Politechnika Wrocławska

Wydział Matematyki
Kierunek studiów: Matematyka stosowana
Specjalność: –

Praca dyplomowa – inżynierska

UCZENIE MASZYNOWE - MOŻLIWOŚCI I ZASTOSOWANIA SIECI NEURONOWYCH

Anna Szymanek

słowa kluczowe:
uczenie maszynowe, głębokie sieci neuronowe, sieci z długą pamięcią krótkotrwałą, generowanie muzyki

krótkie streszczenie:
Głównym celem pracy jest przeanalizowanie możliwości zastosowania głębokich sieci neuronowych do generowania utworów muzycznych. W pracy została przedstawiona teoria rekurencyjnych sieci neuronowych z długą pamięcią krótkotrwałą oraz metody ich trenowania. W języku programowania Python zaimplementowano dwie głębokie sieci neuronowe z różnym podejściem do zagadnienia generowania rytmu. Na podstawie stworzonych modeli wygenerowano sekwencje muzyczne oraz przeprowadzono ankietę walidującą rezultaty ich działania.

Opiekun pracy dyplomowej	dr inż. Radosław Michalski	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	ocena	podpis

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

a) kategorii A (akta wieczyste)

b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)

** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław, rok 2020

Wrocław University
of Science and Technology

Faculty of Pure and Applied Mathematics
Field of study: Applied Mathematics
Specialty: –

Engineering Thesis

MACHINE LEARNING - THE CAPABILITIES AND APPLICATIONS OF NEURAL NETWORKS

Anna Szymanek

keywords:
machine learning, deep neural network, long short-time memory networks, generative music

short summary:
The main purpose of the work is analysis the possibilities of using deep neural networks to generate music. The paper presents the theory of recurrent neural networks with long short-term memory and methods of training them. Two deep neural networks have been implemented in the programming language Python with different approaches to the issue of rhythm generation. Based on the created models music sequences were generated and a survey validating the results of their activities was carried out.

Supervisor	dr inż. Radosław Michalski	.....	.....
	Title/degree/name and surname	grade	signature

*For the purposes of archival thesis qualified to:**

a) category A (perpetual files)

b) category BE 50 (subject to expertise after 50 years)

** delete as appropriate*

stamp of the faculty

Wrocław, 2020

Spis treści

Wstęp

Sztuczna inteligencja udaje ludzki proces myślowy nie tylko w zadaniach pasywnych, takich jak rozpoznawanie obiektów na obrazach czy prowadzenie samochodu, ale również w zadaniach kreatywnych. To co kilka lat temu wydawało się niewyobrażalne, na naszych oczach staje się prawdą.

Latem 2015 r. firma Google zaprezentowała efekty pracy algorytmu DeepDream, który miał nas bawić psychodylicznym obrazem oczu psów i paranoidalnych artefaktów. W 2016 r. zaprezentowano aplikację Prisma, która zamieniała zdjęcia na obrazy stylizowane na dzieła wielkich mistrzów. Tegoż samego roku powstał film krótkometrażowy „Sunspring" na podstawie scenariusza napisanego w całości, łącznie z dialogami, przez algorytm LSTM [1].

Coraz częściej sieci neuronowe zaczęto stosować w celu komponowania muzyki.

Oczywiście dzieła tworzone przez sztuczną inteligencję nie dorównują tym stworzonym przez prawdziwych artystów - geniuszy. Sztuczna inteligencja jeszcze nie potrafi osiągnąć poziomu wybitnych kompozytorów, scenarzystów czy malarzy. Nigdy też jej celem nie było zastępowanie ludzi. Stosowanie sztucznej inteligencji ma na celu wniesienie do ludzkiego życia większej ilości inteligentnych rozwiązań. Często będzie ona stosowana tylko jako narzędzie wspierające kreatywne rozwiązania naszego mózgu.

Podczas tworzenia dzieła praca artysty w dużej mierze polega na prostym rozpoznawaniu wzorców i zastosowania umiejętności technicznych. Te etapy procesu twórczego często są uważane za mniej atrakcyjne. Właśnie tu jest miejsce do zastosowania sztucznej inteligencji - algorytmy uczenia maszynowego mogą nauczyć się tych statycznych struktur naszego języka czy sztuki [12].

Modele uczenia maszynowego są w stanie nauczyć się ukrytej przestrzeni utworów muzycznych, dzieł literackich czy obrazów, a następnie wygenerować z tych przestrzeni próbki o podobnym charakterze, które mogą zostać wykorzystane w celu stworzenia nowego dzieła sztuki. Oczywiście takie próbkowanie jest tylko zbiorem matematycznych operacji, nie jest samo w sobie niczym kreatywnym. Znaczenie temu, co zostało wygenerowane może nadać tylko człowiek, jego interpretacja bazująca na ludzkich przeżyciach, emocjach i doświadczeniach. W rękach artysty wygenerowane dane mogą zostać odpowiednio zmodyfikowane i nabrać piękna, mogą być narzędziem wspomagającym artystę, modyfikowac jego kreatywność i poszerzać wyobraźnię [1]. Jeden z pionierów muzyki elektronicznej i algorytmicznej, prawdzimy wizjoner, Iannis Xenakis, wyraźił podobną myśl:

"Kompozytor zwolniony z obowiązku wykonywania skrupulatnych obliczeń może poświęcić się pracy nad ogólnymi problemami nowej formy muzycznej i przyglądać się uważnie wszelkim szczegółom tej formy, modyfikując wartości danych wejściowych. Może np. sprawdzić wszystkie kombinacje instrumentalne: od solistów, przez małe orkiestry, do dużych orkiestr. Dzięki pomocy komputerów kompozytor zaczyna działać jak pilot: naciska przyciski, wprowadza współrzędne i nadzoruje trajektorie lotu kosmicznej kapsuły dźwięku przez muzyczne konstelacje i galaktyki, które do niedawna mógł sobie tylko wyobrazić [16]."

Rozdział 1

Motywacja

Za początek historii generowania danych sekwencyjnych można uznać opracowanie algorytmu LSTM (Long Short-Term Memory) w 1997 r. przez Seppa Hochreitera oraz Jürgena Schmidhubera.

W roku 2002 Douglas Eck, pracujący wtedy w laboratorium Schmidhubera w Szwajcarii, po raz pierwszy zastosował algorytm LSTM celem generowania muzyki i uzyskał obiecujące rezultaty.

Jednak skuteczne generowanie danych sekwencyjnych przy użyciu rekurencyjnych sieci neuronowych zyskało popularność w roku 2016, na początku takie sieci wykorzystywano do generowania tekstów znak po znaku [17].

Obecnie Douglas Eck pracuje w Google Brain, gdzie w 2016 r. założył grupę badawczą Magenta Studio „badającą rolę uczenia maszynowego jako narzędzia w procesie twórczym".

Zespół Google Brain stworzył pakiet do tworzenia muzyki typu „open-source" przy użyciu modeli uczenia maszynowego. Pakiet zawiera cztery narzędzia: Continue, Generate, Interpolate i GrooVAE. Muzycy mogą używać modeli na swoich plikach MIDI. Obecnie zestaw narzędzi jest dostępny jako samodzielna aplikacja Electron oraz jako wtyczka do oprogramowania sekwencera muzycznego i cyfrowej stacji roboczej Ableton Live, która początkowo została zaprojektowana jako instrument do występów na żywo, ale obecnie jest również szeroko stosowana do komponowania, nagrywania, aranżowania, miksowania i masteringu [12].

Inne istniejące na rynku rozwiązania:

- MuseNet (2019),
- Wavenet (2016),
- MuseGAN (2017).

Celem tej pracy jest sprawdzenie, czy możliwe jest efektywne generowanie sekwencji muzycznych z wykorzystaniem rekurencyjnych sieci neuronowych z właściwością długiej pamięci krótkotrwałej.

Wyżej wymienione rozwiązania funkcjonują na rynku i są wykorzystywane do generowania muzyki np. do gier komputerowych. Zamierzeniem tej pracy jest stworzenie algorytmu, który pozwoli na generowanie utworów muzycznych, które będą trudno rozróżnialne od tych napisanych przez prawdziwych kompozytorów.

Rozdział 2

Teoria

2.1 Dlaczego sieci rekurencyjne są tak wyjątkowe?

Pałkarz uderza piłkę. Zawodnik natychmiast zaczyna biec, przewidując trajektorię piłki. Śledzi ją, dostosowuje swoje ruchy i wreszcie łapie (po czym otrzymuje burzę oklasków).

Przewidywanie przyszłości jest tym, co robimy cały czas, bez względu na to, czy kończymy zdanie znajomego, czy spodziewamy się zapachu kawy na śniadanie.

Rekurencyjne sieci neuronowe (RNN) to klasa sieci, które potrafią przewidzieć przyszłość (oczywiście do pewnego stopnia). Mogą analizować szeregi czasowe takich danych jak na przykład ceny akcji i informować, kiedy coś kupić czy sprzedać. W autonomicznych systemach jazdy mogą przewidywać trajektorie samochodów i pomagają unikać wypadków. Mówiąc bardziej ogólnie, mogą pracować na sekwencjach o dowolnej długości, zamiast na wejściach o stałej wielkości, co odróżnia je od innych sieci. Na przykład jako dane wejściowe mogą wziąć zdania, całe dokumenty lub pliki audio, co czyni je niezwykle przydatnymi do przetwarzania języka naturalnego (NLP), w zastosowaniach takich jak: automatyczne tłumaczenie, synteza mowy na tekst lub analiza sentymentów (np. czytanie recenzji filmów i klasyfikacji opinii oceniającego na temat filmu) [2].

Co więcej, zdolność przewidywania rekurencyjnych sieci neuronowych sprawia, że są w stanie zaskoczyć kreatywnością. Możemy je zapytać o przewidzenie najbardziej prawdopodobnych kolejnych nut danej melodii, a następnie losowo wybrać jedną z tych nut i zagrać ją. Następnie poprosić sieć o kolejne najbardziej prawdopodobne nuty, zagrać i powtórzyć cały proces. Zanim się spostrzeżesz, twoja sieć RNN skomponuje melodię na wzór tej wyprodukowanej przez Google Magenta. Efekt może nie jest tak spektakularny, jak przy dziełach Bacha czy Mozarta, ale kto wie, co będzie za kilka lat?

W architekturach nierekurencyjnych sieci neuronowych, wejście o stałym rozmiarze zostaje przetworzone na wyjście o stałym rozmiarze. Nawet sieci konwolucyjne stosowane w rozpoznawaniu obrazów są spłaszczane do poziomu stałego wektora wyjściowego. Rekurencyjne sieci neuronowe eliminują to ograniczenie. Nazwa rekurencyjne sieci neuronowe, wzieła się stąd, że RNN mogą obsługiwać sekwencje o zmiennej długości dzięki zdefiniowaniu przez nie rekurencyjnej relacji [18].

Zdolność przetwarzania dowolnych sekwencji wejścia sprawia, że sieci RNN nadają się do takich zadań, jak modelowanie języków lub rozpoznawanie mowy. W rzeczywistości, sieci RNN teoretycznie mogą być stosowane do każdego problemu, ponieważ udowodniono, że są one kompletne w sensie Turinga [11]. To znaczy, że teoretycznie mogą symulować dowolny program, z którym nie jest w stanie sobie poradzić zwykły komputer [8].

2.2 Budowa Rekurencyjnej Sieci Neuronowej

2.2.1 Neuron rekurencyjny (z ang. *Recurrent Neurons*)

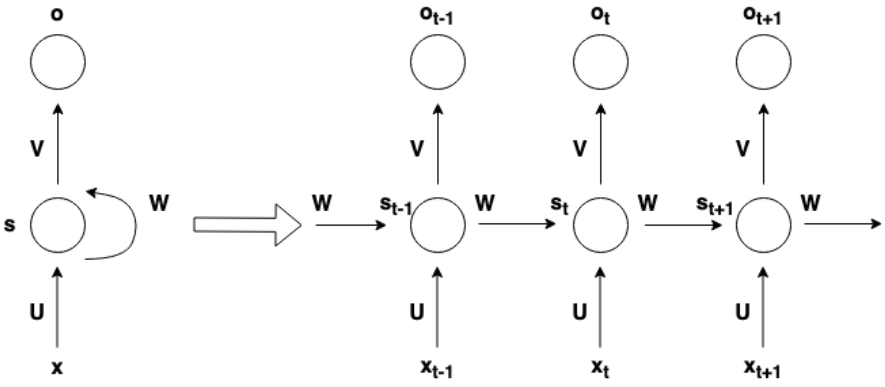
W nierekurencyjnych sieciach neuronowych, aktywacje przebiegają tylko w jednym kierunku, od warstwy wejściowej do warstwy wyjściowej (za wyjątkiem kilku sieci). Rekurencyjna sieć neuronowa wygląda bardzo podobnie z tym wyjątkiem, że ma również połączenia skierowane w drugą stronę. Rozważmy najprostszą możliwą RNN, złożoną z jednego neuronu, który odbiera dane wejściowe, generując dane wyjściowe i wysyła je z powrotem do siebie [18].

Sieć RNN zapiszmy w postaci relacji rekurencyjnej, która pokazuje jak zmienia się stan neuronu w kolejnych krokach. Wyraża się to przez funkcję:

$$S_t = f(S_{t-1}, X_t)$$

Gdzie: S_t — stan neuronu w kroku t, obliczany przez funkcję f na podstawie stanu $t-1$ poprzedniego kroku oraz bieżącego wejścia X_t, f - dowolna różniczkowalna funkcja (np. $tanh$), W - liniowa transformacja (jednego stanu w drugi), U - liniowa transformacja (wejścia w określony stan), o_t - wyjście generowane przez sieć.

Możemy przedstawić tę małą sieć na osi czasu (Rysunek 2.1), nazywamy to rozwijaniem sieci w czasie.



Rysunek 2.1: Po lewej: $S_t = S_{t-1}W + X_tU$ - ilustracja rekurencyjnej relacji w sieci RNN, z wyjściem danym jako: $o_t = VS_t$. Po prawej: rozwijanie sieci w czasie: $t-1, t, t+1$. Źródło: Opracowanie własne.

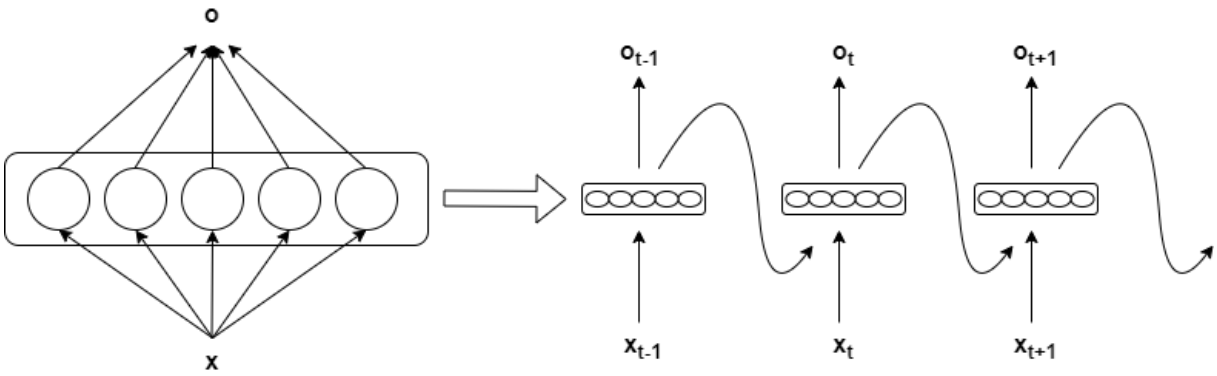
2.2.2 Warstwa (z ang. *Layer*)

Można łatwo stworzyć warstwę neuronów rekurencyjnych (Rysunek 2.2). W każdym kroku czasowym t, każdy neuron w warstwie otrzymuje dwa wektory, ponieważ teraz zarówno wejścia, jak i wyjścia są wektorami (kiedy był tylko jeden neuron, wyjście było skalarem).

Każdy neuron rekurencyjny posiada dwa zestawy wag: jeden dla sygnałów wejściowych x_t, a drugi dla sygnałów wyjściowych w poprzedniego kroku czasowego, o_{t-1}. Nazwijmy te wektory wag w_x i w_o [2]. Dane wyjściowe warstwy rekurencyjnej mogą być obliczone według wzoru:

$$o_t = \phi(x_t^T w_x + o_{t-1}^T w_o + b)$$

Gdzie: b to błąd systematyczny (z ang. *bias*), a ϕ to funkcja aktywacji.



Rysunek 2.2: Po lewej:warstwa rekurencyjna. Po prawej: rozwinięcie sieci w czasie. Źródło: Opracowanie własne.

Podobnie jak w przypadku innych sieci neuronowych, możemy obliczyć dane wyjściowe warstwy rekurencyjnej za jednym razem dla całego „mini-batcha" (pewnej partii danych) z wykorzystaniem zwektoryzowanej postaci poprzedniego równania:

$$O_t = \phi(X_t W_x + 0_{t-1} W_o + b) = \phi([X_t 0_{t-1}]W + b), \text{gdzie } W = \begin{bmatrix} W_x \\ W_o \end{bmatrix}$$

Oznaczenia:

O_t - macierz m x $n_{neuronów}$ zawierająca dane wyjściowe warstwy w kroku czasowym t dla każdego wystąpienia w mini-batchu (m jest liczbą wystąpień w mini-batchu i $n_{neuronów}$ to liczba neuronów).

X_t - macierz m x $n_{wejść}$ zawierającą dane wejściowe dla wszystkich neuronów ($n_{wejść}$ to liczba danych wejściowych cechy).

W_x - macierz $n_{wejść}$ x $n_{neuronów}$ zawierająca wagi połączeń danych wejściowych bieżącego kroku czasowego.

W_o - macierz $n_{wejść}$ x $n_{neuronów}$ zawierająca wagi z poprzedniego kroku czasowego.

b - wektor długości równej liczbie neuronów, zawierający błąd systematyczny każdego neuronu (z ang. *bias*).

Łatwo zauważyć, że O_t jest funkcją X_t oraz O_{t-1}, która jest funkcją X_{t-1} i $Ot - 2$, która jest funkcją X_{t-2} i O_{t-3}, i tak dalej.

To sprawia, że $Y_(t)$ jest funkcją wszystkich danych wejściowych od czasu $t = 0$ (to znaczy, że $X_0, X_1, \ldots, X_t$). W pierwszym kroku czasowym, $t = 0$, nie ma wcześniejszych wyników, więc zwykle przyjmuje się, że wszystkie są zerami.

2.2.3 Komórka pamięci (z ang. *Memory cell*)

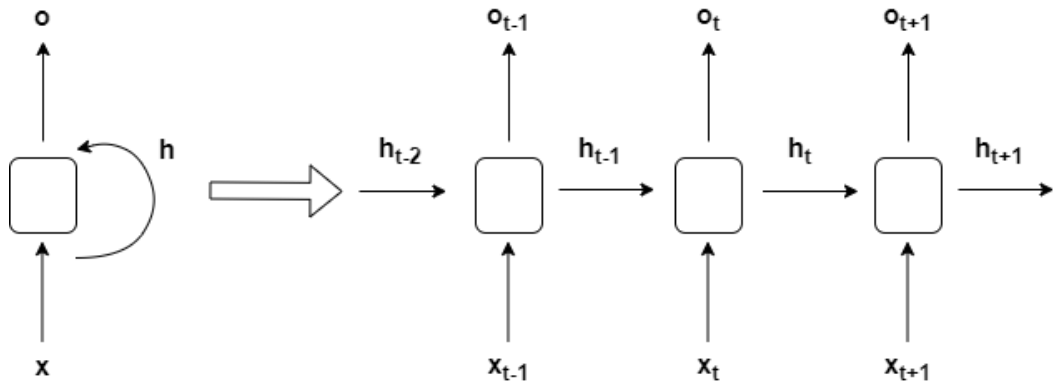
Ponieważ wyjście neuronu rekurencyjnego w kroku czasowym t jest funkcją wszystkich danych wejściowych z poprzednich kroków czasowych, można powiedzieć, że ma formę pamięci. Część sieci neuronowej, która zachowuje pewien stan podczas kolejnych kroków nazywamy komórką pamięci.

Pojedynczy neuron rekurencyjny lub warstwa rekurencyjna to bardzo podstawowe rodzaje komórek, ale w istnieją bardziej złożone i skomplikowane komórki.

Ogólnie stan komórki w kroku czasowym t, oznaczamy h_t (z ang. od słowa *hidden*), jest to funkcja niektórych danych wejściowych z tego kroku czasowego i stanu z poprzedniego kroku czasowego:

$$h_t = f(h_{t-1}, x_t).$$

Jego dane wyjściowe w kroku czasu t, oznaczamy o_t, jest to również funkcja poprzedniego stanu i bieżących sygnałów wejściowych. W przypadku podstawowych komórek, wynik jest po prostu równy stanowi, ale w bardziej złożonych komórkach nie zawsze jest tak, co pokazano na Rysunku 2.3 (stan ukryty i wyjście mogą być różne).



Rysunek 2.3: Po lewej: komórka ze stanem ukrytym h. Po prawej: rozwinięcie sieci w czasie. Źródło: Opracowanie własne.

2.3 Trenowanie sieci rekurencyjnych

Najczęstszymi metodami szkolenia rekurencyjnych sieci neuronowych są wsteczna propagacja w czasie (z ang. *Backpropagation Through Time (BPTT)*) i uczenie rekurencyjne w czasie rzeczywistym (z ang. *Real-Time Recurrent Learning (RTRL)*), podczas gdy BPTT jest podstawową metodą. Główną różnicą między BPTT i RTRL jest sposób aktualizacji wag. W oryginalnej formule LSTM zastosowano kombinację BPTT i RTRL. Dlatego w skrócie omówmy oba algorytmy uczenia się [9].

2.3.1 Wsteczna propagacja w czasie (z ang. *BPTT*)

Jak sugeruje nazwa, algorytm bazuje na algorytmie propagacji wstecznej [7]. Główna różnica polega na konieczności rozwijania sieci rekurencyjnej w czasie na przestrzeni określonej liczby kroków czasowych.

Algorytm propagacji wstecznej wykorzystuje spadek gradientu do nauki wag w warstwach sieci wielowarstwowych. Działa iteracyjnie, zaczynając od warstwy wyjściowej w kierunku warstwy wejściowej. Wymagane jest różniczkowalność funkcji aktywacji [14].

Zwykle wagi sieci neuronowej są inicjowane przez małe, znormalizowane liczby losowe przy użyciu wartości odchylenia. Następnie algorytm propagacji wstecznej wprowadza wszystkie próbki treningowe do sieci neuronowej i oblicza wejścia i wyjścia każdej komórki dla wszystkich warstw wyjściowych i ukrytych.

Zbiór wszystkich komórek sieci możemy podzielić na trzy rozłącze zbiory I, H, O, które są odpowiednio zestawami komórek wejściowych, ukrytych i wyjściowych. Oznaczamy

komórki wejściowe przez i, ukryte przez h oraz wyjściowe przez o. Zdefiniujemy również zbiór komórek nie-wejściowych, składający się z H i O. Do tego zbioru należy komórka u, wejście do u jest oznaczone przez x_u, jej stan przez s_u, błąd systematyczny przez b_u, a wyjście przez y_u. Biorąc pod uwagę komórki $u, v \in U$, waga, która łączy u i v, będzie oznaczona przez $W_{[u,v]}$. Aby modelować wejście zewnętrzne sieci neuronowej, zdefiniujemy zewnętrzny wektor wejściowy $x = [x_1, \ldots, x_n]$. Dla komórki $u \in U$, wyjście y_u, jest definiowane za pomocą sigmoidalnej funkcji aktywacji jako:

$$y_u = \frac{1}{1 + e^{-s_u}},$$

gdzie s_u - to stan u, definiowany jako: $s_u = z_u + b_u$, w którym z_u jest ważonym wejściem u.

Począwszy od warstwy wejścia, dane wejściowe są propagowane do przodu przez sieć, aż do warstwy wyjścia. Następnie, komórki wyjściowe wytwarzają obserwowalny wynik (wyjście sieciowe) y.

Następnie algorytm propagacji wstecznej propaguje błąd do tyłu, a wagi i odchylenia są aktualizowane w taki sposób, że redukujemy błąd w odniesieniu do obecnej próby uczącej [14]. Począwszy od warstwy wyjściowej, algorytm porównuje wydajność sieci y_o z odpowiednim pożądanym docelowym wynikiem d_o. Błąd e_o dla każdego neuronu wyjściowego jest obliczany jako:

$$e_o = d_o - y_o.$$

Możemy zdefiniować ogólne pojęcie błędu sieci:

$$E = \frac{1}{2} \sum_{o \in O} e_o^2.$$

Do zaaktualizowania wagi $W_{[u,v]}$ używamy:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E}{\partial W_{[u,v]}},$$

gdzie η to współczynnik uczenia.

Zdefiniujmy sygnał błędu dla komórki wyjścia o jako:

$$\vartheta_o = -\frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial s_o}.$$

Uwzgledniając stan wejść, dla komórek wyjścia możemy napisać:

$$\vartheta_o = (d_o - y_o) y_o (1 - y_o).$$

Stąd możemy aktualizowac wagi między ukrytą komórką ukrytą h a komórką wyjścia o, w następujący sposób:

$$\Delta W_{[h,o]} = \eta \vartheta_o y_h.$$

Pojęcie błedu jest powiązane z tym w jakim stopniu ukryta komórka h wpłyneła na wadliwą predykcję, możemy więc rozpropagować błąd na kolejne komórki, które korzystają z wysyłanych sygnałów.

Biorąc to pod uwagę możemy udowodnić uniwersalny wzór na aktualizację wag:

$$\Delta W_{[v,u]} = \eta \vartheta y_v.$$

Aktualizujemy wagi za każdym razem, aż wszystkie wyjścia sieciowe znajdą się w zadanym zakresie lub zostaną spełnione inne warunki zakończenia.

Aby skorzystać z wyżej opisanego algorytmu musimy rozwinąć sieć w czasie. Rozłożoną sieć można wytrenować przy użyciu algorytmu propagacji wstecznej [14]. Po zakończeniu sekwencji treningowej sieć jest rozwijana. Błąd jest obliczany dla jednostek wyjściowych z istniejącymi wartościami docelowymi za pomocą wybranej miary błędu. Następnie błąd jest podawany do tyłu i przeprowadza się aktualizację wag dla wszystkich kroków czasowych [13].

Obliczamy sygnał błędu dla komórki dla wszystkich kroków czasowych w jednym przebiegu, przy użyciu następującego iteracyjnego algorytmu propagacji wstecznej. Zmienną τ będziemy indeksowac kolejne kroki czasowe. Punkt początkowy oznaczymy jako t', a czas końcowy jako t. Przedział czasowy $[t', t]$ nazywamy epoką. Niech U będzie zbiorem jednostek nie-wejściowych i niech $\mathtt{f}u$ będzie różniczkowalną, nieliniową funkcja spłaszczającą. Wtedy wyjście y_u w kroku czasowym τ definiujemy jako:

$$y_u(\tau) = \mathtt{f}_u(z_u(\tau)),$$

gdzie z_u jest ważonym wejściem u.

Wagi w rekurencyjnych sieciach neuronowych są aktualizowane z wykorzystaniem sumy delt we wszystkich krokach czasowych.

Funkcją kosztu jest suma błędów $E_{total}(t', t)$ w całej epoce, którą chcemy zminimalizować za pomocą algorytmu uczenia. Taki całkowity błąd jest określony jako:

$$E_{total}(t', t) = \sum_{\tau=t'}^{t} E(\tau).$$

Aby uaktualnić wagi, używamy sygnału błędu $\vartheta_u(\tau)$ komórki u w kroku czasowym τ. Po wykonaniu obliczeń algorytmu propagacji wstecznej do czasu t', aktualizujemy wagę $\Delta W_{[u,v]}$. To odbywa się poprzez zsumowanie odpowiednich aktualizacji wag dla wszystkich kroków czasowych τ:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E_{total(t',t)}}{\partial W_{[u,v]}}.$$

2.3.2 Uczenie rekurencyjne w czasie rzeczywistym (z ang. *RTRL*)

Algorytm uczenia rekurencyjnego w czasie rzeczywistym (z ang. *Real-Time Recurrent Learning*) nie wymaga propagacji błędów [9]. Wszystkie informacje niezbędne do obliczenia gradientu są gromadzony w miarę wchodzenia strumienia danych wejściowych do sieci. Algorytm ma znaczny koszt obliczeniowy na cykl, a informacje przechowywane są nielokalnie; tzn. potrzebujemy dodatkowego pojęcia zwanego wrażliwością wyników. Niemniej jednak wymagana pamięć zależy tylko od wielkości sieci, a nie od wielkości wejścia. Podobnie jak w przypadku poprzedniego algorytmu będziemy używac komórek v, u oraz komórki nie-wejściowej k, a także przedziału czasowego $[t', t]$.

W odróżnieniu od BPTT, w RTRL zakładamy istnienie etykiety $d_k(\tau)$ dla każdego kroku czasowego τ każdej nie-wejściowej jednostki k, więc trenowanie sieci ma na celu zminimalizowanie ogólnego błędu sieci, który jest podawany w kroku czasu τ:

$$E(\tau) = \frac{1}{2} \sum_{k \in U} (d_k(\tau) - y_k(\tau))^2.$$

Tak jak poprzednio gradient całkowitego błędu jest również sumą gradientów wszystkich poprzednich kroków czasowych i bieżącego kroku czasowego.

W sieci kumuluje się wartości gradientu na każdym etapie. W ten sposób możemy śledzić zmiany wagi $\Delta W_{[u,v]}(\tau)$. Na koniec ogólna zmiana wagi dla $W_{[u,v]}$ jest następnie dana wzorem:

$$\Delta W_{[u,v]} = \sum_{\tau=t'+1}^{t} \Delta W_{[u,v]}(\tau),$$

gdzie $\Delta W_{[u,v]}(\tau)$ jest dana jako:

$$\Delta W_{[u,v]}(\tau) = -\eta \sum_{k \in U} (d_k(\tau) - y_k(\tau)) \frac{\partial y_k(\tau)}{\partial W_{[u,v]}}.$$

Błąd $e(\tau)$ jest zawsze znany. Drugi czynnik musimy zdefiniować jako zmienną, która mierzy wrażliwość wyjścia jednostki k w czasie τ na niewielką zmianę w wadze $W_{[u,v]}$, z należytym uwzględnieniem wpływu takiej zmiany na wage w całym przejściu sieci od czasu t' do t. Waga $W_{[u,v]}$ nie musi być podłączona z komórką k, co powoduje, że algorytm jest nielokalny (tzn. lokalne zmiany w jednym miejscu sieci mogą mieć wpływ na inne dowolne miejsce sieci). W RTRL informacja o gradiencie jest propagowana do przodu. Stąd wyjście y_k w kroku czasu $t+1$ jest dane jako:

$$y_k(t+1) = \mathtt{f}_k(z_k(t+1)),$$

gdzie z_k jest ważonym wejściem. Stąd wrażliwość wyjścia jest dana jako:

$$p^k_{uv}(t+1) = \mathtt{f'}_k(z_k(t+1)) \left[\delta u_k X_{[u,v]}(t+1) + \sum_{l \in U} W_{[k,l]} p^l_{uv}(t) \right],$$

gdzie δ_{uk} to delta Kroneckera, a wrażliwość w pierwszym kroku czasowym jest równa 0.

Algorytm RTRL jest algorytmem przyrostowym, dzięki czemu możemy użyć go do trenowania modelu, ponieważ otrzymujemy nowe dane wejściowe (w czasie rzeczywistym) i nie musimy już na nich dokonywać propagacja wstecznej w czasie. Znając wartość początkową możemy rekurencyjnie obliczyć wrażliwośći dla pierwszego i wszystkich kolejnych kroków czasowych [13].

Ostateczną zmianę wagi dla $W_{[u,v]}$ można obliczyć korzystając z tego, że:

$$p^k_{uv}(\tau) = \frac{\partial y_k(\tau)}{W_{[u,v]}}.$$

2.4 Sieci z długą pamięcią krótkotrwałą

Sieci pamięci krótkoterminowej - zwykle nazywane „LSTM”(z ang. *Long Short-Time Memory*) - są specjalnym rodzajem RNN, zdolnym do nauczenia się zależności długoterminowych.

LSTM zostały zaprojektowane w celu uniknięcia problemu długoterminowych zależności. Zapamiętywanie informacji przez długi czas jest praktycznie ich domyślnym zachowaniem. Rozwiązują one problem zanikającego gradientu [4]. LSTM może nauczyć się, jak pokonać minimalne opóźnienia przekraczające 1000 dyskretnych kroków czasowych. Rozwiązanie wykorzystuje karuzele stałego błędu (z ang. *(CEC)*, które wymuszają stały przepływ błędów

w specjalnych komórkach. Dostęp do komórek jest obsługiwany przez multiplikatywne bramki, które uczą się, kiedy przyznać dostęp [15].

Wszystkie RNN mają postać „łańcucha" powtarzających się komórek pamięci sieci neuronowej. W standardowych RNN ta komórka będzie miała bardzo prostą strukturę. W sieciach LSTM zamiast pojedyńczej wartwy, w środku każdej komórki znajdują się cztery specjalne warstwy współpracujące ze sobą.

2.4.1 Notacja

Aby dokładniej opisać działanie sieci LSTM, będziemy używać następującej notacji:

- η - szybkość uczenia się (z ang. *learning rate*);
- τ - jednostka czasu, początek epoki oznaczamy jako t', a koniec jako t;
- N - zbiór komórek pamięci sieci (z ang. *units*), w ogólności $y, v, l, k \in N$;
- I - zestaw komórek wejść, komórka wejścia $i \in I$;
- O - zestaw komórek wyjść, komórka wyjścia $o \in O$;
- U - zestaw komórek nie-wejściowych, komórka $y \in U$;
- y_u - wyjście komórki u, zwane również aktywacją, (jest to wartość, a nie macierz);
- $\texttt{Pre}(u)$ - zestaw komórek u wraz z połączeniami do tej komórki (czyli jej poprzednikami);
- $W_{[u,v]}$ - waga łącząca komórkę u z komórką v;
- $X_{[u,v]}$ - wejście komórki u pochodzące z komórki v;
- z_u - ważone wejście modułu u;
- b_u - błąd systematyczny/przesunięcie (z ang.*bias*) komórki u;
- s_u - stan komórki u;
- $\texttt{f}_u$ - funkcja spłaszczająca komórki u;
- e_u - błąd komórki u;
- ϑ_u - sygnał błędu komórki u;
- p^k_{uv} - czułość wyjściowa modułu k w odniesieniu do wagi $W_{[u,v]}$.

2.4.2 Karuzela stałych błędów (z ang. *CEC*)

Załóżmy, że mamy tylko jedną komórkę u z jednym połączeniem ze sobą. Lokalny błąd przepływu wstecznego u w jednym kroku czasowym τ wynika z równania jest dany przez:

$$\vartheta_u(\tau) = f'_u(z_u(\tau))W_{[u,u]}\vartheta_u(\tau+1).$$

W celu zapewnienia stałego przepływu błędów musi zachodzić:

$$f'_u(z_u(\tau))W_{[u,u]} = 1.0$$

i po scałkowaniu otrzymujemy:

$$f_u(z_u(\tau)) = \frac{z_u(\tau)}{W_{[u,u]}}.$$

Z tego dowiadujemy się, że f_u musi być liniowe, a aktywacja u musi pozostać stała w czasie; np.:

$$y_u(\tau+1) = f_u(z_u(\tau+1)) = f_u(y_u(\tau)W_{[u,u]}) = y_u(\tau).$$

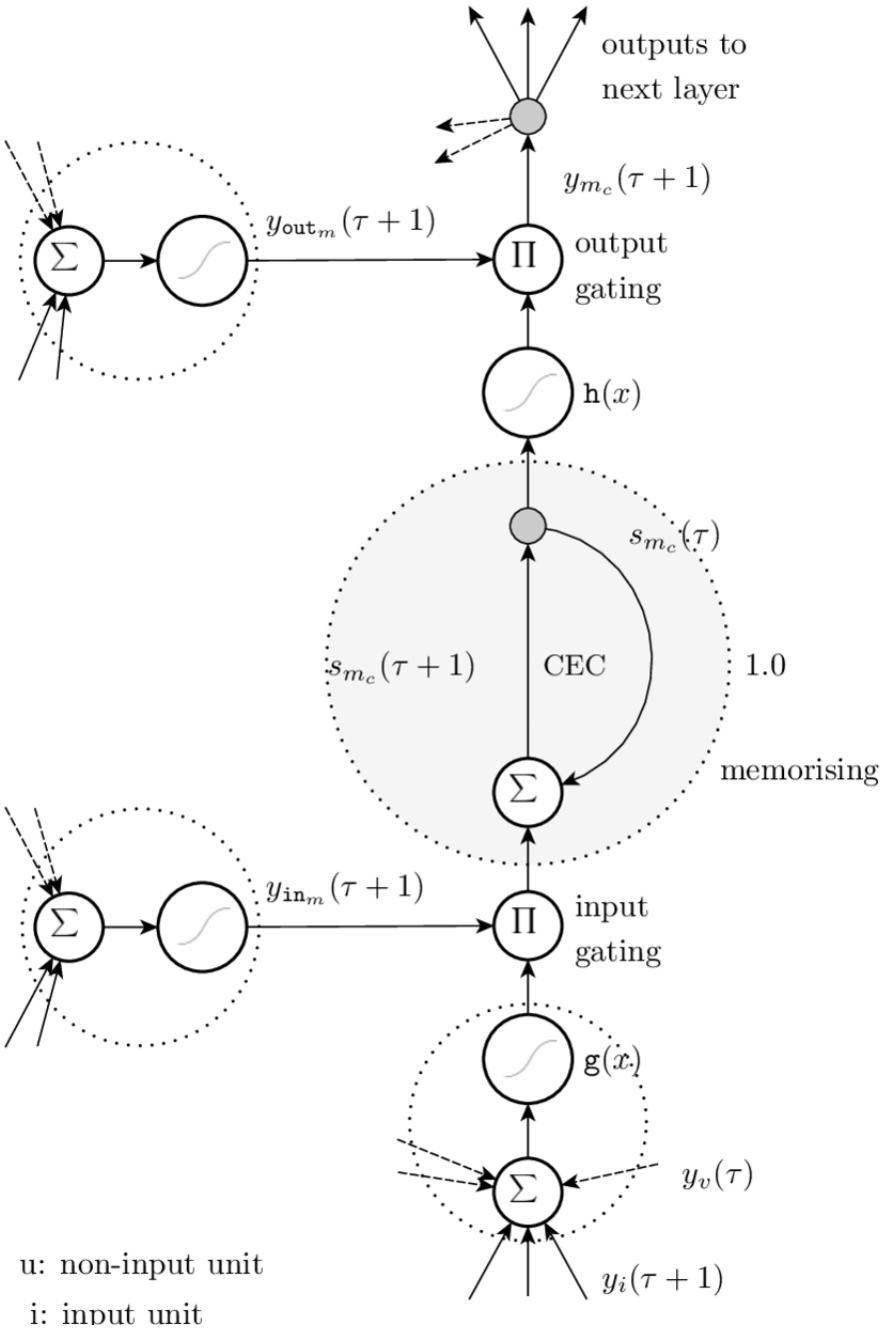
Jest to zapewnione przez użycie funkcji tożsamości $f_u = id$ i poprzez ustawienie $W_{[u,u]} = 1.0$. To zachowanie błędu nazywa się karuzelą stałych błędów (CEC) i jest to centralna cecha LSTM, w której osiąga się krótkotrwałą pamięć przez dłuższy czas. Oczywiście nadal musimy obsługiwać połączenia od innych modułów do modułu u.

2.4.3 Bloki pamięci

Wobec braku nowych danych wejściowych do komórki wiemy teraz, że przepływ zwrotny CEC pozostaje stały. Jednak jako część sieci neuronowej CEC jest nie tylko połączony ze sobą, ale także z innymi komórkami pamięci w sieci neuronowej. Potrzebujemy uwzględnić te dodatkowe ważone wejścia i wyjścia. Przychodzące połączenia z neuronem u mogą mieć sprzeczne sygnały aktualizacji wagi, ponieważ ta sama waga służy do przechowywania i ignorowania danych wejściowych. Dla ważonej wydajności połączeń z neuronu u, te same wagi mogą być użyte do odzyskania zawartości neuronu u oraz uniemożliwienia przepływu wyjściowego do innych neuronów w sieci.

Aby rozwiązać problem sprzecznych aktualizacji wag, LSTM rozszerza CEC z bramkami wejściowymi i wyjściowymi podłączonymi do warstwy wejściowej sieci i do innych komórek pamięci. W rezultacie powstaje bardziej złożona jednostka LSTM, zwana blokiem pamięci; jego standardowa architektura jest pokazana na Rysunku 2.4.

Bramki wejściowe, które są prostymi sigmoidalnymi jednostkami progowymi z funkcją aktywacji z wartościami na przedziale $[0, 1]$, kontrolują sygnały przychodzące z sieci do komórki pamięci poprzez odpowiednie skalowanie; gdy bramka jest zamknięta, aktywacja jest bliska zeru. Ponadto mogą nauczyć się, jak chronić przechowywaną zawartość przed zakłóceniami przez nieistotne sygnały. Aktywacja CEC przez bramkę wejściową jest zdefiniowana jako stan komórki. Bramki wyjściowe mogą nauczyć się kontrolować dostęp do zawartości komórki pamięci, która chroni inne komórki pamięci przed zakłóceniami pochodzący z u. Widzimy więc, że jest to podstawowa funkcja bramki multiplikatywnej. Komórki mają zezwalać lub zabraniać dostępu do stałego przepływu błędów przez CEC.



Rysunek 2.4: Standardowy blok pamięci sieci LSTM. Źródło: [13]

2.4.4 Trenowanie sieci LSTM

Aby zachować CEC w komórkach bloków pamięci LSTM, w oryginalnym sformułowaniu LSTM zastosowano kombinację dwóch algorytmów uczenia: BPTT do treningu komponentów sieciowych zlokalizowanych za komórkami oraz RTRL do trenowania komponentów sieciowych zlokalizowanych przed komórkami i w nich. Te ostatnie jednostki działają z RTRL, ponieważ istnieją pewne częściowe pochodne (związane ze stanem komórki), które muszą być obliczane na każdym etapie, bez względu na to, czy została podana wartość docelowa, czy nie została ona podana w tym kroku. Na razie pozwalamy jedynie na propagację gradientu komórki w czasie, obcinając resztę gradientu dla innych powtarzających się połączeń. Definiujemy kroki dyskretne w postaci $\tau = 1, 2, 3, \ldots$. W każdym kroku

mamy przekazanie do przodu i przekazanie do tyłu; w przekazaniu do przodu, obliczane są zawsze wyjścia / aktywacje wszystkich jednostek, podczas gdy w przekazaniu do tyłu obliczane są sygnały błędów dla wszystkich wag.

Przekazanie do przodu

Niech M będzie zbiorem bloków pamięci. Niech m_c będzie c-tą komórką pamięci w bloku pamięci m, a $W[u,v]$ będzie wagą łączącą komórkę u z komórką v. W oryginalnym sformułowaniu LSTM każdy blok pamięci m jest powiązany jedną bramką wejściową $\mathtt{in}_m$ i jedną bramką wyjściową $\mathtt{out}_m$. Wewnętrzny stan pamięci komórki m_c w czasie $\tau+1$ jest aktualizowany zgodnie z jej stanem $s_{m_c}(\tau)$ i zgodnie z wagą wejścia $z_{m_c}(\tau+1)$ pomnożoną przez aktywację bramki wejściowej $y_{\mathtt{in}_m}(\tau+1)$. Następnie używamy aktywacji bramki wejściowej $z_{\mathtt{out}_m}(\tau+1)$ do obliczenia aktywacji komórki $y_{m_c}(\tau+1)$.

Aktywacja $y_{\mathtt{in}_m}$ wejściowej bramki $\mathtt{in}_m$ jest obliczana jako:

$$y_{\mathtt{in}_m}(\tau+1) = f_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1)),$$

z wejściem bramki wejściowej:

$$z_{\mathtt{in}_m}(\tau+1) = \sum_u W_{[\mathtt{in}_m,u]} X_{[u,\mathtt{in}_m]}(\tau+1), \text{gdzie } u \in \mathtt{Pre}(\mathtt{in}_m)$$

$$= \sum_{v\in U} W_{[\mathtt{in}_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\mathtt{in}_m,i]} y_v(\tau+1).$$

Aktywacja bramki wyjściowej $\mathtt{out}_m$:

$$y_{\mathtt{out}_m}(\tau+1) = f_{\mathtt{out}_m}(z_{\mathtt{out}_m}),$$

z wyjściem bramki wejścia:

$$z_{\mathtt{out}_m}(\tau+1) = \sum_u W_{[\mathtt{out}_m,u]} X_{[u,\mathtt{out}_m]}(\tau+1), \text{gdzie } u \in \mathtt{Pre}(\mathtt{out}_m)$$

$$= \sum_{v\in U} W_{[\mathtt{out}_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\mathtt{out}_m,i]} y_i(\tau+1).$$

Wyniki bramek sa skalowane przy pomocy nieliniowej funkcji spłaszczenia $f_{\mathtt{in}_m} = f_{\mathtt{out}_m} = f$ zdefiniowanej jako:

$$f(s) = \frac{1}{1+\exp^{-s}},$$

tak aby wartości znajdowały się w zakresie $[0,1]$. Zatem dane wejściowe dla komórki pamięci będą w stanie przejść, tylko jeśli sygnał na bramce wejściowej jest wystarczająco blisko „1".

Dla komórki pamięci m_c w bloku pamięci m waga wejścia $z_{m_c}(\tau+1)$ jest zdefiniowana jako:

$$z_{m_c}(\tau+1) = \sum_u W_{[m_c,u]} X_{[u,m_c]}(\tau+1), \text{gdzie } u \in \mathtt{Pre}(m_c)$$

$$= \sum_{v\in U} W_{[m_c,v]} y_v(\tau) + \sum_{i\in I} W_{[m_c,i]} y_i(\tau+1).$$

Przypomnijmy, że stan wewnętrzny $s_{m_c}(\tau+1)$ jednostki w komórce pamięci w czasie $(\tau+1)$ jest obliczany inaczej; ważone dane wejściowe są spłaszczane i mnożone przez aktywację bramki wejściowej, a następnie dodaje się stan ostatniego kroku $s_{m_c}(\tau)$.

Jest to opisane równaniem:

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) + y_{\mathtt{in}_m}(\tau=1)\mathtt{g}(z_{m_c}(\tau+1)),$$

gdzie $s_{m_c}(0) = 0$ i nieliniowa funkcja spłaszczenia dla wejścia komórki:

$$\mathtt{g}(z) = \frac{4}{1+\exp^{-z}} - 2,$$

co w tym przypadku skaluje wynik do przedziału $[-2, 2]$.

Wyjście y_{m_c} jest teraz obliczane przez tłumienie i mnożenie stanu komórki s_{m_c} przez aktywację bramki wyjścia $y_{\mathtt{out}_m}$:

$$y_{m_c}(\tau+1) = y_{\mathtt{out}_m}(\tau+1)\mathtt{h}(s_{m_c}(\tau+1)).$$

Z nieliniową funkcją spłaszczenia:

$$\mathtt{h}(z) = \frac{1}{1+\exp^{-z}} - 1$$

z wartościami w przedziale $[-1, 1]$

Zakładając warstwową, rekurencyjną sieć neuronową ze standardowym wejściem, standardowym wyjściem i ukrytą warstwą składająca się z bloków pamięci, aktywacja komórki wyjścia o jest obliczana jako:

$$y_o(\tau+1) = \mathtt{f}_o((z_o(\tau+1)),$$

gdzie:

$$z_o(\tau+1) = \sum_{u \in U-G} W_{[o,u]} y_u(\tau+1),$$

gdzie G jest zbiorem jednostek bramek, wtedy możemy ponownie użyć logistycznej funkcji sigmoidalnej jako funkcji spłaszczającej $\mathtt{f}_o$.

Bramka zapominająca

Samo-złączenie w standardowej sieci LSTM ma stałą wagę ustawioną na „1", aby zachować stały stan komórki wraz z upływem czasu. Niestety stany komórek mają tendencję do rośnięcia liniowo podczas postępu szeregu czasowego wyrażonego przez ciągły strumień wejściowy. Główny negatywny efekt polega na tym, że cała komórka pamięci traci swoją zdolność zapamiętywania i zaczyna działać jak zwykły neuron rekurencyjnej sieci neuronowej.

Przez zastosowanie ręcznego zresetowania stanu komórki na początku każdej sekwencji,wzrost stanu komórki może być ograniczony, ale nie jest to praktyczne w przypadku wprowadzania ciągłego strumienia danych,gdzie nie ma wyraźnego końca, lub podział jest bardzo złożony i wykazuje dużą podatność na błędy. Aby rozwiązać ten problem zasugerowano, że mogłaby to być adaptacyjna bramka zapominająca przywiązana do samo-złączenia. Bramka zapominająca można nauczyć się resetować wewnętrzny stan komórki pamięci, gdy przechowywane informacje nie są już potrzebne. W tym celu zastępujemy wagę „1.0" połączeniem z CEC do multiplikatywnej, zapominającej bramki, której aktywacja y_φ, która jest obliczana przy użyciu metody podobnej jak dla innych bramek:

$$y_{\varphi_m}(\tau+1) = f_{\varphi_m}(z_{\varphi_m}(\tau+1) + b_{\varphi_m}),$$

gdzie f jest funkcją spłaszczania na przedziale $[0,1]$, b_{φ_m} jest przesunięciem bramki zapominającej oraz:

$$z_{\varphi_m}(\tau+1) = \sum_u W_{[\varphi_m,u]} X_{[u,\varphi_m]}(\tau+1), \text{gdzie } u \in \mathtt{Pre}(\varphi_m)$$

$$= \sum_{u\in V} W_{[\varphi_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\varphi_m,i]} y_i(\tau+1).$$

Początkowo b_{φ_m} jest ustawiony na wartość 0, jednak, ustawiamy b_{φ_m} na 1, aby poprawić wydajność LSTM. Zaktualizowane równanie do obliczania wewnętrznego stanu komórki s_{m_c} jest wyrażone wzorem:

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) y_{\varphi_m}(\tau+1) + y_{\mathtt{in}_m}(\tau+1)\mathtt{g}(z_{m_c}(\tau+1)),$$

z warunkiem początkowym $s_{m_c}(0) = 0$ oraz funkcją spłaszczającą $\mathtt{g}$, z wartosciami na przedziale $[-2,2]$.

Wagi początkowe bramek wejściowych i wyjściowych są inicjowane przez ujemnie wartości, a wagi bramki zapominającej są inicjalizowane wartościami dodatnimi. Z tego wynika, że na początku trenowania sieci funkcja aktywacji bramki zapominającej będzie zbliżona do „1.0". Komórka pamięci będzie zachowywać się jak standardowa komórka pamięci sieci bez bramki zapominającej. Zapobiega to zapominaniu komórki pamięci LSTM, zanim faktycznie sieć się czegoś nauczy.

Przekazanie do tyłu

LSTM zawiera elementy zarówno z BPTT, jak i RTRL. W ten sposób rozdzielamy komórki na dwa typy: komórki, których zmiany wagi oblicza się za pomocą BPTT (tj. komórki wyjściowe, komórki ukryte i bramki wyjściowe), oraz te, których zmiany wagi oblicza się przy użyciu RTRL (tj. bramki wejściowe, bramki zapomnienia i bloki) [13]. Zgodnie z notacją stosowaną w poprzednich sekcjach, całkowity błąd sieci w kroku czasu τ wynosi:

$$E(\tau) = \frac{1}{2}\sum_{o\in O}(d_o(\tau) - y_o(\tau))^2.$$

Rozważmy najpierw komórki współpracujące z BPTT. Definiujemy pojęcie indywidualny błąd komórki u w czasie τ:

$$\vartheta_u(\tau) = -\frac{\partial E(\tau)}{\partial z_u(\tau)},$$

gdzie z_u jest wagą wejścia komórki. Możemy rozwinąć pojęcie wagi w następujący sposób:

$$\Delta W_{[u,v]}(\tau) = -\eta\frac{\partial E(\tau)}{\partial W_{[u,v]}} = -\eta\frac{\partial E(\tau)}{\partial z_u(\tau)}\frac{\partial z_u(\tau)}{\partial W_{[u,v]}}.$$

Współczynnik $\frac{\partial z_u(\tau)}{\partial W_{[u,v]}}$ odpowiada sygnałowi wejściowemu, który przechodzi z komórki v do komórki u. Jednak w zależności od natury u indywidualny błąd jest różny. Jeśli u jest równe komórce wyjściowej o, to:

$$\vartheta_o(\tau) = \mathtt{f}'_o(z_o(\tau))(d_o(\tau)y_o(\tau)),$$

tak więc udział wagowy komórek wyjściowych wynosi:

$$\Delta W_{[o,v]} = \eta \vartheta_o(\tau) X_{[v,o]}(\tau).$$

Teraz, jeśli u jest równe ukrytej komórce h, mamy:

$$\vartheta_h(\tau) = \mathtt{f}'_h(z_o(\tau))(\sum_{o \in O} W_{[o,h]} \vartheta_o(\tau)),$$

gdzie O jest zbiorem komórek wyjściowych i udziału wag komórek ukrytych jest:

$$\Delta W_{[h,v]} = \eta \vartheta_h(\tau) X_{[v,h]}(\tau).$$

Wreszcie, jeśli u jest równe wyjściowej bramce $\mathtt{out}_m$ bloku pamięci m, to:

$$\vartheta_{\mathtt{out}_m}(\tau) \stackrel{tr}{=} \mathtt{f}'_{\mathtt{out}_m}(z_{\mathtt{out}_m}(\tau)) \left(\sum_{m_c \in m} \mathtt{h}(s_{m_c}(\tau)) \sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right);$$

gdzie $\stackrel{tr}{=}$ oznacza, że równość zachowuje się tak tylko wtedy, gdy błąd jest obcinany,oznacza to, że błąd nie rozprzestrzenia się ponownie do modułu za pośrednictwem własnego połączenia do tyłu [13].

Udział wagowy dla bramki wyjściowe to:

$$\Delta W_{[\mathtt{out}_m,v]}(\tau) = \eta \vartheta_{\mathtt{out}_m}(\tau) X_{[v,\mathtt{out}_m]}(\tau).$$

Rozważmy teraz komórki współpracujące z RTRL. W tym przypadku komórka błędy bramki wejściowej i bramki zapomnienia obracają się wokół pojedynczego błędu komórek w bloku pamięci. Definiujemy indywidualny błąd komórki m_c bloku pamięci m jako:

$$\vartheta_{m_c}(\tau) \stackrel{tr}{=} -\frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1)$$

$$\stackrel{tr}{=} \frac{\partial y_{m_c}(\tau)}{\partial s_{m_c}(\tau)} \left(\sum_{o \in O} \frac{\partial z_o(\tau)}{\partial y_{m_c}(\tau)} \left(-\frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \right) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1)$$

$$\stackrel{tr}{=} y_{out_m}(\tau) \mathtt{h}'(s_{m_c}(\tau)) \left(\sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1).$$

Zauważmy, że to równanie nie uwzględnia rekurenyjnego połączenia między komórką, a innymi komórkami, uwzględnia tylko swój błąd wynikający z połączenia cyklicznego z samą sobą (uwzględniające wpływ bramki zapominającej). Używamy częściowych pochodnych w celu zwiększenia udziału wagowego w komórce następująco:

$$\Delta W_{[m_c,v]}(\tau) = -\eta \frac{\partial E(\tau)}{\partial W_{[m_c,v]}} = -\eta \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} = \eta \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}},$$

oraz udziału wagowego dla bramek zapomnienia i bramek wejściowych, jak następuje:

$$\Delta W_{[u,v]}(\tau) = -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}} = -\eta \sum_{m_c \in m} \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}} = \eta \sum_{m_c \in m} \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}}.$$

Teraz musimy zdefiniować, jaka jest wartość $\frac{s_{m_c}(\tau+1)}{W_{[u,v]}}$. Zgodnie z oczekiwaniami również ona zależy od charakteru komórki. Jeśli u jest równe komórce m_c, to:

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[m_c,v]}} \overset{tr}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} y_{\varphi_m}(\tau+1) + g'(z_{m_c}(\tau+1)) f_{in_m}(z_{in_m}(\tau+1)) y_v(\tau).$$

Jeśli u jest równe bramce wejściowej in_m, to:

$$\frac{\partial s_{in_m}(\tau+1)}{\partial W_{[in_m,v]}} \overset{tr}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[in_m,v]}} y_{\varphi_m}(\tau+1) + g'(z_{m_c}(\tau+1)) f'_{in_m}(z_{in_m}(\tau+1)) y_v(\tau).$$

Wreszcie, jeśli u jest równe bramce zapomnienia φ_m, to:

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[\varphi_m,v]}} \overset{tr}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\varphi_m,v]}} y_{\varphi_m}(\tau+1) + s_{m_c}(\tau) f'_{\varphi_m}(z_{\varphi_m}(\tau+1)) y_v(\tau),$$

gdzie $s_{m_c}(0) = 0$.

Rozdział 3

Stworzone modele

Modele bazują na sieciach LSTM i wykorzystują wszystkie opisane we wcześniejszych rozdziałach właściwości do komponowania muzyki fortepianowej. Długa pamięć krótkotrwała gwarantuje nam utrzymanie stałej tonacji, rytmu oraz nastroju utworu, co upodabnia generowane sekwencje do melodii stworzonych przez człowieka.

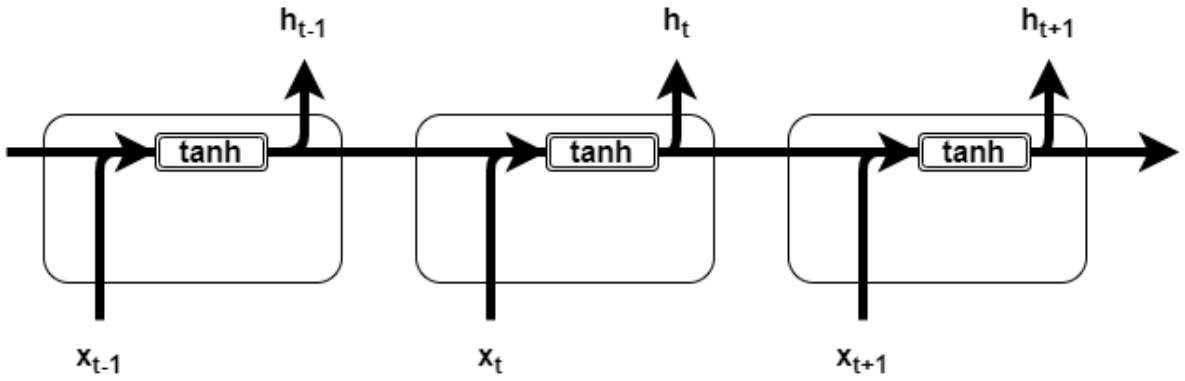
Modele zostały zaimpementowane w języku Python, przy wykorzystaniu następujących bibliotek: MIT's music21, TensorFlow, PyTorch, NumPy oraz FastAI.

Modele zostały przeszkolone na zbiorach danych zawierających próbki MIDI z archiwów zawierających utwory muzyki klasycznej, chociaż w modelach nie ma nic specyficznego dla fortepianowej muzyki klasycznej i inne pliki MIDI (np. sakosofonowe, jazzowe) mogą zostać użyte do trenowania obu modeli.

Podstawowym czynnikiem wpływającym na jakość generowanej muzyki jest metoda kodowania plików MIDI w formacie tekstowym.

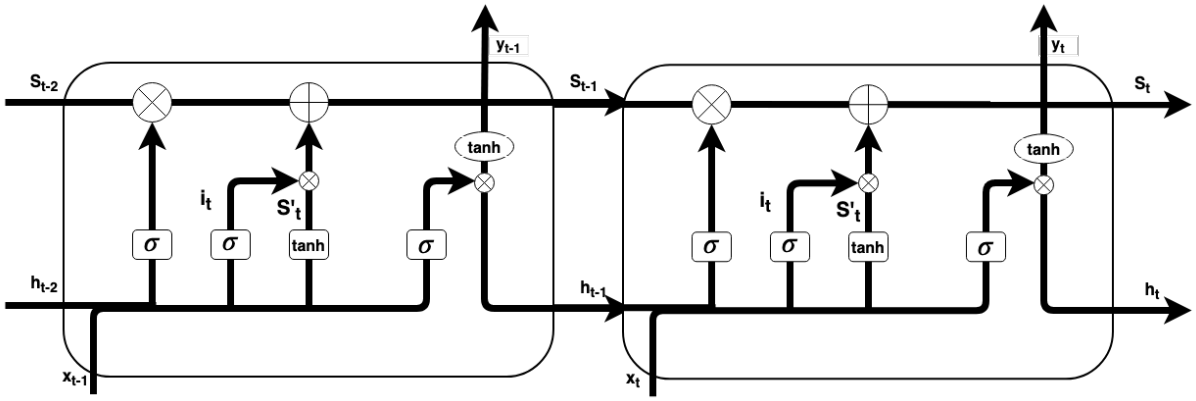
3.1 Intuicyjne zrozumienie działania sieci LSTM

Wszystkie nawracające sieci neuronowe mają postać łańcucha powtarzających się komórek sieci neuronowej [6]. W standardowych RNN ta powtarzalna komórka będzie miała bardzo prostą strukturę, na przykład będzie pojedynczą warstwą *tanh* jak na Rysunku 3.1.



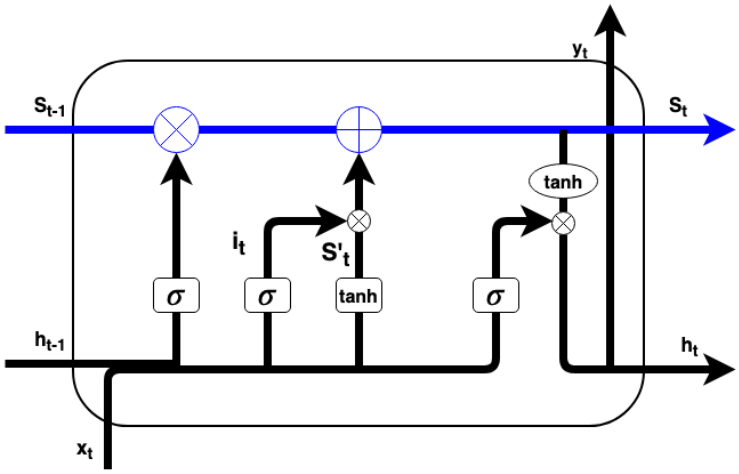
Rysunek 3.1: Komórki sieci rekurencyjnej. Źródło: Opracowanie własne

Sieci LSTM mają również taką łańcuchową strukturą, ale powtarzalna komórka ma bardziej skomplikowaną strukturę. Zamiast jednej warstwy sieci neuronowej istnieją cztery, które oddziałują na siebie w sposób przedstawiony na Rysunku 3.2.



Rysunek 3.2: Komórki sieci rekurencyjnej LSTM. Źródło: Opracowanie własne.

Na schemacie z Rysunku 3.2 każda linia przenosi cały wektor danych, od wyjścia jednego węzła do wejścia innego węzła. Kółka reprezentują operacje punktowe, takie jak np. dodawanie wektorów, podczas gdy pola to wyuczone warstwy sieci neuronowej. Łączące się linie oznaczają konkatenację, podczas gdy linia rozwidlająca oznacza, że treść jest kopiowana, a kopie podążają w kierunku strzałek.



Rysunek 3.3: Linia stanu sieci LSTM. Źródło: Opracowanie własne.

Pozioma linia biegnąca przez górę diagramu na Rysunku 3.3 obrazuje stan komórki S_t.

Stan komórki biegnie wzdłuż całego łańcucha komórek, z niewielkimi tylko liniowymi interakcjami. Informacje mogą łatwo i szybko przepływać bez zmian [5].

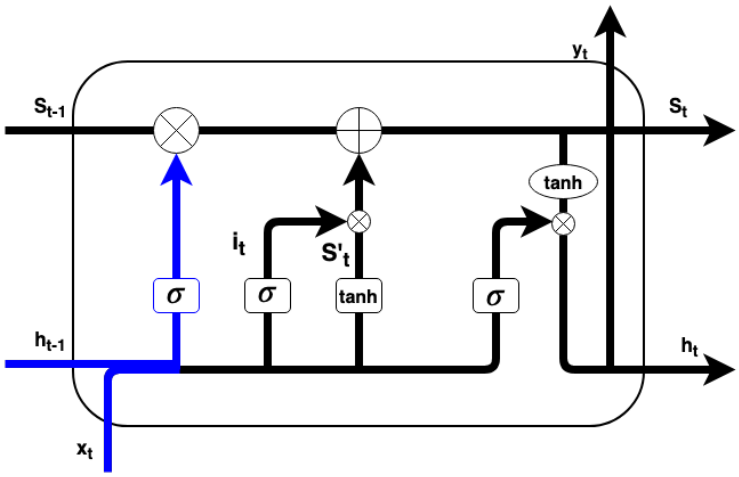
LSTM ma zdolność usuwania lub dodawania informacji do stanu komórki, za pomocą bramek „zapomnij” oraz.

Bramki to sposób na opcjonalne przepuszczanie informacji. Składają się one z warstwy sigmoidalnej i operacji mnożenia punktowego.

Warstwa sigmoidalna zwraca liczby z zakresu $[0, 1]$, opisując ile każdego wejścia należy podac dalej. Wartość 0 oznacza „niczego nie podawaj”, a wartość 1 oznacza „podaj wszystko”.

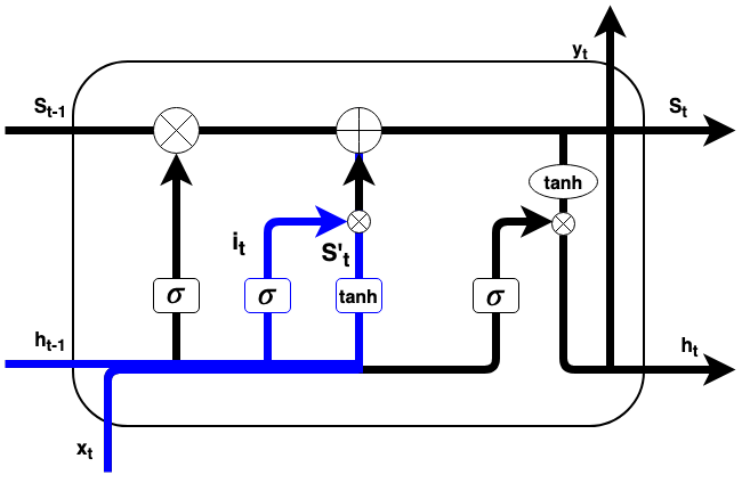
LSTM ma trzy bramki do kontrolowania stanu komórki.

Pierwszym krokiem w sieci LSTM jest decyzja, jakie informacje wyrzucimy ze stanu komórki. Ta decyzja została podjęta przez warstwę sigmoidalną zwaną bramką „zapomnij”,



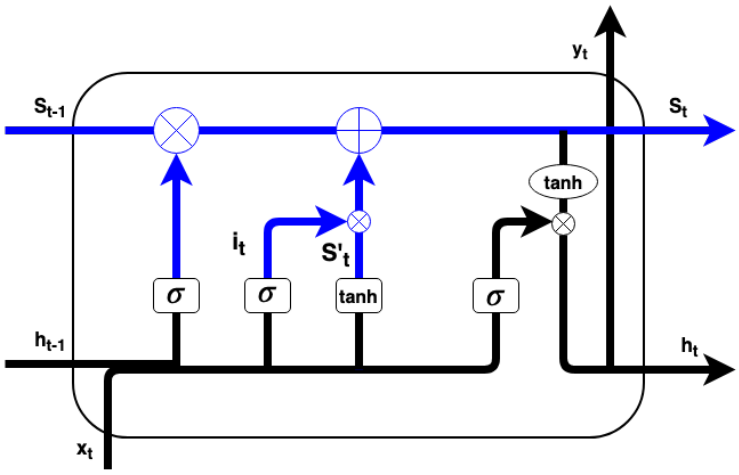
Rysunek 3.4: Bramka „zapomnij". Źródło: Opracowanie własne.

która bierzę pod uwagę h_{t-1} oraz x_t, po czym zwraca liczbę z zakresu $[0, 1]$ dla każdej liczby w stanie komórki S_{t-1}, jak to zostało pokazane na Rysunku 3.4.



Rysunek 3.5: Warstwa wejściowa oraz warstwa *tanh*. Źródło: Opracowanie własne.

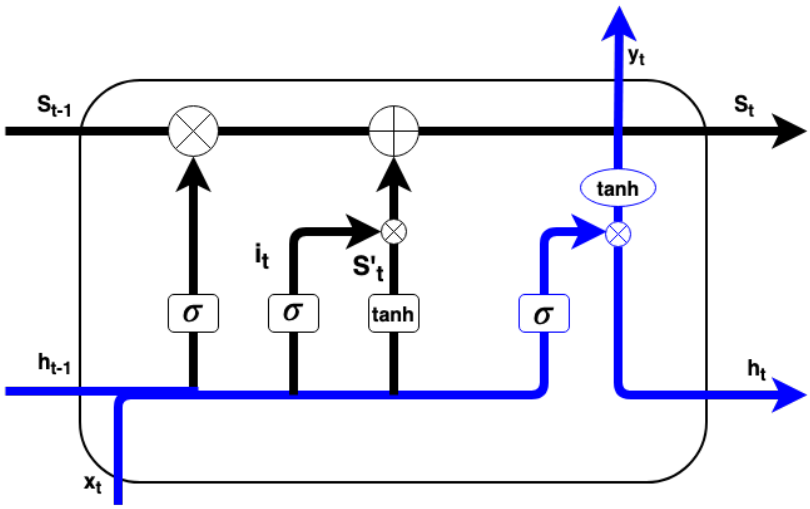
Następnym krokiem jest podjęcie decyzji, jakie nowe informacje będziemy przechowywać w stanie komórki. Ta decyzja składa się z dwóch części. Po pierwsze, warstwa sigmoidalna zwana „warstwą wejściową" decyduje, które wartości zaktualizujemy. Kolejnie warstwa *tanh* tworzy wektor nowych wartości kandydujących, które można dodać do stanu. W następnym kroku za pomocą obu warstw, tworzymy aktualizację stanu jak na Rysunku3.5.



Rysunek 3.6: Aktualizacja stanu komórki. Źródło: Opracowanie własne.

Nastepnie aktualizujemy poprzedni stan komórki, S_{t-1}, do nowego stanu komórki S_t. Poprzednie kroki przedstawione na Rysunkach 3.4 oraz 3.5 już zdecydowały, jak należy zaaktualizować stan komórki.

Mnożymy poprzedni stan przez funkcję f, zapominając o rzeczach, które wcześniej zostały przeznaczone do zapomnienia. Następnie dodajemy nowe wartości kandydujące S'_t, skalowane według wskaźnika aktualizacji i_t.



Rysunek 3.7: Warstwa wyjściowa. Źródło: Opracowanie własne

Dane wyjściowe będą oparte na stanie naszej komórki, ale będą wersją filtrowaną. Najpierw uruchamiamy warstwę sigmoidalną, która decyduje, które części stanu komórki będziemy generować. Następnie poddajemy stan komórki podąża do warstwy *tanh*, gdzie mnożymy go przez wyjście bramki sigmoidalnej, aby na wyjściu zostały tylko te części, na które się zdecydowaliśmy.

3.2 Przetwarzanie plików MIDI

MIDI (z ang.*Musical Instruments Digital Interface*) - cyfrowy system stworzony do przekazywania informacji pomiędzy elektronicznymi instrumentami muzycznymi. Standard MIDI powstał w 1983 roku w celu ujednolicenia komunikacji syntezatorów cyfrowych. MIDI zostało stworzone w celu przenoszenia prostych zbiorów poleceń (np. grając na jednym keyboardzie posiadającym system MIDI, można sterować innym urządzeniem, np. sentyzatorem, poprzez sygnał MIDI).[3]

MIDI pozwala na przekazanie wielu informacji m.in:

- note-on - kiedy klawisz został naciśnięty;
- note-off - kiedy klawisz został puszczony;
- key velocity - siła nacisku na dany klawisz;
- modulacja dźwięku;
- wysokość tonu, itd.

Czasowa synchronizacja zależności pomiędzy dźwiękami realizowana jest za pomocą informacji MIDI clock.

MIDI jest protokołem szeregowym. Pozwala przesłać w jednym momencie 128 informacji (np. rodzaj kanału, rodzaj komunikatu). Przesłanie jeden nuty trwa 1 milisekundę, przesłanie 10-głosowego akordu trwa 10 milisekund. Jeden posrt MIDI pozwala nadawać na 16 kanałach (co pozwala na wykorzystanie 16 instrumentów w jednym momencie)[12].

W celu wykorzystania plików do generowania muzyki, musimy odkodowywać i zakodowywać pliki MIDI. Będziemy korzystać z informacji o numerze kanału, numerze klawisza oraz wciśnięciu oraz puszczeniu danego klawisza. W pierwszym dziesięcio bitowym bajcie (tzw. sterującym) zaraz po najbardziej znaczącym bicie znajdziemy informacje o rodzaju komunikatu i numerze kanału, następnie oddzielone zerami są siedmio bitowe bajty danych zawierające wszystkie potrzebne nam informacje[3].

W takiej formie nie możemy przekazywać danych do modelu. Trzeba zakodować je w innej formie.

Jedną z opcji byłoby zakodowanie informacji o naciśnięciu danego klawisza (tak(1) / nie(0)) dla każdego z 88 klawiszy fortepianu w każdym kroku czasowym. Ponieważ jednak każdy pojedynczy klawisz nie jest wciśnięty przez większość czasu, byłoby bardzo trudno modelowi nauczyć się ryzykować naciśnięcie danego klawisza.[3]

W pierwszym modelu do zakodowania dźwięku użyjemy tylko numeru klawisza(pomijamy rytm) np. `26`. Akordy (czyli kilka dźwięków zagranych w jednym momencie) zakodujemy jako numery klawiszy oddzielone kropką, np. `28.34.38`. Podczas przetwarzania plików MIDI ze zbioru uczącego

W muzyce bardzo często traktujemy ćwierćnuty jako podstawową jednostkę (np. metrum $\frac{4}{4}$, $\frac{3}{4}$). Wtedy odpowiednio ósemka to 1/2 ćwierćnuty, a szesnastka 1/4 ćwierćnuty, co wykorzystamy do generowania rytmu [3].

W drugim modelu wykorzystamy podejście „notewise". Załóżmy, że dla każdej nuty start i stop jest innym słowem (więc słowo „`p28`"ozn. początek nuty 28 zagranej na fortepianie i „`endp28`" ozn. jej koniec). Użyjemy również „wait", aby zaznaczyć koniec kroku czasowego. Przykładowa sekwencja mogłaby wyglądac następująco: `p20 p32 wait3 endp20 p25 endp32 wait12`.

Przy zakodywaniu plików pojawia się problem kodowania oczekiwania, ponieważ może istnieć kilka kroków „`wait`" z rzędu, wtedy istnieje ryzyko, że model przez większość czasu będzie przewidywał pauzy. W celu uniknięcia tego problemu używamy konsolidowania bloków „`wait`".

Reprezentacja wykorzystana do modelu drugiego pozwala na stosunkowo łatwe dodanie kolejnych instrumentów. Główna różnica polega na kodowaniu plików MIDI. Tutaj nuty fortepianu oznaczaliśmy jako `p`, wystarczy np. nuty skrzypiec oznaczyć przedrostkiem `v`, a poza tym trenować model dokładnie w ten sam sposób tylko na odpowiednim zbiorze uczącym. Jedynym problemem jest o wiele mniejsza liczba plików MIDI z utworami na skrzypce / fortepian niż na utwory tylko na fortepian.

3.3 Model 1

Generator powstał na bazie rekurenyjnej sieci LSTM zbudowanej w oparciu o bibliotekę TensorFlow.

Wykorzystano trzy warstwy ukryte. Funkcją aktywacji jest funkcja „softmax". Model generuje utwory tylko ze zbioru nut, które wystąpiły w zbiorze uczącym. Model jest przystosowany tylko do generowania sekwencji dźwięków, melodii, bez rytmu.

Rytm jest generowany osobno. w następujący sposób, losujemy dwa wektory wartości, jeden, który będzie zawierał momenty w których dany dźwięk sekewncji ma się rozpocząć, oraz drugi wektor zawierający czasy trwania poszczególnych dźwięków, które zostały wygenerowane z wartości: cała nuta, półnuta, ćwierćnuta, ósemka i szestnastka, według odpowiednich prawdopodobieństw.

Szybkość uczenia się ustalono na $\eta = 0,005$, liczbę epok(generacji) na 150, długość sekwencji na 100, wielkość „mini-batcha" na 1024, a współczynnik zapominania jest równy $0,3$.

Kod potrzebny do stworzenia modelu powstał w oparciu o repozytorium Classical Piano Composer.

3.4 Model 2

Kolejnym modelem jest sieć LSTM, model wykorzystuje 62-nutowy zakres dźwięków (który pozwala na objęcie zakresu większości klasycznej muzyki fortepianowej) i pozwala na generowanie dowolnej liczby nut i instrumentów na raz.

W porównaniu do pierwszego modelu w tym rytm nie jest generowany losowo.

Generator powstał na bazie rekurencyjnej sieci AWD-LSTM z biblioteki FastAI. Można tu łatwo modyfikować hiperparametry (np. liczbę warstw LSTM). Trenuję model, który ma przewidzieć następną nutę lub akord, biorąc pod uwagę sekwencję wejściową. Po wytrenowaniu modelu tworzę pokolenia, próbując prognozować wyniki, a następnie przekazuje je z powrotem do modelu w celu przewidzenia następnego kroku itd. W ten sposób wygenerowane próbki mogą mieć dowolną długość.

Wykorzystano kodowanie „notewise"(co pozwoliło na generowanie rytmu jednocześnie z melodią), w którym przyjmowanie najlepszych prognoz zwykle prowadzi do tego, że model zapętla się i w kółko powtarza pewną sekwencję, lub generuje tylko rytm dla jednej nuty. Aby sobie z tym poradzić, używamy obciętego próbkowania (z ang. *truncated sampling*), gdzie wykorzystujemy n najbardziej prawdopodobnych kolejnych nut (zgodnie z prawdopodobieństwami przewidywanymi przez model) [10]. W ten sposób generacje

są bardziej zróżnicowane, a pojedyncza sekwencja może wygenerować próbki o dużej różnorodności.

Sieć zawiera cztery warstwy ukryte. Wielkość „mini-batcha" ustalono na 16, do uciętego próbkowania $n = 5$, a długość sekwencji wykorzystywana przy algorytmie BPTT jest równa 200.

Kod powstał w oparciu o repozytorium Musical Neural Net.

Rozdział 4

Ankieta

W ramach pracy inżynierskiej została również przeprowadzona ankieta, której celem było zbadanie, czy ankietowani będą w stanie odróżnić fragmenty utworów wygenerowane przez rekurencyjne sieci neuronowe z długą pamięcią krótkotrwałą od fragmentów stworzonych przez prawdziwych kompozytorów, a także przeanalizowanie wpływu wykształcenia muzycznego na wyniki.

Przy tworzeniu modeli ich jakość, precyzję i dokładność, bada się na zbiorze testowym i sprawdza poprawność odpowiedzi. W przypadku generowania muzyki, trudno jest określić jakość wygenerowanych sekwencji, ponieważ ich ocena będzie zależała od gustu muzycznego słuchaczy. Jednakże, sprawdzenie czy ankietowani potrafią wskazać próbkę wygenerowaną przez sztuczną inteligencję, daje informację zwrotną o podobieństwie muzyki generatywnej do tej tworzonej w tradycyjny sposób.

4.1 Opis ankiety

Ankietowani wybierali z dwóch próbek tę, która ich zdaniem została wygenerowana przez sztuczną inteligencję.

Pierwsza para: próbka wygenerowana przy pomocy pierwszego modelu, wytrenowanego na zbiorze zawierającym dziesięć sonat Wolfganga Amadeusza Mozarta oraz fragmentu autorstwa Igora Strawińskiego pochodzącego z baletu Ognisty Ptak.

Druga para: próbka wygenerowana przy pomocy pierwszego modelu, wytrenowanego na zbiorze zawierającym siedem dzieł Ludwiga van Beethovena oraz fragmentu polifonicznego utworu na fortepian autorstwa Rodiona Konstantinovicha Shchedrina.

Trzecia para: próbka wygenerowana przy pomocy pierwszego modelu, wytrenowanego na zbiorze zawierającym siedem dzieł Ludwiga van Beethovena oraz fragmentu polifonicznego utworu na fortepian autorstwa Rodiona Konstantinovicha Shchedrina.

Czwarta para: próbka wygenerowana przy pomocy pierwszego modelu, wytrenowanego na zbiorze zawierającym czterdziesi osiem dzieł Fryderyka Chopina oraz fragmentu linii melodycznej jazzowej piosenki Lee Morgana „Blue Train".

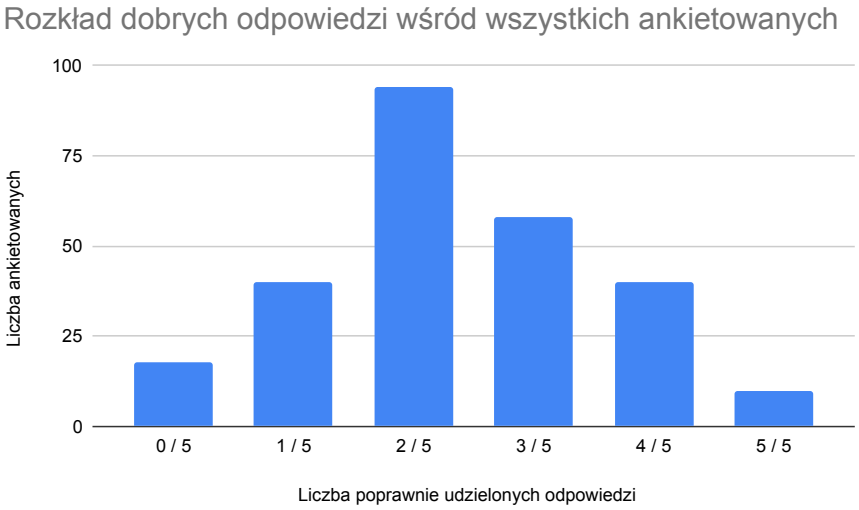
Piąta para: próbka wygenerowana przy pomocy drugiego modelu, wytrenowanego na zbiorze zawierającym utwory największych światowych kompozytorów, oraz fragmentu utworu Bacha pochodzącego z tego samego zbioru danych.

4.2 Opracowanie wyników

W ankiecie wzięło udział 260 osób. Wśród ankietowanych 200 osób nie posiada żadnego wykształcenie muzycznego, zaś 60 z nich uczęszczało do szkoły lub akademii muzycznej.Te dane zostały przedstawione na Rysunku 4.1.

Rysunek 4.1: Wykształcenie muzyczne wśród ankietowanych. Źródło: Opracowanie własne.

Rozkład dobrych odpowiedzi kształtuje się jak przedstawiono na Rysunku 4.2.



Rysunek 4.2: Rozkład dobrych odpowiedzi wśród wszystkich ankietowanych. Źródło: Opracowanie własne.

Podstawowe statystyki:

- Mediana liczby dobrych odpowiedzi to 2.
- Średnia liczba dobrych odpowiedzi wśród wszystkich ankietowanych, to $2,353$.
- Średnia wśród osób bez wykształcenia muzycznego, to: 2.35
- Średnia wśród ankietowanych z wykształceniem muzycznym to: $2,37$.
- Odchylenie standardowe ogółu badanych, to: $1,22$.
- Odchylenie standardowe wśród osób bez wykształcenia muzycznego, to: $1,44$.
- Odchylenie standardowe wśród osób z wykształceniem muzycznym, to: $1,19$.

Co zadziwiające wykształcenie muzyczne nie wpływa na wynik ankiety, osoby wykształcone uzyskiwały taką samą średnią liczbę dobrych odpowiedzi jak te, które nie posiadają specjalistycznej wiedzy. Jedynie wariancja jest mniejsza wśród osób z wykształceniem muzycznym. Co więcej wśród 10 osób, które uzyskały maksymalną liczbę dobrych odpowiedzi tylko 2 posiadają wykształcenie muzyczne, zaś wśród 18 osób z najsłabszym wynikiem, aż 6 posiada wykształcenie muzyczne. Można jedynie zauważyć, że odchylenie standardowe jest najmniejsze właśnie w grupie ankietowanych z wykształceniem muzycznym, co świadczy o najmniejszej zmienności wyników.

Przeanalizujmy szczegółowo dane ze wszystkich odpowiedzi przedstawione na Rysunku 4.3.

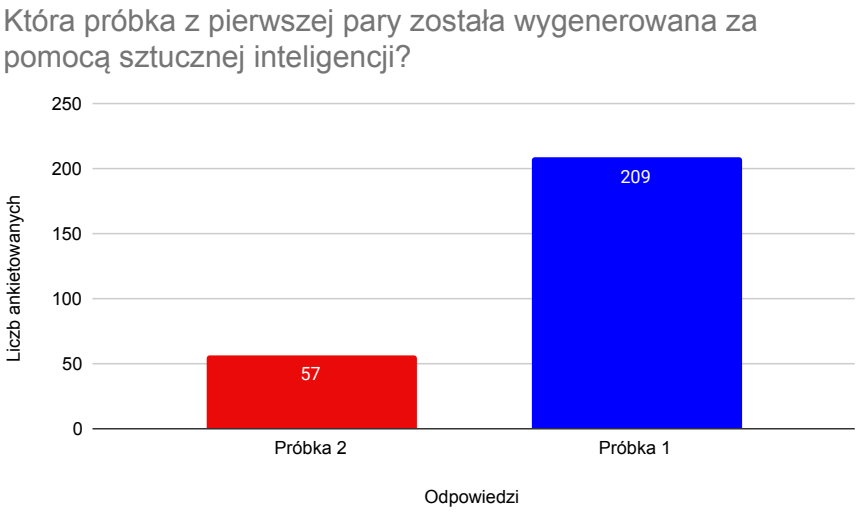
Pierwszym zaskakującym wnioskiem nasuwającym się z analizy wszystkich pytań, jest taki, że próbka z drugiego modelu, która jako jedyna uczy się zarówno melodii jak i rytmu nie zyskała zbyt dużej przewagi nad próbkami wygenerowanymi przez pierwszy model, gdzie rytm jest generowany losowo.

Z próbek wygenerowanych pierwszym modelem najlepsza okazała się próbka z pytania czwartego, co również nie jest takie oczywiste, ponieważ model był trenowany na utworach Fryderyka Chopina, który jak mówią eksperci ma dość ekspresyjny styl, który nie zawiera zbyt wielu schematycznych sekwencji.

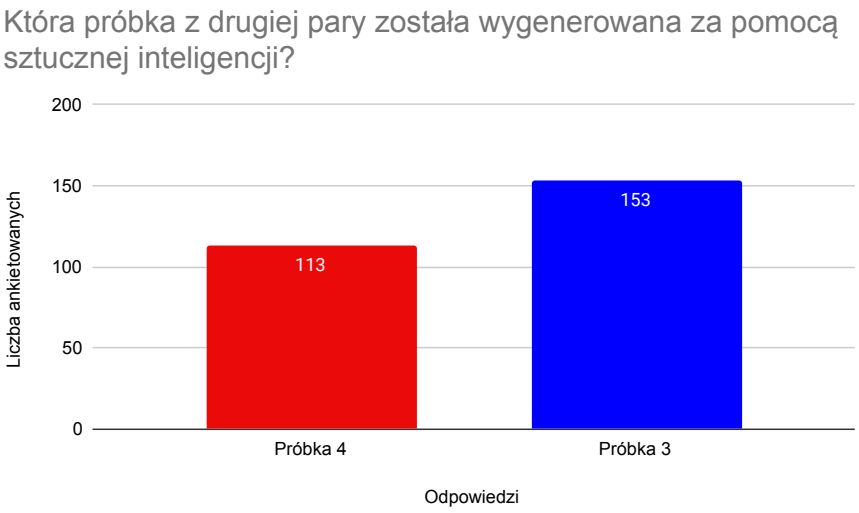
Trudno też doszukiwać się zależności pomiędzy wielkością zbioru danych uczących, a jakością wygenerowanej próbki.

Zastanawiająca jest wzrostowa tendencja w liczbie błędnych wskazań rosnąca wraz z kolejnymi pytaniami. Trudno wnioskować czy ma to związek z jakością wygenerowanych próbek, czy jest zupełnie przypadkowe, ale warto by było to zbadać np. stosując losową kolejność pytań.

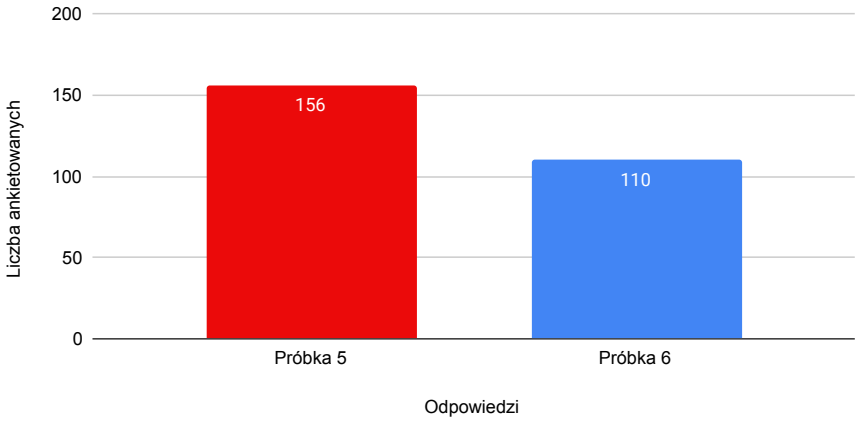
Najważniejszy wniosek wynikający z przeprowadzonej ankiety jest ten weryfikujący zasadność tej pracy. Na podstawie zebranych wyników możemy wysnuć tezę, że muzyka generowana przez rekurencyjne sieci nauronowe z długą pamięcią krótkotrwałą jest w stanie tworzyć sekwencje muzyczne, które są trudno rozróżnialne od fragmentów utworów skomponowanych przez prawdziwych kompozytorów.



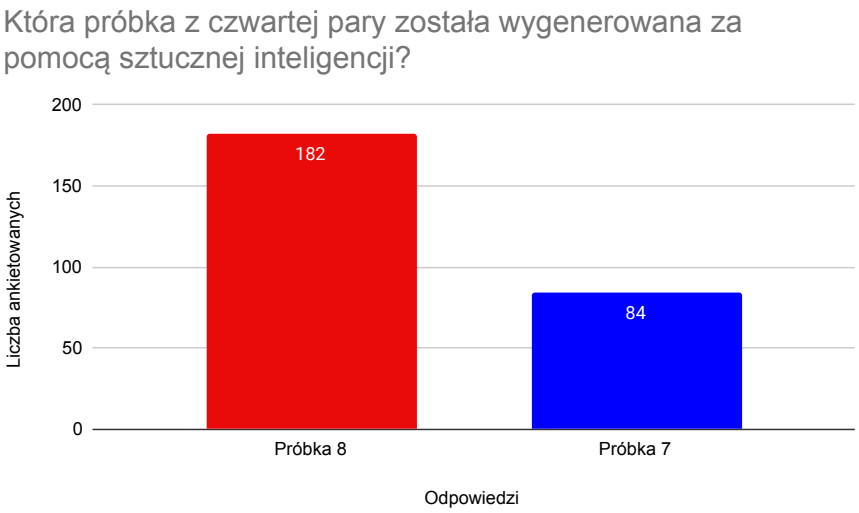
Pytanie 1



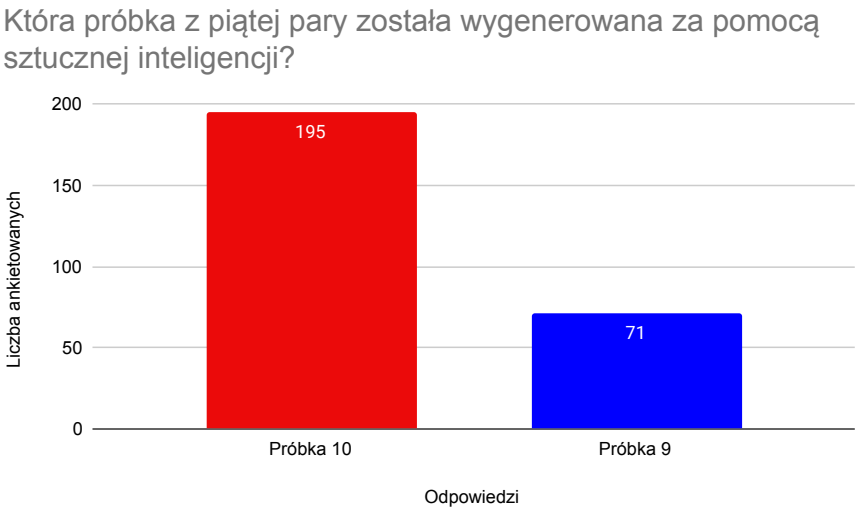
Pytanie 2



Pytanie 3



Pytanie 4



Pytanie 5

Rysunek 4.3: Rozkład poprawnych i niepoprawnych odpowiedzi na pytania ankietowe

Podsumowanie

W pracy przedstawiono metody generowania muzyki oparte na rekurencyjnych sieciach neuronowych z długą pamięcią krótkotrwałą.

Całość pracy rozpoczęło omówienie działania rekurencyjnych sieci neuronowych oraz sieci z długą pamięcią krótkotrwałą. Zostały omówione algorytmy treningu takich sieci.

Podczas pracy powstały dwa modele, które na podstawie zbiorów danych zawierajacych różne utwory generują zupełnie nowe sekwencje muzyczne. Modele różniły się od siebie podejściem do generowania rytmu.

Przeprowadzono badania ankietowe mające na celu weryfikację uzyskanych wyników.

W pracy pokazano, że możemy generować muzykę, która jest łudząco podobna do utworów stworzonych przez człowieka. Wyiki ankiety pokazały również, że wykształcenie muzyczne nie ma wpływu na postrzeganie generowanych sekwencji. Obie grupy ankietowanych miały duży problem ze wskazaniem próbek stworzonych przez sztuczną inteligencję.

Dowodzi to tego, że cel pracy został osiągnięty, generowane sekwencje mogą być interpretowane jako dzieła stworzone przez człowieka.

Na podstawie przeprowadzonych badań wydaje się, że bardzo ciekawym kierunkiem do dalszych rozważań mogą być modele bazujące na innych rodzajach rekurencyjnych sieci neuronowych, testowanie drugiego modelu dla muzyki wieloinstrumentalnej oraz eksperymenty z różnymi zbiorami danych.

Bibliografia

[1] Chollet, F. *Deep Learning. Praca z językiem Python i biblioteką Keras.* Helion SA., ul. Kościuszki 1c, 44-100 Gliwice, 2019.

[2] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2017.

[3] Hewahi, N., AlSaigal, S., AlJanahi, S. Generation of music pieces using machine learning: long short-term memory neural networks approach. *Arab Journal of Basic and Applied Sciences 26*, 1 (2019), 397–413.

[4] Hochreiter, S., Schmidhuber, J. Long short-term memory. *Neural Comput. 9*, 8 (Nov. 1997), 1735–1780.

[5] Mittal, A. Understanding rnn and lstm. `https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e`, 2019.

[6] Olah, C. Understanding lstm networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[7] Olah, C. Deepnlp — lstm (long short term memory) networks with math. `https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235`, 2018.

[8] Olah, C., Carter, S. Attention and augmented recurrent neural networks. *Distill* (2016).

[9] Pascanu, R., Mikolov, T., Bengio, Y. On the difficulty of training recurrent neural networks. In *ICML* (2012).

[10] Payne, C. Clara: Generating polyphonic and multi-instrument music using an awd-lstm architecture. `http://christinemcleavey.com/files/clara-musical-lstm.pdf`, 2018.

[11] Siegelmann, H. T. Computation beyond the turing limit. *Science 268*, 5210 (1995), 545–548.

[12] Spezzatti, A. Neural networks for music generation. `https://towardsdatascience.com/neural-networks-for-music-generation-97c983b50204`, 2019.

[13] Staudemeyer, R. C., Morris, E. R. Understanding lstm - a tutorial into long short-term memory recurrent neural networks. *ArXiv abs/1909.09586* (2019).

[14] Webros, P. J. Backpropagation through time: What it does and how to do it.

[15] Williams, R. J., Zipser, D. Gradient-based learning algorithms for recurrent networks and their computational complexity.

[16] Xenakis, I. *Musique formelles: nouveaux principes formels de composition musicale.* Addison-Wesley, Reading, Massachusetts, 1963.

[17] Yu, A. Generating music with artificial intelligence. `https://towardsdatascience.com/generating-music-with-artificial-intelligence-9ce3c9eef806`, 2018.

[18] Zocca, V., Spacagna, G., Slater, D., Roelants, P. *Deep Learning. Uczenie głębokie z językiem Python.* Helion SA., ul.Kościuszki 1c, 44-100 Gliwice, 2018.