książce po tytulem Perceptrons: An Introduction to Computational Geometry. W swojej książce przedstawiły funkcję XOR jako problem nieliniowy, z którym pojedynczy perceptron lub jednowarstwowa sieć sobie nie poradzi.

p	q	p	XOR	q
0	0		0	
0	1		1	
1	0		1	
1	1		0	

Tabela 3.2: Tabela prawdy dla funkcji XOR

Funkcja XOR przedstawiona na wykresie dowodzi, że jej dane wyjściowe nie są liniowo separowalne w przestrzeni dwuwymiarowej. To znaczy, że nie istnieje jedna prosta, która pozwalałaby na rozdzielenie elementów klasy pozytywnej i negatywnej. Punkty można poprawnie rozdzielić przy pomocy więcej niż jednej prostej. Taką możliwość daje architektura sieci wielowarstwowej. Aby skonstruować sieć, która będzie w stanie nauczyć się problemu XOR wystarczy sieć dwuwarstwowa z jedną warstwą ukrytą. Sieć na wejściu przyjmuje dwie wartości, które potrzebne są do obliczenia funkcji XOR. W tak opracowanej sieci każdy węzeł tworzy hiperpłaszczyznę. Wszystkie hiperpłaszczyzny są łączone przez końcowy perceptron. W efekcie otrzymujemy podprzestrzeń, która jest zbiorem wypukłym. Kolejnym ważnym problemem jest odpowiedni dobór wag, które trzeba wyliczyć automatycznie. Wagi powinny być tak dobrane aby można było je wykorzystać do klasyfikowania i prognozowania danych innych niż te pierwotne wejściowe.

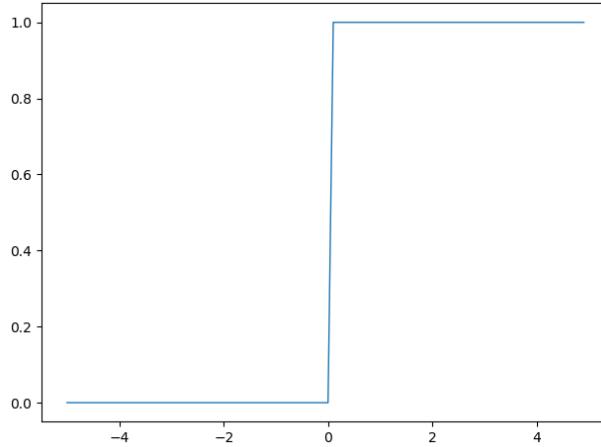
3.5 Funkcje aktywacji

Sieci neuronowe pozwalają na stosowanie szerokiego zakresu funkcji aktywacji. Wybór funkcji aktywacji zależy głównie od tego jaki problem sieć neuronowa powinna rozwiązać. W sieciach wielowarstwowych najczęściej stosowane są funkcje nieliniowe, ponieważ ich nieliniowość jest ważną cechą potrzebną do sprawnego uczenia. Funkcje aktywacje można podzielić na:

- nieliniowe
- liniowe
- skoku jednostkowego (progowa)

Ostatni rodzaj funkcji aktywacji jest najbardziej podstawową aktywacją neuronu. Funkcja ta umożliwia otrzymanie na wyjściu sieci informacji postaci TAK - NIE.

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \quad (3.5)$$



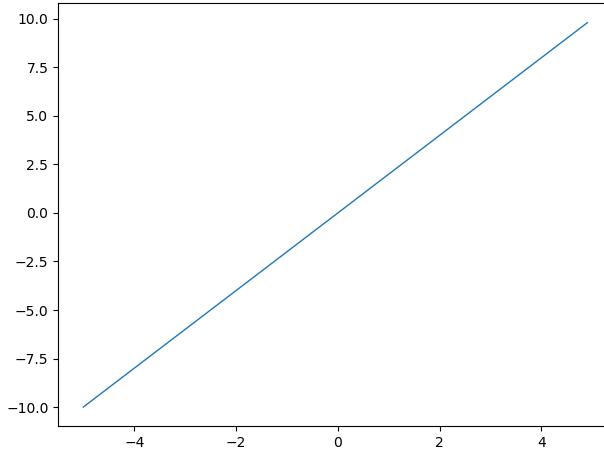
Rysunek 3.9: Funkcja skoku jednostkowego

Jeżeli sieć neuronowa ma spełniać zadanie klasyfikatora binarnego (zwraca TAK lub NIE), to funkcja progowa będzie idealną funkcją aktywacji. Problem może pojawić się jeżeli będziemy chcieli wprowadzić więcej klas klasyfikacji. Wtedy jeżeli więcej niż jeden neuron zostanie aktywowany to reszta neuronów automatycznie wyprowadzi 1 z funkcji progowej. Dla większej ilości klas byłoby lepiej, gdy funkcja aktywacji nie była binarna tylko zwracała wartości w postaci np. 20% dla jednej klasy 50% dla drugiej i reszta dla ostatniej.

3.5.1 Funkcja liniowa

W przypadku funkcji liniowej sygnał wyjściowy przyjmuje postać:

$$y = cx \quad (3.6)$$



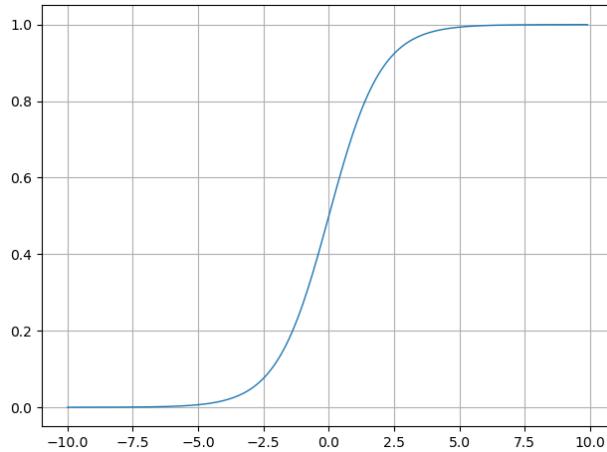
Rysunek 3.10: Funkcja liniowa

W przypadku liniowej funkcji aktywacji aktywacja jest proporcjonalna do sygnału wejściowego neuronu (który jest sumą ważoną). Takie podejście daje szereg różnych aktywacji, nie ma w tym przypadku aktywacji binarnej. Przy treningu neuronu z liniową funkcją aktywacji może wystąpić problem, ponieważ pochodna dla tej funkcji jest stała. Oznacza to, że gradient nie ma związku z x . Jeżeli gradient jest stały to metoda spadku gradientowego również zwróci stały gradient. Jeżeli zostanie użyta wsteczna propagacja błędu to zmiany wprowadzone przez algorytm również będą stałe i nie będą zależne od zmiany wejścia. Bez względu na to z ilu warstw jest zbudowana sieć neuronowa to ostatnia funkcja aktywacji w ostatniej warstwie jest liniową funkcją wejścia pierwszej warstwy.

3.5.2 Funkcja sigmoidalna i tanh

Popularną nieliniową funkcją aktywacji jest funkcja sigmoidalna, której kształt przypomina literę S .

$$f(x) = \frac{1}{1 + e^{-x}} \quad (3.7)$$



Rysunek 3.11: Wykres funkcji sigmoidalnej

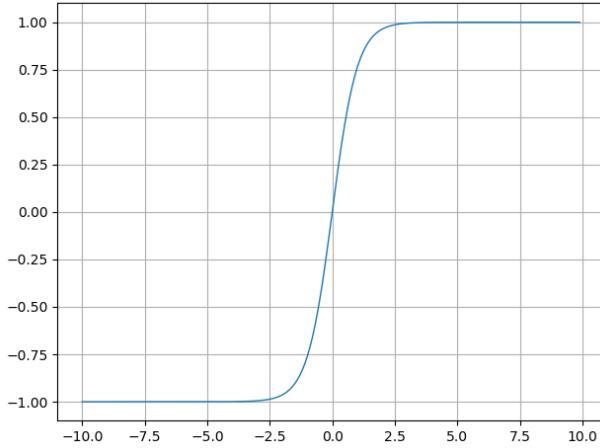
Bardzo ważną cechą funkcji sigmoidalnej jest fakt, że jest różniczkowalna. Warto zauważyć, że pomiędzy wartościami -2 do 2 wartości z osi y są bardzo strome. Każda niewielka zmiana wartości x w tym regionie spowoduje znaczne zmiany wartości y . Ten fakt można wykorzystać do doprowadzania wartości y do jednego z końców krzywej. Wyjście funkcji zawsze będzie znajdować się w zakresie $(0, 1)$. W przypadku gdy wartości zwieracane przez funkcję będą bardzo blisko krańców krzywej to gradient będzie bardzo mały, na tyle mały że nie będzie można dokonać zmiany wag. W takim przypadku może dojść do sytuacji, że sieć nie będzie w stanie się uczyć lub proces uczenia będzie bardzo wolny. Taki problem określa się mianem znikającego gradientu[26].

Kolejną używaną funkcją aktywacji w sieciach neuronowych jest funkcja \tanh :

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad (3.8)$$

Powyzsze równanie można sprowadzić do wcześniejszej wspomnianej funkcji sigmoidalnej:

$$\tanh(x) = 2\text{sigmoid}(2x) - 1 \quad (3.9)$$



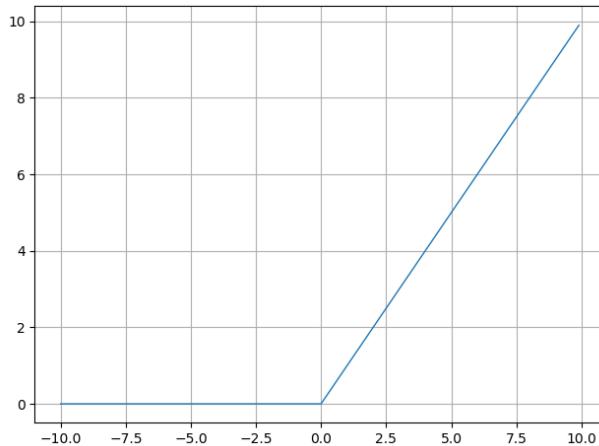
Rysunek 3.12: Wykres funkcji \tanh

Charakterystyka funkcji \tanh jest podobna jak w przypadku funkcji sigmoidalnej. W przypadku funkcji \tanh gradient jest znacznie silniejszy niż w przypadku funkcji sigmoidalnej. Również w przypadku funkcji \tanh występuje problem znikającego gradientu.

3.5.3 Funkcja ReLu

$$f(x) = \max(0, x) \quad (3.10)$$

Funkcja *Rectified Linear Units* jest zalecana do zastosowania w większości jednokierunkowych sieci neuronowych.



Rysunek 3.13: Funkcja ReLu

Funkcja *ReLU* jest funkcją nielinową, która zwraca wartości z przedziału $< 0, \infty$. Z wykresu funkcji można odczytać, że funkcja *ReLU* może rzadko aktywować neuron. Dla ujemnych wartości x funkcja zwraca wartość 0, co wyraźnie widać na wykresie w postaci poziomej linii. W przypadku aktywacji ten obszar funkcji *ReLU* dla gradientu będzie wynosił 0. Oznacza to, że neurony wchodzące w ten obszar przestaną reagować na zmiany. O tym zjawisku mówi się jako o "umieraniu neuronu" lub "problem umierającego ReLu". Ten problem może spowodować sytuacje, że kilka neuronów po prostu umrze i nie będzie reagować przez co część sieci do niczego się nie przyda. Funkcja *ReLU* jest znacznie mniej kosztowna obliczeniowo niż *tanh* i *sigmoid*. Warto ją rozważyć w przypadku projektowania głębszej sieci neuronowej.

Podsumowując, wymagane cechy funkcji aktywacji to:

- ciągłe przejście pomiędzy swoją wartością maksymalną a minimalną (funkcja musi być ciągła)
- łatwa do obliczenia i ciągła pochodna

3.6 Wsteczna propagacja błędu

W sieciach jednokierunkowych gdzie na wejściu przyjmowany jest wektor x dane wejściowe podają początkową informację, która przepływa w górę do ukrytych jednostek w każdej sieci, w rezultacie otrzymujemy \hat{y} . Jedną z głównych zalet sieci neuronowych jest to, że nie musimy ręcznie podawać wag. Można te wagi wytrenować, czyli znaleźć ich w przybliżeniu optymalny zestaw za pomocą algorytmu wstecznej propagacji błędu. Autorami algorytmu są D. E. Rumelhart, G. E. Hinton, i R. J. Williams, swoje wyniki opublikowali na łamach prestiżowego magazynu *Nature*[27] Po mimo tego, że sam algorytm ma ponad 30 lat to jest jednym z najważniejszych algorytmów stosowanych do efektywnego uczenia sieci neuronowych. O samym algorytmie można myśleć jak o wydajnej metodzie, która służy do obliczania pochodnych cząstkowych funkcji kosztu w wielowarstwowych sieciach neuronowych. Problem w wielowarstwowych sieciach polega na tym, że mamy do czynienia z bardzo dużą liczbą współczynników, które trzeba wyznaczyć. W przeciwnieństwie do pozostałych modeli funkcja kosztu w sieci wielowarstwowej nie jest wypukła ani gładka w odniesieniu do parametrów. Istnieje więc wiele nierówności w wielowymiarowej przestrzeni, które trzeba zoptymalizować tak aby zna-

leźć globalne minimum. W algorytmie wstecznej propagacji błędów jest wykorzystywana reguła łańcuchowa, której zadaniem jest obliczenie pochodnej funkcji złożonej.

Fakt 1. Algorytm propagacji wstecznej działa dla dowolnego grafu skierowanego bez cykli.

Twierdzenie 3.6.1. Algorytm propagacji wstecznej zbiega lokalnie do minimalnego błędu średniokwadratowego.

$$\frac{d}{dx} [f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx} \quad (3.11)$$

Reguły łańcuchowej możemy użyć do długiej złożonej funkcji, funkcja ta może reprezentować wielowarstwową sieć neuronową. Założymy, że sieć składa się z pięciu warstw: $F(x) = f(g(g(u(v(x)))))$. Po zastosowaniu reguły łańcuchowej możemy policzyć pochodne ze wzoru:

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx} \quad (3.12)$$

W algorytmie wstecznej propagacji wykorzystuje się pochodną funkcji aktywacji. Pochodna informuje o tym czy dla zadanego zestawu wag funkcja aktywacji jest odpowiednio pobudzona. Dzięki temu można dowiedzieć się w którym kierunku i jak bardzo zaktualizować wagi.

Poniżej zostanie przedstawiony proces uczenia wielowarstwowej sieci neuronowej z użyciem algorytmu wstecznej propagacji błędu na przykładzie trójwarstwowej sieci neuronowej o dwóch wejściach i jednym wyjściu. Przepływ w takiej sieci można wyrazić za pomocą listy kroków:

- Warstwa wejściowa Z jest wyrażona za pomocą równania $Z = A * W$, gdzie A to macierz danych wejściowych, a W to macierz wag dla danych wejściowych
- Iloczyn oby macierzy trafia do funkcji aktywacji A , operację można wyrazić za pomocą wzoru $A = \phi(Z)$
- operacja odbywa się w warstwie ukrytej
- Wynik funkcji aktywacji trafia do warstwy wyjściowej i jest wymnażany przez macierz wag $Z = A * W$
- Obliczone dane w warstwie wyjściowej trafiają do funkcji aktywacji $A = \phi(Z)$, wynik przypisywany jest do z

W taki sposób została dokonana propagacja sygnału przez całą sieć w kierunku od lewej strony do prawej. W algorytmie wstecznej propagacji błędu błąd jest propagowany z prawej do lewej. Proces zaczyna się od obliczenia błędu w warstwie wyjściowej:

$$\delta = z - y \quad (3.13)$$

Sygnał wyjściowy sieci porównywany jest z oczekiwana wartością sygnału wyjściowego. Różnica obu wartości nazywana jest sygnałem błędu δ neurona warstwy wyjściowej. Neurony, które znajdują się w warstwach ukrytych nie są w stanie bezpośrednio określić swojego błędu, ponieważ nie są znane wartości oczekiwane sygnałów wyjściowych z tych neuronów. Dzięki wstecznej propagacji błędu błąd jest rzutowany wstecz od prawej do lewej strony. Dzięki temu można obliczyć błąd w warstwie ukrytej.

$$\delta^{(hidden)} = \delta(W)^T \frac{\partial \phi(Z^{(hidden)})}{\partial Z^{(hidden)}} \quad (3.14)$$

uwzględniając pochodną funkcji aktywacji powyższe wyrażenie można zapisać jako:

$$\delta^{(hidden)} = \delta(W)^T (a^{(h)} (1 - a^{(h)})) \quad (3.15)$$

W dalszej kolejności trzeba przechować pochodną cząstkową każdego węzła każdej warstwie oraz błąd węzła w następnej warstwie. Należy obliczyć $\delta_{i,j}^{(l)}$ dla każdej próbki ze zbioru treningowego.

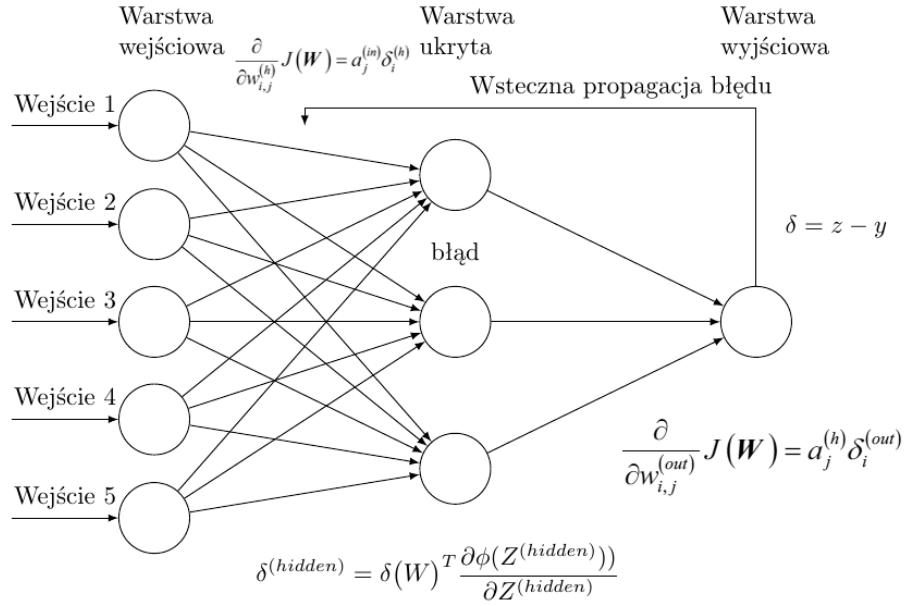
$$\Delta^{(hidden)} = \delta^{(hidden)} + (A^{(input)})^T \delta^{(hidden)} \quad (3.16)$$

$$\Delta = \delta + (A^{(hidden)})^T \delta \quad (3.17)$$

Po wyliczeniu spadku gradientowego należy zaktualizować wagi wykonując przeciwny krok w kierunku gradientu dla każdej warstwy:

$$W^{(l)} := W^{(l)} - \eta \Delta^{(l)} \quad (3.18)$$

Cały proces można przedstawić za pomocą ilustracji poniżej i przedstawić za pomocą listy kroków:



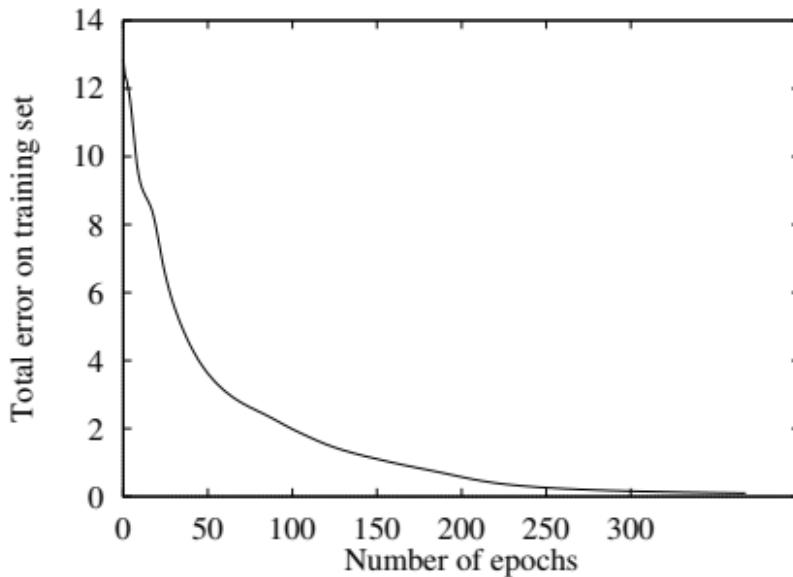
Rysunek 3.14: Ilustracja wsteczonej propagacji błędu

Cały proces można sprowadzić do listy kroków:

- Pobierz dane treningowe
- Przeprowadź propagacje sygnału w celu obliczenia błędu
- Przeprowadź propagację wsteczną aby uzyskać gradienty

- Użyj gradientów aby zaktualizować wagi sieci

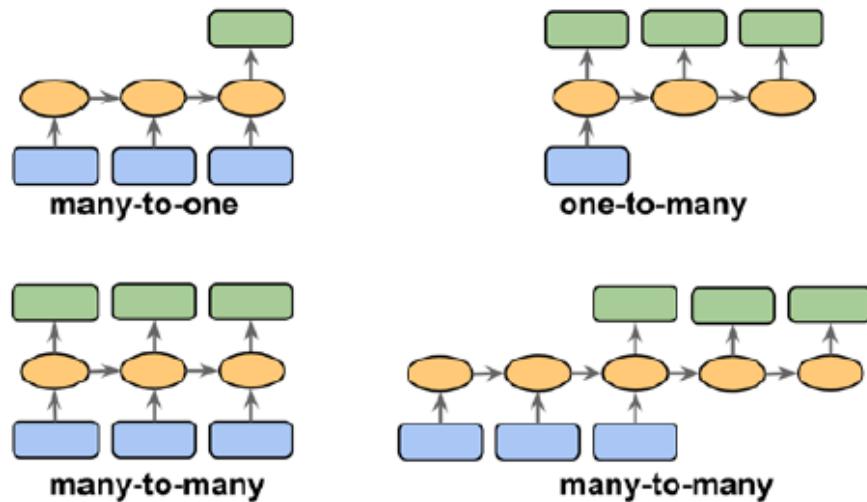
Fakt 2. Zauważmy, że możemy określić pojęcie epoki jako jeden cykl podczas którego wszystkie obiekty treningowe poprawiają wagi, na koniec wyliczany jest błąd sumaryczny całego zbioru treningowego. Algorytm uczenia zatrzymuje się, kiedy błąd przestaje maleć.



Rysunek 3.15: Proces uczenia sieci polega na minimalizacji błędu

3.7 Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe (ang. recurrent neural networks, RNN)(Rumenhart 1986) to rodzina sieci neuronowych służąca do pracy z danymi sekwencyjnymi. Sieci rekurencyjne mają zastosowanie w tłumaczeniu tekstu, rozpoznawaniu obrazu, rozpoznawaniu mowy czy generowaniu muzyki, ponieważ wszystkie te mechanizmy potrzebują do działania sekwencji. Jedną z pierwszych koncepcji systemów uczących się i modeli statystycznych była możliwość współdzielenia parametrów w różnych częściach modelu. Takie podejście pozwala na rozszerzenie modelu i zastosowanie go do przykładów o różnych postaciach. Istnieje kilka różnych kategorii przetwarzania danych sekwencyjnych, które można podzielić ze względu na dane wejściowe i wyjściowe. Takiego podziału dokonał Andrej Karpathy, który opisał w artykule *The Unreasonable Effectiveness of Recurrent Neural Networks*[28] Rysunek poniżej obrazuje różne zależności między danymi wejściowymi, a wyjściowymi.



Rysunek 3.16: Zależności pomiędzy wejściem, a wyjściem w rekurencyjnych sieciach neuronowych [29]

W przypadku kiedy dane wejściowe zawierają sekwencje istnieje kilka różnych przypadków modeli, które to zwrócią różne wyjścia. Można je podzielić na trzy główne kategorie.

- Wiele do jednego - Dane wejściowe to sekwencja, wyjście to wektor o stałym rozmiarze. Dla przykładu na wejściu jest fragment tekstu, a na wyjściu jedno słowo np. emocija opisująca fragment tekstu, inny przykład to sekwencja nut na wejściu, a na wyjściu nazwa gatunku muzycznego pasującego do zadanej sekwencji.
- Jeden do wielu - Dane wejściowe nie są sekwencją, natomiast dane wyjściowe składają się z sekwencji. Przykład to opisywanie obrazów, wejście to obraz, a na wyjściu opis obrazu.
- Wiele do wielu - Dane wejściowe i wyjściowe składają się z sekwencji. Przykład to tłumaczenie jednego języka na inny, generowanie nowego utworu muzycznego na podstawie innego.

W książce *Deep Learning* Yan Goodfellow dokonał podobnego podziału:

- sieci rekurencyjne, które generują wartości wynikowe w każdym kroku czasowym i mają rekurencyjne połączenia między ukrytymi jednostkami
- sieci rekurencyjne, które generują wartości wynikowe w każdym kroku czasowym i mają rekurencyjne połączenia jedynie między wejściem w jednym kroku czasowym, a jednostkami ukrytymi w kolejnym kroku czasowym
- sieci rekurencyjne z rekurencyjnymi połączeniami między ukrytymi jednostkami, które odczytują całą sekwencję, a następnie generują jedno wyjście

Innym historycznym przykładem sieci rekurencyjnej jest sieci **Elmana** (Elman, 1990) - sieć złożona z trzech podstawowych warstw (wejściowej, ukrytej, wyjściowej) oraz dodatkowej warstwy, której połączenia wejściowe są połączeniami wejściowymi warstwy ukrytej. **Sieci Jordana** posiadają podobną strukturę do

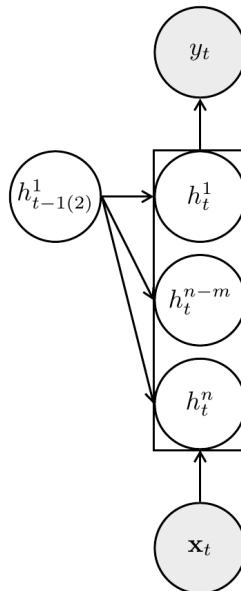
sieci Elmana, połączenia wejściowe do warstwy kontekstowej są połączeniami wyjściowymi z warstwy wyjściowej. **Sieci Hopfielda** nie posiadają wyróżnionych warstw, każdy neuron połączony jest ze wszystkimi neuronami w sieci, poza sobą, neurony nie mają pętli, każda para neuronów ma połączenie symetryczne.

Rekurencyjną sieć neuronową można przedstawić za pomocą poniższego wzoru, który oblicza sekwencje wektorów x stosując zasadę rekurencji dla każdego kroku czasowego:

$$h_t = f_W(h_{t-1}, x_t) \quad (3.19)$$

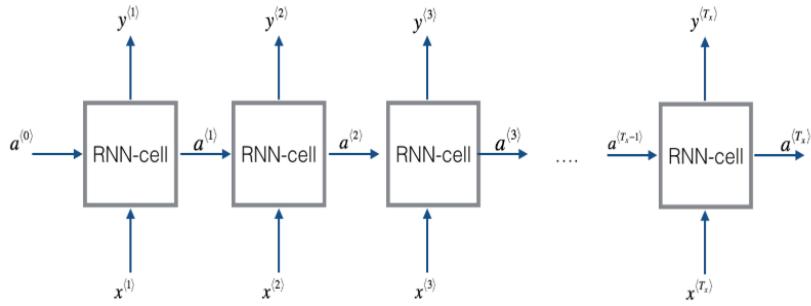
- h_t - nowy stan
- f_W - funkcja aktywacji z parametrem W
- h_{t-1} - poprzedni stan
- x_t - wektor wejścia w kroku czasowym t

Powyższy wzór można przedstawić za pomocą schematu:



Rysunek 3.17: Schemat rekurencyjnej sieci neuronowej z jedną warstwą ukrytą

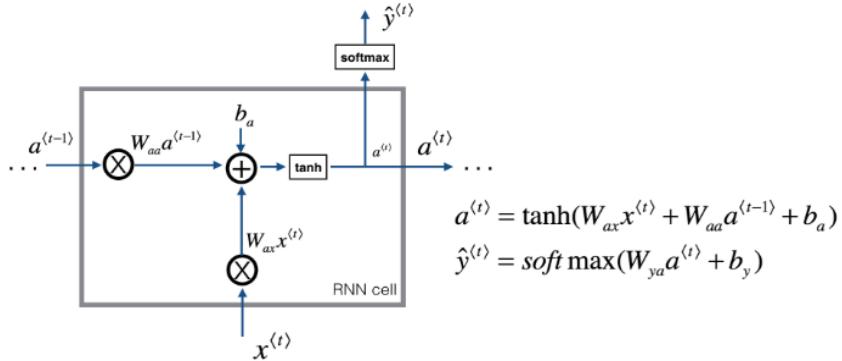
Model sieci rekurencyjnej można rozwinać w czasie, wtedy otrzymamy:



Rysunek 3.18: Rekurencyjna sieć neuronowa rozwinięta w czasie

Sekwencja wejściowa $x = (x^1, x^2, \dots, x^{T_x})$ jest aplikowana do sieci w T_x krokach czasowych. Na wyjściu sieć zwraca sekwencję $y = (y^1, y^2, \dots, y^{T_x})$.

Rekurencyjną sieć neuronową można postrzegać jako powtarzanie pojedynczej komórki. Poniższy rysunek opisuje operacje dla pojedynczego kroku czasowego komórki w rekurencyjnej sieci neuronowej.



Rysunek 3.19: Podstawowa komórka w rekurencyjnej sieci neuronowej.

Powyższa komórka na wejściu otrzymuje x^t oraz a^{t-1} z poprzedniego ukrytego stanu, na wyjściu komórka zwraca a^t , wyjście będzie wykorzystane jako wejście do następnej komórki oraz zostanie użyte do predykcji \hat{y}^t

Każda krawędź skojarzona jest z macierzą wag. Macierz wag jest niezależna od danego kroku czasowego t . Macierze wag, które występują na rysunku powyżej:

- W_{ax} - macierz wag pomnożona przez wektor wejściowy
- W_{aa} - macierz wag pomnożona przez stan ukryty
- W_{ya} - macierz wag pomiędzy warstwą ukrytą, a warstwą wyjściową

Trening rekurencyjnej sieci neuronowej odbywa się z wykorzystaniem algorytmu *Backpropagation Through Time*[30]. Algorytm bazuje na metodzie spadających gradientów. Jego główną ideą jest to aby całkowity błąd L był sumą wszystkich funkcji błędu w czasie $t = 1$ do $t = T$.

$$L = \sum_{t=1}^T L^{(t)} \quad (3.20)$$

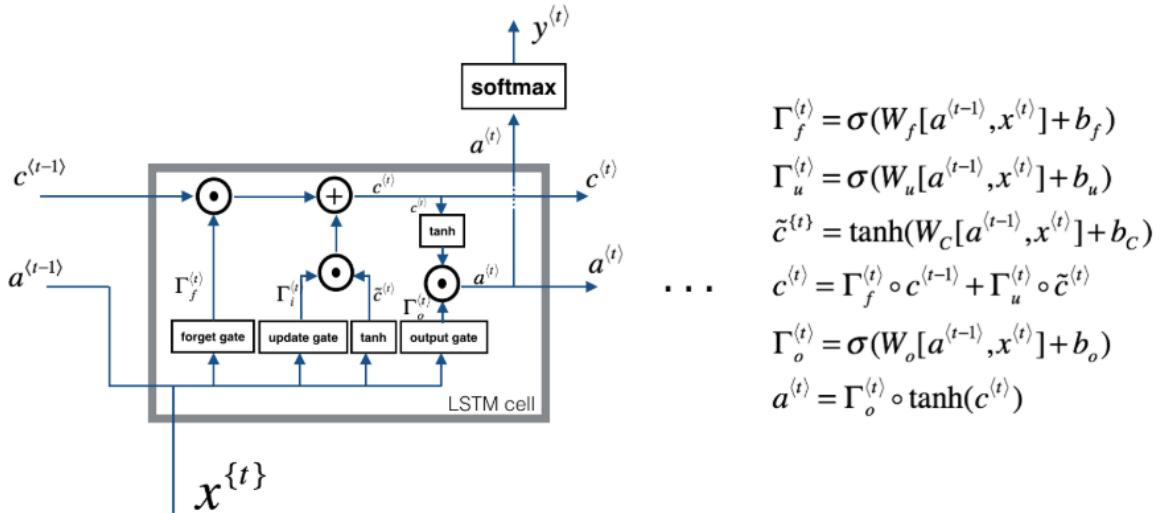
Sieci rekurencyjne stosuje się aby zachowywać długotrwałe zależności w czasie. Jednak niesie to ze sobą pewne problemy, ponieważ im więcej danych nagromadzonych w poszczególnych krokach czasowych to istnieje duże ryzyko, że gradienty obliczane z wykorzystaniem algorytmu BPTT będą się gromadzić. Takie zjawisko może doprowadzić do problemu zanikającego gradientu lub jego eksplozji. Problem ten został dokładnie opisany przez R. Pascanu, T. Mikolov, i Y. Bengio w dziele *On the difficulty of training recurrent neural networks*[31].

Na przestrzeni lat zostały opracowane dwa sposoby, które rozwiążają ten problem:

- Truncated backpropagation through time (TBPTT)[32]
- Long short-term memory (LSTM)[33]

3.7.1 Pamięć Long Short-Term

Współcześnie najbardziej efektywne modele sekwencyjne, które stosowane są w praktyce nazywane są bramkowymi sieciami RNN. Do tych modeli zaliczana jest architektura długiej pamięci krótkoterminowej (ang. long short-term memory). Pierwsza koncepcja architektury LSTM została przedstawiona przez dwóch niemieckich naukowców S. Hochreiter'a oraz J. Schmidhuber'a w 1997 [34]. Głównym blokiem konstrukcyjnym LSTM jest komórka pamięci, która reprezentowana jest przez ukrytą warstwę w rekurencyjnej sieci neuronowej. Sieci rekurencyjne z blokiem LSTM nazywamy sieciami LSTM. Nad poprawieniem pierwotnej koncepcji LSTM pracowało wiele osób przez kilka lat, wyniki poprawek zostały opisane w pracy [35]. Sieci LSTM zostały zaprojektowane w celu uniknięcia problemu długotrwałej zależności. Trening sieci LSTM pozwala na zapamiętanie długotrwałych zależności pomiędzy danymi wejściowymi, a wyjściowymi wynikami zwracanymi przez sieć. Struktura komórki LSTM może przypominać pamięć komputera, ponieważ może odczytywać, zapisywać i usuwać informacje za pomocą odpowiednich bramek. Komórka decyduje, czy przechować lub usunąć informacje w zależności od przypisanej wagi do informacji. Wagi dobierane są w procesie uczenia, z upływem czasu komórka LSTM dowiaduje się, które informacje są ważne, a które nie. Struktura komórki LSTM została przedstawiona na rysunku poniżej.



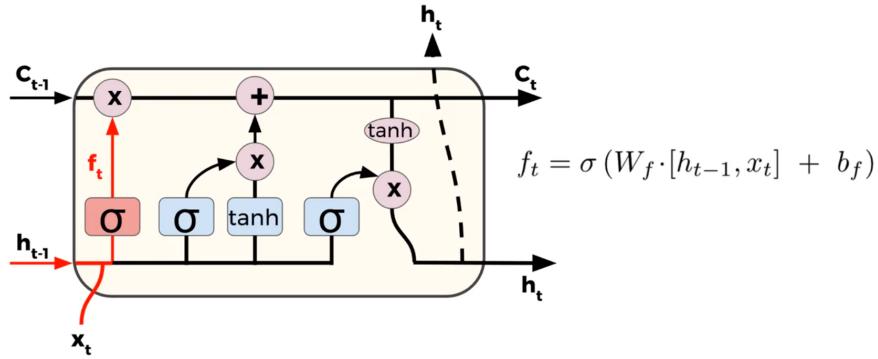
Rysunek 3.20: Komórka LSTM

Głównym rdzeniem komórki LSTM jest wyprowadzenie, które na wejściu przyjmuje pamięć z poprzedniego kroku czasowego $c^{(t-1)}$ i zwraca aktualny stan pamięci $c^{(t)}$. Komórka posiada trzy wejścia. Pierwsze z nich to wektor x^t w kroku czasowym t . $a^{(t-1)}$ oznacza wyjście z poprzedniej komórki LSTM. Wcześniej wspomniane $c^{(t-1)}$ to pamięć z poprzedniej jednostki. Symbol \odot zamieszczony na schemacie to iloczyn Hadamarda.

Komórka LSTM posiada trzy różne typy bramek:

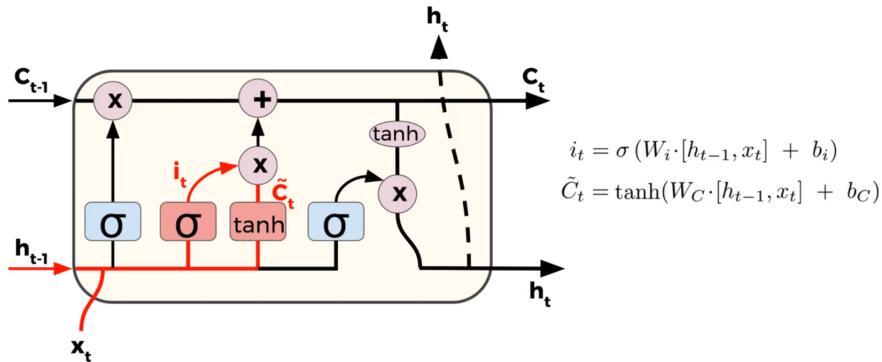
- Bramka zapomnienia (ang. forget gate) pozwala komórce pamięci na wymazanie pamięci. Wektory wejściowe po wymnożeniu przez wagi trafiają do sigmoidalnej funkcji aktywacji. Funkcja aktywacji zwraca wektor, którego elementy są z zakresu $(0, 1)$. Kolejno wektor jest poddany iloczynowi Hadamarda ze wcześniejszym stanem komórki pamięci $c^{(t-1)}$. Kluczową rolę pełni wektor zwrócony przez funkcję aktywacji, ponieważ jeżeli jego elementy są bliskie zera to po zastosowaniu iloczynu Hadamarda z poprzednim stanem pamięci spowoduje wymazanie odpowiednich elementów z wektora $c^{(t-1)}$. W przeciwnym wypadku jeżeli wartości wektora z funkcji aktywacji będą bliskie zera to elementy z wektora $c^{(t-1)}$ zostaną zachowane. Bramka zapomnienia nie była częścią oryginalnej komórki LSTM, została dodana kilka lat później[36].
- Bramka aktualizacji (ang. update gate) jej zadaniem jest aktualizacja stanu komórki. Podobnie jak w przypadku bramki zapomnienia bramka aktualizacji zwraca wektor z wartościami z przedziału $(0, 1)$. Bramka jest powiązana z funkcją aktywacji jaką jest tangens hiperboliczny. Zadaniem funkcji aktywacji jest obliczenie nowych wartości, które mogą zostać wstawione do stanu komórki. Funkcja aktywacji tanh przyjmuje wartości z przedziału $(-1, 1)$ więc otrzymany wektor może zawierać elementy o wartościach ujemnych. Iloczyn Hadmarda dwóch wektorów dodawany jest do stanu pamięci. W wyniku otrzymywany jest wektor c^t - zaktualizowany stan pamięci.
- Bramka wyjścia (ang. output gate) - ostatni element komórki LSTM, który składa się z dwóch komponentów: funkcje aktywacji tanh oraz wyjściową funkcję sigmoidalną.

W pierwszym kroku przepływu danych przez komórkę LSTM dane muszą przejść przez bramkę zapomnienia. W tym kroku bramka decyduje, które informacje mają zostać zapominać, a jakie zachować. Bramka przyjmuje sekwencję danych x^t oraz wektor wyjściowy z poprzedniej komórki h^{t-1} . Schemat przepływu został zilustrowany poniżej. Poniżej ilustracja przedstawiająca schemat budowy komórki LSTM.



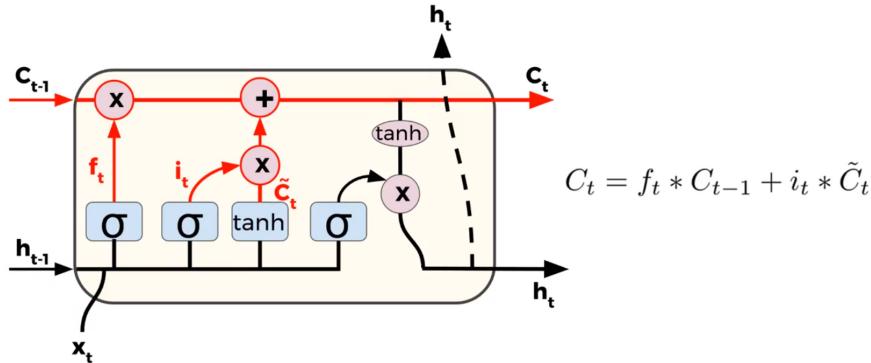
Rysunek 3.21: Pierwszy krok przepływu informacji przez komórkę LSTM [37]

W kolejnym kroku następuła bramka decyduje jaką informację należy zachować. Bramka składa się z dwóch części: funkcji sigmoidalnej oraz tanh. W wyniku otrzymywany jest wektor, który jest nazywany wektorem nowych wartości kandydujących



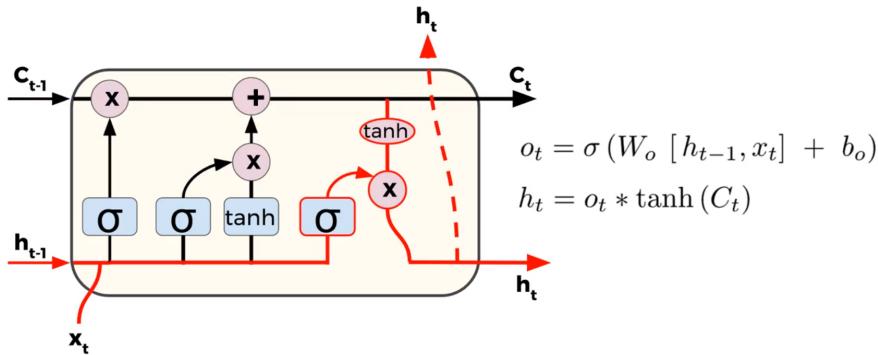
Rysunek 3.22: Drugi krok przepływu informacji przez komórkę LSTM [38]

W kolejnym kroku następuje aktualizacja starego stanu komórki, który jest określony przez c^{t-1} . Zaktualizowany wektor c^t jest odbierany przez kolejną komórkę LSTM w kolejnym kroku czasowym.



Rysunek 3.23: Trzeci krok przepływu informacji przez komórkę LSTM [39]

Ostatni krok do zwrócenie wartości h^t , wartość h^t jest wartością przewidywaną przez komórkę LSTM w czasie t . Połączenie skierowane do góry może posłużyć jako wejście do kolejnej warstwy ukrytej, znajdującej się bezpośrednio nad aktualną. Połączenie skierowane w prawo, stanowi wejście do następnej komórki LSTM.



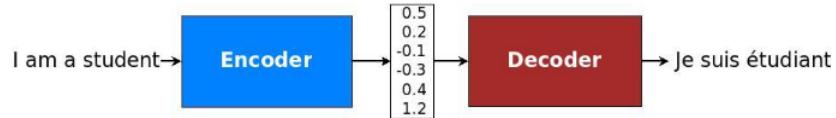
Rysunek 3.24: Czwarty, ostatni krok przepływu przez komórkę LSTM [40]

Badania wykazują, że sieci LSTM znacznie szybciej przychodzą poznawanie długotrwałych zależności, przede wszystkim na sztucznych zbiorach danych, zaprojektowanych do testowania takich możliwości (Bengio et al., 1994; Hochreiter and Schmidhuber, 1997; Hochreiter et al., 2001). Testy odbywają się na specjalnych wymagających zadaniach przetwarzania sekwencji (Graves, 2012 Graves et al., 2013; Sutskever et al., 2014). Nie wszystkie warianty komórek LSTM są takie same jak opisane powyżej. W publikacjach opisujących LSTM można znaleźć wiele innych wariantów, które są lekko odbiegają od oryginalnego modelu. Jednym z popularnych odmian LSTM jest dodatek wprowadzony przez Gersa i Schmidhubera (2000), który wzbogaca standardową komórkę LSTM o tzw. peephole connections. Są to dodatkowe połączenia pomiędzy stanem pamięci, a bramkami. Nieco innym wariantem LSTM jest Gated Recurrent Unit (GRU), koncept ten został wprowadzony przez Cho, et al (2014)[41]. Jednostka GRU łączy bramkę zapomnienia oraz bramkę aktualizacji. Połączony jest także stan komórki ze stanem ukrytym. Uzyskany model jest prostszy niż standardowe modele LSTM i jest coraz bardziej popularny. Inne popularne modele to Depth Gated RNNs autorstwa Yao, et al (2015). Zupełnie inne podejście przedstawił Koutnik w 2014 w pracy Clockwork RNN. Greff, et al. (2015) zrobił porównanie wszystkich popularnych wariantów i stwierdził, że wszystkie działają tak samo. Rafał

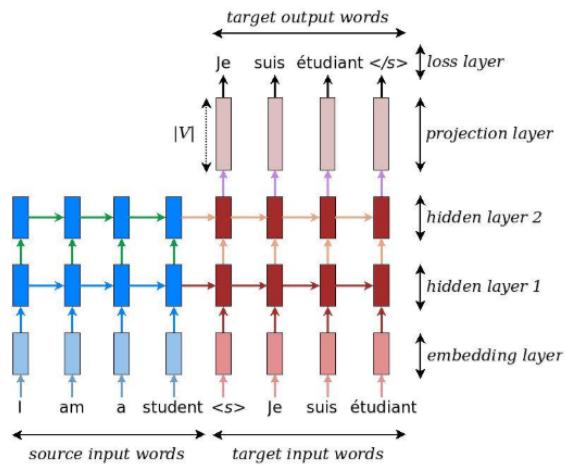
Józefowicz[42] przetestował ponad dziesięć tysięcy różnych architektur i wskazał, że niektóre z nich działają lepiej od standardowej komórki LSTM w zależności od postawionego problemu.

3.7.2 Rodzaj sieci "Sequence to sequence"

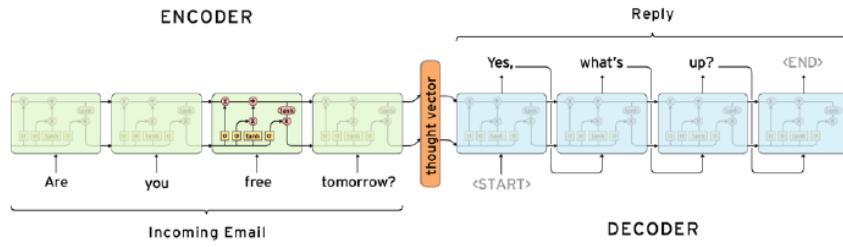
Modele sekwencyjno-sekwencyjne (seq2seq) (Sutskever i in., 2014, Cho i in., 2014)[43] odniosły bardzo duży sukces w modelowaniu zadań dla rekurencyjnych sieci neuronowych takich jak tłumaczenie tekstu, rozpoznanie mowy czy synteza tekstu. Starsze systemy tłumaczenia tekstu rozbijały zdanie na słowa i dokonywały tłumaczenia pojedynczych wyrazów bez zachowania kontekstu zdania. W modelu seq2seq jest przetwarzana cała sekwencja wejściowa i dopiero na podstawie jej kontekstu jest produkowane wyjście. W skład takiego modelu wchodzi mechanizm odpowiedzialny za kodowanie i dekodowanie sekwencji.



Rysunek 3.25: Architektura enkoder - dekoder. Enkoder odpowiedzialny jest za konwersję zdania źródłowego na wektor "znaczeniowy", który jest przekazywany przez dekoder w celu procesu tłumaczenia.[44]



Rysunek 3.26: Przykład architektury enkoder-dekoder, której zadaniem jest przetłumaczenie zdania "I am a student" na język francuski "Je suis étudiant". Specjalny znaczek "< s >" oznacza rozpoczęcie dekodowania, "< /s >" oznacza zatrzymanie dekodowania. [45]

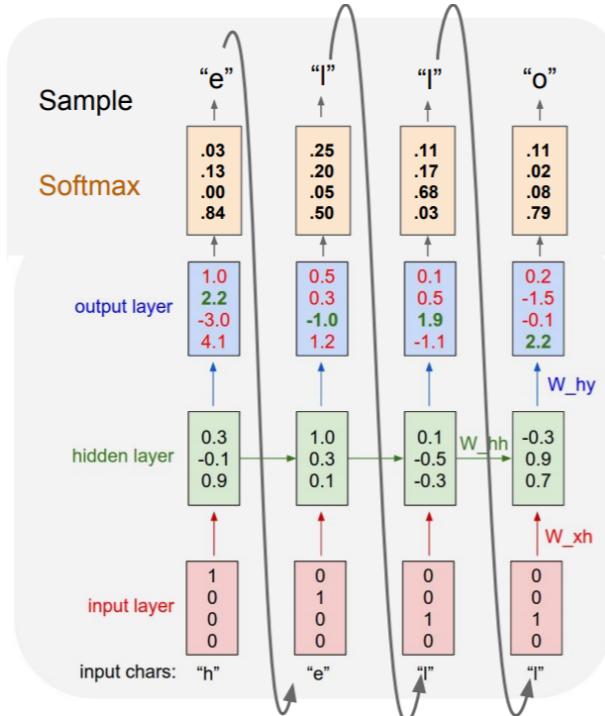


Rysunek 3.27: Inny przykład architektury seq2seq zastosowanej do automatycznej odpowiedzi na wiadomości e-mail

Muzyka jest oparta na sekwencjach, gdzie kolejny zestaw sekwencji do odegrania zależy od poprzedniego zestawu sekwencji. Model seq2seq może zostać wykorzystany z powodzeniem do wygenerowania nut na podstawie nauczonych sekwencji.

3.7.3 Rodzaj sieci ”Character RNN”

Jednym z przykładów sieci operującej bezpośrednio na znakach jest model Character RNN, który został opracowany przez Andreja Karpathego[46]. Zadaniem sieci po wytrenowaniu jest przewidywanie następnego znaku biorąc pod uwagę sekwencję poprzednich znaków. Taki model można zastosować do generowania muzyki pod warunkiem, że dane wejściowe będą zapisane w formacie tekstowym. Jednak w przeciwieństwie do modelu seq2seq character rnn nie jest w stanie zachować dłuższej frazy, ponieważ operuje bezpośrednio na znakach, a nie na sekwencjach. Schemat działania modelu został zilustrowany poniżej.



Rysunek 3.28: Schemat modelu Char RNN [47]

Rysunek powyżej ilustruje przykład, w którym rekurencyjna sieć neuronowa uczy się następstwa znaków w wyrazie *hello*. Każdy ze znaków jest kodowany za pomocą metody one hot encoding[48]. W pierwszym kroku czasowym dla znaku h zostało przypisane zaufanie wynoszące 1.0, dla kolejnej litery h zaufanie wynosi 2.2, dla e -3.0 oraz 1 4.1. W wyrazie hello po literze h występuje litera e, więc sieć powinna zwiększyć zaufanie do tej litery i zmniejszyć do innych. Aby obliczyć w którym kierunku dostosować wagi dla każdej z liter stosuje się algorytm propagacji wstecznej. Cały proces powtarzamy dopóki prognozy sieci będą zgodne z danymi treningowymi. Model ten można przedstawić jako pewne zadanie klasyfikacji, w którym klasy wyników są znakami. Niech y będzie wektorem, który reprezentuje jeden znak zakodowany za pomocą metody one hot encoding. Wektor prawdopodobieństw wystąpienia kolejnego znaku oznaczmy przez \hat{y} to błąd dla pojedynczego znaku można zdefiniować za pomocą $-\sum_{i=0}^n y_i \log(\hat{y}_i)$.

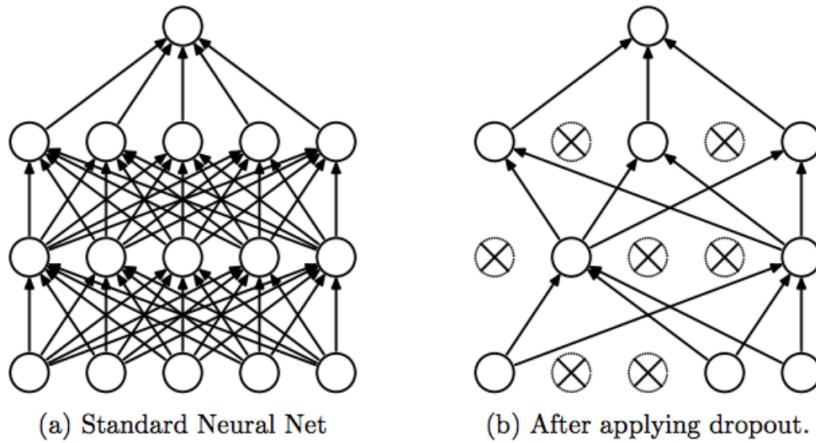
3.7.4 Metody regularyzacji

Jednym z problemów w systemach uczących się jest sposób zaprojektowania algorytmu, który będzie działać poprawnie nie tylko na danych treningowych, ale także na nowych danych wejściowych. Problem związany z uczeniem polega na tym, że w trakcie uczenia nie jest minimalizowany oczekiwany błąd sieci, ale oczekiwany błąd wyznaczony dla zbioru uczącego. Najbardziej pożądaną cechą systemów uczących jest zdolność algorytmu do generalizacji zdobytej wiedzy na nowe przypadki. Dlatego naturalną rzeczą jest dzielenie danych na zbiór testowy i treningowy.

Głównym problemem w systemach uczących się jest sposób zaprojektowania algorytmu, który będzie działać dobrze nie tylko na danych treningowych, ale także na nowych danych wejściowych. Zasadniczym

problemem związanym z podejściem skrótnego przedstawionym powyżej jest to, że w rzeczywistości w trakcie uczenia nie jest minimalizowany oczekiwany błąd sieci, lecz błąd wyznaczony dla zbioru uczącego. Inaczej mówiąc, najbardziej pożądana cecha sieci jest jej zdolność do generalizacji swojej wiedzy na nowe przypadki. Przejawem przedstawionego rozróżnienia między tym, czego chcielibyśmy uczyć sieć, a tym, czego ją naprawdę uczymy jest problem tak zwanego przeuczenia sieci (problem nadmiernego dopasowania).

W przypadku głębokiego uczenia regularyzacja jest sposobem na uniknięcie nadmiernego dopasowania. Zadaniem metod regularizacyjnych jest dodanie tak zwanej kary do funkcji błędu celem zmniejszenia nadmiernego dopasowania. Jedną z metod regularizacji w sieciach neuronowych jest metoda Dropout[49]. Metoda ta polega na "usuwaniu" z pewnym prawdopodobieństwem poszczególnych neuronów. Usuwanie jednostek stosowane jest dla neuronów w ukrytych warstwach sieci w wyższych warstwach sieci. Podczas fazy treningu sieci neuronowej ułamek jednostek ukrytych jest losowo odrzucany przy każdej iteracji z prawdopodobieństwem p_{drop} . Prawdopodobieństwo p_{drop} jest w trakcie konstruowania algorytmu. Powszechnym wyborem jest $p_{drop} = 0.5$. W momencie kiedy część neuronów jest dezaktywowana dokonywane jest przeskalowanie wag aby uwzględnić brakujące neurony. Efekt losowego odrzucenia neuronów zmusza sieć do uczenia się nadmiarowej reprezentacji danych. W takim wypadku sieć nie może polegać na aktywacji jakiegokolwiek zbioru ukrytych neuronów, ponieważ mogą one zostać dezaktywowane w dowolnym momencie podczas treningu.

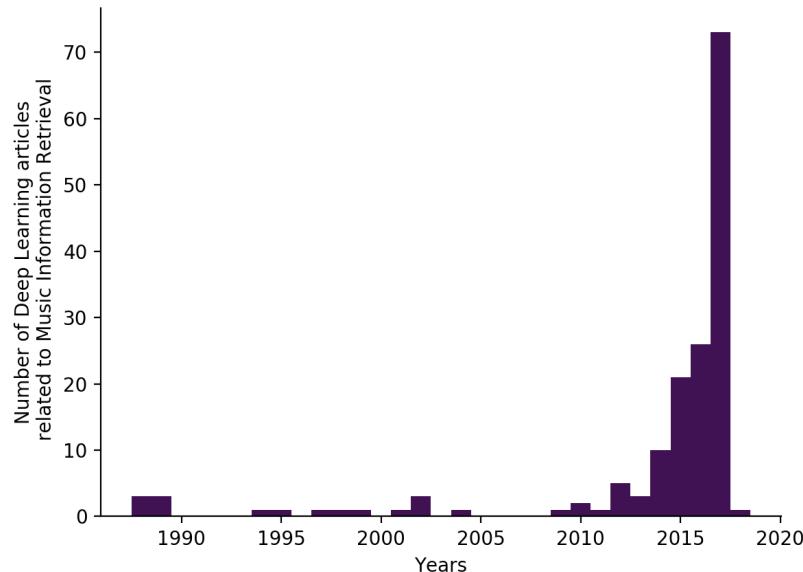


Rysunek 3.29: Standardowa sieć neuronowa na rysunku po lewej i ta sama sieć po zastosowaniu metody dropout po prawej stronie [50]

Rozdział 4

Sieci neuronowe w muzyce

Zastosowanie deep learningu do generowania w ostatnim czasie stało się bardzo popularne. Szybki rozwój frameworków takich jak Tensorflow czy Keras pozwala na sprawne budowanie modeli, a wykorzystanie chmur obliczeniowych zwalnia programistę z używania własnego sprzętu bądź też kupowania drogich procesorów GPU. Poniższa grafika obrazuje liczbę publikacji wydanych na przestrzeni ostatnich trzydziestu lat poświęconych generowaniu muzyki z wykorzystaniem deep learningu.



Rysunek 4.1: Znaczący wzrost publikacji zanotowany w ostatnich latach [51]

Obecnie na rynku można wyróżnić sześć dużych projektów związanych z automatycznym generowaniem muzyki. Poniższe podrozdziały będą miały na celu ich charakteryzację.

4.1 Klasyfikacja narzędzi do tworzenia muzyki

Programy do generowania muzyki z wykorzystaniem metod deep learningu można poddać klasyfikacji ze względu na ich cechy. Poniżej zostanie przeprowadzona klasyfikacja narzędzi:

- Magenta[52] - jest to projekt na licencji open source stworzony przez Google. Jak sami autorzy piszą Magenta jest projektem badawczym, który bada rolę uczenia maszynowego w procesie tworzenia sztuki i muzyki. Wiąże się to z ciągłym poszukiwaniem nowych algorytmów deeplearningu. Projekt stanowi również bogate źródło inspiracji dla muzyków i artystów, ponieważ mogą oni wzbogacić swoje procesy za pomocą modeli deep learningu. Przykładem może być tutaj projekt Duet AI[53]. Magenta powstała na skutek prac inżynierów i badaczy z Google Brain[54]. Niestety projekt nie posiada szczegółowej dokumentacji.



- DeepJazz[55] - jest efektem pracy Ji-Sung Kima podczas 36 godzinnego hackathonu.



- BachBot[56] - projekt badawczy stworzony przez FeynmanLianga na uniwersytecie w Cambridge



- FlowMachines[57] - projekt badawczy, którego celem jest badanie i rozwijanie systemów sztucznej inteligencji zdolnych do generowania muzyki samodzielnie lub we współpracy z ludźmi. W roku 2017 dzięki FlowMachines udało się wygenerować pełny utwór muzyczny gatunku pop



- WaveNet[58] - projekt badawczy naukowców z DeepMind. WaveNet bazuje na Konwolucyjnych Sieciach Neuronowych, ta technika bardzo dobrze sprawdza się w klasyfikacji i generowaniu obrazów. Najbardziej obiecującym celem projektu jest poprawa zastosowań zamiany testu na mowę poprzez wygenerowanie

bardziej naturalnego przepływu wokalu



4.1.1 Klasyfikacja ze względu na dane wejściowe

Klasyfikacja ze względu na dane wejściowe stanowi ważny punkt w momencie planowania architektury sieci neuronowej. Muzyka jest dostępna w różnych cyfrowych formatach. Począwszy od surowego audio (WAV) do formatów, które pozwalają na zapis semantyki utworu muzycznego - są to formaty takie jak MIDI czy notacja ABC. Aby zdecydować, który format danych jest właściwy należy najpierw postawić pytanie o to w jaki sposób sieć neuronowa będzie pracować i jaka będzie jej architektura.

Surowy format audio jest bardzo bogatą reprezentacją muzyki, ponieważ potrafi przechować niemal każdy szczegół utworu muzycznego w zależności od formatu i jakości dźwięku. Sygnały audio zawierają barwę instrumentu na podstawie charakterystyki spektrum. Aby wyodrębnić poszczególne nuty z surowego pliku audio konieczne jest zastosowanie transformaty Fouriera. Ze względu na to zastosowanie plików audio jako wejście do sieci neuronowej jest dość złożonym przedsięwzięciem.

W odniesieniu do powyższych projektów można dokonać podziału na dwie kategorie danych wejściowych:

- Pliki MIDI
- Surowe pliki audio

Projektów, które akceptują pliki MIDI jako dane wejściowe jest znacznie więcej niż tych, które akceptują surowe pliki audio. Do tych pierwszych, nazwijmy to formatem tekstowym zaliczają się:

Magenta - W użyciu biblioteka sprawuje się bardzo dobrze, ponieważ posiada modele wytrenowane na tysiącach plików MIDI. Istnieje możliwość stworzenia własnego modelu na podstawie własnych plików MIDI, które trzeba przekonwertować do protokołu buforowego - NoteSequences stworzonego specjalnie przez Google. Niestety projekt nie posiada szczegółowej dokumentacji. Magenta może generować tylko jeden strumień nut. Nie ma możliwości połączenia kliku modeli np. pianino z perkusją czy gitarą. Trwają prace aby stworzyć model, który będzie w stanie przetwarzać muzykę polifoniczną oraz tworzyć harmonię.

DeepJazz - Projekt na wejściu przyjmuje tylko jeden plik MIDI na podstawie, którego odbywa się trening sieci.

BachBot - Projekt na wejściu przyjmuje pliki MIDI, dokładnie są to chorały JS Bacha.

FlowMachines - System jest w stanie generować nowe partytury nut w oparciu o styl kompozytora na podstawie bazy około 13000 partytur.

Projekty, które operują na surowych plikach audio to:

WaveNet - WaveNet pobiera surowy sygnał audio i syntezuje próbkę wyjściową na podstawie próbki wejściowej. Projekt nie jest oparty o wolny kod źródłowy, ale doczekał się implementacji przez społeczność. Dzięki temu, że sieć używa surowego pliku audio jako wejścia, może generować dowolny instrument. [59] Wykorzystane algorytmy w projekcie są bardzo kosztowne obliczeniowa. Potrzeba kliku minut treningu po to aby wygenerować sekundę dźwięku. Jeden z badaczy pracujący dla Google Sageev Oore z projektu Magenta opisał na swoim blogu jakie wnioski może wyciągnąć kompozytor z wyjścia sieci WaveNet[60]. **GRUV** - GRUV podobnie jak WaveNet próbuje wykorzystać surowy sygnał audio jako dane wejściowe. Naukowcy z Uniwersytetu Stanforda byli jednymi z pierwszych, którzy pokazali w jaki sposób generować nuty za pomocą sieci LSTM wykorzystując surowe pliki audio jako dane wejściowe.

4.1.2 Klasyfikacja ze względu na architekturę

Magenta - Aktualnie Magenta implementuje standardową sieć rekurencyjną oraz dwie sieci LSTM. Projekt posiada trzy typy modeli: BasicRNN, LookbackRNN oraz AttentionRNN.

DeepJazz - Model zawiera dwie warstwy sieci LSTM, które uczą się na podstawie sekwencji nut z plików MIDI.

BachBot - BachBot również wykorzystuje sieci LSTM.

WaveNet - WaveNet bazuje na Konwolucyjnych Sieciach Neuronowych, ta technika bardzo dobrze sprawdza się w klasyfikacji i generowaniu obrazów. Najbardziej obiecującym celem projektu jest poprawa zastosowań zamiany testu na mowę poprzez wygenerowanie bardziej naturalnego przepływu części wokalnej.

GRUV - W przeciwieństwie do WaveNet GRUV wykorzystuje sieci LSTM zamiast CNN. Projekt został opublikowany w 2015 roku[61]. Naukowcy z Uniwersytetu Stanforda byli jednymi z pierwszych, którzy pokazali w jaki sposób generować nuty za pomocą sieci LSTM wykorzystując surowe pliki audio jako dane wejściowe.

4.1.3 Klasyfikacja ze względu na użyte narzędzia

Większość projektów korzysta z wysokopoziomowych frameworków do realizacji zadań. Wyróżnić można tutaj trzy duże frameworki: Tensorflow, Keras, Theano.

Tensorflow - jest wykorzystywany przez projekt Magenta. Tak samo jak projekt Magenta framework Tensorflow powstał w Google. Tensorflow jest wykorzystywany przez frameworki Keras i Theano - takiego zestawu narzędzi użył Ji-Sung Kim konstruując swój projekt DeepJazz.

4.1.4 Klasyfikacja ze względu na źródła finansowania

Wielkość projektu często zależy od źródła jego finansowania na wyżej wymienione projekty można podzielić na trzy grupy ze względu na finansowanie:

- Unia Europejska - projekt badawczy Flow Machines, został sfinansowany przez Europejską radę ds. Badań Naukowych w ramach 70 programu ramowego UE. Badania zostały zapoczątkowane przez francuskiego naukowca Françoisa Pachet w Sony Computer Science Laboratories (Sony SCL Paris) oraz Uniwersytet Piotra i Marii Curie (UPMC)
- Firmy - Google Magenta powstało z wykorzystaniem środków finansowych firmy Google podobnie jak projekt WaveNet
- Non profit - projekt DeepJazz został stworzony podczas 36 godzinnego hackathonu.
- Granty uniwersyteckie - projekt BachBot został stworzony w murach Uniwersytetu Cambridge, podobnie jak projekt GRUV, który powstał w murach uczelni Stanforda

4.2 Własny model sieci

Celem praktycznym pracy jest przygotowanie własnego modelu sieci neuronowej, który po odpowiednim treningu będzie w stanie wygenerować nuty na podstawie danych treningowych. Model sieci będzie wykorzystał komórki LSTM. Na dane treningowe będą składać się pliki MIDI, w pracy wykorzystano jako jeden z

przykładów preludia Fryderyka Chopina. Model został stworzony z wykorzystaniem wysokopoziomowego framework'u Kers, trening odbywał się na maszynie wirtualnej skonfigurowanej w Google Cloud. Projekt składa się z trzech plików:

- prepare.py - plik odpowiedzialny za konwersje plików MIDI do formy znakowej
- model.py - plik odpowiedzialny za wytrenowanie modelu
- generate.py - plik odpowiedzialny za wykorzystanie wytrenowanego modelu
- utils.py - plik zawierający pomocnicze funkcje

Specyfikacja maszyny na której odbywał się trening modelu:

- Procesor: Intel(R) Xeon(R) CPU @ 2.20GHz
- Karta graficzna: NVIDIA Tesla K80
- HDD: 30GB
- System operacyjny: Ubuntu 18.04.1 LTS
- Wersja jądra: 4.15.0-1017-gcp

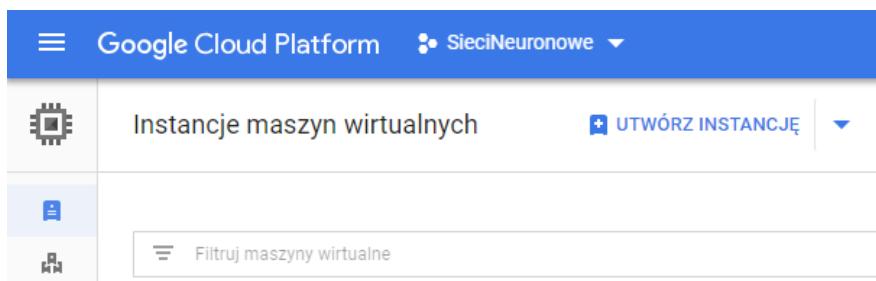
4.2.1 Konfiguracja maszyny w chmurze

Biorąc pod uwagę potęgę ówczesnych procesorów graficznych bardzo często wykorzystuje się do przetwarzania długich obliczeń. W sieci istnieje szereg usług opartych na chmurze, na których to możemy wypożyczyć sobie konkretną maszynę. Jedyni z popularniejszych usługodawców to:

- Google Cloud [62]
- Amazon AWS [63]

Na przykładzie użycia Google Cloud opisany zostanie proces tworzenia wirtualnej maszyny oraz jej konfiguracja w celu uruchamiania programów, które mają wykorzystywać procesor GPU.

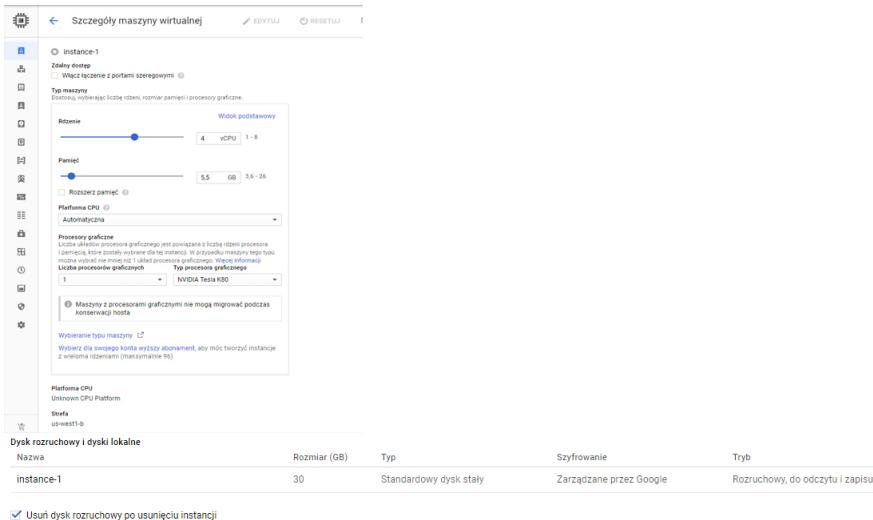
Stworzenie nowej instancji wirtualnej maszyny odbywa się z menu *Compute Engine → Instancje maszyn wirtualnych*. Następnie należy wybrać przycisk *Utwórz Instancję*



Rysunek 4.2: Tworzenie maszyny wirtualnej

Kolejny krok do konfiguracji w zależności od potrzeb. Należy pamiętać o tym, że cena maszyny zależy od jej lokalizacji. Maszyna została skonfigurowana w następujący sposób:

- Region: us-east1
- Liczba procesorów: 4
- RAM: 6GB
- System operacyjny: Ubuntu 18.04
- GPU: Tesla K80
- 30GB SSD
- Stały adres IP



Rysunek 4.3: Konfiguracja maszyny

Po utworzeniu instancji wirtualnej maszyny komunikacja z nią odbywa się za pomocą protokołu SSH z wykorzystaniem kryptografii klucza publicznego i prywatnego. Aby zainstalować biblioteki CUDA należy uruchomić poniższy skrypt, którego zadaniem jest pobranie i instalacja CUDA w wersji 9.2

```

1 sudo su
2 #!/bin/bash
3
4 if ! dpkg--query -W cuda; then
5 sudo apt install gcc-6 g++-6
6 sudo apt-get install linux-headers-$(uname -r)
7 sudo apt install nvidia-384
8 wget -c https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda_9
      .2.88_396.26_linux
9 chmod +x cuda_9.2.88_396.26_linux.run
10 ./cuda_9.2.88_396.26_linux.run --verbose --silent --toolkit --override
11 fi
12 export PATH="$PATH:/usr/local/cuda-9.2/bin"
```

```

13 echo "/usr/local/cuda-9.2/lib64" >> /etc/ld.so.conf
14 ln -s /usr/bin/gcc-6 /usr/local/cuda-9.2/bin/gcc
15 ln -s /usr/bin/g++-6 /usr/local/cuda-9.2/bin/g++

```

Listing 4.1: Skrypt instalacyjny sterowników CUDA

Weryfikacje instalacji można przeprowadzić za pomocą polecenia *nvidia-smi*:

```

1 +
2 | NVIDIA-SMI 396.44                               Driver Version: 396.44 |
3 +-
4 | GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC |
5 | Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
6 +-
7 | 0   Tesla K80      Off  00000000:00:04.0 Off    0 |
8 | N/A  36C     P0    75W / 149W  0MiB / 11441MiB | 100%      Default |
9 +-
10
11 +
12 | Processes:                               GPU Memory |
13 |  GPU        PID  Type      Process name        Usage |
14 +-
15 |  No running processes found               |
16 +

```

Listing 4.2: Wynik polecenia nvidia-smi

Dodatkowo firma NVIDIA wprowadziła bibliotekę o nazwie cuDNN, której zadaniem jest optymalizacja dla głębokich sieci neuronowych. Wersję cuDNN 8 można pobrać ze strony producenta <https://developer.nvidia.com/cudnn>. Poniższy skrypt przeprowadzi instalację pobranego pakietu.

```

1 tar xzvf cudnn-8.0-linux-x64-v5.1.tgz
2 sudo cp cuda/lib64/* /usr/local/cuda/lib64/
3 sudo cp cuda/include/cudnn.h /usr/local/cuda/include/
4 rm -rf ~/cuda
5 rm cudnn-8.0-linux-x64-v5.1.tgz

```

Środowisko jest już gotowe do uruchamiania programów napisanych z użyciem biblioteki CUDA. Do monitorowania pracy karty graficznej może posłużyć narzędzie *nvtop*[64].

4.3 Schemat działania modelu

Inspiracją do stworzenia modelu były rozwiązania zastosowane w modelach przeznaczonych do generowania tekstu. W modelowaniu językowym danymi wejściowymi zazwyczaj jest sekwencja słów, a wynik to sekwencja przewidywanych słów. Dana jest długość sekwencji n . Sieć ma za zadanie nauczyć się liter, które występuje bezpośrednio po sekwencji długości n . Poniżej przykład na podstawie fragmentu Pana Tadeusza.

Listing 4.3: Pan Tadeusz

```

1 Litwo! Ojczyzno moja! ty jesteś jak zdrowie:
2 Ile cię trzeba cenić, ten tylko się dowie,
3 Kto cię stracił. Dziś piękność twą w całej ozdobie
4 Widzę i opisuję, bo tesknię po tobie.

```

Ustalmy długość sekwencji $n = 10$, wówczas ciągi odpowiadające danej literze będą wyglądały następująco:

Listing 4.4: Sekwencje wszystkich możliwych ciągów

```

1 litwo! ojc => z
2 itwo! ojcz => y
3 two! ojczy => z
4 wo! ojczyz => n
5 o! ojczyzn => o
6 ! ojczyzno =>
7 ojczyzno => m
8 ojczyzno m => o
9 jczyzno mo => j
10 czyzno moj => a
11 zyzno moja => !
12 yzno moja! =>
13 zno moja! => t
14 no moja! t => y
15 o moja! ty =>
16 moja! ty => j
17 moja! ty j => e
18 oja! ty je => s
19 ja! ty jes => t
20 a! ty jest => e
21 ! ty jeste =>
22 ty jeste => j
23 ty jeste j => a
24 y jeste ja => k
25 jeste jak =>
26 jeste jak => z
27 este jak z => d
28 ste jak zd => r
29 te jak zdr => o
30 e jak zdro => w
31 jak zdrow => i
32 jak zdrowi => e
33 ak zdrowie => :
34 k zdrowie: =>
35 zdrowie: => i
36 zdrowie: i => l
37 drowie: il => e
38 rowie: ile =>
39 owie: ile => c
40 wie: ile c => i
41 ie: ile ci =>
42 e: ile ci => t
43 : ile ci t => r
44 ile ci tr => z
45 ile ci trz => e
46 le ci trze => b
47 e ci trzeb => a
48 ci trzeba =>
49 ci trzeba => c
50 i trzeba c => e
51 trzeba ce => n
52 trzeba cen => i
53 rzeba ceni => ,
54 zeba ceni , =>
55 eba ceni , => t
56 ba ceni , t => e
57 a ceni , te => n

```

```

58 ceni, ten =>
59 ceni, ten => t
60 eni, ten t => y
61 ni, ten ty => l
62 i, ten tyl => k
63 , ten tylk => o
64 ten tylko =>
65 ten tylko => s
66 en tylko s => i
67 n tylko si =>
68 tylko si => d
69 tylko si d => o
70 ylko si do => w
71 lko si dow => i
72 ko si dowi => e
73 o si dowie => ,
74 si dowie, =>
75 si dowie, => k
76 i dowie, k => t
77 dowie, kt => o
78 dowie, kto =>
79 owie, kto => c
80 wie, kto c => i
81 ie, kto ci =>
82 e, kto ci => s
83 , kto ci s => t
84 kto ci st => r
85 kto ci str => a
86 to ci stra => c
87 o ci strac => i
88 ci straci => .
89 ci straci. =>
90 i straci. => d
91 straci. d => z
92 straci. dz => i
93 traci. dzi =>
94 raci. dzi => p
95 aci. dzi p => i
96 ci. dzi pi => k
97 i. dzi pik => n
98 . dzi pikn => o
99 dzi pikno =>
100 dzi pikno => t
101 zi pikno t => w
102 i pikno tw =>
103 pikno tw => w
104 pikno tw w =>
105 ikno tw w => c
106 kno tw w c => a
107 no tw w ca => e
108 o tw w cae => j
109 tw w caej =>
110 tw w caej => o
111 w w caej o => z
112 w caej oz => d
113 w caej ozd => o
114 caej ozdo => b
115 caej ozdob => i

```

```

116 aej ozdobi => e
117 ej ozdobie =>
118 j ozdobie => w
119 ozdobie w => i
120 ozdobie wi => d
121 zdobie wid => z
122 dobie widz =>
123 obie widz => i
124 bie widz i =>
125 ie widz i => o
126 e widz i o => p
127 widz i op => i
128 widz i opi => s
129 idz i opis => u
130 dz i opisu => j
131 z i opisuj => ,
132 i opisuj , =>
133 i opisuj , => b
134 opisuj , b => o
135 opisuj , bo =>
136 pisuj , bo => t
137 isuj , bo t => s
138 suj , bo ts => k
139 uj , bo tsk => n
140 j , bo tskn => i
141 , bo tskni =>
142 bo tskni => p
143 bo tskni p => o
144 o tskni po =>
145 tskni po => t
146 tskni po t => o
147 skni po to => b
148 kni po tob => i
149 ni po tobi => e
150 i po tobie => .

```

W podobny sposób jak język pisany, muzyka działa jako forma ekspresji, w której kombinacje dźwięków mogą wyrażać emocje. Zatem metody, które są wykorzystywane do generowania tekstu z wykorzystaniem deep learningu mogą zostać przełożone na generowanie muzyki.

Jako dane wejściowe do sieci zostały przygotowane dwa zbiory danych w formacie MIDI:

- 24 Preludia Fryderyka Chopina - 215KB
- 21 Sonatin W. A. Mozarta - 668KB

Pierwszy etap polega na przygotowaniu danych. Polega to na stworzeniu tablicy, która będzie zawierała wszystkie nuty z wybranego zbioru plików. Początkowy wycinek z tablicy, który powstał przy przetwarzaniu dzieł Fryderyka Chopina: [*'C2'*, *'G3'*, *'G2'*, *'C4'*, *'E3'*, *'G4'*, *'E4'*, *'C4'*, *'A4'*, *'A3'*, *'B1'*, *'G3'*, *'G2'*, *'D4'*, *'F3'*, *'G4'*, *'F4'*, *'D4'*, *'A4'*, *'A3'*] odpowiada pierwszemu preludium z Opusu 28. Tablica przechowuje tylko wysokość danej nuty, inne cechy utworu muzycznego nie są wykorzystywane.

Prelude No. 1



Rysunek 4.4: Początkowy fragment preludium

Liczba wszystkich unikalnych nut zależy od wybranego zbioru danych. Kolejnym krokiem jest stworzenie słownika, który przyporządkuje daną nutę lub akord kolejnej liczbie naturalnej. Rozmiar słownika powinien wynosić tyle ile jest unikalnych nut i akordów.

Następnie w trzecim kroku należy stworzyć sekwencje wejściowe dla sieci i ich odpowiednich wyjść. Wyjście dla każdej sekwencji wejściowej będzie pierwszą nutą lub akordem, która pojawia się po sekwencji nut w sekwencji wejściowej na naszej liście nut.

Listing 4.5: Sekwencje wejściowe i odpowiadające im nuty

1	['C2' , 'G3' , 'G2' , 'C4' , 'E3']	=> G4
2	['G3' , 'G2' , 'C4' , 'E3' , 'G4']	=> E4
3	['G2' , 'C4' , 'E3' , 'G4' , 'E4']	=> C4
4	['C4' , 'E3' , 'G4' , 'E4' , 'C4']	=> A4
5	['E3' , 'G4' , 'E4' , 'C4' , 'A4']	=> A3
6	['G4' , 'E4' , 'C4' , 'A4' , 'A3']	=> B1
7	['E4' , 'C4' , 'A4' , 'A3' , 'B1']	=> G3
8	['C4' , 'A4' , 'A3' , 'B1' , 'G3']	=> G2
9	['A4' , 'A3' , 'B1' , 'G3' , 'G2']	=> D4
10	['A3' , 'B1' , 'G3' , 'G2' , 'D4']	=> F3

Do przewidywania prawdopodobieństwa użycie średniego błędu kwadratowego (RMSE) jako funkcji kosztu nie jest właściwe. Jako funkcja kosztu zostanie wykorzystana funkcja entropii krzyżowej kategorycznej.

$$L_i = - \sum_j t_{i,j} \log(p_{i,j}) \quad (4.1)$$

Funkcja ta jest odpowiednia do przewidywania wartości ze zbioru wieloklasowego. Jest również domyślnym wyborem przy połączeniu z funkcją aktywacji softmax. Ostatnim krokiem w przygotowaniu modelu jest normalizacja danych wejściowych oraz zapewnienie one-hot encoding dla danych wyjściowych. Kodowanie wyjścia polega na tym, że na pozycji określającej daną klasę stoi jedynka, a na pozostałych za零. Poniższy przykład ilustruje kodowanie cyfr.

- cyfra '0' zostanie zakodowana jako [1,0,0,0,0,0,0]
- cyfra '1' zostanie zakodowana jako [0,1,0,0,0,0,0]

- cyfra '9' zostanie zakodowana jako [0,0,0,0,0,0,1]

Kodowanie zostanie wykorzystane do zakodowania etykiet poszczególnych nut, które znajdują się we wcześniej stworzonym słowniku.

4.3.1 Architektura modelu

Model sieci neuronowej został przygotowany w oparciu o wysokopoziomowy framework Keras. Domyślna architektura sieci składa się z pięciu warstw:

- LSTM z 128 wejściami
- Dropout
- LSTM z 128 wejściami
- Dropout
- Warstwa wyjścia składająca się z N neuronów, gdzie N odpowiada liczbie unikalnych nut w wybranym zbiorze danych

Do modelu zostały dobrane następujące domyślne parametry:

- Funkcja straty: entropia krzyżowa
- Optymalizator: Adam
- Stała uczenia: 0.001
- Podział na zbiór testowy i walidacyjny: 0.2

Wykorzystując uczenie nadzorowane, model sieci neuronowej ma za zadanie wytrenować model. W modelu używane są dwa zestawy danych jako dane wejściowe. Można je zdefiniować jako: sekwencje nut długości $n - X$ oraz klasa predyktorów czyli pojedyncze nuty, które występują bezpośrednio po danej sekwencji - y . Dane wejściowe zostały podzielone również na zestawy szkoleniowe i walidacyjne aby móc sprawdzić poprawność działania modelu.

4.4 Szczegółowy opis programu

Pierwszym etapem przy pracy z programem jest przygotowanie danych wejściowych do sieci. Ze względu na to, że format MIDI nie zawiera bezpośrednio danych tekstowych trzeba je wydobyć. Do konwersji plików MIDI na dane tekstowe wykorzystana została biblioteka music21. Jeżeli w utworze muzycznym występują akordy to są one rozbijane na poszczególne nuty składowe. Po konwersji wszystkich plików dane w postaci binarnej są zapisywane do pliku aby przy ponownym uruchomieniu programu uniknąć ponownej konwersji plików. Domyślna nazwa katalogu, w którym są przechowywane pliki MIDI to *midi*. Dodatkowo za pomocą programu można dokonać transpozycji wszystkich plików midi do wybranej tonacji. Pomoc do pliku można wyświetlić za pomocą przełącznika *-h*.

```

1 usage: prepare.py [-h] [--midi_dir MIDI_DIR] [--out_file OUT_FILE]
2 [--transpose TRANSPOSE]
3
4 optional arguments:
5 -h, --help            show this help message and exit
6 --midi_dir MIDI_DIR    MIDI files directory containing .mid files (default:
7 midi/)
8 --out_file OUT_FILE    Path to file containing done parse MIDI files
9 (default: data/notes)
10 --transpose TRANSPOSE
11 Key to transpose all MIDI files (default: None)

```

Listing 4.6: Pomoc do programu *prepare.py*

Dane w postaci binarnej są zapisywane do pliku z wykorzystaniem biblioteki *pickle*. Dane domyślnie zapisywane są w katalogu *data*.

Po wstępny przygotowaniu danych należy użyć pliku *train.py*. Plik ten, zawiera domyślny model sieci neuronowej oraz szereg parametrów, które można zmieniać za pomocą odpowiednich przełączników. Program jest odpowiedzialny za przygotowanie danych do sieci oraz za trening modelu. Aby użyć modelu, plik z danymi może być wcześniej przygotowany przez program *prepare.py*. Jeżeli wcześniej przygotowany plik z danymi nie istnieje, można użyć domyślnego lub za pomocą opcji *--midi_dir* przygotować własny. Pomoc do pliku można wyświetlić za pomocą przełącznika *-h*.

```

1 $ python3 train.py -
2 usage: train.py [-h] [--midi_dir MIDI_DIR] [--data_file DATA_FILE]
3 [--results_dir RESULTS_DIR] [--lstm_size LSTM_SIZE]
4 [--layers LAYERS] [--learning_rate LEARNING_RATE]
5 [--sequence_size SEQUENCE_SIZE] [--batch_size BATCH_SIZE]
6 [--dropout DROPOUT]
7 [--optimizer {sgd,rmsprop,adagrad,adadelta,adam,adamax,nadam}]
8 [--activation {softmax,sigmoid,linear,tanh,elu,selu,softplus,softsign,relu}]
9
10 optional arguments:
11 -h, --help            show this help message and exit
12 --midi_dir MIDI_DIR    MIDI files direcotry containing .mid files to use for
13 training neural network (default: midi)
14 --data_file DATA_FILE
15 Path to file containing done parse MIDI files
16 (default: data/notes-preludia)
17 --results_dir RESULTS_DIR
18 Directory to store model in JSON format, logs for
19 tensorboard and weights (default: results)
20 --lstm_size LSTM_SIZE
21 Size of LSTM layer (default: 128)
22 --layers LAYERS        Number of layers in the neural network (default: 2)
23 --learning_rate LEARNING_RATE
24 Learning rate (default: 0.0001)
25 --sequence_size SEQUENCE_SIZE
26 Sequence size for notes (default: 100)
27 --batch_size BATCH_SIZE
28 Batch size (default: 128)
29 --dropout DROPOUT      Dropout percentage value. One of regularization method
30 (default: 0.3)
31 --optimizer {sgd,rmsprop,adagrad,adadelta,adam,adamax,nadam}
32 Optimization algorithm to use (default: rmsprop)
33 --activation {softmax,sigmoid,linear,tanh,elu,selu,softplus,softsign,relu}

```

```
34 Activation function to use (default: softmax)
```

Listing 4.7: Pomoc do programu *train.py*

Każdorazowe uruchomienie programu *train.py* powoduje utworzenie na katalogu *results* podkatalogu, który zawiera:

- bieżący model sieci neuronowej w formacie JSON
- wagi
- logi przeznaczone do wyświetlenia w *tensorboard*[65]

Po wczytaniu pliku z danymi program musi odpowiednio zmapować dane, ponieważ sieć neuronowa potrzebuje danych numerycznych, a w tym momencie są dostępne dane w formie łańcuchów znaków. Po konwersji danych na dane liczbowe następuje przygotowanie sekwencji wejściowych i odpowiednich wyjść dla danej sekwencji. Jest tutaj wykorzystywany parametr o nazwie *-sequence_size*, który informuje o długości sekwencji, domyślnie ma wartość 100, którą można zmodyfikować.

Następnie dane są normalizowane oraz przygotowywane jako wejście do sieci z wykorzystaniem techniki one-hot-encoding. Podczas treningu modelu wagi są co jakiś czas zapisywane do plików .hdf5 po to aby po zakończeniu treningu można było wybrać odpowiedni zestaw wag biorąc pod uwagę błąd oraz dopasowanie modelu. Jest to też pewnego rodzaju zabezpieczenie przed nieoczekiwany zatrzymaniu modelu.

Proces uczenia sieci można obserwować w czasie rzeczywistym wydając polecenie:

```
1 tensorboard --logdir resuts/
```

Listing 4.8: Wykorzystanie *tensorboard* do wizualizacji wyników

Po wytrenowaniu modelu można podjąć próbę wygenerowania nut. Do tego celu jest potrzebny ostatni plik *generate.py*. Uruchomienie pliku wymaga podania dwóch argumentów. Pierwszy z argumentów wskazuje na wybrany plik z wagami. Drugi musi wskazywać na wcześniej stworzony plik z nutami, który był wykorzystywany do trenowania sieci.

```
1 python3 generate.py --weights weights.hdf5 --notesfile filepath
```

Program domyślnie podejmie próbę wygenerowania 1000 nut, można tę długość zmienić za pomocą opcjonalnego parametru *-len*:

```
1 python3 generate.py --weights weights.hdf5 --notesfile filepath --len 500
```

Aby wykorzystać sieć neuronową do generowania muzyki należy wykorzystać dokładnie tą samą sieć, która była użyta do treningu. Jednak sieć nie będzie ponownie szkolona. Zostanie wypełniona tymi samymi danymi oraz jako wagi zostaną wykorzystane wagi, które powstały na skutek treningu sieci w poprzednim kroku. Po załadowaniu wag do modelu można go użyć do generowania nut. Jako punkt startowy wybierany jest losowy indeks z danych wejściowych. Pozwala to na kliku krotne generowanie nut na tych samych danych wagowych z różnymi wynikami przy każdej próbie. Ze względu na to, że sieć posiada tylko dane numeryczne konieczne jest zdekodowanie każdej wartości liczbowej na odpowiednią nutę. Do każdej wygenerowanej nuty, która ma być wygenerowana musi zostać przypisana odpowiednia sekwencja. Pierwsza sekwencja jest wybierana losowo. Z każdej kolejnej sekwencji, która jest używana jako wejścia usuwana jest pierwsza nuta i wstawiany jest wynik z poprzedniej iteracji. Do określenia najbardziej prawdopodobnej nuty na podstawie danych wejściowych z sieci wybierany jest indeks, który posiada największe prawdopodobieństwo wystąpienia po danej sekwencji. Cały proces jest powtarzany, a dane są gromadzone w tablicy.

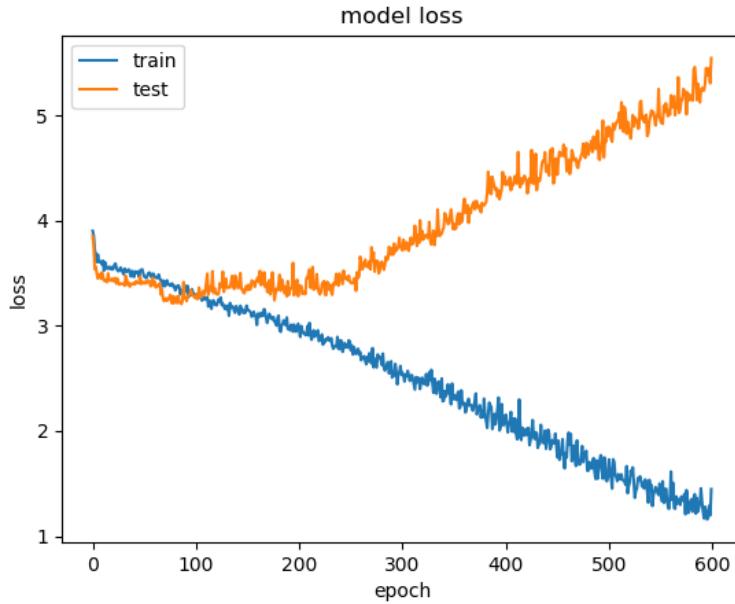
4.4.1 Użyte narzędzia

W projekcie użyto następujące wersje narzędzi:

- **Python 3.6** - język programowania powszechnie używany w uczeniu maszynowym
- **Keras 2.2.2** - wysokopoziomowa biblioteka, która umożliwia tworzenie sztucznych sieci neuronowych. Biblioteka używa TensorFlow lub Theano.
- **Tensorflow GPU 1.10.0** - biblioteka stworzona przez zespół Google Brain, która ma zastosowanie w uczeniu maszynowym oraz głębokich sieciach neuronowych. Obliczenia mogą być wykonywane na procesorze CPU lub GPU.
- **Tensorboard** - narzędzie, które umożliwia wizualizowanie sieci neuronowej podczas nauki. Pozwala to na wcześniejsze ocenienie czy proces uczenia będzie pomyślny.
- **music21** - rozbudowana biblioteka służąca do pracy z muzyką. Za jej pomocą można analizować pliki MIDI i je tworzyć.
- **sklearn** - biblioteka, która ma zastosowanie w uczeniu maszynowym.
- **numpy** - biblioteka służąca do wykonywania obliczeń matematycznych
- **matplotlib** - biblioteka służąca do rysowania wykresów
- **CUDA 9.0.176** - *Compute Unified Device Architecture* jest to opracowana przez firmę NVIDIA architektura procesorów GPU, która umożliwia wykorzystanie procesorów GPU do rozwiązywania problemów numerycznych

4.4.2 Wyniki

W zależności od układu i liczby warstw wyniki działania sieci neuronowej są różne. Częstym problemem w otrzymanych wynikach było to, że sieć została przeuczona co wynikało z tego, że krzywa zbioru walidacyjnego znacznie odbiegała od krzywiej zbioru testowego. Nie przeszkodziło to jednak w generowaniu utworu. Chociaż można usłyszeć, że wygenerowany utwór zawiera frazy, które zostały wyciągnięte bezpośrednio ze danych uczących się. Poniżej przedstawiono kilka lepszych rezultatów z wielu podejmowanych prób w różnych konfiguracjach sieci.



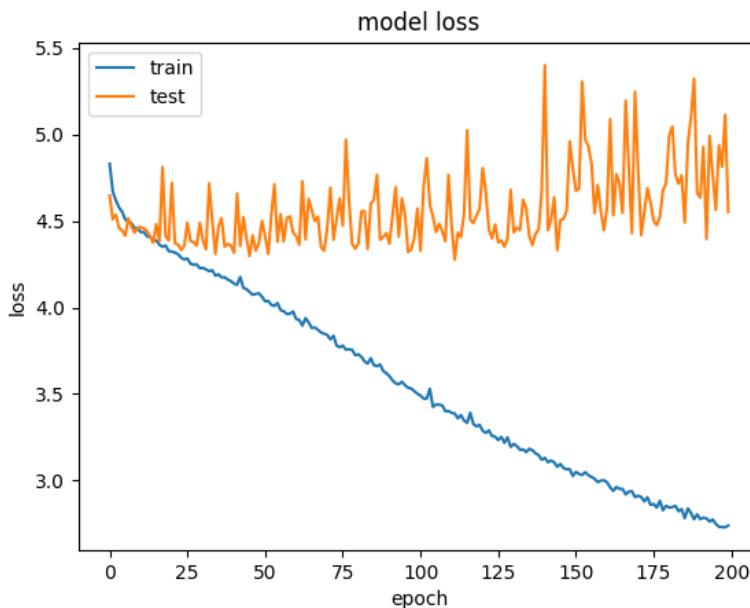
Rysunek 4.5: Przeuczenie sieci

Dzięki temu, że podczas treningu model zapisuje wagi co jakiś czas można było wykorzystać wagi z epoki 200 do wygenerowania nowych nut. W wyniku powstał plik MIDI, który muzycznie może przypominać styl preludiów. Co ważne, wygenerowany utwór nie jest chaosem losowo wybranych nut co może świadczyć o potencjale sztucznych sieci neuronowych na polu generowania muzyki. Poniżej fragment wygenerowanego utworu.



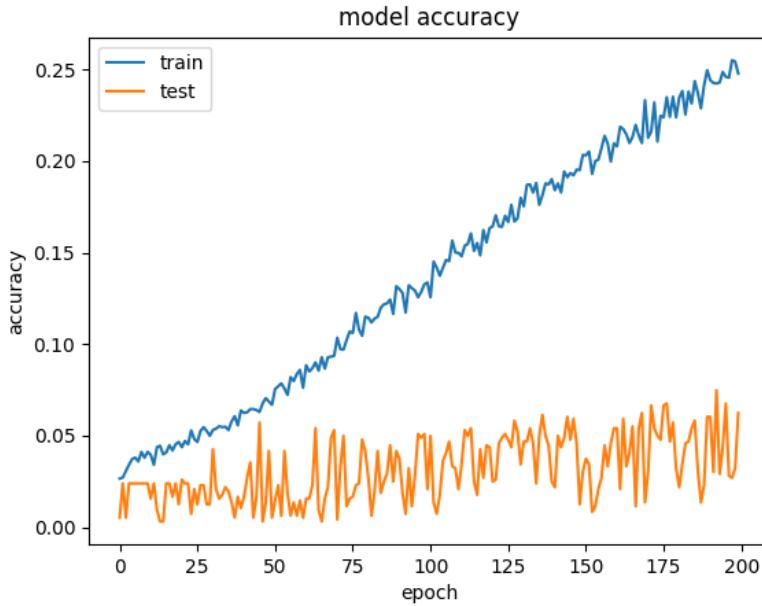
Rysunek 4.6: Początkowy fragment wygenerowanego utworu

Inny wynik został uzyskany wykorzystując model, który składał się z dwóch warstw LSTM.



Rysunek 4.7: Wykres błędu

Wykres niestety też nie jest zadowalający, ale nie przeszkadza to przy wygenerowaniu utworu. Poniżej wykres przedstawiający dopasowanie modelu.



Rysunek 4.8: Dopasowanie modelu



Rysunek 4.9: Fragment utworu, który został wygenerowany z użyciem dwóch warstw LSTM

4.4.3 Wnioski

Wykorzystanie sieci neuronowych sprawdza się w przypadku generowania muzyki. Sieci są w stanie przetworzyć duże zbiory danych. Problem pojawia się w przypadku uczenia modelu, ponieważ ten przeucza się. Rozwiążanie tego problemu może leżeć w nieprawidłowym przygotowaniu danych wejściowych. Należaałoby oddziennie zająć się harmonią (lewą ręką) i melodią (prawą ręką)[66] dodatkowo takie elementy jak tempo czy dynamika mogłyby być rozpoznawane przez sieć. Wymaga to dodatkowych eksperymentów na polu rekurencyjnych sieci neuronowych. Niestety nie ma jednoznacznego przepisu co do ilości warstw, algorytmu optymalizacji czy funkcji aktywacji należy na zasadzie prób i błędów wybrać odpowiednią konfigurację. Możliwe, że zastosowanie komórki podobnej do LSTM - GRU[67] (*Gated recurrent unit*), które są lżejsze

obliczeniowo wpłynęły na otrzymane wyniki. W ostatnim czasie dużą popularność zyskały sieci typu *GAN* (Generative Adversarial Nets) [68]. *Generative Adverial Nets* są również wykorzystywane na polu generowania muzyki[69].

Zakończenie

Współczesne techniki informatyczne w połączeniu z matematyką dają szerokie pole manewru w dziedzinie generowania muzyki bądź w dziedzinie komponowania muzyki. Niewątpliwie ważną cechą jest prawidłowe przygotowanie danych. Sieci neuronowe w kontekście muzyki mają duże znaczenie, a łańuchy Markowa i gramatyki mogą pośredniczyć w przygotowaniu końcowego modelu. W ostatnim czasie powstały sieci służące typowo do zadań generatywnych *Generative adversarial networks*, które zostały przedstawione przez Iana Goodfellow w 2014. Rozległość i ciągły rozwój metod sztucznej inteligencji pozwala na szeroką gamę możliwości empirycznego eksperymentowania w dziedzinie generowania muzyki czy też wspomagania kompozytorów w tworzeniu muzyki.

Appendices

Dodatek A

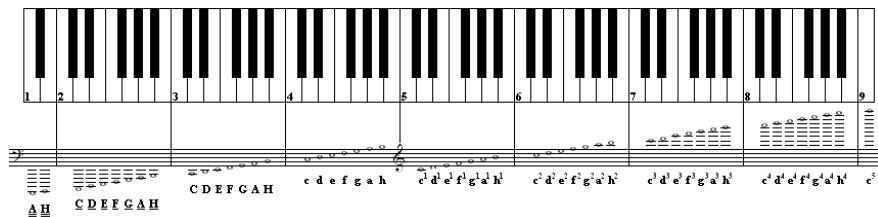
Elementy muzyki

Dodatek opisuje podstawowe elementy teorii muzyki zaczynając od notacji muzycznej. Pojedyncza nuta jest opisywana za pomocą czterech właściwości.

- Wysokość dźwięku
- Czas trwania dźwięku
- Głośność dźwięku
- Barwa dźwięku, która zależy od instrumentu

A.1 Wysokość dźwięku

Klawiatura fortepianu składa się z pewnych stale powtarzających się ugrupowań klawiszy. Można wyróżnić tutaj siedem białych i pięć czarnych klawiszy, które znajdują się pomiędzy białymi takie ugrupowanie nazywane jest oktawą. Jest to podstawowa jednostka materiału dźwiękowego. Dźwięki odległe od siebie o oktawę są bardzo do siebie podobne. Wszystkie dźwięki odległe od siebie o oktawę noszą te same nazwy.



Rysunek A.1: Układ dźwięków na klawiaturze fortepianu [70]

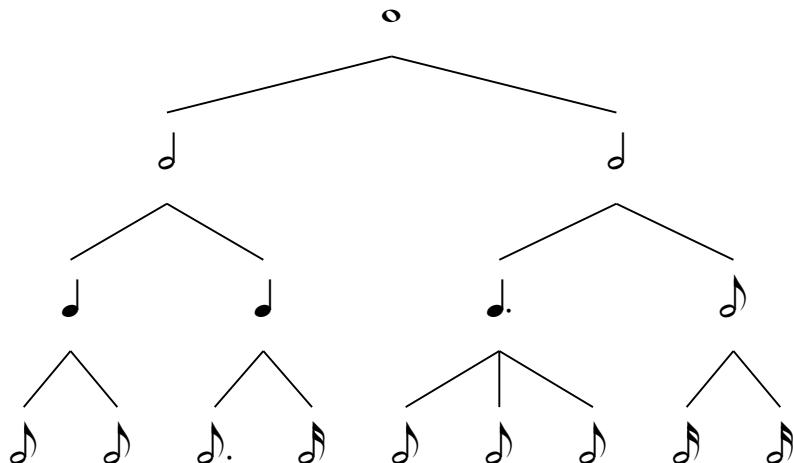
Jeżeli struna drga 261.63 razy w ciągu sekundy, mówimy że drga z częstotliwością 261.63 Hz. Taką częstotliwość posiada środkowy dźwięk C. Standard ten został przyjęty w 1955 roku przez Międzynarodową Organizację Normalizacyjną[71]. Standardowa klawiatura fortepianu posiada 88 klawiszy częstotliwości poszczególnych klawiszy są w relacji z dźwiękiem C, relacje możemy opisać za pomocą prostego wzoru:

$$freq(C) * 2^{\frac{s}{12}} \quad (\text{A.1})$$

gdzie, $freq(C)$ oznacza częstotliwość dźwięku C, a s to liczba z przedziału $< -39, 48 >$, która oznacza interwał do danej nuty od nuty C. Każdy z 12 półtonów ma przyporządkowaną literę.

A.2 Wartości nut

Podstawową oraz największą wartością czasową w muzyce jest cała nuta (•), która odpowiada podstawowej wartości rytmicznej. Półnuta to nuta o czasie trwania połowie całej nuty (♩). Ćwierćnuta trwa $\frac{1}{4}$ całej nuty, czyli połowę półnuty (♪). Ósemka to nuta o czasie trwania $\frac{1}{8}$ całej nuty (♪). Szesnastka trwa $\frac{1}{16}$ całej nuty (♪). Różnice w wartościach czasowych nut ilustruje poniższy przykład. Kropka po nucie zwiększa jej czas trwania o połowę.



Rysunek A.2: Podział wartości rytmicznych nut

Wartości te są wynikiem podziału dwójkowego. Zasada tego podziału polega na dzieleniu większej wartości czasowej na dwie mniejsze. Obok podziału dwójkowego istnieje również podział trójkowy.

A.3 Metrum

Utwór muzyczny jest podzielony w grupy za pomocą taktów. Metrum określa schemat rytmiczny taktu. Za jego pomocą określone jest z ilu i jakich jednostek rytmicznych składa się takt. Metrum zapisywane jest na początku tekstu w postaci pary cyfr. Dolna cyfra odpowiada za rodzaj jednostki rytmicznej w takcie (1 - •, 2 - ♩, 4 - ♪). Popularne metra muzyczne:

- $\frac{4}{4}$ - używane w muzyce klasycznej i pop

- $\frac{2}{2}$ - używane w marszach i muzyce orkiestrowej
- $\frac{2}{4}$ - używane w polkach
- $\frac{3}{4}$ - używane w walcach i balladach



Rysunek A.3: Przykład zapisu metrum na pięciolinii

A.4 Melodia i harmonia

Podczas słuchania muzyki słyszać jednocześnie wiele dźwięków. Melodię można opisać jako dźwięki słyszane kolejno - zwykle jest grana przez prawą rękę w kluczu wiolinowym, a harmonię jako dźwięki słyszane w tym samym czasie (akordy) - zwykle grana przez lewą rękę w kluczu basowym.

Szereg dźwięków ułożonych według stałego schematu nazywa się skalą. Natomiast skalę muzyczną rozpoczynającą się od określonego dźwięku nazywamy gamą. W muzyce można wyróżnić kilka podstawowych skali:

- C major - C, D, E, F, G, A, B
- D major - D, E, F♯, G, A, B, C♯
- C naturalna minor - C, D, D♯, F, G, G♯, A♯
- C melodyczna minor - C, D, D♯, F, G, A, B
- C pentatonika - C, D, F, G, A

Skale są definiowane przez początkową wartość. Wyróżnia się system dur-moll, który jest systemem tonalnym. Tworzą go dwa podstawowe rodzaje skali: durowe (majorowa) i molowa (minorowa). Muzyka pisana w tonacji durowej posiada wyraźny charakter pogodny i wesoły, natomiast muzyka pisana w tonacji molowej posiada wyraźny charakter smutny.

A.5 Transpozycja

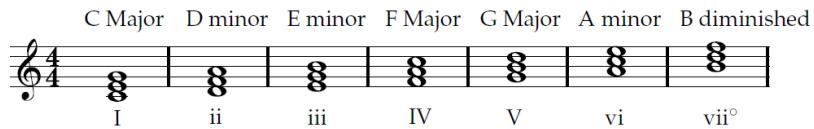
Transpozycja jest procesem, którego celem jest przeniesienie całego utworu muzycznego lub jego części o ustalony interwał w górę lub w dół. Efektem tego procesu jest zmiana tonacji utworu. Poniżej przykład linii melodycznej, która została przetransponowana o trcję wielką w dół.



Rysunek A.4: Melodia w pierwszej linii jest w tonacji D-dur, natomiast po transpozycji w tonacji B♭

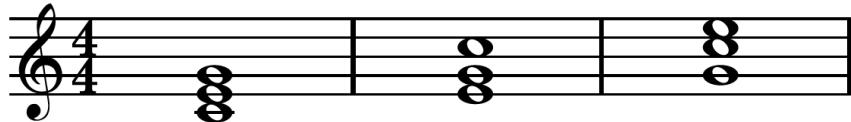
A.6 Akordy

Akordem nazywamy współbrzmieniem co najmniej trzech dźwięków o różnej wysokości i nazwie. Akordy budowane są na różnych stopniach gamy zgodnie z zasadą istnienia szeregu harmonicznego czyli tercjami. Dźwięki wchodzące w skład akordu nazywane są składnikami akordu. Pierwszy składnik nazywany jest prymą, drugi - tercją, trzeci - kwintą, czwarty - septymą, piąty - noną, szósty - undecymą, siódmy - terdecymą.



Rysunek A.5: Przykłady akordów na różnych stopniach gamy C-dur

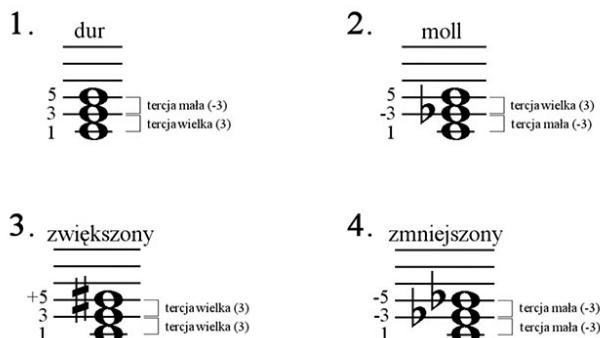
Jeżeli najniższy dźwięk akordu nie jest jego podstawą to mówimy o jego przewrocie. Poniżej zostały przedstawione dwa przewroty akordu C-dur.



Rysunek A.6: Przewroty akordu C-dur

Akord może występować w jednym z czterech podstawowych typów co ilustruje rysunek poniżej:

TYPY AKORDÓW

*Rysunek A.7: Typy akordów***A.7 Progresje**

Progresją akordów nazywamy uporządkowaną sekwencję akordów. Progresje akordów w utworach muzycznych są zwykle powtarzane kilka razy. W muzyce jazzowej korzysta się z tak zwanej progresji II - V - I, w której akordy budowane są kolejno na drugim stopniu gamy, piątym i pierwszym. Taka sekwencja jest powtarzana cztery razy. Po głębszej analizie harmonii dzieł zachocniczych dostrzec można schematy w sposobach łączenia akordów.

Bibliografia

- [1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*, MIT Press, 2016
- [2] David Cope. *Experiments in music intelligence*, University of California, 1987
- [3] George Tzanetakis. *Bayes and Markov Listen to Music*, University of Victoria, 2017
- [4] Douglas McIlwraith, Haralambos Marmanis, Dmitry Babenko. *Inteligentna Sieć, wydanie 2*, Helion, 2017
- [5] Franciszek Wesołowski. *Zasady muzyki*, Polskie Wydawnictwo Muzyczne SA, Kraków 2010
- [6] Bogdan Hołownia. *O harmonii jazzowej: zapiski z szuflady*, Polskie Wydawnictwo Muzyczne SA, 2014
- [7] Bin Cui, Jialie Shen, John Shepherd. *Intelligent Music Information Systems: Tools and Methodologies*, Idea Group Reference; 1 edition (August 23, 2007)
- [8] Christophear Olah <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [9] Shi Yan *Understanding LSTM and its diagrams*,
<https://medium.com/@shiyuan/understanding-lstm-and-its-diagrams-37e2f46f1714>
- [10] Walter Schulze. *A Formal Language Theory Approach To Music Generation*, Department of Mathematics, Applied Mathematics and Computer Science, University of Stellenbosch, 2016
- [11] Douglas Eck, Jurgen Schmidhuber. *A First Look at Music Composition using LSTM Recurrent Neural Networks*, Technical Report No. IDSIA-07-02
- [12] Andrzej Blikle. *Automaty i gramatyki: wstęp do lingwistyki matematycznej*, PWN, 1971
- [13] M. Chemillier. *Toward a formal study of jazz chord sequences generated by Steedman's grammar*, Soft Computing 8 (2004) 1 – 6 Springer-Verlag 2004
- [14] Nicolas Privault. *Understanding Markov Chains - Examples and Applications*, Springer, 2013
- [15] Erlijn J. Linskens. *Music Improvisation using Markov Chains*, University College Maastricht Maastricht University, 2014
- [16] Walter Schulze and Brink van der Merwe Stellenbosch University. *Music Generation with Markov Models*, IEEE, 2011
- [17] Adam Jakubowski. Repetytorium z przedmiotu "Miara i prawdopodobienstwo" dla kierunku "Informatyka" 2002/2003

- [18] Adam Jakubowski. *Statystyka i eksploracja danych Repetytorium z teorii prawdopodobienstwa*, UMK, 2011
- [19] Holický, Milan. *Introduction to Probability and Statistics for Engineers*, Springer, 2013
- [20] David Wright. *Mathematics and Music*, 2009
- [21] Gunhild Elisabeth Berget. *Using Hidden Markov Models for Musical Chord Prediction*, Norwegian University of Science and Technology, 2017
- [22] Sebastian Raschka. *Python Machine Learning*, Packt, 2015
- [23] Jason Brownlee. *Long Short-Term Memory Networks With Python*, 2017
- [24] François Chollet, *Deep Learning with Python*, MANNING, 2018
- [25] Nikhil Buduma. *Fundamentals of Deep Learning*, O'Reilly, 2017
- [26] Nipun Agarwala, Yuki Inoue, Axel Sly. *Music Composition using Recurrent Neural Networks*, Stanford University
- [27] Keunwoo Choi, George Fazekas, Mark Sandler. *Text-based LSTM networks for Automatic Music Composition*, Queen Mary University of London
- [28] Antonio Gulli, Sujit Pal. *Deep Learning with Keras*, Packt, 2017

Przypisy

- [1] Hiller, Lejaren Arthur, and Leonard M. Isaacson. Experimental Music; Composition with an electronic computer. Greenwood Publishing Group Inc., 1979
- [2] Horner, Andrew, and David E. Goldberg. "Genetic algorithms and computer-assisted music composition." (1991): 337-441
- [3] http://www.cs.colorado.edu/mozer/Research/Selected_Publications/reprints/Mozer1994.pdf
- [4] <http://people.idsia.ch/juergen/blues/IDSIA-07-02.pdf>
- [5] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768>
- [6] http://www.kms.polsl.pl/mi/pelne_9/31.pdf
- [7] https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm
- [8] https://en.wikipedia.org/wiki/LZ77_and_LZ78
- [9] https://www.youtube.com/watch?v=mXIJO-af_u8

- [10] Iannis Xenakis. *Formalized Music Thought and mathematics in composition*, Pendragon Press, 1992
- [11] <http://www.musicxml.com/>
- [12] https://en.wikipedia.org/wiki/Document_type_definition
- [13] <https://www.finalemusic.com/>
- [14] <http://www.avid.com/sibelius>
- [15] <https://musescore.org/pl>
- [16] <https://medium.com/@omgimanerd/generating-music-using-markov-chains-40c3f3f46405>
- [17] <https://github.com/omgimanerd/markov-music/>
- [18] <https://dzone.com/articles/algorithm-week-generate-music>
- [19] <https://mdoege.github.io/PySynth/>
- [20] <http://github.com/jcbozonier/MarkovMusic/tree/master>
- [21] <https://github.com/johnmryan/music-generator>
- [22] Ian Goodfellow, Yoshua Bengio, Aaron Couville, *Deep Learning Systemy uczące się*, PWN, Warszawa 2018
- [23] https://sebastianraschka.com/Articles/2015_singlелayer_neurons.html
- [24] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943
- [25] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957
- [26] <https://arxiv.org/pdf/1211.5063.pdf>
- [27] Learning representations by back-propagating errors, D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature, 323: 6088, strony 533–536, 1986
- [28] <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- [29] Sebastian Raschka, *Python Machine Learning*, Packt Publishing; 1 edition (September 23, 2015), 540
- [30] Backpropagation Through Time: What It Does and How to Do It (Paul Werbos, Proceedings of IEEE,

78(10):1550-1560, 1990).

[31] <https://arxiv.org/pdf/1211.5063.pdf>

[32] <http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf>

[33] <http://www.bioinf.jku.at/publications/older/2604.pdf>

[34] Sepp Hochreiter, Jurgen Schmidhuber, *Long Short-Term Memory*, Neural Computation

[35] In addition to the original authors, a lot of people contributed to the modern LSTM. A non-comprehensive list is: Felix Gers, Fred Cummins, Santiago Fernandez, Justin Bayer, Daan Wierstra, Julian Togelius, Faustino Gomez, Matteo Gagliolo, and Alex Graves.

[36] Learning to Forget: Continual Prediction with LSTM, F. Gers, J. Schmidhuber, and F. Cummins, Neural Computation 12, 2451-2471, 2000

[37] Jose Portilla, *Complete Guide to TensorFlow for Deep Learning with Python*

[38] Jose Portilla, *Complete Guide to TensorFlow for Deep Learning with Python*

[39] Jose Portilla, *Complete Guide to TensorFlow for Deep Learning with Python*

[40] Jose Portilla, *Complete Guide to TensorFlow for Deep Learning with Python*

[41] <http://arxiv.org/pdf/1406.1078v3.pdf>

[42] <http://proceedings.mlr.press/v37/jozefowicz15.pdf>

[43] <https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>

[44] <https://github.com/tensorflow/nmt>

[45] <https://github.com/tensorflow/nmt>

[46] <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

[47] http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

[48] <https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/>

[49] Dropout: a simple way to prevent neural networks from overfitting, Nitish Srivastava and. others, Journal of Machine Learning Research 15.1, pages 1929-1958, 2014,
<http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>).

[50] Srivastava, Nitish, et al. *Dropout: a simple way to prevent neural networks from overfitting*, JMLR 2014

- [51] <https://github.com/ybayle/awesome-deep-learning-music/blob/master/dl4m.bib>
- [52] <https://magenta.tensorflow.org/>
- [53] <https://experiments.withgoogle.com/ai-duet>
- [54] <https://ai.google/research/teams/brain>
- [55] <https://deepjazz.io/>
- [56] <http://bachbot.com/>
- [57] <http://www.flow-machines.com/>
- [58] <https://deepmind.com/blog/wavenet-generative-model-raw-audio/>
- [59] <https://github.com/ibab/tensorflow-wavenet>
- [60] <https://magenta.tensorflow.org/2016/09/23/learning-music-from-learned-music>
- [61] <https://github.com/MattVitelli/GRUV>
- [62] <https://cloud.google.com/>
- [63] <https://aws.amazon.com/>
- [64] <https://github.com/Syllo/nvtop>
- [65] https://www.tensorflow.org/guide/summaries_and_tensorboard
- [66] Yoav Zimmerman, *A Dual Classification Approach to Music Language Modeling*, 2016
- [67] <https://arxiv.org/pdf/1702.07787.pdf>
- [68] <https://arxiv.org/pdf/1406.2661.pdf>
- [69] <https://salu133445.github.io/musegan/>
- [70] https://pl.wikipedia.org/wiki/Plik:Scales_and_keyboard.png
- [71] Randel, D.M. (2003 November). The Harvard Dictionary of Music. 4th edn. Belknap Press of Harvard University Press.