Spis treści

Wstęp

O algorytmicznym komponowaniu muzyki myślano na długo przed powstaniem szybkich komputerów. Jako pionierów generowania sztucznej muzyki uważa się Hillera i Isaacsona [1]. Obaj panowie jako pierwsi w 1957 roku użyli komputera do stworzenia utworu muzycznego. 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]. 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 cytuje Xenakisa i Lejarena Hillera jako swoje inspiracje. Łańcuchy Markowa mogą jedynie generować podsekwensje, które pochodzą z utworów, na podstawie których łańcuch został wytrenowany. 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 i kliku innych naukowców.

[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

Rozdział 1

Łańcuchy Markowa

1.1 Podstawowe pojęcia i definicje

W tym podrozdziale dokonamy przeglądu kliku 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 $\mathcal{A}^{\mathcal{C}} \in \mathcal{F}$.
- Jeżeli $\mathcal{A}_1, \mathcal{A}_2, ..., \in F$, 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 $\mathcal{R} = \{A_1, A_2, ..., A_n\}$ 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 $(\Omega, \mathcal{F})$, 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 $(\Omega, \mathcal{F})$ stanowią przykłady przestrzeni mierzalnych:

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

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

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

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

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

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

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

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

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*.
- $P : \mathcal{F} \to [0, 1]$ jest prawdopodobieństwem na $(\Omega, \mathcal{F})$.

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:

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

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 $X : (\Omega, \mathcal{F}) \to \mathbb{R}^1$ 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: $\Omega = \{(i,j); 1 \leqslant i,j \leqslant 6\}$. Rozważmy funkcję $X : \Omega \to \mathbb{R}^1$. Określoną wzorem:

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

Symbolem $\mathcal{F}$ oznaczmy wszystkie podzbiory zbioru Ω. Wówczas trójka $(\Omega, \mathcal{F}, \mathcal{P})$ gdzie $P : \mathcal{F} \to [0,1]$ 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,
- $P : \mathcal{F} \to [0,1]$ jest prawdopodobieństwem na $(\Omega, \mathcal{F})$
- $P(\emptyset) = 0$
- $P(\Omega) = 1$
- $P(A) = p$ dla $0 \leqslant p \leqslant 1$

Wówczas:

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

1.2 Łańcuch Markowa i jego cechy

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 $\mathbb{R}$ nazywamy ciąg zmiennych losowych X_0, X_1, X_2,... taki, że:

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

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

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 $X_0, X_1, ..., x_{t-2}$

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 .

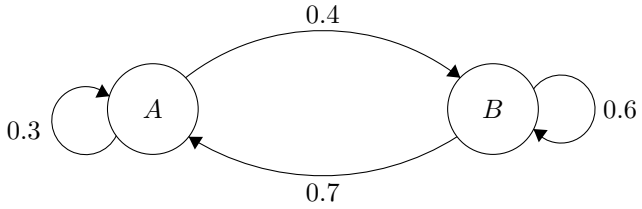
$$M = \begin{bmatrix} p_{0,0}(t) & p_{0,1}(t) & p_{0,2}(t) & \dots & p_{0,j}(t) \\ p_{1,0}(t) & p_{1,1}(t) & p_{1,2}(t) & \dots & p_{1,j}(t) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{i,j}(t) & x_{i,j}(t) & p_{i,j}(t) & \dots & p_{i,j}(t) \end{bmatrix}$$

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, ze stanu B istnieje sześćdziesięcio procentowe prawdopodobieństwo, że zostanie osiągnięty z powrotem stan B. Macierzy M odpowiada automat z przejściami przedstawiony poniżej.



Inną równoważną definicją 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,
- $p : Q \times Q \to [0, 1]$ - prawdopodobieństwo osiągnięcia stanu w momencie przejścia,
- $\pi : Q \to [0, 1]$ - 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:

$P(X_{t_2} = 0|X_{t_2} = 1) = 0.3 = p_{1,0}$
$P(X_{t_2} = 1|X_{t_2} = 1) = 0.7 = p_{1,1}$
$P(X_{t_2} = 1|X_{t_2} = 0) = 0.2 = p_{0,1}$
$P(X_{t_2} = 0|X_{t_2} = 0) = 0.8 = p_{0,0}$

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 ogólniej rozkład zmiennej X_{t+s}. Taki rozkład można łatwo znaleźć korzystając z macierzy M.
Ze wzoru na prawdopodobieństwo całkowite wynika:

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

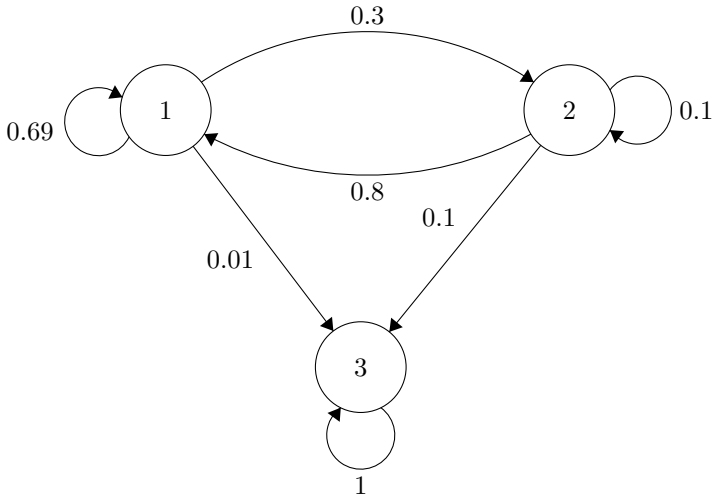
Mówiąc prościej, mamy tutaj do czynienia z mnożeniem macierzy przez wektor, co można zapisać: $\pi(t) \cdot M = \pi(t + 1)$

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.

$P(X_n = 1|X_n = 2) = 0.3 = p_{2,1}$
$P(X_n = 1|X_n = 1) = 0.6 = p_{1,1}$
$P(X_n = 2|X_n = 1) = 0.8 = p_{1,2}$
$P(X_n = 2|X_n = 2) = 0.1 = p_{2,2}$

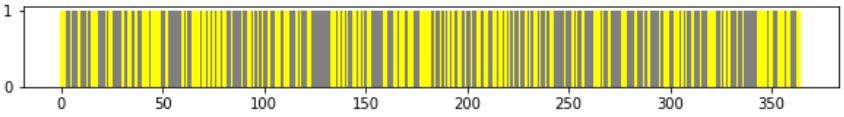
Prawdopodobieństwa można zobrazować w postaci automatu z przejściami:



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 będzie 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. $S, S, C, C, S, S, S, C, S, C, S, C, ..., n$. Mając rozkład w postaci symboli można go zobrazować w postaci wykresu.

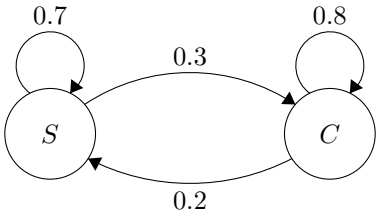


Rysunek 1.1: Wykres prawdopodobieństwa

Powyższa ilustracja obrazuje działanie modelu, pogoda każdego dnia jest niezależna od poprzednich dni, pobierana jest również z tego samego rozkładu. W następnym 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:

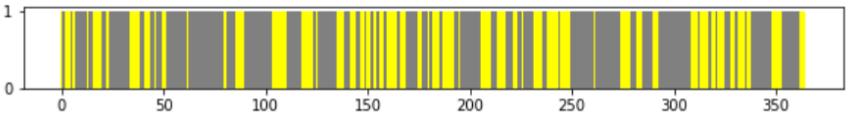
$$P = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix}$$

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



Rysunek 1.2: Graf prawdopodobieństw równoważny z macierzą przejść

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
23         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 Cechy Łańcuchów Markowa

W teorii prawdopodobieństwa oraz statystyce rozważa sie inne modele matematyczne, które bazują na probabilistyce 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,
- $a : Q \times Q \to [0, 1]$ - prawdopodobieństwo osiągnięcia stanu pomiędzy stanami ukrytymi,
- $b : Q \times \sum \to [0, 1]$ - prawdopodobieństwo emisji określonego symbolu alfabetu w danym ukrytym stanie,
- $\pi : Q \to [0, 1]$ - 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 $X = X_1, ..., X_n$ i $s = s_1, ..., s_n$, gdzie $X_i \in \sum$ oraz $s_i \in Q$. W tym przypadku używamy następującej notacji:

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

$P(x)$ 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 każda 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

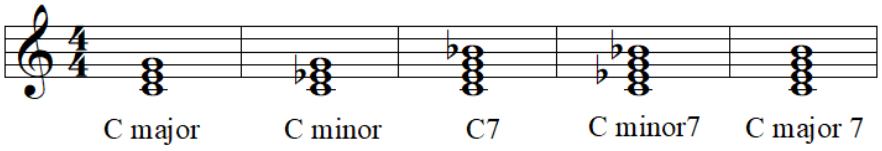
kostką rzucamy zależy tylko i wyłącznie od stanu ukrytego łańcucha Markowa. Macierz przejścia zdefiniujemy następująco:

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

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:

$$\prod = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{2} \end{bmatrix}$$

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 odpowiedzialne 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 odpowiedzialna 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 *music*21 będzie można wygenerować akordy na pięciolinii.

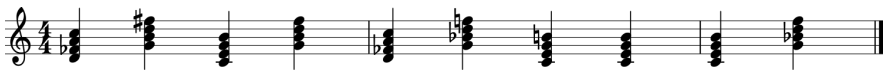
```
1  import numpy as np
2  from hmmlearn import hmm
3  from music21 import *
```

```
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

Możemy powiedzieć, że standardowe Łańcuchy Markowa maja 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:

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

Jeżeli $t \leqslant N$ to q_t jest stanem startowym oraz nadmierne stany q_t gdzie $t \leqslant 0$ są reprezentowane przez puste słowo λ. W przypadku Łańcucha Markowa N-tego rzędu będziemy używać zapisu $a(q_{t-N}, ..., q_{t-1}, q_t)$ 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 łańcuchów Markowa uwidacznia się w przypadku, gdy nie można przyjąć założenia o niezależności zdarzeń i zmiennych losowych. Do zalet prognozowania na podstawie łańcuchów 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 łańcuchów Markowa jest użycie ich w popularnym algorytmie wyszukiwarki **Google - PageRank**. Algorytm można interpretować jako znajdowanie ustalonego stanu z łańcucha Markowa. W tym przypadku łańcuch 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 stronę i $\frac{1-\alpha}{N}$, które nie są połączone, gdzie α jest stałą równą 0.85[1]. Łańcuch 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 łańcuchu Markowa. Czasami aplikacje proszą o możliwość dostępu np. do emaili aby na ich podstawie zbudować bazę prawdopodobieństw. Mając bazę prawdopodobieństw, czyli w tym wypadku łańcuch Markowa, aplikacje wykorzystują model probabilistyczny n-gram, który jest wykorzystywany do pobierania kolejnego elementu z sekwencji łańcucha 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 [2].

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 osobników 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).

[1] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768
[2] http://www.kms.polsl.pl/mi/pelne_9/31.pdf

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 algorytmy do kompresji danych jest bezstratny algorytm LZMA[3], który łączy łańcuchy Markowa z kompresją Lempel-Ziv[4] aby osiągnąć bardzo wysokie współczynniki kompresji. Jako prekursora teorii informacji uważa się Clauda 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. Przykładam jest wykorzystanie łańcuchów Markowa do egzogenicznego modelowania cen kapitału własnego w ogólnym ustawieniu równowagi. Innym przykładem jest sporządzenie przez agencje ratingowe rocznych tabeli 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[5]. Swoją pracę opisał w książce "Formalized Music: Thought and Mathematics in Composition"[6]

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 zastosowanie 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.

[3] https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm
[4] https://en.wikipedia.org/wiki/LZ77_and_LZ78
[5] https://www.youtube.com/watch?v=mXIJO-af_u8
[6] https://monoskop.org/images/7/74/Xenakis_Iannis_Formalized_Music_Thought_and_Mathematics_in_Composition.pdf

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 1.6: 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 dostępym formatem służącym do zapisu informacji o utworze muzycznym jest standard MusicXML[7]. Jest to znacznikowy format prezentacji graficznej notacji muzycznej, który oparty jest na wzorcach dokumentowych DTD[8]. Aby zrozumieć notację zapisu MusicXML wystarczy podstawowa znajomość języka XML, który jest dość powszechnie wykorzystywany. Format MusicXML jest wspierany przez ponad 230 programów służących do zapisu notacji muzycznej takich jak Finale[9], Sibelius[10] czy MusicScore[11]. 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>
```

[7] http://www.musicxml.com/
[8] https://en.wikipedia.org/wiki/Document_type_definition
[9] https://www.finalemusic.com/
[10] http://www.avid.com/sibelius
[11] https://musescore.org/pl

```
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      <atrributes>atrybut miary</atrributes>
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 `<identyfication>`, 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:

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

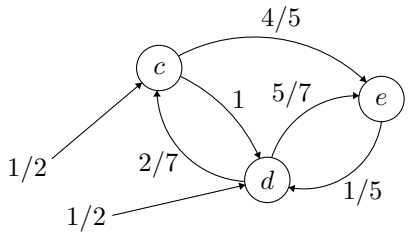
gdzie $\#(q_i \to q_j)$ jest ilością możliwych przejść ze stanu q_i do stanu q_j. Macierz prawdopodobieństw wygląda następująco:

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

oraz:

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

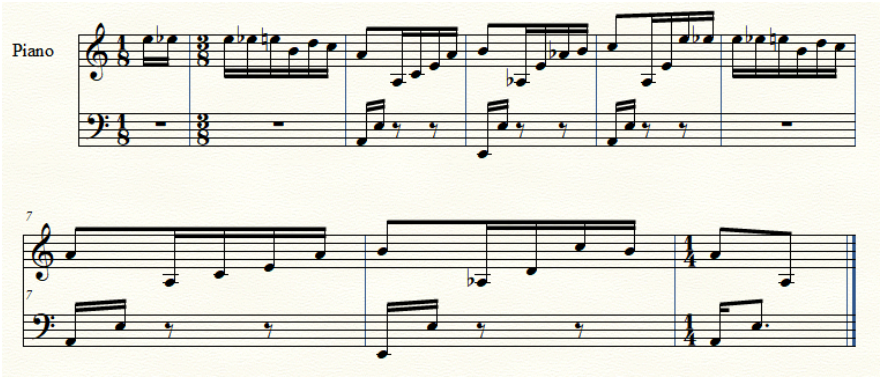
Graf prawdopodobieństw dla powyższej macierzy przedstawia się w sposób następujący:



Rysunek 1.7: Graf prawdopodobieństw równoważny z macierzą przejść

1.2.4 Przykłady

Alvin Lin w artykule pod tytułem: "Generating Musik Using Markov Chains"[12] opisał proces generowania muzyki z wykorzystaniem łańcucha Markowa na podstawie pliku MIDI. Wyniki jego rozważań można zobaczyć w praktyce uruchamiając przygotowane przez niego skrypty[13] 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 1.8: 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.

[12]https://medium.com/@omgimanerd/generating-music-using-markov-chains-40c3f3f46405
[13]https://github.com/omgimanerd/markov-music/

```
$ 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=36 notated_32nd_notes_per_beat=8 time=512>
<meta message time_signature numerator=1 denominator=4 clocks_per_click=24 notated_32nd_notes_per_beat=8 time=10752>
<meta message end_of_track time=972>
Track 1: Piano
<meta message device_name name=u'SmartMusic SoftSynth 1' time=0>
<meta message track_name name=u'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=7 value=85 time=0
control_change channel=0 control=10 value=51 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
```

Rysunek 1.9: Fragment pliku MIDI po dokonaniu analizy

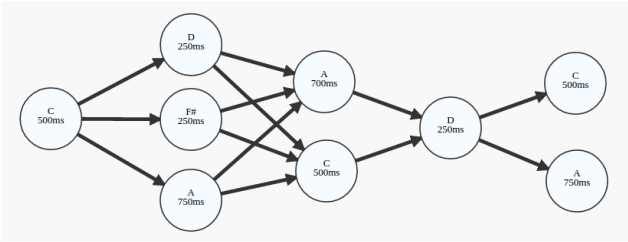
W dalszej części widać już poszczególne wartości nut razem z czasem oraz głośnością.

```
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 1.10: 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 1.11: 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	

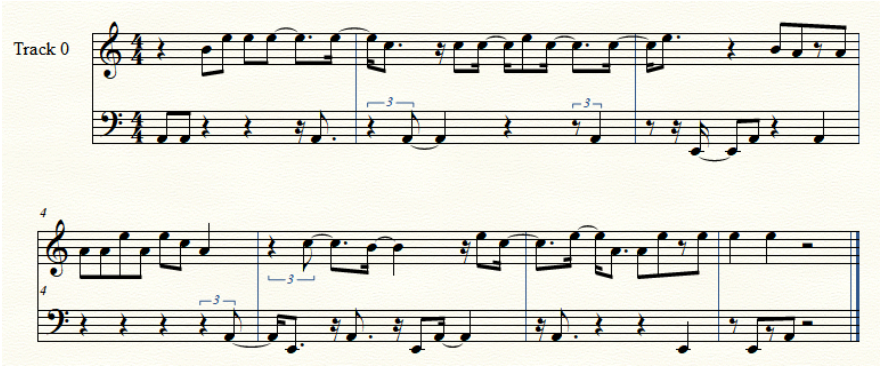
Listing 1.4: Macierz przejść

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

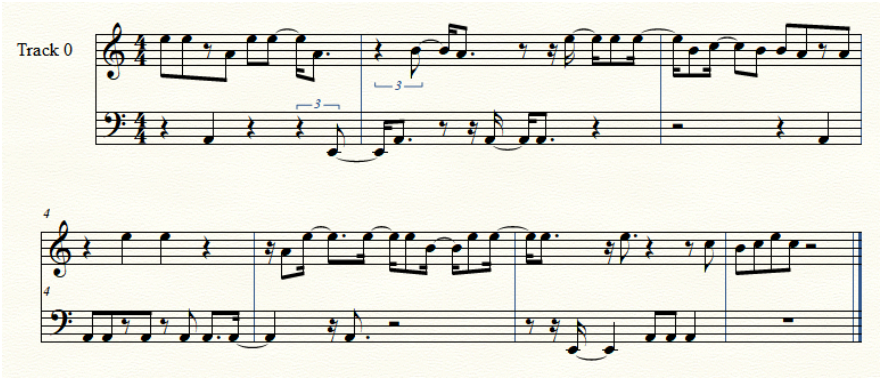
	40:250	71:250	45:250	72:250	76:250	69:250
64	2	2	0	0	0	0
68	0	0	1	1	0	0
76	0	0	0	0	2	0
72	0	0	1	0	0	1

Rysunek 1.12: 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 1.13: Wyniki - przykład 1



Rysunek 1.14: 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"[14]. W przykładzie wykorzystywana jest biblioteka PySynth[15] 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 półnutę, 4 ćwierć nutę, 8 ósemkę i 16 szesnastkę. Cały kod dostępny jest w serwisie GitHub[16].

```
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[17], 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 łańcuchy 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.

[14] https://dzone.com/articles/algorithm-week-generate-music
[15] https://mdoege.github.io/PySynth/
[16] http://github.com/jcbozonier/MarkovMusic/tree/master
[17] https://github.com/johnmryan/music-generator

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 $V = \{a_1, ..., a_n\}$. 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, różnicy. 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 $w = a_1...a_n \in A^*$ nazywamy słowo $\overline{w} = a_n...a_1$. Odbiciem zwierciadlanym języka $L \subset A^*$ nazywamy język $L = \{\overline{w} \in A^* : w \in L\}$

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 $G = \langle \sum, V, P, S \rangle$ gdzie:

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

Definicja 2.1.7. Produkcje gramatyki określają relację **bezpośredniego wyprowadzenia** $\Rightarrow$ na słowach nad alfabetem $(\sum \cup V)$ następująco:
x$\Rightarrow$y wtedy i tylko wtedy, gdy istnieją słowa x_0 i x_1 oraz reguła $\alpha \to \beta$ 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 $x_0, ..., x_n n \geqslant 0$, taki że $x_i \Rightarrow x_{i+1}$ dla $i = 0, ..., n-1$ 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: $A \to \alpha B$ lub $A \to \alpha$ dla $A, B \in V$ i $\alpha \in \sum *$

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 $X^* = \cup_{n=0} \infty X^n \in R$

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:

- $\{a, b\} \cdot \{a, b\} = \{aa, ab, bc, bb\}$ 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 konkatenacji języków skończonych

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

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

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

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

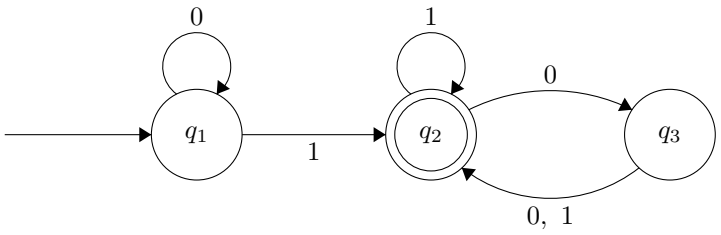
- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, Xw) = \delta^*(\delta(q, x), w)$

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

Twierdzenie 2.1.1. Dla dowolnego języka $L \subset \sum^*$ 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 $\underline{A} = \langle \sum, Q, F, \delta, q_o \rangle$, taki że $L = L(\underline{A})$ - to znaczy, że język L jest akceptowany przez $\underline{A}$

Automat jest urządzeniem, który posiada nieskończoną taśmę wejściową, służącą tylko do czytania słów nad alfabetem $\sum$. 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 $L = \{w \in \{0,1\}^* | w$ 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. Równoważnie 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 $A \to \beta$, gdzie $A \in V, \beta \in (\sum \cup V)^*$

Przykład 2.1.2. Gramatyka $G = (\{a, b\}, S, S \to aSb|\epsilon, S)$ 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 skończoną pamięcią, niezależną od długości danych wejściowych. W przypadku automatów ze stosem, dysponujemy także nieskończoną 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

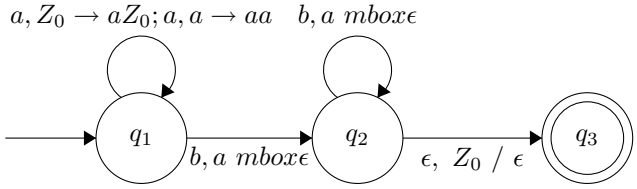
Maszyna będąc w określonym stanie czyta literę słowa wejściowego oraz sprawdza, jaki symbol znajduje się na wierzchołku stosu. Na tej podstawie podejmowana jest decyzja o zmianie stanu, następnie zdejmowany jest element z wierzchołka stosu. Na zdjętym elemencie umieszczane jest słowo złożone z symboli 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 $L \subset \sum^*$ jest generowany przez gramatykę bezkontekstową wtedy i tylko wtedy gdy L jest akceptowalny przez automat ze stosem.

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



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

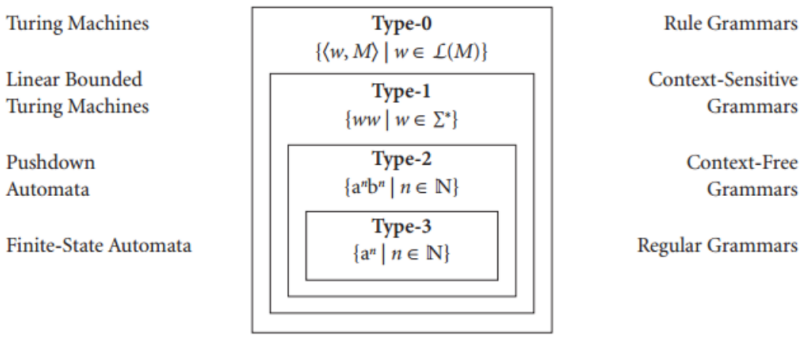
2.1.3 Hierarchia Chomsky'ego

Amerykański językoznawca Noam Chomsky zaproponował cztery typy gramatyk, które maja swoją hierarchę. Hierarchia Choomsky'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 $G = \langle \sum, V, P, S \rangle$ będzie dowolną gramatyką.

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

Poniższy diagram przedstawia graficzną reprezentacje języków formalnych.



Rysunek 2.1: Hierarchia języków formalnych

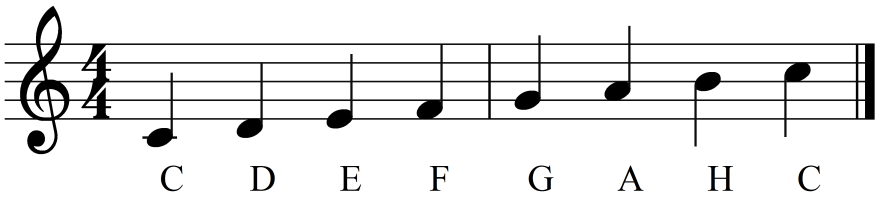
Definicja 2.1.15. Język należy do danej klasy wtedy i tylko wtedy, gdy jest możliwe zbudowanie gramatyki formalnej, która generuje dany język, a której reguły przestrzegają ograniczeń dla danej klasy.

2.2 Gramatyki a 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 Prosty przykład zastosowania gramatyki 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 oktaw (dwie oktawy: C, D, E, F, G, A, H, C, D, E, F, G, A, H). Metrum melodii czyli czynnik porządkujący ugrupowania rytmicznie za pomocą regularnie powtarzających się akcentów metrycznych wyniesie 4/4 (cztery czwarte). Górna liczba metrum oznacza ile ma być w takcie jednostek miarowych oznaczonych przez liczbę dolną. Przykładowo określenie 2/4 wskazuje, że w jednym takcie mają być dwie ćwierćnuty lub wartości dające w sumie dwie ćwierćnuty. 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 ułożona 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 $G = \langle \sum, V, P, S \rangle$ gdzie:

- $\sum = \{n.k; n \in \langle 0, 87 \rangle, k \in \langle 1, 3 \rangle\}$ - 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:
  $S \rightarrow Beg\ Mid\ End$
  $Beg \rightarrow 47'_1\ 49'_1\ 51'_1\ 52'_1$

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

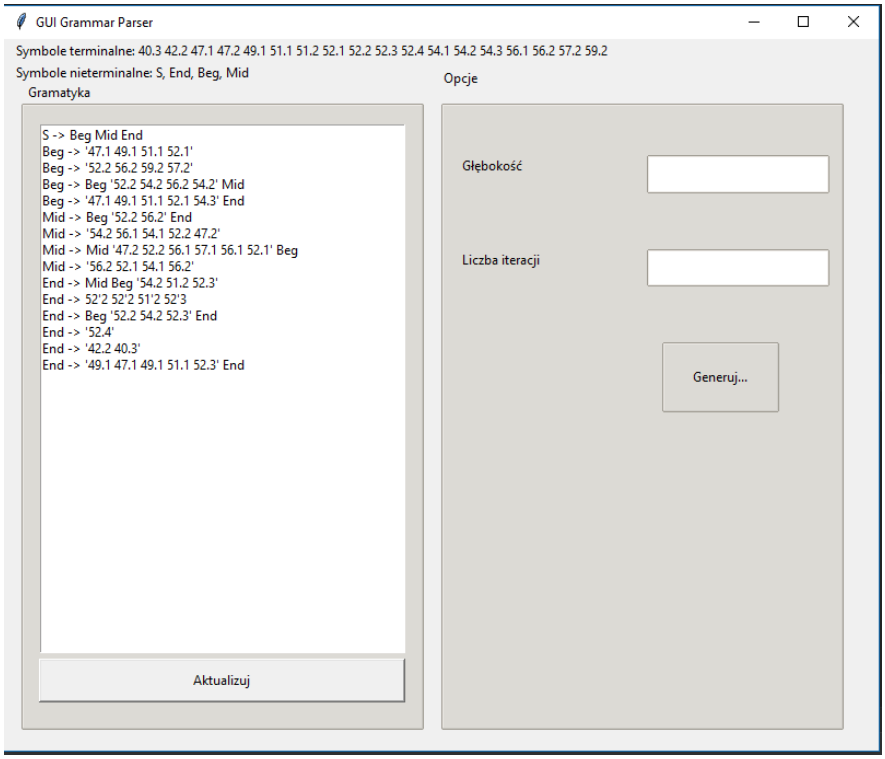
Za pomocą języka Python można zautomatyzować proces wyprowadzeń. W tym celu będzie pomocna biblioteka nltk, która służy do pracy z przetwarzaniem języka naturalnego.

```
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(gramma)
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
56  for i, k in enumerate(tracks_notes):
57   print(k)
58   conv.write(k, fmt='lilypond', fp='/mnt/c/Users/Lukasz/MGR_Code/Grammar/examples/' +
         str(i), subformats=['png', 'midi'])
```

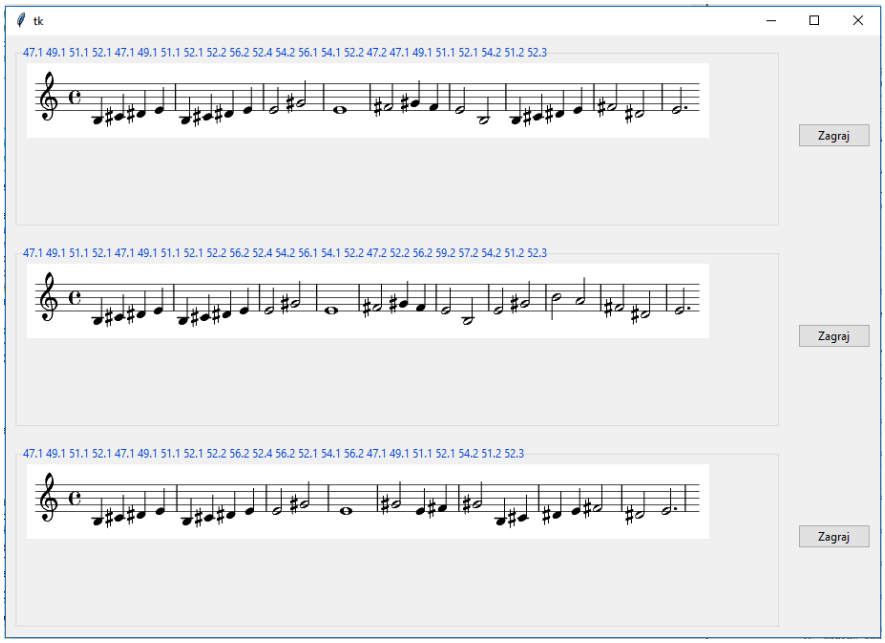
Listing 2.1: Generowanie wyprowadzeń z gramatyki bezkontekstowej

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 liczbie iteracji równej 3

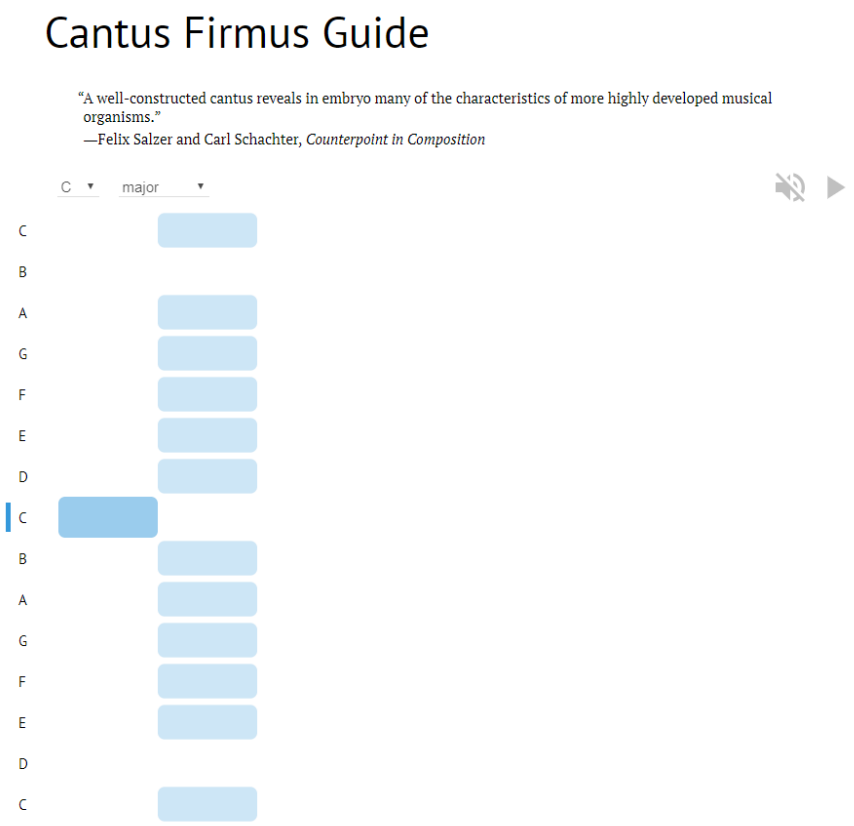
Po pomyślnym wyprowadzeniu słów program wyświetla wyniki w nowym oknie.

2.2.2 Programy wykorzystujące gramatyki bezkontekstowe

MusicMachine.io - system stworzony przez Johna Leszczynskiego. 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.



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 bezkonetkstowej, 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 Księżycowej:

```
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:

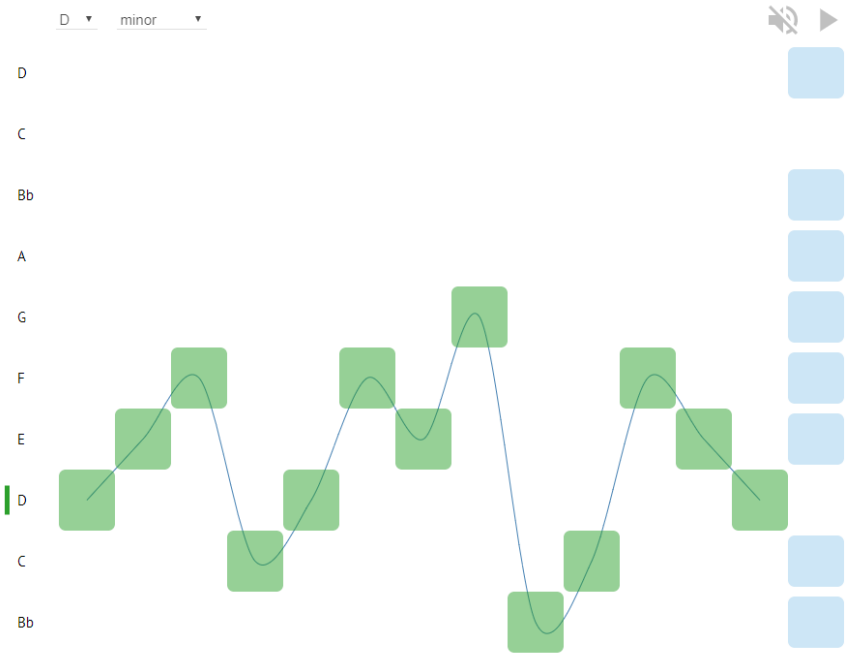
```
var upOnly = {

 // designed to be an infinite phrase
 UpPhrase: ['UpLeap DownStepPhrase UpPhrase',
 'UpStepPhrase DownPhrase'],

 // all choices must be prepared for a potential down leap in downphrase
 UpStepPhrase: ['Up2Phrase',
 'Up3Phrase',
 'UpLeapForwardPhrase'],

 // after 2, can reverse direction or continue up 2 or 3
 Up2Phrase: ['2',
 '2 Up2Phrase',
 '2 Up3Phrase'],

 // after 3, can reverse direction or continue up 2
 Up3Phrase: ['3',
 '3 Up2Phrase'],

 // up leap must be recovered with down step
 // prepare for a potential downard leap by adding another UpPhrase
 UpLeapForwardPhrase: ['2 UpLeapForward DownStepPhrase UpPhrase'],

 // allowed up leaps after already moving a second up
 UpLeapForward: ['4', '5'],

 // allowed leaps at the beginning or after a direciton change
 UpLeap: ['4', '5', '6', '8']
}
```

Symbole terminalne: 2, 3, 4, 5, 6, 8, oznaczają odległości od nut - interwały. Symbole nie-terminalne: UpPhrase, UpStepPhrase, Up2Phrase, Up3Phrase, UpLeapForwardPhrase, UpLearForward, 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:

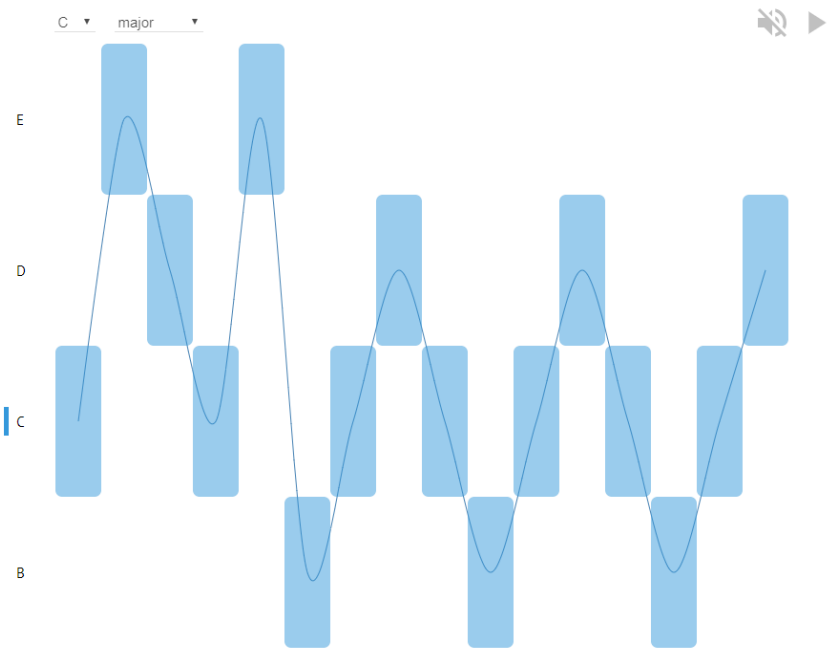
```
this.isValid = function () {
 var cf = this.cf()

 // is it long enough
 if (cf.length < MIN_CF_LENGTH || cf.length > MAX_CF_LENGTH) {
  return false
 }

 // is last note tonic?
 if (Pitch(cf[cf.length - 1]).pitchClass() !== guide.tonic()) {
  return false
 } else if (Pitch(cf[0]).pitchClass() === guide.tonic()) {
 // if first note is tonic, last note should end in the same octave
 // if first note is not tonic, this is probably a first species counterpoint
  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 CantusFirmus



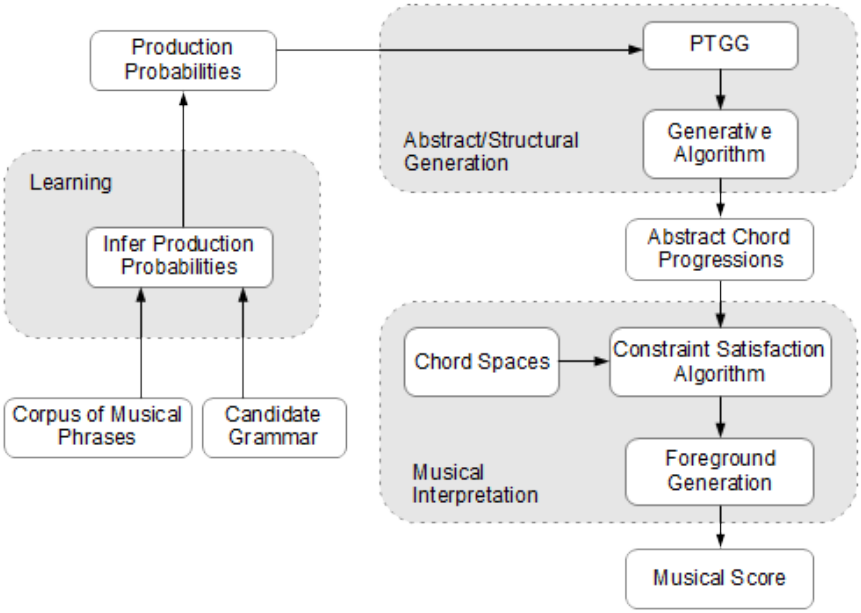
Rysunek 2.7: Nieprawidłowa sekwencja

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

```
var CantusFirmus = require('counterpoint').CantusFirmus
var cantus = new CantusFirmus('G major')
cantus.choices()     => ['G']
cantus.addNote('G4')

cantus.choices()     => [ 'A4', 'B4', 'C5', 'D5', 'E5', 'G5',
'F#4', 'E4', 'G3', 'B3', 'C4', 'D4' ]

cantus.addNote('E5')
cantus.choices()     => [ 'D5', 'C5' ]

cantus.addNote('D5')
cantus.choices()     => [ 'E5', 'F#5', 'G5', 'A5', 'B5',
'C5', 'B4', 'G4', 'A4' ]
cantus.addNote('F#5')
cantus.choices()     => [ 'G5', 'E5', 'F#4', 'A4', 'B4' ]

cantus.addNote('G5')
cantus.choices()     => [ 'A5', 'B5', 'F#5', 'E5', 'G4', 'B4', 'C5', 'D5' ]

cantus.addNote('B4')
cantus.choices()      => [ 'C5', 'D5' ]
cantus.addNote('C5')
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                                    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	$T^t \to T^t$	0.64	$D^t \to D^t$
0.30	$T^t \to T^{t/2}T^{t/2}$	0.09	$D^t \to D^{t/2}D^{t/2}$
0.16	$T^t \to D^{t/2}T^{t/2}$	0.27	$D^t \to S^{t/2}D^{t/2}$
0.12	$T^t \to T^{t/2}D^{t/2}$	0.95	$S^t \to S^t$
		0.05	$S^t \to S^{t/2}S^{t/2}$

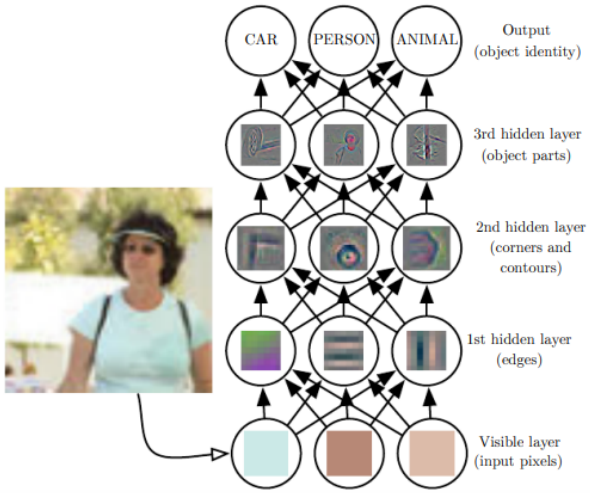
Rysunek 2.9: Przykład PTGG nad symbolami funkcji akordowych, tonika (T), dominanta (D) i subdominanta(S), sparametryzowana przez czas trwania jako indeks górny. Przedstawione prawdopodobieństwa produkcyjne pochodzą z głównej części potrzebnej do generowania chorałów

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 głosu, 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.

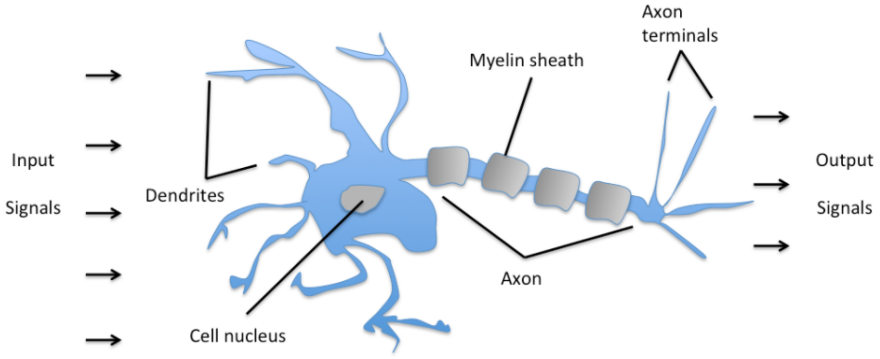


Rysunek 3.1:

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 porównań 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 zasadzę 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 następnych 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.

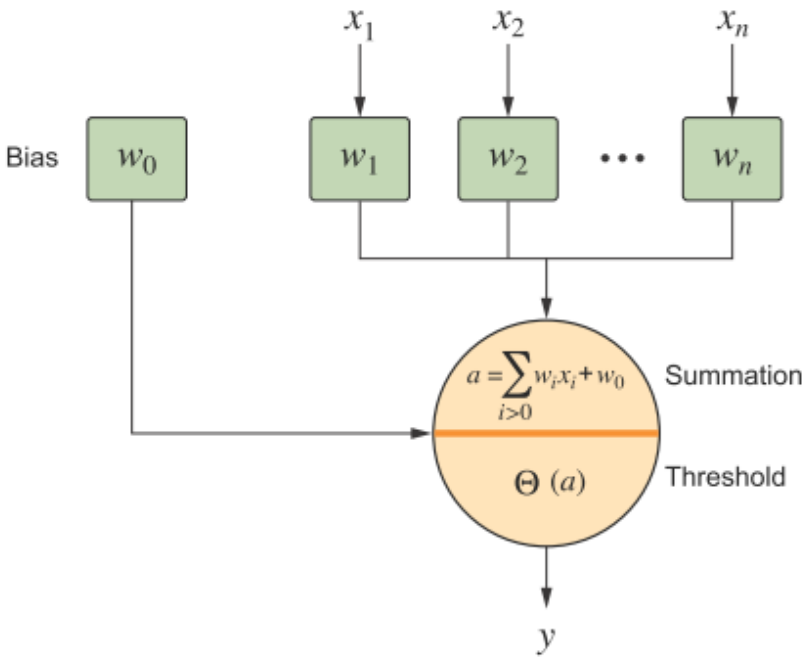


Rysunek 3.2: Schemat biologicznego neuronu

Warren McCulloch i Walter Pitts w roku 1943 opracowali model sztucznego neuronu, nazwali go **MCP** od McCullock-Pitts[1] swoje wyniki opisali w dziele o nazwie *A Logical Calculus of the Ideas Immanent in Nervous Activity.* Naukowcy przestawili 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. Klika lat później

[1]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

w roku 1953 amerykański uczony Frank Rosenblatt zaproponował sztuczną sieć neuronową[2], 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, 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 $a = w_1x_1 + ... + w_mx_n$, 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:

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

[2]F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957

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 nie).

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.

$$\phi(a) = \begin{cases} 1, & \text{if } a \geqslant \theta \\ -1, & \text{w przeciwnym wypadku} \end{cases}$$

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 wagi wartości wejściowych mogę 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.

Twierdzenie 3.2.1. Jeśli zbiór danych jest liniowo separowalny, a współczynnik szybkości uczenia η jest wystarczająco mały to algorytm uczenia perceptronu jest zbieżny.

Twierdzenie 3.2.2. Jeśli zbiór danych nie jest liniowo separowalny to algorytm zbiega lokalnie do minimalnego błędu średniokwadratowego.

3.3 Adaline - ADAptive LInear NEuron

Neuron bazujący na modelu Adaline [3] jest bardzo podobny do Perceptronu. Oba modele różnią się algorytmem uczenia. Autorami modelu są Bernard Widrow oraz jego student Tedd Hoff. Swoje badania opublikowali klika lat po algorytmie perceptronu stworzonym przez Franka Rosenblatta. Model Adaline pokazuje w jaki sposób zdefiniować i zminimalizować ciągłą funkcję kosztu. Dzięki czemu Adaline stanowi podstawę bardziej zaawansowanych algorytmów uczenia takich jak regresja logistyczna czy maszyny wektorów nośnych. Główna różnica, która rozróżnia neuron Adaline od perceptronu Rosenblatta to taka, że Adaline wykorzystuje ciągłą funkcję aktywacji do aktualizowania wag, ta reguła jest zwana jako reguła Widrow-Hoffa. Neuron typu Adaline porównuje sygnał wzorcowy d z sygnałem s na wyjściu liniowej części neuronu (sumatora), błąd można opisać wzorem $\delta = d - s$. Wartość ciągłej funkcji aktywacji posłuży aby obliczyć błąd modelu i zaktualizować wagi. W algorytmach uczenia nadzorowanego jedną z kluczowych składowych jest zdefiniowanie funkcji celu, którą trzeba zoptymalizować podczas procesu uczenia się. Funkcję celu często nazywa się funkcją kosztu J, którą trzeba zminimalizować. W przypadku Adaline uczenie neuronu sprowadza się do minimalizacji funkcji błędu średnio kwadratowego pomiędzy obliczonym wynikiem, a prawdziwą etykietą klasy.

$$J(w) = \frac{1}{2}\sum(y^{(i)} - \phi(z^{(i)}))^2$$

Ze względu na to, że 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.

3.4 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 $J(w)$.

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

[3] An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2, B. Widrow and others, Stanford Electron Labs, Stanford, CA, October 1960

Gdzie zmiana wagi Δw zdefiniowana jest jako:

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

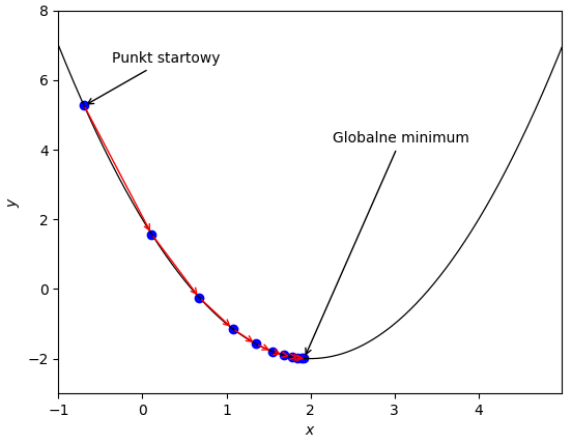
gdzie η to stała uczenia - szybkość schodzenia w dół. Aby obliczyć gradient funkcji kosztu potrzeba wyliczyć pochodną cząstkową funkcji kosztu dla danej wagi w_j:

$$\frac{\partial J}{\partial w_j} = -\sum (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{3.1}$$

Zatem reguła aktualizacji wag może wyglądać w następujący sposób:

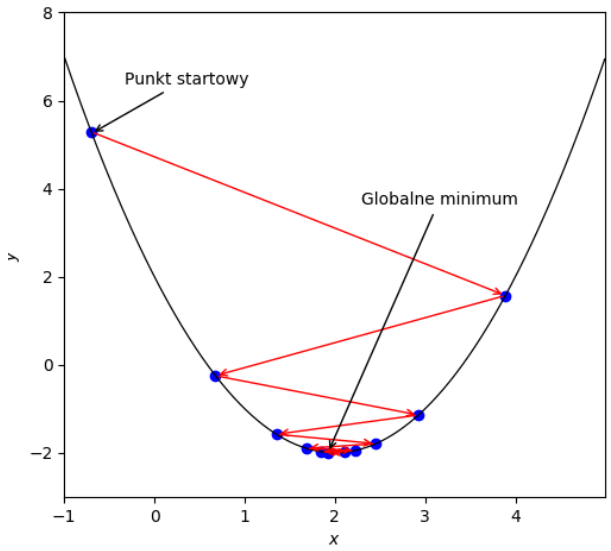
$$\delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{3.2}$$

Przykład 3.4.1. 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.



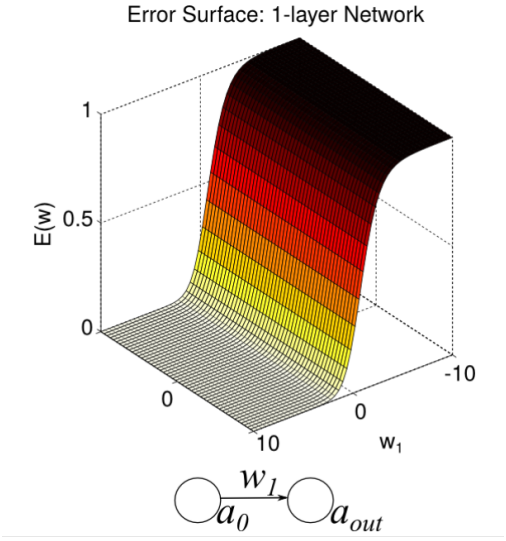
Rysunek 3.4: Metoda spadku gradientowego

Poniżej wynik działania algorytmu za dużą stałą uczenia η:

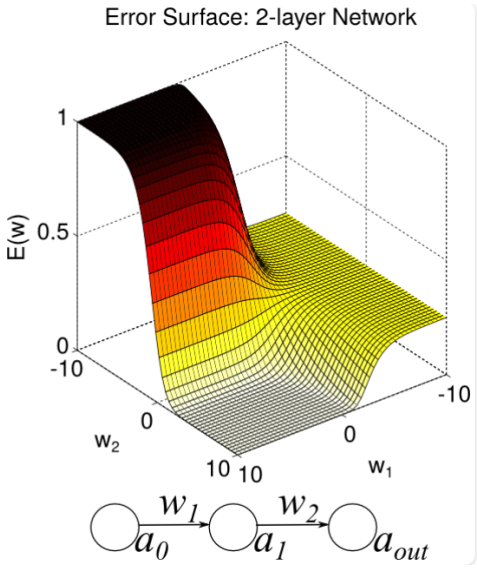


Rysunek 3.5: Za duży współczynnik uczenia

Przykład 3.4.2. 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

3.5 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 akceptujący 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. Załóżmy sytuacje, w której mamy trzy funkcje $f^{(1)}, f^{(2)} i f^{(3)}$ połączone w łańcuch, tworząc $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. Sieci wielowarstwowe najczęściej reprezentuje 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 wyniki głębokość modelu.

Twierdzenie 3.5.1. Każda funkcja boolowska może być reprezentowana przez sieć z jedną warstwą ukrytą, ale może wymagać wykładniczej liczby jednostek ukrytych.

Twierdzenie 3.5.2. Każda ograniczona funkcja ciągła może być aproksymowana z dowolnie małym błędem przez sieć z jedną warstwą ukrytą [Cybenko1989; Hornik et al. 1989]

Twierdzenie 3.5.3. 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 tytułem Perceptrons: An Introduction to Computational Geometry. W swojej książce przedstawili 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 3.1: 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 hiperpłaszczyzna, która pozwalałaby na rozdzielenie elementów klasy pozytywnej i negatywnej. Punkty można poprawnie rozdzielić przy pomocy więcej niż jednej hiperpłaszczyzny. Taką możliwość daje architektura sieci wielowarstwowej. Aby skonstruować sieć, która będzie wstanie nauczyć się problemu XOR wystarczy sieć dwuwarstwowa z jedną warstwą ukrytą. Sieć na wejściu przyjmuje dwie wartości, które potrzebne są do obliczenia funkcji XOR. Na wyjściu sieć zwraca jedną wartość 0 lub 1 w zależności od danych wejściowych. W tak opracowanej sieci każdy węzeł tworzy hiperpłaszczyznę. Wszystkie hiperpłaszczyzny są łączone przez końcowy perceptron. W efekcie otrzymujemy podprzestrzeń, która jest zbiorem wielościennym 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.

https://maviccprp.github.io/a-neural-network-from-scratch-in-just-a-few-lines-of-python-code/

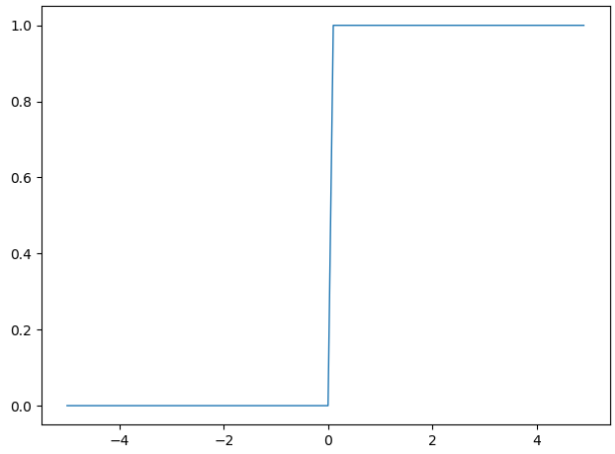
3.6 Funkcje aktywacji

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 aktywacji neuronu. Funkcja ta umożliwia otrzymanie na wyjściu sieci informacji postaci TAK - NIE.

$$\text{f(x)}= \begin{cases} 0, & x < 0 \\ 1, & x \geqslant 0 \end{cases} \tag{3.3}$$



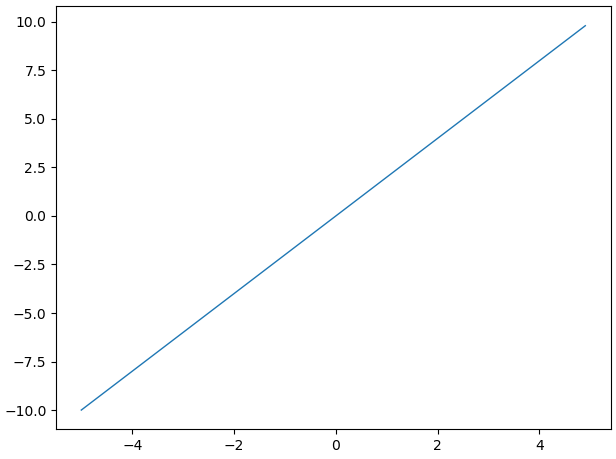
Rysunek 3.8: Funkcja skoku jednostkowego

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.

3.6.1 Funkcja liniowa

W przypadku funkcji liniowej sygnał wyjściowy przyjmuje postać:

$$y = cx \tag{3.4}$$



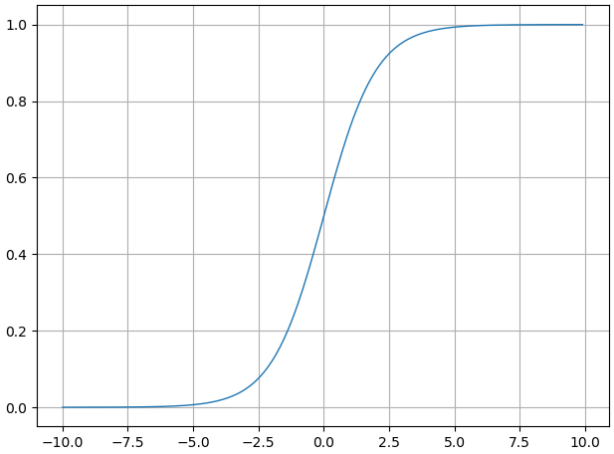
Rysunek 3.9: Funkcja liniowa

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. Mając funkcję $y = cx$, gdzie c to pewna stała pochodna w stosunku do x wynosi c. 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.

3.6.2 Funkcja sigmoidalna

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

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$



Rysunek 3.10:

Bardzo ważną cechą funkcji sigmoidalnej jest fakt, że jest różnoczkowalna. 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 zwracane przez funkcją 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[4].

3.6.3 Funkcja than

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

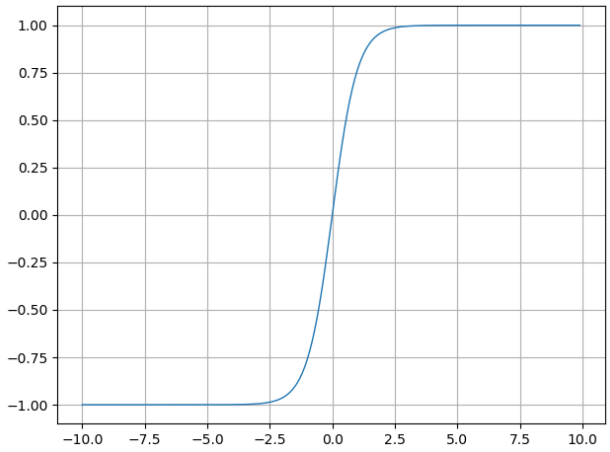
$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.6}$$

powyższe równanie można sprowadzić do wcześniej wspomnianej funkcji sigmoidalnej:

$$tanh(x) = 2sigmoid(2x) - 1$$

(3.7)

[4]https://en.wikipedia.org/wiki/Vanishing$_g radient_p roblem$



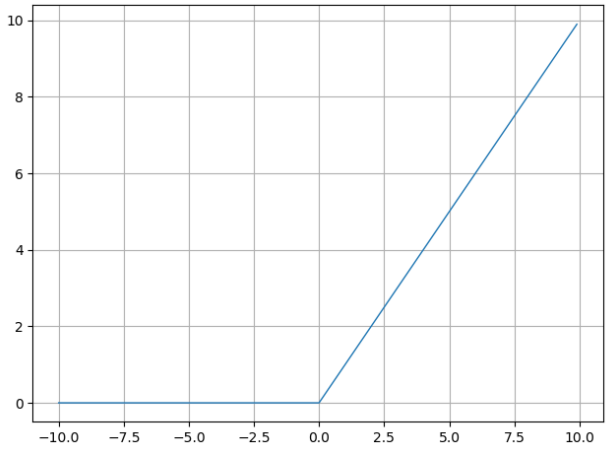
Rysunek 3.11: Wykres funkcji tanh

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 than występuje problem znikającego gradientu.

3.6.4 ReLu

$$f(x) = max(0, x) \tag{3.8}$$

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



Rysunek 3.12: Funkcja ReLu

Funkcja *ReLu* jest funkcją nieliniową, 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 ja rozważyć w przypadku projektowania głębokiej 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

3.7 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*[5] 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 przeciwieństwie 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 znaleźć 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 3.7.1. Algorytm propagacji wstecznej zbiega lokalnie do minimalnego błędu średniokwadratowego.

$$\frac{d}{dx}\big[f(g(x))\big] = \frac{df}{dg} \cdot \frac{dg}{dx} \tag{3.9}$$

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

[5]Learning representations by back-propagating errors, D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature, 323: 6088, strony 533–536, 1986

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx} \tag{3.10}$$

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. Wprowadźmy następujące oznaczenia:

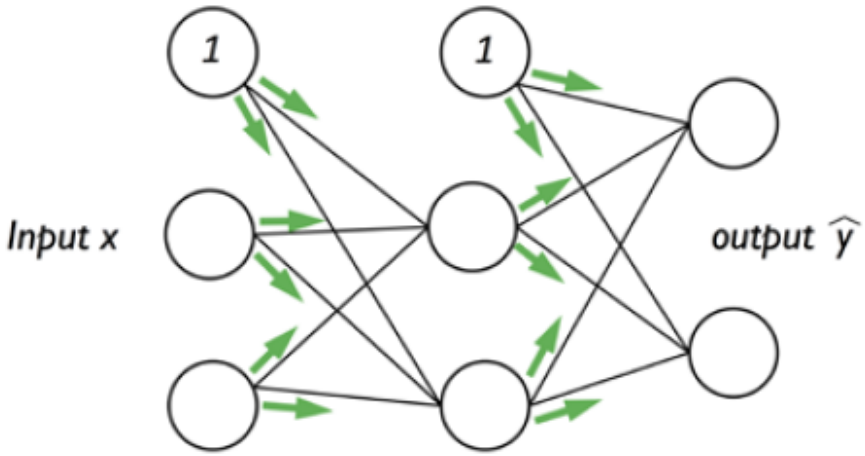
$$Z^{(h)} = A^{(wejscie)}W^{h}(\text{wejście dla warstwy ukrytej}) \tag{3.11}$$

$$A^{(h)} = \phi(Z^{(h)})(\text{funkcja aktywacji warstwy ukrytej}) \tag{3.12}$$

$$Z^{(wyjcie)} = A^{(h)}W^{(wyjcie)}(\text{wejście dla warstwy wyjściowej}) \tag{3.13}$$

$$A^{(wyjcie)} = \phi(Z^{(wyjcie)})(\text{funkcja aktywacji warstwy wyjściowej}) \tag{3.14}$$

Powyższe wzory ilustrują sieć skierowaną przedstawioną na rysunku poniżej:



Rysunek 3.13:

W algorytmie wstecznej propagacji błędu błąd jest propagowany z prawej do lewej. Proces zaczyna się od obliczenia błędu w warstwie wyjściowej:

$$\delta^{(out)} = a^{(out)} - y \tag{3.15}$$

Gdzie y oznacza etykietę klasy prawdziwej. Następnie obliczany jest błąd warstwy ukrytej:

$$\delta^{(h)} = \delta^{(wyjcie)} \big(W^{(wyjcie)}\big)^T \frac{\partial \phi(z^{(h)}))}{\partial z^{(h)}} \tag{3.16}$$

po obliczeniach można zapisać to jako:

$$\delta^{(h)} = \delta^{(wyjcie)} \big(W^{(wyjcie)}\big)^T \big(a^{(h)}\big(1 - a^{(h)}\big)\big) \tag{3.17}$$

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.

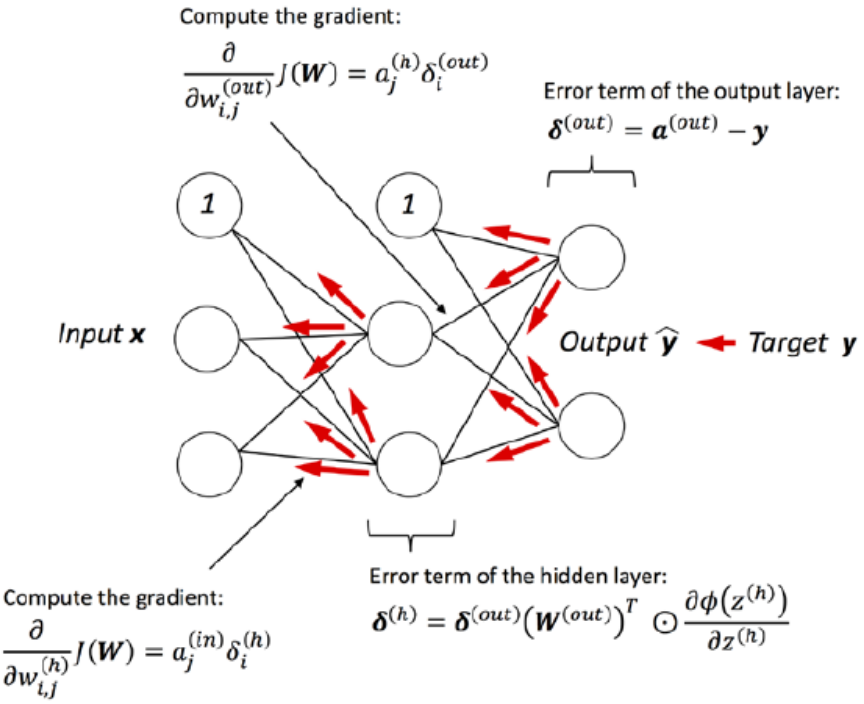
$$\Delta^{(h)} = \delta^{(h)} + \big(A^{(wejcie)}\big)^T \delta^{(h)} \tag{3.18}$$

$$\Delta^{(wyjcie)} = \delta^{(wyjcie)} + \big(A^{(h)}\big)^T \delta^{(wyjcie)} \tag{3.19}$$

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

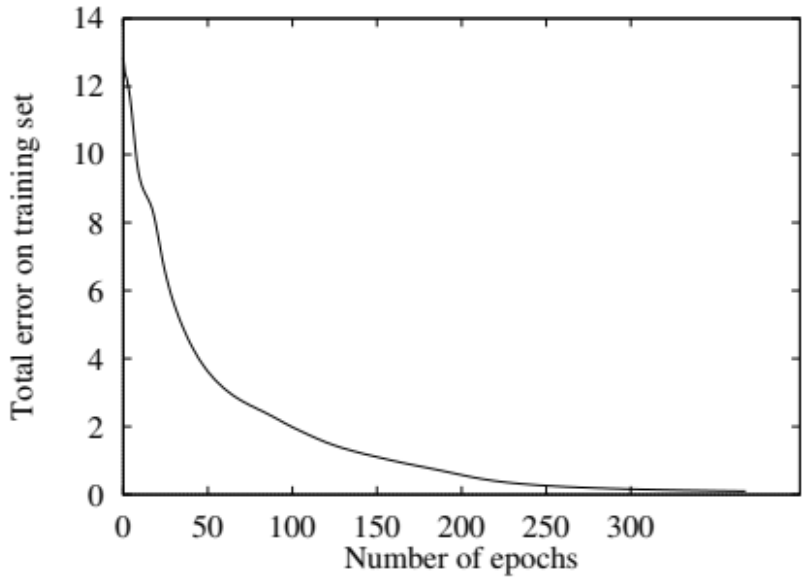
$$W^{(l)} := W^{(l)} - \eta \Delta^{(l)} \tag{3.20}$$

Cały proces można przedstawić za pomocą ilustracji:



Rysunek 3.14:

Fakt 2. Epoka to 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.

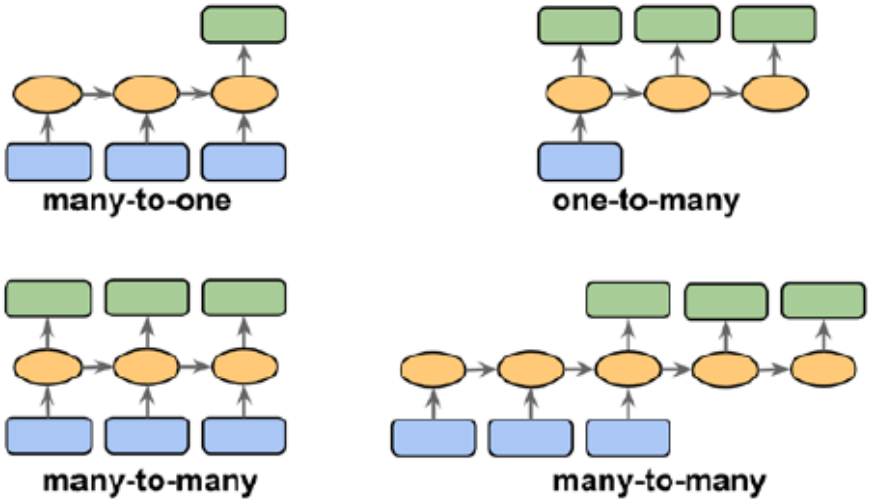


Rysunek 3.15:

3.8 Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe (ang. recurrent neural netorks, RNN)(Rumenlhart 1986) to rodzina sieci neuronowych służąca do pracy z danymi sekwencyjnymi. Sieci rekurencyjne mają zastosowanie między innymi 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 klika 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*[6] Rysunek poniżej obrazuje różne zależności między danymi wejściowymi, a wyjściowymi.

[6] http://karpathy.github.io/2015/05/21/rnn-effectiveness/



Rysunek 3.16:

W przypadku kiedy dane wejściowe zawierają sekwencje istnieje klika różnych przypadków modeli, które to zwrócą 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. emocja 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 sekwencje, a następnie generują jedno wyjście

Inne historyczne przykłady sieci rekurencyjnych to:

- 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:

$$h_t = f_W(h_{t-1}, x_t) \tag{3.21}$$

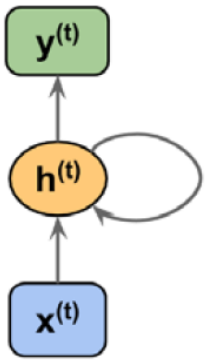
- h_t - nowy stan

- f_W - funkcja aktywacji z parametrem W

- h_{t-1} - poprzedni stan

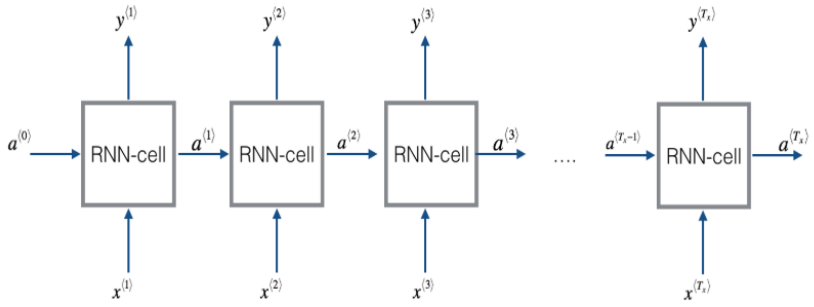
- x_t - wektor wejścia w kroku czasowym t

Powyższy wzór można przedstawić za pomocą schematu:



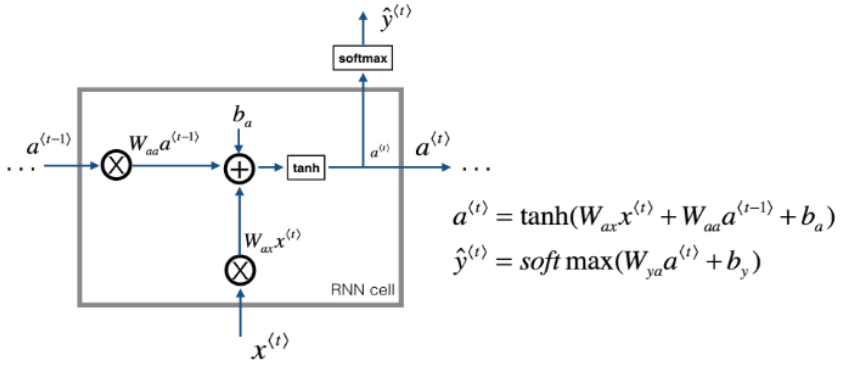
Rysunek 3.17: Schemat rekurencyjnej sieci neuronowej z jedną warstwą ukrytą

Powyższy model można rozwinąć w czasie, wtedy otrzymamy:



Rysunek 3.18: Sieć rozwinięta w czasie. Sekwencja wejściowa $x = (x^1, x^2, ..., x^{T_x})$ jest aplikowana do sieci w T_x krokach czasowych. Sieć zwraca sekwencję $y = (y^1, y^2, ..., y^{T_x})$

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. 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 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:

- $\boldsymbol{W}_{ax}$ - macierz wag pomnożona przez wektor wejściowy
- $\boldsymbol{W}_{aa}$ - macierz wag pomnożona przez stan ukryty
- $\boldsymbol{W}_{ya}$ - macierz wag pomiędzy warstwą ukrytą, a warstwą wyjściową

Rekurencyjne sieci neuronowe uczy się za pomocą algorytmu *Backpropagation Through Time: What It Does and How to D* 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 $t = T$.

$$L = \sum_{t=1}^{T} L^{(t)} \tag{3.22}$$

Sieci rekurencyjne stosuje się aby zachowywać długie zależności w czasie. Jednak niesie to ze sobą pewne problemy, ponieważ im więcej mamy 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*[7].

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

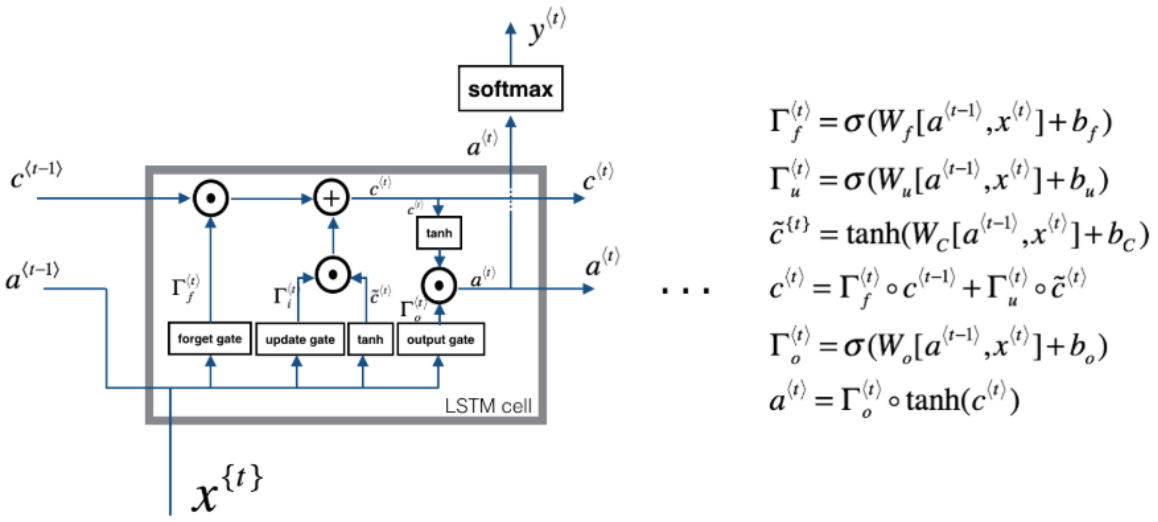
- Truncated backpropagation through time (TBPTT)

- Long short-term memory (LSTM)[8]

3.8.1 Long short-term memory

W obecnych czasach 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. 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 [9] 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.

[7] https://arxiv.org/pdf/1211.5063.pdf
[8] http://www.bioinf.jku.at/publications/older/2604.pdf
[9] 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.



$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\{t\}} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Rysunek 3.20: Komórka LSTM

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

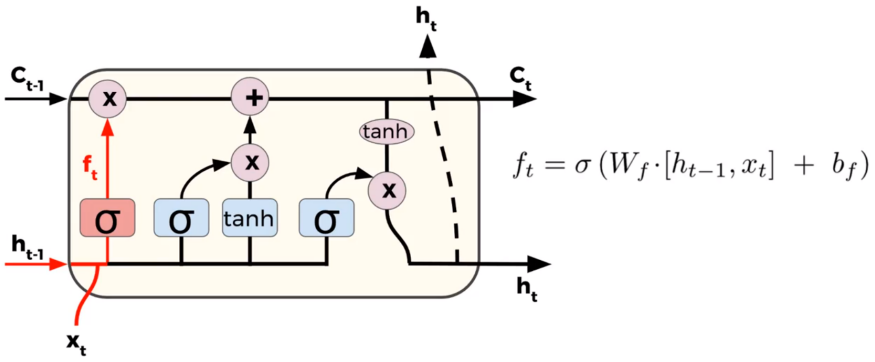
- Bramka zapomnienia (ang. forget gate) pozwala komórce pamięci na wymazanie pamięci. Wektory wejściowe po wymnożeniu przez wagi trafiają do sigmoidalnej funkcji aktywacji. Funkcja aktywacji zwraca wektor, którego elementy są z zakresu (0, 1). Kolejno wektor jest poddany iloczynowi Hadamarda ze wcześniejszym stanem komórki pamięci $c^{\langle t-1 \rangle}$. 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 $c^{\langle t-1 \rangle}$. W przeciwnym wypadku jeżeli wartości wektora z funkcji aktywacji będą bliskie zeru to elementy z wektora $c^{\langle t-1 \rangle}$ zostaną zachowane. Bramka zapomnienia nie była częścią oryginalnej komórki LSTM, została dodana kilka lat później[10].

- 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.

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

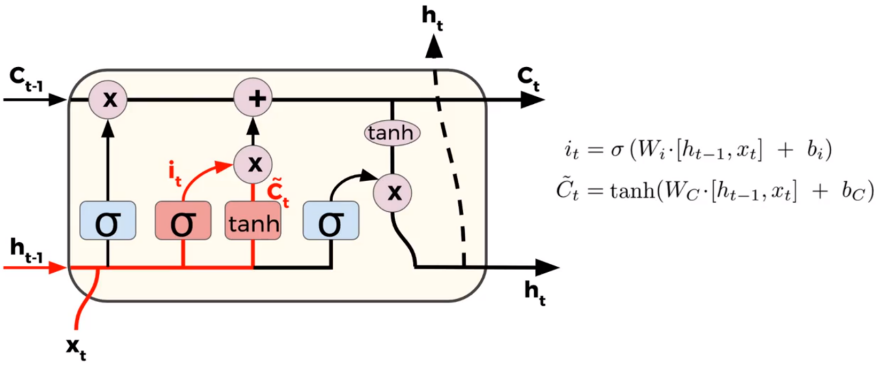
- Bramka wyjścia (ang. output gate) - ostatni element komórki LSTM, który składa się z dwóch komponentów: funkcje aktywacji tanh oraz wyjściową funkcję sigmoidalną.

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



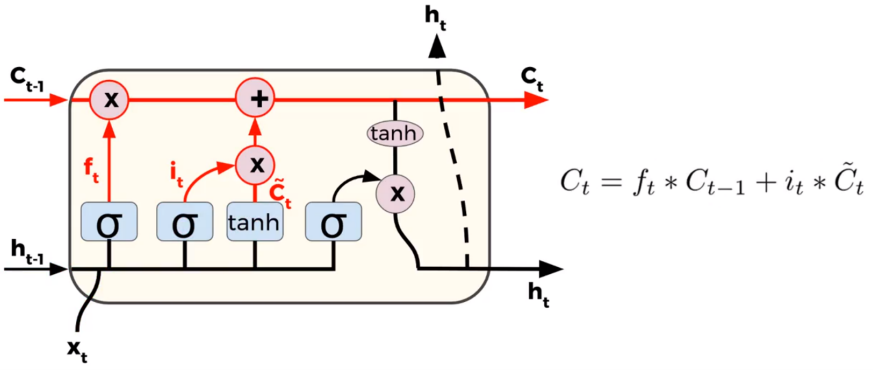
Rysunek 3.21: Pierwszy krok przepływu informacji przez komórkę LSTM

W poprzednim kroku bramka decydowała o tym jaką informacje należy zapomnieć. W kolejnym kroku następna bramka decyduje jaką informacje należy zachować. Bramka składa się z dwóch części: funkcji sigmoidalej oraz tanh. W wyniku otrzymywany jest wektor, który jest nazywany wektorem nowych wartości kandydujących



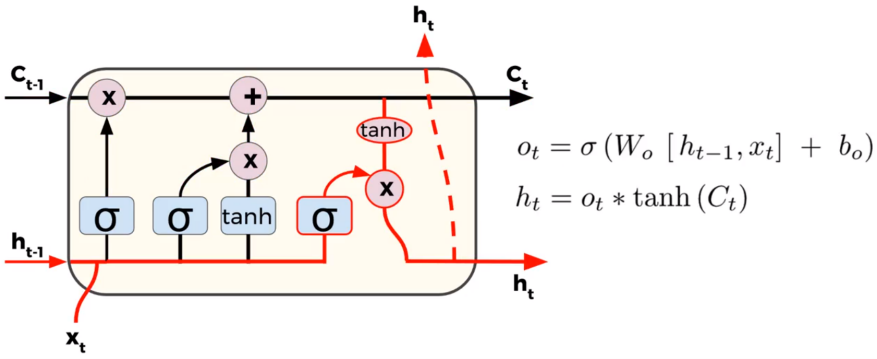
Rysunek 3.22: Drugi krok przepływu informacji przez komórkę LSTM

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

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

Badania wykazują, że sieci LSTM znacznie szybkiej przychodzi 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)[11]. 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 authorstwa Yao, et al

[11] http://arxiv.org/pdf/1406.1078v3.pdf

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

[12] http://proceedings.mlr.press/v37/jozefowicz15.pdf