Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

KIERUNEK: Elektronika (EKA)

PRACA DYPLOMOWA INŻYNIERSKA

TYTUŁ PRACY:

Projekt i implementacja wybranych algorytmów sztucznej inteligencji w procesie rekonstrukcji nagrań fonicznych

AUTOR:

Przemysław Szczotka

PROMOTOR:

dr inż. Maciej Walczyński

WROCŁAW 2023

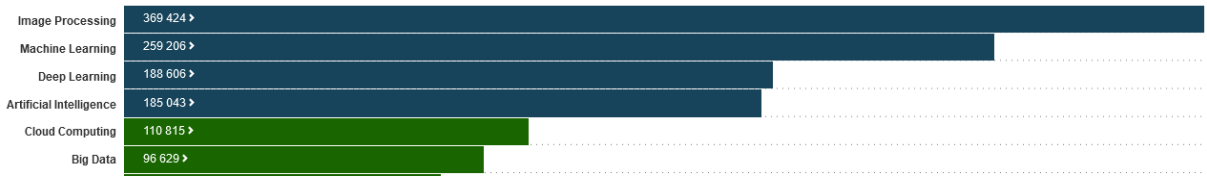
Za inspirację, wyrozumiałość oraz wsparcie w relacji niniejszej pracy pragnęzłożyć serdeczne podziękowania mojemu promotorowi dr inż. Maciejowi Walczyńskiemu

Spis treści

Rozdział 1

Wstęp

Od wielu lat sztuczna inteligencja (ang. *Artificial Intelligence*) oraz uczenie maszynowe (ang. *Machine Learning*), w tym również głębokie ucznie maszynowe (ang. *Deep Learning*), są cieszącymi się popularnością obszarami nauk związanych z techniką informatyczną. W ostatnich latach rozwój wyżej wymienionych dziedzin gwałtownie przyspieszył, dzięki spopularyzowaniu narzędzi wykorzystujących algorytmy generatywnej sztucznej inteligencji, co potwierdzają statystyki przedstawione na rysunku 1.1 udostępnione przez IEEE (Instytut Inżynierów Elektryków i Elektroników, ang. *Institute of Electrical and Electronics Engineers*) w bibliotece IEEE Xplore.



Rysunek 1.1 Najczęściej wyszukiwane wyrażenia, dane z dnia 12.12.2023 [3]

Dzisiejszy rozwój technologii oraz ilości generowanych danych spowodowało, iż sztuczna inteligencja oraz głębokie uczenie maszynowe, jest wykorzystywane w wielu obszarach, takich jak przetwarzanie obrazu, rozpoznawanie mowy, generowanie treści oraz rekonstrukcja danych w tym sygnałów fonicznych, którym poświęcona jest niniejsza praca. Szczególnym przypadkiem który zostanie omówiony w tej pracy są neuronowe sieci konwolucyjno-rekurencyjne dwukierunkowe (BCRNN, ang. *Bidirectional Convolutional Recurrent Neural Networks*), które zostaną omówione w rozdziale 4.4.

Celem pracy było zaimplementowanie algorytmów sztucznej inteligencji w procesie rekonstrukcji nagrań fonicznych. Projekt został oparty o architekturę BCRNN. Projekt jest pracą eksperymentalną.

W kolejnym rozdziale zostały opisane zagadnienia teoretyczne , które maja na celu dostarczyć wymaganą wiedzę do zrozumienia dalszych części pracy. W rozdziale trzecim została przedstawiona struktura bazy próbek audio oraz proces jest przetwarzania. W rozdziale czwartym przedstawiona została implementacja modelu wraz z informacjami dotyczącymi warstw użytych w przetwarzaniu danych. Dodatkowo na koniec czwartego rozdziału opisany został proces inwersji spektrogramu na format audio. Piąty rozdział składał się z podsumowania wyników.

Rozdział 2

Zagadnienia teoretyczne

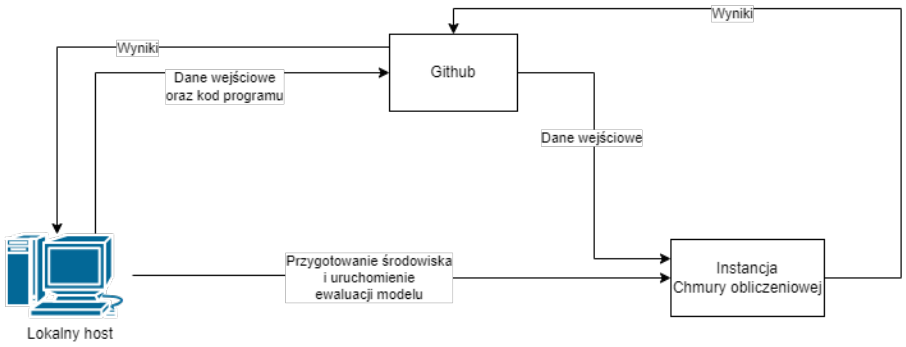
Projekt modelu pozwala na przetworzenie plików z zapisanym sygnałem fonicznym w formacie WAV (ang. *waveform audio format*), format został opisany w rozdziale 3.2 poświęconemu przygotowaniu bazy do ewaluacji modelu, przez sieć neuronową w celu zrekonstruowaniu defektów, a następnie otrzymanie wyjściowego pliku po przetworzeniu przez model sieci neuronowej.

W niniejszym rozdziale opisane zostały zagadnienia związane z przygotowaniem środowiska do implementacji modelu sieci neuronowej oraz zagadnienia związane z budową i rodzajami sieci neuronowych.

2.1 Środowisko sprzętowe

Proces ewaluacji modelu sieci neuronowej został przeprowadzony na instancji E2 dostarczanej przez chmury obliczeniowe Google (Google Cloud), które oferuje wirtualne środowiska obliczeniowe o konfigurowalnych parametrach [2]. Za pomocą usługi Compute Engine (infrastruktura do uruchamiania maszyn wirtualnych z wybranym systemem[2]) wybrano maszynę e2-highmem-8 z obrazem systemu Linux. Dane były dostarczane do oraz z chmurą obliczeniową przez platformę Github, natomiast przygotowanie środowiska oraz uruchomienie ewaluacji modelu wykonywane było poprzez standard komunikacyjny SSH (ang. *Secure Shell*) za pomocą programu Visual Studio Code Który został opisany w podrozdziale 2.2. .

Standard komunikacyjny ssh jest standardem protokołów komunikacyjnych używanych w architekturze klient-serwer/serwer-klient [13].



Rysunek 2.1 Schemat blokowy połaczenia z chmurą obliczeniową

Wykorzystanie chmury obliczeniowej e2-highmem-8 umożliwiło wydajniejszą ewalu-

ację modelu oraz powiększenie bazy próbek wykorzystanych do treningu.

2.2 Środowisko programistyczne

Aplikacja została napisana z wykorzystaniem języka obiektowego Python w wersji 3.9.2. Python jest językiem programowania ogólnego przeznaczenia, który często jest wykorzystywany do skryptów. Język Python często nazywany jest językiem zorientowanym obiektowo skryptowym językiem.[31]

Python jest jednym z najpopularniejszych języków programowani wykorzystanym w analizie danych, dzięki temu zawiera olbrzymią bazę dodatkowych bibliotek[38].
W niniejszej pracy zostały wykorzystane następujące biblioteki

- Numpy - biblioteka stworzona do obliczeń numerycznych na wielowymiarowych tablicach oraz macierzach, dodatkowo wspomniana biblioteka wspiera także funkcje matematyczne, co sprawia że jest jedną z najbardziej popularnych bibliotek do operacji numerycznych. Pakiet Numpy jest otwarto-źródłowym oprogramowaniem napisanym głównie w języku programowania C oraz zawiera elementy napisane w języku Fortran. Wykorzystanie języków niskiego poziomu pozwala przetwarzać dane w pamięci operacyjnej w efektywniejszy sposób, co jest istotne w operacjach na dużych zbiorach danych [9].

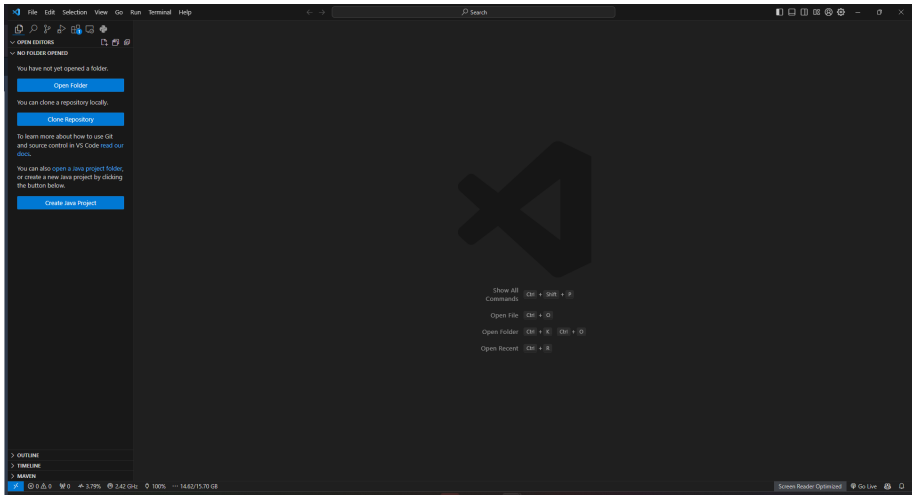
- Tensorflow - otwarto-źródłowa biblioteka programistyczna autorstwa Google Brain Team wydana w 2015. roku. Niniejszy pakiet wykorzystywany jest w uczeniu maszynowym oraz głębokich sieciach neuronowych. Biblioteka została zaimplementowana w języku C++ oraz dodatkowo zaimplementowane są frontend'y (warstwy aplikacji odpowiedzialne za współdziałanie z użytkownikiem) napisane w językach programowania takich jak Python oraz C++. Dodatkowo posiada warstwę API, która zapewnia interfejs używany w modelach głębokiego uczenia [16]. W najwyższej warstwie znajdują się interfejsy wysokopoziomowe takie jak Keras oraz Estimator API, z których to Keras został wykorzystany do utworzenia modelu sieci neuronowej.

- Matplotlib - otwarto-źródłowa biblioteka stworzona przez John'a Hunter'a do celów wizualizacji danych [8]. Niniejszy pakiet oprogramowania umożliwia wizualizację danych za wykresach w formacie 2D oraz 3D

- Pandas - pakiet języka Python stworzony do analizy oraz manipulacji danymi, zwłaszcza w kontekście danych tabelarycznych [33].

- Librosa - biblioteka dedykowana analizie oraz przetwarzaniu sygnałów dźwiękowych poprzez ekstrakcję cech dźwięku, analizy widmowej oraz przetwarzania sygnałów audio [4]

Do celów implementacji pracy inżynierskiej wykorzystano środowisko programistyczne Visual Studio Code 2.2 stworzone przez firmę Microsoft w 2015 roku. Powyższe środowisko jest przeznaczone do edycji, debugowania oraz testowania kodu napisanego w różnych językach programowania [21] w tym języka Python, który został użyty w celu implementacji modelu sieci neuronowej.



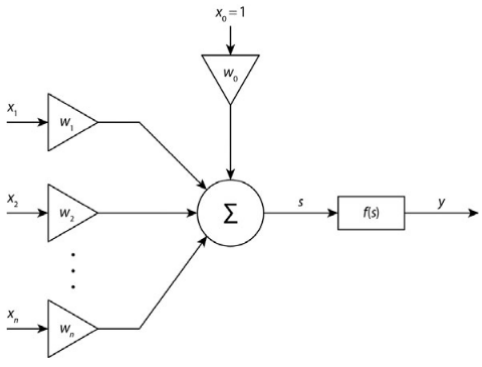
Rysunek 2.2 Zdjęcie poglądowe programu Visual Studio Code

2.3 Sztuczna sieć neuronowa

Sztuczna sieć neuronowa (ang. *Artificial Neural Network, ANN*), często nazywana siecią neuronową jest jednym z najważniejszych algorytmów uczenia maszynowego, szybko ewoluującym wraz z koncepcją uczenia głębokiego (ang. *Deep learning*) [30].

Budowa sztucznego neuronu bazowana jest na budowie i działaniu neuronu naturalnego [27]. W celu przedstawienia działania pojedynczego neuronu (rysunek 2.3) użyto następujących oznaczeń

- $x = [x_1, x_2, ..., x_n]$ - wektor wartości wejściowych
- $w = [w_1, w_2, ..., w_n]$ - wektor wag
- w_0 - wartość progowa pobudzenia
- s - wartość zsumowana
- y - wartość wyjściowa
- $f(s)$ - funkcja aktywacji



Rysunek 2.3 Model sztucznego neuronu [27]

Działanie neuronu wyraża się zależnością

$$y = \int(s) \quad [27] \tag{2.1}$$

gdzie,

$$s = \sum_{i=0}^{n} x_i * w_i \quad [27] \tag{2.2}$$

Powyższy przykład budowy pojedynczego sztucznego neuronu działa na zasadzie otrzymania iloczynu wartości wejściowych x przez odpowiadające im wagi w, a następnie zostają zsumowane w członie sumującym $\sum$, po czym suma zostaje przekształcona za pomocą funkcji aktywacji f do wartości wyjściowej y [27].

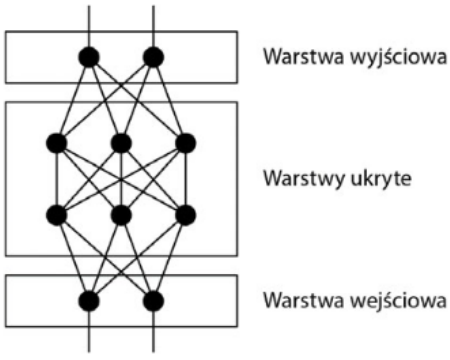
Ważną rolę w przetwarzaniu danych przez neuron pełni waga pobudzenia w_0, której początkowa wartość wynosi 1. Waga pobudzenia jest parametrem przypisanym do każdego wejścia sztucznego neuronu i ma wpływ na aktywację [27]. Zależnie od wartości jaką posiada waga pobudzenia, która może być dodatnia, ujemna lub równa 0, taki wpływ będzie miała na aktywację danego neurony w warstwie.

Rozbudowane sieci neuronowe składają się z wielu sztucznych neuronów połączonych między sobą w sieci, w których można wydzielić 3 rodzaje warstw 2.4

- warstwa wejściowa - głównym celem warstwy wejściowej jest przyjmowanie danych, które następnie są normalizowane i przekazane do kolejnych warstw. W typowej sieci neuronowej warstwa wejściowa charakteryzuje się liczbą neuronów równą liczbie cech lub wymiarów. W niniejszej pracy zostały wykorzystane sieci BCRNN, w których warstwa wejściowa filtruje dane, a następnie skaluje je w celu zmniejszenia złożoności obliczeniowej poprzez przechowywanie zmniejszonej liczby parametrów do analizy, rozszerzony opis warstw dwukierunkowych sieci konwolucyjno-rekurencyjnych został przedstawiony w rozdziale 4.4 poświęconemu opisowi zastosowanego modelu.

- warstwy ukryte - warstwy znajdujące się pomiędzy warstwą wejściową, a wyjściową

- warstwa wyjściowa - niniejsza warstwa na podstawie otrzymanych informacji z warstwy ukrytej generuje decyzje i prognozy.



Rysunek 2.4 Przykład sztucznej sieci neuronowej z 2 warstwami ukrytmi [27]

W niniejszej pracy została zastosowana wariacja sztucznej sieci neuronowej w postaci dwukierunkowej konwolucyjno-rekurencyjnej sieci neuronowej.

2.4 Sztuczna inteligencja w przetwarzaniu sygnałów fonicznych

W ostatnich latach wykorzystanie sztucznych inteligencji do przetwarzania sygnałów fonicznych zyskało dużą popularność. Z użyciem algorytmów sztucznej inteligencji rozwiązywane zostały przykładowe problemy:

- systemy rekomendacji, które filtrują materiały audio, na podstawie historii odsłuchu danego użytkownika;
- identyfikacja instrumentów, w tym także separacja dźwięku w ścieżkach audio, przykładem zastosowania jest aplikacja PixelPlayer, która separuje ścieżki audio, a następnie mapuje je na obrazie wideo to pikseli odpowiadających źródle danego sygnału audio [10];
- klasyfikowanie utworów muzycznych na gatunki, bądź też detekcja sygnałów fonicznych otoczenia, bazująca na analizie tempa, tonu, rytmu, itp.. Przykładem aplikacji wykorzystującej algorytmy do rozpoznawania i klasyfikacji utworów jest np. Shazam [45]
- rozpoznawanie mowy, wykorzystywane w metodach biometrycznych oraz w asystentach głosowych takich jak Siri lub asystent Google [46, 44]

Pomimo wykorzystania algorytmów sztucznej inteligencji do analizy dźwięków, nie jest to temat popularny wśród publikacji naukowych [42]. Rozwijanie badań na sztuczną inteligencją oraz głębokim uczeniem wymaga duży baz próbek audio, które posiadają wiarygodną ocenę testów subiektywnych [42]

Rozdział 3

Baza danych audio

3.1 Wprowadzenie

W celu uczenia modelu sieci neuronowych została przygotowana baza sygnałów fonicznych. Całość bazy została opracowana na bazie utworu Antonio Vivaldi'ego - "Cztery pory roku - Wiosna cz. I Allegro".

3.2 Parametry próbek audio

Według dostępnych źródeł w przetwarzaniu danych audio, najczęściej stosowany jest czas trwania sygnału 1 - 10s [34]. Sygnał został podzielony na próbki o czasie trwania 3s w celu optymalnego przetworzenia przez sieć neuronową w procesie nauki z podziałem na sygnał niezdegradowany oraz po różnych formach degradacji. Przygotowane próbki zostały poddane degradacji w sposób stochastyczny. Bazowy sygnał w postaci utworu był zapisany w postaci pliku WAV z częstotliwością próbkowania 44100 Hz.

Format plików audio WAV (ang. *waveform audio format*) został stworzony przez firmę Microsoft wraz z wprowadzaniem systemu Windows 3.1 w 1992. Następnie niniejszy format został przyjęty jako podstawa dla normy europejskiej platformy nadawczej (ang. *European Broadcasting Union*, EBU) [23]. Format WAV przechowuje dane audio w jakości bezstratnej, co oznacza że nie są poddawane żadnej kompresji w celu zredukowania rozmiaru w pamięci. Dodatkowo pliki wav są przetwarzane w zapisie bitowym (ang. *Bit Depth*, takim jak 8-bit, 16-bit oraz 32-bit, co wpływa na precyzję reprezentacji amplitudy [22]. W niniejszej pracy wykorzystywano próbki nagrań monofonicznych z głębokością bitową 32-bit, co daje przepływność danych na poziomie 1411kbps.

Jednymi z najczęściej wykorzystywanych cech do zadań związanych z analizą sygnałów dźwiękowych to [35]:

- cechy widmowe, które bazują na danych otrzymanych z częstotliwościowej reprezentacji sygnału;
- cechy cepstralne bazujące na rozkładzie sygnału zgodnie z modelowaniem uwzględniającym źródło sygnału oraz nałożone filtry. Jednym z najczęściej wykorzystywanych cech cepstralnych są współczynniki MFCC (ang. *Mel Frequency Cepstral Coefficients*);
- cechy czasowe obliczane na podstawie przebiegów czasowych, np. obwiednia sygnału lub gęstość przejść przez zero.

- cechy oparte na obrazie spektrogramu, które uwzględniają czas oraz widmo badanego sygnału.

W niniejszej pracy do ewaluacji modelu wykorzystano spektrogramy z filtrami melowymi. Zostały one opisane w rozdziale 3.5.

3.3 Przygotowanie bazy danych

Zgodnie z posiadaną wiedzą autora, nie istnieje żadna oficjalna baza próbek reprezentująca zniekształcenia w sygnale fonicznym. W tym celu została sporządzona baza próbek na podstawie wybranego utworu. Baza została stworzona za pomocą skryptu napisanego w języku Python - split_file_to_sample 3.1

```
def split_file_to_sample(fileName: str,
                         dirOutput : str = "./dump",
                         distortion : bool = False,
                         noise : bool = False,
                         intermodulation : bool = False,
                         flutter : bool = False,
                         remove_part : bool = False,
                         step_size = 0.2):
```

Rysunek 3.1 Definicja funkcji do dzielenia podstawowego pliku na pomniejsze próbki

Powyższa definicja funkcji przyjmuje następujące dane na wejściu:

- fileName - zmienna, która jest równocześnie nazwą pliku źródłowego w formacie łańcucha snaków (ang *string*), które służą do przechowywania informacji tekstowych, a także dowolnych zbiorów bajtów [31]. W przypadku pliku źródłowego wymagane jest dostarczanie plików w formacie WAV ze względu na wykorzystanie funkcji wczytujących dostosowanych do tego formatu.

- dirOutput - zmienna, która przyjmuje ścieżkę do folderu wyjściowego, w którym zostaną umieszczone wyjściowe próbki

- distortion - zmienna, przekazująca wartość logiczną (ang. *boolean*, bool), przekazanie wartości 'Prawda' (ang. *True*) uruchamia dany warunek w funkcjach typu 'jeśli' (ang. *if*). Zmienna distortion uruchamia możliwość użycia zmiennych noise, remove_part, intermodulation oraz flutter, które odpowiadają kolejno za szum, brakujące dane, zniekształcenia intermodulacyjne oraz zniekształcenia typu flutter

- step_size - zmienna zmiennoprzecinkowa typu float, która określa skok pomiędzy kolejnymi próbkami tworzonymi na podstawie pliku źródłowego.

Na powyższym obrazku 3.2 została przedstawiona cała funkcja służąca do utworzenia bazy plików treningowych. Na początku wczytywane są dane pliku audio (*audio_data*) oraz częstotliwość próbkowania (*sample_rate*), do wczytywania plików audio w formacie WAV została użyta funkcja *read()* z klasy wavfile, która była zawarta w bibliotece Scipy [11]. Biblioteka scipy została opisana w rozdziale 2.2. Następnie jest odczytywana długość pliku, a na podstawie częstotliwości próbkowania oraz zmiennej *step_size* określana jest długość skoku dla pętli. Do celów numeracji próbek została stworzona zmienna licząca *count_steps*. W celu poprawnego zapisu został stworzony warunek sprawdzający, czy podana ścieżka wyjściowa jest zgodna z wybranym folderem, a w przypadku niezgodności zostaje utworzony folder docelowy.

```python
def split_file_to_sample(fileName: str,
                         dirOutput : str = "./dump",
                         distortion : bool = False,
                         noise : bool = False,
                         intermodulation : bool = False,
                         flutter : bool = False,
                         remove_part : bool = False,
                         step_size = 0.2):
    sample_rate, audio_data = wavfile.read(fileName)
    audio_length = len(audio_data)
    loop_step = int(sample_rate * step_size)
    count_steps = 0
    if not os.path.isdir(dirOutput):
        os.mkdir(dirOutput)
    for i in range(0, audio_length-(3*sample_rate), loop_step):
        output_name = f"{dirOutput}/sample_{count_steps:04d}.wav"
        start = i
        end = start + (3*sample_rate)
        data = audio_data[start:end]
        if distortion:
            if intermodulation:
                data = add_intermodulation(data, sample_rate)
            if noise:
                data = add_noise(data)
            if flutter:
                data = add_flutter(data, sample_rate)
            if remove_part:
                data = remove_part_data(data)
        wavfile.write(output_name, rate=sample_rate, data=data.astype(np.float32))
        count_steps += 1
        os.system('clear')
        print(f"Progress: {count_steps} / {int(audio_length/loop_step)}, {output_name}")
```

Rysunek 3.2 Funkcja wykorzystywana do dzielenia pliku źródłowego na próbki

Następnie tworzona jest pętla która iteruje po próbkach w pliku audio zgodnie z wcześniej obliczonym skokiem pomiędzy kolejnymi próbkami. W celu podziału plików audio na fragmenty o tym samym czasie trwania, tj. 3s, iteracja po pętli kończy się na próbce będącej na miejscu równym trzykrotności częstotliwości próbkowania od końca.

Następnie jest wydzielany fragment pliku audio trwający 3s. W przypadku tworzenia bazy plików z zniekształceniami, należy wybrać wyżej opisane zmienne w momencie wywoływania funkcji *split_file_to_sample* 3.3. Na podstawie spełnienia tego warunku zostaje sprawdzone, które rodzaje zniekształceń powinny zostać wprowadzone do pliku wyjściowego.

```python
ft.split_file_to_sample("./input/example_temp.wav", "./prepare_data/02_output_noise", distortion=True, noise=True, step_size=0.04)
```

Rysunek 3.3 Przykładowe wywołanie funkcji do tworzenia próbek ze zniekształceniami

Następnie funkcja wykorzystywana do dzielenia plików na próbki zapisuje dane audio do plików za pomocą funkcji read z biblioteki scipy [12], która wymagał podania kolejno ścieżki pliku wraz z nazwą oraz rozszerzeniem, częstotliwości próbkowania oraz danych audio do zapisu.

W celu dodania zniekształceń do plików audio zostały przygotowane funkcje odpowiedzialne za szum 3.4, zniekształcenia intermodulacyjne oraz zniekształcenia typu flutter.

```
def add_noise(audio_data):
    level = np.random.randint(1,100)/1000
    noise=np.random.normal(0, level, audio_data.shape[0])
    noisy_audio = audio_data + noise
    if noisy_audio.max() > 1:
        noisy_audio = (noisy_audio/np.max(np.abs(noisy_audio)))
    return noisy_audio
```

Rysunek 3.4 Przykład funkcji do zniekształceń szumowych

Powyżej została przedstawiona funkcja (rysunek 3.4) dodająca losowe zniekształcenia szumowe, która w momencie wywołania otrzymuje zmienną z plikiem audio, a następnie stochastycznie dobiera maksymalny poziom szumu, który zostanie dodany do nagrania. W przypadku przekroczenia maksymalnej wartości w zapisie plików w typie float32 (zapis zmiennoprzecinkowy z głębokością bitów 32-bit [14]), tj. wartość 1, sygnał zostaje znormalizowany.

```
def add_intermodulation(audio_data, sample_rate = 44100, fileReturn : bool = False):
    max_amp = np.max(np.abs(audio_data))
    factor = np.random.randint(0,5)
    inter_audio = audio_data + factor * (audio_data**2)
    inter_audio = (inter_audio/np.max(np.abs(inter_audio)))*max_amp

    if fileReturn:
        wavfile.write(f"./dump/inter_file.wav", rate=sample_rate, data=inter_audio)
    else:
        return inter_audio
```

Rysunek 3.5 Funkcja dodająca zniekształcenia intermodulacyjne

Powyższa funkcja (rysunek 3.5) dodaje zniekształcenia intermodulacjne. Niniejsza funkcja przyjmuje dane audio, z których następnie jest odczytywana największa wartość bezwzględna, następnie dobierana jest w sposób losowy wartość, która będzie odpowiedzialna za poziom zniekształceń intermodulacyjnych po przemnożeniu przez kwadrat sygnału audio. W celu normalizacji zakresu sygnału audio do wartości pomiędzy -1.0, a 1.0, sygnał jest dzielony przez maksymalną wartość zniekształconego już sygnału (*inter_ audio*), a następnie obliczany jest iloczyn z pobranej na początku funkcji wartości maksymalnej bezwzględnej sygnału przed dodaniem zniekształceń.

```
def add_flutter(audio_data, sample_rate = 44100, fileReturn : bool = False):
    flutter_time = np.linspace(0, len(audio_data) / sample_rate, len(audio_data))
    depth = 0.001
    rate = np.random.randint(1,30) / 10
    flutter_sinus = 1.0 + depth * np.sin(2.0 * np.pi * rate * flutter_time)
    flutter_audio = np.interp(flutter_time, flutter_time/flutter_sinus, audio_data)
    if fileReturn:
        wavfile.write(f"./dump/flutter_file.wav", rate=sample_rate, data=flutter_audio)
    else:
        return flutter_audio
```

Rysunek 3.6 Funkcja dodająca zniekształcenia typu flutter

Funkcja dodająca zniekształcenia typu flutter (rysunek 3.6) przyjmuje dane audio oraz opcjonalnie informację o częstotliwości próbkowania oraz zapisie do pliku. Kolejnym krokiem jest utworzenie listy czasu (*flutter_ time*) oraz parametrów wpływu na amplitudę audio (*depth*) i częstość drgań zniekształcenia flutter (*rate*). Na podstawie tych zmiennych

jest tworzona funkcja sinusoidalna (*flutter_sinus*), która jest interpolowana z sygnałem audio (*flutter_audio*).

Funkcja usuwająca fragmenty danych, w sposób stochastyczny dobierała miejsca usunięcia 20ms próbek danych.

3.4 Wstępne przetwarzanie

W celu sterowania parametrami bazy treningowej oraz modelu sztucznej sieci neuronowej zostały wprowadzone hiperparametry.

Na poniższym rysunku 3.7 zostały przedstawione parametry, które były wykorzystane do przygotowanie przetworzenia bazy plików audio

- *sample_rate* - Częstotliwość próbkowania (ang *sample rate* - liczba próbek na sekundę przetwarzanych w pliku audio
- *n_fft* - Próbki (*n_fft*) wykorzystane do obliczenia krótko-czasowej transformaty Fouriera, która została opisana szerzej w niniejszym rozdziale
- *win_length* - Długość okna (ang. *widnows length*) wykorzystywanego do obliczenia transformaty Fouriera
- *hop_length* - Długość kroku (ang. *hop length*) liczba próbek pomiędzy kolejnymi początkami okien
- *min_level_db* - Minimalny poziom w decybelach
- *ref_level_db* - Poziom referencyjny - wykorzystywany do konwersji amplitudy na jednostki decybelowe [1]
- *num_mels* - liczba filtrów w banku melowym
- *mel_fmin/fmax* - zakres częstotliwości filtrów melowych

```
hparams = HParams(
    batch_size=50,
    n_samples = 10000,
    sample_rate=44100,
    n_fft=2048,
    win_length=2048,
    hop_length=512,
    min_level_db=-100,
    ref_level_db=20,
    num_mels=128,
    mel_fmin=0,
    mel_fmax=22050,
    power=1.5,
    griffin_lim=50,
)
```

Rysunek 3.7 Przykładowe hiperparametry

Sygnały foniczne w formacie audio zostały wczytane do aplikacji poprzez funkcję *load_files*

```
def load_files(*args, start = 0, end= 100, distract_part=0):
    data = {
        'audio': [],
        'sampleName': [],
        'target': [],
    }
    if end>len(os.listdir(args[0])): end = len(os.listdir(args[0]))
    random_numbers = random.sample(range(start,end), k=int(end*distract_part))

    dir_correct = args[0]
    dir_distract = args[1]
    for fileNumber, fileName in enumerate(sorted(os.listdir(dir_correct))[start:end]):
        filePath = dir_distract if fileNumber in random_numbers else dir_correct
        stat = 'correct' if filePath == dir_correct else 'noise'
        if fileName.endswith('.wav'):
            wavPath = os.path.join(filePath, fileName)
            audio, _ = sf.read(wavPath)
            audio_2, _ = sf.read(os.path.join(dir_correct, fileName))

            data['sampleName'].append(tf.cast(f"{stat} - {fileName}", dtype=tf.string))
            data['audio'].append(tf.cast(audio[768:131532], dtype=tf.float32))
            data['target'].append(tf.cast(audio_2[768:131532], dtype=tf.float32))
    return data
```

Rysunek 3.8 Funkcja służąca do ładowania plików WAV

Funkcja przedstawiona na rysunku 3.8 przyjmuje listę w postaci argumentów (*args*), w których są przesyłane ścieżki do plików. Ze względu na budowę modelu, należy załadować dane bez zniekształceń oraz zawierające zniekształcenia. Do celów testowania modelu na mniejszych zbiorach zaimplementowano zmienne ograniczające zakres plików pobranych z wybranych katalogów (*start, end*). Dodatkowo w celu określenia stopnia wymieszania próbek został dodany parametr *distract_part*. Na rysunku 3.9 zostało przedstawione wywołanie niniejszej funkcji. W wywołaniu uwzględniono katalog plików z nieniekształconymi plikami (*"./prepare_data/01_output_correct"*) oraz plikami zniekształconymi za pomocą szumu (*"./prepare_data/02_output_noise"*)

```
data_1 = load_files("./prepare_data/01_output_correct",
                    "./prepare_data/02_output_noise",
                    min_range=200,
                    size = hparams.n_samples,
                    distract_part= 0.2)
```

Rysunek 3.9 Wywołanie funkcji ładowania plików

Początkowym etapem wykonywania funkcji jest przygotowanie słownika (*data*) z wydzielonymi kategoriami, tj.

- *audio* - zapis pliku audio z formatu WAV
- *sampleName* - nazwa próbki, dodatkowo dopisywane będzie źródło próbki
- *target* - docelowy plik audio, czyli próbka niezawierająca zniekształceń

Następnie funkcja sprawdza czy podany parametr *end* sprawdza, czy nie został podany większy niż ilość plików w katalogu i w takim przypadku zostaje ograniczony. Kolejnym

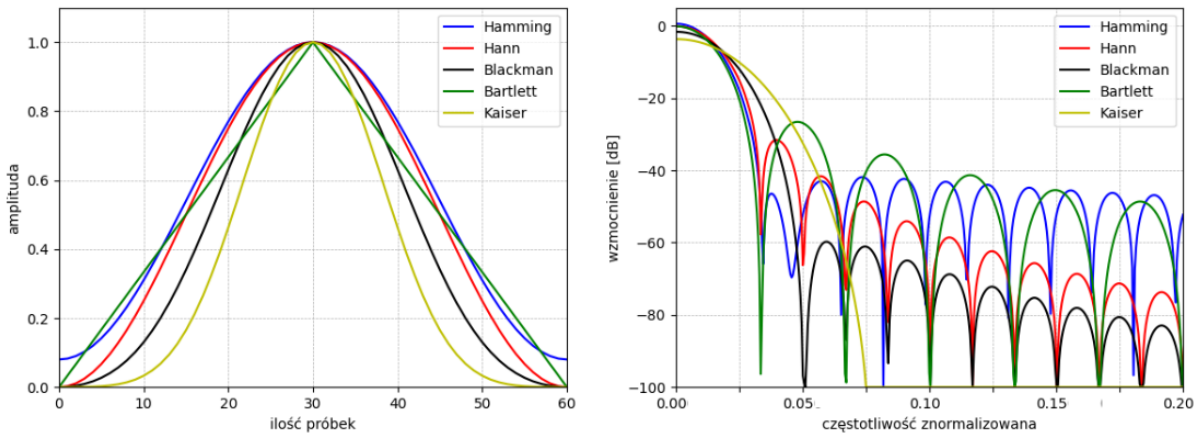
krokiem funkcji jest dobranie w sposób stochastyczny indeksów próbek zniekształconych według parametru *distract_part*, w tym celu została wykorzystana biblioteka random, która służy do generowania pseudolosowych liczb na podstawie otrzymanych parametrów. W przypadku funkcji *sample* z biblioteki random przyjmuje ona tablicę liczb z której następnie losuje k-próbek bez powtarzania.

Kolejnym krokiem wykonywanym w funkcji jest pętla, która iteruje po plikach w folderze z poprawnymi danymi (ze względu nazywanie próbek w ten sam sposób dla każdego typu plików, nie jest wymagana iteracja po każdym folderze osobno). W trakcie wykonywania się pętli zostaje sprawdzone, czy wczytywany plik powinien być plikiem zawierającym zniekształcenia oraz zostaje wczytana wersja z zniekształceniami oraz wersja pliku bez zniekształceń. Po wczytaniu obu plików są one formatowane na typ danych, który współpracuje z biblioteką Tensorflow, tj. *tf.float32, tf.string* i dodawane do odpowiadającej im kategorii w słowniku.

Dodatkowo zostały wczytane próbki testowe, które nie znajdują się w zestawie treningowym.

Następnym krokiem jaki należało wykonać przed przystąpieniem do trenowania modelu, to dostosowanie wczytanych danych do przetwarzania przez sztuczna sieć neuronową, w tym celu została przygotowana klasa zawierająca metody do przetwarzania plików audio na spektrogramy z wykorzystaniem filtrów melowych.

W celu ekstrakcji cech sygnału w dziedzinie częstotliwości należało dobrać wartości ramki oraz okienkowania, a następnie obliczyć widmo częstotliwościowe. W tm celu ze względu na możliwość zapewnienia możliwie największej rozdzielczości, nie wpływając nadmiernie na czas przetwarzania. W zadaniach związanych z przetwarzaniem akustycznym najczęściej wykorzystywane są ramki o długość 20 - 60 ms z nakładkowaniem większym niż 50%. Wczytywane dane zostały podzielone na ramki o długości wynoszącej 2048 próbek, czyli 46,4ms (dla sygnału próbkowanego z częstotliwością 44 100 Hz) z odległością pomiędzy kolejnymi początkami ramek wynoszącą 512 próbek, czyli dało to nakładkowanie na poziomie 75%. W celu minimalizacji efektu wycieku widma częstotliwościowego zastosowano nieprostokątne okno Hanna, które jest jednym z najczęściej wykorzystywanych ze względu na duże tłumienie listków bocznych oraz tylko dwukrotnie poszerza główny listek w porównaniu do innych rodzajów okien (rysunek 3.10) [35].



Rysunek 3.10 Porównanie okien czasowych w dziedzinie czasu i częstotliwości [35]

W celach obliczeniowych została wykorzystana dyskretna transformata Fouriera (ang. *Discrete Fourier Transform*, DFT), ze względu na możliwość efektywnej analizy sygnałów dźwiękowych [43]. Obliczanie DFT na podstawie ramek nazywane jest krótko-czasową transformatą Fouriera i jest obliczana z następującego twierdzenia [43]:

$$X[t,j] = \sum_{m=0}^{N-1} \omega[m]x[tN+m]e^{\frac{-i2\pi mj}{N}} \tag{3.1}$$

gdzie:

- j - numer składowej dyskretnej transformaty Fouriera
- t - numer okna
- $\omega[m]$ funkcja okienkowania, w przypadku niniejszej pracy to okno Hanna
- N - długość ramki

Powyższe obliczenia zostały wykonane na pomocą dedykowanej funkcji *signal.stft()* z biblioteki Tensorflow. Budowa funkcji (rysunek 3.11) tworzącej spektrogram bazowała na bibliotece Librosa [5]. Dodatkowo po przeliczeniu składowych krótko-czasowej transformaty Fouriera wszystkie wartości zostały przeliczone na decybele (dB) (rysunek 3.12), a następnie w celu efektywniejszego przetwarzania przez model sztucznej sieci neuronowej zostały znormalizowane do wartości w zakresie 0 - 1 za pomocą wbudowanej w bibliotece Tensorflow metody *clip_ by_ value*, która ogranicza wartości będące powyżej lub poniżej wybranego zakresu.

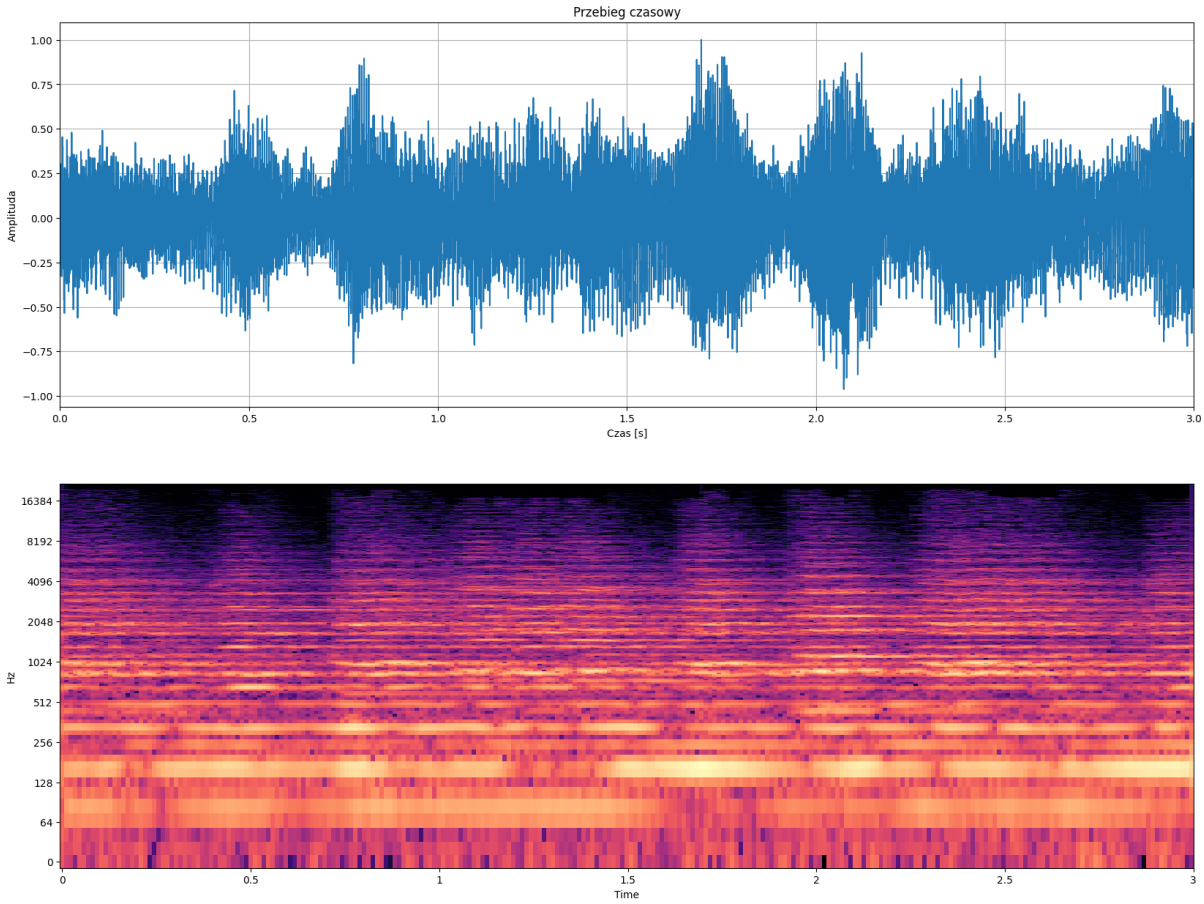
```
def _spektrogram(signals, hparams):
    stft = tf.signal.stft(
        signals,
        hparams.win_length,
        hparams.hop_length,
        hparams.n_fft,
        pad_end=True,
        window_fn=tf.signal.hann_window,
    )
    db_spektrogram = amp_na_db(tf.abs(stft)) - hparams.ref_level_db
    return tf.clip_by_value((db_spektrogram - hparams.min_level_db) / -hparams.min_level_db, 0, 1)
```

Rysunek 3.11 Funkcja przetwarzająca sygnał audio na spektrogram

```
def _log10(x):
    return tf.math.log(x) / tf.math.log(tf.constant(10, dtype='float32'))
def amp_na_db(x):
    return 20 * _log10(tf.clip_by_value(tf.abs(x), 1e-5, 1e100))
```

Rysunek 3.12 Funkcje przeliczające wartość amplitudową na decybele

Uzyskane wyniki STFT to dwuwymiarowa macierz reprezentująca sygnał audio w postaci widma częstotliwościowego, czyli amplitudę składowych stałych w dziedzinie czasu (rysunek 3.13). Uzyskane wyniki zostały przekonwertowane za pomocą banku filtru melowych do skali melowej (rozdział 3.5).



Rysunek 3.13 Przykładowy przebieg czasowy oraz spektrogram

3.5 Skala melowa

Spektrogram z zastosowaną skalą melową w dziedzinie czasu umożliwia dokładniejszą symulację działania ludzkiego narządu słuchu [26]. Celem funkcji melowej m jest uwzględnienie percepcji słuchu dla poszczególnych zakresów częstotliwości.

$$m = 2595 * \log_{10}\left(1 + \frac{f}{700}\right) \tag{3.2}$$

gdzie m reprezentuje funkcję melową, a f częstotliwość [26].

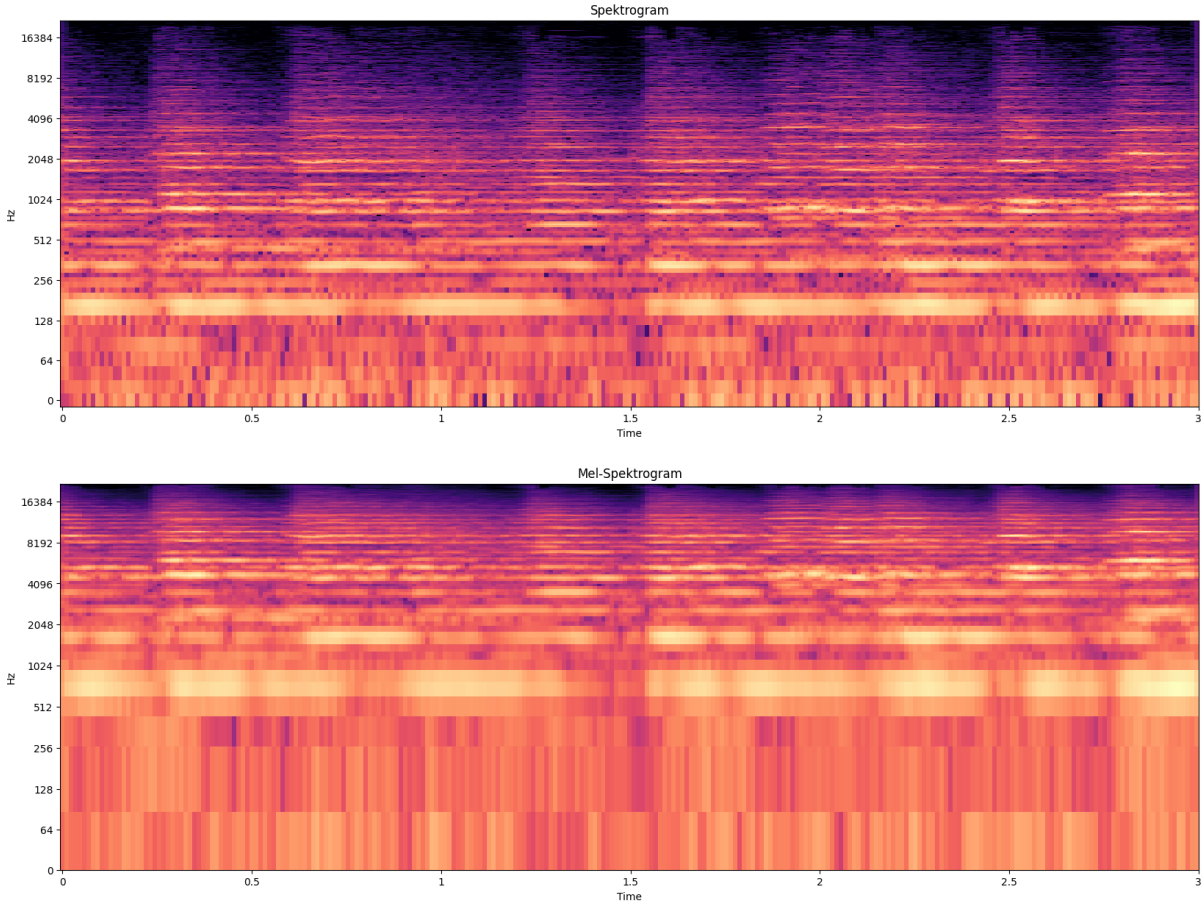
Zastosowanie skali melowej do analizy spektrogramów (rysunek 3.14) przeprowadzą się za pomocą banku filtrów melowych (ang. *Mel filter bank*). Bank filtrów melowych jest zestawieniem nakładających się na siebie filtrów trójkątnych, których częstotliwości centralne rozłożone są równomiernie na osi reprezentującej wartości w skali melowej [35]. W celu obliczenia współczynników spektrogramu melowego żeby posłużyć się następującym wzorem [41]:

$$S_{mel}[k,t] = \sum_{l=0}^{L-1} m_k[l]|S[l,t]^2| \tag{3.3}$$

gdzie:

- L - liczba składowych częstotliwości źródłowego sygnału

- $m_k[l]$ - filtr trójkątny z banku filtrów melowych indeksowany l
- S_{mel} współczynnik wektora spektrogramu melowego i indeksie k oraz ramce t



Rysunek 3.14 Przykładowy przebieg spektrogramu oraz spektrogramu z wykorzystaniem banku filtrów melowych

Poniżej została przedstawiona funkcja przygotowująca bank filtrów melowych w postaci macierzy (rysunek 3.15). Funkcja została zaimplementowana za pomocą pakietu Tensorflow na podstawie funkcji zawartych w bibliotece Librosa w celu dopasowania przetwarzanych danych do modelu sztucznej sieci neuronowej [15, 36]. W niniejszej pracy zostało zastosowane 128 filtrów melowych.

Przygotowanie banku filtrów melowych (rysunek 3.15) wykonywane jest na podstawie hiperparametrów (rysunek 3.7) za pomocą funkcji z biblioteki Tensorflow *linear_to_mel_weight_matrix*, która przetwarza otrzymane parametry na wagi służące do przetworzenia spektrogramów w postaci tensorów na spektrogramy melowe [20]. Następnie zostaje pobrana lista środkowych częstotliwości filtrów melowych za pomocą funkcji *mel_frequencies* z biblioteki Librosa [7].Przed ostatnim krokiem jest nałożenie filtrów banku melowego względem pobranych częstotliwości. Na koniec w celu efektywnego przetwarzania danych przez sieć neuronową dane są normalizowane do wartości w zakresie 0 - 1 [15].

```
bank_mel_filter = tf.signal.linear_to_mel_weight_matrix(
    num_mel_bins=hparams.num_mels,
    num_spectrogram_bins=int(hparams.n_fft/2)+1,
    sample_rate=hparams.sample_rate,
    lower_edge_hertz=hparams.mel_fmin,
    upper_edge_hertz=hparams.mel_fmax,
    dtype=tf.dtypes.float32,
)

mel_f = librosa.mel_frequencies(n_mels=hparams.num_mels + 2, fmin=hparams.mel_fmin, fmax=hparams.mel_fmax,)

mel_norm = tf.dtypes.cast(tf.expand_dims(tf.constant(2.0 / (mel_f[2 : hparams.num_mels + 2] - mel_f[:hparams.num_mels])), 0), tf.float32)

bank_mel_filter = tf.multiply(bank_mel_filter, mel_norm)
bank_mel_filter = tf.divide(bank_mel_filter, tf.reduce_sum(bank_mel_filter, axis=0))
```

Rysunek 3.15 Przygotowanie banku filtrów melowych

Poniżej została przedstawiona funkcja przetwarzająca sygnał audio na spektrogram melowy (rysunek 3.16).

```
example['spectrogram'] = _spektrogram(example['audio'], self.hparams)
example['spectrogram'] = tf.expand_dims(tf.tensordot(example['spectrogram'], self.bank_mel_filter, 1), axis=2)
```

Rysunek 3.16 Przykładowe przetworzenie danych audio na spektrogram

Powyższy wycinek kodu (rysunek 3.16) przedstawia przekształcenie próbki nieprzetworzonego sygnału audio na macierz składowych stałych
w dziedzinie częstotliwości.Następnie zostaje wywołana funkcja, która przetwarza spektrogram przez bank filtrów melowych w celu przeskalowania spektrogramu na skalę melową.

Tak przetworzone dane zostały zapisane w postaci tensorów, które zostały wykorzystane do ewaluacji modelu sztucznej sieci neuronowej.

W procesie uczenia dla każdego z rodzajów zniekształceń, wykorzystano 10 000 próbek do celów trenowania, z czego 20% próbek to próbki zniekształcone dobrane stochastycznie. Dodatkowo została utworzona baza próbek testowych składająca się z 200 próbek,
w tym 90% próbek było zniekształconych.

Rozdział 4

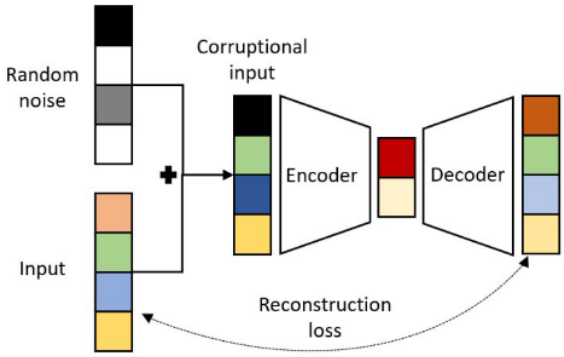
Implementacja

4.1 Wstęp

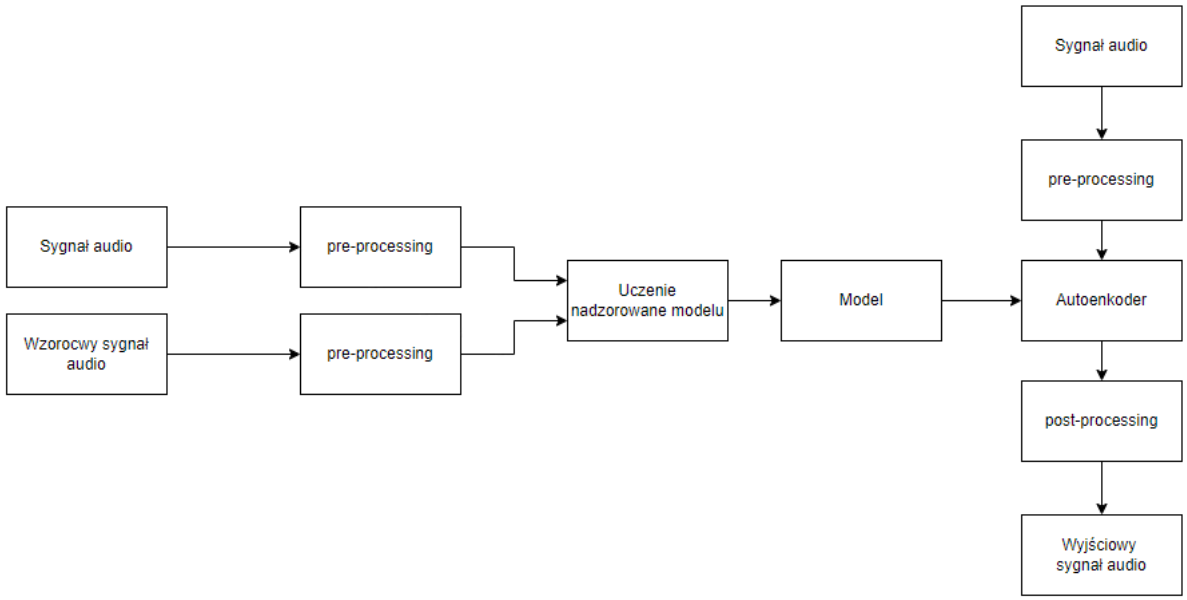
Niniejszy rozdział opisuje eksperymentalny model sztucznej sieci neuronowej bazującej na budowie autoenkodera, mająca na celu rekonstrukcję sygnału fonicznego. W kolejnym podrozdziale został opisany model działania sztucznej sieci neuronowej. Następnie zostały opisane warstwy sieci neuronowej oraz funkcje aktywacji.

4.2 Schemat modelu

Projekt modelu oparty jest na sztucznych sieciach neuronowych bazujących na budowie autoenkodera. W procesie uczenia model wykorzystuje bazę próbek wejściowych oraz odpowiadające każdej próbce sygnały wzorcowe na podstawie których dostosowywane są wagi wewnątrz modelu. Uczenie wykorzystujące dane wejściowe oraz dane wyjściowe nazywane jest uczeniem nadzorowanym (ang. *Supervised learning*) [29]. W trakcie nauki model przetwarza dane wejściowe, czyli bazę danych zawierającą sygnał zniekształcony oraz wzorcowy, a następnie na podstawie wyniku wyjściowego z modelu oraz danych wzorcowych obliczany jest gradient średniego błędu kwadratowego (ang. *mean square error* - MSE), który optymalizuje wagi modelu (rysunek 4.1). Ogólnym twierdzeniem jest to iż czym mniejszy średni błąd kwadratowy tym model jest lepszy [40]. Średnio błąd kwadratowy został opisany w kolejnym podrozdziale.



Rysunek 4.1 Przykład działania autoenkodera z uczeniem nadzorowanym [25]



Rysunek 4.2 Schemat blokowy działania modelu

4.3 Parametry modelu

Do celów ewaluacji został zbudowany model sztucznej sieci neuronowej z podziałem na grupę warstw enkodujących oraz dekodujacych (ogólny model został przedstawiony na rysunku 4.1). W celu konfigurowania parametrów zostały użyte hiperparametry modelu przedstawione na rysunku 3.7 oraz użyto dodatkowych hiperparametrów do procesu ewaluacji modelu. W skład hiperparametrów służących do konfiguracji procesu uczenia modelu wchodzą:

- wielkość wsadu (ang. *batch size*)
- funkcja strat (ang. *losses function*)
- liczba epok (ang. *epochs*)
- funkcja aktywacji (ang. *activation function*)
- optymalizator (ang. *optimizer*)

Dane treningowe są przekazywane do sieci neuronowymi pojedynczo lub partiami, których wielkość jest zależna od podanego parametru wielkości wsadu, często nazywanego *batch size* [24]. Partie danych treningowych są przetwarzane tylko raz na cykl, w trakcie którego dostosowywana jest funkcja strat, także nazywana stratą rekonstrukcji. Ilość cykli jest to liczba epok (*epochs*) w trakcie których wagi modelu są dostosowywane do przetwarzanych danych.

W trakcie ewaluacji model wykonał 100 cykli z wielkością wsadu równą 50. Dodatkowo w trakcie każdego cyklu następowała walidacja modelu w celu optymalizacji parametrów modelu.

Dobór funkcji strat jest uzależniony od typu zadania, które ma wykonywać model, w przypadku modelów bazujących na autoenkoderach służących do przetwarzania obrazu, dźwięku lub wideo najczęściej wybierane funkcje straty to [40]:

- średnio błąd kwadratowy (ang. *Mean Square Error*, MSE)

- średni błąd bezwzględny (ang. *Mean Absolute Error*, MAE) item średnia kwadratowa błędu (ang. *Root Mean Square Error*, RMSE), czyli pierwiastek z MSE

W niniejszej pracy, tak jak wspominano w poprzednich podrozdziałach została wykorzystana funkcja MSE na podstawie której obliczany był gradient, który był przekazywany do optymalizatora. Wartość MSE jest definiowana następująco [38]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{4.1}$$

gdzie:

- n - liczba próbek
- y - wartość wzorcowa
- $\hat{y}$ - wartość przewidziana

Poniżej została przedstawiona implementacja funkcji kosztu (rysunek 4.3) za pomocą biblioteki Tensorflow, w trakcie której próbka jest przetwarzana przez model w danym cyklu, a następnie obliczany jest średnio błąd kwadratowy na podstawie próbki wejściowej (x) po przetworzeniu przez model oraz próbki wzorcowej (x_t).

```
def loss_func(self, x, x_t):
    x = self.decoder(self.encoder(x))
    loss = tf.reduce_mean(tf.square(x_t - x))
    return loss
```

Rysunek 4.3 Funkcja obliczająca MSE

Na podstawie dostępnej literatury, wybrano optymalizator Adam (ang. *Adaptive Moment Estimation*), który jest kombinacją śledzenia wykładniczego rozkładu średniej gradientów z poprzednich cykli oraz śledzenia wykładniczego średniego kwadratu gradientów. Niniejszy optymalizator pozwala na efektywne obliczenia, niskim kosztem pamięci, co pozwala wydajnie przetwarzać duże zbiory danych. Optymalizator Adam jest definiowany w następujący sposób [40, 24]:

$$\begin{cases} m &= \beta_1 m - (1-\beta_1)\nabla_\theta J(\theta), \\ s &= \beta_2 s + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta), \\ \widehat{m} &= \frac{m}{1-\beta_1^t}, \\ \widehat{s} &= \frac{s}{1-\beta_2^t}, \\ \theta &= \theta + \eta\widehat{m} \oslash \sqrt{\widehat{s}+\varepsilon} \end{cases} \tag{4.2}$$

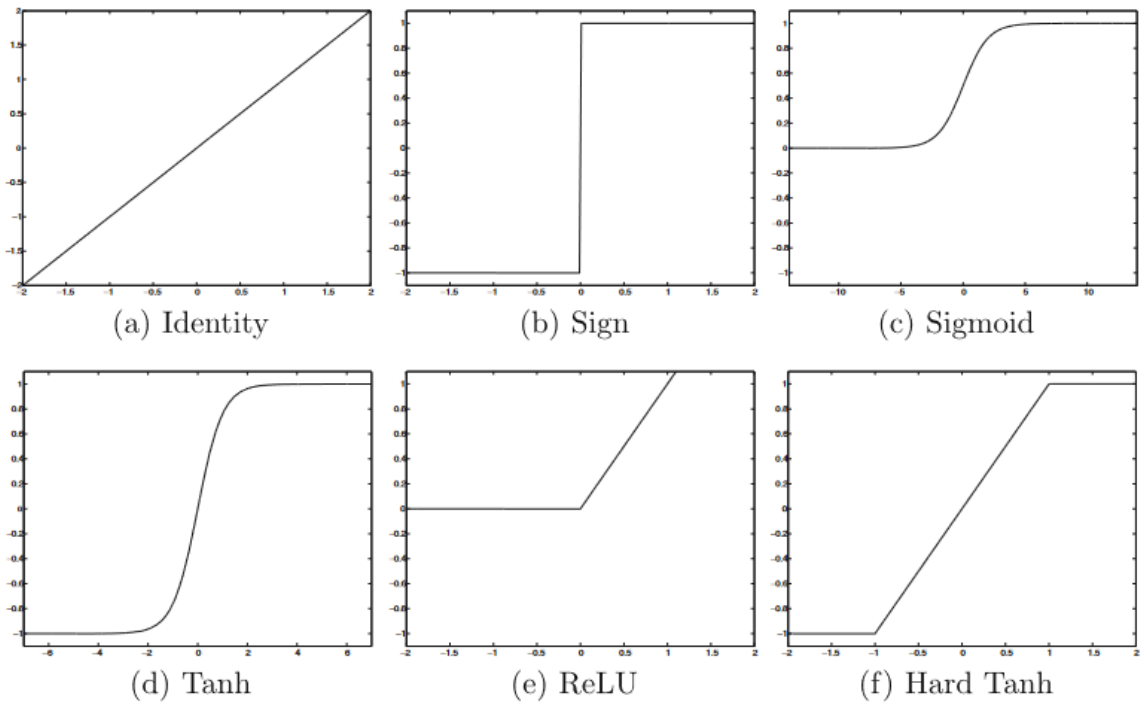
gdzie:

- s, m - wektory
- $\widehat{s}, \widehat{m}$ - wektory skorygowane pod katem obciążenia
- t - numer cyklu
- $\otimes$ - iloczyn Hadamarda

- $\oslash$ - iloraz Hadamarda
- β_1 - parametr pierwszego momentu
- β_2 - parametr drugiego momentu
- η - współczynnik uczenia
- $\nabla_\theta J(\theta)$ i przyspieszenie
- $\theta = \theta + \eta\widehat{m} \oslash \sqrt{\widehat{s} + \varepsilon}$ - człon regulujący, który uniemożliwia dzielenie przez 0

W niniejszej pracy korzystano z gotowej funkcji dostarczającej optymalizator Adam z biblioteki Tensorflow, w której ustawiono parametr η na poziomie 10^{-3} [19].

Ostatnim hiperparametrem, który w znacznym stopniu wpływa na działanie modelu sieci neuronowej jest funkcja aktywacji, którą należy wybrać na podstawie problemu, który ma zostać rozwiązany za pomocą sieci neuronowej. Niektóre funkcje aktywacji ulegają nasyceniu w skutek uczenia (rysunek 4.4), przez co zwracają wartość stałą, co jest niewskazane w procesie rekonstrukcji ze względu na nieliniowość przetwarzanych danych.



Rysunek 4.4 Przykłady najpopularniejszych funkcji aktywacji [24]

W modelu pracy wykorzystano funkcje aktywacji *ReLU* (ang. *Rectified Linear Unit*) oraz *sigmoid*. Funkcja aktywacji ReLU wchodzi w stan nasycenia tylko dla wartości ujemnych, dodatkowo pozwala ograniczyć problem eksplodującego oraz zanikającego gradientu, co wpływa na wydajność procesu uczenia modelu [24, 37]. Niniejsza funkcja aktywacji została zastosowana dla wszystkich warstw MLP, LSTM oraz wszystkich warstw konwolucyjnych z wyjątkiem ostatniej w której zastosowano funkcję sigmoid. Funkcja aktywacji ReLU jest definiowana w następujący sposób[24]:

$$\Phi(\upsilon) = max(0, \upsilon) \tag{4.3}$$

Funkcja sigmoidalna, nazywana także logistyczną, jest funkcją, która przetwarza otrzymane dane i zwraca wartość w zakresie 0 - 1. Niniejsza funkcja została zastosowana w ostatniej warstwie konwolucyjnej w celu normalizacji wartości w macierzy cech widmowych. Funkcja sigmoid jest definiowana w następujący sposób [24]:

$$\Phi(v) = \frac{1}{1 + \exp^{-v}} \tag{4.4}$$

4.4 Sieci neuronowe

W niniejszej pracy zastosowano połączenie dwóch typów sieci neuronowych, czyli konwolucyjne sieci neuronowe (CNN) oraz dwukierunkowe rekurencyjne sieci neuronowe (BRNN) bazujące na długiej pamięci krótko terminowej (ang. *long short-term memory*, LSTM).

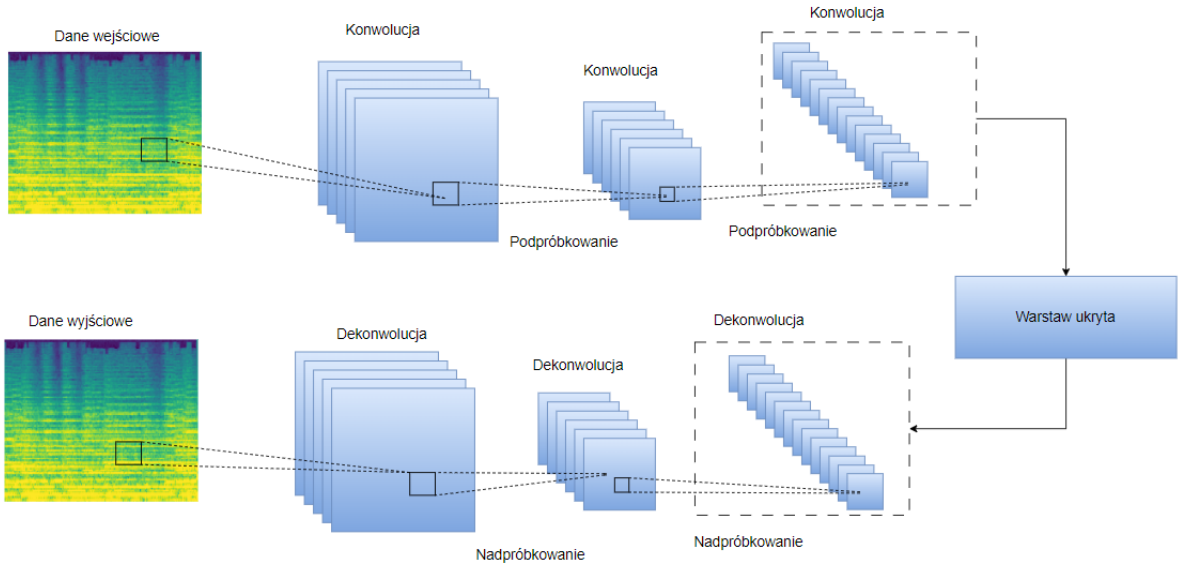
Sieci konwolucyjne są siecią posiadającą co najmniej jedną warstwę konwolucyjną, dzięki której może wyodrębniać cechy z macierzy danych, którymi mogą być obrazy lub spektrogramy [24]. Niniejsza warstwa opiera swoje działanie na operacji matematycznej splotu, który oblicza iloczyn skalarny pomiędzy zbiorem filtrów w postaci macierzy, a danymi wejściowymi [35]. Dodatkowo konwolucyjne sieci neuronowe nie wymagają interakcji każdego neuronu wejściowego z wyjściowym ze względu na wykorzystanie splotów, które pozwalają na redukcję rozmiarów danych, co daje możliwość zmniejszenia wymaganych zasobów sprzętowych oraz poprawia wydajność modelu. W przypadku przetwarzania spektrogramów, które są obrazami dwuwymiarowymi, operacja splotu wykorzystującego filtr dwuwymiarowych jest definiowana następująco [43]:

$$(w * z)[t, u] := \sum_{j=1}^{n} \sum_{k=1}^{p} w[j, k], z[t + j - 0\lceil n/2 \rceil, u + k - \lceil p/2 \rceil]\rangle \tag{4.5}$$

W przypadku autoenkoderów stosowane są także sieci dekonwolucyjne, które zwiększają rozmiar przetwarzanego obrazu w warstwie dekodującej. Jednym z elementów konwolucyjnych warstw są filtry (ang. *Kernels*), będące macierzami o rozmiarze m x n, gdzie m i n określają liczbę komórek, bądź pikseli, które zostaną razem przeanalizowane, co umożliwia wykrywanie różnych typów wzorców.

W trakcie przetwarzania warstw konwolucyjnych lub dekonwolucyjnych na początku zostają operacje kowolucji, na których w następnej kolejności jest funkcja aktywacji, w celu analizy i wykrycia wzorców. Ostatni etap składa się z operacji podpróbkowania dla warstw konwolucyjnych i nadpróbkowania dla dekonwolucyjnych [24]. Na rysunku 4.5 został przedstawiony model autoendkodera z warstwami konwolucyjnymi oraz dekonwolucyjnymi.

Zastosowanie podpróbkowania w sieciach konwolucyjnych pozwala zredukować liczbę parametrów oraz dodatkowo pozwala wykryć obiekty zajmujące obszary większe niż obszar możliwy do przenalizowania przez filtr. W niniejszej pracy wykorzystano technikę konwolucji kroczącej, która polega na ominięciu wybranej liczby pikseli podczas przesuwania filtra konwolucyjnego.



Rysunek 4.5 Przykład działania autoenkodera konwolucyjnego

W przypadku sieci rekurencyjnych (RNN) bazujące na LSTM łączą w sobie możliwość obsługi danych sekwencyjnych, w tym przechowywania informacji o poprzednich krokach oraz możliwość przechowywania danych z większej ilości kroków. Dodatkowo bazowanie na sieciach LSTM umożliwia lepsze radzenie sobie sieci z znikającym gradientem. Dodatkowo niniejsza sieć posiada możliwość użycia jej jako dwukierunkowej [24], w której każda z warstw posiada drugą warstwę odwrotną, taki wariant sieci nazywany jest dwukierunkową siecią rekurencyjną (ang. *Bidirectional Recurrent Neural Networks*). Działanie takich warstw opiera się na przetwarzaniu sekwencji danych w pierwszej warstwie od początku do końca, a w kolejnej od końca do początku sekwencji danych. Stosowanie sieci dwukierunkowych jest efektywnym rozwiązaniem w problemach analizy sygnałów audio [28]. Po przeprowadzeniu testów modelu na obu wersjach sieci rekurencyjnych zdecydowano się zastosować warstwy dwukierunkowe.

4.5 Implementacja modelu

Zgodnie z założeniem niniejszej pracy zbudowano model sztucznej sieci neuronowej, która bazowała na budowie autoenkodera. W tym celu zostały utworzone 2 grupy sekwencji dzielące się na warstwy enkodujące oraz dekodujące. W pierwszej kolejności zostaną przedstawione warstwy enkodujące.

Grupa warstw enkodujących składała się z 5 warstw konwolucyjnych z rozmiarami filtrów (*kernel size*) wynoszącymi 3 oraz krokiem (ang. *stride*) o wielkości 2. Pomiędzy warstwami konwolucyjnymi zostały zaimplementowane warstwy *Dropout*, które w sposób stochastyczny wyłącza jednostki z procesu uczenia w cyklu, co jest spowodowane zapobieganiem przeuczenia modelu [24]. W trakcie przekształceń danych przez warstwy konwolucyjne został pomniejszony rozmiar z 256 x 128 do wymiaru 8 x 4. Następnie mapa atrybutów utworzona na podstawie ilości filtrów w ostatniej warstwie konwolucyjnej enkodera jest przekształcana do dwuwymiarowej macierzy w celu przetworzenia jej przez warstwę BRNN. Na koniec dane zostały przetworzone przez 2 warstwy ukryte MLP. Zestawienie użytych warstw oraz filtrów zostało przedstawione w tabeli 4.1 oraz implementacja języka python przedstawiona została na rysunku 4.6.

W bibliotece Tensorflow funkcje odpowiadające poszczególnym warstwom to:

- *Conv2D* - dwuwymiarowa warstwa konwolucyjna
- *Dense* - ukryte warstwy MLP
- *LSTM* - warstwy rekurencyjne
- *Bidirectional* - przekształcenie wartstwy na dwukierunkową
- *Conv2DTranspose* - dwuwymiarowa warstwa dekonwolucyjna, wykorzystywna w dekoderze
- *Reshape* - przekształcenie wymiarów danych

Tabela 4.1 Architektura enkodera

Warstwa	Liczba filtrów	Rozmiar filtru	Rozmiar kroku	Funkcja aktywacji
Conv2D	16	(3, 3)	(2, 2)	ReLU
Dropout	0.2			
Conv2D	32	(3, 3)	(2, 2)	ReLU
Dropout	0.2			
Conv2D	64	(3, 3)	(2, 2)	ReLU
Dropout	0.2			
Conv2D	128	(3, 3)	(2, 2)	ReLU
Dropout	0.2			
Conv2D	128	(3, 3)	(2, 2)	ReLU
Reshape	8, 4 * 128			
BLSTM	100 jednostek			
Dense	256 jednostek			
Dense	256 jednostek			

```
enkoder = [
        # 1st Convolutional Layer 256x128 => 128x64
        Conv2D(filters=4, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        Dropout(0.2),
        # 2nd Convolutional Layer 128x64 => 64x32
        Conv2D(filters=8, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        Dropout(0.2),
        # 3nd Convolutional Layer 64x32 => 32x16
        Conv2D(filters=32, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        Dropout(0.2),
        # 4th Convolutional Layer 32x16 => 16x8
        Conv2D(filters=64, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        Dropout(0.2),
        # 5th Convolutional Layer 16x8 => 8x4
        Conv2D(filters=128, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        #Reshape to 2D data
        Reshape(target_shape=(8, 4*128)),
        Bidirectional(LSTM(units=100, activation='relu')),
        Dense(units=256),
        Dense(units=256),
    ]
```

Rysunek 4.6 Implementacja warstw enkodera

W dekoderze dane na wejściu zostają przetworzone przez warstwę MLP, a następnie zostają przekształcone przez warstwę powtarzającego wektora (ang. *Reapet vector*) mająca na celu dostosowanie danych do liczby określonej wymiarów [17] pozwalającej na przetworzenie ich przez warstwy BLSTM. Tak otrzymane dane zostają przetworzone przez warstwę TimeDistributed zawierającą warstwę MLP (T-DENSE), co umożliwia zastosowanie warstwy MLP do każdego kroku w sekwencji danych [18]. Zestawienie parametrów warstw zostało przedstawione w tabeli 4.2 oraz implementacja na rysunku 4.7.

Tabela 4.2 Architektura dekodera

Warstwa	Liczba filtrów	Rozmiar filtru	Rozmiar kroku	Funkcja aktywacji
Dense	256 jednostek			
RepeatVector	8			
BLSTM	100 jednostek			
T-DENSE	4 * 128			
Reshape	8, 4, 128			
Conv2DTranspose	128	(3, 3)	(2, 2)	ReLU
Conv2DTranspose	64	(3, 3)	(2, 2)	ReLU
Conv2DTranspose	32	(3, 3)	(2, 2)	ReLU
Conv2DTranspose	16	(3, 3)	(2, 2)	ReLU
Conv2DTranspose	1	(1, 1)	(1, 1)	sigmoid
Reshape	256, 128, 1			

```
dekoder = [
        Dense(units=256),
        RepeatVector(8),
        Bidirectional(LSTM(units=100, activation='relu', return_sequences=True)),
        TimeDistributed(Dense(4*128)),
        Reshape(target_shape=(8, 4, 128)),
        # 1st Deconvolutional Layer 8x4 => 16x8
        Conv2DTranspose(filters=128, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        # 2st Deconvolutional Layer 16x8 => 32x16
        Conv2DTranspose(filters=64, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        # 3st Deconvolutional Layer 32x16 => 64x32
        Conv2DTranspose(filters=32, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        # 4st Deconvolutional Layer 64x32 => 128x64
        Conv2DTranspose(filters=16, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        # 5st Deconvolutional Layer 128x64 => 256x128
        Conv2DTranspose(filters=8, kernel_size=3, strides=(2, 2), activation='relu', padding='same'),
        Conv2DTranspose(filters=1, kernel_size=1, strides=(1, 1), activation='sigmoid', padding='same'),
        Reshape(target_shape=(256, 128, 1)),
    ]
```

Rysunek 4.7 Implementacja warstw dekodera

W wyniki ewaluacji modelu średni błąd kwadratowy wyniósł 0,0035. Co oznacza, że wartości otrzymane na wyjściu według algorytmu obliczania MSE są zbliżone do wartości wzorcowych. Wyniki wraz z wykresami oraz spektrogramami zostały przedstawione w rozdziale 5.

Do celów przetworzenia otrzymanych spektrogramów melowych na wyjściu na pliki audio wykorzystano funkcję bazujące na bibliotekach librosa [6, 15]. W niniejszej funkcji została wykorzystana odwrócona krótko-czasowa transformata Fouriera (ang. *Inverse Short-Time Fourier Transform*, ISTFT) [47] oraz metoda iteracji Griffin-Lim, która służy

do odzyskiwania sygnału dźwiękowego z widma fazowego sygnału [32]. Dodatkowo spektrogram przed przetworzeniem algorytmu Griffin-lim należało przekształcić z powrotem na wartości decybelowe z zakresu 0 - 1, a następnie przekształcić na wartość amplitudową. Na rysunku 4.8 została przedstawiona implementacja algorytmu Griffin-Lim.

```
degree = np.exp(2j * np.pi * np.random.rand(*x.shape))
_x = np.abs(x).astype(np.cdouble)
y = librosa.istft(_x * degree, hop_length=hparams.hop_length, win_length=hparams.win_length, n_fft=hparams.n_fft)
for _ in range(hparams.griffin_lim):
    faza = np.exp(1j * np.angle(librosa.stft(y, hop_length=hparams.hop_length, win_length=hparams.win_length, n_fft=hparams.n_fft)))
    y = librosa.istft(_x * faza, hop_length=hparams.hop_length, win_length=hparams.win_length, n_fft=hparams.n_fft)
```

Rysunek 4.8 Implementacja algorytmu Griffin-Lim

Zastosowane hiperpametry zostały przedstawione na rysunku 3.7 w rozdziale 3.4

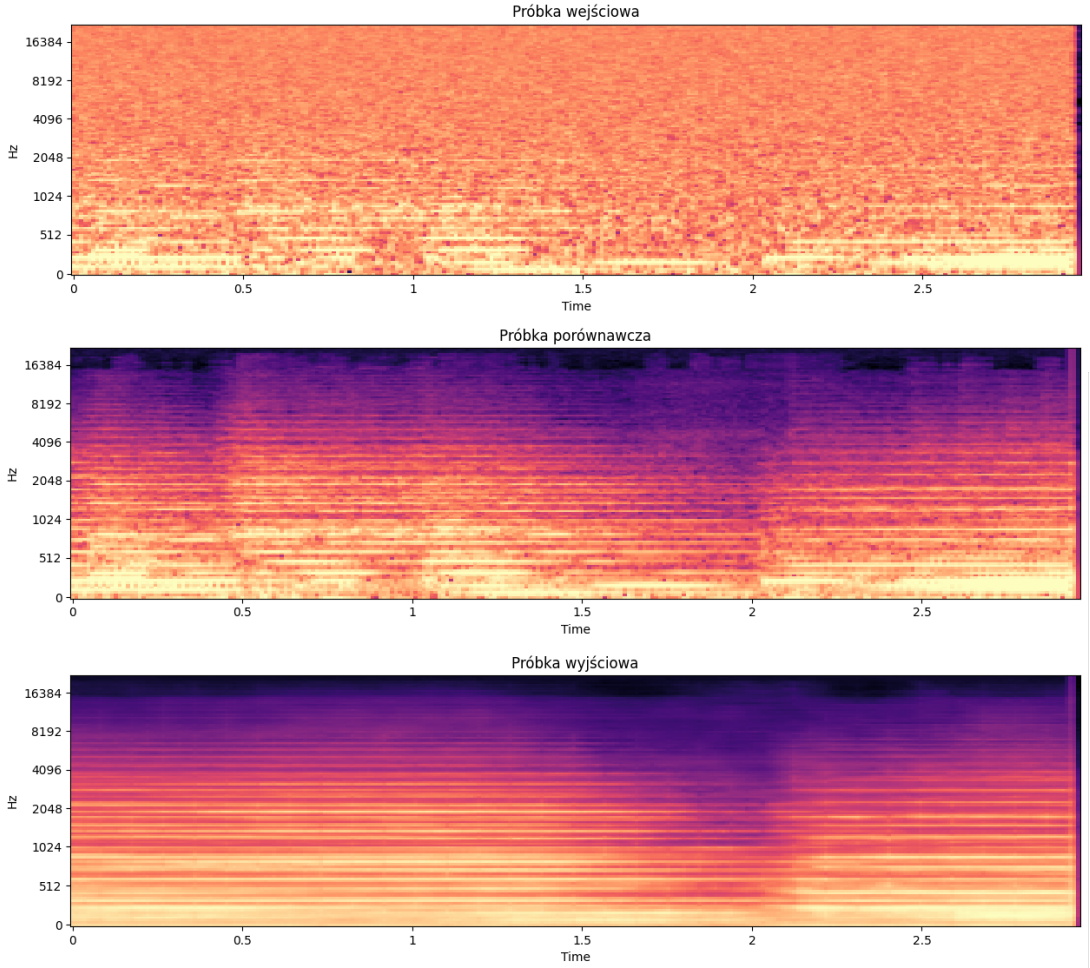
Rozdział 5

Wyniki

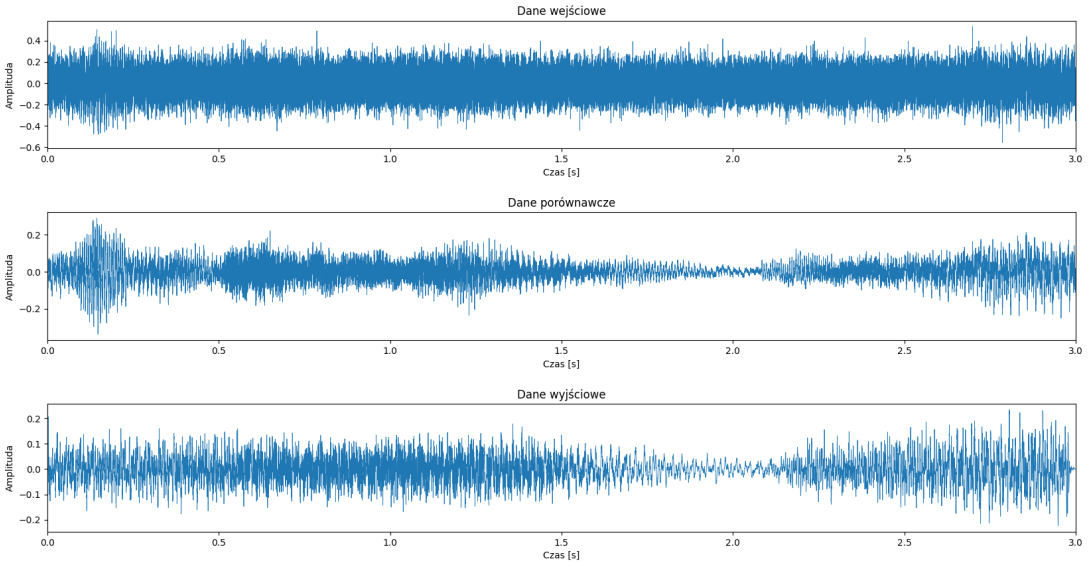
W niniejszym rozdziale zostały opisane otrzymane wyniki dla 4 zniekształceń (szum, brakujące fragmenty danych, zniekształcenia intermodulacyjne oraz flutter). Ewaluacja dla jednego zniekształcenia ze względu na ilość warstw oraz próbek trwała około 6h. Wyniki zostały przeanalizowane na podstawie opisu graficznego w postaci spektrogramu i przebiegu czasowego oraz poprzez odsłuch. Wyniki modelu dla każdego zniekształcenia zostały przedstawione poprzez wskazanie próbki przed przetworzeniem przez model, wskazanie próbki porównawczej oraz próbki po przetworzeniu przez sztuczną sieć neuronową.

5.1 Zniekształcenia szumowe

W przypadku zniekształceń szumowych analiza graficzna (rysunek 5.1 oraz 5.2) wykazuje, że model konwolucyjno-rekurencyjnych sieci neuronowych bardzo dobrze radzi sobie z odszumianiem danych w postaci obrazów [40], którym jest spektrogram. Jednakże w przypadku spektrogramów pomimo zbliżonego wyglądu spektrogramu próbki porównawczej oraz wyjściowej, próbka wyjściowa jest widocznie bardziej wygładzona pod względem pikseli co jest spowodowane dopisywaniem próbek na podstawie próbek mapy obrazu, który został otrzymany po przetworzeniu przez warstwy enkodera [39]. W przypadku analizy odsłuchowej sygnał foniczny jest wyraźnie odszumiony, jednakże w skutek procesu przetwarzania przez model oraz algorytmu Griffin-Lim, w celu inwersji spektrogramu na plik audio, sygnał jest zniekształcony, przez co słychać wyraźny brum. Prawdopodobnie dalsze zwiększanie liczby iteracji algorytmu Griffin-Lim mogłoby częściowo zredukować ten problem. Kolejnym rozwiązaniem na występowanie tego zniekształcenia to próba dobrania innych warstw oraz filtrów.



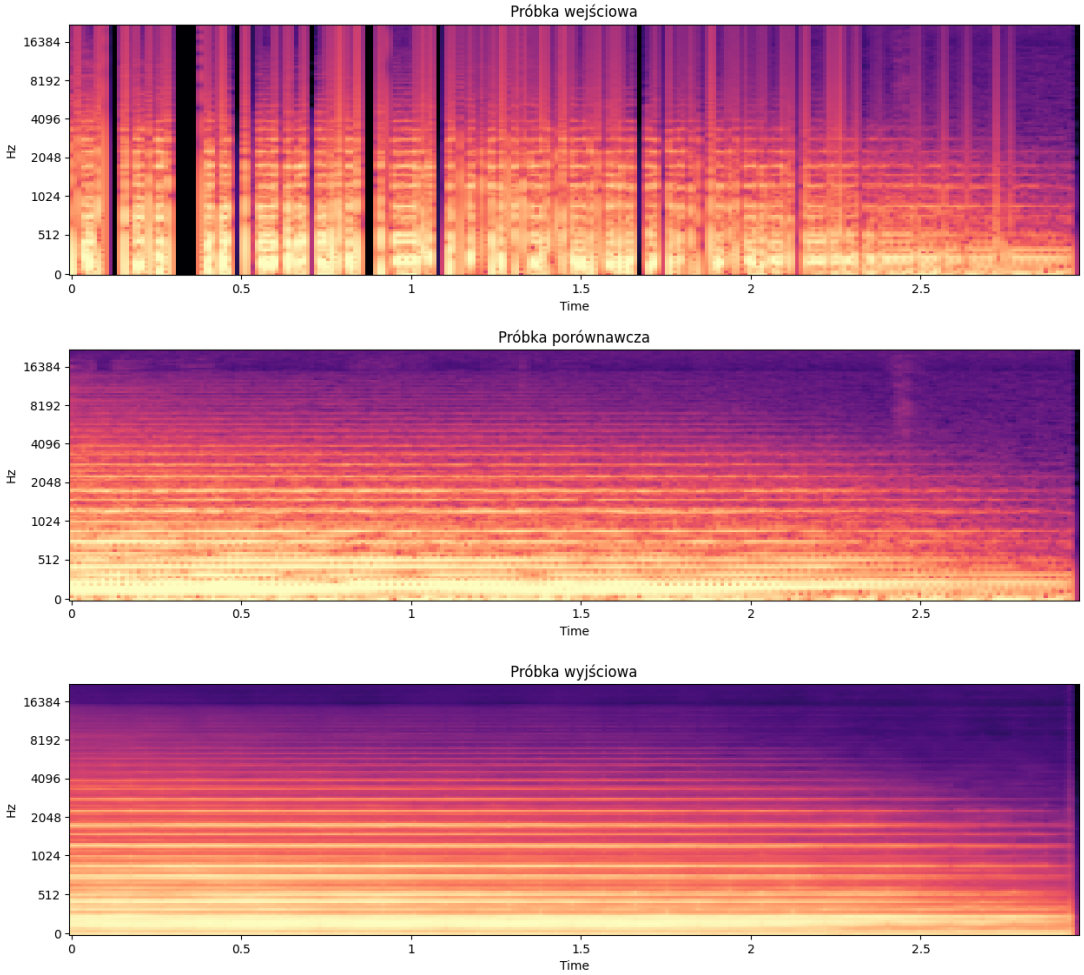
Rysunek 5.1 Próbka z zniekształceniem w postaci szumu



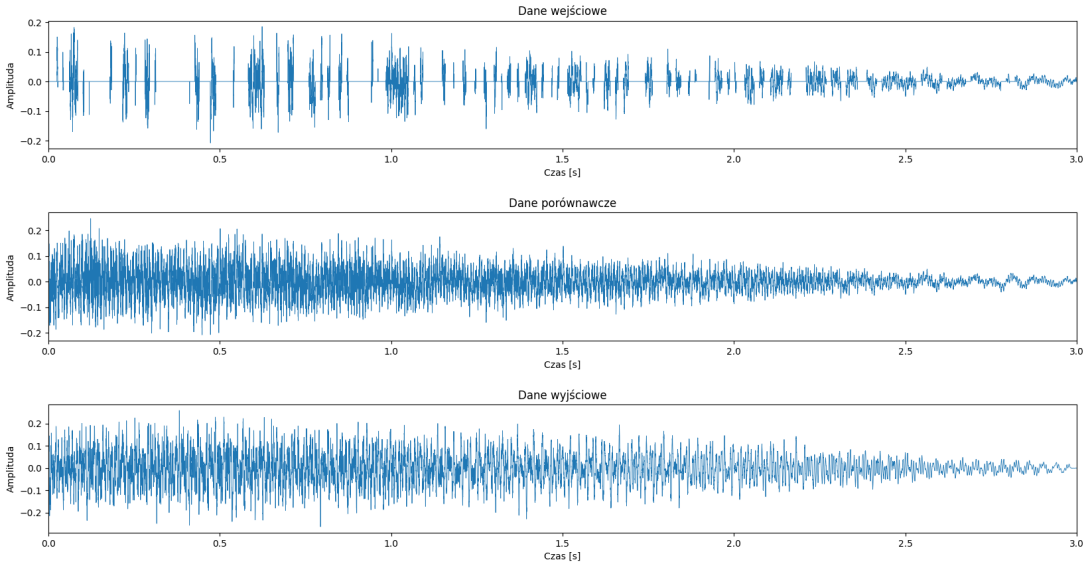
Rysunek 5.2 Próbka z zniekształceniem w postaci szumu

5.2 Brakujące dane

Konwolucyjno-rekurencyjne sieci radzą sobie bardzo dobrze z rekonstruowaniem brakujących danych [24] co wskazują analizy graficzne próbek z brakującymi danymi (rysunek 5.3 oraz 5.4). Przerwy w danych zostały wypełnione i w dużym stopniu pokrywały się z plikiem porównawczym, jednakże podobnie tak jak w przypadku zniekształceń szumowych sygnał z zrekonstruowanymi brakującymi danymi posiada zniekształcenia w postaci brumu.



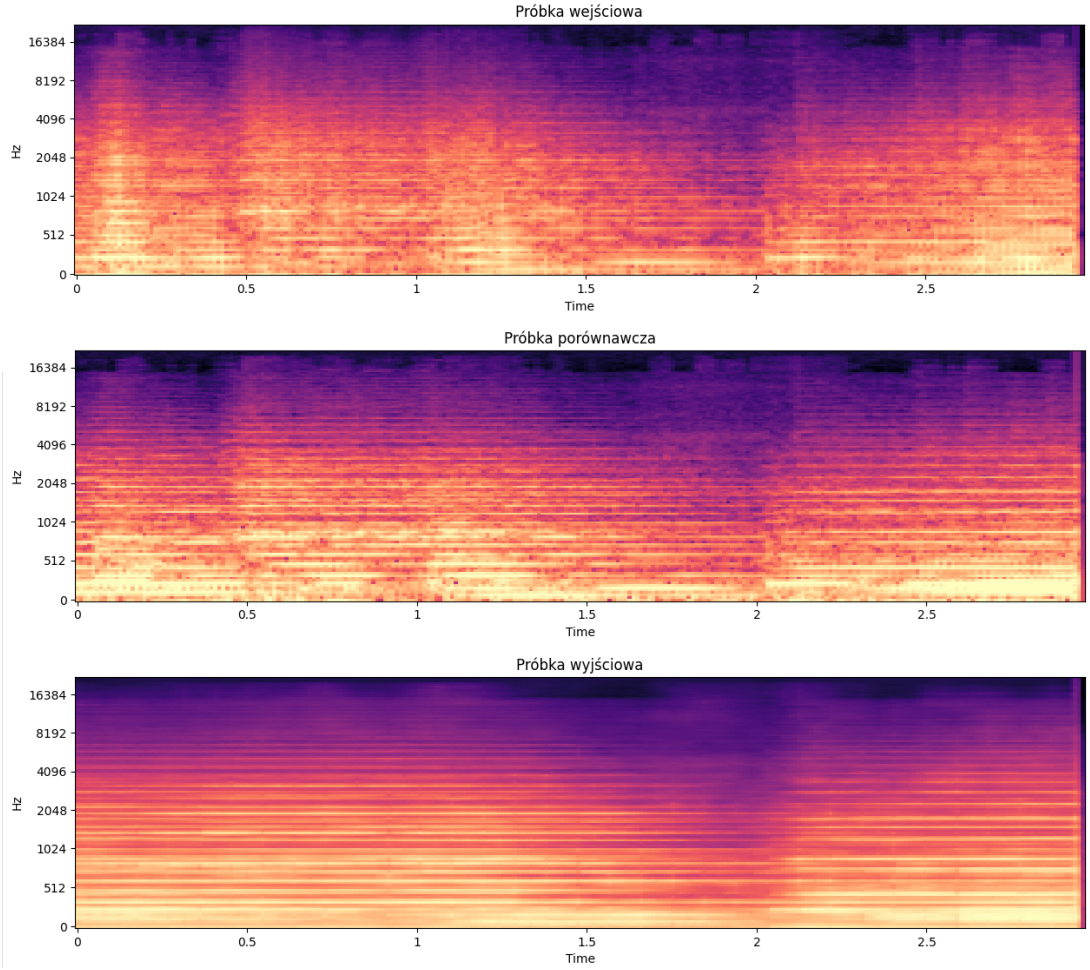
Rysunek 5.3 Próbka z zniekształceniem w postaci brakujących danych



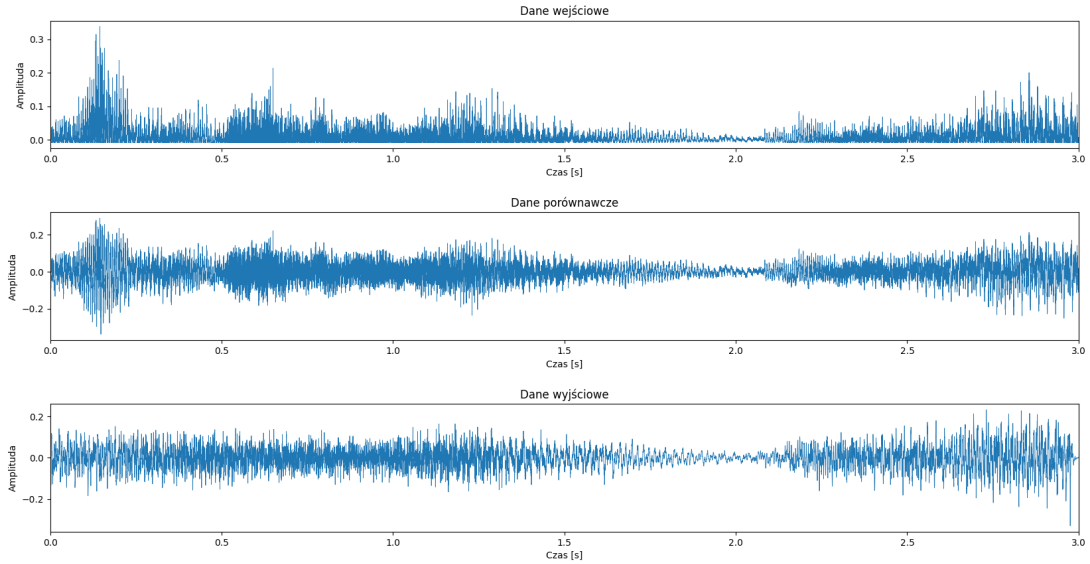
Rysunek 5.4 Próbka z zniekształceniem w postaci brakujących danych

5.3 Zniekształcenia intermodulacyjne

Graficzna analiza spektrogramów oraz przebiegów czasowych próbek, które posiadały zniekształcenia intermodulacyjne, wskazuje że w przypadku spektrogramów nie są widoczne duże różnice, co jest spowodowane typem zniekształcenia, jednakże w przypadku przebiegów czasowych model konwolucyjnej sieci neuronowej zrekonstruował częściowo sygnał. Na graficznym przedstawieniu wyników można zauważyć, że niniejszy model nie radzi sobie z wzrostami głośności, jednakże niniejszy problem można rozwiązać poprzez wybranie innych parametrów sztucznej sieci neuronowej, w tym dobranie innych rodzajów warstw. Dodatkowo sygnał posiada typ zniekształceń, który pojawiał się w poprzednich typach rekonstrukcji zniekształceń sygnału fonicznego.



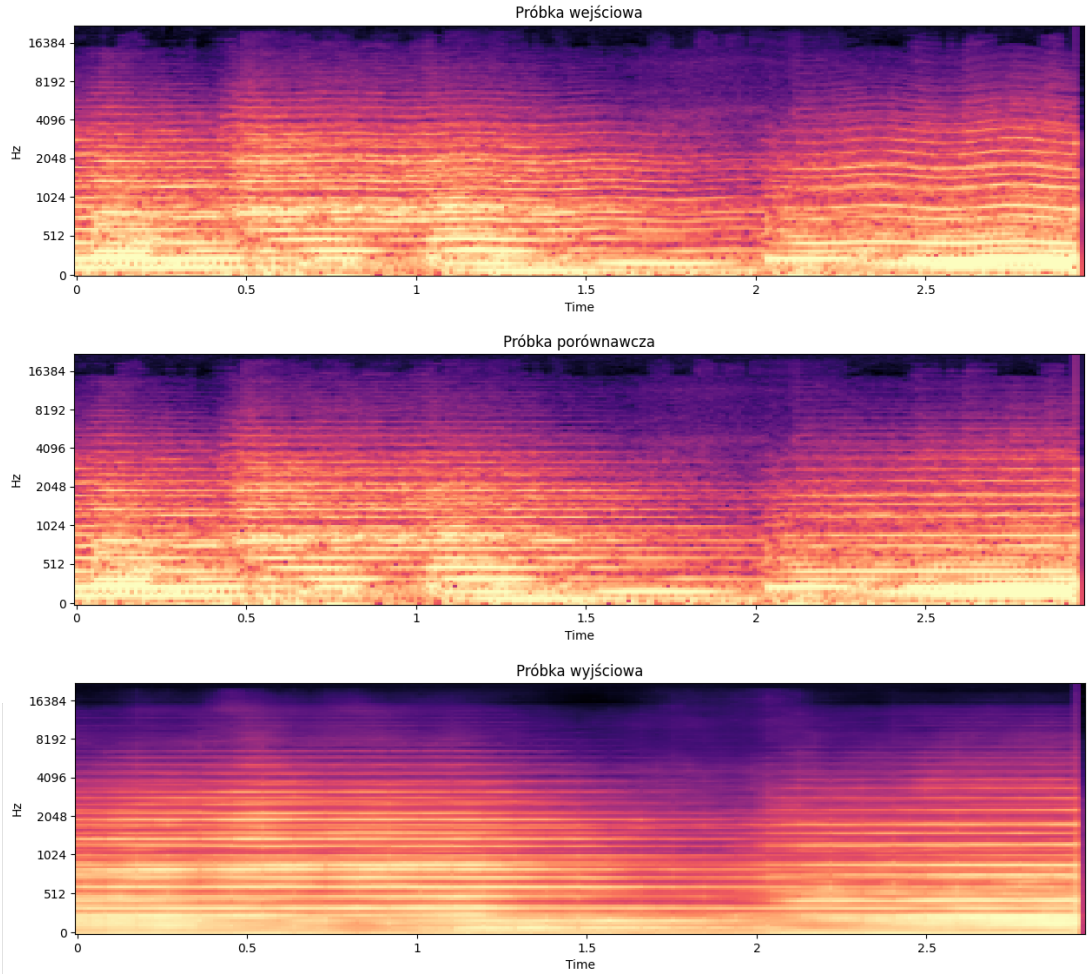
Rysunek 5.5 Próbka z zniekształceniami intermodulacyjnymi



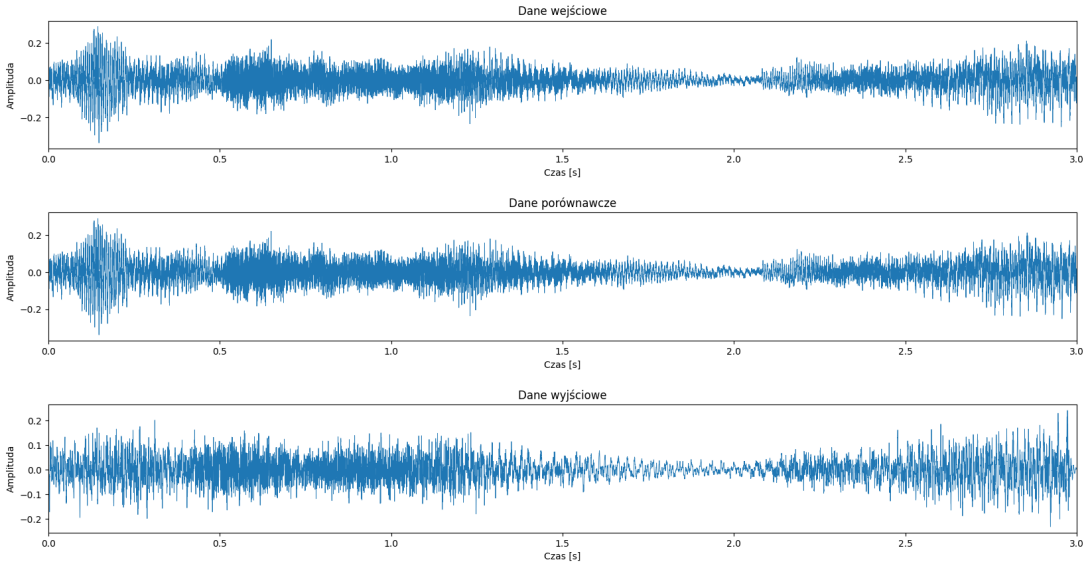
Rysunek 5.6 Próbka z zniekształceniami intermodulacyjnymi

5.4 Zniekształcenia flutter

Ostatnim zniekształceniem, które zostało użyte w procesie nauki modelu w celu rekonstrukcji sygnału fonicznego jest zniekształcenie typu flutter. Analiza graficzna wskazuje że model nie radził sobie dobrze z tego typu zniekształceniami, co może być spowodowane małymi różnicami w postaci danych liczbowych oraz graficznych (spektrogram), przez co model nie odnajdywał w sposób poprawny wzorców w celu rekonstrukcji danych. W przypadku odsłuchu sygnału zrekonstruowanego, zniekształcenia pojawiające się wskutek przekształceń spektrogramu melowego na format audio, które pojawiają się w poprzednich analizach, spowodowały brak możliwości wykrycia poprawy względem redukcji zniekształcenia typu flutter.



Rysunek 5.7 Próbka z zniekształceniami typu flutter



Rysunek 5.8 Próbka z zniekształceniami typu flutter

5.5 Podsumowanie

Wyniki przeprowadzonej analizy na modelu konwolucyjno-rekurencyjnej sieci neuronowej wskazują iż wyniki otrzymane po przetworzeniu przez wyuczoną sieć neuronową są zależne od typu problemu z jakimi model ma do czynienia. W przypadku rekonstrukcji brakujących danych oraz usuwania szumu model radził sobie na zadowalającym poziomie, jednakże w przypadku rekonstrukcji sygnału fonicznego zawierającego zniekształcenia intermodulacyjne lub flutter, należałoby wykorzystać model z innym doborem hiperparametrów oraz warstw. Dodatkowo inwersja spektrogramu do formatu audio wprowadza zniekształcenia, które mógłby zostać zniwelowane poprzez wykorzystanie większej ilości iteracji algorytmu Griffin-Lim, co spowodowałoby większe zużycie pamięci, co za tym idzie większych kosztów sprzętowych.

Rozdział 6

Konkluzja

W niniejszej pracy został zbudowany model sieci neuronowej bazującej na sieciach konwolucyjnych oraz rekurencyjnych dwukierunkowych. Do celów treningowych została stworzona baza próbek składająca się z sygnałów bez zniekształceń, które równocześnie były sygnałami wzorcowymi oraz próbek sygnałów z podziałem na typ zniekształcenia, tj. szum, brakujące dane, zniekształcenia intermodulacyjne oraz flutter. W dalszej kolejności zostały przedstawione teoretyczne aspekty sieci neuronowych oraz hiperparametrów, które pozwalają na konfigurowanie procesu uczenia się sieci neuronowej. Następnie została opisana budowa modelu, w tym wykorzystane warstwy oraz ich parametry.

Zaproponowany model jest modelem eksperymentalnym, który poprzez dalsze badanie w dziedzinie wpływu algorytmów sztucznych inteligencji na sygnały foniczne, mogłoby rozwiązać problem powstającego brumu oraz pozwolić lepiej przetwarzać wzorce sygnałów dźwiękowych.

Przedstawiony projekt inspirowany był rosnącą popularnością wykorzystania algorytmów sztucznej inteligencji w wielu dziedzinach, takich jak: rozpoznawanie mowy, rekonstrukcja obrazu, modele generujące treści itp.. W tym wypadku przedstawione zostało wykorzystanie algorytmów do rekonstrukcji sygnałów fonicznych, jednakże temat nie został w pełni wyczerpany, a sam model oraz temat ma wiele kierunków rozwoju, które pozwoliłby na bardziej zautomatyzowane przetwarzanie sygnałów fonicznych.

Bibliografia

[1] Convert a power-spectrogram to decibel units in tensorflow. https://gist.github.com/dschwertfeger/f9746bc62871c736e47d5ec3ff4230f7. Dostęp: 2023-12-16.

[2] Google cloud platform. https://en.wikipedia.org/wiki/Google_Cloud_Platform. Dostęp: 2023-12-12.

[3] Ieee xplore - top searches and popular content. https://ieeexplore.ieee.org/popular/all. Dostęp: 2023-12-12.

[4] Librosa. https://librosa.org/doc/latest/index.html. Dostęp: 2023-12-20.

[5] librosa/core/spectrum.py. https://github.com/librosa/librosa/blob/main/librosa/core/spectrum.py#L2744. Dostęp: 2023-12-16.

[6] librosa.feature.inverse.mel_to_stft. https://librosa.org/doc/main/generated/librosa.feature.inverse.mel_to_stft.html#librosa.feature.inverse.mel_to_stft. Dostęp: 2023-12-2.

[7] librosa.mel_frequencies. https://librosa.org/doc/main/generated/librosa.mel_frequencies.html. Dostęp: 2023-12-2.

[8] Matplotlib. https://matplotlib.org. Dostęp: 2023-12-01.

[9] Numpy. https://numpy.org. Dostęp: 2023-12-01.

[10] Pixelplayer. http://sound-of-pixels.csail.mit.edu/. Dostęp: 2023-12-20.

[11] scipy.io.wavfile.read. https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.read.html. Dostęp: 2023-12-14.

[12] scipy.io.wavfile.write. https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.write.html. Dostęp: 2023-12-14.

[13] Secure shell. https://pl.wikipedia.org/wiki/Secure_Shell. Dostęp: 2023-12-12.

[14] Single-precision floating-point format. https://en.wikipedia.org/wiki/Single-precision_floating-point_format. Dostęp: 2023-12-14.

[15] Spectrogramming, mel scaling, and inversion in tensorflow. https://colab.research.google.com/github/timsainb/tensorflow2-generative-models/blob/master/7.0-Tensorflow-spectrograms-and-inversion.ipynb. Dostęp: 2023-12-1.

[16] Tensorflow. https://pl.wikipedia.org/wiki/TensorFlow. Dostęp: 2023-12-01.

[17] tf.keras.layers.repeatvector. `https://www.tensorflow.org/api_docs/python/tf/keras/layers/RepeatVector`. Dostęp: 2023-12-18.

[18] tf.keras.layers.timedistributed. `https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed`. Dostęp: 2023-12-18.

[19] tf.keras.optimizers.adam. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam`. Dostęp: 2023-12-18.

[20] tf.signal.linear_to_mel_weight_matrix. `https://www.tensorflow.org/api_docs/python/tf/signal/linear_to_mel_weight_matrix`. Dostęp: 2023-12-1.

[21] Visual studio code dokumentacja. `https://code.visualstudio.com/docs`. Dostęp: 2023-12-03.

[22] Wav. `https://en.wikipedia.org/wiki/WAV`. Dostęp: 2023-12-13.

[23] Wave audio file format. `https://www.loc.gov/preservation/digital/formats/fdd/fdd000001.shtml`. Dostęp: 2023-12-13.

[24] C. C. Aggarwal. *Neural Networks and Deep Learning*. Springer Cham, 2018.

[25] T. Hodson. Root-mean-square error (rmse) or mean absolute error (mae): when to use them or not. *Geoscientific Model Development*, 15(14):5481 – 5487, 2022.

[26] O. P. Jena, A. R. Tripathy, A. A. Elngar, Z. Polkowski. *Computational Intelligence and Healthcare Informatics*. Wiley-Scrivener, 2021.

[27] F. Kurp. *Sztuczna inteligencja od podstaw*. Helion, 2023.

[28] Y. Li, X. Li, Y. Zhang, W. Wang, M. Liu, X. Feng. Acoustic scene classification using deep audio feature and blstm network. *2018 International Conference on Audio, Language and Image Processing (ICALIP)*, strony 371–374, 2018.

[29] Y.-F. Li, L.-Z. Guo, Z.-H. Zhou. Towards safe weakly supervised learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(1):334–346, 2021.

[30] Y. Liu, Hayden. *Python Machine Learning by Example: Build intelligent systems using Python, Tensorflow 2, PyTorch, and scikit-learn*. Helion, 2020.

[31] M. Lutz. *Python Wprowadzenie*. Helion, 2011.

[32] Y. Masuyama, K. Yatabe, Y. Koizumi, Y. Oikawa, N. Harada. Deep griffin–lim iteration. *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, strony 61–65, 2019.

[33] W. McKinney. *Python for Data Analysis*. O'Reilly, 2017.

[34] X. Min, G. Zhai, J. Zhou, M. C. Q. Farias, A. C. Bovik. Study of subjective and objective quality assessment of audio-visual signals. *IEEE Transactions on Image Processing*, 29:6054–6068, 2020.

[35] K. Organiściak. Automatyczna klasyfikacja wybranych zniekształceń sygnałów muzycznych z wykorzystaniem konwolucyjno-rekurencyjnych sieci neuronowych bez porównania do sygnału referencyjnego, 2023.

[36] D. P. Pau, T. A. Naramo, M. Dimbiniaina. Coding mel spectrogram using keras and tensorflow for home appliances tiny classification. *2023 IEEE International Conference on Consumer Electronics (ICCE)*, strony 1–5, 2023.

[37] S. Płaczek, A. Płaczek. Uczenie wielowarstwowych szerokich sieci neuronowych z funkcjami aktywacji typu relu w zadaniach klasyfikacji. *Poznan University of Technology Academic Journals. Electrical Engineering*, No. 96:47–58, 2018.

[38] S. Raschka, V. Mirjalili. *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and Tensorflow*. Helion, 2017.

[39] F. H. Raswa, F. Halberd, A. Harjoko, Wahyono, C.-T. Lee, Y.-H. Li, J. C. Wang. Multi-loss function in robust convolutional autoencoder for reconstruction low-quality fingerprint image. *2022 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, strony 428–431, 2022.

[40] G. Rodewald. Autoenkodery. podstawy budowy wydajnych modeli uczenia maszynowego. Raport instytutowy, Warszawska Wyższa Szkoła Informatyki, 2022.

[41] R. Singh. *Profiling Humans from their Voice*. Springer Singapore, 2019.

[42] M. Torcoli, T. Kastner, J. Herre. Objective measures of perceptual audio quality reviewed: An evaluation of their application domain dependence. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:1530–1541, 2021.

[43] T. Virtanen, M. Plumblej, D., D. Ellis. *Computational Analysis of Sound Scenes and Events*. Springer, 2018.

[44] Wikipedia contributors. Google assistant — Wikipedia, the free encyclopedia, 2023. [Online; accessed 20-December-2023].

[45] Wikipedia contributors. Shazam (application) — Wikipedia, the free encyclopedia, 2023. [Online; accessed 20-December-2023].

[46] Wikipedia contributors. Siri — Wikipedia, the free encyclopedia, 2023. [Online; accessed 20-December-2023].

[47] H. ZHIVOMIROV. On the development of stft-analysis and istft-synthesis routines and their practical implementation. *TEM Journal*, 8(1):56 – 64, 2019.