UNIWERSYTET IM. ADAMA MICKIEWICZA W POZNANIU
WYDZIAŁ MATEMATYKI I INFORMATYKI

Cezary Adam Pukownik

Kierunek: Analiza i przetwarzanie danych
Numer albumu: 444337

Generowanie muzyki przy pomocy głębokiego uczenia

Music generation with deep learning

Praca magisterska
napisana pod kierunkiem
dr hab. Tomasza Góreckiego

POZNAŃ 2020

Poznań, dnia .....

OŚWIADCZENIE

Ja, niżej podpisany Cezary Pukownik, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt: "Generowanie muzyki przy pomocy głębokiego uczenia", napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

$[TAK]^\star$ - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

$[TAK]^\star$ - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

.....

Spis treści

Streszczenie

Abstract

Wstęp

Uczenie maszynowe w ostatnich latach mocno zyskało na popularności. Zastosowania i możliwości różnych algorytmów uczenia maszynowego czasami przekraczają nasze wyobrażenie o tym co komputer może zrobić. Niektóre aplikacje potrafią wręcz zaskoczyć użytkowników tym co potrafią zrobić. Wśród takich aplikacji znajdują się takie, które potrafią przewidywać następne wartości akcji giełdowych, rozpoznawać na filmie obiekty w czasie rzeczywistym, czy nawet prowadzić samochód. Algorytmy wyuczone proponują nam spersonalizowane reklamy, czy produkty na podstawie naszych upodobań. Najczęstsze zastosowania dotyczą przetwarzania obrazów lub tekstu, natomiast zastosowania w przetwarzaniu muzyki są niszowe i rzadziej spotykane.

Celem tej pracy jest stworzenie modelu sieci neuronowej, którego zadaniem będzie generowanie krótkich multi instrumentalnych klipów muzycznych.

W pierwszej części swojej pracy przedstawię podstawowe koncepcje związane z muzyką oraz sposobami jej reprezentacji. Następnie opiszę w jaki sposób działają sieci neuronowe, jak się uczą oraz podstawowe architektury sieci, które pomogą zrozumieć model który wykorzystałem.

Następnie przedstawię koncepcję działania modelu, jakie idee stały za wyborami, które podjąłem w projektowaniu sieci. W szczegółowy sposób opiszę sposób ekstrakcji danych tak aby mogły być one wykorzystane przez model. Opiszę architekturę którą wybrałem oraz przedstawię i opiszę fragmenty kodu w języku python.

W kolejnym rozdziale skupimy się na rezultatach pracy, przedstawię zalety i wady modelu. Przeprowadzę analizę jakie muzyczne koncepcje model się nauczył na podstawie danych oraz doprowadzę do ostatecznej konkluzji czy wygenerowana muzyka może być przyjemna dla odbiorcy.

Wprowadzenie do sieci neuronowych

Aby lepiej zrozumieć w jaki sposób odpowiednio skonstruowane sieci neuronowe potrafią sprostać takiemu zadaniu jak generowanie muzyki, w tym rozdziale przedstawię od podstaw zasady działania sieci neuronowych. Opiszę w jaki sposób można od regresji liniowej przejść do prostych sieci oraz w jaki sposób uczy się sieci neuronowe. Ostatecznie przedstawię architektury, które wykorzystałem w projekcie.

1.1. Regresja liniowa

Podstawą wszystkich sieci neuronowych jest regresja liniowa. W statystyce wykorzystywana, aby wyjaśnić liniowe zależności między zmiennymi.

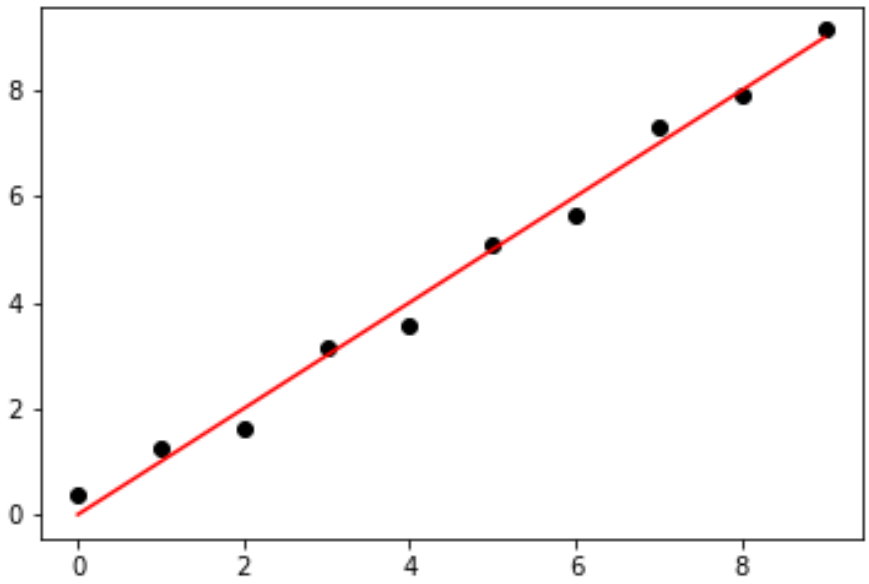
Prosty model regresji liniowej dla jednej zmiennej można opisać wzorem

$$y = ax + b + \epsilon,$$

gdzie
— y jest zmienną objaśnianą,
— x jest to zmienna objaśniająca,
— a jest parametrem modelu,
— b jest wyrazem wolnym modelu,
— ϵ jest składnikiem losowym [7].

Zadaniem jest znalezienie takiego parametru $a \in \mathbb{R}$ oraz wyrazu wolnego $b \in \mathbb{R}$, aby dla znanych wartości $x \in \mathbb{R}$ oszacowanie zmiennej objaśnianej $\hat{y} \in \mathbb{R}$ najlepiej opisywała zmienną objaśnianą $y \in \mathbb{R}$. Tak zdefiniowany model opisuje zmienną y z dokładnością do składnika losowego. W praktyce oznacza to, że szacowane modele będą przybliżeniem opisywanych zależności.

Wartość zmiennej objaśnianej y można również opisać za pomocą wielu zmiennych objaśniających. Wtedy dla zmiennych objaśniającej $x_1, x_2, ..., x_p \in \mathbb{R}$ szukamy parametrów $\theta_1, \theta_2, ..., \theta_p \in \mathbb{R}$, gdzie $p \in \mathbb{N}$ jest liczbą cech. Otrzymany w ten sposób model nazywany jest również hipotezą i oznaczamy go $h(x)$.



Rysunek 1.1. Regresja liniowa jednej zmiennej

$$h(x) = b + \theta_1 x_2 + \theta_2 x_2 + ... + \theta_p x_p + \epsilon = b + \sum_{i=1}^{p} \theta_i x_i + \epsilon.$$

Rysunek 1.1 przedstawia przykładowy model regresji liniowej jednej zmiennej, dopasowany do zbioru.

1.2. Uczenie modelu

Celem uczenia modelu jest znalezienie ogólnych parametrów, aby model dla wartości wejściowych x zwracał wartości predykcji $\hat{y}$ najlepiej opisującej całe zjawisko według pewnego kryterium. Formalnie, aby suma wszystkich różnic między predykcją, a rzeczywistością była najmniejsza.

$$\text{błąd} = \sum_{i=1}^{n} |\,\text{predykcja} - \text{rzeczywistość}\,|,$$

gdzie $n \in \mathbb{N}$ jest wielkością zbioru danych jakim dysponujemy. Minimalizując błąd dla modelu jesteśmy w stanie znaleźć przybliżenie funkcji $h(x)$.

1.2.1. Funkcja kosztu

W tym celu używa się funkcji $J_\theta(h)$, która zwraca wartość błędu między wartościami $h(x)$ oraz y dla wszystkich obserwacji. Taka funkcja nazywana jest funkcją kosztu (*ang. cost function*).

Dla przykładu regresji liniowej funkcją kosztu może być błąd średniokwadratowy (*ang. mean squared error*). Wtedy funkcja kosztu przyjmuje postać:

$$J_\theta(h) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h(x_i))^2.$$

Przy zdefiniowanej funkcji kosztu proces uczenia sprowadza się do znalezienia takich parametrów funkcji $h(x)$, aby funkcja kosztu była najmniejsza. Jest to problem optymalizacyjny sprowadzający się do znalezienia globalnego minimum funkcji.

1.2.2. Znajdowanie minimum funkcji

Aby znaleźć minimum funkcji f możemy skorzystać z analizy matematycznej. Wiemy, że jeśli funkcja f jest różniczkowalna, to funkcja może przyjmować minimum lokalne, gdy $f'(x_0) = 0$ dla pewnego x_0 z dziedziny funkcji f. Dodatkowo jeśli istnieje otoczenie punktu x_0, że dla wszystkich punktów z tego otoczenia spełniona jest nierówność:

$$f(x) > f(x_0),$$

to znaleziony punkt x_0 jest minimum lokalnym. W teorii należałoby zatem wybrać taką funkcję kosztu, aby była różniczkowalna. Rozwiązać równanie $J'_\theta(h) = 0$, następnie dla otrzymanych wyników sprawdzić powyższą nierówność oraz wybrać najmniejszy wynik ze wszystkich [6]. W praktyce rozwiązanie takiego równania ze względu na jego złożoność może się okazać niewykonalne. Aby rozwiązać ten problem powstały inne metody, które pozwalają szukać ekstremów funkcji, jednak nigdy nie będziemy mieli pewności, że otrzymany wynik jest minimum globalnym funkcji kosztu.

1.2.3. Metody gradientowe

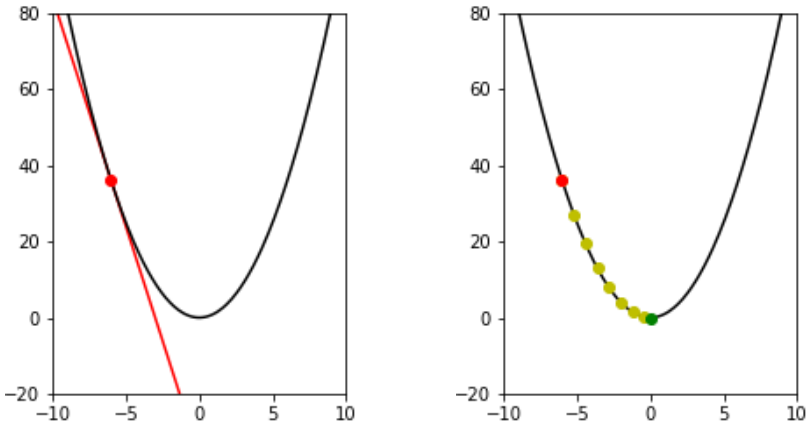
Metody gradientowe (*ang. gradient descent*) są to iteracyjne algorytmy służące do znajdowania minimum funkcji. Aby móc skorzystać z metod gradientowych analizowana funkcja musi być ciągła i różniczkowalna. Sposób ich działania można intuicyjnie opisać w następujących krokach.

1. Wybierz punkt początkowy.
2. Oblicz kierunek, w którym funkcja maleje.

3. Przejdź do kolejnego punktu zgodnie obliczonym kierunkiem o pewną małą wartość.

4. Powtarzamy, aż osiągniemy minimum funkcji.

Wizualizację algorytmu została przedstawiona na rysunku 1.2.



(a) Wyznaczenie gradientu **(b)** Iteracja kolejnych punktów

Rysunek 1.2. Wizualizacja algorytmu gradientu prostego

Dla funkcji $h(x)$ należy ustalić wartość początkową Θ_0 dla wszystkich parametrów θ_1 ... θ_p.

$$\Theta_0 = [\boldsymbol{\theta_1}, \boldsymbol{\theta_2}, ..., \boldsymbol{\theta_n}].$$

Następnie policzyć wszystkie pochodne częściowe $\frac{\partial J_\theta(h)}{\partial \theta_i}$. Otrzymamy w ten sposób gradient $\nabla J_\theta(h)$, gdzie

$$\nabla J_\theta(h) = \left[\frac{\boldsymbol{\partial J_\theta(h)}}{\boldsymbol{\partial \theta_1}}, \frac{\boldsymbol{\partial J_\theta(h)}}{\boldsymbol{\partial \theta_2}}, ..., \frac{\boldsymbol{\partial J_\theta(h)}}{\boldsymbol{\partial \theta_p}}\right].$$

Następnie obliczyć element Θ_{k+1} ze wzoru

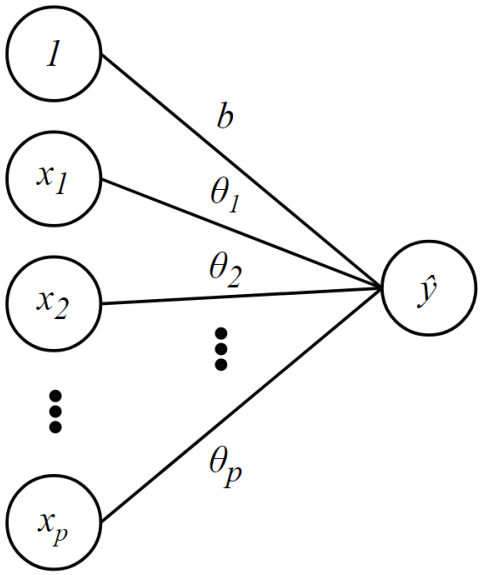
$$\Theta_{k+1} = \Theta_k - \alpha \nabla J_\theta(h),$$

gdzie $\alpha \in \mathbb{R}$ jest współczynnikiem uczenia (*ang. learning rate*), a $k \in \mathbb{N}$ jest kolejną iteracją algorytmu. Proces ten należy powtarzać do pewnego momentu. Najczęściej z góry określoną liczbę razy lub do momentu, gdy uzysk funkcji kosztu spowodowany następną iteracją jest mniejszy niż ustalona wartość. Otrzymany w ten sposób wektor parametrów Θ_k jest wynikiem algorytmu [1].

Wykorzystując metody gradientowe otrzymujemy wyuczony model. Parametry θ_i modelu $h(x)$ zostały ustalone w taki sposób, aby błąd między predykcją, a rzeczywistością był najmniejszy.

1.3. Regresja liniowa jako model sieci neuronowej

Omawiany model regresji możemy zapisać w sposób graficzny tak jak przedstawiono na rysunku 1.3.



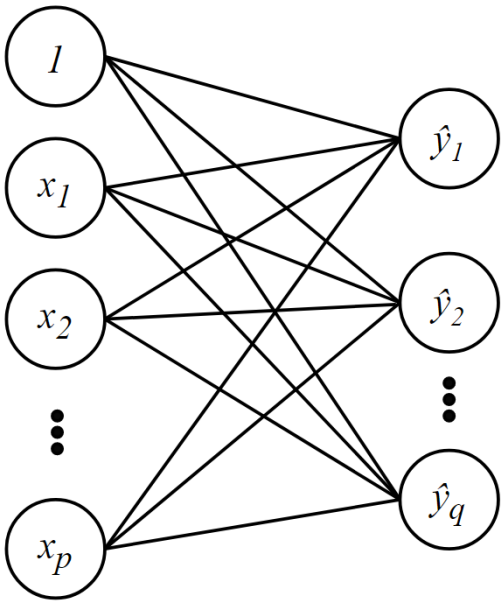
Rysunek 1.3. Regresja liniowa jako model sieci neuronowej

Każdy węzeł z lewej strony reprezentuje zmienną objaśniającą x_i. Połączenia nazywane są wagami (*ang. weights*) i reprezentują one parametry θ_i. Węzeł z prawej strony oznaczony jako $\hat{y}$ jest sumą iloczynów wag oraz wartości węzłów z prawej strony. Wtedy

$$\hat{y} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} b & \theta_1 & \theta_2 & \dots & \theta_p \end{bmatrix} = b + x_1\theta_1 + x_2\theta_2 + \cdots + x_p\theta_p = b + \sum_{i=1}^{p} x_i\theta_i,$$

co jest równoważne omawianemu modelowi regresji liniowej. Węzły sieci nazywane są neuronami, a wyraz wolny modelu b nazywany jest biasem (*ang. bias*).

W łatwy sposób możemy rozbudować ten model do regresji liniowej wielu zmiennych. Predykcją modelu nie będzie jak do tej pory jedna wartość $\hat{y}$, tylko wektor wartości $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_q$, który oznaczać będziemy jako $\hat{Y}$. Model ten został przedstawiony na rysunku 1.4.



Rysunek 1.4. Regresja liniowa wielu zmiennych jako model sieci neuronowej

Dla uogólnienia pojedyncze wagi modelu zapisywać będę jako w_{pq}, natomiast macierz wag jako W. Algebraicznie zapisaliśmy ten model jako

$$\begin{bmatrix} 1 & 1 & \ldots & 1 \\ x_{11} & x_{12} & \ldots & x_{1q} \\ x_{21} & x_{22} & \ldots & x_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ x_{p1} & x_{p2} & \ldots & x_{pq} \end{bmatrix} \begin{bmatrix} b_1 & w_{11} & w_{12} & \ldots & w_{1p} \\ b_2 & w_{21} & w_{22} & \ldots & w_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_q & w_{q1} & w_{q2} & \ldots & w_{qp} \end{bmatrix} = \begin{bmatrix} h_1(x) \\ h_2(x) \\ \vdots \\ h_q(x) \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_q \end{bmatrix}$$

$$\boldsymbol{b} + \boldsymbol{X}\boldsymbol{W} = \hat{\boldsymbol{Y}},$$

gdzie p jest liczbą zmiennych niezależnych, q jest liczbą zmiennych zależnych, X jest rozszerzonym do macierzy o rozmiarach q x p wektorem zmiennych

objaśniających, w taki sposób że $x_{i1} = x_{i2} = \cdots = x_{ip}$ dla $i = 1, 2, ..., q$, W jest macierzą wag o rozmiarach p x q, natomiast b jest sumą wyrazów wolnych $b_1, ..., b_q$. Możemy zauważyć, że model dla wielu zmiennych jest wieloma modelami dla jednej zmiennej, gdzie każdy model operuje na tych samych danych wejściowych. Taki model może być uznany za sieć neuronową i nazywany jest perceptronem.

1.4. Funkcje aktywacji

Omawiany model służy rozwiązywaniu problemu regresji, ponieważ wartości predykcji nie są uregulowane i mogą przyjmować wartości z $\mathbb{R}$. W celu przekształcenia tego modelu, aby móc go wykorzystać do rozwiązania problemu klasyfikacji, należy dodatkowo na otrzymanym wektorze $\hat{Y}$ wykonać pewną funkcję, która przekształci wynik. W tym celu używamy funkcji aktywacji (*ang. activation function*). Istnieje wiele różnych funkcji aktywacji, a każda posiada inną charakterystykę i wpływ na model. Najpopularniejszą grupą funkcji są funkcje sigmoidalne (*ang. sigmoid functions*). Jedną z nich jest funkcja logistyczna (*ang. logistic curve*)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

oraz wykresie przedstawionym na rysunku 1.5

Funkcja logistyczna ma pewne użyteczne właściwości, które pozwolą kontrolować wartości węzłów oraz rzutować wartości z całego $\mathbb{R}$ do wartości z przedziału $(0, 1)$. Dzięki tej właściwości funkcja logistyczna jest często używana, aby otrzymać prawdopodobieństwo wystąpienia pewnego zdarzenia. Dodatkowo funkcja logistyczna szybko przyjmuje wartości skrajne, co oznacza że dla bardzo dużych wartości ujemnych i bardzo dużych wartości dodatnich funkcja staje się mało wrażliwa na zmiany wartości wraz ze zmianą wartości argumentu [4].

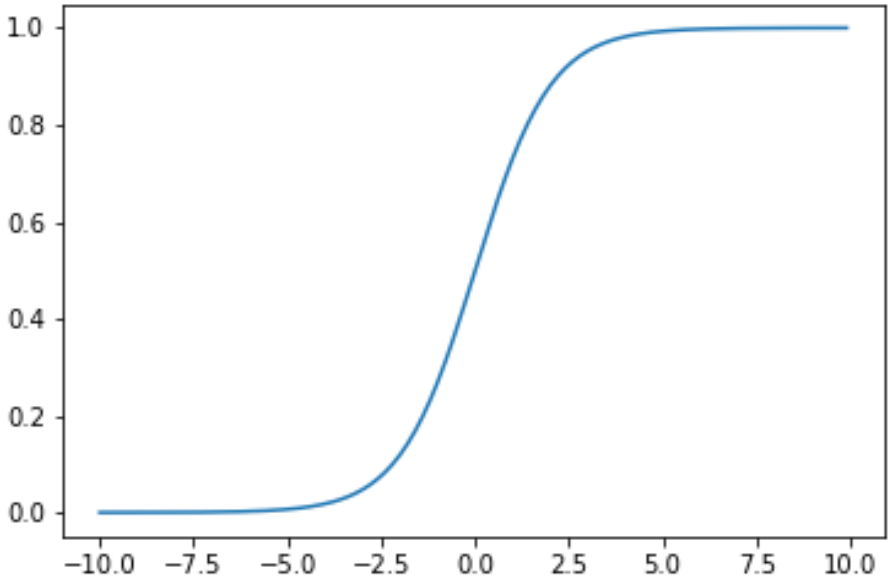
W ten sposób możemy w łatwy sposób zmienić model regresji liniowej na model regresji logistycznej.

$$\sigma(\boldsymbol{b} + \boldsymbol{X}\boldsymbol{W}) = \hat{Y}.$$

W dalszych częściach pracy, kiedy będę używał funkcji aktywacji nie wskazując na konkretną funkcję, będę wykorzystywał oznaczenie $AF(x)$.

1.5. Wielowarstwowe sieci neuronowe

Model omawiany wcześniej może posłużyć jako podstawowy element do budowania bardziej skomplikowanych modeli. Aby to zrobić, należy potraktować



Rysunek 1.5. Funkcja logistyczna

otrzymany wektor $\hat{\boldsymbol{Y}}$ jako wektor wejściowy do następnego podstawowego modelu. Składając ze sobą wiele perceptronów w jeden model, tworzymy warstwy (*ang. layers*) sieci neuronowej.

Wyróżniamy trzy rodzaje warstw:

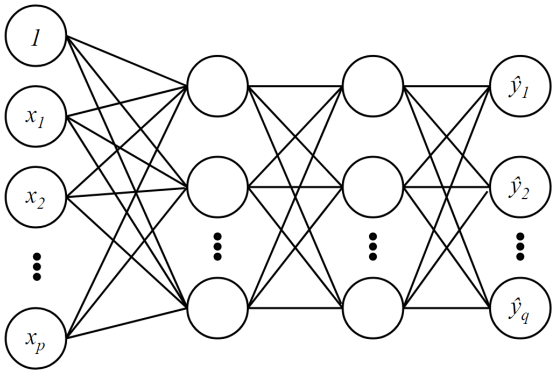
— warstwę wejściową (*ang. input layer*), która jest pierwszą warstwą modelu,
— warstwę wyjściową (*ang. output layer*), która jest ostatnią warstwą modelu,
— warstwy ukryte (*ang. hidden layer*), które są warstwami pomiędzy warstwą wejściową i wyjściową.

Na rysunku 1.6 przedstawiono sieć posiadającą warstwę wejściową, dwie warstwy ukryte oraz warstwę wyjściową.

Tego typu modele są głębokimi sieciami neuronowymi (*ang. deep neural networks*). Istnieje wiele różnych architektur głębokich sieci neuronowych, które wykorzystują te podstawowe koncepcje i rozszerzają je o dodatkowe warstwy, połączenia, funkcje aktywacji czy neurony o specjalnych właściwościach.

1.5.1. Jednokierunkowe sieci neuronowe

Jednokierunkowe sieci neuronowe (*ang. feedforward neural networks*) są to najprostsze sieci neuronowe, które wprost czerpią z omówionych wcześniej podstawowych warstw. Możemy się również spotkać z nazwą wielowarstwowy



Rysunek 1.6. Przykład modelu wielowarstwowej sieci neuronowej

perceptron (*ang. multi layer perceptron - MLP*) ze względu na fakt, że jest zbudowany z wielu perceptronów zaprezentowanych w rozdziale 1.3. Działają one w taki sposób, że zasila się je danymi do warstwy wejściowej, następnie sukcesywnie wykonuje się obliczenia do momentu dotarcia do końca sieci. Każdy krok z warstwy $k-1$ do warstwy k obliczany jest zgodnie ze wzorem [1]

$$\boldsymbol{X_k} = AF(\boldsymbol{b_k} + \boldsymbol{W_k}\boldsymbol{X_{k-1}}).$$

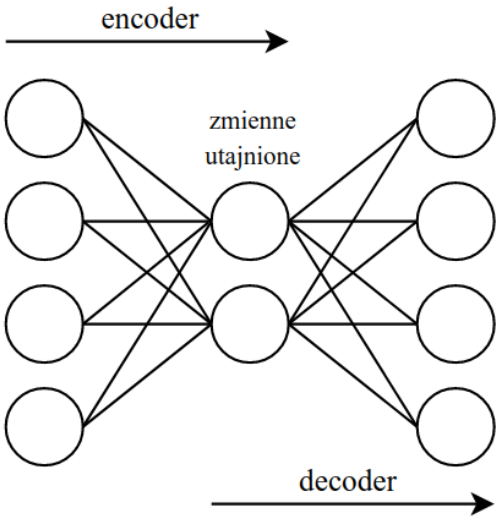
Propagacja wsteczna błędu

Kiedy używamy jednokierunkowych sieci neuronowych, zasilamy je danymi wejściowymi x ostatecznie otrzymując predykcję $\hat{y}$. Taki sposób działania nazywa się propagacją wprzód (*ang. forward propagation*). Podczas uczenia sieci kontynuuje się ten proces obliczając koszt $J(h)$. Propagacja wsteczna (*ang. back-propagation*) pozwala na przepływ informacji od funkcji kosztu wstecz sieci neuronowej, aby ostatecznie obliczyć gradient. Zasada działania algorytmu propagacji wstecznej błędu polega na sukcesywnym aktualizowaniu wag i biasów oraz przesyłaniu wstecz po warstwach sieci. Dzięki temu jesteśmy w stanie wyuczyć sieć oraz obliczyć optymalne wagi i biasy dla całej sieci neuronowej.

1.5.2. Autoencoder

Autoencoder jest szczególnym przypadkiem sieci neuronowej. Posiada jedną warstwę ukrytą, a rozmiar warstwy wejściowej musi być równy rozmiarowi warstwy wyjściowej, tworząc w ten sposób symetryczną sieć, której kształt przypomina klepsydrę. Przykład autoencodera przedstawiono na rysunku 1.7.

Podczas uczenia autoencodera przedstawia się dane wejściowe jako cel. W ten sposób ta architektura stara się odtworzyć funkcje identycznościowe. Zada-



Rysunek 1.7. Przykład modelu autoencodera

nie nie jest trywialne jak mogłoby się zdawać, ponieważ zazwyczaj ukryta warstwa jest mniejszego rozmiaru niż dane wejściowe. Z tego względu autoencoder jest zmuszony do wydobycia istotnych cech danych wejściowych, skompresowania, a następnie jak najwierniejszego ich odtworzenia. Część kompresująca dane nazywana jest encoderem, natomiast część dekompresująca decoderem. Cechy, które zostały odkryte przez autoencoder nazywane są zmiennymi utajnionymi (*ang. latent variables*). Zarówno encoder jak i dekoder można wyodrębnić z autoencodera i wykorzystywać go jako osobną sieć neuronową.

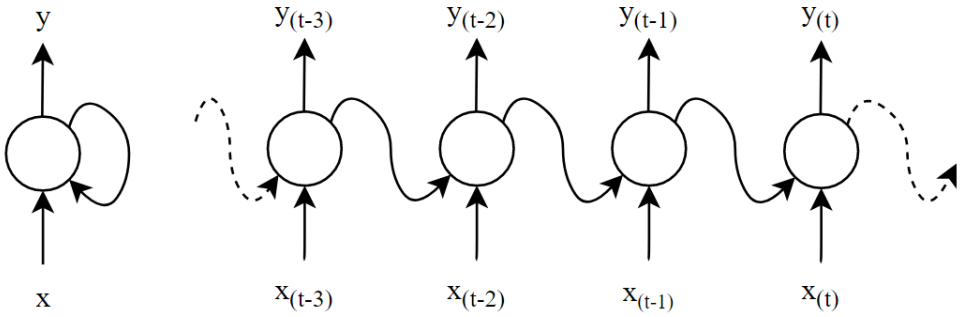
Ciekawą cechą decodera jest jego generatywny charakter, ponieważ dostarczając zupełnie nowe informacje jako zmienne wejściowe, decoder odtworzy je na podobieństwo danych, na których został nauczony.

1.5.3. Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe (*ang. recurrent neural networks; RNN*) w uproszczeniu są to MLP posiadające pamięć. Wykorzystywane są do analizowania i przewidywania sekwencji wartości uporządkowanych w czasie. Rekurencyjne sieci neuronowe znalazły zastosowanie w przetwarzaniu języka naturalnego, np. tłumaczenia na różne języki świata. Potrafią poradzić sobie z różnej długości sekwencjami od krótkich zawierających kilka elementów do bardzo długich jak próbki audio, czy tekst zawierający dziesiątki tysięcy kroków czasu.

Rekurencyjne sieci neuronowe działają podobnie do omawianych w roz-

dziale 1.5.1 sieci jednokierunkowych z tym wyjątkiem, że kierunek przepływu informacji płynie również wstecz sieci. Jeden neuron sieci RNN otrzymuje dane wejściowe $x_{(t)}$, wytwarza dane wyjściowe $y_{(t)}$, a następnie wysyła te dane wyjściowe z powrotem do samego siebie. W ten sposób neuron RNN posiada dwa wejścia $x_{(t)}$ oraz $y_{(t-1)}$. Możemy również zaprezentować sieć RNN w postaci odwiniętej w czasie (*ang. unrolled through time*) tak jak zaprezentowano to na rysunku 1.8.



Rysunek 1.8. Rekurencyjny neuron (po lewej) odwinięty w czasie (po prawej)

Gdyby rozważyć całą warstwę neuronów tego typu, wtedy warstwa przyjmowała by dwie macierze wag $\boldsymbol{W_x}$ oraz $\boldsymbol{W_y}$. Dane wyjściowe całej warstwy zostaną obliczone wtedy zgodnie ze wzorem

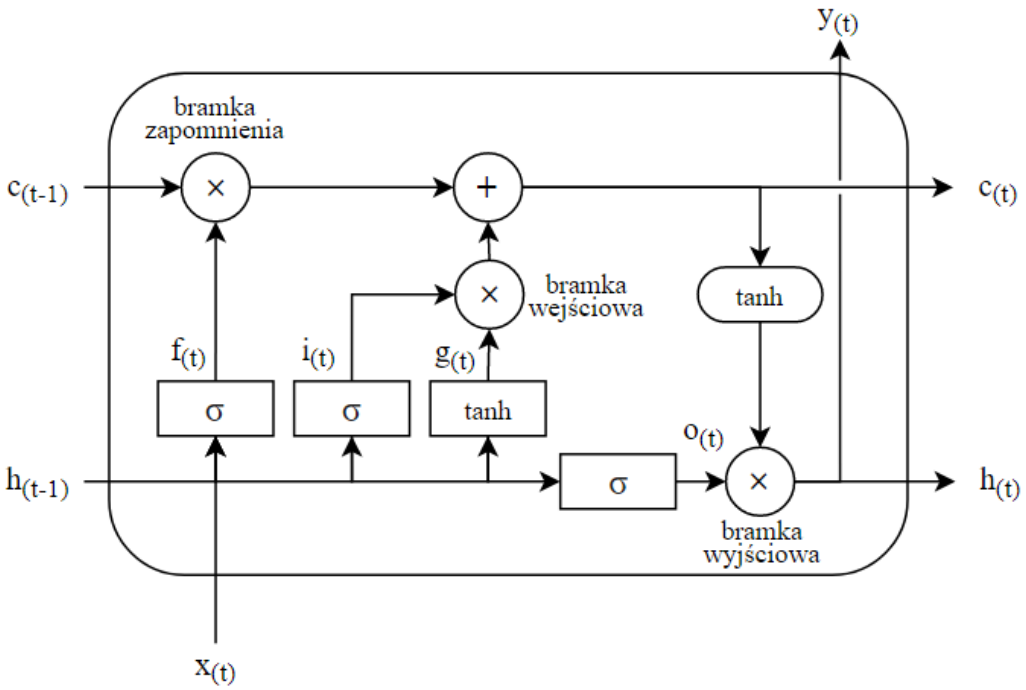
$$y_{(t)} = AF(\boldsymbol{W_x}^\top x_{(t)} + \boldsymbol{W_y}^\top y_{(t-1)} + b).$$

Aby wytrenować sieć neuronową stosuje się propagację wsteczną w czasie (*ang. backpropagation through time; BPTT*). Polega ona na odwinięciu sieci RNN, a następnie zastosowania zwykłej metody wstecznej propagacji[3].

1.5.4. LSTM

Komórki LSTM (*ang. long-short term memory*) są rozszerzeniem neuronów sieci rekurencyjnych. Pozwalają wykrywać zależności w danych w długim okresie. Posiadają dwa wektory opisujące stan neuronu. Wektor $\boldsymbol{h_{(t)}}$ określa stan krótkookresowy i wektor $\boldsymbol{c_{(t)}}$ określa stan długookresowy.

Główny pomysł na funkcjonowanie komórek LSTM był taki, aby sieć sama mogła się nauczyć jakie informacje są istotne i je przechować, a które informacje można pominąć, zapomnieć. Schemat komórki LSTM przedstawiono na rysunku 1.9. Aby to osiągnąć powstała idea bramek (*ang. gates*), oraz kontrolerów bramek (*ang. gate controllers*). W komórce LSTM wyróżniamy trzy



Rysunek 1.9. Komórka LSTM

bramki. Bramkę zapomnienia (*ang. forget gate*) sterowaną przez $f_{(t)}$, bramkę wejściową(*ang. input gate*) sterowaną przez $i_{(t)}$, oraz bramkę wyjściową (*ang. output gate*), sterowaną przez $o_{(t)}$. Przepływ danych w komórce LSTM zaczyna w miejscu gdzie wektor wejściowy $x_{(t)}$ i poprzedni krótkoterminowy stan $h_{(t-1)}$ trafiają do czterech warstw. Główną warstwą jest ta zwracająca $g_{(t)}$. W podstawowej komórce RNN jest tylko ta warstwa. Pozostałe trzy warstwy po przejściu przez funkcje logistyczne trafiają do bramek. Bramka zapomnienia kontroluje, które informacje z długookresowego stanu $c_{(t-1)}$ powinny zostać wykasowane. Bramka wejściowa kontroluje jakie informacje z $g_{(t)}$ powinny zostać przekazane dalej i dodane do następnego stanu długookresowego $c_{(t)}$. Bramka wyjściowa odpowiada za wybranie odpowiednich elementów z stanu długookresowego i przekazanie ich następnych kroku. Wynik komórki zostaje przekazany do wyjścia komórki $y_{(t)}$ oraz jako następny stan krótkoterminowy $h_{(t)}$.

Kolejne etapy komórki LSTM obliczane są zgodnie z poniższymi wzorami:

$$i_{(t)} = \sigma(\boldsymbol{W}_{xi}^{\top} x_{(t)} + \boldsymbol{W}_{hi}^{\top} h_{(t-1)} + b_i),$$

$$f_{(t)} = \sigma(\boldsymbol{W}_{xf}^{\top} x_{(t)} + \boldsymbol{W}_{hf}^{\top} h_{(t-1)} + b_f),$$

$$o_{(t)} = \sigma(\boldsymbol{W}_{xo}^{\top} x_{(t)} + \boldsymbol{W}_{ho}^{\top} h_{(t-1)} + b_o),$$

$$g_{(t)} = tanh(\boldsymbol{W}_{xg}^{\top} x_{(t)} + \boldsymbol{W}_{hg}^{\top} h_{(t-1)} + b_g),$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)},$$

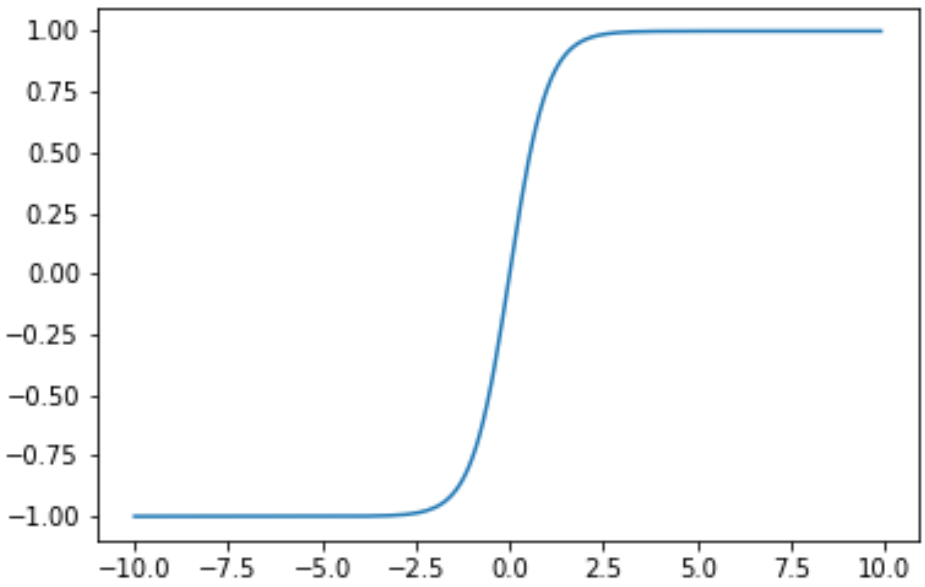
$$y_{(t)} = h_{(t)} = o_{(t)} \otimes tanh\left(c_{(t)}\right),$$

gdzie $\boldsymbol{W}_{xi}$, $\boldsymbol{W}_{xf}$, $\boldsymbol{W}_{xo}$, $\boldsymbol{W}_{xg}$ są to macierze wag dla każdej w czterech warstw połączonych z wektorem wejściowym $x_{(t)}$, $\boldsymbol{W}_{hi}$, $\boldsymbol{W}_{hf}$, $\boldsymbol{W}_{ho}$, $\boldsymbol{W}_{hg}$ są to macierze wag dla każdej w czterech warstw połączonych z poprzednim krótkookresowym stanem $h_{(t-1)}$, a b_i, b_f, b_o, b_g to biasy dla każdej z tych warstw [3].

Funkcja *tanh* to tangens hiperboliczny, jedna z funkcji sigmoidalnych. Wykres funkcji *tanh* został przedstawiony na rysunku 1.10

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

To co różni funkcję *tanh* od σ to zakres przyjmowanych wartości. Tangens hiperboliczny przyjmuje wartości z przedziału $(-1, 1)$.

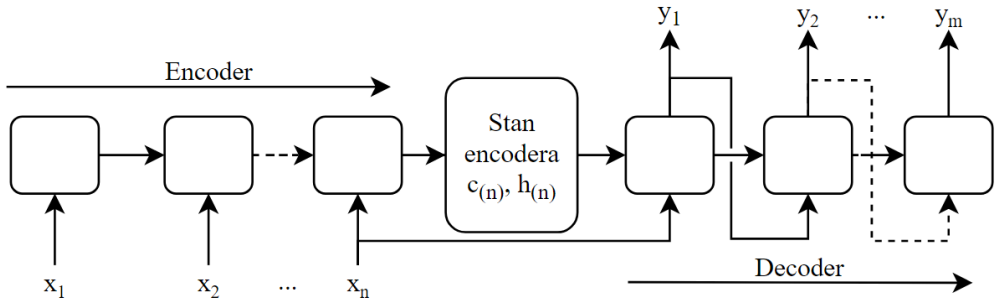


Rysunek 1.10. Tangens hiperboliczny

1.5.5. Sequence-to-sequence

Model w architekturze Sequence-to-sequence (*ang. seq2seq*) został wynaleziony z myślą o tłumaczeniu maszynowym języków, ale zastosowanie dla niego znaleziono również w rozpoznawaniu mowy, opisywaniu wideo, czy tworzeniu chatbotów. Jego główną zaletą jest przetwarzanie sekwencji elementów o różnych długościach. Jest to naturalne, ponieważ tłumacząc z języka na język często tą samą sentencję można wyrazić różną liczbę słów. Dla przykładu zdanie po Polsku „Co dzisiaj robisz?" zawiera trzy słowa, natomiast przetłumaczone na Angielski „What are you doing today?" zawiera pięć słów. Nie można tego osiągnąć zwykłą siecią LSTM, dlatego model seq2seq został zaprojektowany, aby móc go zastosować do tego typu problemów [5].

Model sequence-to-sequence ma dwie części, encoder i decoder. Obie części są w zasadzie dwiema zupełnie osobnymi modelami, połączonymi ze sobą w jedną sieć. Schemat modelu sequence-to-sequence przedstawiono na rysunku 1.11. Zadaniem encodera, podobnie jak zostało to opisane w rozdziale 1.5.2 o autoencoderze, jest wydobycie z wektora wejściowego najistotniejszych informacji i skompresowanie ich. Następnie wektor stanu encodera jest przekazywany do decodera, który na jego podstawie rekonstruuje sekwencję.



Rysunek 1.11. Architektura modelu sequence-to-sequence

Więcej szczegółów technicznych dotyczących modelu sequence-to-sequence przedstawię w dalszych rozdziałach pracy.

Wprowadzenie do teorii muzyki

W tym rozdziale przedstawię podstawowe koncepcje muzyczne oraz sposoby reprezentacji muzyki.

2.1. Podstawowe koncepcje muzyczne

2.1.1. Dźwięk muzyczny

Drgania powietrza z otoczenia człowieka są przetwarzane w mózgu i rozumiane jako dźwięki. Takie drgania nazywamy falą dźwiękową. Dźwięk muzyczny jest to fala dźwiękowa, którą wytwarza instrument muzyczny. Dźwięk muzyczny charakteryzuje się trzema podstawowymi parametrami:

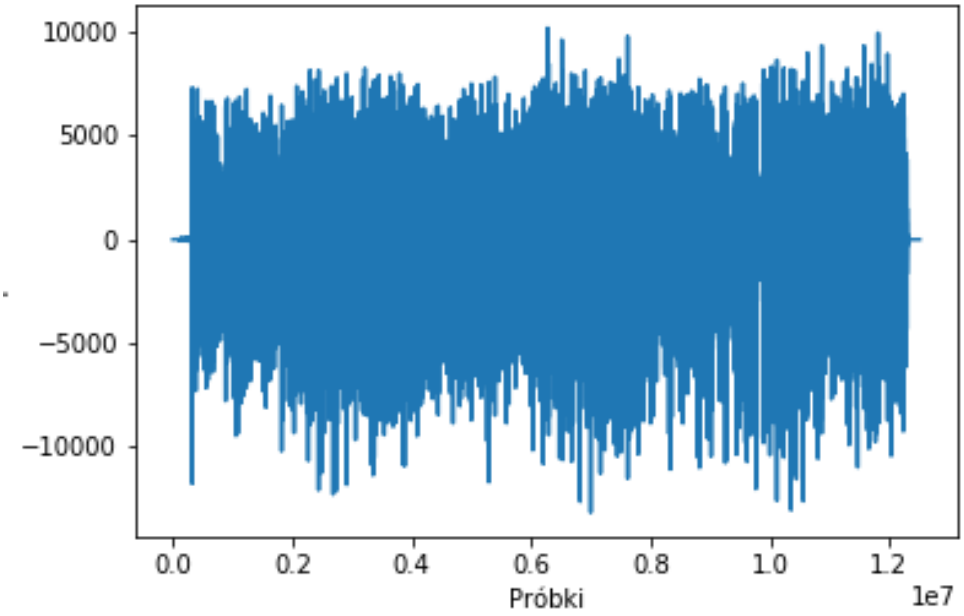
— wysokością (*ang. pitch*) - jest to częstotliwość drgań wyrażona w hercach. Im większa częstotliwość tym dźwięk jest rozumiany jako wyższy. Zakres słyszalny dla człowieka wynosi od 20Hz do 20kHz.
— głośność (*ang. velocity*) - jest to amplituda drgań fali dźwiękowej. Im większa amplituda, tym dźwięk jest odczuwany jako głośniejszy,
— długość (*ang. duration*) - jest to czas z jakim dźwięk wybrzmiewa, np. 2 sekundy.

2.1.2. Sygnał dźwiękowy

W rzeczywistości utwór muzyczny jest zazwyczaj kombinacją wielu fal dźwiękowych, o różnych charakterystykach i nazywany jest sygnałem dźwiękowym. Wizualizację sygnału dźwiękowego przedstawiono na Rysunku 2.1

2.1.3. Zapis nutowy

Reprezentacja muzyki jako sygnału dźwiękowego przechowuje informacje o dokładnym brzmieniu danego utworu tzn. jakie drgania należy wytworzyć, aby móc odtworzyć muzykę. Taki zapis nie informuje nas bezpośrednio jakie instrumenty zostały użyte, jakie wysokości i długości dźwięków zostały wykorzystane. Dlatego ludzkość na przestrzeni wieków opracowała abstrakcyjne obiekty, które reprezentują utwór w czytelny dla człowieka sposób.



Rysunek 2.1. Przykład przebiegu fali dźwiękowej

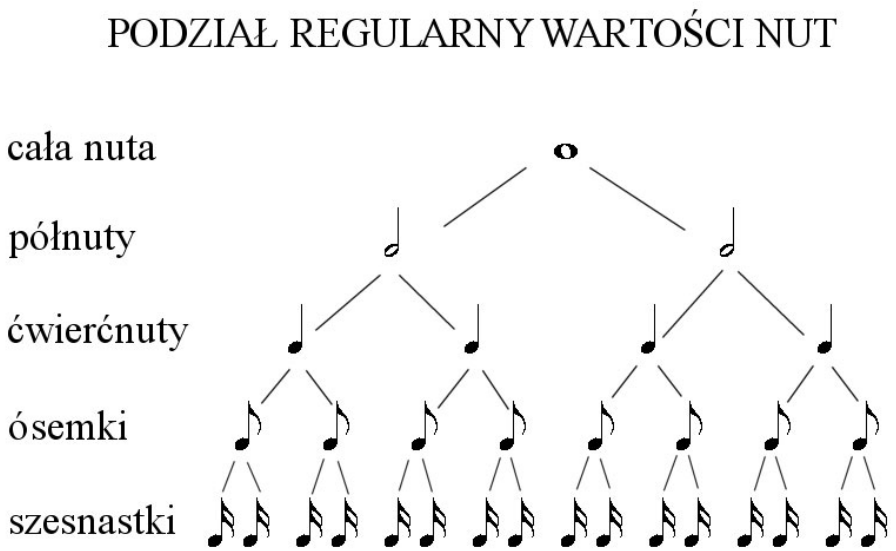
Tempo

W muzyce symbolicznej tempo informuje nas o prędkości utworu. W muzyce klasycznej stosowało się opisowy sposób dostosowywania tempa np. Allegro - Szybko lub Adagio - wolno. Jak można szybko stwierdzić są to zwroty subiektywne i nie wyznaczają tempa jednoznacznie. Obecnie wyraża się tempo w liczbie uderzeń na minutę (*ang. beats per minute; BPM*). I tak Allegro jest to od 120 do 168 BPM a Adagio od 66 do 76 BPM [8].

Nuta

Nuta jest to graficzna reprezentacja dźwięku muzycznego. Informuje nas ona o dwóch parametrach dźwięku, wysokości oraz długości dźwięku. Długość dźwięku nazywa się jej wartością. Podstawową wartością nuty jest ćwierćnuta, odpowiada ona jednemu uderzeniu (*ang. beat*). Ta wartość pozwala nam zrozumieć jak długo należy wygrywać nutę relatywnie do pozostałych nut w utworze. Jeśli obok siebie ustawimy dwie nuty o wartościach ćwierćnuty i ósemki, wiemy że tę drugą nutę powinniśmy zagrać dwa razy krócej niż pierwszą. Aby wiedzieć dokładnie jak długo powinna wybrzmiewać nuta musimy odwołać się do tempa utworu. Dla przykładu w tempie 60 BPM w ciągu minuty zagramy dokładnie 60 ćwierćnut. Kolejne wartości tworzone poprzez sumowanie lub podział dłu-

gości ćwierćnuty. Półnuta trwa tyle co dwie ćwierćnuty, cała nuta tyle co dwie półnuty, ósemka trwa połowę czasu ćwierćnuty, a szesnastka połowę ósemki itd. Zostało to pokazane na rysunku 2.2.



Rysunek 2.2. Podział wartości nut[1]

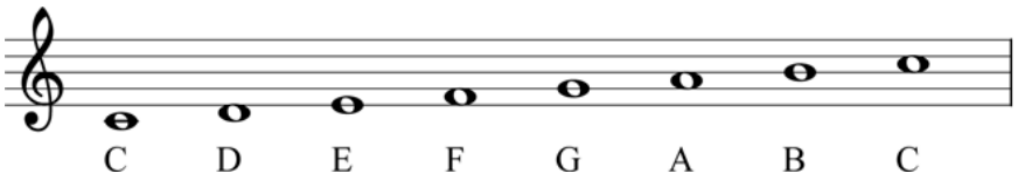
Tak jak pisałem wcześniej, wysokość dźwięku jest to częstotliwość drgań fali dźwiękowej wyrażona w hercach. W muzyce symbolicznej dla uproszczenia wybrane częstotliwości zostały nazwane literami alfabetu C, D, E, F, G, A, H. Każdej literze przypisana jest częstotliwość zgodnie z Tabelą 2.1

Dźwięk	Częstotliwość
C_4	261,6
D_4	293,7
E_4	329,6
F_4	349,2
G_4	391,9
A_4	440,0
H_4	493,9

Tabela 2.1: Dźwięki symboliczne oraz ich częstotliwości

[1] https://www.infomusic.pl/poradnik/46934,poradnik-teoria-muzyki-rytm 5 kwietnia 2020 12:46

W zapisie nutowym, aby nucie nadać wysokość umieszcza się ją w odpowiednim miejscu na pięciolinii. Przedstawione powyżej dźwięki zapisaliśmy w taki sposób jak przedstawiono na Rysunku 2.3



Rysunek 2.3. Zapis nut na pięciolinii[2]

Interwały

O interwałach mówimy, kiedy porównoujemy ze sobą dwie nuty. Interwał jest to odległość między nutami, liczona w półnutach. Półnuta jest to najmniejsza odległość miedzy nutami we współczesnej notacji muzycznej. Oktawa jest podzielona na 12 równych części. Pomiędzy dźwiękami C i D jest odległość dwóch półnut, natomiast miedzy F oraz F# jest odległość jednej półnuty. Dla ludzkiego ucha w celu rozpoznania melodii istotniejsze są interwały między kolejnymi nutami niż konkretna wysokość dźwięków.

Oktawy

Oktawą nazywamy zestaw ośmiu nut od C do H. Podane w Tabeli 2.1 częstotliwości nut odpowiadają dźwiękom w oktawie czwartej. Dlatego w indeksie dolnym nuty widnieje liczba 4. Aby utworzyć dźwięk, np. A_5 należy pomnożyć częstotliwość dźwięku A_4 razy dwa, natomiast aby utworzyć dźwięk A_3, należy tę częstotliwość podzielić przez dwa.

$$A_5 = 440Hz * 2 = 880Hz$$

$$A_3 = 440Hz/2 = 220Hz$$

W ten sposób możemy stworzyć nieskończenie wiele oktaw, jednak w rzeczywistości używa się nut od C0 do C8.

Akord

Gdy w jednym momencie zabrzmią dwie lub więcej różnych nut, wtedy mówimy o akordzie. Akord potrafi dodać emocje do brzmienia całego utworu.

[2] https://amplitudaschool.weebly.com/lekcja-11.html 5 kwietnia 2020 13:24

Skala

Skala jest to zestaw nut, które dobrze ze sobą brzmią. Skalę opisujemy dwoma parametrami, tonacją, oraz modem. Tonacja jest to pierwsza nuta dla skali. Mod natomiast jest to zestaw interwałów liczony od pierwszej nuty. np. C-Dur, gdzie C jest wartością początkową, a Dur opisuje sekwencję interwałów. Możemy utworzyć inne skale, np G-Dur, używając tych samych interwałów, ale zaczynając od innej nuty.

2.2. Cyfrowa reprezentacja muzyki symbolicznej

2.2.1. Standard MIDI

Standard MIDI (ang. Musical Instrument Digital Interface) został stworzony w 1983 aby umożliwić synchronizację i wymianę informacji między elektronicznymi urządzeniami muzycznymi takimi jak syntezatory, keyboardy czy sekwencery. W późniejszych latach został on zaadaptowany do środowiska komputerowego jako cyfrowa reprezentacja muzyki symbolicznej.

```
note_on channel=0 note=48 velocity=100 time=0
note_on channel=0 note=53 velocity=100 time=0
note_on channel=0 note=60 velocity=100 time=0
note_on channel=0 note=48 velocity=0 time=220
note_on channel=0 note=48 velocity=100 time=0
note_on channel=0 note=53 velocity=0 time=0
note_on channel=0 note=55 velocity=100 time=0
note_on channel=0 note=60 velocity=0 time=0
```

Rysunek 2.4. Fragment protokołu MIDI

Wiadomości

Plik MIDI zawiera zestaw wiadomości przesyłanych w czasie rzeczywistym o każdej nucie w utworze. Dwie wiadomości, które są dla nas szczególnie istotne to:

— note_on, który sygnalizuje, aby rozpocząć grać nutę,
— note_off, który sygnalizuje, aby zakończyć grać nutę.

Dla przykładu wiadomość:

note_on channel 0 note 48 velocity 100 time 0

oznacza aby na kanele 0 zagrać dźwięk nr 48 z głośnością 100 w momencie 0 utworu. Nie informuje nas on jednak o długości trwania dźwięku. Aby zakończyć dźwięk, należy wysłać wiadomość:

note_off, channel 0, note 48, velocity 100, time 24.

Zwróćmy uwagę, że aby ustalić wartość nuty potrzebujemy odebrać dwie wiadomości. Różnica między parametrami *time*, informuje nas o długości nuty. W tym przypadku jest to 24. Co oznacza ćwierćnutę.

Rozdzielczość

Czas w MIDI jest reprezentowany jako liczba naturalna i jest on zależny od ustalonego tempa utworu. Standardowa rozdzielczość pliku MIDI to 24. Oznacza to, że jedna jednostka czasu odpowiada jednej dwudziestej czwartej jednego uderzenia.

Kanały

Plik MIDI posiada 16 kanałów numerowanych od 0 do 15. Każdy kanał odpowiada instrumentowi lub ścieżce. Kanał 9 jest kanałem zarezerwowanym na instrumenty perkusyjne.

Nuty

Nuty w formacie MIDI opisane są kolejnymi cyframi naturalnymi w przedziale od 0 do 127. Odpowiada to dźwiękom od C_0 do C_8. Dla przykładu nuta 69 odpowiada A_4, a nuta 47 odpowiada B_2.

Wyjątkiem są nuty z kanału dziewiątego, gdzie istnieją tylko nuty z zakresu od 35 do 81 i każda nuta odpowiada innemu elementowi perkusyjne np. 35 to stopa (*ang. kick*), a 37 to werbel *ang. snare*.

Głośność

Za głośność dźwięku odpowiada parametr *velocity*, który jest liczbą z przedziału od 0 do 127. Im większa jest wartość tym głośniej wybrzmi dźwięk.

Program

Program w kontekście standardu MIDI oznacza instrument który ma zagrać nuty. W standardzie GM (ang. General MIDI), jest 16 grup instrumentów a w każdej z nich znajduje się po 8 instrumentów. Są to pianina, chromatyczne perkusje, organy, gitary, basy, instrumenty smyczkowe, zestawy instrumentów, instrumenty dmuchane blaszane, instrumenty dmuchane drewniane, flety, syntezatory prowadzące, syntezatory uzupełniające, efekty syntetyczne, instrumenty etniczne, perkusjonalia i efekty dźwiękowe.

Ścieżka

Ścieżka (*ang. track*) grupuje nuty aby podzielić utwór muzyczny na różne instrumenty lub partie. Protokół MIDI pozwala, aby grać wiele ścieżek dźwię-

kowych jednocześnie. Wtedy mówimy o muzyce polifonicznej lub multiinstrumentalnej.

Rozdział 3

Projekt

W tym rozdziale opiszę szczegóły działania modelu, który przygotowałem. Najpierw przedstawię pomysł na model, opiszę szczegóły technicze dotyczące przygotowania danych uczących dla modelu. Natępnie zdefiniuję model i rozwinę opis modelu sequence-to-sequence. W tym rozdziale będę również zamieszczał fragmentu kodu napisanego w języku Python.

3.1. Koncepcja

Celem tej pracy, było wykonanie modelu, który przy użyciu głębokiego uczenia będzie w stanie generować krótkie klipy multiinstrumentalne. Zainspirował mnie sposób w jaki tworzy się muzykę w zespole. W przeciwieństwie do muzyki tworzonej przez jednego kompozytora, w zespole każda partia tworzona jest przez muzyka grającego na danym instrumencie. Przykładowy sposób tworzenia utworu w zespole, np. rockowym wygląda jak następuje. Jedna osoba tworzy (generuje) pierwszą partię muzyczną, np. partię na gitary. Ta partia została stworzona bez odniesienia do innych członków zespołu. Następnie taka partia zostaje przedstawiona zespołowi. Każdy z członków zespołu musi teraz stworzyć swoje partie w taki sposób, aby pasowały one muzycznie do pierwszej partii. W ten sposób powstają nam zależności między partiami, tworzącymi cały utwór.

Na podstawie tej idei postanowiłem opracować model składający się z wielu sieci neuronowych, każda z nich odpowiadać będzie jednej partii w utworze, muzykowi w zespole. Jedna z tych sieci będzie generatorem. Ta sieć powinna być skonstruowana w taki sposób, aby zainicjować partię muzyczną. Pozostałe będą dopasowywać swoje partie w taki sposób, aby pasowały pod partię wygenerowaną. Te sieci nazywać będę modelami akompaniującymi. Dzięki temu jesteśmy wstanie stworzyć model wielu sieci, w którym następna sieć będzie produkować swoje partie na podstawie tego, co wygenerowała poprzednia.

Kluczowe było zauważenie podobieństwa między językiem naturalnym oraz muzyką. Zarówno zdanie jak i partia muzyczna składa się z sekwencji elementów rozmieszczonych w czasie. Elementy te są zależne od długoterminowego kontekstu, oraz od tego jaki element był ustawiony wcześniej. Dla języka natu-

ralnego są to słowa, dla muzyki są to nuty i akordy. Dodatkowo pomyślałem, że różne instrumenty można porównać do różnych języków świata. Wtedy aby stworzyć melodię, np. basu, tak aby pasowała pod partię gitary, należy „przetłumaczy" język gitary na język basu. Do tłumaczeń języka naturalnego wykorzystuje się modele sequence-to-sequence, dlatego postanowiłem w modelu generowania muzyki wykorzystać właśnie tą architekturę. Dodatkowo modele sequence-to-sequence mają tę cechę, że liczba elementów sekwencji wejściowej może być inna niż liczba elementów sekwencji wyjściowej. Idealnie sprawdzi się w przypadku muzyki, ponieważ o długości trwania ścieżki muzycznej nie świadczy liczba nut tylko suma ich wartości.

3.2. Wstępne przygotowanie danych do treningu

3.2.1. Muzyczne „słowo"

Na potrzeby dostosowania danych muzycznych do koncepcji słów w zdaniu zakodowałem pojedyncze słowo muzyczne jako

((zbiór wysokości), długość)

W ten sposób byłem w stanie zakodować pojedyńcze nuty i akordy.

Akord C-dur składający się z dźwięków C, E i G o długości ósemki, zapisaliśmy w następujący sposób.

((60, 64, 67), 0.5)

W ten sposób jesteśmy w stanie kodować melodię w sekwencji słów muzycznych. Tak skonstruowane dane mają niestety swoje negatywne aspekty. Nie da się w ten sposób zapisać partii, w której zostaje grana nowa nuta gdy poprzednia jeszcze powinna brzmieć. Nasz zapis zakłada, że melodia jest grana element po elemencie i nowy element wymusza zakończenie poprzedniego. Nie przechowujemy również informacji o dynamice melodii (głośności). Rozszerzenie tego zapisu o informacje o głośności nie jest trudne i nie będzie wymagać przebudowy modelu, natomiast zwiększy liczbę możliwych „słów muzycznych" w słowniku i zwiększy złożoność obliczeniową. Zdecydowałem się na niewykorzystanie tych danych w generowaniu muzyki.

3.2.2. Konwersja MIDI na sekwencje słów muzycznych

Powszechny sposób przechowywania muzyki symbolicznej w formie cyfrowej to pliki *.mid lub *.midi które przechowują informację o potoku wiadomości protokołu MIDI. Aby odczytać wiadomości z plików MIDI wykorzystałem

bibliotekę `pretty_midi`, która zawiera wiele funkcji pozwalających na edycję plików MIDI.

Aby otworzyć pliki midi za pomocą biblioteki `pretty_midi`, należy skorzystać z poniższej składni.

```
>>> import pretty_midi as pm
>>> midi_path = 'example.mid'
>>> midi = pm.PrettyMIDI(midi_path)
>>> melody = midi.instruments[0]
>>> melody.notes

[Note(start=18.873909, end=19.186408, pitch=71, velocity=110),
 Note(start=19.132529, end=19.471968, pitch=76, velocity=114),
 Note(start=19.396538, end=19.768304, pitch=80, velocity=111),
 Note(start=19.655158, end=19.951494, pitch=81, velocity=105),
 Note(start=19.913779, end=20.226278, pitch=80, velocity=99),
 Note(start=20.172399, end=20.452571, pitch=76, velocity=119),
 Note(start=20.431020, end=20.624985, pitch=71, velocity=115),
 Note(start=20.689640, end=20.975200, pitch=69, velocity=114),
 ...]
```

Dzięki bibliotece `pretty_midi` plik midi został odczytany i przechowany w obiekcie `PrettyMIDI`. Ten obiekt posiada atrybut `instruments`, który jest listą ścieżek pliku MIDI. Obiekt ścieżki posiada atrybut `notes`, który jest listą nut tej ścieżki. Możemy zobaczyć, że biblioteka `pretty_midi` zamieniła potok sygnałów protokołu MIDI na konkretne nuty posiadające parametry *start*, *end*, *pitch* orac *velocity*. Aby otrzymać sekwencję danych w takim formacie w jakim potrzebujemy możemy zastosować na obiekcie `Instrument` poniższą funkcję.

```
def parse_pretty_midi_instrument(instrument, resolution,
 time_to_tick, key_offset):
    ''' arguments: a prettyMidi instrument object
        return: a custom SingleTrack object
    '''

    first_tick = None
    prev_tick = 0
    prev_note_lenth = 0
    max_rest_len = 4.0

    notes = defaultdict(lambda:[set(), set()])
    for note in instrument.notes:
        if first_tick == None:
            first_tick = 0

        tick = round_to_sixteenth_note(
      time_to_tick(note.start)/resolution)
```

```
        if prev_tick != None:
            act_tick = prev_tick + prev_note_lenth
            if act_tick < tick:
                rest_lenth = tick - act_tick
                while rest_lenth > max_rest_len:
                    notes[act_tick] = [{-1},{max_rest_len}]
                    act_tick += max_rest_len
                    rest_lenth -= max_rest_len
                notes[act_tick] = [{-1},{rest_lenth}]

        note_lenth = round_to_sixteenth_note(
      time_to_tick(note.end-note.start)/resolution)

        if -1 in notes[tick][0]:
            notes[tick] = [set(), set()]

        if instrument.is_drum:
            notes[tick][0].add(note.pitch)
        else:
            notes[tick][0].add(note.pitch+key_offset)

        notes[tick][1].add(note_lenth)

        prev_tick = tick
        prev_note_lenth = note_lenth

    notes = [(tuple(e[0]), max(e[1])) for e in notes.values()]

  if instrument.is_drum:
    name = 'Drums'
  else :
    pm.program_to_instrument_class(instrument.program)

  return SingleTrack(name,
              instrument.program,
              instrument.is_drum,
              Stream(first_tick, notes))
```

Powyższa funkcja w zamienia wartości absolutne czasu, na wartości względne o ustalonej rozdzielczości przez plik MIDI. Dodatkowo zmniejsza szczegółowość, i zaokrągla czas zagrania nuty po szesnastki. Gdy w tym samym momencie, czyli jeśli kilka nut posiada tą samą wartość start, zostają dodane do jednego słowa muzycznego aby utworzyć akord. Pauzy są kodowane jako -1. Dodatkowo jeśli pauza trwa dłużej niż takt wtedy zostaje podzielona na mniejsze części o długości `max_rest_len`. Funkcja zwraca obiekt `SingleTrack`, który jest obiektem stworzonym aby poza nutami, przechowywać inne istotne informacje na temat ścieżki, którą będą istotne w następnych częściach prze-

twarzania danych. Ostatecznie sekwencje słów muzycznych przechowane są w `notes`.

```
>>> resolution = midi.resolution
>>> time_to_tick = midi.time_to_tick
>>> instrument = melody

>>> single_track = parse_pretty_midi_instrument(instrument,
resolution, time_to_tick, key_offset=0)
>>> single_track.stream.notes

[((-1,), 4.0),
 ((-1,), 0.5),
 ((71,), 0.5),
 ((76,), 0.75),
 ((80,), 0.75),
 ((81,), 0.5),
 ((80,), 0.5),
 ((76,), 0.5),
 ((71,), 0.5),
 ((69,), 0.5),
 ((68,), 0.5),
 ((69,), 0.25),
 ...]
```

3.2.3. Inne aspekty przygotowania danych

Po odczytaniu danych i konwersji je do pożądanego formatu dane należy oczyścić. W mojej pracy zastosowałem kilka operacji, w celu zwiększenia muzycznego sensu danych.

Unormowanie skali

W muzyce istnieje pojęcie skali. Skala jest to zestaw nut, które dobrze ze sobą współgrają. Zostało to szerzej opisane w podrozdziale 2.1.3. W uczeniu maszynowym powoduje to realny problem, ponieważ piosenki wykorzystują różne skale i sieć neuronowa będzie preferować wybranie skali częściej używanej. Dodatkowo zmiana skali nie zmienia znacznie kontentu muzycznego utworu, tj. nawet po zmianie skali melodii człowiek dalej jest w stanie ją rozpoznać. Zmiana wysokości wszystkich nut bez zmiany ich względnych interwałów nazywana jest transpozycją. Aby rozwiązać ten problem zaleca się rozszerzenie danych do wszystkich możliwych skal. W mojej pracy wykorzystałam jednak inne rozwiązanie. Zamiast rozszerzać zbiór danych sprowadziłem wszystkie ścieżki muzyczne do jednej skali C. Dzięki temu model przyłoży większą uwagę na rozumienie wzajemnych relacji, zamiast uczyć się pojęcia tonacji skali [1].

Podział na takty

Długie listy muzycznych słów zostały podzielone na takty (*ang. bars*), o odpowiedniej długości, domyślnie o długości 4, co odpowiada czterem ćwierćnutom. Dzięki temu utwór muzyczny zostanie podzielony na mniejsze sekwencje. Sekwencje te będą posiadały różną liczbę elementów, ale będą tak samo długie, w kontekście muzycznym. Głównym celem takie zabiegu, jest zapewnienie muzycznego sensu sekwenjom. Takt jest naturalnym dla muzyki podziałem dłuższego partii na mniejsze.

3.2.4. Podział danych na dane wejściowe i wyjściowe

Na podstawie przetworzonych danych, należy przygotować dane wejściowe X i wyjściowe Y dla sieci neuronowych, aby przeprowadzić proces uczenia. W tym celu będziemy rozważać pary sekwencji (x, y), gdzie $x \in X$ i $y \in Y$. Każda sekwencja zawierać będzie omówione wcześniej słowa muzyczne. W przygotowanym przeze mnie modelu, występują dwa rodzaje sieci neuronowych, sieć generująca oraz sieć akompaniująca.

Przygotowanie danych dla generatora

Model generatywny będzie tworzył partie muzyczne, na podstawie poprzednich sekwencji tego samego instrumentu. Weźmy partię muzyczną G, która jest uporządkowaną listą elementów g w czasie. Każdy element g jest taktem składających się ze słów muzycznych.

$$G = [g_1, g_2, g_n],$$

gdzie n jest liczbą taktów w partii muzycznej. Pary (x_t, y_t) tworzymy według poniżej reguły

$$x_t = g_t$$
$$y_t = g_{t+1},$$

dla $t \in (1, n-1)$.

Dzięki takiemu zdefiniowaniu danych uczących generator będzie uczył się jak powinien wyglądać następny takt, na podstawie poprzedniego. W ten sposób będziemy w stanie wykorzystać model do generowania muzyki bez danych wejściowych. Wątek zostanie rozwinięty w dalszej części pracy.

Implementacja w Pythonie:

```
def get_data_seq2seq_melody(self, instrument_class,
                x_seq_len=4):
  '''return a list of bars with content for every track
  with given instrument class for melody generation
  x_seq_len and y_seq_len
```

```
  x previous sentence, y next sentence of the same melody line

  '''

  instrument_tracks =
         self.tracks_by_instrument[instrument_class]

  for track_index in instrument_tracks:
    bars = self.tracks[track_index].stream_to_bars()
    bars_indexes_with_content =
              get_bar_indexes_with_content(bars)
    bars_with_content =
        [bars[i] for i in get_bar_indexes_with_content(bars)]

    x_seq = []
    y_seq = []
    for i in range(len(bars_with_content)-x_seq_len-1):
      _x_seq =
      [note for bar in bars_with_content[i:i+x_seq_len]
       for note in bar]
      _y_bar = bars_with_content[i+x_seq_len]
      x_seq.append(_x_seq)
      y_seq.append(_y_bar)

    return x_seq, y_seq
```

Przygotowanie danych dla akompaniamentu

Model akompaniujący natomiast, będzie na podstawie partii instrumentu tworzyć partię na nowy instrument. Niech G, B będą sekcjami muzycznymi różnych instrumentów tej samej długości oraz niech

$$G = [g_1, g_2, ..., g_k],$$
$$B = [b_1, b_2, ..., b_k],$$

wówczas pary dla zbioru uczącego tworzymy w następujący sposób

$$x_t = g_t,$$
$$y_t = b_t,$$

dla $t \in (1, k)$. Istotne jest aby każdy element ze zbioru taktów partii B był rzeczywistą aranżacją tego instrumentu dla taktów partii G oraz aby między elementami g_t oraz b_t była muzyczna relacja.

Implementacja przedstawionej techniki w Pythonie.

```
def get_data_seq2seq_arrangment(self, x_instrument,
   y_instrument, bars_in_seq=4):
```

```
    '''this method is returning a sequences of given length
     by rolling this lists of x and y for arrangement generation
    x and y has the same bar length, and represent the
    same musical phrase played mb different instruments (tracks)

    '''
    x_seq = []
    y_seq = []
    x_bars, y_bars =
    self.get_common_bars_for_every_possible_pair(
      x_instrument, y_instrument)

    for i in range(len(x_bars) - bars_in_seq + 1):
      x_seq_to_add =
      [note for bar in
        x_bars[i:i+bars_in_seq] for note in bar ]
      y_seq_to_add =
        [note for bar in
      y_bars[i:i+bars_in_seq] for note in bar ]
      x_seq.append(x_seq_to_add)
      y_seq.append(y_seq_to_add)

    return x_seq, y_seq
```

3.2.5. Inne aspekty przygotowania zbioru uczącego

Oczyszczenie danych

W przygotowaniu danych dla modelu ważne jest, aby dostarczone dane były jak najlepszej jakości. W tym celu usunąłem powtarzające się pary (x, y) oraz usunąłem takty, które nie zawierały muzycznego kontentu.

Wybór programu dla instrumentu

Podczas etapu ekstrakcji danych z plików MIDI poza informacjami o muzyce, zapamiętuję również informacje o programie partii muzycznej. Każda ścieżka MIDI przechowuje informacje o instrumencie (brzmieniu) danej partii. Istnieje 128 różnych programów, dla zmniejszenia szczególności na potrzeby modelu wyróżniam 16 instrumentów zgodnie z grupą do jakiej należą w podziale General MIDI. Dla każdej z grup, sprawdzam jaki program został najczęściej wykorzystywany i zapisuję go na przyszłość, aby móc wygenerowanej muzyce przy kompilacji do MIDI zdefiniować brzmienie instrumentu zgodnie z najczęściej wykorzystywanym w zbiorze MIDI, który został wykorzystany do stworzenia zbioru uczącego dla modelu.

Melodia

Dodatkowym elementem procesu ekstrakcji danych jest znalezienie ścieżek melodii przewodnich w plikach MIDI. Ścieżki tego typu zamiast być oznaczone nazwą grupy instrumentów do której należą oznaczone są nazwą `Melody`. Melodia jest kategorią ścieżek z podziału ze względu na rolę partii w utworze, zamiast na instrument. Istnieją też inne role instrumentów, jednak często rola jest w pewnym sensie definiowana przez instrument. Nie jest to zasada, bardziej prawidłowość w muzyce. Uznałem że wydobycie tej informacji na temat ścieżki nada więcej muzycznego sensu danym. Aby sprawdzić czy dana ścieżka jest melodią zastosowałem poniższą funkcję:

```
def check_if_melody(self):
'''checks if Track object could be a melody

  it checks if percentage of single notes in
  Track.stream.notes is higher than threshold
  of 90 and there is at least 2 notes in bar per average

'''
events = None
single_notes = None
content_lenth = None

for note in self.stream.notes:
  if self.name not in ['Bass','Drums']:
    events = 0
    content_lenth = 0
    single_notes = 0
    if note[0][0] != -1: # if note is not a rest
      events += 1
      content_lenth += note[1]
      if len(note[0]) == 1: # if note is a single note
        single_notes += 1

if events != None:
  if events == 0 or content_lenth == 0:
    return False
  else:
    single_notes_rate = single_notes/events
    density_rate = events/content_lenth
    if single_notes_rate >= 0.9 and density_rate < 2:
      self.name = 'Melody'
      return True
    else:
      return False
else:
  return False
```

Funkcja sprawdza liczbę pojedynczych nut i akordów w ścieżce oraz zagęszczenie nut w takcie. Jeśli jest więcej niż 90% pojedynczych nut w ścieżce oraz jest średnio więcej nut w takcie niż dwie wtedy uznaję, że partia instrumentalna utworu jest melodią.

3.3. Definicja modelu

Wszystkie modele sieci neuonowych zastosowane w tej pracy, zostały napisane z wykorzystaniem środowiska Keras. Keras jest to środowisko wyższego poziomu, służące do tworzenia modelu głębokiego uczenia.

Model sequence-to-sequence, jest to model składający się z dwóch mniejszych sieci neuronowych, enkodera i dekodera. Dodatkowo inaczej definiuje się model aby go uczyć, a inaczej aby dokowyać predycji.

3.3.1. Model w trybie uczenia

Zdefiniowanie warstw enkodera

Zadaniem enkodera jest wydobycie z przetwarzanej sekwekcji kontekstu, skompresowanej informacji o danych. W tym celu zastosowana została jedna warstwa wejściowa o rozmiarze słownika wejściowego, oraz warstwa LSTM. Definiowane warstwy są atrybutami całej klasy modelu, dlatego w prezentowanym kodzie występują przedrostek `self.`.

```
self.encoder_inputs =
  Input(shape=(None, self.transformer.x_vocab_size))

self.encoder =
  LSTM(latent_dim, return_state=True, dropout=enc_dropout)

self.encoder_outputs, self.state_h, self.state_c =
  self.encoder(self.encoder_inputs)

self.encoder_states = [self.state_h, self.state_c]
```

Ostatecznie jako dane wyjsciowe otrzymujemy wektor danych ukrytych `encoder_outputs` o wymiarze równym ustalonej zmiennej `latent_dim`. Dodatkowo uzywając parametru `return_state=True`, warstwa LSTM zwraca wektory stanu komórek h oraz c. Te wektory są pamięcią wartswy LSTM i posłużą jako stan wejściowy dla modelu dekodera.

Oba wekory zostały zapisane do jednej zmiennej `self.encoder_states`.

Zdefiniowanie warstw dekodera

W części dekodera sieci sequance-to-sequence, definiujemy warstwę wejściową o rozmiarze słownika wynikowego i jedną warstwę LSTM o stanie komórek h i c w enkodera. Na końcu została zdefiniowana prosta warstwa typu `Dense` z funkcją aktywacji `softmax` w celu rozszerzenia wymiaru do wielkości słownika wyjściowego.

```
self.decoder_inputs =
  Input(shape=(None, self.transformer.y_vocab_size))

self.decoder_lstm =
  LSTM(latent_dim,
    return_sequences=True,
    return_state=True,
    dropout=dec_dropout)

self.decoder_outputs, _, _ =
  self.decoder_lstm(self.decoder_inputs,
    initial_state=self.encoder_states)

self.decoder_dense =
  Dense(self.transformer.y_vocab_size, activation='softmax')

self.decoder_outputs = self.decoder_dense(self.decoder_outputs)
```

Jako wynik propagacji modelu otrzymujemy wektor o rozmiarze słownika wyjściowego `self.transformer.y_vocab_size`, który przedstawia zakodowany element sekwencji.

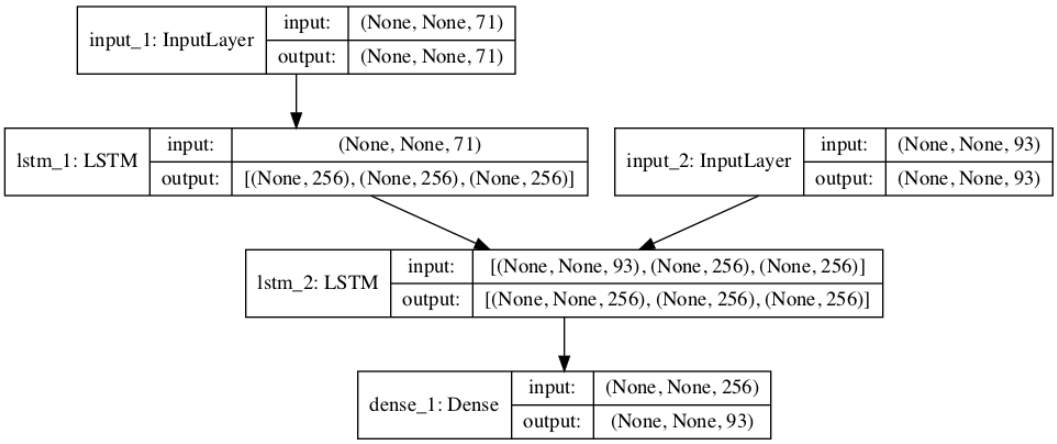
Definicja modelu w trybie uczenia

Ostatnim krokiem, aby otrzymać model gotowy do procesu uczenia należy połączyć ze sobą enkoder i dekoder, tworząc pełny model. Po jego skompilowaniu model mozna przedstawić jako graf tak jak pokazano to na rysunku 3.1

```
self.train_model =
  Model([self.encoder_inputs, self.decoder_inputs],
    self.decoder_outputs)
self.train_model.compile(optimizer='rmsprop',
            loss='categorical_crossentropy')
```

Możemy zauważyć, że w procesie uczenia musimy zaprezentować modelowi dwa wektory danych wejsciowych oraz jednego wektora danych wyjściowych. W pracy zostały one opisane jako:

[1] machinelearningmastery.com 16 czerwca 2020 06:24



Rysunek 3.1. Model seq2seq w trybie uczenia[1]

— `encoder_input_data`, zawierające sekwencję elementów wejściowych. Te dane będą zasilać encoder aby na ich podstawie został wygenerowany kontekst.
— `decoder_input_data`, zawierające sekwencje elementów wyjściowych opóźnione o jeden element w czasie. Te dane będą zasilały dekoder.
— `decoder_target_data`, zawierające sekwencje elementów wyjściowych, które bedą celem.

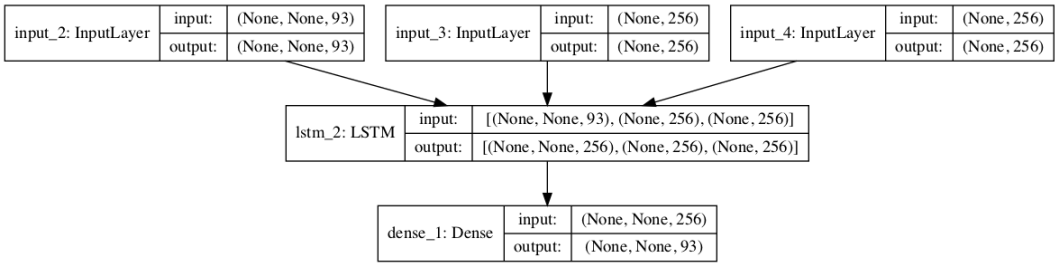
Jest to ciekawe, że musimy dostarczyć do modelu dane, które tak naprawdę chcemy otrzymać. Jednak te dane są przesunięte o jeden element w sekwencji do przodu i pierwszy element tej sekwencji nie zawiera treści. Dekoder uczy się jak na podstawie aktualnego stanu enkodera oraz aktualnego elementu sekwencji wyjściowej jaki powinien być następny element w sekwecji docelowej. Warto też zauważyć, że z definicji modelu nie wynika aby model uczył się przetwarzania całych sekwencji, tylko każdego elementu sekwencji z osobna.

3.3.2. Model w trybie wnioskowania

To co czyni model seq2seq zdolnym do przetwarzania całych sekwencji jest sposób zdefiniowania metody wnioskowania (*ang. inference*). Na tym etapie model w sposób rekurencyjny będzie przewidywał następny element sekwencji, na podstawie tego co przewidział wcześniej oraz aktualizowanych wektorów stanu warstwy LSTM. Musimy w tym celu zdefiniować dekoder tak, aby mógł przyjąć swoje wyjście jako nowe wejście. Dlatego definicja modelu wtedy wygląda nastepująco, a graf przedstawiający dekoder w trybie uczenia zostal przedstawiony na rysunku 3.2.

```
self.decoder_state_input_h = Input(shape=(self.latent_dim,))
```

```
self.decoder_state_input_c = Input(shape=(self.latent_dim,))
self.decoder_states_inputs = [self.decoder_state_input_h,
                self.decoder_state_input_c]

self.decoder_outputs, self.state_h, self.state_c =
  self.decoder_lstm(
      self.decoder_inputs,
      initial_state = self.decoder_states_inputs
      )

self.decoder_states = [self.state_h, self.state_c]

self.decoder_outputs = self.decoder_dense(self.decoder_outputs)

self.decoder_model = Model(
  [self.decoder_inputs] + self.decoder_states_inputs,
  [self.decoder_outputs] + self.decoder_states)
```



Rysunek 3.2. Model seq2seq w trybie uczenia[2]

Dzięki takiej definicji, jesteśmy w stanie w pętli przewidywać następne elementy do momentu osiągnięcia warunku stopu. Implementacja tego procesu wygląda tak:

```
def predict(self, input_seq=None, mode=None):

  if not self.has_predict_model:
    self.init_predict_model()
    self.has_predict_model = True

  if mode == 'generate':
    h = np.random.rand(1,self.latent_dim)*2 - 1
    c = np.random.rand(1,self.latent_dim)*2 - 1
    states_value = [h, c]
  else:
    states_value = self.encoder_model.predict(input_seq)
```

[2] machinelearningmastery.com 16 czerwca 2020 06:24

```
target_seq =
  np.zeros((1, 1, self.transformer.y_vocab_size)

target_seq[0, 0, self.transformer.y_transform_dict['<GO>']] =
   1

stop_condition = False
decoded_sentence = []

while not stop_condition:

  output_tokens, h, c = self.decoder_model.predict(
    [target_seq] + states_value)

  sampled_token_index =
    np.argmax(output_tokens[0, -1, :])
  sampled_char =
    self.transformer.y_reverse_dict[sampled_token_index]
  decoded_sentence.append(sampled_char)

  if (sampled_char == '<EOS>' or
    len(decoded_sentence) > self.transformer.y_max_seq_length):
    stop_condition = True

  target_seq = np.zeros((1, 1, self.transformer.y_vocab_size))
  target_seq[0, 0, sampled_token_index] = 1.

  states_value = [h, c]

return decoded_sentence
```

Encoder w tym trybie ma za zadanie tylko dostarczyć do dekodera wektory kontekstu dla zainicjowania procesu. Dekoder na podstawie wektorów pamięci h i c przewidzi pierwszy element sekwencji i tym samym zainicjuje proces wnioskowania.

W tym miejscu mamy kontrolę twórczą nad dekoderem. Aby zainicjować reakcję wystarczą wektory pamięci, możemy wtedy podać je w dowolny sposób i zobaczyć co wygeneruje dekoder. W mojej pracy próbowałem dwóch sposób na zainicjowanie procesu predykcji z wektorów losowych. Pierwszy polega na wygenerowaniu losowej sekwencji dla encodera i nazywam tę metodę `from_seq`. Druga metoda polega na bezpośrednim definiowaniu stanu h i c `states_value` w sposób losowy tak aby wartości były z przedziału [-1, 1].

3.4. Transformacja danych dla modelu

Na podstawie definicji modelu wiemy, że należy przygotować trzy zestawy danych aby móc wytrenowac model. Dwa zestawy danych wejściowych i jeden zestaw danych wyjściowych. Musimy również zakodować sekwencje elementów w taki sposób aby można było wykonywac na nich obliczenia.

3.4.1. Enkodowanie one-hot

Kazdy element sekwencji jest osobną kategorią, dlatego dane należy potraktować tak jak dane kategorycznym. Wykorzystamy w tym celu enkodowanie one-hot (*one-hot encoding*) Enkodowanie One-Hot jest wykorzystywane w uczeniu maszynowym aby nadać liczbową wartość danych kategorycznych. Polega ona stworzeniu słownika, w którym każde słowo otrzyma swój unikatowy identyfikator, następnie zostanie utworzony wektor o wymiarze słów w słowniku, gdzie na pozycji odpowiadającej indeksowi słowa będzie wartość 1 a na pozostałych będzie wartość zero [2].

Przykład działania One-Hot Encoding

Weźmy sekwencję liter w słowie MATEMATYKA. Znajdźmy unikatowe elementy tej sekwencji, oraz nadajmy im unikatowy identyfikator. Kolejność nie ma znaczenia.

M - 0, A - 1, T - 2, E - 3, Y - 4, K - 5.

Kodując słowo MATEMATYKA, otrzymalibyśmy macierz

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix}$$

3.4.2. Słownik

Aby zakodować sekwencje musimy utowrzyć słownik, w którym każdy element będzie posiadał unikatowy indeks.

```
x_vocab = set([note for seq in x_train for note in seq])
y_vocab = set([note for seq in y_train for note in seq])
```

Do słownika dodajemy dwa dodatkowe elementy, które opiszę w następnym rozdziale.

```
self.x_vocab = sorted(list(x_vocab))
self.y_vocab = ['<GO>','<EOS>']
self.y_vocab.extend(sorted(list(y_vocab)))
```

Następnie tworzymy obiekty typu `dictionary`, aby zmapować elementy sekwncji na liczby. W tym momencie tworzymy również słowniki odwrotne, aby móc później otrzymane wyniki zamienić na z powrotem na muzyczne słowa.

```
self.x_transform_dict = dict(
[(char, i) for i, char in enumerate(self.x_vocab)])
self.y_transform_dict = dict(
[(char, i) for i, char in enumerate(self.y_vocab)])
self.x_reverse_dict = dict(
(i, char) for char, i in self.x_transform_dict.items())
self.y_reverse_dict = dict(
(i, char) for char, i in self.y_transform_dict.items())
```

3.4.3. Elementy specjalne

Dodaje się dwóch elementów specjalnych. Te elementy mają za zadanie oznaczenie początku i końca sekwencji. Token początku senwekcji zazwyczaj oznaczany jest jako `<SOS>` (*ang. start of sequance*). Można spotkać również inne oznaczenia,np. `<GO>` lub `<s>`. Ten element ma na celu wypełnienie przestrzenie pierwszego pustego miejsca w danych wejściowych dla dekodera. Token końca sekwecji oznaczany jako `<EOS>` (*ang. start of sequance*) lub `</s>`. Ten element jest wykorzystywany aby wywołać warunek stopu w procesie predykcji opisanym w rozdziale 3.3.2.

```
_y_train = []
for i, seq in enumerate(y_train):
  _y_train.append([])
  _y_train[i].append('<GO>')
  for note in seq:
      _y_train[i].append(note)
  _y_train[i].append('<EOS>')
```

3.4.4. Zakodowanie sekwencji

Ostatecznie tworzymy tensory o rozmiarach n x s x p, gdzie n jest liczbą obserwacji `len(x_train)`, s jest maksymalną liczbą elementów w sekwencji `self.x_max_seq_length` a p jest rozmiarem słownika, czyli liczbą cech `self.x_vocab_size`.

```
self.x_max_seq_length =
  max([len(seq) for seq in x_train])
```

```
self.y_max_seq_length =
  max([len(seq) for seq in y_train])

encoder_input_data = np.zeros(
  (len(x_train),
  self.x_max_seq_length,
  self.x_vocab_size),
  dtype='float32')

decoder_input_data = np.zeros(
  (len(x_train),
  self.y_max_seq_length,
  self.y_vocab_size),
  dtype='float32')

decoder_target_data = np.zeros(
  (len(x_train),
  self.y_max_seq_length,
  self.y_vocab_size),
  dtype='float32')

for i, (x_train, y_train) in enumerate(zip(x_train, y_train)):
  for t, char in enumerate(x_train):
    encoder_input_data[i, t,  self.x_transform_dict[char]] = 1.
  for t, char in enumerate(y_train):
    decoder_input_data[i, t,  self.y_transform_dict[char]] = 1.
    if t > 0:
      decoder_target_data[i, t - 1,  self.y_transform_dict[char]] = 1.
```

W ten sposób otrzymujemy trzy zestawy danych potrzebne do przeprowadzenia procesu uczenia modelu.

Warto zwrócić uwagę, że przed opisaną transformacją sekwencje były różnej długości a po niej, rozmiar sekwencji został rozszerzony do rozmiaru sekwencji posiadającej najwięcej elementów. Taki zabieg był niezbędny ponieważ rozmiar sieci neuronowej jest stały dla wszystkich prób ze zbioru uczacego. Nie wpływa to jednak na same sekwencję ponieważ podczas predykcję kończymy w momencie wygenerowania tokenu `<EOS>`.

3.5. Ekperyment

W tej części pokażę jak wykorzystać oprogramowanie, które stworzyłem aby wygenerować muzykę na przykładzie. Omówię cały proces, następnie zademonstruję wyniki.

3.5.1. Oprogramowanie

Stworzone przeze mnie oprogramowanie, napisane w języku Python, składa się z 2 bibiotek oraz 3 skryptów.

— `midi_procesing.py` - zawiera funkcję potrzebne do pracy w plikami midi.
— `model.py` - zawiera definicję modelu sieci neuronowej
— `extract.py` - służy do wydobycia w plików midi zbioru danych w postacji sekwencji.
— `train.py` - wykonując ten skrypt wykorzystujemy wygenerowane dane, aby wytrenować zestaw sieci neuronych.
— `generate.py` - wykorzystuje wytrenowane modele aby wygenerować ostatecznie plik midi.

Fragmenty bilbiotek `midi_procesing.py` i `model.py` zostały szczegółowo omówione w rozdziałach 3.2 i 3.3. Natomiast skrypty `extract.py`, `train.py` i `generate.py` zostały napisane aby ułatwić proces generowania oraz zapewniają powtarzalność i skalowalność prowadzonych ekperymentów.

3.5.2. Zbiór danych

W omawianym przykładzie wykorzystałem zbiór wybranych utworów midi zespołu The Offstring. Został on skompletowany ze źródeł dostępnych na stronie internetowej https://www.midiworld.com/. Składa się z 7 utworów.

— The Offspring - All I Want.mid
— The Offspring - Change the World.mid
— The Offspring - Nitro.mid
— The Offspring - Original Prankster.mid
— The Offspring - Self Esteem.mid
— The Offspring - The Kids Arent Alright.mid
— The Offspring - Why Dont You Get a Job.mid

3.5.3. Wydobycie danych

Aby wydobyć dane z plików midi wykorzystamy skypt `extract.py`. Można uzyć flagi `-a`, aby najpierw zapoznać się z zawartością muzyczną zbioru plików midi.

```
>>> python extract.py offspring -a
1098 of Drums
1037 of Guitar
704 of Melody
528 of Bass
1 of Organ
```

Dzięki temu możemy zaobserować że w procesowanym zbiorze danych jest 1098 taktów perkusji, 1037 taktów gitar, 704 melodii, 528 basu oraz 1 takt organ.

Na tym etapie musimy zdecydować, który instrment będzie generatorem oraz jakie będą zalezności między partiami w zespole sieci neuronowych.

W tym przykładzie, zdecydowałem że gitara będzie generowana na podstawie losowego wektora, a melodia, bass oraz perkusja będe tworzone na podstawie gitary.

Uruchamiając skrypt `extract.py` bez flagi `-a` zdefiniujemy omawiany przepływ za pomocą prostego konfiguratora.

```
>>> python extract.py offspring
Please specify number of instruments
4
Please specify a workflow step
>>> Guitar m
Please specify a workflow step
>>> Melody Guitar a
Please specify a workflow step
>>> Drums Guitar a
Please specify a workflow step
>>> Bass Guitar a

Exporting: 'Guitar'
Exporting: ('Guitar', 'Melody')
Exporting: ('Guitar', 'Drums')
Exporting: ('Guitar', 'Bass')
Done.
```

Po tym etapie zostały utworzony pliki zawierające oczyszczone pary sekwencji dla każdej sieci neuronowej. Zostało wygenerowanych

— 263 próby dla modelu gitary,
— 622 próby par melodii i gitary,
— 948 prób par perkusji i gitary,
— 385 prób par basu i gitary.

Na podstawie takiego zbioru danych, w nastepnym kroku zostaną wytrenowane czrety sieci neuronowe, po jednej dla każdego intrumentu.

3.6. Trenowanie modelu

Uzywając skypt `train.py` możemy w prosty sposób wytrenować wszystkie modele, a wagi zapiszą i będzie można je wykorzystać w celu generowania, lub w celu dalszego uczenia.

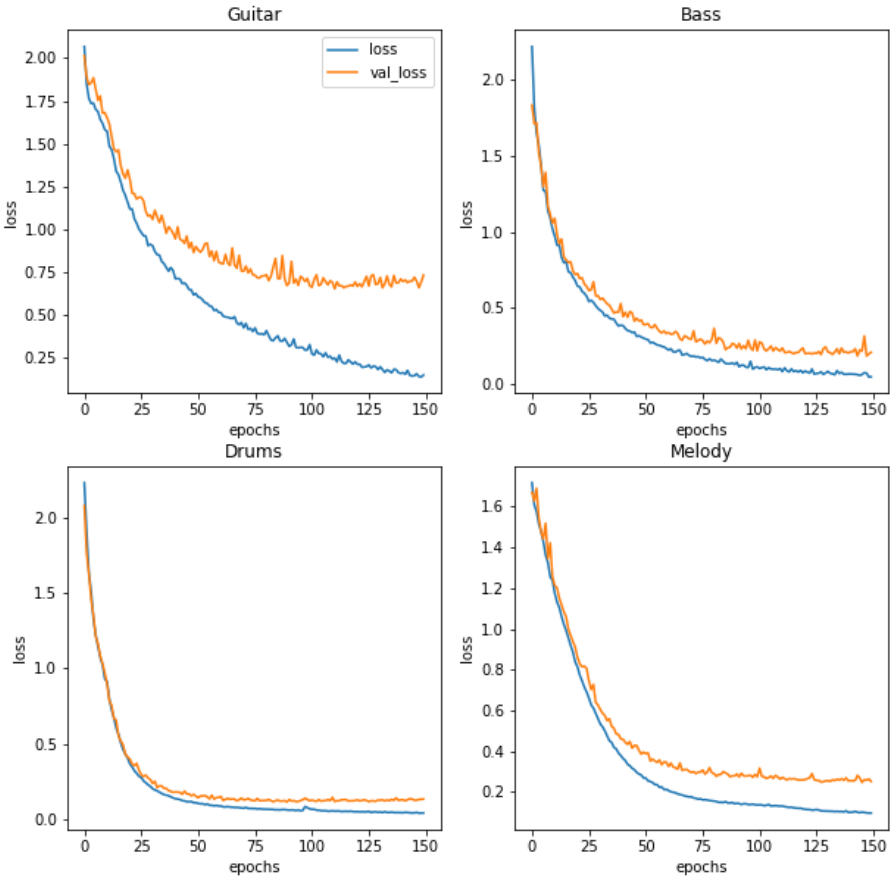
```
>>> python train.py offspring --e 1

Using TensorFlow backend.

Training: Guitar
Train on 210 samples, validate on 53 samples
Epoch 1/1
210/210 [==============================]
  - 2s 10ms/step - loss: 2.1553 - val_loss: 2.0384
Training: Melody
Train on 497 samples, validate on 125 samples
Epoch 1/1
497/497 [==============================]
  - 6s 11ms/step - loss: 1.7045 - val_loss: 1.6693
Training: Drums
Train on 758 samples, validate on 190 samples
Epoch 1/1
758/758 [==============================]
  - 9s 13ms/step - loss: 2.2218 - val_loss: 2.1255
Training: Bass
Train on 308 samples, validate on 77 samples
Epoch 1/1
308/308 [==============================]
  - 4s 12ms/step - loss: 2.2721 - val_loss: 1.7813
```

Na potrzeby badań trenowałem i generowałem klipy muzyczne dla epok 1, 10, 25, 50, 75, 100, 150. Dzięki temu mogę porównać ze sobą poszczególne etapy treningu sieci neuronowych. Wykresy funkcji kosztów dla zbioru uczącego oraz testowego zaprezentowane zostały na rysunku 3.3.

3.7. Generowanie muzyki przy pomocy wytrenowanych modeli

Gdy zdefiniowane modele zostaną wytrenowane możemy wykorzystać skrypt `generate.py`, wtedy generująca sieć neuonowa zostanie zasilona losowym wekorem aby wygenerować partię. W tym przykładzie gitary a wygenerowana partia posłuży jako dane wejściowe na pozostałych modeli. Ostatecznie otrzymane sekwencje zostaną skompilowane do pliku MIDI. W tym momencie zostają wykorzystane informacje o programach dla każdego z intrmentów, a tempo utworu domyślnie ustawione jest na 120 BPM. Możemy również zdecydować, czy zasilenie dekodera modelu generującego odbędzie się za pomocą losowej sekwencji elementów ze słownika (*from_seq*), czy losowy wektor zasili bezpośrednio stany wewnętrzne dekodera h i c (*from_state*).

```
>>> python generate.py offspring --i 10 --m from_state
```



Rysunek 3.3. Wartości kosztu dla poszczególnych modeli.

```
Using TensorFlow backend.
Loading models...
Generating music...
Done.
```

Parametr `--i` służy do okreslenia liczby wygenerowanych utworów a parametr `--m` pozwala zdefiniować metodę generowania, omówioną wyżej.

Przekazywanie wygenerowanych patrii to odpowiednich modeli wykonywane jest przy pomocy poniższego fragmenu kodu.

```
  notes = dict()
```

```
    for instrument, (model, program, generator) in band.items():
        if generator == None:
            notes[instrument] = model.develop(mode=MODE)
        else:
            input_data = seq_to_numpy(notes[generator],
                                      model.transformer.x_max_seq_length,
                                      model.transformer.x_vocab_size,
                                      model.transformer.x_transform_dict)
            notes[instrument] = model.predict(input_data)[:-1]
```

Słownik `band` przechowuje dane dotyczące całego zespołu modeli. Zmienna `intrument` jest nazwą intrumentu, `model` przechowuje obiekt modelu sequance-to-sequance, `program` jest liczbą naturalną odpowiającą programowi MIDI z kolekcji GM a `generator` to nazwa intrumentu na podstawie którego powinna zostać wygenerowana kolejna partia intrumentalna.

Kompilacja do MIDI zachodzi z wykorzystaniem klas biblioteki `midi_processing.py`, które wspierają format sekwencji słów muzycznych omówionych w rozdziale 3.2.1.

```
generated_midi = MultiTrack()
for instrument, (model, program, generator) in band.items():
  if instrument == 'Drums':
    is_drums = True
  else:
    is_drums = False

  stream = Stream(first_tick=0, notes=notes[instrument])
  track = SingleTrack(name=instrument,
            program=program,
            is_drum=is_drums,
            stream=stream)
  generated_midi.tracks.append(track)

generated_midi.save(save_path)
```

W ten sposób pomyślnie zostały wygenerowane fragmenty muzyczne przy pomocy głębokiego uczenia.

3.8. Wyniki

3.9. Wnioski

Podsumowanie

Ostateczne wnioski, czy muzyka generowana komputerowa da się lubić? Czy to pozytywnie wpłynie na przemysł muzyczny? Tak i nie. Może służyć jako inspiracja dla muzyków, proces wspierający. Z drugiej strony może obniży koszty produkowania muzyki pop, która i tak jest już bardzo powtarzalna. Czy sieci neuronowe nauczą się produkować Hity?

Bibliografia

[1] Briot, J.P., Hadjeres, G., Pachet, F.D. (2019): *Deep Learning Techniques for Music Generation - A Survey. arXiv:1709.01620v3*

[2] DeepAI (2019): *One Hot Encoding*

[3] Géron, A. (2019): *Hands-on machine learning with scikit-learn, keras and TensorFlow. O'Reilly.*

[4] Goodfellow, I., Bengio, Y., Courville, A. (2016): *Deep Learning. MIT Press.*

[5] Kostadinov, S. (2019): *Understanding Encoder-Decoder Sequence to Sequence Model*

[6] Krysicki, W., Włodarski, L. (1999): *Analiza matematyczna w zadaniach, PWN.*

[7] Sobczyk, M. (2006): *Statystyka. UMCS.*

[8] Swinney, A. (2020): *What is a tempo marking?*

[9] Zocca, V., Spacagna, G., Slater, D., Roelants, P. (2018): *Deep Learning. Uczenie głębokie z językiem Python. Helion.*

[10] Brownlee, J. (2017): *How to Develop a Seq2Seq Model for Neural Machine Translation in Keras, machinelearningmastery.com*