FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Potlačování ruchů v hudebních nahrávkách pomocí neuronových sítí

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie
Autor práce: **Bc. Miloš Fejgl**
Vedoucí práce: prof. Ing. Zbyněk Koldovský, Ph.D.

Liberec 2024

Tento list nahraďte
originálem zadání.

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.
Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.
Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.
Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.
Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.
Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.
Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

6. 1. 2025 Bc. Miloš Fejgl

Potlačování ruchů v hudebních nahrávkách pomocí neuronových sítí

Abstrakt

Tato diplomová práce se zabývá problematikou redukce ruchů v hudebních nahrávkách a vývojem VST (Virtual Studio Technology) pluginu, který tuto redukci umožní v reálném čase v libovolném DAW (Digital Audio Workstation, softwarový nástroj pro produkci a editaci hudebních nahrávek). Zvolenou metodikou pro tuto úlohu bylo využití strojového učení a neuronových sítí. Z toho důvodu je součástí mé práce i tvorba datové sady pro trénování a testování modelů neuronových sítí. Cílem výsledného VST pluginu je poskytnout audioinženýrům nástroj, díky kterému budou moci nahrávat vokály i mimo profesionální studia, aniž by tím příliš utrpěla kvalita výsledné nahrávky.

Klíčová slova: redukce ruchů, strojové učení, neuronové sítě, hudební nahrávky, VST plugin, vokály, DSP

Noise reduction in musical recordings using neural networks

Abstract

This thesis addresses the issue of noise reduction in music recordings and the development of a VST (Virtual Studio Technology) plugin that enables real-time noise reduction in any DAW (Digital Audio Workstation, a software tool for music production and editing). The chosen methodology for this task involves the use of machine learning and neural networks. Consequently, part of this work includes the creation of a dataset for training and testing the neural network models. The goal of the resulting VST plugin is to provide audio engineers with a tool that allows them to record vocals outside professional studios without noticeable compromises in the quality of the final recording.

Keywords: noise reduction, machine learning, neural networks, musical recordings, VST plugin, vocals, DSP

Poděkování

Chtěl bych poděkovat prof. Ing. Zbyňku Koldovksému, Ph.D. za vedení této diplomové práce a za jeho cenné rady. Dále pak také panu Vojtěchu Meluzínovi ze společnosti MeldaProduction za inspiraci na vybrané téma.

Obsah

Seznam zkratek . . . 10

1 Úvod **11**

2 Problematika potlačování ruchů **13**
2.1 Ruch . . . 13
2.2 Druhy ruchů . . . 13
2.2.1 Ambientní ruch . . . 14
2.2.2 Reverberace . . . 14
2.2.3 Elektromagnetický šum . . . 14
2.2.4 Impulzní a tranzientní ruch . . . 14
2.2.5 Zkreslení . . . 15
2.2.6 Výpadky . . . 15
2.3 Redukce ruchů . . . 15
2.4 Metriky kvality redukce . . . 16
2.4.1 SNR, SI-SNR a SI-SNRi . . . 16
2.4.2 SDR a SDRi . . . 17
2.4.3 PESQ . . . 17

3 Potlačování ruchů pomocí neuronových sítí **18**
3.1 Architektury pro separaci zdrojů . . . 18
3.1.1 TasNet . . . 18
3.1.2 Conv-TasNet . . . 19
3.1.3 U-Net a Wave-U-Net . . . 20
3.1.4 Demucs . . . 21
3.1.5 Band Split RoFormer . . . 22
3.1.6 Mel-Band RoFormer . . . 22
3.1.7 Spleeter . . . 23
3.2 Architektury pro redukci šumů . . . 24
3.2.1 RNNoise . . . 24
3.2.2 Spiking-FullSubNet . . . 25

4 Datová sada **26**
4.1 MUSDB18 . . . 27
4.2 Externí ruchy . . . 27
4.3 Augmentace datové sady . . . 28

4.3.1 Přidání clicktracku . . . 28
4.3.2 Přidání dynamického šumu . . . 29
4.3.3 Přidání statického šumu . . . 29
4.3.4 Přidání náhodných tónů . . . 29
4.3.5 Lehké zkreslení . . . 30
4.3.6 Mix čistého vokálu a ruchu . . . 30
4.3.7 Normalizace . . . 30
4.4 Finální datová sada . . . 31

5 Vytváření modelů neuronové sítě **33**
5.1 Požadavky . . . 33
5.2 Vývoj modelů . . . 34
5.2.1 Neúspěšný vývoj vlastní neuronové sítě . . . 34
5.2.2 Úspěšné řešení - Fine-Tuning modelů architektury Demucs . . . 35
5.2.3 Trénování . . . 37
5.2.4 Testování . . . 41
5.2.5 Export . . . 42

6 VST plugin **43**
6.1 Seznámení s využitými nástroji pro vývoj . . . 43
6.1.1 JUCE framework . . . 43
6.1.2 TorchScript . . . 44
6.1.3 CMake . . . 45
6.2 Vývoj VST pluginu . . . 46
6.2.1 Sestavení XCode projektu . . . 46
6.2.2 Programování VST pluginu . . . 47
6.2.3 Sestavení VST . . . 48

7 Výsledný VST plugin CleanWave **49**
7.1 Uživatelské rozhraní . . . 49
7.1.1 Design, rozvržení a funkcionalita . . . 50
7.2 Zpracování zvuku . . . 51
7.2.1 Neuronová síť . . . 52
7.2.2 Gate . . . 54
7.2.3 Gain . . . 55

8 Testování **56**
8.1 Testování modelů . . . 56
8.2 Testování pluginu CleanWave . . . 59

9 Závěr **61**

Seznam obrázků

2.1 PESQ diagram [10] 17

3.1 Architektura TasNet [12] 19
3.2 Architektura Conv-TasNet [13] 20
3.3 Architektura Wave-U-Net [15] 21
3.4 Architektura Demucs [16] 21
3.5 Architektura Band Split RoFormer [18] 22
3.6 Architektura RNNoise [21] 24
3.7 Porovnání ANN a SNN [22] 25
3.8 Architektura Spiking-FullSubNet [23] 25

5.1 Potlačování clicku vlastním modelem NN 35
5.2 Modifikovaná architektura Demucs [28] 36
5.3 Ukázka průběhu tréninku modelu 39

7.1 VST plugin CleanWave 50
7.2 Diagram zpracování audia 51

8.1 Graf redukce samotného šumu pomocí nejúspěšnějšího modelu 57
8.2 Graf redukce šumu s vokály pomocí nejúspěšnějšího modelu 58
8.3 Graf redukce šumu z celé nahrávky pomocí nejúspěšnějšího modelu . 58
8.4 Graf redukce šumu pomocí jiného modelu 59

Seznam tabulek

4.1 Přehled datové sady 31
4.2 Pravděpodobnosti výskytu ruchů v různých verzích datové sady . . . 32

5.1 Varianty předtrénovaných modelů knihovny *denoiser* 37
5.2 Výsledky testů nejlepších modelů 41

Seznam zkratek

AAX	avid audio extension
Adam	Adaptive Moment Estimation
ANN	artificial neural network
API	application programming interface
AU	audio unit
CNN	konvoluční neuronová síť
CPU	central processing unit
CUDA	compute unified device architecture
DAW	digital audio workstation
DNN	hluboká neuronová síť
DSP	digital signal processing
GSN	gated spike neural model
GRU	gated recurrent unit
GPU	graphics processing unit
GUI	graphical user interface
IDE	integrated development environment
ISTFT	inverse short-time Fourier transform
JSON	JavaScript Object Notation
JIT	just-in-time
LSTM	long short-term memory
MFCC	Mel-frequency cepstrum
MIDI	musical instrument digital interface
MPS	metal performance shaders
MUSDB18	music separation database
MUSDB18-HQ	music separation database - high quality
NN	neuronová síť
RNN	rekurentní neuronová síť
RoPE	rotary position embedding
SDR	signal distortion ratio
SDRi	signal distortion ratio improvement
SiSEC	signal separation evaluation campaign
SNN	spiking neural network
SNR	signal-to-noise ratio
SI-SNR	scale invariant signal-to-noise ratio
SI-SNRi	scale invariant signal-to-noise ratio improvement
SGD	Stochastic Gradient Descent
STFT	short-time Fourier transform
VST	virtual studio technology
VST3	virtual studio technology 3
WAV	waveform audio file format

1 Úvod

V této diplomové práci jsem se rozhodl zkombinovat mé dvě hlavní oblasti zájmů, programování a hudbu. Jednou z mnoha problematik, které v průniku těchto oblastí můžeme řešit, je redukce ruchů v hudebních nahrávkách. Jelikož jsou ruchy poměrně častým jevem, se kterým se běžně setkávám i osobně při tvorbě hudby, rozhodl jsem se zvolit si jejich redukci jako ústřední téma mé diplomové práce.

Pořídit kvalitní nahrávku, která by nebyla ovlivněna žádnými ruchy, je poměrně náročné. Studiové mikrofony jsou obecně velice citlivá zařízení zachycující téměř vše, co se okolo nich děje. S ruchy se musejí vypořádávat jak amatérští, tak profesionální audioinženýři v kvalitních, dobře navržených studiích.

Nejčastěji tento problém postihuje právě nahrávky vokálů. Ty jsou svojí energií v porovnání například s elektrickou kytarou, basou, či bicími nejslabší, a proto také nejvíce náchylné pro znatelné působení ruchů.

Mým cílem tedy bylo vytvořit software, který by uměl různé ruchy z nahrávek odfiltrovat. Proto jsem se rozhodl v rámci své diplomové práce vytvořit vlastní VST plugin, který pomocí neuronové sítě detekuje šum a odstraní ho, aniž by znatelně narušil kvalitu dané nahrávky.

VST (Virtual Studio Technology) pluginy jsou dnes naprosto nepostradatelným nástrojem prakticky každého audioinženýra. Jedná se o komponentu pro zpracování zvuku, která se používá v rámci hostitelské aplikace, nejčastěji nějakého DAW [1]. DAW (Digital Audio Workstation) jsou softwarové programy pro produkci hudby, které obsahují komplexní sadu nástrojů pro nahrávání, editaci a produkci digitálního zvuku. Díky své schopnosti fungovat jako kompletní nahrávací studio se DAW staly základním nástrojem pro hudebníky, producenty a zvukaře všech úrovní [2]. VST pluginů pro tyto DAW dnes existuje velké množství a plní různé funkce - můžeme se setkat s ekvalizátory, reverby, chorusy, syntetizátory, s emulacemi různých zesilovačů a mnoha dalšími. Všechny mají však jedno společné: musejí fungovat v reálném čase. To může být v kombinaci s využitím neuronové sítě v některých případech výzva.

Šumy a přeslechy se řeší už od počátků zpracování digitálních signálů. Z toho důvodu se také pro redukci šumu v průběhu času vyvinulo mnoho různých technik, které dosahovaly poměrně uspokojivých výsledků. V současnosti jsme ale stále vprostřed významného rozmachu neuronových sítí, které nabízejí nová řešení problémů v téměř každém odvětví. Proto není překvapením, že se pomocí nich začala řešit i redukce šumu. Podobně jako u jiných odvětví, i zde neuronové sítě dosahují dříve nepředstavitelných výsledků a překonávají tak všechny dříve běžně užívané metody.

VST pro tuto problematiku už dnes samozřejmě existují, například Clarity VX společnosti Wave [3] nebo Clear Voice Separator společnosti Supertone [4]. Není

jich však mnoho a nové, stále lepší architektury neuronových sítí každým rokem přibývají. Výše uvedená VST si také nedokáží poradit s každým druhem šumu. Proto je mým cílem vytvořit moderní plugin, který v některých ohledech předčí výše zmíněné state-of-the-art pluginy pro redukci ruchů a poskytnout tak uživatelům spolehlivý a dostupný nástroj pro nahrávání v neideálních podmínkách. Tak vznikl můj VST plugin CleanWave.

2 Problematika potlačování ruchů

2.1 Ruch

Ruch můžeme definovat jako nechtěnou modifikaci signálu, která svým působením zhoršuje jeho kvalitu. Takováto modifikace může nastat buď při pořizování daného signálu (například snímáním externích zdrojů), při jeho přenosu (například působením elektromagnetického pole), nebo také při jeho zpracování (například kvantizací).

Matematicky si ruch můžeme definovat jako aditivní zkreslení definované v časové doméně podle rovnice:

$$y(t) = x(t) + n(t),$$

kde $x(t)$ je průběh zdrojového signálu pro časový index t z množiny reálných čísel, $n(t)$ je šumový signál a $y(t)$ je šumem zatížený signál zachycený mikrofonem [5].

Většina nahrávek je také prováděna v prostředí s dozvukem. Signály zatížené šumem i dozvukem lze vyjádřit jako:

$$y(t) = h(t) * x(t) + n(t),$$

kde $h(t)$ představuje akustickou odezvu místnosti [5].

Ruchy mohou vznikat také zkreslením signálu nebo jeho kvantizací, tedy působením určité nelineární funkce. Signál zatížený šumem, dozvukem a nelinearitou lze vyjádřit jako:

$$y(t) = f(h(t) * x(t) + n(t)),$$

kde funkce f vyjadřuje libovolnou nelineární funkci. V případě zkreslení se může jednat například o funkci tzv. klipování, saturace a harmonického či exponenciálního zkreslení, v případě kvantizace se pak jedná o funkce logaritmické a funkci $sign$.

2.2 Druhy ruchů

Z uvedené definice je tedy jasné, že šumy a ruchy mohou nabývat mnoha různých podob. Dle jejich povahy a původu je můžeme rozdělit například do následujících kategorií: ambientní ruchy, elektromagnetické ruchy, impluzní a tranzientní ruchy, reverberace, zkreslení a výpady. V následujících podkapitolách si každou z těchto kategorií dále rozebereme.

2.2.1 Ambientní ruch

Ambientní ruch je ruch, který vzniká působením jakéhokoliv jiného zdroje než zdroje žádaného a který je zároveň čistě akustické povahy. Může se tedy jednat například o zvuk projíždějícího auta, lidskou mluvu, nebo také šum zapnuté klimatizace či větráku. Obecně se však jako ambientní ruchy označují především ruchy s dobou trvání nad 50 ms. Kratší ruchy akustického původu častěji označujeme jako impluzní či tranzientní.

2.2.2 Reverberace

Reverberace je ruch způsobený akustickými vlastnostmi prostředí, ve kterém nahrávání probíhá. Na rozdíl od ambientního ruchu však uvažujeme pouze jeden zdroj signálu, tedy signál žádaný. Ruch zvaný reverberace pak vzniká odrazem tohoto zdrojového signálu od stěn či jiných objektů v nahrávací místnosti.

2.2.3 Elektromagnetický šum

Elektromagnetický šum způsobuje téměř každé elektronické zařízení, které generuje, spotřebovává, nebo přenáší energii. Intenzita tohoto ruchu roste s napětím, proudem, či fyzickou blízkostí daného zařízení. Mezi typické zdroje elektromagnetického rušení patří transformátory, vysílače, mobilní telefony, mikrovlnné přenosy, elektrické rozvody, motory, generátory, relé, oscilátory, zářivky a elektrické bouře [6].

Tyto ruchy lze rozdělit na elektrostatické, které jsou generovány přítomností napětí, a magnetické, vznikající proudem nebo permanentním magnetismem. Oba typy mohou vznikat současně, což ztěžuje jejich redukci. Magnetická pole indukují napětí ve vodičích při interakci s magnetickými siločarami, například v okolí střídavých elektrických rozvodů, kde se pole rozpíná a smršťuje, nebo při pohybu vodiče Zemským magnetickým polem. Tyto vlastnosti elektromagnetického ruchu představují významnou výzvu při jeho redukci, zejména kvůli jeho různorodým zdrojům a komplexnímu charakteru [6].

2.2.4 Impulzní a tranzientní ruch

Impulzní a tranzietní ruchy jsou svou povahou velice podobné. Impulzní ruch tvoří krátké impulzy s vysokou amplitudou, které trvají jen několik ms, obvykle tedy do 3 ms. Mohou být způsobeny spínacím rušením, špatným stavem kanálu nebo fyzickým poškozením záznamových médií. Tento ruch vzniká v určitém čase a prostoru, šíří se kanálem, kde je jeho tvar modifikován vlastnostmi kanálu, a může být zpracován jako odezva systému na impulz. Tranzientní ruch má podobný původ, ale jeho podoba je složitější – zahrnuje počáteční ostrý impulz následovaný klesajícími oscilacemi nízké frekvence, které mohou trvat až 50 ms. Tyto oscilace vznikají v důsledku rezonance kanálu, kterou vyvolá počáteční impulz. Příkladem tranzientního ruchu jsou škrábance na gramofonových deskách, kde je impulzní fáze způsobena reakcí přehrávacího systému na fyzické narušení povrchu a následné oscilace odrážejí charakteristickou odezvu přehrávacího zařízení. Oba druhy ruchů jsou úzce spojené,

protože impulzní ruch často spouští tranzientní jevy, což umožňuje jejich společnou charakterizaci jako odezvu systému na krátkodobé impulzní narušení [6].

2.2.5 Zkreslení

Zkreslení je v akustice a elektronice jakákoli změna signálu, která mění základní průběh nebo vztah mezi různými frekvenčními složkami. Obvykle se jedná o degradaci signálu. Přímé zesílení nebo zeslabení beze změny obálky signálu se obvykle za zkreslení nepovažuje. Amplitudové zkreslení se týká nerovnoměrného zesílení nebo zeslabení různých frekvenčních složek signálu a fázové zkreslení se týká změn fázových vztahů mezi harmonickými složkami komplexní vlny. Intermodulační zkreslení je výsledkem nelinearit v systému, kdy jedna frekvenční složka má tendenci modulovat jinou frekvenční složku - například vysoká zvuková frekvence moduluje nízkou zvukovou frekvenci. V audiosystémech jsou nejzřetelnějšími typy zkreslení amplitudová, frekvenční a intermodulační [7].

2.2.6 Výpadky

Povaha výpadkových ruchů je zřejmá už z názvu. Jedná se o dočasná přerušení přenášeného signálu. Důvodem těchto výpadků může být při nahrávacím procesu například špatný kontakt v konektoru či poškození kabelu. Tomuto druhu ruchů se však dá při nahrávacím procesu poměrně snadno předcházet, proto je již dále v této práci nebudeme uvažovat.

2.3 Redukce ruchů

Redukce ruchů digitálního zvukového signálu (neboli denoising) se mnohdy překrývá s problematikou enhancementu signálu či separace zdrojů. Obecně zde řešíme otázku, jak ze zašumělého signálu získat signál zbavený ideálně všech působících ruchů. Díky možnostem, které s sebou přinášejí neuronové sítě, se jedná o rychle se rozvíjející oblasti. Počátky této problematiky lze však vysledovat již na přelomu 70. a 80. let, kdy byly představeny algoritmy spektrálního odečítání a Wienerova filtru [5].

Následně byla úspěšně přijata metodika zvaná beamforming. Ačkoli se jednalo o významný pokrok, beamforming nebyl příliš praktický kvůli několika omezením. Prvním z nich je fakt, že k provedení redukce šumu je zapotřebí více mikrofonů. Zlepšení poměru čistého signálu a šumu (neboli SNR, viz podkapitola Metriky kvality redukce) je vysoce korelováno s počtem mikrofonů a výpočetní složitost roste s tímto počtem kvadraticky [5].

V posledních několika letech se zvýšil zájem o výzkum redukce šumu pouze s jedním mikrofonem. Konfigurace zařízen s jedním mikrofonem jsou totiž všudypřítomné a využití hlubokých neuronových sítí umožnilo dosahovat velice kvalitních výsledků. Jak bylo uvedeno výše, ze všech dostupných současných metod dosahují

nejlepších výsledků právě neuronové sítě. Než si však ukážeme, jak tyto metody redukce ruchů fungují, představíme si tři základní metriky kvality redukce, se kterými se v této problematice běžně setkáváme.

2.4 Metriky kvality redukce

Pro měření kvality redukce šumu existuje mnoho různých měřítek. Ta se často také využívají jako ztrátové funkce pro trénování neuronových sítí. Zde si představíme metriky SI-SNR, SDR a PESQ a jejich varianty, které jsou v problematice redukce ruchů v hudebních nahrávkách nejrelevantnější.

2.4.1 SNR, SI-SNR a SI-SNRi

Poměr signálu k šumu (Singal-to-Noise Ratio, zkráceně tedy SNR) je veličina využívaná ve vědě a technice k porovnání úrovně požadovaného signálu s úrovní šumu na pozadí. SNR je definován jako poměr výkonu signálu k výkonu šumu a nejčastěji se vyjadřuje v decibelech.

V problematice redukce šumů či separace zdrojů se častěji setkáme s variantou SI-SNR, tedy Scale-Invariat Signal-to-Noise Ratio. Ta je měřítkově invariantní, změna celkové hlasitosti signálu tedy nezmění výsledné SI-SNR [5].

Pro jednu vstupní zvukovou vlnu $s \in \mathbb{R}^L$, což je reálný vektor délky L se střední hodnotou nula, a odpovídající výstupní zvukovou vlnovou formu $\hat{s} \in \mathbb{R}^L$, jež představuje rekonstruovaný signál, je SI-SNR definován jako:

$$\text{SI-SNR} := 10 \cdot \log_{10} \left(\frac{\|s_{\text{target}}\|^2}{\|e_{\text{noise}}\|^2} \right) [5],$$

kde jsou jednotlivé složky získány následujícím způsobem:

$$s_{\text{target}} := \frac{\langle \hat{s}, s \rangle}{\|s\|^2} \cdot s, \quad e_{\text{noise}} := \hat{s} - s_{\text{target}} [5].$$

zde jsou definice užitých proměnných a operací:

- s je čistý zvukový signál s nulovou střední hodnotou, platí tedy $\frac{1}{L} \sum_{i=1}^{L} s_i = 0$
- $\hat{s}$ je rekonstruovaný signál, který je výstupem zpracování
- operace $\langle \hat{s}, s \rangle$ označuje skalární součin vektorů $\hat{s}$ a s.
- operace $\|s\|^2$ je druhá mocnina euklidovské normy vektoru s
- s_{target} je projekce rekonstruovaného signálu $\hat{s}$ na původní signál s
- e_{noise} je zbytkový šum, který představuje rozdíl mezi rekonstruovaným signálem a jeho projekcí na původní signál

V problematice redukce ruchů se často setkáme především s metrikou SI-SNRi, kde i značí zlepšení neboli improvement. SI-SNRi tedy vyjadřuje rozdíl SI-SNR naměřeného na původních datech a SI-SNR naměřeného na datech zpracovaných neuronovou sítí. Proto je SI-SNRi často využíváno pro vyhodnocení kvality redukce.

2.4.2 SDR a SDRi

Poměr signálu ke zkreslení (Signal-to-Distortion Ratio, zkráceně tedy SDR) je měřítko používané ke kvantifikaci kvality signálu ve srovnání s jeho zkreslenou nebo zašuměnou verzí. Běžně se používá v různých oblastech, včetně zpracování zvuku, telekomunikací a analýzy signálů, k posouzení vlivu zkreslení nebo šumu na původní signál [8]. I zde je obvyklou jednotkou decibel.

Vypočítává se pomocí tohoto vzorce:

$$\mathrm{SDR} = 10 \cdot \log_{10}\left(\frac{E_s}{E_d}\right)$$

kde E_s je energie referenčního signálu a E_d je energie zkreslení a šumu v zašumělém signálu. Energie E digitálního konečného signálu $x[n]$ je daná následujícím vzorcem:

$$E = \sum_{n=0}^{N-1} |x[n]|^2$$

kde N je počet vzorků v nahrávce a n je index z množiny přirozených čísel. SDR se vyjadřuje v decibelech (dB) a představuje, o kolik je původní signál hlasitější ve srovnání se zkreslením nebo šumem přítomným v degradované verzi [8].

I zde se často sektáme s variantou značící zlepšení SDR zvanou SDRi, kde i opět značí improvement. Obdobně jako u SI-SNRi, i zde je SDRi určuje rozdíl mezi SDR původních dat a SDR dat vygenerovaných neuronovou sítí.

2.4.3 PESQ

Perceptual Evaluation of Speech Quality (zkráceně PESQ) je metrika určená především pro analýzu nahrávek s lidskou řečí. Zveřejněna byla už v roce 2001. Jedná se o standard kvality zvuku, který zohledňuje charakteristiky jako jeho ostrost, hlasitost hovoru, šum na pozadí, oříznutí, rušení a podobně. PESQ vrací skóre v rozmezí <-0,5 , 4,5>, přičemž vyšší skóre znamená lepší kvalitu [9].

Na rozdíl od výše uvedených metrik se však nejedná o jednoduchý vzorec, ale komplexnější algoritmus. Diagram celého algoritmu můžeme vidět zde.

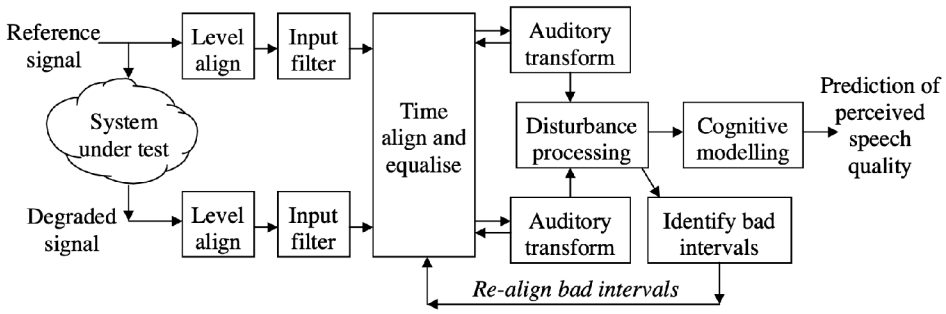


Figure 1: Structure of perceptual evaluation of speech quality (PESQ) model.

Obrázek 2.1: PESQ diagram [10]

3 Potlačování ruchů pomocí neuronových sítí

Neuronové sítě jsou typem algoritmu strojového učení inspirované lidským mozkem. Skládají se ze vzájemně propojených uzlů neboli umělých neuronů uspořádaných do vrstev. Každý neuron přijímá vstupní signály, zpracovává je a výsledek předává další vrstvě. Neuronové sítě se prostřednictvím tréninku na velkých souborech dat učí rozpoznávat vzory a na základě nich generovat předpovědi nebo rozhodnutí [11].

Jak bylo uvedeno v úvodu, neuronové sítě dnes stále přinášejí průlomy v mnoha různých odvětvích a ani problematika potlačování ruchů není výjimkou. Z toho důvodu pro tuto problematiku existuje několik různých přístupů a z nich vycházejících architektur. V této kapitole se s některými z těchto přístupů seznámíme a podrobněji si rozebereme i některé konkrétní architektury.

3.1 Architektury pro separaci zdrojů

Problematika redukce ruchů pomocí neuronových sítí je v současnosti poměrně úzce spjata s problematikou zvanou source separation, tedy separace zdrojů signálu. Ta se zabývá řešením otázky, jak ze signálu, který vznikl lineární kombinací libovolného počtu jiných signálů, získat původní zdrojové signály.

I redukci ruchů totiž můžeme považovat za separaci dvou zdrojů: signálu chtěného a nechtěného, tedy čistého signálu a ruchu. Pokud budeme uvažovat výše uvedenou rovnici:

$$y(t) = x(t) + n(t),$$

řešíme v problematice separace zdrojů otázku, jak ze signálu $y(t)$ který vznikl lineární kombinací signálů $x(t)$ a $n(t)$ získat pouze signál $x(t)$, a to bez konkrétní znalosti $n(t)$. Dalo by se tedy říci, že separace zdrojů je v jistém smyslu obecnější variantou problému redukce šumu.

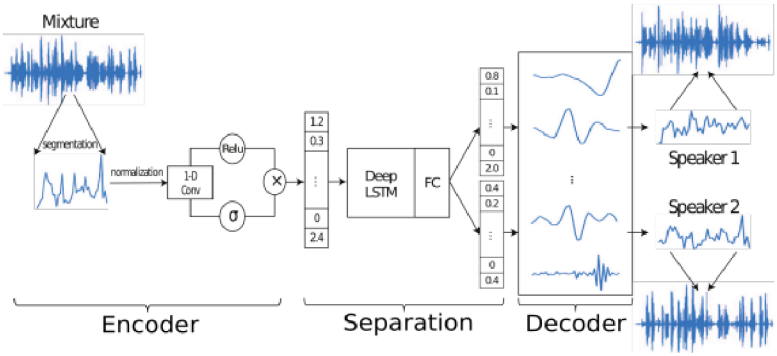
Díky této souvislosti se pro redukci ruchů často používají právě architektury neuronových sítí pro separaci zdrojů, popřípadě jejich modifikace. Proto si v následující kapitole uvedeme nejzásadnější a nejpoužívanější z nich.

3.1.1 TasNet

Architektura TasNet, zveřejněná v roce 2018, byla v problematice separace zdrojů přelomem. Všechny předešlé architektury totiž nebyly dostatečně efektivní pro práci

v reálném čase. Robustní zpracování řeči v prostředích s více mluvčími vyžaduje účinnou separaci řeči. Moderní systémy využívající hluboké učení přinesly v této oblasti významný pokrok, avšak stále představovaly výzvu, zejména v aplikacích s krátkou latencí v reálném čase. Většina metod se dříve snažila konstruovat masku pro každý zdroj v časově-frekvenčním zobrazení smíšeného signálu, což ovšem nemuselo vést k optimální reprezentaci pro separaci řeči. Kromě toho také tato časově-frekvenční dekompozice přinášela inherentní problémy, jako je rozdělení fáze a amplitudy. Dalším problémem bylo také příliš dlouhé časové okno, které bylo nutné pro dosažení dostatečného frekvenčního rozlišení [12].

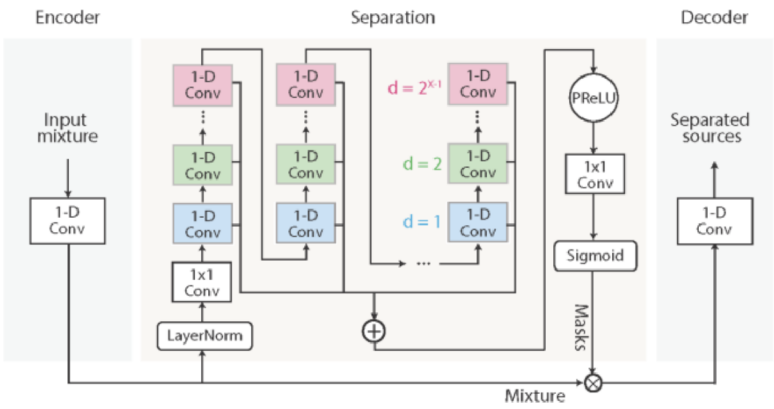
Pro překonání těchto omezení byla navržena tato síť fungující jen a pouze v časové doméně. Tento přístup přímo modeluje signál pomocí rámce enkodér-dekodér a provádí separaci zdrojů na výstupech enkodéru. Tím se eliminuje krok frekvenční dekompozice a problematiku separace redukuje na odhad masek zdrojů na výstupech enkodéru, které jsou následně syntetizovány dekodérem. Tato architektura v roce 2018 překonala nejlepší soudobé algoritmy pro separaci řeči, a to jak kauzální, tak nekauzální. Zároveň také snížila výpočetní náročnost na separaci řeči a s tím související latenci výstupu. Díky tomu je i dnes TasNet vhodnou architekturou pro aplikace, kde je žádoucí nízkoenergetická implementace v reálném čase, jako jsou zařízení pro poslech a telekomunikační přístroje [12].



Obrázek 3.1: Architektura TasNet [12]

3.1.2 Conv-TasNet

Architektura Conv-TasNet je následníkem výše uvedeného TasNetu. Liší se zejména návrhem enkodéru a dekodéru. V těch na rozdíl od TasNetu využívá konvoluční vrstvy. Obvykle používá rozšířené konvoluce, které nahušťují konvoluční kernely vložením děr mezi jednotlivé prvky. Díky tomu mají větší receptivní pole a mohou účinně detekovat závislosti na velké vzdálenosti ve vstupních signálech. To síti umožňuje zachytit komplexnější vztahy mezi jednotlivými zdroji signálu, díky čemuž také dosahuje lepších výsledků než TasNet, ovšem za cenu větší výpočetní náročnosti. Conv-TasNet má 5,1 milionů parametrů a dosahuje SI-SNRi 7,8 dB [13].

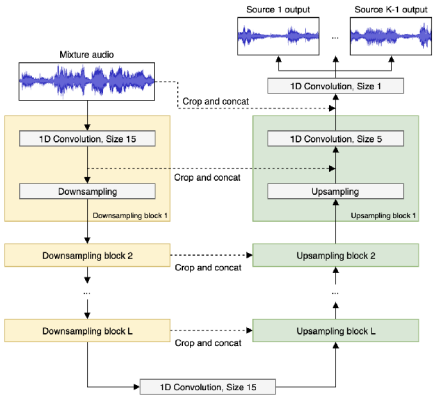


Obrázek 3.2: Architektura Conv-TasNet [13]

3.1.3 U-Net a Wave-U-Net

Architektura U-Net je konvoluční neuronová síť původně navržená za účelem segmentace obrazu v oboru biomedicíny. Skládá se z kontrakční a expanzní části. Kontrakční část obsahuje opakované aplikace dvou 3x3 konvolucí, z nichž každá je následována aktivační funkcí ReLU a max-poolingem o velikosti 2x2 pro snížení rozměru. Při každém kroku downsamplingu (neboli podvzorkování) se pak zdvojnásobí počet příznakových map. Expanzní část provádí upsampling (neboli nadvzorkování) příznakové mapy, ten je následovaný 2x2 konvolucí, která snižuje počet příznakových map. Následně dochází ke spojení s odpovídající oříznutou příznakovou mapou z kontrakční části. Poté následují dvě 3x3 konvoluce, každá s aktivační funkcí ReLU. Na konci sítě je 1x1 konvoluce, která převádí příznakové mapy na požadovaný počet tříd. Celkem síť obsahuje 23 konvolučních vrstev [14].

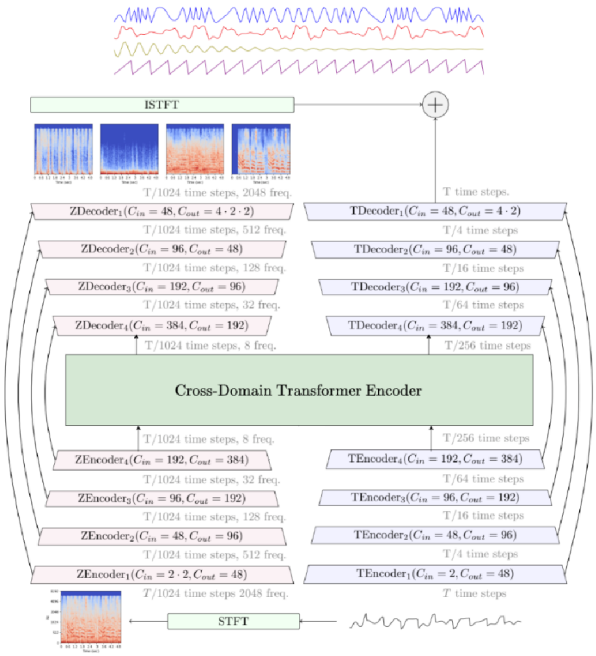
Architektura sítě U-Net i přes své účely využití v oboru biomedicíny inspirovala několik různých architektur pro účely separace zdrojů a redukce ruchů. Jednou z nich je architektura Wave-U-Net. Tato síť byla navržena pro separaci zdrojů přímo v časové doméně, čímž odstraňuje závislost na spektrálních transformacích a umožňuje modelování fázové informace. Wave-U-Net opakovaně převzorkovává mapy příznaků, což umožňuje zpracování a kombinaci příznaků na různých časových škálách. Mezi významné inovace této architektury patří výstupní vrstva, která zajišťuje aditivitu zdrojů. Další významnými inovacemi je využití techniky nadvzorkování, tedy tzv. upsamplingu, nebo také širší rámec predikce zohledňující kontext, což snižuje množství nechtěných výstupní artefaktů. Studie zaměřená na separaci zpěvu ukazuje, že Wave-U-Net dosahuje srovnatelných výsledků jako spektrálně založené varianty U-Netu, a to při zpracování stejných dat. Tato architektura tak představuje efektivní přístup k separaci zvukových zdrojů přímo v časové doméně [15]. Její architekturu můžete vidět na následujícím diagramu.



Obrázek 3.3: Architektura Wave-U-Net [15]

3.1.4 Demucs

Architektura Demucs je založena na konvoluční architektuře výše uvedené sítě Wave-U-Net. Liší se především tím, že nejvnitřnější vrstvy jsou nahrazeny transformačním kodérem. Od jejího zveřejnění se vyvinuly různé verze modelů Demucs. Zde se budeme věnovat verzi v4. Verze v4 obsahuje hybridní transformer, který využívá jak časovou, tak frekvenční oblast daného signálu [16]. Jeho nejvnitřnější vrstvy pak nahrazeny transformačním cross-domain enkodérem. Transformer využívá vlastní pozornost v rámci každé samostatné domény a cross-attention napříč všemi doménami [17]. Architekturu sítě Demucs v4 můžete vidět na následujícím obrázku.

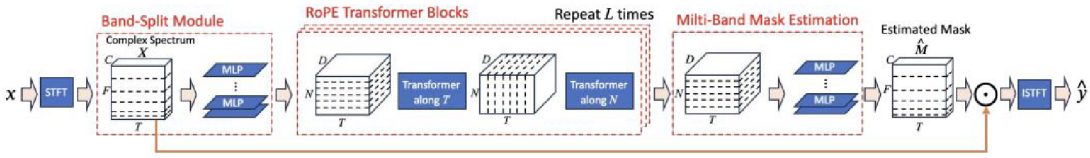


Obrázek 3.4: Architektura Demucs [16]

Jak lze z tohoto obrázku vyčíst, vstupní vlnová forma je zpracována jak pomocí časového enkodéru, tak pomocí STFT (krátkodobá Fourierova transformace), po níž následuje spektrální enkodér. Obě reprezentace jsou následně sečteny, pokud jejich dimenze odpovídají. Dekodér je vytvořen symetricky. Výstupní spektrogram prochází inverzní STFT (ISTFT) a je sečten s výstupy vlnové formy, čímž vzniká finální výstup modelu. Pro spektrální vrstvy je používán prefix Z, zatímco pro časové vrstvy prefix T [16].

3.1.5 Band Split RoFormer

Architektura Band Split RoFormer je navržena pro práci ve frekvenční doméně, vstupní signál tedy nejprve transformuje pomocí STFT, výstup pak transformuje zpátky do časové oblasti pomocí ISTFT. Využívá modul pro rozdělení pásem, který promítá vstupní komplexní spektrogram do reprezentací na úrovni dílčích pásem. Poté uspořádá sadu hierarchických transformátorů pro modelování vnitřních pásem i mezipásmových sekvencí pro odhad vícepásmové masky [18]. Diagram této architektury můžete vidět na následujícím obrázku.



Obrázek 3.5: Architektura Band Split RoFormer [18]

Tato architektura využívá transformační bloky Rotary Position Embedding (zkráceně RoPE). Jedná se o strukturu, která využívá hierarchický přístup k zpracování dat na časové a frekvenční ose. Každý transformer blok má dva hlavní moduly – tzv. časový (time-transformer) a sub-bandový transformer (subband-transformer). Časový transformer se zaměřuje na modelování časových souvislostí v rámci jednotlivých frekvenčních pásem, což znamená, že analyzuje, jak se signál mění v čase uvnitř daného pásma. Subband-transformer naopak zajišťuje, aby byla propojena informace mezi různými frekvenčními pásmy. Tento modul tedy zachytává vzájemné vazby mezi pásmy, což je důležité pro globální pochopení spektrální struktury signálu. Oba typy používají stejnou architekturu a obsahují moduly pro pozornost (attention) a pro dopřednou výpočetní vrstvu (feedforward) [18].

Po trénování a testování na datové sadě MUSDB18 dosahuje architektura Band Split RoFormer průměrného SDRi 9,8 dB.

3.1.6 Mel-Band RoFormer

Mel-Band RoFormer je pokročilá architektura pro oddělování hudebních zdrojů založená na multi-band spektrálních přístupech, podobně jako předchozí model Band-Split RNN (BSRNN). Band-Split RNN prokázal, že rozdělení spektra na několik

pásem může při zpracování zvuku přinést kvalitní výsledky. Proto na tento přístup navázal model BS-RoFormer, který kombinuje koncept rozdělení na pásma z BSRNN a hierarchický Transformer s rotačním pozičním emdeddingem (RoPE), což umožňuje efektivní modelování vztahů jak uvnitř pásem, tak mezi nimi, čímž se dosahuje vysoce přesného odhadu masek pro oddělení zvukových zdrojů [19].

Mel-Band RoFormer se od BS-RoFormeru odlišuje především způsobem rozdělení pásem. Zatímco BS-RoFormer využívá rozdělení pásem, které je založeno na heuristice a neobsahuje žádná teoretická odůvodnění, Mel-Band RoFormer využívá melovské pásmové schéma. Toto schéma přerozděluje frekvenční složky do překrývajících se subpásem podle melové škály, která je inspirována lidským vnímáním zvuku, a lépe tak odpovídá přirozenému vnímání rozdílů mezi frekvencemi.

Díky tomuto přístupu Mel-Band RoFormer vykazuje ve srovnání s BS-RoFormerem lepší výsledky. Konkrétně model lépe zvládá separaci vokálů, bicích a dalších hudebních složek, což z něj činí vhodnější architekturu pro náročné úlohy separace zvukových zdrojů [19].

3.1.7 Spleeter

Spleeter je open-source knihovna pro separaci hudebních zdrojů navržená na základě frameworku Tensorflow. Zahrnuje předtrénované modely pro separaci samostatných vokálů, dále pro separaci čtyř stemů (vokály, basa, bicí a ostatní) a separaci pěti stemů (zde přibývá extra stem pro klavír). Předtrénované modely jsou dvanáctivrstvé sítě U-Net s architekturou CNN (Convolution Neural Network - konvoluční neuronová síť) enkodér-dekodér a s využívanými skip-connections (6 vrstev pro enkodér a 6 pro dekodér). Jedná se o jeden z prvních úspěšných modelů neuronových sítí pro separaci hudebních zdrojů. Byl uveřejněn týmem společnosti Deezer [20].

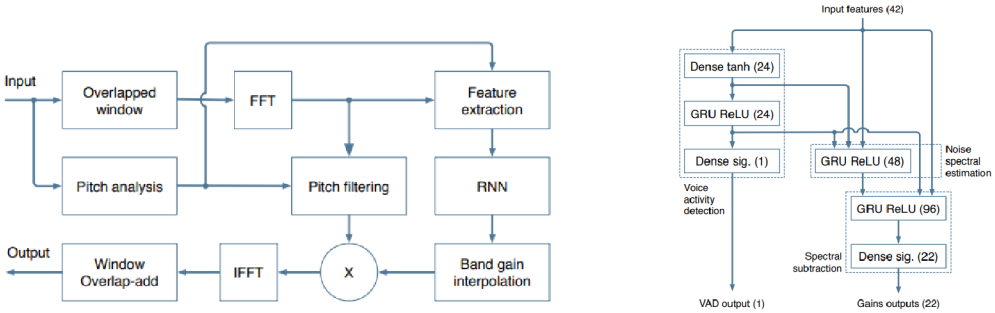
Vzhledem k tomu, že celou separační linku lze spustit na GPU a že model je založen na CNN, jsou výpočty efektivně paralelizovány a predikování modelu je velmi rychlé. Spleeter je například schopen rozdělit celou sadu testovacích dat musdb18 (přibližně 3 hodiny a 27 minut zvuku) na 4 stemy za méně než 2 minuty za použití jednoho grafického procesoru GeForce RTX 2080 a dvojitého procesoru Intel Xeon Gold 6134 @ 3,20 GHz (CPU se používá pouze pro načítání souborů mixu a export kmenových souborů). V tomto nastavení je Spleeter schopen zpracovat 100 sekund stereofonního zvuku za méně než 1 sekundu, což jej činí velmi užitečným pro efektivní zpracování velkých souborů dat [20].

3.2 Architektury pro redukci šumů

Kromě výše uvedených neuronových sítí určených primárně pro problematiku separace zdrojů, které se následně dají snadno modifikovat pro účely redukce šumů, se můžeme setkat i s modely, které byly vyvíjeny pouze za účelem redukce šumů. Dochází tedy k separací jen dvou signálů, čistého signálu a ruchu. Signál ruchu u těchto sítí bývá často ignorován. Tyto architektury už tedy není nutné jakkoliv modifikovat pro potřeby redukce. V následujících podkapitolách si představíme dva takové modely, a to Spiking-FullSubNet a RNNoise.

3.2.1 RNNoise

Ačkoliv byl tento model publikován už v roce 2018, stále je poměrně hojně využíván nezávislými vývojáři v různých real-time aplikacích pro redukci šumu. Jak už název napovídá, jedná se o architekturu využívající rekurentní neuronové sítě (RNNs) [21]. Zde je blokový diagram celého algoritmu.



(a) Blokový diagram celého algoritmu [21]

(b) RNN [21]

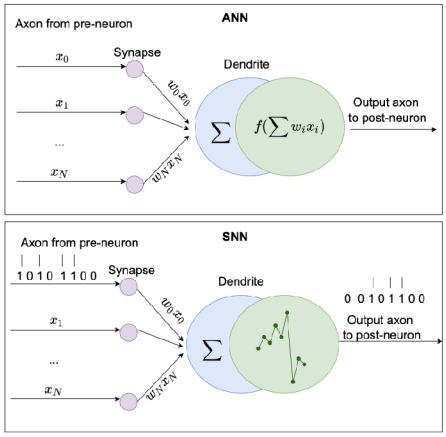
Obrázek 3.6: Architektura RNNoise [21]

Architektura RNNoise kombinuje prvky tradičních algoritmů pro potlačení šumu s pokročilými technikami strojového učení, Rekurentní vrstvy zde slouží k modelování časových závislostí v audiosignálu, což umožňuje efektivnější potlačení šumu. Celková struktura algoritmu zahrnuje 215 jednotek rozdělených do 4 skrytých vrstev, přičemž největší vrstva obsahuje 96 jednotek. Namísto tradičních LSTM (Long Short-Term Memory) se zde používají gated recurrent units (GRU), které jsou jednodušší na implementaci a v této úloze dosahují mírně lepší výkonnosti. RNNoise také využívá spektrální analýzu signálu jako vstup, což odpovídá tradičním přístupům k redukci šumu, kde je signál rozdělen do frekvenčních pásem a analyzován v čase. Výstup neuronové sítě je následně použit pro modifikaci spektra vstupního signálu, což zajišťuje potlačení šumu při zachování kvality původního zvuku. [21].

3.2.2 Spiking-FullSubNet

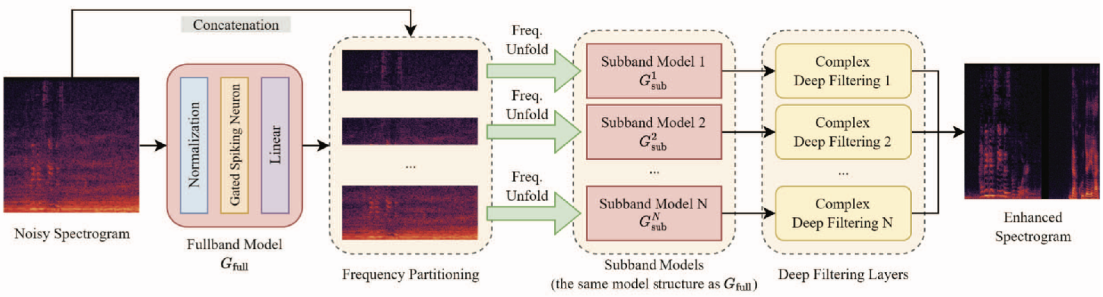
Spiking-FullSubNet vychází z méně známého, ale v současnosti čím dál více rozrůstajícího se konceptu zvaného Spiking Neural Networks (SNNs). Tyto architektury se oproti klasickým ANN (Aritifial Neural Network) architekturám snaží svým fungováním ještě více přiblížit fungování reálných neuronových sítí, protože jako hlavní nosiče informace využívají časované diskrétní špice (tedy spike) [23].

Rozdíl mezi neuronem klasické ANN a SNN ilustruje následující obrázek. Jak můžeme vidět, hlavním rozdílem je binární vektor časového výskytu špic, který je prvním vstupem každého neuronu a zároveň také jeho výstupem. Místo nelineární funkce jako u ANN se zde využívá časové mapování hodnot, ze kterého se následně generuje další binární časový vektor.



Obrázek 3.7: Porovnání ANN a SNN [22]

Architektura Spiking-FullSubNet staví na tomto konceptu neuronových sítí a rozšiřuje ho dále implementací gated spinking neural modelu (GSN), který umožňuje dynamicky kontrolovat paměťové úložiště neuronové sítě. SSN architektury totiž mívají problém se zachováváním dočasných více-škálových informací. Díky implementaci GSN však FullSubNet umí lépe uchovávat klíčovou historii, což je pro kvalitní redukci šumu zásadní [23]. Přehled této architektury poskytuje následující obrázek.



Obrázek 3.8: Architektura Spiking-FullSubNet [23]

4 Datová sada

Jak je všeobecně známo, úspěšnost NN modelů stojí a padá na kvalitě trénovacích dat. Pro potřeby této diplomové práce jsem se rozhodl vycházet především z jednoho veřejně dostupné datové sady, která je mezi akademickými pracemi standardem pro porovnávání jejich úspěšnosti - jedná se MUSDB18 [24].

Z této datové sady jsem extrahoval pouze nahrávky vokálů, na které se v této diplomové práci zaměřuji. Na těch jsem následně provedl tzv. augmentaci. Augmentace je v kontextu strojového učení proces rozšíření nebo obohacení existující datové sady o další prvky, vlastnosti či informace s cílem zlepšit její funkčnost či kvalitu nebo. V kontextu redukce ruchů je tak označován především proces přidávání ruchů do nahrávek. Tuto augmentaci jsem prováděl pomocí vlastních programů napsaných v jazyce Python.

Cílem bylo napodobit ruchy, se kterými se audioinženýři během nahrávání vokálů běžně setkávají a které vznikají lineární kombinací čistého signálu a ruchu. Zdá se totiž, že principiálně by každý z těchto ruchů měl jít redukovat stejnou metodou, respektive stejnou architekturou. Proto jsem se zaměřil na ambientní ruchy, elektromagnetické šumy a impulzní či tranzientní šumy v podobě tzv. clicku. Redukce reverberace a kompenzace výpadků by si zřejmě žádaly jiné, pravděpodobně i komplexnější přístupy, proto jsem je v této práci neuvažoval. Experimentuji zde však i s lehkým zkreslením, které je svou povahou nelineární. Mohlo by totiž být zajímavé pozorovat, jak si s tím výše uvedené modely poradí.

Přidané šumy jsem rozdělil do následujících kategorií:

1. přeslechy clicktracků ze sluchátek
2. šumění okolní elektroniky
3. hluky z pozadí
4. náhodná zkreslení signálu vadou kabelů
5. náhodné vysoké či subbasové tóny

V následujících podkapitolách si blíže představíme datovou sadu MUSDB18, povahu použitých externích ruchů a také části mého programu pro augmentaci.

4.1 MUSDB18

Datová sada MUSDB18 je největší datovou sadou s otevřeným přístupem určená pro problematiku separace zdrojů, v rámci které je také zřejmě nejvyužívanějším datovou sadou. Jedná se o 150 hudebních nahrávek (cca 10 hodin celkového času) různých žánrů. Jsou rozděleny do kategorie trénovací, ve které se nachází 100 nahrávek, a do kategorie testovací, ve které se nachází zbylých 50 nahrávek.

Nahrávky v této datové sadě jsou stereofonní a zakódované se vzorkovací frekvencí 44,1 kHz. Každá z nich je rozdělena na stopu bicích, basy, vokálů, ostatních nástrojů a samozřejmě také mixu všech stop, který odpovídá jejich součtu [24].

Tato datová sada je dostupná ve dvou variantách: MUSDB18 a MUSD18-HQ. V první variantě jsou všechny soubory zakódovány ve stem formátu společnosti Native Instruments stems s koncovkou `.mp4` s kódováním AAC (Advanced Audio Coding) s přenosovou rychlostí 256 kbps. V této variantě má datová sada velikost 4,4 Gb. Vzhledem k tomu, že směs je samostatně kódována jako AAC, existuje malý rozdíl mezi součtem všech zdrojů a směsí. Výsledkem komprese je tedy šířka pásma omezená na 16 kHz. Oproti tomu varianta MUSDB18-HQ nabízí nekomprimované `.wav` soubory a je vhodná především pro modely, jejichž cílem je predikovat v šířce pásma až 22 kHz, celková velikost této verze je tedy 22.7 Gb. Jinak jsou tyto datové sady totožné. Pro porovnávání výsledků daných neuronových sítí se však v literatuře a v kampani SiSEC (Signal Separation Evaluation Campaign) využívá výhradně komprimovaná varianta [24].

K této datové sadě je také veřejně dostupná Python knihovna `musdb` určená pro jednoduchou práci s MUSDB18, především pro parsing a zpracování, dále také pro snadnou integraci s běžně používanými knihovnami jako `numpy`, `tensorflow` nebo `pytorch` [25].

Pro potřeby této diplomové práce jsem využil komprimovanou verzi této datové sady, ze které jsem pomocí knihovny `musdb` mohl snadno extrahovat stopy všech vokálů.

4.2 Externí ruchy

Kromě samostatných vokálů bylo také potřeba sehnat nahrávky různých šumů, které by čisté nahrávky vokálů z datové sady MUSDB18 narušily. Proto jsem na webové stránce *freesound.org* [26] našel 28 různých nahrávek šumu, které mi připadaly dostatečně podobné tomu, s čím se při nahrávání můžeme běžně setkat: elektrické šumy, hluky větráků či klimatizace, nebo také hluky z ulice, které často přeznívají i skrz zavřená okna.

K tomu jsem také na výše uvedené stránce sehnal 7 různých vzorků tzv. clicku. Jedná se o velice jednoduché impulzní zvuky, které se opakují v pevně daném tempu nahrávky, čímž vzniká tzv. clicktrack. Ten si hudebníci často pouští do sluchátek, aby při nahrávání neztratili o daném tempu pojem. Clicktrack však při nahrávání na kvalitní mikrofony může ze sluchátek přeznívat a být tak na výsledné nahrávce slyšitelný. Proto jsem se rozhodl zahrnout do redukce šumu i tento jev.

4.3 Augmentace datové sady

Pro augmentaci datové sady jsem vytvořil vlastní program v jazyce Python 3.9. Ten postupně otevírá každou nahrávku čistého vokálu z trénovací i testovací sady MUSDB18 a nejprve provádí monofonizaci a převzorkování. V dalším kroku k dané nahrávce přičítá výše uvedené ruchy, popřípadě různé náhodné vysoké či subbasové tóny, nebo také danou nahrávka lehce zkreslí. Po těchto operacích je nutné signál normalizovat, aby bylo zaručeno, že se jeho výsledná magnituda pohybuje v intervalu $\langle -1, 1 \rangle$. Všechny tyto procesy jsou konkrétněji popsány v následujících podkapitolách. Napříč celým kódem používám především knihovny `torchaudio` a `torch`, příležitostně také knihovny `librosa` či `numpy` a samozřejmě i `os` a `glob`.

Každému z výše uvedených jevů je přidělena nějaká pravděpodobnost výskytu. Rozhodl jsem se tak proto, aby natrénovaná neuronová síť automaticky nepředpokládala výskyt těchto jevů v každém případě a naučila se tedy detekovat, kdy daný jev nastává a kdy ne, což by mělo vést k lepší schopnosti generalizace.

Kromě toho je do procesu augmentace zapojeno i několik náhodných veličin, které můžou ovlivnit jak intenzitu a hlasitost daného jevu, tak i jeho parametry. Tím se opět snažím zaručit kvalitnější schopnost generalizace výsledné neuronové sítě.

Nakonec se všechna data uloží ve formátu `.wav` se vzorkovací frekvencí 16 kHz s bitovou hloubkou 32 bitů a enkódováním PCM_F do příslušných složek. Zvlášť se ukládají trénovací a testovací data.

Vzhledem k potřebám mnou využívaných modelů se také zvlášť ukládají stopy, které obsahují pouze šum a žádný čistý signál. Tomu tak je právě kvůli výše zmíněnému konceptu separace signálů, který pro fázi učení potřebuje znát povahu obou zdrojových signálů - tedy jak čistého signálu, tak ruchu.

Poslední věc, kterou si k jednotlivým nahrávkám ukládám, jsou soubory JSON, které v sobě o každé nahrávce nesou informaci o tom, jaké ruchy na ní byly aplikovány, s jakou intenzitou, popřípadě také s jakými parametry. To slouží pouze pro moji kontrolu kvality výsledného datové sady.

4.3.1 Přidání clicktracku

Python funkce, která zajišťuje přidání clicktracku, nejprve detekuje tempo vokálové nahrávky pomocí funkce `beat_track` knihovny `librosa`.

Poté vezme vzorek clicku a nakopíruje ho do prázdného vektoru o délce nahrávky vokálů tak, aby byl každý úder clicku v tempu dané nahrávky. Jak to také u clicků bývá, každý jeho úder na první dobu je posunut o dva tóny výše než ostatní. Zde je má funkce, která clicktrack vytváří:

```
1 def create_click_track(self, click, track, fs, note,
    pitch_click_period):
2
3     len_voc = self.torch_len(track)
4     len_click = self.torch_len(click)
5     tempo, _ = librosa.beat.beat_track(y=track[0].numpy(), sr=fs)
6     if isinstance(tempo, np.ndarray):
```

```
        tempo = tempo.item()
    if tempo == 0:
        tempo = 129
    samples_per_beat = int(60 * fs // float(tempo))
    samples_per_note = int(samples_per_beat / note)
    note_idxs = torch.arange(0, len_voc, samples_per_note)
    note_idxs = note_idxs[note_idxs <= len_voc - len_click]
    click_track = torch.zeros(len_voc)
    pitched_click = torchaudio.functional.pitch_shift(click, fs,
        pitch_click_by_semitones)
    cnt = 0
    for note_idx in note_idxs:
        if cnt % pitch_click_period == 0:
            click_track[note_idx:note_idx + len_click] =
                pitched_click
        else:
            click_track[note_idx:note_idx + len_click] = click
        cnt += 1
return click_track
```

Jednotlivé vzorky clicků nejsou vybírány náhodně, jejich rozložení napříč datovou sadou je téměř rovnoměrné díky jejich ukládání do zásobníku. Jakmile zásobník vydá svůj poslední vzorek, znovu se vrátí k prvnímu.

4.3.2 Přidání dynamického šumu

Funkce pro přidání dynamického šumu je značně jednodušší. Načtený signál šumu pouze nakopíruje či zkrátí tak, aby působil po celou dobu nahrávky.

```
def noise_len_to_audio_len(self, noise, len_noise, len_audio):
    if len_noise < len_audio:
        repetitions = len_audio // len_noise + 1
        return torch.tile(noise, (repetitions,))[:len_audio]
    elif len_noise > len_audio:
        return noise[:len_audio]
```

Stejně jako u vzorků clicku, i zde jsou jednotlivé nahrávky šumu vybírány ze zásobníku pro zaručení jejich téměř rovnoměrného rozložení napříč datovou sadou.

4.3.3 Přidání statického šumu

Statický šum vytvářím jednoduše pomocí *torch* generátoru náhodných vektorů o velikosti dané nahrávky s hodnotami v rozmezí <-1, 1> s normálním rozložením hodnot.

```
static_noise = torch.randn(audio.shape[1]) * 2 - 1
```

4.3.4 Přidání náhodných tónů

Do daných nahrávek jsou přidávány buďto vysoké tóny v rozmezí 6 kHz až 15 khHz, nebo basové a subbasové tóny v rozmezí 30 Hz až 80 Hz. Konkrétní hodnoty

výchozích frekvencí jsou vždy generovány náhodně. Na základě nich se dále vytvoří sinusoida následujícím způsobem.

```
t = torch.arange(audio.shape[1]) / fs
sinus = torch.sin(2 * torch.pi * f * t)
```

4.3.5 Lehké zkreslení

Existuje velké množství způsobů jak signál zkreslit. Pro potřeby této práce však stačil jednoduchý postup pomocí funkce *clamp* z knihovny *torch*. Tato funkce slouží k omezování hodnot tensoru na určitou intervalovou hranici. Jeho hodnoty omezí tak, že jakýkoli prvek, který je menší než minimální hodnota, bude nastaven na tuto minimální hodnotu, a jakýkoli prvek větší než maximální hodnota bude nastaven na dané maximum. Tomu se jinými slovy říká také klipování signálu. Tím se dá u zvukových signálů jednoduše dosáhnout efektu zkreslení.

4.3.6 Mix čistého vokálu a ruchu

Pro kombinaci čistého signálu a ruchu jsem si vytvořil vlastní funkci. Jejím vstupem je čistý signál, zašumělý signál a mixovací koeficient. Ten se pohybuje v intervalu <0,1>, kde 0 znamená absenci zašumělého signálu a 1 absenci čistého signálu ve výsledné stopě.

Jelikož však přimíchávané ruchy byly často mnohem hlasitější než čisté vokály, bylo potřeba do tohoto procesu zahrnout i energii jednotlivých signálů. Tato energie je vypočítána jak pro čistý signál, tak pro zašumělý. Následně je zašumělý signál škálován následujícím způsobem, který zaručí, že mají oba signály energii stejnou:

$$x_{noise}[n] := \sqrt{\frac{E_{clean}}{E_{noise}}} \cdot x_{noise}[n]$$

kde E_{clean} označuje energii čistého vokálu a E_{noise} energii přidaného šumu. Samotný mix pak probíhá na základě následující rovnice:

$$x_{mix}[n] = (1 - \alpha) \cdot x_{clean}[n] + \alpha \cdot x_{noise}[n]$$

kde $x_{\text{mix}}[n]$ označuje vektor výsledného signálu, $x_{\text{clean}}[n]$ čistý signál a α koeficientem mixu.

4.3.7 Normalizace

V nejjednodušším smyslu je normalizace zvuku proces měření určitého aspektu zvukového signálu a jeho úpravy tak, aby odpovídal předem definovanému cíli. V zásadě může být měřeným a upravovaným prvkem libovolný měřitelný prvek tohoto signálu. V praxi se téměř vždy jedná o úroveň jeho peaků nebo o jeho průměr [27].

Pro potřeby této práce jsem si vytvořil vlastní normalizační funkci, níže je její kód. Tato funkce spočítá střední hodnotu a odchylku daného signálu. Následně se od vstupního signálu odečte střední hodnota a celý signál se poté vydělí odmocninou z

jeho odchylky. Pokud se však odchylka blíží nule, dochází pouze k odečtení střední hodnoty.

```
def normalize_audio(self, audio):
    mean = torch.mean(audio)
    variance = torch.var(audio)
    if variance > 1e-9:
        audio = (audio - mean) / torch.sqrt(variance)
    else:
        audio = audio - mean
    return audio
```

Tuto normalizaci aplikuji jak na zašumělé nahrávky, tak na referenční nahrávky čistého vokálu, aby při procesu tréninku modelu neuronové sítě nevznikala zbytečná ztráta způsobená pouze rozdílem v hlasitosti referenční a vstupní nahrávky.

Jelikož je normalizace aplikována jak na zašumělá, tak referenční data, je úkolem sítě rekonstruovat signál bez ohledu na absolutní hlasitost, ale se zaměřením na odstranění šumu. Díky tomu se síť učí na relativních vlastnostech signálu a zachovává dynamické charakteristiky, které jsou klíčové pro subjektivní kvalitu zvuku. Možnou změnu hlasitosti, která by ve výsledné aplikaci takto trénovaná síť způsobila, je pak možné jednoduše řešit efektem zvaným gain, o kterém se v práci zmíním později.

4.4 Finální datová sada

Finální datová sada je tedy tvořena zašumělými nahrávkami vokálů z veřejně dostupné sady MUSDB18. Všechny nahrávky jsou monofonní se vzorkovací frekvenci 16 kHz a bitovou hloubku 32 bitů ve formátu `.wav` s enkódování PCM_F. Zde je přehled jeho velikosti:

Tabulka 4.1: Přehled datové sady

Kategorie	**Počet nahrávek**	**Celková doba [h]**	**Celková velikost [MB]**
Train	100	6.35	1392.07
Test	50	3.46	761.35

Pro kvalitnější výsledky by bylo určitě vhodné tuto datovou sadu rozšířit o další nahrávky vokálů a šumů, dále také nepracovat s nahrávkami se vzorkovací frekvencí 16 kHz ale 44,1 kHz, což je pro hudební produkci standardem. Pro potřeby této diplomové práce jsem se však rozhodl pracovat pouze s těmito omezenými zdroji z důvodu výpočetní a časové náročnosti, kterou si každé trénování modelu žádá. Kromě toho také vycházím pouze z datové sady MUSDB18, protože je téměř standardem pro měření úspěšnosti separace zdrojů. Bylo tedy užitečné mít možnost porovnání s výsledky, kterých na této datové sadě dosahují jiné modely.

Během fáze vývoje neuronových sítí jsem se rozhodl vytvořit pět verzí této datové sady. Jednotlivé verze se od sebe liší dominantním zastoupením jednotlivých typů ruchů, tedy jejich pravděpodobností výskytu a intenzitou působení. Každá verze

odpovídá jednomu dominantnímu ruchu, tedy clicktracku, dynamickému šumu, statickému šumu, náhodným tónům a lehkému zkreslení. V každé verzi však do určité míry figurují všechny výše uvedené ruchy, aby při trénování docházelo k lepší generalizaci. Zde je tabulka pravděpodobností výskytů těchto ruchů v jednotlivých verzích datové sady.

Tabulka 4.2: Pravděpodobnosti výskytu ruchů v různých verzích datové sady

Verze	**Clicktrack**	**Dyn. šum**	**Stat. šum**	**Náh. tóny**	**Zkreslení**
1	0.8	0.3	0.2	0.3	0.1
2	0.2	0.8	0.4	0.3	0.1
3	0.3	0.2	0.8	0.3	0.1
4	0.1	0.3	0.4	0.8	0.1
5	0.2	0.2	0.1	0.2	0.8

5 Vytváření modelů neuronové sítě

Po úspěšném vytvoření vlastní datové sady bylo dalším krokem zvolení vhodného modelu neuronové sítě, která se na daných datech bude učit. Experimentoval jsem s vlastními i cizími architekturami, různými trénovacími přístupy i ztrátovými funkcemi. Mým cílem bylo vytvořit pět různých modelů, každý z nich natrénovaný na jiné verzi mé trénovací datové sady, aby si mezi nimi mohl uživatel cílové aplikace vybírat dle specifik šumu, se kterým se potýká.

Než si však ukážeme vývoj těchto modelů, je nejprve potřeba si uvést, jaké jsou požadavky na modely určené k využití ve VST pluginu. Dále si také ukážeme některé z mých neúspěšných pokusů o vývoj vhodné neuronové sítě, následně samozřejmě i úspěšné řešení.

5.1 Požadavky

Zásadními parametry pro modely neuronových sítí využitých ve VST pluginu jsou latence, paměťové a výpočetní zatížení a samotná kvalita redukce šumu.

VST pluginy jsou charakteristické tím, že musejí fungovat v reálném čase. Z toho tedy vyplývá, že latence daného modelu (tedy čas potřebný k vygenerování predikce daným modelem) bude zásadním parametrem. Vrchní hranice latence se však v tomto případě nedá snadno stanovit. V každém DAW, které bude výsledné VST využívat, si totiž uživatel může vybrat libovolnou velikost vstupních bufferů i vzorkovací frekvenci. Tyto parametry zásadně ovlivňují, jak často bude ve VST daná neuronová síť volána. Kromě toho také ovlivňuje horní hranici latence také samotný počítač, na kterém bude k daným výpočtům docházet.

Z toho důvodu jsem se rozhodl nejprve vycházet z předpokladu pevně dané velikosti vstupních bufferů, pro kterou jsem zvolil 1024 vzorků. Tato velikost je vhodná pro využití DAW v rámci editace, ačkoliv při nahrávání způsobuje příliš velkou latenci mezi přijetím nahrávaného signálu a jeho přehráním v rámci umělcova odposlechu. Vzhledem k tomu, že redukce šumů se obvykle provádí až ve fázi editace, to však nepovažuji za problém. Vzorkovací frekvenci jsem také uvažoval jako pevně danou, a to 44,1 kHz. Zvoleným referenčním strojem je notebook, na kterém tuto práci vyvíjím, tedy MacBook Air M1. Po krátkém experimentování se ukázalo, že horní hranicí pro plynulé přehrávání s těmito podmínkami je přibližně 20 ms.

Nezanedbatelná je i velikost výsledného modelu. Je důležité si uvědomit, že profesionální audioinženýr během své práce běžně pracuje například se sto a více stopami, přičemž na každé z nich může používat libovolné množství VST (obvykle však okolo

šesti). Každé VST na každé stopě tedy musí běžet současně. Ani v tomto případě neexistuje jasně stanovený maximální limit pro velikost výsledného VST pluginu, ale obecně platí, že čím méně paměti RAM zabírá, tím lépe.

Posledním důležitým kritériem je pak samozřejmě samotná kvalita redukce. Jako metriku pro porovnávání kvality redukce jsem uvažoval SI-SNRi. Modely uvedené v mé rešerši dosahují na datové sadě MUSDB18 SI-SNRi mezi 7,8 dB až 16 dB, proto bylo mým cíle, aby mé modely dosahovaly podobných výsledků, ideálně co nejblíže hodnotě 16 dB.

5.2 Vývoj modelů

Vývoj vhodné neuronové sítě byl zřejmě nejvíce časově náročnou fází této práce. Jak jsem již uváděl, cílem během této fáze bylo vytvořit několik modelů s různými vlastnostmi. Testoval jsem mnoho různých přístupů: vývoj vlastní architektury, modifikaci architektur pro separaci zdrojů, či Fine-Tuning modelů pro redukci šumů. Kromě pokusu o vývoj vlastní neuronové sítě jsem tak během této fáze testoval většinu architektur z mé rešerše, konkrétně tedy TasNet, Conv-TasNet, Demucs, RNNoise a Spiking-FullSubNet.

Některé architektury se ukázaly jako nevhodné pro má trénovací data, jiné způsobovaly příliš velkou latenci, další se mi i přes veškerou snahu nepodařilo natrénovat vůbec kvůli špatné kompatibilitě s mým operačním systémem macOS, u kterého je velice problematická práce s jazykem Python ve verzi 3.7, o kterou se některé z těchto architektur opíraly.

Než si tedy představíme funkční řešení, rád bych zde uvedl alespoň jeden z mých neúspěšných pokusů. Nakonec i neúspěch je nedílnou součástí každého vývoje. Zvláště pak u neuronových sítí, jejichž kvalita se předem těžko předpovídá. Jako nejzajímavější z nich považuji pokus o vývoj mé kompletně vlastní architektury.

Nakonec si v této kapitole uvedeme, jak vznikly úspěšné modely, které jsem ve výsledném VST pluginu skutečně použil.

5.2.1 Neúspěšný vývoj vlastní neuronové sítě

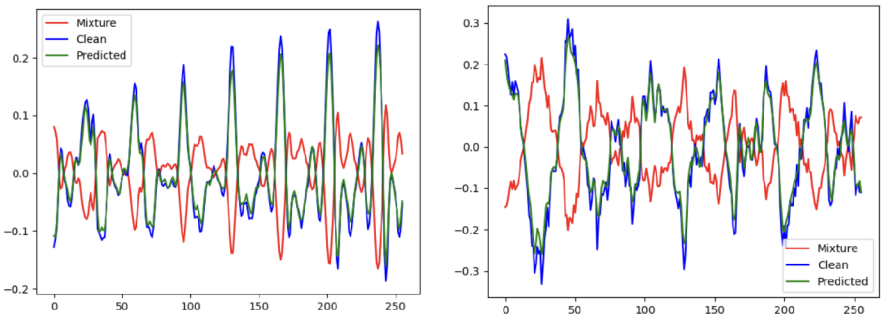
Zpočátku jsem si dal za cíl vytvořit pro tuto aplikaci vlastní neuronovou síť navrženou s ohledem na specifika cílového využití. Vzhledem k tomu, že se jedná o komplexní úlohu, jsem se rozhodl zaměřit se na redukci jediného typu šumu, konkrétně toho, se kterým jsem se z osobní zkušenosti setkával nejčastěji - přeslechy clicktracku ze sluchátek. Jedná se o pravidelný impluzní či tranzientní šum, který však může být hůře detekovatelný právě kvůli svému krátkodobému působení.

Na malém množství trénovacích dat jsem experimentoval s různými variantami modelů. V zásadních prvcích jsem se inspiroval u ozkoušených architektur z mé rešerše. Podobně jako ve většině uvedených modelů, i zde jsem zvolil typ enkodér-separátor-dekodér. Jak vyplývá z většiny studií na téma redukce šumu pomocí neuronových sítí, místo frekvenční reprezentace daných signálů se lépe osvědčilo pracovat

se signály pouze v časové doméně. Proto jsem pro enkodér a dekodér využíval různý počet za sebe řazených 1D konvolučních vrstev.

Zásadnější otázkou bylo provedení separátoru. Zatímco některé sítě nepoužívají separátory žádné a problematiku separace raději přenášejí na enkódovací a dekódovací vrstvy (které často využívají skokové spoje, neboli skip connections), jiné sítě zase mezi tyto enkódovací a dekódovací vrstvy vkládají různé více či méně komplexní systémy. V mém případě jsem pro separátor volil obvykle méně komplexní prvky, které měly zajistit rychlou predikci a malou velikost výsledné sítě. Ze všech těchto poměrně jednoduchých experimentů se mi nejvíce osvědčil model, který jako separátor využívá vrstvu LSTM. Ta totiž umožňuje uchovávat krátkodobou paměť o minulých stavech signálu.

Na následujících obrázcích můžete vidět, jak si jedna z podob této neuronová síť dokázala poradit s tranzietním ruchem v podobě clicku. Jsou zde vyobrazeny buffery o 250 vzorcích. Zvolené vzorkování bylo v tomto případě 44,1 kHz.



Obrázek 5.1: Potlačování clicku vlastním modelem NN

Pro odstraňování clicku tedy byl tento jednoduchý model dostačující, potlačování tranzientních a impulzních šumů v signálu tedy bylo spolehlivé. Jakmile ale byl tento model trénován na datové sadě s dalšími přidanými ruchy, bylo již z průběhu hodnot ztrátové funkce během trénovacího procesu jasné, že pro komplikovanější podoby šumy už tato architektura dostatečná nebude.

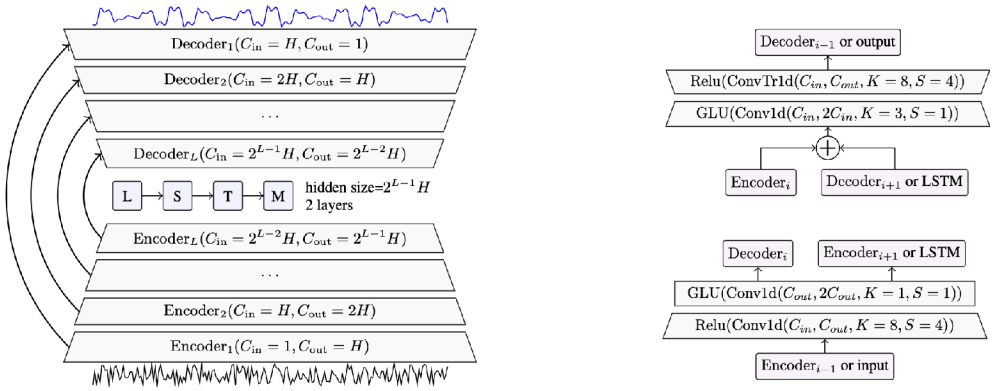
Z toho důvodu jsem vytvořil ještě několik dalších modelů, které svou architekturou nabíraly na komplexnosti a svou podobou se nejvíce blížily architektuře Conv-Tasnet, avšak výrazně lepších výsledků bohužel tyto modely nedosahovaly. Proto jsem se následně rozhodl, že rozumnějším postupem bude využít již dostupné architektury a přizpůsobit si je vlastním potřebám.

5.2.2 Úspěšné řešení - Fine-Tuning modelů architektury Demucs

Fine-Tuning neuronové sítě je proces, při kterém se použije síť, která je již předtrénována na velkém množství dat. Ta se dalším trénováním přizpůsobuje konkrétnímu úkolu či datové sadě. Tento proces zahrnuje úpravu vah sítě na základě nových tréninkových dat, přičemž některé vrstvy sítě mohou zůstat takzvaně zmražené, jinými

slovy neměnné, zatímco jiné se dále trénují. Cílem tohoto postupu je rychle dosáhnout vysoké přesnosti při použití menšího množství dat, což značně urychluje celý trénovací proces.

Pro metodu Fine-Tuning jsem experimentoval s několika různými architekturami, mimo jiné například s modely architektur Spiking-FullSubNet a RNNoise či upravenými modely architektur TasNet a Conv-TasNet. Nejlepších výsledků jsem však dosáhl na předtrénovaných sítích vycházejících z architektury Demucs. Tyto předtrénované modely jsou volně dostupné v knihovně `denoiser` od výzkumného týmu Facebook AI Research [28]. Jsou modifikované a natrénované pro potřeby redukce šumu v lidské řeči a každý z nich by měl být schopen funkce v reálném čase. Od klasického Demucs modelu se liší především tím, že nevyužívají frekvenční reprezentaci vstupního signálu, ale pouze časovou. Architektura těchto modelů tedy vypadá následovně:



(a) Celá architektura upravené architektury Demucs [28]

(b) Enkodér a dekodér upravené architektury Demucs [28]

Obrázek 5.2: Modifikovaná architektura Demucs [28]

Enkodérová vrstva E dostává na vstup signál zatížený šumem x vytváří její latentní reprezentaci $E(x) = z$. Každá vrstva se skládá z konvoluční vrstvy s velikostí jádra $K \in \mathbb{N}$ a krokem $S \in \mathbb{N}$ s $2^{i-1}H$ výstupními kanály, kde $H \in \mathbb{N}$ určuje počet skrytých kanálů a $i \in \mathbb{N}$ je indexem dané vrstvy. Za touto konvoluci následuje aktivační funcke ReLU s 1x1 konvolucí s 2^iH výstupními kanály a nakonec aktivační funkcí GLU, která zpětně převede počet kanálů na $2^{i-1}H$ [28].

Sekvenční modelovací síť R dostává jako vstup onu latentní reprezentaci z a vytváří z ní její nelineární transformaci stejné velikosti $R(z) = \text{LSTM}(z) + z$, označovanou jako $\hat{z}$. LSTM síť se skládá ze dvou vrstev a $2^{L-1}H$ skrytých jednotek, kde $L \in \mathbb{N}$ určuje počet těchto vrstev. Pro kauzální modely se používá jednosměrná LSTM, zatímco pro nekauzální modely se využívá obousměrná LSTM, následovaná lineární vrstvou pro sloučení obou výstupů [28].

Dekodérová síť D pak přijímá $\hat{z}$ jako vstup a vytváří odhad čistého signálu $D(\hat{z}) = \hat{y}$. Každá i-tá vrstva dekodéru přijímá $2^{i-1}H$ kanálů, aplikuje 1x1 konvoluci s 2^iH kanály. Následuje GLU aktivace, která vytvoří $2^{i-1}H$ kanálů a poté

transponovaná konvoluce s velikostí jádra 8, krokem 4 a $2^{i-2}H$ výstupními kanály doprovázená ReLU aktivační funkcí. Poslední vrstva má na výstupu pouze jeden kanál a ReLU aktivaci nepoužívá [28].

Velkou výhodou této architektury je mimo jiné také fakt, že díky využití 1D konvolucí nemá pevně danou velikost vstupu. Síť je tedy schopna zpracovat signály o libovolné délce. To je možné díky tomu, že 1D konvoluce provádí výpočty lokálně na malých úsecích signálu (daných velikostí kernelu), tedy nezávisle na jeho celkové délce. Dopředný průchod sítě tak pracuje postupně s jednotlivými částmi signálu a generuje výstupní tensor se stejnými rozměry jako měl tensor vstupní.

Knihovna `denoiser` poskytuje tuto modifikovanou architekturu Demucs pro real-time redukci šumů ve třech variantách. Liší se počtem skrytých vrstev (tedy výše uvedený hyperparametr H) a datovými sadami neboli využitými na trénování. Datová sada DNS obsahuje celkem 500 hodin nahrávek lidské řeči v několika jazycích s přidaným šumem se SNR mezi 20 dB a 5 dB, vzorkovací frekvencí 16 kHz a bitovou hloubkou 16 bitů [5]. Datová sada Valentini byla navržena pro metody enhancementu řeči operující se vzorkovací frekvencí 48 kHz [29]. Obsahuje celkem 824 zašumělých nahrávek. Parametr H pak určuje počet skrytých kanálů vstupní vrstvy [28], od které se odvíjí výsledná velikost celé sítě. Zde je tabulka všech dostupných variant.

Tabulka 5.1: Varianty předtrénovaných modelů knihovny *denoiser*

Název	**Parametr H**	**Datové sady**	**Velikost (MB)**
dns48	48	DNS	75,6
dns64	64	DNS	134,2
master64	64	DNS, Valentini	134,2

S ohledem na výše uvedené požadavky na model neuronové sítě pro VST plugin se mi nejvíce osvědčila síť `dns48`. Otestoval jsem si její latenci, jejíž průměrná hodnota na 100 testovaných tensorech o rozměru $1 \times 1 \times 1024$ byla 7,59 ms se zanedbatelnou odchylkou, což vyhovuje výše uvedeným předpokladům. Modely `dns64` a `master64` měly průměrnou latenci 13 ms. Po trénování na mé trénovací sadě a testování na sadě testovací dosahovaly tyto modely přibližně o 0,5 dB až 1 dB lepšího SI-SNRi. Jelikož však při testování na tensorech o rozměru $2 \times 1 \times 1024$, tedy pro stereo nahrávky, dosahovaly průměrné latence 26 ms, dále jsem je neuvažoval. Za těchto podmínek totiž překračovaly stanovený limit 20 ms.

Po výběru sítě a její lehké modifikaci bylo dalším krokem vytvořit několik modelů natrénovaných na mé datové sadě, tedy provést samotný Fine-Tuning. Tím se budeme zaobírat v následující podkapitole.

5.2.3 Trénování

Pro Fine-Tuning těchto modelů jsem vytvořil vlastní program v jazyce Python 3.9. Tento program si nejprve načte seznam cest k trénovacím nahrávkám a také samotný předtrénovaný model `dns48`. Poté probíhá trénování tohoto modelu ve třech vnořených smyčkách:

1. **Smyčka epoch**: uvnitř této smyčky se několikrát iteruje přes celou trénovací datovou sadu či přes její část

2. **Smyčka nahrávek**: uvnitř této smyčky se postupně načítá každá zašumělá nahrávka a odpovídající čistá nahrávka z trénovací datové sady či z její části

3. **Smyčka dávky bufferů**: uvnitř této smyčky se z obou nahrávek vybere dávka o určitém počtu bufferů o velikosti 1024, na dávce ze zašumělé nahrávky se dále provede dopředný průchod modelem a z jeho predikce (opět v podobě stejnorozměrné dávky) a dávky čistého signálu se spočítá ztráta, která je zpětně propagována modelem

Generování dávek bufferů z nahrávek obstarává má funkce `select_buffer`. Ta je volána se vstupní proměnnou `start_idx`, která určuje počáteční index, od kterého se z nahrávky vybere `batch_size` × `buffer_size` vzorků. Tento počáteční index je zvolen tak, aby se části signálu mezi trénovacími kroky během jedné epochy neopakovaly, tedy aby nedocházelo k překryvu mezi jednotlivými dávkami. Nejlépe se mi osvědčilo používat dávku 256ti bufferů velikosti 1024, tedy celkem 262 144 vzorků pro jeden trénovací krok. Zde je kód této dávkovací funkce:

```
def select_batch(audio, buffer_size, batch_size, start_idx):
    buffers = []
    for i in range(batch_size):
        idx = start_idx + i * buffer_size
        if idx + buffer_size > audio.size(-1):
            break
        buffer = audio[:, idx:idx + buffer_size]
        buffers.append(buffer)
    return torch.stack(buffers) if buffers else None
```

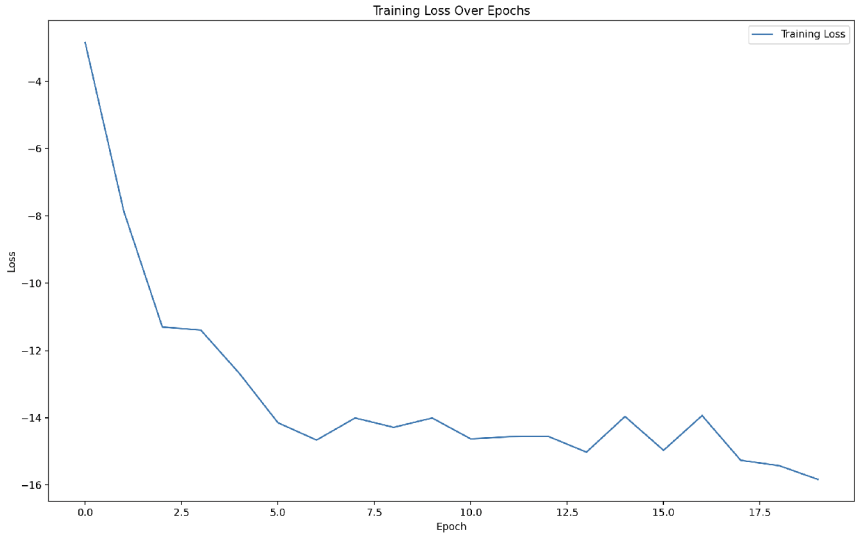
Tato funkce je tedy využívána na vytvoření dávek z čistých a zašumělých nahrávek. Ty jsou následně využity v trénovacím kroku uvnitř smyčky dávky bufferů pro samotné učení daného modelu. Jeden trénovací krok zde probíhá následovně:

```
optimizer.zero_grad()
predicted_batch = model(noisy_batch).to(device).float()
loss = my_sdr_loss(predicted_batch, clean_batch)
loss.backward()
optimizer.step()
```

Jedná se o základní smyčku zpětné propagace během trénování neuronové sítě. Prvním krokem je volání funkce `optimizer.zero_grad()`, která vynuluje gradienty všech parametrů modelu. Gradienty se během každé iterace sčítají, takže jejich nulování zajišťuje, že se aktualizace parametrů provádí pouze na základě aktuální dávky dat. Následně je vstupní zašumělá dávka `noisy_batch` předána modelu, což který provede dopředný průchod a vrátí tak dávku svých predikcí, které uloží do proměnné `predicted_batch`. Výsledná dávka je převedena na zařízení MPS (viz podkapitola GPU akcelerace pomocí MPS) a reprezentována ve formátu s plovoucí desetinnou čárkou, což je standardní typ pro numerické operace uvnitř této neuronové sítě. Funkce `my_sdr_loss` následně vypočítá ztrátovou funkci na základě rozdílu mezi

předpovězeným signálem a referenčním čistým signálem `clean_batch`, její implementace je dále rozebrána v podkapitole Ztrátová funkce SI-SNR. Následuje zpětná propagace chybového signálu skrze síť pomocí metody `loss.backward()`, která vypočítá gradienty parametrů modelu. Nakonec je volána metoda `optimizer.step()`, která aktualizuje parametry modelu na základě spočtených gradientů a optimalizačního algoritmu Adam, ten rozebírám v podkapitole Optimalizace Adam. Tento proces se následně opakuje pro každou dávku.

Na následujícím obrázku můžete vidět průběhů učení modelu `dns48_1117_018569` na druhé verzi mé trénovací datové sady během dvaceti epoch. Ztráta je zde uvedena v záporných číslech z optimalizačních důvodů, tedy aby při trénování docházelo k hledání minima této funkce místo jejího maxima. Vertikální osa (tedy Loss, neboli ztráta) má z povahy metriky SI-SNR jednotku dB, horizontální osa pak značí jednotlivé epochy.



Obrázek 5.3: Ukázka průběhu tréninku modelu

Během tohoto trénování ukládá můj program po každé epoše současné váhy neuronové sítě. Po všech dvaceti epochách se z těchto vah označí jako výsledné ty s nejmenší ztrátou. V tomto případě se tedy jednalo o poslední epochu se ztrátou -15,9 dB.

Optimalizace Adam

Pro optimalizaci jsem zvolil algoritmus Adaptive Moment Estimation, známý jako Adam. Oproti klasickému algoritmu Stochastic Gradient Descent (SGD), který používá pevnou rychlost učení, dokáže Adam dynamicky přizpůsobit rychlost učení jednotlivým parametrům. Výzkumy ukazují, že Adam je zvláště vhodný pro problémy, kde gradienty mají různou velikost nebo jsou řídké [30]. Při zpracování hudebních dat je běžné, že některé frekvence nebo prvky mají dominantní zastoupení, zatímco jiné jsou méně výrazné. Adam pomáhá zajistit, že optimalizace nebude přetížena

dominantními složkami, což by mohlo vést k horší generalizaci nebo pomalejšímu učení. Jako výchozí hodnotu pro rychlost učení (neboli learning rate) jsem zvolil 3 $\times 10^{-4}$, u několika modelů jsem však experimentoval i s hodnotami až do 1×10^{-3}.

Ztrátová funkce SI-SNR

Ve strojovém učení se k hodnocení výkonu modelu používá ztrátová funkce, která měří odchylku predikcí modelu od správných hodnot. Optimalizace modelu spočívá v úpravě jeho parametrů tak, aby minimalizovala hodnotu této funkce. Ztrátová funkce je specifickým typem cílové funkce, jejíž minimalizace nebo maximalizace reprezentuje cíl tréninku modelu. Pokud jsou predikce přesné, ztráta je malá, zatímco nepřesné predikce vedou ke ztrátě vysoké. Ztrátové funkce tak umožňují matematicky definovat cíl strojového učení, kterým je hledání minima (popřípadě maxima) této funkce [31].

V případě mé práce tedy ztrátová funkce měří rozdíl mezi nezašumělým signálem vokálů z datové sady MUSDB18 a predikcí čistého signálu, který daný model vygeneroval na základě zašumělého signálu z mé datové sady.

Jako ztrátovou funkci pro trénování jsem si zvolil SI-SNR, jelikož jsem s ní dosahoval lepších výsledků než při použití SDR a jelikož využití PESQ výrazně navyšovalo časovou náročnost procesu učení.

V jazyce Python jsem si vytvořil její vlastní implementaci s využitím knihovny `torch`. Zde je její kód:

```
def si_snr(pred, target, eps=1e-8):
    target_energy = torch.sum(target ** 2, dim=-1, keepdim=True)
    target_projection = torch.sum(pred * target, dim=-1, keepdim=
        True) * target / (target_energy + eps)
    noise = pred - target_projection
    si_snr_value = 10 * torch.log10((torch.sum(target_projection **
         2, dim=-1) + eps) / (torch.sum(noise ** 2, dim=-1) + eps))
    return -si_snr_value.mean()
```

GPU akcelerace pomocí MPS

Celý projekt vyvíjím na počítači MacBook Air M1. Z toho důvodu nebylo možné využít běžně používaný nástroj pro grafickou akceleraci zvaný CUDA.

Existuje ovšem verze PyTorch určená právě pro práci na zařízeních společnosti Apple, která využívá backend Metal Performance Shaders (MPS) pro akceleraci trénování na GPU (Graphics Processing Unit). Tento MPS backend rozšiřuje framework PyTorch a poskytuje skripty a možnosti pro nastavení a spouštění operací na počítači Mac. MPS využívá specifických vlastností GPU v čipech Apple (například M1 a M2) nebo AMD GPU, které jsou běžné v Mac počítačích. Poskytuje knihovny a shaderové funkce optimalizované pro výpočty potřebné v oblasti strojového učení, jako jsou operace s maticemi, konvoluce, aktivační funkce a další typické operace NN. Umožňuje tedy paralelizaci operací tak, aby se efektivně využily víceprocesorová jádra GPU [32].

MPS je tedy nástrojem, díky kterému je trénování neuronových sítí na macOS časově srovnatelné s jinými platformami využívajícími CUDA, ačkoliv v některých případech může být výkon stále nižší. Pro vývojáře je MPS důležitým nástrojem k tomu, aby mohli pracovat s hlubokým učením přímo na počítačích společnosti Apple bez nutnosti připojení ke vzdáleným serverům s výkonnými GPU.

Využití MPS při trénování je jednoduché. Nejprve je potřeba si definovat zařízení:

```
1   device = torch.device("mps" if torch.backends.mps.is_available
        () else "cpu")
```

Tento řádek kódu kontroluje, zda je k dispozici MPS backend pro PyTorch. Pokud dostupný je, nastaví MPS jako zařízení pro výpočty. Pokud není MPS k dispozici, výpočty probíhají na CPU.

Následně je možné přesunout libovolná `torch` data (model, tensor apod.) z CPU na GPU pomocí příkazu `.to(device)`. Tento krok zajistí, že všechny výpočty spojené s tímto tensorem budou probíhat na GPU (pokud je MPS zařízení dostupné), což může výrazně urychlit trénování a inferenci neuronových sítí.

Ve svém trénovacím procesu tímto způsobem přesouvám na GPU trénovaný model a dávky trénovacích dat, což urychluje trénovací proces až čtyřikrát.

5.2.4 Testování

Pro testování modelů jsem si opět vytvořil vlastní program v jazyce Python 3.9. Testování probíhalo na padesáti nahrávkách z mé testovací datové sady, které byly opět rozděleny do tensorů dávek o stejných rozměrech jako při trénování. Pro porovnání kvality redukce jsem zvolil metriku SI-SNRi, kromě toho jsem zde využil také vlastní implementaci SDRi. Zde je tabulka dosažených výsledků pěti nejúspěšnějších modelů pro dané dominantní typy šumů. Každý z nich byl tedy trénován a testován na odpovídající verzi datové sady.

Tabulka 5.2: Výsledky testů nejlepších modelů

Název modelu	**Verze sady**	**SI-SNRi (dB)**	**SDRi (dB)**
dns48_1009_012554	5	8,2	9,9
dns48_1114_020112	1	12,2	10,1
dns48_1030_013781	4	12,7	9,8
dns48_1115_059257	3	14,5	11
dns48_1117_018569	2	15,6	9,3

Jak lze na této tabulce pozorovat, soudě dle měřítka SI-SNRi si modely upravené architektury Demucs poradily nejlépe s redukcí dynamického šumu, nejhůře pak s rekonstrukcí lehce zkresleného signálu, což bylo vzhledem k nelineární povaze tohoto ruchu očekávatelné. Zároveň si také můžeme povšimnout, že lepší SI-SNRi nemusí vést k lepšímu SDRi. Ve výsledném VST pluginu si tedy uživatel bude moci mezi těmito modely libovolně vybírat dle specifik upravované nahrávky.

5.2.5 Export

Export těchto modelů bylo také potřeba přizpůsobit jejímu cílovému využití v rámci programu jazyku C++. Jak si blíže rozebereme v další kapitole, funkci této PyTorch neuronové sítě bude v pluginu obsluhovat knihovna TorchScript, která vyžaduje specifický formát modelu. Zde můžete vidět kód, pomocí kterého tuto neuronovou síť ukládám.

```
example_input = torch.randn(1, 1, 1024)
traced_script_module = torch.jit.trace(model, example_input)
traced_script_module.save(MODEL_PATH)
```

6 VST plugin

VST plugin je komponenta pro zpracování zvuku, která se používá v rámci hostitelské aplikace, nejčastěji nějakého DAW. DAW poskytuje VST pluginu bloky signálů či proudy událostí zvané event streams, které plugin následně zpracovává svým vlastním kódem. VST plugin tak dokáže například přijmout zvukový signál, provést na něm určitý proces, a poté vrátit výsledek zpět do hostitelské aplikace. K výpočtům obvykle využívá pouze CPU. Zvukový signál přichází v blocích neboli bufferech, které hostitel předává postupně za sebou. Velikost těchto bufferů je řízena hostitelem a jeho aktuálním nastavením. Plugin si udržuje stav všech svých parametrů potřebných pro probíhající proces [1].

Než si ukážeme specifika vývoje VST pluginu, je důležité se seznámit s nástroji, které jsem se rozhodl pro své VST využít.

6.1 Seznámení s využitými nástroji pro vývoj

Vývoj VST pluginů může probíhat mnoha různými způsoby a v různých programovacích jazycích. Já se rozhodl pro zřejmě nejrozšířenější přístup, kterým je vývoj v jazyce C++. Jazyk C++ je vhodný díky své vysoké výkonnosti, nízkoúrovňové kontrole nad hardwarem a pamětí, rozsáhlé podpoře zvukových knihoven, nebo také možnosti snadného přenosu mezi různými platformami.

Jako základní framework pro funkcionalitu VST pluginu jsem se rozhodl využít populární framework JUCE. Především díky tomu že je open-source, jeho podpoře, široké funkcionalitě, dobře zpracované dokumentaci a mnoha dostupným online materiálům pro seznámení se se základy používání tohoto frameworku.

Využití PyTorch neuronové sítě, která je nativně vytvořená v jazyce Python, uvnitř C++ programu umožňuje knihovna TorchScript, která zprostředkovává rozhraní pro překladač a interpret TorchScript JIT.

Pro společnou integraci frameworku JUCE a knihovny TorchScript jsem využil software CMake, dnes už prakticky standardní způsob jak definovat proces sestavení (tzv. build) komplexnějších C++ kódů.

V následujících podkapitolách krátce představím výše zmíněné nástroje.

6.1.1 JUCE framework

JUCE je cross-platform open-source C++ framework pro vývoj VST pluginů i jiných zvukových aplikací. První verze tohoto frameworku byla zveřejněna už v roce 2004,

nyní patří společnosti Anti-Piracy Inc [33].

JUCE na úrovni zvukových zařízení automaticky řeší rozdíly mezi operačními systémy (Windows, macOS, Linux, iOS a Android) a formáty VST pluginů (VST, VST3, AU, AUv3, AAX a LV2). Díky knihovně stavebních bloků pro digitální zpracování zvuku (neboli DSP) JUCE umožňuje rychle vytvářet prototypy a vydávat nativní aplikace a pluginy s konzistentním uživatelským prostředím na všech podporovaných platformách. Pomocí jednoho JUCE projektu je tak možné vyvíjet ze stejného zdrojového kódu všechny výše zmíněné formáty pluginů. Používání JUCE také chrání aplikace vyvíjené v tomto frameworku před možnými problémy s kompatibilitou, které by mohly být způsobeny budoucími aktualizacemi operačního systému a hostitelských modulů daných pluginů [34].

Lze ho také snadno integrovat do již existujícího build systému. Každý JUCE modul je distribuován jako zdrojový kód v jazyce C++14. Poskytuje také program Projucer, nástroj pro konfiguraci sestavení a kompilace JUCE projektů v nativních vývojových nástrojích, jako jsou Visual Studio, Xcode, Android Studio, Code::Blocks a Makefiles [34].

JUCE dále poskytuje univerzální UI (User Interface, uživatelské rozhraní) abstrakci, kterou lze spustit na libovolné platformě s možností hardwarové akcelerace prostřednictvím OpenGL. Zvládá také vykreslování 2D a 3D grafiky či poskytuje výběr obrazových formátů a písem. Všechny interaktivní grafické prvky uživatelského rozhraní, tzv. widgety, mohou být také nadefinované uživatelem [34].

Díky všem zmíněným funkcionalitám a dlouhodobé existenci a podpoře tohoto frameworku se v současnosti jedná o nejvyužívanější framework pro vývoj zvukových aplikací. Ve svých produktech ho využívají společnosti s hudebním zaměřením, jako například Fender, Korg, Pioneer DJ, Roland a Moog, ale také společnosti bez hudebního zaměření jako Netflix, Adobe, nebo Google. Díky tomu, že je open-source, je také často využíván nezávislými vývojáři.

6.1.2 TorchScript

TorchScript je rozšířením frameworku PyTorch, které umožňuje převést PyTorch modely do formátu, ve kterém se dají optimalizovat a serializovat. Serializace znamená proces převodu modelu do strukturovaného formátu, který lze uložit a později načíst pro další použití. Tento formát pak umožňuje využití modelů mimo standardní Python runtime (neboli „běhové prostředí", tedy specifické prostředí nebo software, který umožňuje spuštění a běh programu), například v produkčním prostředí nebo na zařízeních s omezenými zdroji. Využívá rozhraní API PyTorch. Toto rozhraní nabízí tři klíčové funkcionality:

1. Načítání a spouštění serializovaných modelů TorchScript, které byly původně definovány v jazyce Python [35].

2. Rozhraní API pro definici vlastních operátorů, které uživatelům umožňuje rozšířit standardní sadu operací jazyka TorchScript [35].

3. Just-in-time (JIT) kompilace programů v jazyce TorchScript přímo z jazyka C++ [35].

První funkcionalita je obzvláště užitečná pro případy, kdy jsou modely vyvinuty v jazyce Python, ale je třeba je exportovat do C++, což je případ této diplomové práce. API pak umožňuje vývojářům vytvářet a serializovat vlastní operátory pro inferenci v C++, zatímco funkce `torch::jit::compile` poskytuje přímý přístup ke kompilátoru jazyka TorchScript z C++ [35].

6.1.3 CMake

CMake je volně dostupný multiplatformní nástroj s otevřeným zdrojovým kódem určený pro automatizaci, sestavování, testování, balení a instalaci softwaru s využitím přístupu nezávislého na kompilátoru. Spíše než jako samotný systém pro sestavování projektů generuje CMake sestavovací soubory pro jiné systémy. Podporuje složité adresářové struktury a projekty, které jsou závislé na více knihovnách. Dokáže spolupracovat s nativními sestavovacími prostředími jako jsou Make, Ninja, Qt Creator, Android Studio, Apple Xcode a Microsoft Visual Studio. Jeho závislosti jsou minimální, potřebuje pouze kompilátor C++ v systému, na kterém je CMake sestaven [36].

Historie CMake sahá až do roku 1999. Při jeho vývoji bylo cílem vytvořit jednotný multiplatformní sestavovací systém, který by nahradil dvojí sestavovací prostředí pro Unix a Windows. Zavedl funkce jako je automatické skenování závislostí, podpora více cílů sestavení (spustitelné soubory, knihovny, zásuvné moduly) a introspekce systému pro zvýšení přenositelnosti. Byl navržen tak, aby minimalizoval vlastní závislosti, vyžaduje tedy pouze kompilátor C++. Navzdory počátečním problémům si CMake díky svému přístupu získal širokou oblibu při sestavování projektů v jazyce C či C++ [37]. Ačkoliv už z dnešního pohledu může působit lehce zastarale a méně intuitivně než podobné nástroje pro jiné jazyky než C++, jedná se stále o standardní nástroj pro vytváření komplexnějších C++ projektů.

Klíčovou komponentou CMake pro sestavování projektů je soubor s názvem `CMakeLists.txt`. Jak už přípona napovídá, jedná se o textový soubor, který v sobě nese všechny důležité informace o parametrech sestavení a také adresy všech využívaných zdrojových kódů daného projektu. Využívá na to vlastní imperativní skriptovací jazyk, který podporuje definování proměnných, metody pro manipulaci se stringy, pole, deklarace funkcí i maker a především začleňování modulů [38]. Pro export tohoto daného projektu do libovolného IDE podporující C++ se pak používá konzolový příkaz `cmake`.

6.2 Vývoj VST pluginu

Když už známe nástroje využité pro vývoj tohoto VST pluginu, můžeme si nyní přiblížit specifika tohoto vývoje. Od vzniku XCode projektu pomocí programu CMake až po sestavení výsledného VST pluginu.

6.2.1 Sestavení XCode projektu

Jak bylo uvedeno výše, tvorba souboru `CMakeLists.txt` je základem pro práci s programem CMake, který umožňuje propojit framework JUCE s knihovnou TorchScript a dalšími zdroji. JUCE nabízí pro práci s CMake vlastní API, které celé propojení usnadňuje a zpřehledňuje. Zde si uvedeme pár nejpodstatnějších řádků tohoto textového souboru a popíšeme si jeho funkci.

```
cmake_minimum_required(VERSION 3.24.1)
```

Tímto příkazem je stanovena minimální verze nástroje CMake, která zajišťuje kompatibilitu s nejnovějšími funkcemi a standardy C++. Verze 3.24.1 byla vybrána, protože podporuje moderní nástroje nezbytné pro tento projekt, včetně frameworku JUCE a knihovny PyTorch.

```
find_package(Torch REQUIRED)
```

Tento příkaz hledá a zajišťuje dostupnost knihovny PyTorch, která v kódu zpřístupňuje své rozšíření TorchScript.

```
add_library(SharedCode INTERFACE)
target_link_libraries("${PROJECT_NAME}" PRIVATE SharedCode "${
    TORCH_LIBRARIES}")
```

Knihovna `SharedCode` slouží ke sdílení zdrojových souborů mezi různými částmi projektu. Pomocí `target_link_libraries` jsou tyto zdrojové soubory propojeny s knihovnami PyTorch `TORCH_LIBRARIES`, což umožňuje efektivní práci s neuronovými sítěmi v reálném čase.

```
target_compile_definitions(SharedCode
    INTERFACE
    JUCE_WEB_BROWSER=0
    JUCE_USE_CURL=0
    JUCE_VST3_CAN_REPLACE_VST2=0
    CMAKE_BUILD_TYPE="${CMAKE_BUILD_TYPE}"
    VERSION="${CURRENT_VERSION}"
    PRODUCT_NAME_WITHOUT_VERSION="CleanWave")
```

Příkaz `target_compile_definitions` nastavuje globální definice pro kompilaci projektu. Zde jsou specifikovány například omezení použití některých komponent JUCE a verze kompilace. Tento krok optimalizuje výsledný kód a redukuje nepotřebné závislosti.

```
set(CMAKE_CXX_FLAGS
"${CMAKE_CXX_FLAGS} ${TORCH_CXX_FLAGS}")
```

Tímto příkazem jsou přidány specifické vlajky potřebné pro správnou kompilaci projektu, včetně optimalizací pro PyTorch. Tyto vlajky zajišťují, že JUCE a knihovny

PyTorch budou správně propojeny a projekt bude efektivně využívat dostupné systémové zdroje.

Po nachystání souboru `CMakeLists.txt` probíhá sestavení vývojového projektu pomocí následujícího příkazu v příkazovém řádku.

```
cmake -G Xcode -B Builds
```

Tento příkaz slouží k vytvoření projektu ve vývojovém prostředí Xcode. Parametr `-B Builds` určuje výstupní složku, kde CMake uloží vygenerované soubory projektu, v tomto případě složku s názvem Builds. Parametr `-G Xcode` pak definuje, že výsledný projekt bude určen pro Xcode – vývojové prostředí pro macOS.

6.2.2 Programování VST pluginu

JUCE poskytuje pomocí sestavovacího programu Projucer několik základních šablon pro různé typy JUCE projektů. Jednou z nich je i šablona pro vývoj VST pluginů, z té jsem také vycházel. Tato šablona obsahuje třídy PluginProcessor a PluginEditor. JUCE důkladně rozděluje definice daných proměnných a funkcí od jejich implementace, proto v každé z těchto tříd najdeme header soubor s příponou `.h`, ve kterém probíhají všechny definice, a také soubor s příponou `.cpp`, ve kterém se programuje veškerá funkcionalita.

U VST obecně platí konvence striktní separace uživatelského rozhraní od kódu který zpracovává zvukový signál. Této separaci odpovídají třídy PluginProcessor a PluginEditor. Jak už názvy napovídají, PluginProcessor se stará o plynulé zpracování zvuku, zatímco PluginEditor o funkci grafického rozhraní. U každého VST má vlákno pracující s audiem větší prioritu než ostatní vlákna.

Pro propojení funkcionality PluginEditoru a PluginProcessoru obvykle PluginEditor obsahuje vlastní instanci třídy PluginProcessor, což umožňuje zapisovat hodnoty z uživatelského rozhraní do vlákna pro zpracování zvuku.

Třída PluginProcessor

PluginProcessor je základní třídou každého JUCE VST programu. Tato třída zprostředkovává komunikaci se zvukovou kartou a probíhá v ní veškeré zpracování signálů.

Třída AudioProcessor slouží jako základní třída pro zvukové procesory nebo pluginy. Je navržena tak, aby byla dostatečně univerzální pro použití v různých formátech, jako jsou VST, AU, AAX a další, nebo pro interní použití. Kromě toho ji využívá také kód pro hostování pluginů, kde slouží jako obal pro instance načtených pluginů.

Nejpodstatnější funkcí této třídy je funkce `processBuffer`. Ta na svém vstupu obdrží buffer zvukového či MIDI signálu a následně provádí samotné zpracování. Pro plynulý průběh v reálném čase je naprosto zásadní, aby kód v této funkci byl co nejefektivnější. Je tedy potřeba se vyvarovat všem operacím, které zbytečně zvyšují výpočetní náročnost. Pokud je to možné, je ideální se vyhnout jakékoliv alokaci paměti. Všechny potřebné proměnné je tedy vhodnější definovat v header souboru této třídy. Dále je také dobré se vyhnout přílišnému kopírování dat, místo toho se

doporučuje používat ukazatele a reference. Nedoporučuje se ani používat mnoho vnořených smyček, které mohou výrazně navýšit výpočetní náročnost.

Třída PluginEditor

Třída PluginEditor na separátním vlákně obstarává funkcionalitu grafického rozhraní VST. V header souboru se tedy definují například veškeré slidery, tlačítka a další vizuální komponenty. V souboru s koncovkou `.cpp` třídy PluginEditor se pak implementuje jejich funkcionalit a umístění, dále také velikost samotného okna VST pluginu či jeho pozadí.

6.2.3 Sestavení VST

JUCE umožňuje sestavení VST pluginu ve formátech VST, VST3, AU, AUv3, AAX a LV2. Z hlediska programování nehraje výběr výsledného formátu příliš velkou roli, jelikož JUCE obstarává konverzi do těchto formátů bez nutnosti zásahu programátora. Tyto formáty se liší především svou kompatibilitou s různými DAW, podporovanými platformami a principy fungování. Já se rozhodl pro formát VST3.

Oproti ostatním formátům má VST3 řadu výhod. Je navržen pro moderní DAW a operační systémy, což znamená, že nabízí lepší stabilitu a dlouhodobou podporu než starší formáty. VST3 pluginy se také aktivují pouze když z DAW přijímají signál. Pokud žádný signál nepřichází, plugin se automaticky uspí, nevyužívá tedy systémové zdroje. To je výrazná výhoda oproti staršímu formátu VST nebo formátu AU, ve kterých pluginy běží neustále. Další významnou výhodou je také fakt, že DAW aplikace mohou s VST3 pluginy lépe komunikovat. Poskytují tak automatizace parametrů a zvukových tras. Tento formát umožňuje pluginům poskytnout DAW informace o všech svých funkcích, což vede k intuitivnější integraci a ovládání pluginu přímo z DAW [1].

7 Výsledný VST plugin CleanWave

Po několika měsících vývoje se mi podařilo přijít s funkčním a zároveň vkusným VST, které jsem pojmenoval jednoduše CleanWave, jelikož se jedná o nástroj čistící zvukové vlny od ruchů.

V následujících kapitolách si přiblížíme podobu a funkcionalitu jeho uživatelského rozhraní, následně také zásadní části kódu pro zpracování zvuku, které obstarávají funkci neuronové sítě a další efekty.

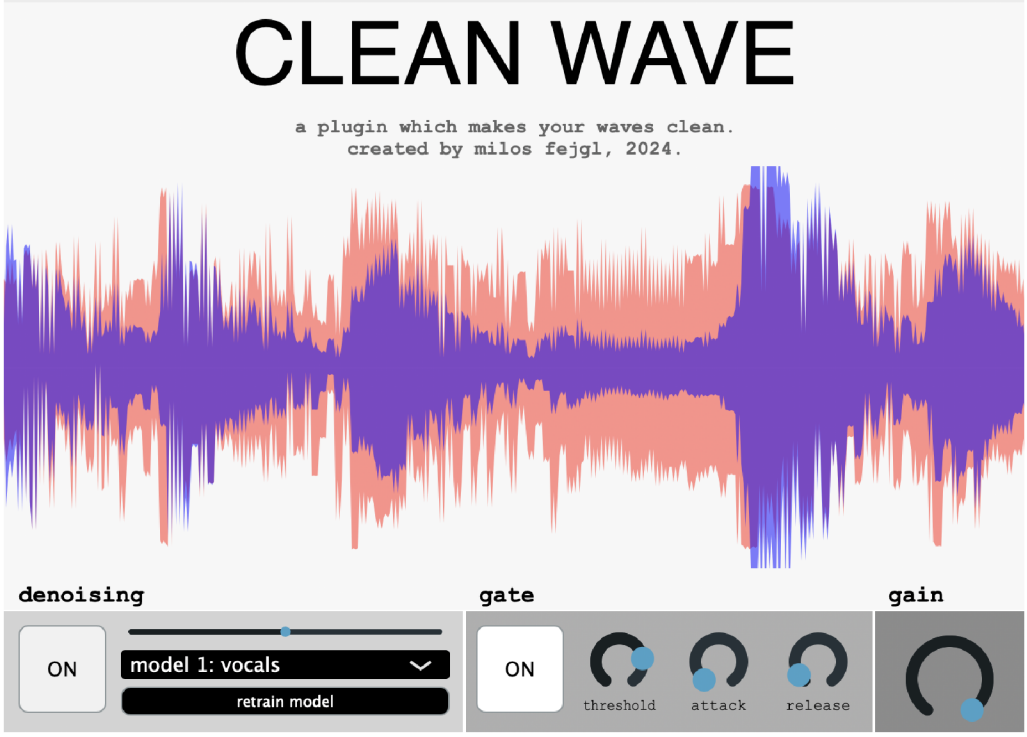
7.1 Uživatelské rozhraní

Uživatelské rozhraní VST pluginů je nezanedbatelně důležité. Trh VST pluginů je dnes poměrně silně saturovaný, proto pouhá funkcionalita nestačí. Pro kvalitní VST GUI (Graphical User Interface, grafické uživatelské rozhraní) je tedy potřeba zvážit následující parametry:

1. Přehlednost.
2. Estetika.
3. Hravost.

Každý z těchto parametrů je pro VST plugin důležitý. Audioinženýři si nejčastěji vybírají nástroje, jejichž fungování snadno a rychle pochopí a které na ně budou působit příjemně. Ideální VST plugin je pak také zábavné používat.

Po zvážení všech těchto faktů jsem po pár experimentech přišel s následujícím designem.



Obrázek 7.1: VST plugin CleanWave

7.1.1 Design, rozvržení a funkcionalita

Z hlediska designu jsem záměrně zvolil minimalistický přístup. Vycházel jsem ze znalostí, které jsem se naučil z knihy Simple and Usable: Web, Mobile, and Interaction Design od Gilese Colbornea [39]. Při návrhu designu jsem bral v potaz fakt, že se jedná o VST s ryze praktickým využitím. Přesto jsem ale usiloval o to, aby i tato veskrze rutinní a méně zábavná činnost působila v rámci možností hravě.

Největší část tohoto pluginu zabírá vizuální komponenta zobrazující právě přehrávaný signál. Červenou barvou je zobrazen vstupní signál který do pluginu přichází, zatímco modrou jeho výstupní signál. Obě tyto barvy mají nastavenou 50 % průhlednost, díky čemuž signály ve svých průnicích vytvářejí fialovou barvu. To uživateli umožňuje vizualizovat, jak plugin CleanWave ovlivňuje vstupní signál, podle čehož je následně možné vhodně nastavit další parametry, aby bylo například možné co nejlépe zachovat původní hlasitost. Touto vizualizací také lze kontrolovat, v jakých místech dochází k redukci šumu a v kterých ne, a pomocí dalších parametrů tak přidat či ubrat vliv neuronové sítě či zvolit síť jinou, nebo také vhodně přizpůsobit parametry funkce gate či gain, které si rozebereme dále.

Pod touto komponentou se nachází uživatelské menu pro obsluhu pluginu se třemi záložkami. Záložka `denoising` slouží k obsluze neuronové sítě. Nachází se zde tlačítko, které umožňuje daný model vypnout či zapnout. Dále se zde nachází posuvný slider s hodnotami v intervalu <0,1>, který ovlivňuje mix vstupního a výstupního signálu neuronové sítě. Pod tímto sliderem se nachází rozevírací seznam (neboli ComboBox), ve kterém si uživatel může vybrat, který model bude pro redukci ruchů

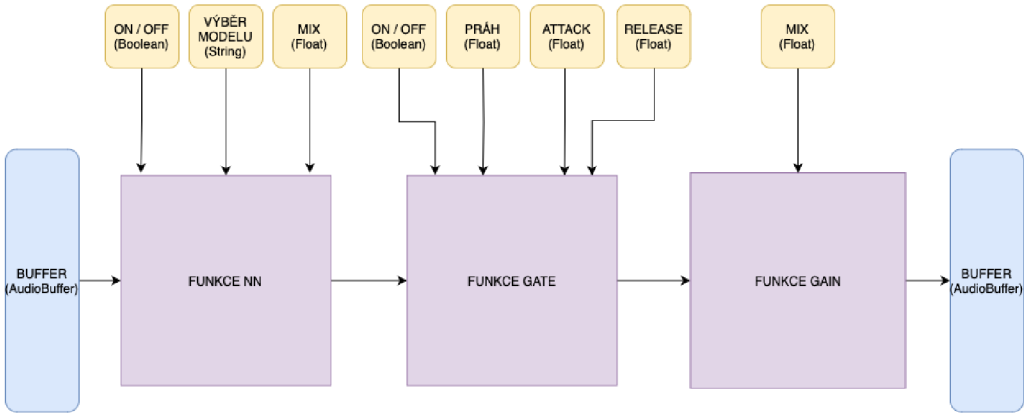
využívat. Měnit používaný model je možné i během přehrávání vstupní nahrávky, po dobu nahrávání nového modelu se redukce šumu na malou chvíli deaktivuje, po jeho nahrání se aktivita obnoví. Nakonec se zde nachází také tlačítko `retrain model`, které slouží k otevření menu pro nahrávání vlastních vzorků šumu a k následnému spuštění tréninku vybraného modelu na těchto datech. Tato funkcionalita však v tuto chvíli není plně funkční.

Záložka `gate` slouží k obsluze parametrů efektu šumové brány, která může být na daný signál aplikována. Je zde tlačítko pro její zapnutí a vypnutí, dále také parametry `threshold`, `attack` a `release`, jejichž funkci přiblížím v kapitolách zaobírajících se jejich implementací.

Poslední záložkou je `gain`, který uživateli pomocí otočného slideru umožňuje přizpůsobovat hlasitost výstupního signálu dle jeho potřeb. Tento slider používá logaritmickou stupnici, jeho minimální hodnotou je -60 dB a jeho maximální hodnotou je 6 dB. Nahrávku je tedy možné zeslabit tak, že je prakticky neslyšitelná, nebo naopak zesílit na dvojnásobnou hlasitost.

7.2 Zpracování zvuku

Zpracování zvuku kompletně obstarává třída PluginProcessor, kterou jsem si přizpůsobil svým potřebám, a to především vytvořením funkce pro práci s neuronovou sítí, funkce gate a také funkce gain. Každá z těchto funkcí provádí úpravy na právě přijatém bufferu, který je uložený v datovém typu AudioBuffer knihovny JUCE. Velikost tohoto bufferu není pevně daná a audioinžernýr si ji může libovolně nastavit ve svém DAW. Každá z výše uvedených funkcí má také další parametry, které její fungování ovlivňují. Diagram celého zpracování audio mého VST pluginu CleanWave můžete vidět na následujícím obrázku.



Obrázek 7.2: Diagram zpracování audia

Důvody využití funkcí gate a gain jsou prosté. Neuronová síť nemusí vždy odfiltrovat všechny drobné ruchy, které se v nahrávce vyskytují. Vhodně nastavená funkce gate může pročistit úseky signálu, kde nepůsobí žádný čistý signál, tím, že

daný vstupní signál utlumí. Neuronová síť i funkce gate zároveň mohou ovlivnit hlasitost výsledného signálu. Pro kompenzaci této změny hlasitosti je zde také funkce gain, která může výsledný signál dle potřeb zesílit nebo zeslabit.

Co se týká pořadí těchto funkcí, zabýval jsem se i otázkou, zda by nevedlo k lepším výsledkům umístění funkce gate před funkci neuronové sítě, což se na první pohled může zdát jako logičtější přístup. Toto pořadí se ale ukázalo jako méně intuitivní a v některých ohledech nepraktické. Jednak odvádí pozornost od toho, že neuronová síť je primární funkcionalitou tohoto VST. Je z uživatelského hlediska jednodušší na využití než gate, jelikož nepotřebuje žádné komplikovanější nastavování parametrů. Gate by tedy měl být pouze nástroj, který audioinženýr použije pouze pokud není s výsledky neuronové sítě spokojen. Neuronová síť je navíc trénována specificky na datech, které tyto ruchy obsahují i v úsecích, kde se nevyskytuje žádný čistý signál, je tedy sama o sobě schopna tyto úseky tlumit. Tento její aspekt pouze není stoprocentně zaručitelný.

V následujících podkapitolách si hlouběji rozeberme princip a realizaci jednotlivých funkcí.

7.2.1 Neuronová síť

Nejprve si ukážeme, jak probíhá načítání vybraného modelu pomocí knihovny TorchScript. Následně si ukážeme a popíšeme části kódu, které umožňují načtenému modelu generovat své predikce v C++ s ohledem na pokud možno co největší efektivitu celého procesu.

Načítání modelu

Model neuronové sítě se načítá při inicializaci třídy PluginProcessor. Toto načítání probíhá následujícím způsobem:

```
std::stringstream modelStream;
modelStream.write(  BinaryData::dns48_12_11_pt,
                    BinaryData::dns48_12_11_ptSize);
denoise_model = torch::jit::load(modelStream);
```

Tento kód zajišťuje načtení modelu do proměnné `denoise_model` pomocí knihovny TorchScript. Proces začíná vytvořením objektu `stringstream` s názvem `modelStream`, který slouží jako proud dat, kam jsou následně načtena binární data uložená v proměnných `dns48_12_11_pt` a `dns48_12_11_ptSize`. Tyto proměnné obsahují uložený natrénovaný model ve formátu TorchScript a informaci o jeho velikosti, což umožňuje efektivní distribuci a načítání neuronové sítě přímo z paměti programu. Metoda `modelStream.write()` pak zapíše tato binární data do datového toku. Jedná se o surová data, která nejsou nijak interpretována nebo strukturována.

Jakmile je model v datovém proudu načten, následuje volání `load(modelStream)`, které převede obsah proudu `modelStream` na instanci PyTorch modelu uloženou v proměnné `denoise_model`. Tento model je nyní připraven k dalšímu použití v aplikaci.

Obdobným způsobem pak probíhá i načítání modelu v případě, že si uživatel v GUI vybere jiný model.

Využití modelu

Dopředný průchod tímto modelem probíhá uvnitř cyklicky volané funkce `processBuffer`. Než však k samotnému predikování dojde, je nutným prvním krokem konverze vstupní proměnné datového typu AudioBuffer do datového typu akceptovatelného PyTorch modelem. Knihovna TorchScript umožňuje využít přímo datový typ Tensor z knihovny `torch`. Pro potřeby modelu se jedná o trojrozměrný tensor, který má při své inicializaci rozměry $2 \times 1 \times 1024$. První rozměr odpovídá počtu kanálů (standardně 2, tedy stereo), počet za sebou jdoucích bufferů v jednom kanálu a počet vzorků v jednom bufferu.

Velikost bufferů je však v každém DAW volitelná uživatelem, navíc je možné ji měnit i za pochodu VST. Proto jsem vytvořil funkci `prepareToPlay`, která se automaticky spouští vždy před voláním funkce `processBuffer`, a pokud to bude potřeba, přenastaví velikost všech datových struktur odpovídajícím způsobem. Zde můžete vidět její kód.

```
1  void PluginProcessor::prepareToPlay (double sampleRate, int
       samplesPerBlock)
2  {
3      juce::ignoreUnused (sampleRate);
4      numChannels = getNumInputChannels();
5      if (samplesPerBlock != lastNumSamples or samplesPerBlock !=
           lastNumChannels) {
6          unprocessedBuffer.setSize(getNumInputChannels(),
               samplesPerBlock, false, true, true);
7          processedBuffer.setSize(getNumInputChannels(),
               samplesPerBlock, false, true, true);
8          inputTensor = torch::zeros({1, 1, samplesPerBlock}, torch::
               kFloat);
9          inputTensorPtr = inputTensor.data_ptr<float>();
10         outputTensor = torch::zeros({1, 1, samplesPerBlock}, torch
               ::kFloat);
11     }
12     lastNumSamples = samplesPerBlock;
13     lastNumChannels = numChannels();
14 }
```

Proměnná `inputTensorPtr` a je ukazatelem na začátek pole obsahujícího hodnoty `torch` tenzoru `inputTensor` v paměti, kde jsou uloženy v datovém typu float. Proměnná `outputTensor` pak slouží jako výstup této sítě. Proměnné `unprocessedBuffer` a `processedBuffer` jsou typu AudioBuffer frameworku JUCE.

Jak jsem však uváděl dříve, upravená architektura sítě Demucs, kterou v této práci využívám pro redukci šumů, nemá pevně danou velikost vstupu. Vrací tedy predikce s rozměry které odpovídají vstupnímu bufferu. Proto nebylo nutné nijak přizpůsobovat kód pro predikci různým velikostem bufferů. Samotná predikování tedy probíhá uvnitř funkce `proccessBuffer` následujícím způsobem:

```
auto* channelData = buffer.getWritePointer(channel);
unprocessedInBuffer.copyFrom (channel, 0, channelData, buffer.
    getNumSamples());
if (nnOn) {
    std::memcpy(inputTensorPtr, channelData, buffer.getNumSamples()
        * sizeof(float));
    outputTensor = denoise_model.forward({inputTensor}).toTensor();
    processedBuffer.copyFrom (channel, 0, outputTensor.data_ptr<
        float>(), buffer.getNumSamples() );
}
```

Nejprve se do proměnné `channelData` uloží ukazatel na data bufferu vybraného kanálu, což umožňuje přímý přístup k zvukovým vzorkům pro čtení a zápis. Kopie tohoto bufferu se uloží také do proměnné `unprocessedInBuffer`. Tuto kopii využívám proto, aby uživatel mohl dle své potřeby kombinovat upravený a neupravený signál. Následující `if` podmínka kontroluje, zda v uživatelském rozhraní je či není funkce neuronové sítě spuštěna. Pokud spuštěna je, následuje první volání funkce `std::memcpy`, které kopíruje zvukové vzorky z bufferu (na které ukazuje `channelData`) do oblasti paměti tenzoru `inputTensor` pomocí ukazatele `inputTensorPtr`. Tento způsob umožňuje efektivní přenos dat z jednoho místa v paměti (zdrojového bufferu) na jiné místo (paměť tensoru) s minimální režií spojenou s přístupem k datům. Následně je na vstup právě načteného modelu `denoise_model` předán vstupní tenzor voláním metody `forward`, která zajišťuje dopředný průchod tímto modelem. Výstup je následně uložen do tenzoru `outputTensor`. Tento výstup obsahuje buffer zbavený predikovaného šumu. Druhé volání funkce `std::memcpy` zkopíruje data z výstupního tenzoru `outputTensor` do proměnné `processedBuffer`. Tento buffer je následně smíchán s původním bufferem, na kterém redukce provedena nebyla, tedy s `channelData`.

Ačkoliv se toto řešení může zdát na první pohled zbytečně komplikované, dosahuje nejmenší výpočetní náročnosti oproti všem jiným způsobům, které jsem zkoušel. Zejména kvůli specifickému způsobu přenášení dat z proměnné typu AudioBuffer do proměnné datového typu Tensor.

7.2.2 Gate

Gate je typ efektu pro dynamické zpracování zvuku, které se používá v hudební produkci a ve zvukovém inženýrství. Jeho hlavní funkcí je řízení hlasitosti zvukového signálu. Konkrétně funguje tak, že ztlumí signál, když jeho hlasitost klesne pod určitou hranici, čímž účinně odstraní nežádoucí šum [40]. Působení tohoto efektu se pak nazývá gating.

Funguje na základě nastavení určité prahové úrovně. Když hlasitost zvukového signálu klesne pod tuto prahovou hodnotu, brána se zavře a signál se ztlumí. Naopak, když hlasitost signálu překročí prahovou hodnotu, brána se otevře a signál projde [40].

Gate se dá implementovat mnoha různými způsoby a můžeme u něj uvažovat několik různých vstupních parametrů. Já jsem si pro potřeby tohoto pluginu vybral verzi, ve které si uživatel v UI může vybrat vlastní práh (neboli threshold), attack

a release. Význam těchto parametrů je následující:

1. **Threshold** - úroveň hlasitosti v decibelech, pokud vstupní signál tuto úroveň překročí je propuštěn nezměněný, pokud nepřekročí, je ztlumený.
2. **Attack** - čas v milisekundách, určuje, jak rychle se gate otevře poté, co úroveň signálu překročí prahovou hodnotu.
3. **Release** - čas v milisekundách, určuje, jak rychle se gate zavře poté, co úroveň signálu klesne pod prahovou hodnotu.

Níže je má realizace efektu gate. Využíval jsem pouze knihovnu JUCE pro převod vstupních dat do decibelů a funkci `getSampleRate` pro získání vzorkovací frekvence vstupního bufferu, dále pak exponencielu ze standardní C++ knihovny `std`. Tento kód je součástí `for`-smyčky, ve které se iteruje přes jednotlivé vzorky ve vstupním bufferu uložené v proměnné `channelData` s indexem `sample`.

```
if (gateOn) {
    inputLevel = juce::Decibels::gainToDecibels(channelData[sample
        ]);
    targetEnvelope = (inputLevel < gateThreshold) ? 0.0f : 1.0f;
    attackCoef = std::exp(-1.0f / (gateAttack * 0.001f *
        getSampleRate()));
    releaseCoef = std::exp(-1.0f / (gateRelease * 0.001f *
        getSampleRate()));
    coef = (targetEnvelope > gateEnvelope) ? attackCoef :
        releaseCoef;
    gateEnvelope = coef * gateEnvelope + (1.0f - coef) *
        targetEnvelope;
    channelData[sample] *= gateEnvelope;
}
```

Tato funkce se stará o dynamické řízení úrovně signálu. Když je gate aktivní, kód porovnává úroveň vstupního signálu s nastaveným prahem v proměnné `gateThreshold`. Pokud je vstupní úroveň nižší než práh, je výstupní obálka nastavena na nulu, signál je tedy ztlumen. Jinak je obálka nastavena na hodnotu jedna, což znamená, že signál prochází. Dynamické přepínání mezi stavy zajišťují koeficienty attack a release, které upravují obálku signálu plynule podle hodnoty proměnné `attackCoef` a `releasekCoef`, což zabraňuje skokovým změnám hlasitosti.

7.2.3 Gain

Gain udává rozdíl mezi vstupním a výstupním signálem v decibelech. Umožňuje tedy vstupní i výstupní signál libovolně zesilovat nebo zeslabovat. Tento jednoduchý efekt jsem implementoval pomocí nativní metody `.applyGain()` třídy AudioBuffer frameworku JUCE, která na daný buffer aplikuje gain, tedy vynásobí všechny prvky daného bufferu danou hodnotou. Tuto hodnotu si nastaví uživatel v grafickém rozhraní VST pluginu pomocí otočného knoflíku, kterému jsem implementoval logaritmickou škálu pro převod do decibelů.

8 Testování

V této kapitole si ukážeme, jakých výsledků mé řešení redukce šumů v hudebních nahrávkách s využitím neuronových sítí dosahuje.

Nejprve se budu zabírat testováním samotných modelů. Dříve už jsme si sice ukázali dosažené SI-SNRi a SDRi jednotlivých modelů, zde si však vyobrazíme, jak konkrétně na dané nahrávky působí, z čehož můžeme odvodit, jak svých výsledků dosahují a pro jaké případy mohou být jednotlivé modely vhodné.

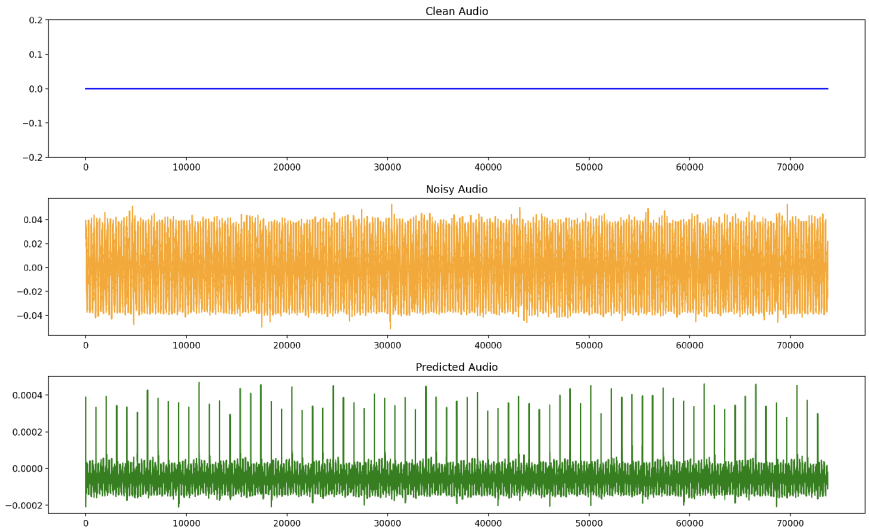
Následně se věnuji testování současné verze VST pluginu CleanWave. Nabízejí se totiž podstatné otázky: podařilo se výše uvedeným způsobem skutečně docílit real-time redukce ruchů? Funguje toto řešení plynule? A byl každý z využitých nástrojů pro tvorbu tohoto pluginu vhodnou volbou?

8.1 Testování modelů

Testování modelů probíhalo opět v jazyce Python 3.9 s využitím knihoven `torch` pro práci s modely, `torchaudio` pro práci se zvukovými signály, `matplotlib` pro zobrazování dat a `os` pro práci se soubory.

Na testovacích nahrávkách dosahuje můj měřitelně nejúspěšnější model `dns48_1117_018569` průměrného SI-SNRi 15,6 dB a SDRi 9,3 dB. Nyní by ale bylo vhodné si demonstrovat, co tato čísla znamenají v praxi. Ukážeme si, jaké výsledky tento model generuje na základě některých zašumělých nahrávek z mé testovací datové sady.

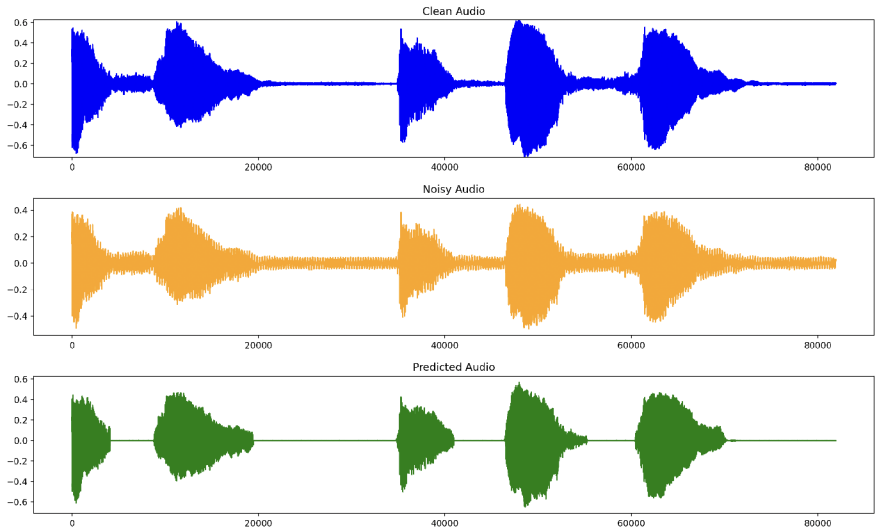
Na následujícím grafu můžeme vidět průběh úseku signálu o 75 000 vzorcích. Jedná se o úsek z první nahrávky druhé verze mé testovací sady. Je zde vyobrazen referenční čistý signál, zašumělý signál a signál predikovaný.



Obrázek 8.1: Graf redukce samotného šumu pomocí nejúspěšnějšího modelu

Zmíněná nahrávka na těchto prvních 75 000 vzorcích neobsahuje žádný čistý signál, žádný vokál, proto se referenční signál drží na nule a zašumělý signál obsahuje pouze samotný šum s magnitudou v rozmezí přibližně <-0,6 , 0,6>. Konkrétně se jedná o mix dynamického šumu způsobeného elektrikou, statického šumu, clicku a subbasového tónu o frekvenci 59 Hz. Signál predikovaný mým modelem také obsahuje šum, ale pouze v rozmezí <-0,004 , 0,006>. Z grafu je zřejmé, že signál predikovaný neuronovou sítí nemá nulovou střední hodnotu, což může být obecně u zvukových signálů problematické, avšak díky nízkým hodnotám magnitudy lze tento potenciální problém zanedbat. Lze tedy vidět, že samotný šum dokáže tento model potlačovat spolehlivě. Šum ve výše uvedeném rozmezí je na standardních studiových sluchátkách neslyšitelný.

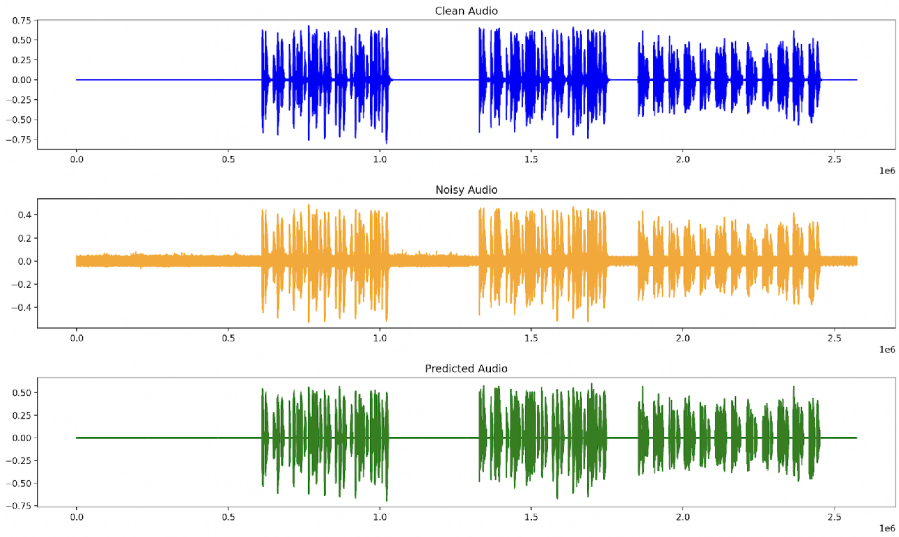
Další graf pak zobrazuje, jak si model `dns48_1117_018569` poradí se zašumělými nahrávkami vokálů.



Obrázek 8.2: Graf redukce šumu s vokály pomocí nejúspěšnějšího modelu

Lze pozorovat tendenci tohoto modelu potlačovat zašumělý signál na některých místech příliš radikálně. Stačí sledovat rozdíly mezi čistým a zašumělým signálem. V některých úsecích tento model generuje hodnoty blížící se nule, ačkoliv se v referenčním čistém signálu nacházejí hodnoty v rozmezí přibližně <-0,1 , 0,1>. Dalo by se tedy říci, že se tento model naučil provádět efekt zvaný gating, jehož definici jsem uváděl výše. Tento efekt je lidským uchem lehce znatelný.

Na třetím grafu pak můžeme vidět průběh celé první testovací nahrávky.

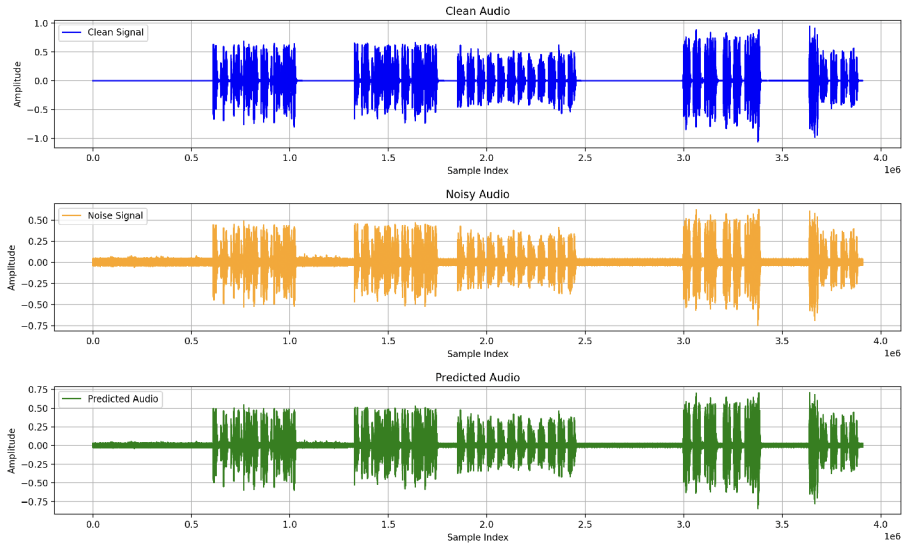


Obrázek 8.3: Graf redukce šumu z celé nahrávky pomocí nejúspěšnějšího modelu

Zde vidíme, že celkovou rekonstrukci čistého signálu zvládá tento model poměrně dobře. Jeho jediným problémem je výše uvedený gating. Ten je však v některých případech řešitelný, a to buď výše uvedeným nastavením parametru `mix` pro na-

stavení poměru vstupního a výstupního signálu funkce neuronové sítě. Toto řešení samozřejmě vede i k lehkému propouštění šumu, který se na vstupu vyskytuje, avšak vhodnou volbou tohoto parametru lze dosáhnout toho, že propouštěný šum je prakticky neslyšitelný, zatímco dříve ignorovaná část vokálů je dostatečně slyšitelná.

Dalším možným řešením je pak samozřejmě volba jiného modelu. Například modely `dns48_1114_020112` či `dns48_1030_013781` sice nedosahují tak vysokého SI-SNRi jako výše testovaný model `dns48_1117_018569`, propouští tedy o něco více šumu, zato u nich však takto radikálně působící gating neprobíhá. Zde je například ukázka predikce modelu `dns48_1030_013781` pro druhou nahrávku z čtvrté verze mé testovací datové sady.



Obrázek 8.4: Graf redukce šumu pomocí jiného modelu

Uživatel VST CleanWave si tedy může vybrat jeden z pěti předtrénovaných modelů dle vlastních požadavků a dle povahy samotného šumu či čistého signálů. Pokud uživateli nebude na vokálech vadit lehký gating a bude chtít především silnou a kvalitní redukci šumu, bude pro něj nejvhodnější využít model `dns48_1117_018569`. Pokud by gating efekt byl problematický, může uživatel použít kterýkoliv jiný model.

8.2 Testování pluginu CleanWave

Celkovou funkcionalitu výsledného VST pluginu CleanWave jsem testoval pomocí DAW jménem Reaper. Reaper je komplexní aplikace pro digitální produkci zvuku, která nabízí plnou sadu nástrojů pro vícestopé nahrávání, úpravu, zpracování, mix a mastering audio i MIDI signálů. Podporuje širokou škálu hardwaru, digitálních formátů a pluginů [41], včetně formátu VST3, který využívá plugin CleanWave. V IDE XCode jsem si nastavil automatické spuštění tohoto programu s každým úspěšným sestavením. V Reaperu jsem následně vytvořil novou zvukovou stopu,

do které jsem vložil jednu z testovacích nahrávek. Na té jsem následně aktivoval svůj VST plugin, čímž se otevřelo uživatelské rozhraní. Během tohoto testování jsem experimentoval s různými velikostmi vstupních bufferů, vzorkovací frekvenci jsem však považoval za fixní, a to 16 kHz. Celkově se tedy jednalo o testování v prostředí cílového využití. Prováděl jsem ho opět na procesoru M1 společnosti Apple, konkrétně na MacBook Air M1.

Toto testování bohužel nebylo tak úspěšné jako testování samotných neuronových sítí, potýkalo se totiž se zásadním problémem. Plugin CleanWave v této podobě i přes mnoho pokusů o vylepšení stále nedosahuje dostatečně nízké výpočetní náročnosti pro fungování v reálném čase. Důvodem je příliš velké zatížení CPU způsobené funkcí neuronové sítě. Ačkoliv implementace modelu v jazyce Python vyhovuje požadavku real-time využití, implementace v jazyce C++ bohužel negeneruje predikce dostatečně rychle. Při přehrávání právě upravené části nahrávky tím pádem často dochází k zásekům, playback tedy není plynulý. Tyto záseky jsou pozorovatelné pouze v případě, kdy je aktivována neuronová sít. Vizualizace signálů, efekty gate i gain fungují plynule. Funkce neuronové sítě funguje spolehlivě pouze v případě exportu výsledné stopy, na které je plugin CleanWave aktivován, což nevyžaduje problematické přehrávání v reálném čase.

Snažil jsem se tuto funkci co nejvíce optimalizovat, aby v ní docházelo k minimální alokaci paměti či aby nedocházelo ke zbytečnému kopírování dat. Ukázalo se však, že největší zpoždění způsobuje právě samotné predikování neuronové sítě. To totiž z nějakého důvodu funguje v jazyce C++ skrz implementaci výše uvedené knihovny TorchScript téměř třikrát pomaleji než v klasickém Python frameworku PyTorch, vzniká tedy příliš velká latence. Vzhledem k tomu, že PyTorch využívá ve svém backendu právě jazyk C++ (jako většina frameworků pro práci s neuronovými sítěmi), se jednalo o poměrně nečekané zjištění.

Jak se po několika hodinách procházení různých pomocných online fór ukázalo, nejsem zdaleka jediný, kdo se s tímto překvapivým zjištěním setkal. Někteří uživatelé dokonce hlásili, že jejich PyTorch modely implementované v jazyce C++ skrz knihovnu TorchScript fungovaly více než třikrát pomaleji než v jazyce Python [42]. Na žádném z těchto fór jsem bohužel nenašel žádné funkční řešení, které by nezahrnovalo zásadní zmenšení modelu neuronové sítě.

Při testování pluginu CleanWave se tedy bohužel ukázalo, že využití knihovny TorchScript bylo slepou cestou. V následujícím vývoji tedy bude nutné přijít s jiným řešením využití modelu PyTorch v kontextu C++ aplikace.

9 Závěr

Díky této diplomové práci se mi podařilo proniknout do problematiky redukce ruchů pomocí neuronových sítí. Seznámil jsem se s devíti nejvyužívanějšími modely a šest z nich jsem také otestoval.

Vytvořil jsem trénovací a testovací datové sady s celkem 9 hodinami a 48 minutami nahrávek zašumělých vokálů, a to pomocí extrahovaných vokálů z datové sady MUSDB18, volně dostupných nahrávek dynamického šumu a clicku a také vlastních programů v jazyce Python.

Následně jsem pomocí svých programů v jazyce Python a knihoven Denoiser a PyTorch vytvořil a natrénoval 5 úspěšných PyTorch modelů vycházejících z architektury Demucs, které by měly vyhovovat požadavkům využití v reálném čase, tedy které jsou dostatečně malé, rychlé a přesné. Nejúspěšnější z těchto modelů dosahuje na trénovací sadě SI-SNRi 15,6 dB, kvalitativně se tedy pohybuje v popředí současných architektur pro redukci šumů a enhancement řeči. Tento model dosahuje ve své Python implementaci průměrného latence pouze 7,2 ms, což se zdálo jako dostatečné.

Dále jsem také vytvořil VST plugin CleanWave, který umožňuje využít vytvořené modely v libovolném DAW a který zároveň poskytuje pomocné funkcionality jako parametrizovatelné efekty gate a gain, nebo také vizuální komponentu zobrazující čistý a upravený signál. Tento VST plugin jsem vytvořil pomocí programu pro správu C++ knihoven CMake, frameworku pro tvorbu zvukových programů JUCE a knihovny pro implementaci PyTorch modelu v C++ TorchScript.

Jak se bohužel příliš pozdě ukázalo, knihovna TorchScript pro takovéto využití nebyla správnou volbou. Oproti implementaci v jazyce Python totiž TorchScript model nečekaně generuje své predikce přibližně třikrát pomaleji, což je bohužel pro fungování v reálném čase nedostatečné.

Řešení v této diplomové práci je tedy spíše důkazem konceptu. Výše uvedené problémy nejsou neřešitelné a na jejich odstranění budu nadále pracovat. Knihovnu TochScript plánuji nahradit využitím frameworku ONNX, který umožňuje konvertovat libovolný model do `.onnx` formátu, ten je následně možné využít v jazyce C++ bez výrazného navýšení výpočetní náročnosti. Dále také plánuji natrénovat jednotlivé modely na nahrávkách se vzorkovací frekvencí 48 kHz, což je v hudební produkci standardem. Vzorkovací frekvenci 16 kHz jsem využil pouze pro ušetření času potřebného na trénování jednotlivých modelů. Stejně tak by bylo vhodné nadále experimentovat s větším počtem trénovacích epoch. Mám nachystané také separátní trénovací a testovací datové sady pro redukci šumů v nahrávkách bicích, kytar, bas a dalších nástrojů. Pro každý z nich natrénuji vlastní model, aby VST plugin Cle-

anWave umožňoval kvalitní redukci ruchů nejen pro vokály, ale i pro výše uvedené druhy hudebních signálů.

VST plugin CleanWave tedy ještě čeká další vývoj, tato diplomová práce dokumentuje jeho počátek. Podařilo se mi v něm využít mnoho vědomostí, kterých jsem během svého magisterského studia nabyl, a otestovat si je v praxi, konkrétně v průniku problematik programování a hudby.

Literatura

[1] STEINBERG. *Welcome to VST SDK 3.7*. Online. Steinbergmedia.github.io. Dostupné z: https://steinbergmedia.github.io/vst3_doc/vstsdk/index.html. [cit. 2025-01-06].

[2] NEURAL DSP. *DAWs (digital audio workstations)*. Online. Neuraldsp.com. Dostupné z: https://neuraldsp.com/getting-started/what-is-a-daw?g_keywordid=dsa-1456167871416&g_adtype=search&g_keyword=&g_campaign=%23GAds+-+Search+-+ROTW+-+DSA+-+2022-06&g_adgroupid=141271465870&g_campaignid=17432858297&g_network=g&g_acctid=332-563-6797&g_adid=602275143336&gad_source=1&gclid=Cj0KCQiAi_G5BhDXARIsAN5SX7pVonSxbxhRfBZ08nOmpCL05RFQTw3EuO3GIMDQpA7KX3FgjswWMVAaAro wcB. [cit. 2024-11-19].

[3] WAVES. *Clarity™ Vx*. Online. Waves.com. Dostupné z: https://www.waves.com/plugins/clarity-vx?w_campaign=19897215833&gad_source=1&gclid=Cj0KCQiAi_G5BhDXARIsAN5SX7pfb-R_BxMs7sXLjJJyOLNv18N4N8EDgAt61PY6r8R8fyyF_nrQYtQaAlM4EALw_wcB. [cit. 2024-11-19].

[4] SUPERTONE. *Introducing Supertone Clear: A Clear Standout in Voice Separation*. Online. Supertone.ai. Dostupné z: https://product.supertone.ai/clear. [cit. 2024-11-19].

[5] TIMCHECK, Jonathan; SHRESTHA, Sumit Bam; RUBIN, Daniel Ben Dayan; KUPRYJANOW, Adam; ORCHARD, Garrick et al. *The Intel neuromorphic DNS challenge*. Online. Neuromorphic Computing and Engineering. 2023, č. 3, s. 1-8. Dostupné z: https://iopscience.iop.org/article/10.1088/2634-4386/ace737/pdf. [cit. 2024-09-19].

[6] V. VASEGHI, Saeed. *Noise and Distortion*. Online. In: Advanced Digital Signal Processing and Noise Reduction. 2. Wiesbaden: Vieweg+Teubner Verlag, 2000. ISBN 0-470-84162-1. Dostupné z: https://dsp-book.narod.ru/295.pdf. [cit. 2024-11-20].

[7] HOSCH, William L. *Distortion*. Online. Britannica.com. 2006. Dostupné z: https://www.britannica.com/technology/distortion-communications. [cit. 2024-11-11].

[8] PRIYANKA, Poudel. *Evaluation Metrics for Speech(Audio) Signal Processing*. Online. Medium.com. 2023. Dostupné z: https://medium.com/@poudelnipriyanka/audio-metrics-their-importance-and-their-necessity-417950b0d848. [cit. 2024-11-08].

[9] LIGHTNING AI. *Perceptual Evaluation of Speech Quality (PESQ)*. Online. Lightning.ai. Dostupné z: https://lightning.ai/docs/torchmetrics/stable/audio/perceptual_evaluation_speech_quality.html. [cit. 2024-11-30].

[10] RIX, Antony W.; BEERENDS, John G.; HOLLIER, Michael P. a HEKSTRA, Andries P. *Perceptual Evaluation of Speech Quality (PESQ) – A New Method for Speech Quality Assessment of Telephone Networks and Codecs*. Online. Dostupné z: http://www.recursosvoip.com/docs/english/pap465.pdf. [cit. 2024-11-30].

[11] IBM. *What is a neural network?* Online. Ibm.com. Dostupné z: https://www.ibm.com/topics/neural-networks. [cit. 2024-11-19].

[12] LUO, Yi a MESGARANI, Nima. *TasNet: Time-Domain Audio Separation Network for Real-Time, Single-Channel Speech Separation*. Online. ICASSP. 2018. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8462116. [cit. 2024-10-22].

[13] LUO, Yi a MESGARANI, Nima. *Conv-TasNet: Surpassing Ideal Time-Frequency Magnitude Masking for Speech Separation*. Online. Arxiv.org. 2019. Dostupné z: https://arxiv.org/pdf/1809.07454. [cit. 2024-09-17].

[14] RONNEBERGER, Olaf; FISCHER, Philipp a BROX, Thomas. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. Online. Dostupné z: https://arxiv.org/pdf/1505.04597. [cit. 2024-12-26].

[15] STOLLER, Daniel; EWERT, Sebastian a DIXON, Simon. *Wave-U-Net: A Multi-Scale Neural Network for End-To-End Audio Source Separation*. Online. ISMIR Conference. 2018, roč. X, č. 19. Dostupné z: http://ismir2018.ircam.fr/doc/pdfs/205_Paper.pdf. [cit. 2024-11-16].

[16] DÉFOSSEZ, Alexandre. *Hybrid Spectrogram and Waveform Source Separation*. Online. Proceedings of the MDX Workshop. 2022. Dostupné z: https://arxiv.org/pdf/2111.03600. [cit. 2024-11-16].

[17] VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion et al. *Attention Is All You Need*. Online. Neural Information Processing Systems. 2023, č. 31. Dostupné z: https://arxiv.org/pdf/1706.03762. [cit. 2024-11-16].

[18] LU, Wei-Tsung; WANG, Ju-Chiang; KONG, Qiuqiang a HUNG, Yun-Ning. *Music Source Separation with Band-Slit RoPE Transformer*. Online. 2023. Dostupné z: https://arxiv.org/pdf/2309.02612. [cit. 2024-11-12].

[19] WANG, Ju-Chiang; LU, Wei-Tsung a WON, Minz. *Mel-Band Roformer for Music Source Separation*. Online. 2023. Dostupné z: https://arxiv.org/pdf/2310.01809. [cit. 2024-11-12].

[20] HENNEQUIN, Romain; KHLIF, Anis; VOITURET, Felix a MOUSSALLAM, Manuel. *Spleeter: a fast and efficient music source separation tool with pre-trained models*. Online. Journal of Open Source Software. 2020. Dostupné z: https://doi.org/10.21105/joss.02154. [cit. 2024-11-16].

[21] VALIN, Jean-Marc. *A Hybrid DSP/Deep Learning Approach to Real-Time Full-Band Speech Enhancement*. Online. 2018. Dostupné z: http://arxiv.org/pdf/1709.08243. [cit. 2024-09-19].

[22] YAMAZAKI, Kashu; VO-HO, Viet-Khoa; BULSARA, Darshan a LE, Ngan. *Spiking Neural Networks and Their Applications: A Review*. Online. Brain Sci. 2022, article 12, s. 863. Dostupné z: https://doi.org/10.3390/. [cit. 2024-09-22].

[23] HAO, Xiang; MA, Chenxiang; YANG, Qu; TAN, Kay Chen a WU, Jibin. *When Audio Denoising Meets Spiking Neural Network*. Online. IEEE. 2024, s. 1525-1528. Dostupné z: https://doi.org/10.1109/CAI59869.2024.00275. [cit. 2024-09-20].

[24] *MUSDB18*. SigSep [online]. [cit. 2024-02-23]. Dostupné z: https://sigsep.github.io/datasets/musdb.html.

[25] *Musdb*. GitHub [online]. [cit. 2024-02-23]. Dostupné z: https://github.com/sigsep/sigsep-mus-db

[26] textitFreesound. Online. Př. n. l. Dostupné z: https://freesound.org. [cit. 2024-09-18].

[27] STEWART, Ian. *What Is Audio Normalization?* Online. Izotope.com. 2023. Dostupné z: https://www.izotope.com/en/learn/audio-normalization.html. [cit. 2024-11-07].

[28] DE'FOSSEZ, Alexandre; SYNNAEVE, Gabriel a ADI, Yossi. *Real Time Speech Enhancement in the Waveform Domain*. Online. 2020. Dostupné z: https://arxiv.org/pdf/2006.12847. [cit. 2024-11-26].

[29] MAGDY, Muhammad. *Valentini Noisy Speech Database*. Online. Kaggle.com. 2023. Dostupné z: https://www.kaggle.com/datasets/muhmagdy/valentini-noisy/data. [cit. 2024-11-26].

[30] KINGMA, Diederik P. a BA, Jimmy Lei. *ADAM: A Method for Stochastic Optimization*. Online. ICLR. 2015. Dostupné z: https://arxiv.org/pdf/1412.6980. [cit. 2024-11-28].

[31] BERGMANN, Dave a STRYKER, Cole. *What is a loss function?* Online. Ibm.com. 2024. Dostupné z: https://www.ibm.com/think/topics/loss-function. [cit. 2024-11-12].

[32] APPLE. *Accelerated PyTorch training on Mac*. Online. Developer.apple.com. Dostupné z: https://developer.apple.com/metal/pytorch/. [cit. 2024-11-11].

[33] *JUCE Announces Acquisition by PACE*. Online. Web.archive.org. 2020. Dostupné z: https://web.archive.org/web/20200419092952/https://juce.com/discover/stories/juce-announces-acquisition-by-pace. [cit. 2024-09-17].

[34] *Build Audio Applications and Plug-ins*. Online. JUCE. Př. n. l. Dostupné z: https://juce.com/. [cit. 2024-09-17].

[35] *PyTorch C++ API*. Online. PYTORCH.ORG. Př. n. l. Dostupné z: https://pytorch.org/cppdocs/. [cit. 2024-09-17].

[36] *CMake: A Powerful Software Build System*. Online. Cmake.org. Př. n. l. Dostupné z: https://cmake.org/. [cit. 2024-09-17].

[37] BILL HOFFMAN, Bill a MARTIN, Kenneth. *The Architecture of Open Source Applications (Volume 1) CMake*. Online. Cmake.org. Př. n. l. Dostupné z: https://aosabook.org/en/v1/cmake.html. [cit. 2024-09-17].

[38] *Cmake-language(7)*. Online. Cmake.org. CMake. Př. n. l. Dostupné z: https://cmake.org/cmake/help/latest/manual/cmake-language.7.html. [cit. 2024-09-17].

[39] COLBORNE, Giles. *Simple and Usable: Web, Mobile, and Interaction Design*. New Riders, 2011. ISBN 0321703545.

[40] *What is a Noise Gate?* Online. Recordmixandmaster.com. Dostupné z: https://recordmixandmaster.com/2010-02-what-is-a-noise-gate. [cit. 2024-11-11].

[41] *This is REAPER*. Online. Reaper.fm. Dostupné z: https://www.reaper.fm/. [cit. 2024-11-12].

[42] FARRUKH, Waleed. *C++ forward 3x slower than python for traced model*. Online. Pytorch.org. 2019. Dostupné z: https://discuss.pytorch.org/t/c-forward-3x-slower-than-python-for-traced-model/52882. [cit. 2024-11-30].